Starfish: A high throughput BFT protocol on uncertified DAG with linear amortized communication complexity

Nikita Polyanskii
IOTA Foundation
nikitapolyansky@gmail.com

Sebastian Müller
Aix-Marseille Université
sebastian.muller@univ-amu.fr

Ilya Vorobyev
IOTA Foundation
ilia.vorobev@iota.org

Abstract

We present Starfish, the first provably live uncertified DAG-based Byzantine fault-tolerant (BFT) protocol. Existing uncertified DAG protocols such as Cordial Miners and Mysticeti provide low end-to-end latency but lack rigorous liveness proofs—recent work has demonstrated explicit counterexamples. We introduce a new pacemaker mechanism that establishes liveness for uncertified DAG-based protocols by ensuring validators synchronize after Global Stabilization Time. Beyond liveness, Starfish achieves optimal communication complexity: by decoupling transaction data from block metadata with Reed-Solomon coding and integrating data availability guarantees into the DAG construction, Starfish achieves linear $O(Mn)$ payload costs while increasing latency by only one round. Furthermore, using compression techniques such as threshold signatures and delayed dissemination, we design Starfish-C which achieves $O(\kappa n^3)$ metadata complexity in the worst case and $O(\kappa n^2)$ in the happy case—the first quadratic metadata bound for any full-DAG BFT protocol, tight up to the security parameter κ. Our empirical evaluation demonstrates that Starfish exhibits better performance and robustness under Byzantine attacks than existing uncertified DAG-based protocols. In addition, under good conditions, Starfish achieves higher peak throughput in large-scale scenarios than state-of-the-art BFT protocols.

1 Introduction

State Machine Replication (SMR) is a fundamental problem in distributed computing, requiring fault-tolerant services through state replication across multiple servers. Byzantine Fault-Tolerant (BFT) consensus protocols address this problem in the presence of adversarial behavior. The partial synchrony model [14] serves as a practical foundation for BFT protocols: it assumes that the network eventually stabilizes after an unknown Global Stabilization Time (GST), after which messages are delivered within a known bound Δ. In this setting, protocols can tolerate up to f Byzantine validators among $n = 3f + 1$ total validators—the optimal resilience for deterministic BFT consensus.

DAG-based BFT consensus has emerged as a promising approach to improve throughput by allowing parallel block proposals. Two primary classes exist: *certified DAGs* [1, 11, 33, 36, 37], which use reliable broadcast (RBC) to guarantee data availability and prevent equivocation, and *uncertified DAGs* [2, 23], which use best-effort broadcast to reduce latency. Certified DAGs achieve linear amortized communication complexity using efficient RBCs [12] but incur high end-to-end latency—11δ in Sailfish [33] using communication-efficient RBC, or 6δ with optimistic protocols that sacrifice linear complexity [34]. Uncertified DAGs achieve the *lowest latency* among DAG-based protocols (as low as 4.5δ in Mysticeti [2]), making them attractive for latency-sensitive applications.

Despite their practical adoption (e.g., Mysticeti in Sui [4] and IOTA [21]), uncertified DAGs have fundamen-

tal theoretical and practical gaps. First, existing protocols lack rigorous liveness proofs. Recent work [29] provides an explicit counterexample demonstrating a liveness attack on Mysticeti where honest validators become permanently desynchronized. The underlying vulnerability—validators advancing rounds without creating blocks—is inherent to uncertified DAG protocols; Cordial Miners [23], which originates the same round advancement mechanism, assumes synchronization after GST without proof. Second, the tight coupling of transaction data with consensus metadata results in quadratic $O(Mn^2)$ communication complexity for transaction data (where M is average payload per round), negating the efficiency gains of DAG parallelism. Third, the $O(\kappa n^2)$ and $O(\kappa n^4)$ consensus metadata storage and communication costs per round (where κ is the security parameter) leads to scalability issues: for 120 validators running the Mysticeti DAG at pace 20 rounds per second, the consensus database takes almost 1 TB within 24 hours and the bandwidth usage could reach 10 Gbps. These gaps motivate our work: can uncertified DAGs be made provably live while achieving the payload efficiency of certified DAGs, the low latency of uncertified DAGs, and reduced metadata complexity beyond what either class has achieved?

Our Contributions. We introduce *Starfish*[1], an uncertified DAG-based BFT protocol with the first rigorous liveness proof for its class, achieving the payload efficiency of certified DAGs, the low latency of uncertified DAGs, and reduced metadata complexity. Our contributions address each of the identified gaps:

- **New pacemaker for liveness of uncertified DAGs.** We address the liveness gap identified above with new pacemaker conditions (Section 4). The counterexample in [29] exploits validators advancing rounds without creating blocks, leaving "holes" that prevent leader commitment. Condition (A2) closes this gap: validators must create their own block before advancing rounds. Condition (C3) enables safe catch-up needed for synchronization after GST: upon receiving $2f + 1$ current-round blocks, a lagging validator can immediately create its block—if $2f + 1$ round-r blocks exist, at least $f + 1$ honest validators have advanced, making progress safe without waiting for timeout. Together with a proof of synchronization after GST (Appendix F), these constitute the first rigorous liveness analysis for uncertified DAG-based BFT protocols and provide a general fix applicable to Mysticeti and Cordial Miners.
- **Linear payload communication complexity.** While certified DAGs can achieve $O(Mn)$ transaction complexity via communication-efficient RBC such as [12, 8] at the cost of 2x increased latency, existing uncertified DAGs incur $O(Mn^2)$ due to tight coupling between transaction data and consensus metadata and cordial dissemination of blocks, which require pushing potentially missing ancestors to all peers. Starfish decouples transaction data from block headers: validators broadcast Reed-Solomon encoded shards rather than full transaction data, and transactions are sequenced only after forming Data Availability Certificates (DACs) directly on the DAG. Starfish integrates data availability proofs into the consensus protocol itself—block headers include a dedicated acknowledgment field referencing blocks whose transaction data is locally available, and $2f + 1$ such acknowledgments reachable from a committed leader form a DAC without additional network round-trips. This achieves optimal $O(Mn)$ communication cost per round while increasing the low latency of uncertified DAGs by only one round (Section 5).
- **Reduced metadata communication complexity.** All prior DAG protocols, including base Starfish, incur $O(\kappa n^4)$ metadata communication per round. Starfish-C introduces two compression techniques (Section 6):
  (1) *Threshold signatures* reduce non-leader block headers from $O(\kappa n)$ to $O(\kappa)$ by compressing both ancestors (non-leader blocks contain only a self-reference and an optional leader-reference with partial signature, while leader blocks retain full $O(n)$ references for DAG traversal) and acknowledgments (each validator sends partial signatures on data commitments back to block creators, who aggregate $2f + 1$ partial signatures into a single $O(\kappa)$-sized threshold signature serving as an explicit DAC). This reduces worst-case complexity to $O(\kappa n^3)$.

[1]Following the marine species naming tradition in DAG-based BFT protocols (Narwhal, Bullshark, Mysticeti, Shoal, Sailfish), we name our protocol Starfish. Like its namesake, Starfish survives damage—losing a limb (or transaction data) doesn't halt it. Some starfish regenerate entirely from one arm, akin to our protocol reconstructing transaction data from encoded shards.

Protocol	Broadcast primitive	Avg. e2e latency	One failure leader	Metadata costs (bits/round)		Payload costs
				happy case	worst case	worst case
Sailfish 1 [33]	RS-BRB [12]	11δ	$+(8\Delta + 2\delta)$	$O(\kappa n^4)$	$O(\kappa n^4)$	$O(Mn)$
Sailfish++ [34]	Opt BRB [34]	6δ	$+(4\Delta + 2\delta)$	$O(\kappa n^3 \log n)$	$O(\kappa n^3 \log n)$	$O(Mn^2)$
Angelfish [40]	BRB [11] + BEB	$\frac{6x-2}{x}\delta$	$+(4\Delta + 2\delta)$	$O(\kappa x n^2)$	$O(\kappa x n^3)$	$O(Mn^2)$
Mysticeti [2]	BEB	4.5δ	$+4\Delta$	$O(\kappa n^4)$	$O(\kappa n^4)$	$O(Mn^2)$
Starfish	**BEB**	5.5δ	$+2\Delta$	$O(\kappa n^4)$	$O(\kappa n^4)$	$O(Mn)$
Starfish-C	**BEB**	6.5δ	$+4\Delta$	$O(\kappa n^2)$	$O(\kappa n^3)$	$O(Mn)$

Table 1: Comparison of partially synchronous DAG-based BFT protocols. E2e latency is measured from when a transaction is received by a validator until it is sequenced by all honest validators; average is computed by averaging over all transactions. BRB denotes Byzantine reliable broadcast; RS-BRB denotes Reed-Solomon-based BRB [12, 8]; Opt BRB denotes optimistic BRB with dispersal and voting; BEB denotes best-effort broadcast. M is average payload per round, n is number of parties, κ is security parameter, x is the average number of blocks with transaction data per round in Angelfish (ranging from 1 in leader-only mode to n in full DAG mode; when $x = n$, latency becomes 6δ and happy-case metadata complexity becomes $O(\kappa n^3)$), δ is actual message delay, and Δ is known upper bound on δ.

(2) *Delayed history dissemination*: validators broadcast only their own blocks immediately and delay unknown history by 2Δ, achieving $O(\kappa n^2)$ when all validators are honest and synchronized. This bound is tight up to the security parameter: with n proposers per round, each block must reach a quorum of $\Omega(n)$ validators, yielding an $\Omega(n^2)$ information-theoretic lower bound.

- **Formal latency analysis.** Prior work provides latency bounds only in terms of rounds [23, 2] or RBC latency [33], making cross-protocol comparison difficult. We provide the first rigorous end-to-end latency analysis for uncertified DAG protocols in terms of actual message delay δ (Appendix F.3): Starfish achieves 7.5δ average latency with all honest validators, degrading gracefully to $O(n\delta + f\Delta)$ with f Byzantine validators. When a single Byzantine leader fails, Starfish adds only $+2\Delta$ delay compared to $+4\Delta$ or higher in other protocols.

Comparison with Related Work. Table 1 compares Starfish with state-of-the-art DAG-based BFT protocols under an idealized (oversimplified) synchronization assumption[2]. The comparison includes certified DAGs (Sailfish, Sailfish++, Angelfish) and best-in-class uncertified DAGs (Mysticeti). Starfish achieves linear payload complexity ($O(Mn)$) like certified DAGs using communication-efficient RBC [12], but with 50% lower latency (5.5δ vs 11δ for Sailfish). The "One failure leader" column shows Starfish's resilience: only $+2\Delta$ additional delay when a Byzantine validator fails during its leader turn, compared to $+4\Delta$ or higher in other DAG-based protocols. A detailed analysis appears in Appendix G.

A concurrent work, Angelfish [40], takes a different approach for the consensus metadata issue: it hybridizes leader-based and certified DAG-based protocols, dynamically switching between leader-only mode ($x = 1$) for optimal 4δ latency and full DAG mode (with $x = n$ proposer) for high throughput. However, Angelfish incurs $O(Mn^2)$ payload complexity even in leader mode. In contrast, Starfish offers two optimized regimes: the base Starfish achieves 5.5δ latency with $O(Mn)$ linear payload and faster Byzantine leader recovery ($+2\Delta$ vs $+4\Delta$); Starfish-C trades one additional round (6.5δ) for reduced metadata complexity ($O(\kappa n^2)$ happy case, $O(\kappa n^3)$ worst case)—the first quadratic metadata bound for any DAG protocol with n proposers, improving by a factor of n over Angelfish's $O(\kappa n^3)$—while maintaining linear payload..

[2]We assume perfect synchronization: all validators advance rounds simultaneously, each block references all blocks from the previous round, and messages are delivered within time δ. This assumption, also used in [33, 1], enables consistent comparison across protocols.

Paper Organization. Section 2 presents the system model and preliminaries. Section 3 defines the uncertified DAG model and commit rules common to protocols like Mysticeti and Cordial Miners. Section 4 introduces our pacemaker mechanism for liveness. Section 5 describes the Starfish protocol with Reed-Solomon encoded cordial dissemination and data availability certificates. Section 6 presents Starfish-C with threshold signature compression. Full proofs and performance evaluation appear in the appendix.

2 Preliminaries

System Model. We consider a message-passing system $\mathcal{V} := \{v_1, \ldots, v_n\}$ consisting of $n = 3f + 1$ validators or more generally parties. An adversary may control up to f validators referred to as *Byzantine* or *faulty*. These validators may deviate arbitrarily from the protocol. A validator that follows the protocol throughout the execution is considered *correct* or *honest*. A standard assumption is that the adversary is computationally bounded. This ensures that standard cryptographic properties, such as the security of hash functions and digital signatures, hold. A message x digitally signed by validator v using its private key is denoted as $\langle x \rangle_v$. The size of a hash evaluation is assumed to be κ, whereas a signature size is $O(\kappa)$.

We adopt the partial synchrony model of Dwork et al. [14]. In this model, the network initially operates in an asynchronous state, during which an adversary can arbitrarily delay messages sent by honest parties. However, after an unknown time called the *Global Stabilization Time* (GST), the adversary loses this ability, and all messages sent by honest parties are delivered to their intended recipients within a bounded delay Δ, which is known to the parties. Let δ be an (unknown) upper bound on the actual message delay and assume that $\delta \leq \Delta$ holds after GST. Furthermore, we assume that local clocks of parties exhibit *no clock drift* and are free from *arbitrary clock skew*.

Byzantine Reliable Broadcast and Atomic Broadcast. Byzantine Reliable Broadcast (RBC) [7] ensures that if an honest sender broadcasts a message, all honest parties eventually deliver it (validity), and no two honest parties deliver different values (agreement); see Appendix B for the formal definition. Certified DAGs use RBC to guarantee data availability and prevent equivocation. Byzantine Atomic Broadcast (BAB) extends RBC by ensuring a total order of message delivery across all honest parties:

Definition 1 (Byzantine Atomic Broadcast)**.** *Each honest validator $v_i \in \mathcal{V}$ can call $a\_bcast_i(m, r)$ and attempt to output $a\_deliver_i(r, v_k)$, where $v_k \in \mathcal{V}$. A Byzantine atomic broadcast protocol satisfies reliable broadcast properties (validity, totality, and agreement) as well as:*

- ***Total order.*** *If an honest party v_i outputs $a\_deliver_i(r, v_k)$ before $a\_deliver_i(r', v_\ell)$, then no honest party v_j outputs $a\_deliver_j(r', v_\ell)$ before $a\_deliver_j(r, v_k)$.*

Reed-Solomon Codes. For efficient payload communication, Starfish uses Reed-Solomon (RS) codes [31], which are maximal-distance separable codes [35]. An (n, k) RS code encodes k data symbols into n symbols. RS codes support two primary decoding modes: *erasure decoding* reconstructs from any k symbols when erasure positions are known, while *error-correcting decoding* reconstructs from any $k + 2e$ symbols, where up to e could be erronoues. Starfish uses systematic $(n, k) = (3f + 1, f + 1)$ codes: transaction data is divided into k *information shards* and encoded to produce $n - k = 2f$ *parity shards*. The encoding and decoding complexity is $O(M \log n)$ for messages of size M using FFT-based algorithms over binary extension fields [25]; see Appendix C.4 for details.

3 Uncertified DAG Model and Commit Rules

We present the model and commit rules common to uncertified DAG-based BFT protocols such as Cordial Miners [23] and Mysticeti [2]; Starfish is built on top of this model. Unlike certified DAGs, uncertified DAGs do not use reliable broadcast, so Byzantine validators may equivocate (create conflicting blocks). The commit rules must therefore ensure that all honest validators reach consistent decisions despite equivo-

cations.

Block format. Each block of validator v_i is signed and has the following format $B = (r, \texttt{ancestors}, \texttt{data})_{v_i}$, where r is the round number; the set `ancestors` contains n hash references[3] to the latest blocks created at or before round $r-1$ by distinct validators such that there are $2f+1$ blocks created at round $r-1$; the payload of the block is `data`.

Block DAG and rounds. Validators concurrently create blocks in rounds starting with round 1. The n blocks of round 0 are predefined and known to all validators. All blocks form a directed acyclic graph (DAG) or a block DAG, where each block serves as a unique vertex, and a directed edge from block B to block B' indicates that B explicitly references B'. Every validator creates a block, signs it, and multicasts it. The conditions triggering block creation and round advancement are discussed in Section 4, describing the Pacemaker mechanism.

Leader block. To get the total order over the blocks in the DAG, validators use the concept of leaders. Specifically, they agree beforehand on the round-robin leader scheduler: validator v_i proposes leader blocks in rounds $r = i, i+n, i+2n \ldots$ [4]. Leader blocks have the same structure as other blocks, but the difference is that validators wait for at least a timeout in round $r+1$ until they receive a leader block of round r.

Leader slot. A *leader slot* is a pair of a validator and round (v_i, r), where v_i is a leader in round r. Each slot is initially in the state UNDECIDED. By interpreting the DAG structure, each validator attempts to make a decision TO-COMMIT or TO-SKIP for every leader slot.

Patterns for leader slot. To make the decision, it is essential to find specific patterns that create certificates or skip the attempt to find certificates for leader blocks:

1. The *certificate pattern* for leader slot (v_i, r): blocks from at least $2f+1$ validators at round $r+1$ reference the same block B of round r created by the leader v_i. We then say that B is *certified*. Any block from round $r+2$ that contains in its history such a pattern is called a *quorum certificate* (QC) for the block B.
2. The *skip pattern* for leader slot (v_i, r): blocks from at least $2f+1$ validators at round $r+1$ *do not* reference a block of round r created by leader v_i. Note that there might be no proposal for the leader slot or more than one proposal in the case of equivocations. The slot is skipped by a validator if, for any block proposal, the validator observes $2f+1$ blocks in the next round that do not reference it.

Commit rule for leader slots. Validators first apply *direct decision rules* to make decisions for UNDECIDED leader slots.

1. *Direct commit rule:* The validator marks a leader slot as TO-COMMIT if the local DAG contains QCs from $2f+1$ distinct validators for that slot. The unique certified leader block from that slot is then called *committed*.
2. *Direct skip rule*: The direct decision rule marks a leader slot as TO-SKIP if the local DAG contains a skip pattern for that slot.

If the direct decision rule fails to mark a leader slot as either TO-COMMIT or TO-SKIP, the slot remains UNDECIDED and the validator resorts to the *indirect decision rule*. It initially searches for an *anchor*, which is defined as the leader slot with the smallest round number $r' \geq r+3$ that is marked as either UNDECIDED or TO-COMMIT. If the anchor is still marked as UNDECIDED, the validator leaves the slot as UNDECIDED.

[3] The hash reference of a block is the hash of the block, which includes the signature

[4] Cordial Miners assigns only one leader per three rounds, while Mysticeti supports even multiple leaders in every round. We stick with the option of one leader per round since we prove the latency improvement in this case, compared to Cordial Miners, while we have not observed latency improvements in both theory and practice in the case of the multileader option.

Decision	Rounds	Condition
Direct Commit	3	Exist $2f+1$ QCs at round $r+2$
Indirect Commit	≥ 6	Exists a path to a QC at round $r+2$ from the next committed leader in round at least $r+3$
Direct Skip	2	Exists a skip pattern at round $r+1$
Indirect Skip	≥ 6	No path to a QC at round $r+2$ from the next committed leader in round at least $r+3$

Table 2: Decision rules for a leader slot at round r

1. *Indirect commit rule*: if the anchor is marked as TO-COMMIT, the validator marks the leader slot as TO-COMMIT if the anchor contains a path to a QC for a block from the leader slot. The unique certified leader block from that slot is *committed*.
2. *Indirect skip rule*: if the anchor is marked as TO-COMMIT, the validator marks the leader slot as TO-SKIP if there is no path from the anchor to a QC for a block from the leader slot.

We summarize the rules in Table 2 and demonstrate examples of direct and indirect decision rules in Figures 6 and 7 in the appendix.

Sequence of committed leader blocks. After applying direct and indirect decisions to all leader slots, each validator derives an ordered sequence of committed leader blocks. Specifically, the validator iterates over that sequence of leader slots, gets certified leader blocks from slots marked as TO-COMMIT and skips all slots marked as TO-SKIP until it reaches the first slot marked as UNDECIDED.

Linearization of DAG. Let $\mathsf{hist}(B)$ denote the *causal history* of block B—the set of all blocks transitively reachable from B via ancestor references. Since the blocks are signed, one can derive a fixed topological ordering sort on blocks in the DAG. Given a sequence of committed leader blocks $C_1, \ldots, C_N$, one constructs a total ordering over all blocks in $\mathsf{hist}(C_N)$ as follows

$$\mathsf{sort}(\mathsf{hist}(C_1)), \mathsf{sort}(\mathsf{hist}(C_2) \setminus \mathsf{hist}(C_1)), \ldots, \mathsf{sort}\left(\mathsf{hist}(C_N) \setminus \bigcup_{i=1}^{N-1} \mathsf{hist}(C_i)\right). \tag{1}$$

Safety properties. The safety of uncertified DAG follows from the quorum intersection arguments: any two sets of $2f+1$ validators share at least one honest validator. We restate the key properties established in prior work [2, 23] here (for completeness, we provice the proofsin Appendix E):

Lemma 1 (Unique certified leader)**.** *At most one leader block from a given leader slot can be certified.*

Lemma 2 (Commit consistency)**.** *If a leader block is directly committed by any honest validator, then every honest validator will eventually commit the same block (either directly or indirectly).*

Lemma 3 (Skip consistency)**.** *If a leader slot is directly skipped by any honest validator, then no honest validator can commit a block from that slot.*

4 Pacemaker for Liveness of Uncertified DAGs

In partially synchronous BFT protocols, a *pacemaker* coordinates validators to ensure progress after GST. For uncertified DAG-based protocols, the pacemaker governs three events: (1) when validators advance rounds, (2) when they create blocks, and (3) when they disseminate consensus-related data (e.g., blocks) to peers.

Advance round $(r-1) \to r$	**Create block of round** r	**Broadcast history**
A1: received $2f+1$ blocks of round $r-1$ AND **A2:** created block of round $r-1$	**C1:** wait for leader and votes: **L1:** received leader block of round $r-1$ AND **L2:** received $2f+1$ votes for leader of round $r-2$ OR exists *skip pattern* for leader of round $r-2$ OR **C2:** timeout $\delta_{TO} = 2\Delta$ expired OR **C3:** received $2f+1$ blocks of round r	**B1:** created block OR **B2:** advanced round

Table 3: New pacemaker mechanism for uncertified DAGs. Our modifications are highlighted in blue.

Existing uncertified DAG protocols lack rigorous liveness proofs. Specifically, Mysticeti's Lemma 8 [2] and Cordial Miners' Proposition 38 [23] claim validator synchronization after GST but leave gaps in their arguments [33]. Recent work [29] provides an explicit counterexample for Mysticeti: an infinite execution trace in which no leader vertex is ever committed. The vulnerability arises because validators may jump rounds without creating blocks, and slow validators may never catch up—a pattern common to uncertified DAG protocols with deterministic leader scheduling. We address these gaps by introducing new pacemaker conditions (highlighted in blue in Table 3), enabling a formal proof that all honest validators synchronize within time Δ after GST. These conditions provide a general fix applicable to both Mysticeti and Cordial Miners.

We first recall the concept of the unknown history and cordial dissemination, which are essential for tracking block propagation. We then present our pacemaker conditions and state the key liveness properties they enable.

Validator's local view. Recall that $\mathsf{hist}(B)$ denotes the causal history of block B (Section 3). Every validator may have a different perception of the current DAG. Let us denote by DAG_i the local DAG of validator v_i. Every block expresses knowledge about the existence of other blocks in its causal history. Therefore, we can define the set of blocks known to validator v_j from the point of view of validator v_i as follows:

$$\mathsf{known}_i(j) = H_{i,j} \cup S_{i,j}, \tag{2}$$

where $H_{i,j} = \bigcup \mathsf{hist}(B)$ is the union over all blocks B in DAG_i created by validator v_j and $S_{i,j}$ is all blocks that are sent from v_i to v_j. A potentially unknown history of validator v_j from perspective of v_i is then defined as

$$\mathsf{unknown}_i(j) = \mathsf{DAG}_i \setminus \mathsf{known}_i(j). \tag{3}$$

Cordial dissemination. A received block is integrated in the local DAG only when its ancestors are already in the local DAG. Since Byzantine validators could equivocate or send their blocks exclusively to some validators, each validator has to ensure that every honest recipient of its newly proposed block B is aware of $\mathsf{hist}(B)$. For this purpose, validators follow the principle of *cordial dissemination* [32] which states: "Send to others blocks you know and think they need." Thereby, each validator tracks which blocks are known or already sent to each peer. When sending its block B, validator v_i also broadcasts to peer v_j a potentially unknown history of that block defined as the intersection $\mathsf{hist}(B) \cap \mathsf{unknown}_i(j)$.

Pacemaker conditions. The pacemaker conditions are summarized in Table 3, with our modifications compared to Mysticeti [2] and Cordial Miners [23] highlighted in blue. The protocol operates as follows:

Round advancement. A validator advances from round $r-1$ to round r when two conditions are met: it has received $2f+1$ blocks from round $r-1$ (**A1**), and it has created its own block in round $r-1$ (**A2**,

new). The latter prevents validators from "jumping" ahead without contributing blocks that serve as votes and certificates needed for leader commitment.

Block creation. After advancing to round r, a validator creates its block when one of three conditions is satisfied. The primary condition (**C1**) requires receiving the leader block from round $r-1$ (**L1**) and observing that the leader of round $r-2$ either received sufficient votes or was skipped (**L2**, *modified to add skip pattern*). This waiting ensures honest leaders get committed. If **C1** cannot be satisfied (e.g., due to a faulty leader), a timeout of $\delta_{TO} = 2\Delta$ (**C2**) allows the validator to proceed. The safe jump condition (**C3**, *new*) allows a validator to create its block upon receiving $2f+1$ blocks from round r, enabling faster synchronization when lagging behind. We also omit Cordial Miners' condition **L3** (waiting for $2f+1$ certificates) since it only improves round count, not actual latency.

Broadcasting history. A validator broadcasts the unknown history $\mathsf{unknown}_i(j)$, defined in (3), to peers whenever it creates a block (**B1**) or advances to a new round (**B2**, *new*). Broadcasting on round advancement is essential: it ensures that after GST, all honest validators advance rounds and create blocks within time Δ of each other. We show in Section 6 that broadcasting unknown history after a timeout can improve the communication costs in a happy case, but at the costs of increased latency for the case of Byzantine validators.

Liveness proof gaps. Mysticeti's Lemma 8 [2] claims that after GST all honest validators enter the same round within Δ, but the proof is incomplete [33]—an explicit counterexample demonstrates an infinite trace where no leader ever commits [29]. Cordial Miners' Proposition 38 [23] assumes without proof that all honest miners are in the same round after GST; the same round-skipping vulnerability applies since both protocols use deterministic leader scheduling. Our conditions **C3** and **B2** together fix these gaps for the entire class of uncertified DAG protocols: if a validator receives $2f+1$ round-r blocks but **C1** is unsatisfied, at least $f+1$ honest validators have already advanced past round r, so **C3** allows catching up without waiting for timeout.

Liveness properties. The new pacemaker conditions enable rigorous liveness proofs (full proofs in Appendix F). Let $r_{\max}$ denote the largest round among honest validators at GST. A key intermediate result is:

Lemma 4 (Synchronicity after GST)**.** *All honest validators enter any round $r > r_{\max}$ within time Δ of each other. Moreover, all honest validators create their round-r blocks within time Δ of each other.*

This property, which was assumed without proof in prior work [2, 23], yields the main liveness results:

Lemma 5 (Honest leaders committed)**.** *Any leader block created by an honest validator in round $r \geq r_{\max}$ will be marked* TO-COMMIT *by the direct decision rule.*

Lemma 6 (All leaders decided)**.** *After GST, any undecided leader block will eventually be decided, i.e., marked as* TO-COMMIT *or* TO-SKIP.

4.1 Latency Guarantees after GST

The above results establish liveness using the known network bound Δ. However, when the actual network delay δ is smaller than Δ, the pacemaker achieves tighter guarantees. Specifically, Lemma 4 holds with δ replacing Δ: all honest validators enter any round $r > r_{\max}$ within time δ of each other.

This δ-synchronization property enables precise latency analysis. The key insight is that Byzantine leaders can only add bounded delay:

Lemma 7 (DAG progress rate)**.** *After GST, the time between round r and round $r+1$ blocks created by honest validators is:*

1. *at most $\delta + 2\Delta$ when a leader of round r or round $r-1$ is Byzantine (and creates blocks).*

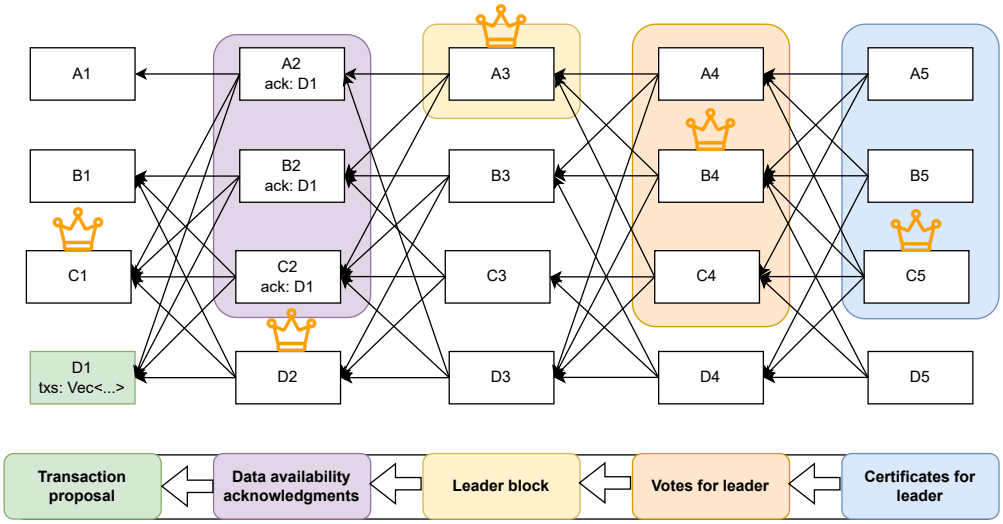

Figure 1: Commit rule in Starfish. Transactions in block D1 become ready for sequencing when $2f + 1$ acknowledgments (A2, B2, C2) are reachable from a committed leader block (A3), which serves as a Data Availability Certificate. Leader A3 is committed when $2f + 1$ certificates (A5, B5, C5) each observe $2f + 1$ votes (A4, B4, C4). Sequencing requires 5 rounds: proposal, acknowledgments, leader, votes, certificates.

2. *at most δ when the leader of round r is honest and the round-$(r-1)$ leader is honest or a Byzantine failure (that fails to create a block).*

This lemma shows that an active Byzantine leader adds at most 2Δ delay for up to two consecutive rounds (affecting conditions **C1** in rounds r and $r+1$), while a Byzantine failure with an honest successor adds only one 2Δ delay due to the skip pattern clause in **L2**. Combined with specific commit rules, this yields concrete latency bounds (see Appendix F.3 for detailed proofs).

5 Starfish Protocol

Starfish builds on the uncertified DAG framework described earlier, including the commit rules (Section 3) and the pacemaker (Section 4). Starfish introduces two key innovations:

1. *Decoupling transaction data from block headers*: Block headers contain only a commitment to the encoded transaction data, allowing leader block headers to be committed before transaction data is fully available. Transactions are sequenced using Data Availability Certificates (DACs) formed directly on the DAG.

2. *Encoded cordial dissemination*: Transaction data is encoded using Reed-Solomon codes, and validators share encoded shards (with proofs in case of erasure-correcting decoder) rather than full transaction data.

These reduce the amortized communication complexity for transaction data from $O(n^2)$ to $O(n)$, while increasing the latency to sequence transactions by only one additional round. The block header metadata complexity remains $O(\kappa n^4)$ per round; in Section 6, we show how threshold signatures can reduce this to $O(\kappa n^3)$. We refer to the compressed variant as *Starfish-C*. While Starfish-C reduces metadata complexity, it incurs a latency trade-off: the average block latency increases from 5δ to 6δ due to an additional round required for DAC aggregation via threshold signatures, and the timeout increases from 2Δ to 4Δ to ensure liveness (see Table 1). We refer to Appendix C for formal details.

Block format. Unlike prior uncertified DAG protocols (which use the format described in Section 3), Starfish decouples transaction data from block headers. The signed block header of validator v_i has the following format $B_{\text{header}} = (r, \texttt{ancestors}, \texttt{acks}, \texttt{dataCommit})_{v_i}$, where r is the round number; `ancestors` contains hash references to parent block headers; `acks` contains references to data commitments of blocks whose transaction data is locally available; and `dataCommit` is a commitment to the encoded transaction data (e.g., a Merkle root). Since each `dataCommit` is uniquely associated with a block header, acknowledgments implicitly reference the corresponding blocks. This separation allows leader block headers to be committed based on headers alone, while transaction data availability is tracked separately via acknowledgments. The full block includes the header and the transaction data: $B = (B_{\text{header}}, \texttt{data})$.

Reed-Solomon Coding and Encoded Dissemination. In Starfish, we employ Reed-Solomon (RS) coding with parameters $n = 3f + 1$ and $k = f + 1$. Specifically, the transaction data of each block is divided into $k = f + 1$ *information shards* and encoded to obtain $n - k = 2f$ *parity shards*. A commitment (e.g., a Merkle root) over all shards is included in the block header. Shards are verified against the commitment using *shard proofs*—in the case of erasure decoding, these are Merkle proofs; for error-correcting decoders, proofs may be omitted as the decoder itself allows for invalid shards. For $i \in [n]$, denote by $\mathsf{shard}_i(B)$ the ith encoded shard of block B (together with its proof if required).

When the broadcast history event is triggered (see Table 3), validator v_i sends to each peer v_j the following:

- New block created by v_i (in case of **B1**);
- All block headers unknown to v_j, i.e., the set $\mathsf{unknown}_i(j)$ defined for block headers similar to (3);
- For blocks B of other validators whose data became available since the last broadcast, send $\mathsf{shard}_i(B)$.

Data Availability Certificates. Since block headers are decoupled from transaction data, the data of a block may not yet be available to all validators when the header is received. To track availability, each block header includes an `acks` field containing references to data commitments of blocks whose transaction data the validator has locally verified. The transaction data of a block B becomes ready for sequencing when $2f + 1$ acknowledgments for B are *reachable* from a committed leader block—we call such a leader block a *Data Availability Certificate (DAC)* for B. Figure 1 illustrates the commit rule and data availability mechanism in Starfish.

Sequencing Transaction Data. For each block B, let $\mathsf{hist}_{\mathsf{DA}}(B)$ denote the set of all blocks D in $\mathsf{hist}(B)$ such that B is a DAC for the transaction data of D. Given the sequence of committed leader blocks $C_1, \ldots, C_N$ and the fixed topological ordering sort, we construct a total ordering over the sequenced transaction data analogously to (1), but restricted to blocks with data available to a quorum of validators:

$$\mathsf{sort}(\mathsf{hist}_{\mathsf{DA}}(C_1)), \mathsf{sort}(\mathsf{hist}_{\mathsf{DA}}(C_2) \setminus \mathsf{hist}_{\mathsf{DA}}(C_1)), \ldots, \mathsf{sort}(\mathsf{hist}_{\mathsf{DA}}(C_N) \setminus \bigcup_{i}^{N-1} \mathsf{hist}_{\mathsf{DA}}(C_i)). \tag{4}$$

Communication complexity of Starfish. Each validator broadcasts its own blocks and relays other validators' block headers exactly once per peer. Encoded shards are sent once per block. We analyze communication complexity of block header metadata and transaction data separately.

Block header metadata. Each block header has on average size $O(\kappa n)$ due to n hash references each in `ancestors` and `acks`. With n validators each broadcasting to $n-1$ peers, and each broadcast containing up to $n - 1$ other headers, the total communication per round is $O(\kappa n^4)$.

Transaction data. Assuming total transaction data averages M per round (regardless of distribution across validators), each block creator sends full data to $n - 1$ peers, while other validators send encoded shards of size $M/(n(f + 1))$ plus shard proofs of size P. The total cost is $O(Mn) + O(Pn^3)$, where $P = \kappa \log n$

for erasure decoding or $P = 0$ for error-correcting decoding. Since $\Theta(M)$ data is sequenced per round, error-correcting codes achieve linear amortized communication complexity for transaction data.

6 Compressing Block Header Communication in Starfish

Building on the Starfish protocol described in Section 5, we now address the communication overhead of block header metadata. While the compression techniques presented in this section can be applied to prior uncertified DAG protocols, we describe them in the context of Starfish, which additionally includes acknowledgments for tracking data availability. Recall that Section 5 introduced a block header format containing `ancestors` (hash references to parent block headers) and `acks` (references to data commitments of blocks whose data is available), each with on average $\Theta(n)$ entries. This results in block headers of size $O(\kappa n)$, contributing $O(\kappa n^4)$ to the communication complexity per round. In this section, we demonstrate two compression techniques that reduce this overhead:

1. *Threshold signatures*: By restructuring the block header format and introducing threshold signatures, we can reduce the size of non-leader block headers from $O(\kappa n)$ to $O(\kappa)$, while leader blocks retain $O(\kappa n)$ size to enable DAG traversal. This reduces the communication complexity for block headers to $O(\kappa n^3)$ per round in the worst case.
2. *Delayed unknown history dissemination*: By delaying the broadcast of unknown history by 2Δ (while sending own blocks immediately), we can achieve $O(\kappa n^2)$ communication complexity per round for block headers in a happy case of all honest validators.

Threshold Signature Preliminaries. We employ a $(2f + 1)$-of-n threshold signature scheme, where any subset of $2f + 1$ validators can collaboratively produce a valid aggregate signature. Each validator holds a share of the signing key and can produce a partial signature. Given $2f + 1$ partial signatures on the same message, these can be aggregated into a single threshold signature of size $O(\kappa)$. Such schemes are standard in the literature [5, 6] and can be instantiated using BLS signatures.

We introduce the following notation: let $\langle m \rangle_{v_i}$ denote validator v_i's partial signature on message m together with m itself, and let $\langle m \rangle_{\text{agg}}$ denote the aggregated threshold signature on m formed from $2f + 1$ partial signatures. In the compressed version of Starfish, we use three types of partial signatures:

- $\langle r \rangle_{v_i}$: partial signature on round r, aggregated into $\langle r \rangle_{\text{agg}}$ as round quorum proof;
- $\langle \texttt{leader}_{r-1} \rangle_{v_i}$: partial signature on round-$(r-1)$ leader reference, aggregated into $\langle \texttt{leader}_{r-1} \rangle_{\text{agg}}$ as leader certificate;
- $\langle \texttt{dataCommit}_B \rangle_{v_j}$: partial signature on data commitment of block B, aggregated into $\langle \texttt{dataCommit}_B \rangle_{\text{agg}}$ as data availability proof.

Compressing Ancestors. In the standard Starfish block format (Section 5), the `ancestors` field contains hash references to all observed block headers from the previous round—up to n references of size $O(\kappa)$ each, totaling $O(\kappa n)$ per block. The key observation is that non-leader blocks do not require explicit ancestor references for DAG traversal, since the leader blocks are sufficient for that purpose. We replace the `ancestors` field in non-leader blocks with a compact `ancestors` structure containing:

- `self`: a hash reference to the validator's own previous block header, size $O(\kappa)$;
- $\langle \texttt{leader}_{r-1} \rangle_{v_i}$: reference to round-$(r-1)$ leader with partial signature, size $O(\kappa)$; optionally included if the leader block header is available (i.e., block created via condition **C1** rather than **C2** or **C3**).

Each component has size $O(\kappa)$, reducing the non-leader block header from $O(\kappa n)$ to $O(\kappa)$. Only leader

blocks retain the full `ancestors` array to enable complete DAG traversal required for transaction sequencing.

Quorum Proofs. The compressed block header format for round r uses a `quorums` field containing two types of aggregated threshold signatures:

Round quorum $\langle r-1 \rangle_{\text{agg}}$: Each block of round r includes a partial signature $\langle r-1 \rangle_{v_i}$ on the round number. When a validator collects $2f+1$ blocks from round $r-1$, it aggregates their partial signatures into $\langle r-1 \rangle_{\text{agg}}$, proving it observed a quorum before advancing to round r. This round quorum is required in every block.

Leader quorum $\langle \texttt{leader}_{r-2} \rangle_{\text{agg}}$: In the standard Starfish protocol, leader certificates are implicit—a leader block is certified by a block B when B observes $2f+1$ blocks in the previous round, each referencing the leader block in their `ancestors` field. With compressed ancestors, this is no longer possible. Instead, each block at round $r-1$ optionally includes $\langle \texttt{leader}_{r-2} \rangle_{v_i}$—a reference to the round-$(r-2)$ leader with a partial signature. Each validator at round r attempts to aggregate $2f+1$ such partial signatures from round-$(r-1)$ blocks into $\langle \texttt{leader}_{r-2} \rangle_{\text{agg}}$, forming an explicit leader certificate. This leader quorum is optional and included when a block is created via **C1** rather than **C2** or **C3**.

Compressing Acknowledgments. In the standard Starfish protocol (Section 5), each block header contains an `acks` field with references to data commitments of other validators' blocks whose transaction data has been locally verified—on average n references requiring $O(\kappa n)$ per header. The key insight is to reverse the direction of acknowledgments: instead of "I acknowledge others," we use "others acknowledged me."

When validator v_j verifies the transaction data of block B (created by v_i), it sends a partial signature $\langle \texttt{dataCommit}_B \rangle_{v_j}$ on B's data commitment back to v_i. Once v_i collects $2f+1$ such partial signatures, it aggregates them into a threshold signature $\langle \texttt{dataCommit}_B \rangle_{\text{agg}}$ and includes it in its next block header. This aggregated signature serves as an explicit Data Availability Certificate for block B, proving that $2f+1$ validators verified its transaction data. The `acks` field thus becomes a vector of $\langle \texttt{dataCommit} \rangle_{\text{agg}}$ threshold signatures for v_i's pending blocks, each of size $O(\kappa)$, replacing the $O(\kappa n)$ acknowledgments field.

Compressed Block Format. Combining all compression techniques, we restructure the block header format. Non-leader blocks achieve $O(\kappa)$ size, while leader blocks retain $O(\kappa n)$ to enable DAG traversal.

Non-leader block headers. A non-leader block header at round r by validator v_i has the format:

$$B_{\text{header}} = (\langle r \rangle_{v_i}, \texttt{ancestors}, \texttt{quorums}, \texttt{acks}, \texttt{dataCommit})_{v_i},$$

where $\texttt{ancestors} = \{\texttt{self}, \langle \texttt{leader}_{r-1} \rangle_{v_i}?\}$, $\texttt{quorums} = \{\langle r-1 \rangle_{\text{agg}}, \langle \texttt{leader}_{r-2} \rangle_{\text{agg}}?\}$ (where ? denotes optional fields), and `acks` is a vector of $\langle \texttt{dataCommit} \rangle_{\text{agg}}$ proofs for own transaction data. Each field has size $O(\kappa)$ on average, giving total header size $O(\kappa)$ on average.

Leader block headers. The leader block header at round r requires explicit ancestor references for DAG traversal:

$$B_{\text{header}} = (\langle r \rangle_{v_\ell}, \texttt{ancestors}, \texttt{quorums}, \texttt{acks}, \texttt{dataCommit})_{v_\ell},$$

where `ancestors` contains n hash references to the latest blocks from distinct validators, with at least $2f+1$ from round $r-1$ and potentially one leader reference with partial signature $\langle \texttt{leader}_{r-1} \rangle_{v_i}$, and $\texttt{quorums} = \{\langle r-1 \rangle_{\text{agg}}, \langle \texttt{leader}_{r-2} \rangle_{\text{agg}}?\}$. The total size of a leader block header is $O(\kappa n)$.

As in Section 5, the full block includes the header and the transaction data: $B = (B_{\text{header}}, \texttt{data})$.

Protocol Operations with Compressed Format. The compressed block format preserves all protocol operations while reducing communication complexity for block header metadata to $O(\kappa n^3)$ per round in the worst case.

Commit rule. Certificate pattern detection requires identifying when $2f + 1$ validators at round $r + 1$ vote for the same leader block at round r. In the compressed format, each non-leader block includes an optional partial signature $\langle \texttt{leader}_{r-1} \rangle_{v_i}$ on the previous round's leader. The presence of this signature constitutes a vote for that leader; its absence indicates no vote. Skip patterns are thus detected when $2f + 1$ blocks omit the $\langle \texttt{leader}_{r-1} \rangle$ field. When $2f + 1$ partial signatures are present, they aggregate into $\langle \texttt{leader}_{r-2} \rangle_{\text{agg}}$ in subsequent blocks, serving as an explicit leader certificate.

Sequencing transaction data. In Starfish, transaction sequencing (Section 5) requires computing $\mathsf{hist}_{\mathsf{DA}}(B)$ for committed leader blocks—the set of blocks in the causal history whose data has been certified as available. The compressed format supports both components: (1) leader blocks contain n hash references to the latest blocks from each validator (with at least $2f + 1$ from round $r - 1$), enabling traversal of $\mathsf{hist}(B)$ by following `self` chains and leaders' ancestors; (2) the `acks` field contains aggregated threshold signatures $\langle \texttt{dataCommit} \rangle_{\text{agg}}$ for pending transaction data, that was acknowledged by a quorum of the network, and acknowledgment elements serve as an explicit Data Availability Certificate. Thus $\mathsf{hist}_{\mathsf{DA}}(B)$ is directly computable by traversing $\mathsf{hist}(B)$ and filtering to blocks with DACs in their acknowledgment fields.

Pacemaker modifications. To achieve $O(\kappa n^2)$ communication complexity for block headers in the happy case, we modify several pacemaker conditions. We denote the modified conditions with a star (*) to distinguish them from the original Starfish conditions:

- **A1*** *(Round advancement):* A validator can advance from round $r - 1$ to round r if it has received $2f + 1$ blocks from round $r - 1$, *or* if it has received a block containing the aggregated signature $\langle r - 1 \rangle_{\text{agg}}$ proving that a quorum of round-$(r - 1)$ blocks exists. This relaxes **A1** by allowing round advancement based on cryptographic proof of quorum existence. Condition **A2** remains unchanged.
- **B1*** *(Broadcast on block creation):* When a block is created, the validator broadcasts its own block immediately. After 2Δ, it sends unknown history to each peer, but only blocks that are not yet in that peer's DAG (as determined by the peer's latest known blocks).
- **B2*** *(Broadcast on round advancement):* When advancing to a new round, the validator waits 2Δ and then sends unknown history to each peer, but only blocks that are not yet in that peer's DAG.
- **C2*** *(Timeout):* The timeout is increased from $\delta_{TO} = 2\Delta$ to $\delta_{TO} = 4\Delta$.

When all validators are honest and synchronized, they advance rounds together and exchange only their own blocks via **B1***, avoiding redundant retransmission of others' blocks. When Byzantine validators cause desynchronization, the delayed broadcast via **B2*** ensures lagging validators eventually receive missing history. The increased timeout **C2*** guarantees that all round-$(r - 1)$ blocks from honest validators are delivered before the timeout expires, compensating for the delayed broadcasts.

7 Related Work

Byzantine Fault-Tolerant (BFT) consensus protocols have been extensively studied, e.g., [9, 24, 19, 20, 26, 11, 17, 13] with DAG-based protocols promising to improve throughput, scalability, and latency. In this section, we focus on the research most relevant to Starfish. In particular, we focus on round- and leader-based DAG protocols under the lens of certified vs. uncertified DAGs, data availability mechanisms, security considerations, and communication complexity and latency trade-offs. Round-based DAG-based protocols structure consensus by allowing validators to advance rounds only when a quorum of blocks from the current round is observed. A protocol can additionally be leader-based to optimize latency by introducing special commit rules for leader blocks, forming a "backbone sequence" that partitions the DAG into slices and enables deterministic sequencing.

Certified vs. uncertified DAGs. DAG-based BFT protocols such as Aleph [19], DAG-Rider [22], Tusk [11], Bullshark [37], Shoal [36], Shoal++ [1] Sailfish [33, 34] make use of Byzantine Reliable Broadcast (RBC) to guarantee data availability and to prevent equivocation. This ensures that validators construct a *certified DAG*, where every block is signed by a quorum, preventing conflicts but increasing communication overhead. Bullshark uses disjoint waves of two rounds and assigns one leader to each wave. By local interpretation of edges in the DAG, validators in Bullshark can commit the leader block and sequence all blocks in the causal past. Shoal [36] reduces the latency of non-leader blocks by interleaving two instances of Bullshark. BBCA-Chain [27] introduced a hybrid approach, applying RBC only to leader blocks while using Best-Effort Broadcast (BEB) for other blocks. However, when the Byzantine validators "selectively" send their blocks only to the leader, additional latency can be incurred to download the missing blocks, and the leader gets responsible for propagating a linear number of blocks. Sailfish [33] and Shoal++ [1] are two certified DAG-based protocols that similarly optimize their commit rules for leader blocks (by using first messages instead of certified blocks to count votes for leader blocks) and support multiple leaders in every round, improving the consensus latency. Sailfish additionally proposes using a no-vote certificate (an analogue of skip pattern in the uncertified DAG approach) to faster skip leader blocks from validators that fail to create their blocks in leader rounds and improve the latency in case of such misbehaviour. Shoal++, in turn, supports multiple instances of DAGs that improve end-to-end transaction latency. We note, however, that neither the multi-leader versions of these protocols nor the multi-DAG version of Shoal++ have proven to have performance improvements in the case of Byzantine nodes. Concurrent work Angelfish [40] presents a hybrid approach that allows non-leader parties to send lightweight votes via best-effort broadcast instead of reliably broadcasting full DAG vertices. While Angelfish uses a certified DAG and achieves improved communication complexity under moderate load by reducing the number of vertices, Starfish operates on an uncertified DAG and achieves linear transaction complexity through erasure coding and data availability certificates directly integrated into the DAG structure.

Hashgraph [3] pioneered the uncertified DAG-based approach, using Best-Effort Broadcast (BEB) for block dissemination, resulting in *uncertified* or *optimistic* DAGs where equivocating blocks may appear, and data availability has to be guaranteed by the consensus protocol. We note that all further uncertified DAG-based protocols inherit similar mechanisms for resolving equivocations from Hashgraph: validators always have to reference their own last blocks, and they sequence only blocks certified by $2f + 1$ validators. Here, a block B *certifies* another block L in its causal past if B can reach blocks from $2f + 1$ validators such that each block V from these $2f + 1$ *votes* for L, i.e., L is reachable from V and V can't reach any equivocation of the validator that created L. Hashgraph sequences certified blocks in a leaderless way using received timestamps of the respected blocks. Cordial Miners [23] introduced disjoint 3-round waves, and one leader was assigned in the first round of each wave. By committing leader blocks and deciding on the sequence of committed leader blocks, validators agree on how to slice the DAG and order blocks. Such a leader-based commit rule significantly reduced the commit latency. Mysticeti [2] is built on top of Cordial Miners and suggests pipelining waves with leaders, further reducing the latency. It was also proposed to use multiple leaders every round; however, the multi-leader version does not guarantee any improvement in latency in the case of Byzantine validators and, instead, may lead to a larger latency due to the undecided state of Byzantine leader blocks. We refer to [30] for a more detailed description and comparison between certified and uncertified approaches.

Starfish builds on top of Cordial Miners and Mysticeti with just pipelined leaders. Compared to the prior work, we separated block creation and round advancement events, each triggering broadcast of the unknown history to other validators, and added a new block creation condition **C3**. This design allows us to formally prove the liveness property for an uncertified DAG-based approach.

Data Availability and Dissemination. Certified DAGs outsource the data-availability problem to the additionally used RBC primitive, as blocks need to be certified before entering the consensus protocol. In addition, implementation of many certified DAG-based protocol makes use of Narwhal [11], which sep-

arates transaction dissemination from consensus using quorum-based availability certificates. Blocks in certified DAGs, in this case, become lightweight since they contain only digests of certified transaction batches. However, this separate layer increases the resulting end-to-end latency due to additional network trips for forming certificates for transaction batches. To improve the latency, Raptr [38] introduces a hybrid approach that integrates transaction certification into the consensus, allowing nodes to vote on a prefix of (uncertified) transaction batches anchored by the proposed leader block. Autobahn [18] similarly separates transaction data dissemination, by allowing each validator to create its own transaction "lane", a chain of transaction batches, and other validators' votes can shortly acknowledge the availability of a prefix of the lane.

For the uncertified DAG approach, Cordial Miners [23] proposed the "cordial dissemination" where validators "push" possibly unknown history of the DAG to other validators. This approach has quadratic communication complexity in theory; furthermore, it reaches practical limitations due to high bandwidth usage. Mysticeti uses a "pull" strategy, where validators can request missing history from their peers. This reduces bandwidth usage but might increase latency in bad network conditions due to these "missing ancestors" requests.

Using a new block structure, Starfish decouples transaction data dissemination from the consensus task. In Starfish, the transaction data from blocks is included in the final ordering only after creating data availability certificates ($2f + 1$ acknowledgments) directly on the DAG. This transaction dissemination concept is different from Narwhal, Raptr, and Autobahn. Compared to Starfish, in Narwhal, validators vote on blocks that include already "well-disseminated" certified transaction batches - this increases the end-to-end latency. Both Autobahn and Raptr can operate with uncertified transaction batches. To sequence transactions, they require Proofs-of-Availability consisting of $f + 1$ acknowledgments, as it guarantees that at least one honest validator has transaction data locally available. Starfish requires $2f + 1$ acknowledgments primarily because of the usage of Reed-Solomon codes - f Byzantine validators could lie, and $k = f + 1$ encoded shards are required for reconstruction (having this parameter k smaller increases the redundancy of the code, resulting in higher bandwidth requirements). PoAs in Starfish rely on the block's signatures. Autobahn leaves the task of constructing PoAs to the data dissemination layer. Both Starfish and Raptr integrate constructing PoAs into the consensus layer, but Starfish allows for a more granular transaction commitment. We note that a similar idea of encoding with RS codes could be applied to both Autobahn and Raptr - this would allow achieving linear amortized communication complexity for the transaction dissemination layer in the Byzantine environment.

Security and Adversarial Strategies. As pointed out in [30], uncertified DAGs are more vulnerable to Byzantine behaviour than certified DAGs, as the additional RBC prevents attacks focusing on equivocations and data availability.

Equivocating attack: Uncertified DAG-based protocols such as Cordial Miners, Mysticeti, and Starfish come with security proofs and their own method to handle equivocating blocks. Cordial Miners suggested how to avoid handling blocks from equivocators: once an equivocator is detected, an honest validator first references two equivocating blocks and then stops directly referencing blocks from it. Once it has learnt about the equivocation, any other honest validator stops directly referencing blocks from the equivocator. However, there could be malicious validators that do not create equivocation but reference equivocators, allowing multiple blocks from the same validator from the same round to exist in the resulting DAG. Fortunately, this kind of malicious behaviour can be handled similarly, and after GST, one can effectively exclude all equivocators from the consensus by processing $O_f(1)$ equivocating blocks. Nevertheless, equivocations can happen in practice by validators unintentionally, e.g., after crashing and restarting the consensus module, and such a strict punishing equivocators might be dangerous for the liveness of the protocol.

Data availability attack: The paper [19] discusses the chain-bomb attack, where spam forces honest valida-

tors to process excessive data, risking system overload.[5] It argues that DAG-based protocols require reliable broadcast to prevent such attacks, claiming that liveness is not guaranteed otherwise. In Adelie [10], such attack vectors are discussed in the context of Mysticeti and new validity rules for blocks were proposed. While this was shown to work in practice in the steady-state network, the suggested implicit validation of prior own blocks can impose serious liveness risks when no validator is able to create its block. In addition, the above papers present no quantitative results for actual attack scenarios. In this paper, we are the first to implement specific attack strategies and compare their influence on uncertified DAG-based BFT protocols.

Communication Complexity and Erasure Coding. The use of erasure and error-correcting codes in constructing Reliable Broadcast primitives is well established [12, 8]. Notably, Honeybadger BFT [28] applies erasure coding [8] to Bracha's RBC [7], reducing its communication complexity to achieve linear amortized communication complexity. Similarly, DAG-Rider [22] and Dumbo-NG [16] leverage erasure codes to optimize RBC communication costs. Certified DAG-based BFT protocols rely on an additional RBC layer that can be implemented using the RBC from [12], leading to a linear amortized communication complexity. These approaches, however, treat erasure coding as a separate optimization within the RBC protocol and do not integrate it directly into the DAG structure as we do in Starfish.

Our usage of RS codes is similar to [8] for RBC. While using RS codes tolerating *errors* (not *erasures*) in [12] allows getting rid of extra $\log n$ factor in the RBC communication complexity by avoiding the usage of Merkle trees, we stick with the erasure decoder for RS codes. First, the error correcting decoder does not help to improve the requirement for the message size in Starfish to achieve linear amortized complexity. Second, the existing error-correcting decoders have a larger constant factor in front of $n \log n$ in decoding complexity.

8 Conclusion and Outlook

We have presented Starfish, a novel uncertified DAG-based BFT consensus protocol that relies on a new commit rule for transaction data and an efficient data availability mechanism. Starfish achieves linear communication complexity in the worst case for large enough transaction data while maintaining low end-to-end transaction latency. Encoded Cordial Dissemination ensures that push-based broadcast remains scalable, addressing concerns about the inefficiency of naive push-based approaches. Our performance evaluation demonstrates that Starfish outperforms state-of-the-art certified DAG protocols and performs better than existing uncertified DAG protocols under adversarial conditions.

Implicit certification of own blocks. While Starfish allows for pushing the unknown history from the theoretical point of view, it can practically meet situations when a history to be pushed is over the bandwidth and cannot be received in full after Δ. An interesting research direction to further strengthen uncertified DAG protocols is to introduce implicit certification on a chain of blocks of a given validator. Specifically, each validator could be required to implicitly certify one of its own previous blocks when creating a new block. This constraint would naturally limit block flooding attacks; see a similar attempt in [10]. Such a mechanism could potentially bridge the security-performance gap between certified and uncertified DAGs without compromising the linear communication complexity. The theoretical analysis of this approach, particularly its impact on liveness, remains an open question.

Unlink transaction data from block header. It is clear that if transaction data is disseminated without binding its content to a block (using the Merkle root in the case of Starfish), it could reduce the end-to-end latency. Indeed, in this way, validators can make their acknowledgments faster since blocks in Starfish are created only after triggering one of the events **C1**, **C2**, or **C3**. To overcome this issue, one could either allow creating blocks more frequently, i.e., a chain of blocks by one validator in one round, or allow disseminating transaction data without associating it with any block. The first approach is similar to a multi-DAG version

[5] This attack is referred to as the "Fork-Bomb Attack" in [19].

of Shoal++ [1]. The second approach can be implemented in a similar fashion as it is done in Raptr [38] or Autobahn [18] with proofs-of-availability being created directly on the DAG.

9 Acknowledgment

We would like to thank Alexander Sporn and Piotr Macek from the IOTA Foundation for the discussions and improvements to the Starfish testbed design. We also thank Alberto Sonnino from Mysten Labs for extensive technical discussions around storage optimizations, block dissemination strategies, and performance considerations in DAG-based BFT protocols.

A Performance Evaluation

Starting with the Mysticeti codebase[6], we implement and open-source[7] our prototype of the Starfish protocol in Rust. It uses tokio[8] for asynchronous networking and utilizes TCP sockets for communication without relying on any RPC frameworks. For cryptographic operations, we rely on the ed25519-consensus[9] signature scheme and blake3[10] for cryptographic hashing. For the serialization and deserialization of protocol messages and internal data structures, we utilize bincode[11]. Unlike the originally implemented WAL (Write-Ahead Logging) storage, we stick with RocksDB[12] for persistent storage of the consensus data, such as blocks and commits. For encoding and decoding the transaction data, we employ the Reed-Solomon-SIMD[13] crate that implements Reed-Solomon codes over the field $\mathbb{F}_{2^{16}}$ with erasure decoding based on the Fast-Fourier transform. It also leverages SIMD instructions when working with shards that could be larger than two bytes (the field size $q = 2^{16}$). Our prototype does not contain the execution and ledger storage components to measure solely the consensus performance, similar to how it is done in all other past works; see [37, 1, 33, 2].

Core thread and connection tasks. Similar to the Mysticeti implementation[14], we use one dedicated thread for most critical operations such as updating the local DAG with received blocks, creating new blocks, applying the commit rule of Starfish and sequencing transactions. The preprocessing and verification of blocks received from a given validator is performed by a separate thread that handles all incoming and outgoing network messages with that peer. Preprocessing starts with checking whether the block was not yet included in the local DAG and then continues with encoding the transaction data, hashing the encoded data, constructing the Merkle tree and verifying the signature. Suppose a block with transaction data is verified. In that case, the connection thread creates two copies of the block together with their serializations: one contains the full transaction data and aims for storage, and another contains only one encoded shard of the transaction data and is used for potential further transmission. With one dedicated thread, this approach allows for the effective optimization of the critical path and the avoidance of race conditions.

To compare apples-with-apples, we include in our testbed the state-of-the-art uncertified DAG-based BFT protocols: we implement Cordial Miners [23] (whose implementation was not yet available) and include the Mysticeti [2] implementation[15] with several target optimizations and modifications. Recall that Mysticeti

[6] https://github.com/asonnino/mysticeti/tree/paper
[7] https://github.com/iotaledger/starfish
[8] https://tokio.rs
[9] https://docs.rs/ed25519-consensus/
[10] https://docs.rs/blake3/
[11] https://docs.rs/bincode/
[12] https://rocksdb.org/
[13] https://crates.io/crates/reed-solomon-simd
[14] https://github.com/MystenLabs/mysticeti
[15] https://github.com/MystenLabs/mysticeti

is currently deployed on the Sui[16] and IOTA[17] blockchains. In addition, for our baseline comparison, we used a publicly available code[18] for Sailfish [33], that was shown to improve the end-to-end latency of existing certified DAG BFTs such as Bullshark [37] (was adopted by the Sui blockchain before June 2024) and Shoal [36] by 10-25%. We note that Sailfish is implemented on top of Narwhal [11], which uses an additional worker layer to continuously disseminate transaction data. This allows for a higher throughput, but increases the resulting latency[19]. As demonstrated in [1], including transaction data directly to blocks allows for decreasing the latency by an additional 10-20%. [20]

In our evaluation, we seek to answer the following questions:

Q1: How well can a push-based dissemination scale with increasing validator count and transaction load?

Q2: How does Starfish's performance compare to state-of-the-art DAG-based BFT protocols in terms of latency and throughput?

Q3: What performance advantages does push-based dissemination with encoding offer over Mysticeti's pull-based approach and Cordial Miners' push-based approach without encoding under Byzantine attacks?

Q4: What computational overhead does Starfish imply compared to Mysticeti due to encoding and decoding operations, hashing more data, and Merkle tree computations?

Experimental Setup. We use the Amazon Web Services (AWS) to mimic the deployment of a globally decentralized network. We evaluate performance using Amazon EC2 m5d.4xlarge instances, each equipped with 16 vCPUs, 64 GiB RAM, 10 Gbps network bandwidth, and 600 GB of NVMe SSD storage. Our testbed consists of $\{10, 25, 40, 100\}$ validators spread evenly[21] across ten global regions: US East (Virginia, USE1), US West (California, USW1), Canada Central (Quebec, CAC1), EU West (Ireland, EUW1), EU South (Madrid, EUS2), EU North (Stockholm, EUN1), South America East (São Paulo, SAE1), Asia Pacific South (Mumbai, APS1), Asia Pacific Southeast (Sydney, APSE2), and Asia Pacific Northeast (Tokyo, APNE1). Before starting the simulation, we have measured round-trip latencies between different regions. To construct the latency matrix, Table 4, we deploy virtual machines in selected AWS regions and execute ping tests to measure round-trip times (RTT). Each value is averaged over 10 measurements and rounded up. The gathered latencies provide inter-region communication delays for our specific measurements. The network round-trip times (RTTs) range from 14ms between nearby regions to 309ms between distant regions, with cross-continental latencies typically between 200-300ms.

Each honest validator is connected to one client, which issues a continuous stream of random transactions, each consisting of 512 bytes. The end-to-end latency for validator v_i is measured as the time between when a transaction arrives to a validator v_j who includes it in the block and when the transaction is eventually sequenced and ready to be executed by v_i. We average the latency measurements across all validators. When referring to throughput, we mean the number of sequenced transactions over the entire duration of the run divided by the duration in seconds. We run each experiment for over three minutes.

We use a timeout $\delta_{TO} = 2\Delta$ of 600ms in each round for Starfish, Cordial Miners, Mysticeti; Sailfish by default uses a 3s timeout. We note that 600ms is much larger than the largest RTT (see Table 4). However, the algorithm assumes an instant reaction, which is not possible in case of a high load due to the overhead

[16] https://sui.io/mysticeti

[17] https://docs.iota.org/about-iota/iota-architecture/consensus

[18] https://github.com/nibeshrestha/sailfish

[19] We note that in the concurrent work of Sailfish++ [34], the Narwhal layer is removed and a signature-free implementation is used. This allowed the authors to significantly improve the end-to-end latency.

[20] We do not include Shoal++ [1] in our comparison due to the lack of a publicly available codebase at the time we carried out our simulations.

[21] In case of 25 validators, we use three validators for each of the first five regions and two for each of the remaining regions.

	USE1	USW1	CAC1	EUW1	EUS2	EUN1	SAE1	APS1	APSE2	APNE1
USE1	1	65	14	68	104	110	112	201	198	146
USW1	65	1	78	127	163	172	175	226	137	108
CAC1	14	78	1	67	106	103	122	189	196	142
EUW1	68	127	67	1	29	38	176	125	254	199
EUS2	104	163	106	29	1	50	215	143	281	238
EUN1	110	172	103	38	50	1	220	148	268	245
SAE1	112	175	122	176	215	220	1	299	309	254
APS1	201	226	189	125	143	148	299	1	150	140
APSE2	198	137	196	254	281	268	309	150	1	101
APNE1	146	108	142	199	238	245	254	140	101	1

Table 4: Round-trip latencies between regions (milliseconds)

Protocol	$n = 10$	$n = 40$	$n = 100$
Sailfish	1.815	1.870	2.874
Mysticeti	0.510	0.505	0.533
Cordial-Miners	0.540	0.610	0.826
Starfish	0.654	0.674	0.895

Table 5: Baseline (all honest) e2e latency (in sec.) at minimal load for different validator counts.

in the block creation and flushing all consensus data to the disk. Nevertheless, we deem the chosen timeout is sufficient for all experiments we provided. For Mysticeti, we enabled only one leader in each round since multiple leaders in each round do not show any benefit in the case of Byzantine nodes. In addition, validators in our modification of the Mysticeti implementation request missing parents not from a limited number of other validators like it is done in the original testbed[22], but whenever a block has this missing parent. This modification allows for obtaining more reasonable results for Mysticeti under Byzantine attacks.

Baseline Performance. Figure 2 illustrates throughput and end-to-end p50 latency for 10, 40, and 100 validators under steady-state conditions (no Byzantine nodes and stable latencies between validators). We include error bars indicating p25 and p75 latencies for all protocols except Sailfish. Table 5 shows the baseline p50 latency under minimal load.

For **Q1** (scalability of push-based dissemination), Starfish demonstrates robust scaling behavior, maintaining sub-second latency up to approximately 150K TPS across all validator counts (10, 40, and 100). The consistent performance patterns across different validator counts show that push-based dissemination scales well with increasing network size. However, Cordial Miners, with its push-based broadcast approach *without* encoding, fails to achieve comparable performance due to significantly higher communication complexity, resulting in increased bandwidth requirements and CPU load.

For **Q2** (comparison with state-of-the-art), Starfish achieves competitive performance against other DAG-based protocols. While Mysticeti (uncertified DAG BFT) shows lower baseline latency under small load (see Table 5), Starfish maintains more stable latency scaling for large validator counts (40 and 100) and high throughput. This advantage likely stems from high-frequency block regimes, where validators process blocks out of expected order, potentially causing missing ancestors or history requests in Mysticeti. A single request may prove insufficient in Mysticeti, leading to delayed block creation and increased core thread task. Compared to Sailfish (certified DAG BFT), which shows higher baseline latency (1.8-2.8s), Starfish consistently delivers significantly better latency across all throughput ranges. Starfish substantially

[22] `https://github.com/MystenLabs/mysticeti`

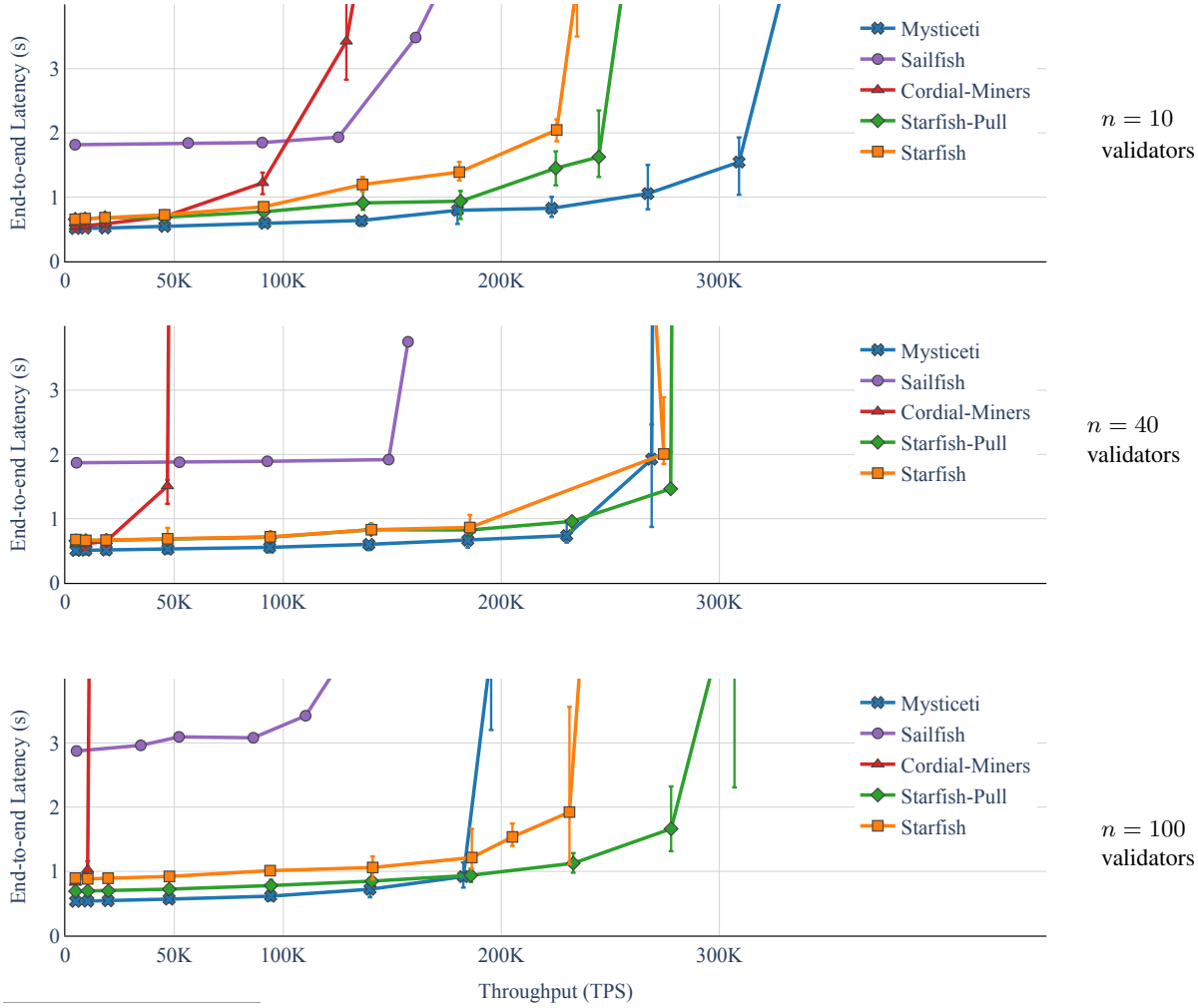

Figure 2: Performance results (e2e latency vs throughput in txs/sec with 512B-sized transactions) for 10, 40 and 100 validators in a geo-distributed network over 10 regions in the steady state. Comparison includes Sailfish (certified DAG), Mysticeti and Cordial Miners (uncertified DAGs), and two versions of Starfish.

outperforms Cordial Miners beyond 50K TPS due to Cordial Miners' scaling limitations.

Performance Under Attack Scenarios. To evaluate **Q3** (advantages of push-based dissemination with encoding), we analyze performance under Byzantine attacks with $n = 25$ validators and 40,000 tx/sec load. Table 6 reports end-to-end latency (seconds) and bandwidth efficiency ratio—defined as average validator bandwidth divided by (total sequenced transactions × transaction size). This metric approximates $A(n)/n$, where $A(n)$ is the amortized communication cost per byte of sequenced transaction.

We implement two closely related attacks: *Chain Bombs* and *Equivocating Chains Bomb*. In a Chain Bombs attack, Byzantine nodes operate without equivocation but exploit the leader schedule and use a special broadcasting strategy. For each $i \in \{1, \ldots, 8\}$, Byzantine node v_{3i} occupies position $3i$ in the leader schedule. This arrangement represents the worst-case scenario for both Mysticeti and Starfish commit rules, as their indirect decision rules require three consecutive honest leader blocks in the DAG to break the dependency between potentially undecided states of Byzantine leader blocks. Each Byzantine node i employs a selective broadcasting strategy: when acting as a leader, it pushes its chain of blocks exclusively to the validator scheduled as leader at position $3i + 1$, while ignoring all other communication and block requests.

Byzantine strategy	Cordial Miners	Mysticeti	Starfish	
Baseline	22.65	**1.05**	3.98	Bandwidth efficiency
(all honest validators)	0.721	**0.521**	0.690	Latency (in sec)
Chain Bomb Attack	16.81	**1.03**	3.05	Bandwidth efficiency
(8 attackers)	4.944	23.146	**1.106**	Latency (in sec.)
Equivocating Chains Bomb	21.94	**1.08**	4.67	Bandwidth efficiency
(1 attacker)	0.967	18.450	**0.776**	Latency (in sec.)

Table 6: Performance comparison of uncertified DAG-based BFT protocols under honest and Byzantine scenarios with $n = 25$ nodes distributed over 10 regions and 40,000 txs/sec load. First value shows bandwidth efficiency ratio (average validator bandwidth divided by total sequenced data), second shows p50 end-to-end latency (in seconds).

The Equivocating Chains Bomb attack differs by concentrating malicious behavior in a single Byzantine validator that creates $n - 1$ equivocating blocks per round. For each $i \in \{1, ..., n - 1\}$, the ith equivocating block references only its counterpart (the ith block) from the previous round. The Byzantine validator strategically times its attack by sending each chain of ith blocks to the ith validator immediately before the round when that validator becomes the leader.

Under both attacks, Starfish's push-based dissemination with encoding demonstrates key advantages:

- *Latency Resilience*: Starfish maintains the lowest latency among all protocols. In contrast, Mysticeti's pull-based approach, while bandwidth-efficient (ratio close to 1), suffers significant latency increases and potential liveness issues under adversarial network conditions.
- *Bandwidth Efficiency*: Starfish achieves a bandwidth efficiency ratio around 4. This is substantially better than Cordial Miners' 16-23. This efficiency stems directly from encoding: Cordial Miners, using uncoded push-based broadcast, requires processing significantly more data, resulting in higher latencies.
- *Balanced Design*: Unlike Mysticeti's pull-based optimization that trades latency for bandwidth, Starfish's encoded push-based approach maintains both low latency and reasonable bandwidth usage under attack.

These results demonstrate that Starfish's push-based dissemination with encoding effectively balances performance metrics under Byzantine conditions, addressing limitations of both pull-based (Mysticeti) and uncoded push-based (Cordial Miners) approaches.

Computational overhead for processing data. To evaluate **Q4** (computational overhead), we first estimate the computation overhead in theory and then measure the performance of different components of our implementation of Mysticeti and Starfish.

As mentioned in Section 2, encoding and decoding complexities for messages of size M using Reed-Solomon codes of length n (number of validators) is $O(M \log n)$. Assume that hashing of data of size M takes time $O(M)$, signature generation and verification are $O(1)$. Merkle tree construction from encoded shards then takes $O(M) + O(n)$ time, whereas Merkle proof verification for a given shard takes time $O(M/n) + O(\log(n))$. Block creation in Starfish is then $O(M \log n) + O(n)$ compared to $O(M)$ in Mysticeti. Block verification of a block from an honest validator is $O(M \log n)$ compared to $O(m)$ in Mysticeti. Block verification and reconstruction of a block from a Byzantine validator could take $O(M \log n) + O(n \log n)$ in the worst case in Starfish (after receiving block headers and block shards from all other validators). We note that in the current architecture that we used, where all communication with different peers is processed concurrently (not sequentially), verification of a block in Mysticeti could take (across all connections) time up to $O(Mn)$ as the same block could be processed by coming from different peers.

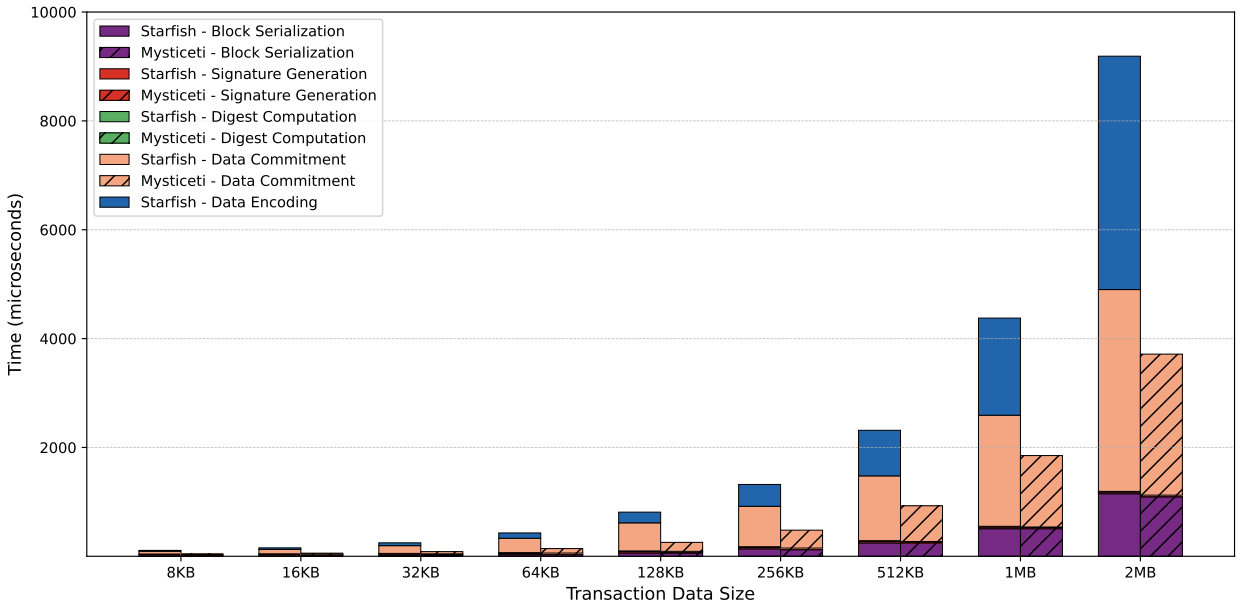

Figure 3: Timing breakdown for block creation in Mysticeti and Starfish for different transaction data sizes and 100 validators. Only operations with significant time are represented in the plots: encoding transaction data (converting transactions to encoded shards), computing data commitment (hashing and Merkelization of encoded shards in Starfish or hashing serialized transaction data in Mysticeti), digest computation, and signature generation.

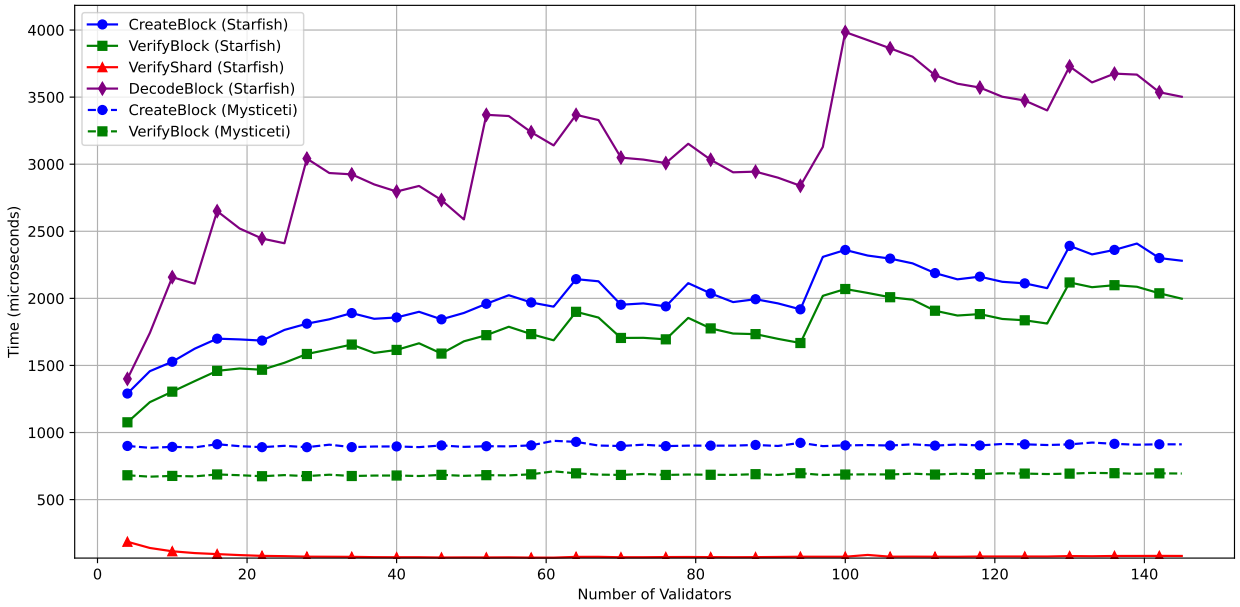

Figure 4: Performance of key functions in Starfish and Mysticeti across increasing committee sizes (4 to 145 validators) with a fixed transaction data size of 512KB. The plot shows `CreateBlock` and `VerifyBlock` for both protocols, and `VerifyShard` and `DecodeBlock` for Starfish only.

In Figure 3, we provide a timing breakdown for block creation for different transaction data sizes. We use 100 validators for all computations in this case. One can observe that encoding takes a significant portion of time in Starfish. In addition, computing data commitment for Starfish requires hashing more data and building a Merkle tree, compared to simple hashing of the actual transaction data in Mysticeti.

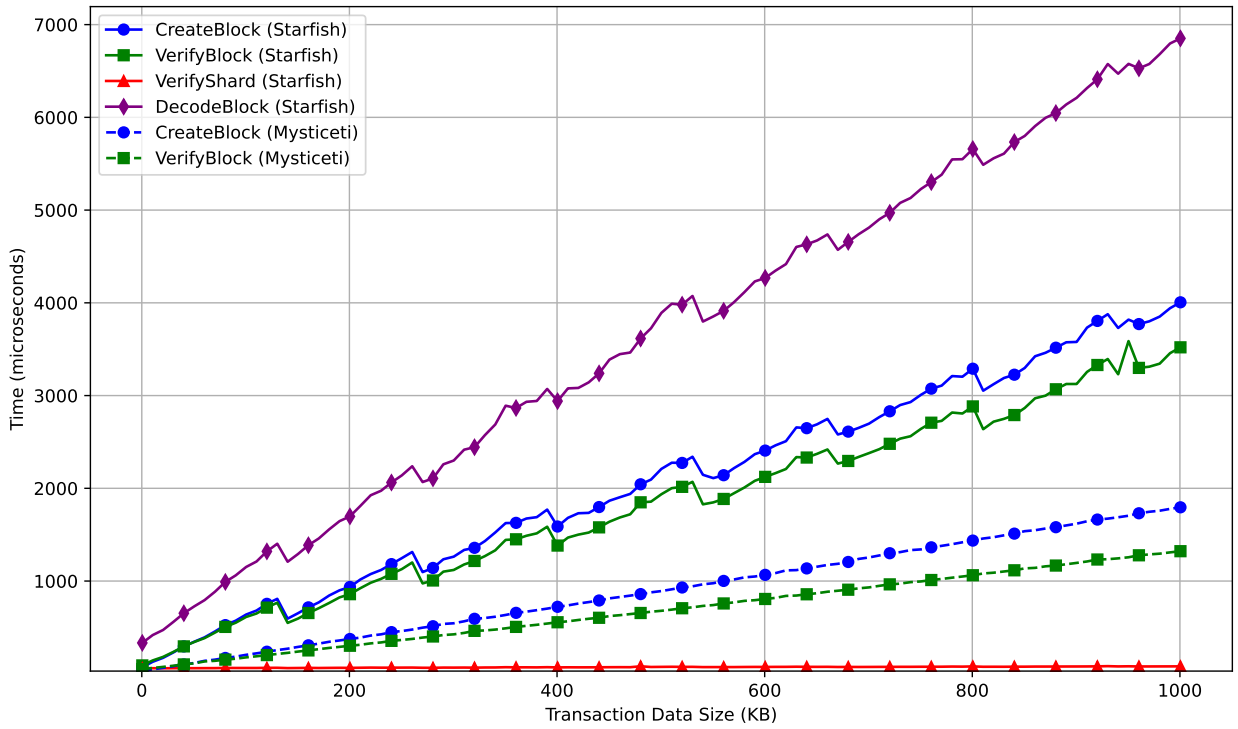

Figure 5: Timing performance of key functions in Starfish and Mysticeti across increasing transaction data sizes (0.5 KB to 1 MB) with a fixed committee size of 100 validators. The plot shows `CreateBlock` and `VerifyBlock` for both protocols, and `VerifyShard` and `DecodeBlock` for Starfish only, as Mysticeti does not involve sharding or decoding in this context.

In Figure 4 and Figure 5, we present the key functions in the Starfish and Mysticeti protocols in two scaling regimes. **Network scaling**: the network size increases from 4 to 145 validators, with a fixed transaction data size of 512 KB. **Transaction data scaling**: the number of validators is set to 100 and the transaction data increases from 512B to 1MB. The plots includes timings for block creation (`CreateBlock`), block with full transaction data verification (`VerifyBlock`), block with one shard verification (`VerifyShard`), and block decoding from $f+1$ shards (`DecodeBlock`) for Starfish, and `CreateBlock` and `VerifyBlock` for Mysticeti, measured in microseconds. In the network scaling regime, these functions remain mostly constant (with a very slight increase to more hash references in blocks) for Mysticeti; Starfish requires more time for creating and verifying blocks, aligning with expected $O(M \log n)$ behaviour. Note that some jumps are due to library-related RS encoding and decoding optimizations. We also note that `VerifyShard` time decreases as committee size grows, due to smaller shard sizes with more validators. In the transaction data scaling regime, all functions for both Mysticeti and Starfish grow linearly (in M) as expected.

To put things in perspective, we consider one example: the target throughput 128.000 txs/sec with 100 validators. In this case, each validator creates (approximately) 10 blocks every second, i.e., each block contains on average 128 transactions, thereby having transaction data of size 64KB. Block creation takes 430 µs in Starfish vs 142 µs in Mysticeti. Block verification of the block with full transaction data takes 418 µs in Starfish vs 130 µs in Mysticeti. Decoding takes 560 µs and verification of one shard 64 µs, out of which Merkle proof verification takes 7 µs. It is possible for $f = 33$ Byzantine validators to distribute data only among $f + 1$ honest validators, forcing the remaining validators to decode transaction data. Decoding is guaranteed to be correct since at least one honest validator had the original data and made verification of the data commitment. In the worst case, the decoding takes 560 (time to decode) $\times$ 33 (Byzantine validators) $\times$ 10 (blocks per second) = 184.4 ms, which can be handled by a single thread. Block with one shard

verification has to be parallelized as it takes at most 64 µs (time to verify a block with a shard) $\times$ 100 (total number of shards) $\times$ 33 (Byzantine validators) $\times$ 10 (blocks per second) = 2.112 seconds.

In summary, Starfish's encoding-based approach incurs a computational overhead in steady-state operation, with a theoretical complexity increase of $O(\log n)$ for block creation and verification due to Reed-Solomon encoding, transaction data reconstruction due to Reed-Solomon decoding. Empirically, in typical blockchain scenarios, this overhead manifests as a multiplicative factor of 3-4. In worst-case adversarial scenarios, where Byzantine validators control network delays and concurrent block processing is employed, Starfish's computational complexity can be better than the one of Mysticeti, which may scale to $O(Mn)$ due to redundant block processing. This trade-off underscores Starfish's suitability for security-critical distributed systems, balancing higher CPU usage with robust performance under adversarial conditions.

B Byzantine Reliable Broadcast Definition

In a Byzantine Reliable Broadcast (RBC), a designated sender v_k invokes $r\_bcast_k(m, r)$ to propagate its input m in some round $r \in \mathbb{N}$. Each party v_i attempts to output value $r\_deliver_i(r, v_k)$, where v_k is the designated sender and r is the round number in which v_k sent the message. The reliable broadcast primitive [7] satisfies the following properties:

- **Validity.** If an honest party v_k calls $r\_bcast_k(m, r)$, then every honest party v_i eventually outputs $m = r\_deliver_i(r, v_k)$.
- **Totality.** If some honest party outputs a value, then eventually all honest parties will output a value.
- **Agreement.** If two honest parties v_i and v_j output their values for a given round r and designated sender v_k, then the output values are the same: $r\_deliver_i(r, v_k) = r\_deliver_j(r, v_k)$.

C Detailed Description of Starfish Protocol

In this section, we provide a detailed description of Starfish. Pseudocodes of the core algorithms used for block creation, round advancement, commit rules, and sequencing can be found in Appendix D.

History of Known Block Headers. We denote $\mathsf{hist}(B)$ as the causal *history* (sometimes called the *past cone*) of a block B, representing the set of all block headers that are transitively reachable from B via its hash references (in Ancestors). Every validator may have a different perception of the current DAG. Let us denote by DAG_i the local DAG of validator v_i. Due to network latency, a validator does not necessarily know the local DAGs of the other validators. However, every block expresses knowledge about the existence of other blocks in its causal history. Therefore, we can define the history of known block headers known to a validator v_j from the point of view of validator v_i as follows:

$$\mathsf{hist}_i(j) = H_{i,j} \cup P_{i,j}, \tag{5}$$

where $H_{i,j} = \bigcup \mathsf{hist}(B)$ is the union over all blocks B in DAG_i created by validator v_j and $P_{i,j}$ is all blocks that are sent (or pushed) from v_i to v_j. The latter is added to the union $H_{i,j}$ since v_i already pushed the unknown history even though some of the pushed blocks might not have yet reached v_j.

C.1 Identifying DAG Patterns

Proposer Slot. We use the concept of *proposer slot* as in Mysticeti. A proposer slot represents a tuple (validator, round) and can be either empty or contain the validator's proposal(s) for the respective round. Starfish uses a round-robin mechanism to assign proposer slots to validators. Validator v_i has its proposer slot in rounds $i, i + n, i + 2n \ldots$.

State of Proposer Slots. Every proposer slot is in one of the following states: TO-COMMIT, TO-SKIP, or UNDECIDED. Initially, all slots are set to be UNDECIDED. The goal of the commit rule of Starfish is to ensure that each validator, by interpretation of local DAG, makes a decision TO-COMMIT or TO-SKIP for every proposer slot.

The TO-COMMIT state allows to commit the leader block in a proposer slot. The crucial state is the UNDECIDED, which forces all subsequent proposer slots to wait, mitigating the risk of non-deterministic commitments due to network asynchrony without the need for a buffer round as prior work [22, 11, 37, 16]. Finally, the TO-SKIP state allows to exclude proposer slots that will not get enough votes or certificates to be skipped.

Proposing, Voting and Certifying Round. If a proposer slot at round r is fixed, we call round r *proposing*, round $r+1$ *voting*, and round $r+2$ *certifying*. The latter two are used to make the decision for the proposer slot.

Voting for a Leader Block. Recall that each block header from validator v_i created at round $r+1$ contains a set Ancestors. This set includes hash references to block headers from $2f+1$ validators created at round r. Each block from voting round $r+1$ can vote on a leader from the proposer round r. A block B from round $r+1$ is *voting* on a leader block L from round r if L is the first block in Ancestors of B from the leader of the proposer slot of round r.

Patterns for Leader Blocks. Starfish, as its predecessors, operates by interpreting the structure of the DAG and finding patterns that create certificates or skip the attempt to find certificates for leader blocks:

1. The *skip pattern*: blocks from at least $2f+1$ validators at round $r+1$ *do not* vote for a leader block from round r. Note that there might be no proposal for a leader slot or more than one proposal for a leader slot in the case of equivocations. The slot is skipped if, for any block proposal, we observe $2f+1$ blocks in the next round that do not vote for it.
2. The *certificate pattern*: blocks from at least $2f+1$ validators at round $r+1$ *vote* for a leader block B of round r. We then say that B is *certified*. Any block from certifying round $r+2$ that contains in its history such a pattern is called a *quorum certificate* (QC) for the block B.

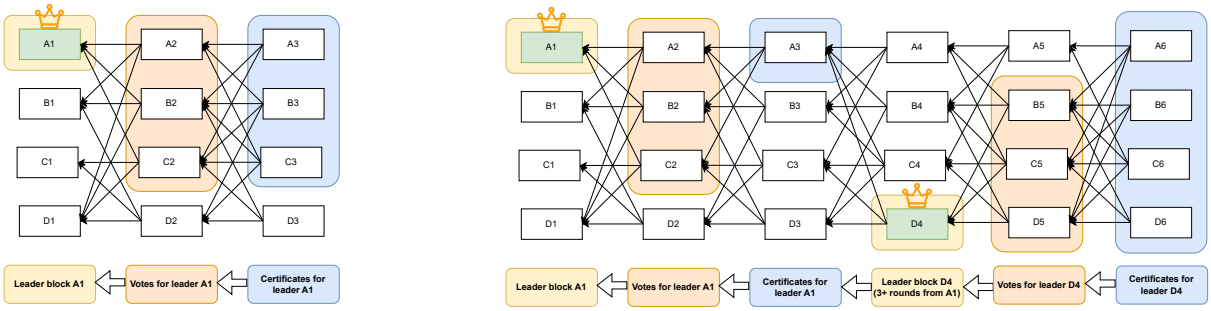

Figure 6: Direct (on the left) and indirect (on the right) commit of leader block $A1$

Quorum Certificates for Leader Blocks. Using these patterns, we derive quorum certificates implicitly for leader blocks by interpreting the DAG structure. Certification ensures that at most one leader block from a given proposer slot can be certified. This prevents conflicts caused by equivocation, as conflicting blocks cannot simultaneously meet the $2f+1$ quorum requirement.

Pattern for Data Availability. In Starfish, we decouple block headers from their transaction data. This means that unlike Cordial Miners and Mysticeti, we can't immediately sequence all transactions by *slicing* the DAG using the committed leader blocks. One has to check the data availability of the transaction data

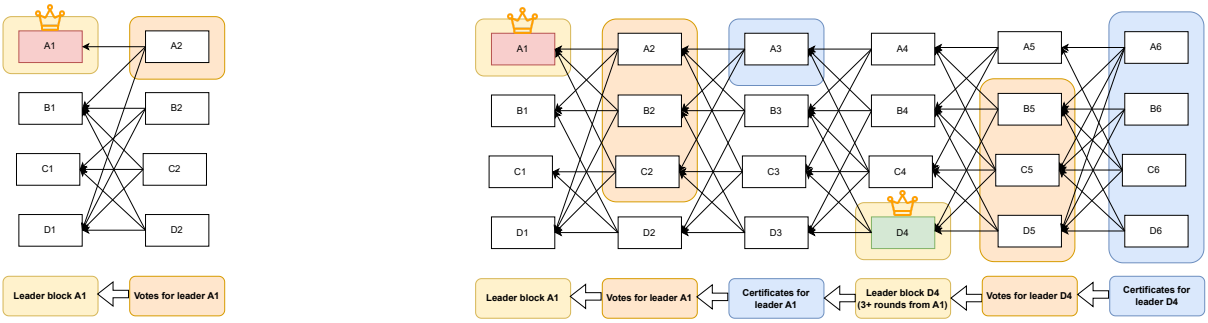

Figure 7: Direct (on the left) and indirect (on the right) skip of leader slot $(A, 1)$

within the slices by inspecting Acknowledgment fields of blocks in the slices.

Similar to the *certificate pattern* above, we introduce

3. The *acknowledgment pattern*: blocks from at least $2f + 1$ validators at rounds $> r$ *acknowledge the availability of the transaction data of a block* B from round r.

We say that B is *certified for data availability*. In contrast to the certificate pattern, we do not require the acknowledgment pattern to appear in one specific round $r + 1$ but allow it to appear at any later point after round r.

Data Availability Certificates for Blocks. The *Data Availability Certificate (DAC)* ensures that the transaction data $\mathsf{tx}(B)$ associated with a block header B is available to a quorum of validators. Any subsequent block that contains in its causal history such an acknowledgment pattern is called a *Data Availability Certificate (DAC)* for the block B.

C.2 Sequence of Committed Leader Blocks

This section describes the mechanism to find a sequence of committed leader blocks; it is the same as in Mysticeti.

All proposer slots are initially in the UNDECIDED state. The goal is to mark all proposer slots as either TO-COMMIT or TO-SKIP by detecting certificate or skip patterns or relying on an *indirect decision rule*. The whole decision rule operates in three steps:

Step 1: Direct Decision Rule. The validator applies the following *direct decision rule* to attempt to determine the status of the proposer slot.

The validator marks a proposer slot as TO-COMMIT if it observes QCs from at least $2f+1$ validators for that slot. In case of equivocation and multiple proposals for one slot, the proposal or block header that received a required quorum of QC is committed.

The direct decision rule marks a proposer slot as TO-SKIP if it observes a *skip pattern* for that slot.

If the direct decision rule fails to mark a slot as either TO-COMMIT or TO-SKIP, the slot remains UNDECIDED and the validator resorts to the *indirect decision rule* presented in step 2 below.

Step 2: Indirect Decision Rule. If the direct decision rule leaves a slot undecided, the validator resorts to the indirect decision rule to attempt to decide. This rule operates in two stages. It initially searches for an *anchor*, which is defined as the first proposer slot with the round number $(r' \geq r+3)$ that is already marked as either UNDECIDED or TO-COMMIT.

If the anchor is marked as UNDECIDED the validator marks the slot as UNDECIDED. Conversely, if the

anchor is marked as TO-COMMIT, the validator marks the slot either as TO-COMMIT if the anchor contains a path to a QC for a leader block from the proposer slot or as TO-SKIP otherwise.

Step 3: Commit Sequence. After processing all proposer slots, the validator derives an ordered sequence of leader blocks. Subsequently, the validator iterates over that sequence of proposer slots, committing all slots marked as TO-COMMIT and skipping all slots marked as TO-SKIP until it reaches the first slot marked as UNDECIDED.

We will prove, in Lemma 6, that, eventually, every proposer slot will be decided.

C.3 Sequencing Transaction Data

In Starfish, we need to specify how the actual transactions are ordered. For each block B let us denote $\mathsf{hist}_{\mathsf{DA}}(B)$ the set of all blocks D in $\mathsf{hist}(B)$ such that B is a Data Availability Certificate for $\mathsf{tx}(D)$ in $\mathsf{hist}(B)$. Given the sequence of committed leader blocks $C_1, \ldots, C_N$, derived in Appendix C.2, and the fixed topological ordering of block headers in the DAG, this allows to construct a total ordering over the sequenced transaction data in $\mathsf{hist}_{\mathsf{DA}}(C_N)$ as follows

$$\mathsf{sort}(\mathsf{hist}_{\mathsf{DA}}(C_1)), \mathsf{sort}(\mathsf{hist}_{\mathsf{DA}}(C_2) \setminus \mathsf{hist}_{\mathsf{DA}}(C_1)), \ldots, \mathsf{sort}(\mathsf{hist}_{\mathsf{DA}}(C_N) \setminus \bigcup_{i}^{N-1} \mathsf{hist}_{\mathsf{DA}}(C_i)). \tag{6}$$

This gives a total order of the blocks with availability guarantees. One has to remove potential equivocating blocks from this ordering by preferring the first block of a given validator from a given round. By removing equivocation, we guarantee the integrity property of Starfish as BAB and establish a total ordering of the transactions contained in the remaining blocks.

C.4 Encoded Cordial Dissemination

Compared to push-based cordial dissemination, Cordial Miners [23], we propose modifying the broadcast mechanism and introducing a more communication-efficient method. Our approach consists of two steps aimed at reducing communication overhead. First, we decouple the broadcast of the "block header" from the dissemination of transaction data. Second, we employ an erasure code to broadcast encoded parts of the transaction data for blocks of others' validators.

Reed-Solomon Codes. An (n, k) Reed-Solomon code over a Galois field $\mathbb{F}_q$ with $n \leq q$ encodes k data symbols into a codeword of n symbols. We use systematic encoding, meaning the first k encoded symbols coincide with the input data. Let $\mathrm{encRS}(m, n, k)$ denote the encoding algorithm that takes a message m of k symbols and outputs n encoded symbols. Reed-Solomon codes support two decoding modes:

Erasure decoding. When the positions of missing or invalid symbols are known (erasures), any k valid symbols suffice to reconstruct the original message. The decoding complexity is $O(n \log^2 n)$ using fast algorithms based on the Fast Fourier Transform [25].

Error-correcting decoding. When the positions of corrupted symbols are unknown (errors), any $k + 2e$ symbols suffice to correct up to e errors. Standard algorithms include Berlekamp-Welch [39] and Gao's algorithm [15].

In Starfish, we use a systematic Reed-Solomon code of length $n = 3f + 1$ and dimension $k = f + 1$ over alphabet $\mathbb{F}_q$ where q is a power of two. With these parameters, collecting $2f + 1$ shards allows correcting up to f errors since $2f + 1 = (f + 1) + 2f = k + 2f$. For erasure decoding, any $k = f + 1$ valid shards suffice. In the default configuration, Starfish uses erasure decoding with Merkle proofs for shard verification. Alternatively, error-correcting decoding can be used without Merkle proofs—the validator collects $2f + 1$ shards, runs the error-correcting decoder, and verifies the reconstructed data against the data commitment `dataCommit` in the block header. This approach reduces bandwidth since shard proofs of size $O(\kappa \log n)$

per shard are not required.

Encoding and Merkle Tree Construction. We make use of Merkle trees to ensure integrity verification for encoded transaction data in case of erasure decoding.

1. *Encoding the Transaction Data:* The transaction data $\mathsf{tx}(B)$ is encoded into $n = 3f + 1$ shards using the Reed-Solomon encoding function:

$$c = (c_1, c_2, \ldots, c_n) = \text{encRS}(\mathsf{tx}(B), n = 3f + 1, f + 1).$$

   The first $f + 1$ shards, $c_1, c_2, \ldots, c_{f+1}$, are called information shards. They represent the transaction data; the other pieces are parity shards. The original transaction data can be recovered from any $f + 1$ shards.

   More precisely, by setting $k = f + 1$, the transaction data is represented as

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,k} \\ \ldots & \ldots & \ldots & \ldots \\ x_{t,1} & x_{t,2} & \ldots & x_{t,k} \end{pmatrix}$$

   where each $x_{i,j} \in \mathbb{F}_q$ and $t = \frac{M}{(f+1)\log_2 q}$ with M being the size (in bits) of $\mathsf{tx}(B)$[23]. Every row is treated as an information sequence and encoded into a length-n sequence of elements from $\mathbb{F}_q$ with the Reed-Solomon code

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,n} \\ \ldots & \ldots & \ldots & \ldots \\ x_{t,1} & x_{t,2} & \ldots & x_{t,n} \end{pmatrix}.$$

   Every column is called a *shard*, the first k are information shards, which represent the transaction data, the last $n - k$ are parity shards. In other words, k shards $c_1, \ldots, c_k$ were encoded into n shards $c_1, \ldots, c_n$ by computing parity shards $c_{k+1}, \ldots, c_n$.

2. *Creating the Merkle Tree:* After encoding the data, we compute the hash of every shard:

$$h_i = \text{hash}(c_i) \quad \text{for} \quad i = 1, \ldots, n.$$

   These hashes are used to construct a Merkle tree, where the leaf nodes are the hashes of the shards. The root of the Merkle tree, h, is computed as:

$$h = \text{MerkleRoot}(h_1, h_2, \ldots, h_n).$$

   The Merkle root h acts as a commitment to the entire set of encoded shards and is used by other validators to verify the integrity of the data.

3. *Merkle proofs:* To validate the integrity of an individual shard c_i, the ith validator provides a Merkle proof. A Merkle proof

$$p_i = \text{MerkleProof}(c_i)$$

   is a sequence of sibling hashes from the Merkle tree that allows a validator to reconstruct the Merkle root h using only the shard c_i and its corresponding proof p_i.

Broadcasting Unknown History. Whenever the broadcast history event is triggered, see Table 3, a validator v_i broadcasts to v_j a batch of blocks. This batch contains all blocks that are currently in DAG_i but not in

[23] In case, M is not divisble by $(f + 1)\log_2 q$ we use the ceiling operation and zero padding

$\mathsf{hist}_i(j)$, see (2). In addition, let A_i denote the set of all blocks whose transaction data is available to v_i. The batch also includes all blocks that were not in A_i when the previous unknown history event was triggered

Recall the structure of blocks, see Section 2. Each block consists of a block header and a block body, where the block header

$$B_{\text{header}} = (r, \texttt{ancestors}, \texttt{acks}, \texttt{dataCommit})_{v_i}$$

is signed. The block body optionally contains the transaction data and an encoded shard of the transaction data, along with Merkle proof for verification. A full block is not signed and defined as:

$$B = \langle B_{\text{header}}, \text{Option(TransactionData)}, \text{Option(EncodedShard, MerkleProof)} \rangle.$$

Specifically, all block headers in the set difference of DAG_i and $P_i(j)$ are sent in a batch. The block bodies of the corresponding blocks are formed depending on whether the block was created by v_i or not.

Block Body for Own Blocks. Once the block creator has completed the encoding and Merkle tree construction, the Merkle root is added to the block header, which is then signed. The original transaction data is added to the block body. The block creator broadcasts this block within the batch.

Block Body of Blocks from Other Validators. There are two options for the block body depending on whether a block was received from the block creator or not:

Opt 1: If the original transaction data of block B is not yet available to validator v_i, then v_i does not include anything in the block body and v_i broadcasts just the block header of B. In this case, B was not in the history of known block headers $\mathsf{hist}_i(j))$ before this transmission, see (2).

Opt 2: If validator v_i receives or reconstructs the full transaction data of a block B, then it performs the encoding and constructing the Merkle root described above to calculate the tuple (c_i, p_i), where

- c_i is the ith encoded shard derived from the transaction data.
- p_i is the Merkle proof that allows any receiving validator to verify the integrity of c_i.

This tuple is then added to the block body for block B and sent to all validators that have not yet acknowledged the availability of $\mathsf{tx}(B)$.[24] In this case, B was included to A_i after the previous unknown history event was triggered.

Transaction data verification. The transaction data $\mathsf{tx}(B)$ for a block B can be reconstructed or verified by validators in two distinct scenarios:

1. *Verification of Transaction Data Broadcasted by the Block Creator*:
   - The block creator directly broadcasts the original transaction data $\mathsf{tx}(B)$ to the validator.
   - Upon receiving $\mathsf{tx}(B)$, the validator encodes transaction data, computes the Merkle root h from the encoded data, and compares it to the Merkle root included in the block header.
   - If the computed root matches the one in the block header, the integrity of the transaction data is verified. The validator will participate in the cordial dissemination and include acknowledgments of availability for tx(B) in the next block headers.
2. *Reconstruction from Encoded Shards*:

[24] We note, that, it is possible that one validator v_i first broadcasts to v_j the block header of block B and after a while the block B including in its block body the encoded shard. In this case block B became a part of $\mathsf{hist}_i(j)$ when the first broadcast was called. Nevertheless, validators *always* are required to send their encoded shards once the transaction data becomes available to them.

- A block sent by a block creator is not reached by a validator; instead, the validator collects encoded shards. In this case the validator collects $f + 1$ encoded shards c_i, each accompanied by its Merkle proof p_i.
- For each shard c_i, the validator uses the proof p_i to verify its integrity against the Merkle root h from the block header of B. Only valid fragments are retained for reconstruction.
- Once $f + 1$ valid shards $d_1, \ldots, d_{f+1}$ are collected and arranged to the required positions, the codeword of the RS code is reconstructed using the decoding function:

$$\hat{c} = (\hat{c}_1, \ldots, \hat{c}_n) = \text{decRS}((d_1, d_2, \ldots, d_{f+1}), n, f + 1).$$

- The validator will also participate in the cordial dissemination of that block (even though the block header could have already been broadcasted) by including the corresponding encoded shard. The validator includes the acknowledgment of availability for tx(B) in the next block header.

Trade-offs Between Decoding Modes. As discussed above, Starfish supports both erasure and error-correcting decoding. Erasure decoding with Merkle proofs is the default due to its more computationally efficient implementation. We note that [12] made improvements over [8] in the communication complexity of RBC primitives by using RS error-correcting decoding instead of erasure decoding. While error-correcting decoding eliminates Merkle proof overhead of $O(\kappa \log n)$ per shard, the dominant communication cost in Starfish comes from block header metadata of size $O(\kappa n)$ per block due to ancestors and acknowledgments. This overhead is addressed by the compression techniques in Section 6 rather than by the choice of RS decoding mode. Additionally, error-correcting decoding incurs higher computational overhead: when shards arrive from multiple sources with potentially corrupted data, the decoder may need to attempt reconstruction up to f times with different shard subsets, resulting in $O(Mn^2 \log n)$ worst-case complexity compared to $O(M \log n)$ for erasure decoding.

D Algorithm description for Starfish

This section presents a selection of more detailed algorithms for Starfish. Let B be a block with the header

$$B_{\text{header}} = (r, \texttt{ancestors}, \texttt{acks}, \texttt{dataCommit})_{v_i}.$$

For simplicity of notation, we will write in algorithm descriptions $B.author$ for validator index i, $B.round$ for round number r, $B.ancestors$ for hash references of the parents `ancestors`, $B.acks$ for references to data commitments indicating transaction data availability `acks`.

Starfish uses several global variables shown at the beginning of Algorithm 1. This algorithm illustrates the overview and core mechanics of the Starfish protocol, detailing how validators execute rounds, create blocks, advance rounds and participate in encoded cordial dissemination. Conditions of these steps are also given in Table 3. Such an overview was not given in Mysticeti [2]. We also provide algorithms for consensus helper functions and algorithms for commit rule and sequencing transactions in Algorithm 2 and Algorithm 3. For better comparability with prior work, we highlight important changes in procedures and functions. When a function is similar to its counterpart in Mysticeti, the differences are marked in blue, while entirely Starfish-specific functions and procedures are highlighted in green.

E Safety Analysis

Starfish was largely inspired by prior uncertified DAG-based BFTs such as Cordial Miners and Mysticeti. This section contains proofs for totality, agreement, and total order, 3 out of 4 properties of BAB, see Sec-

Algorithm 1: Starfish Execution for Validator v_i

Global variables and constants:

$\delta_{TO} \leftarrow 2\Delta$ // Timeout duration; Δ is (known) upper bound for network delay after GST
$n \leftarrow 3f + 1$ // Number of validators; f is maximum number of Byzantine validators
$r_{decided} \leftarrow 0$ // Latest round of block in the sequence of decided leaders
$r_{highest} \leftarrow 0$ // Highest round of block in local DAG
$r_{last} \leftarrow 0$ // Round of latest own block
DAG $\leftarrow$ GENESISBLOCKS() // Local DAG of v_i initialized with predefined blocks of round 0

procedure EXECUTESTARFISH() ◇ *Execute Starfish protocol*
  for $r = 1, 2, \ldots$ **do**
    EXECUTEROUND(r)

procedure EXECUTEROUND(r) ◇ *Execute given round in Starfish*
  Step 1:
  BROADCASTUNKNOWNHISTORY() // Encoded Cordial Dissemination, Appendix C.4
  $\tau_{enter}(r) \leftarrow$ LOCALTIME() // Local time of validator v_i
  Step 2:
  while TRYCREATEBLOCK($r, \tau_{enter}(r)$) $=$`false` **do**
    RECEIVEBLOCKSFROMVALIDATORS() // Updates DAG, $r_{highest}$
    TRYDECIDE($r_{decided}, r_{highest}$) // Try commit leader blocks and sequence transactions
  Step 3:
  $r_{last} \leftarrow r$
  BROADCASTUNKNOWNHISTORY() // Encoded Cordial Dissemination, Appendix C.4
  Step 4:
  while TRYADVANCETONEXTROUND(r)=`false` **do**
    RECEIVEBLOCKSFROMVALIDATORS() // Updates DAG, $r_{highest}$
    TRYDECIDE($r_{decided}, r_{highest}$) // Try commit leader blocks and sequence transactions

function TRYADVANCETONEXTROUND(r) ◇ *Check conditions **A1** and **A2** in Table 3 to enter next round*
  if NUMBERVALIDATORWITHBLOCKSINROUND(r) $\geq 2f + 1$ ***and*** $r_{last} = r$ **then**
    return `true`
  return `false`

function TRYCREATEBLOCK($r, \tau_{enter}(r)$) ◇ *Check conditions **C1**, **C2** or **C3** in Table 3 to create new block*
  if LEADERCONDITIONSMET(r) ***or*** RECEIVEDBLOCKS(r) $\geq 2f + 1$ ***or*** LOCALTIME() $\geq \tau_{enter}(r) + \delta_{TO}$ **then**
    CREATEBLOCK(r)
    return `true`
  return `false`

function LEADERCONDITIONSMET(r) ◇ *Check for given round condition **C1** in Table 3*
  L1 $\leftarrow$ "leader block of round r is received" // Boolean variable true or false
  L2 $\leftarrow$ "received $2f + 1$ blocks voting for leader of round $r - 1$" **or** "exists skipping pattern for leader of round $r - 1$"
  return L1 & L2

tion 2. These proofs are based on similar considerations as in the prior work.

E.1 Safety of Starfish

Lemma 8. *If at a round k, $2f + 1$ blocks from distinct validators certify a leader block B from round $k - 2$, then all leader blocks at future rounds $> k$ will have a path to a certificate for B from round k.*

Proof. Consider any leader block B' at round $r > k$. The block B' has `ancestors` containing at least $2f + 1$ blocks from round $r - 1$. By quorum intersection with the $2f + 1$ certifiers at round k, there exists a

Algorithm 2: Helper Functions for Consensus

```
function GetPredefinedLeader(r)                    ◇ Return index of leader validator in proposer slot with given round
    return (r mod n) + 1

function IsDirectlySkippedSlot(r)                  ◇ Return true if leader slot is directly skipped
    leader ← GetPredefinedLeader(r)
    Votes ← DAG[r + 1]                                                  // All blocks in voting round
    Proposals ← {B ∈ DAG[r] s.t. B.author = leader}                     // All blocks by leader in proposer slot
    for L ∈ Proposals do
        res ← |{B.author : B ∈ Votes s.t. IsVote(B, L) = false}|
        if res < 2f + 1 then
            return false                                                // Found a block by proposer with not at least 2f + 1 of blame
    if |Proposals| = 0 & |{B.author : B ∈ Votes}| < 2f + 1 then
        return false
    return true                                                         // All blocks by the proposer have enough blame

function DirectlyCommittedLeaderBlock(r)           ◇ Return directly committed leader block if any
    leader ← GetPredefinedLeader(r)
    Proposals ← {B ∈ DAG[r] s.t. B.author = leader}                     // All blocks by leader in proposer slot
    Certificates ← DAG[r + 2]                                           // All blocks in certifying round
    for L ∈ Proposals do
        res ← |{B.author : B ∈ Certificates s.t. IsCert(B, L) = true}|
        if res ≥ 2f + 1 then
            return L
    return Undecided(r)

function IsVote(B_vote, B_leader)                  ◇ Return true if one block votes for another
    RoundCondition ← "B_vote.round = B_leader.round + 1"
    VoteCondition ← "B_leader is a first hash reference of B_leader.author in B_vote.ancestors"
    return RoundCondition & VoteCondition

function IsCert(B_cert, B_leader)                  ◇ Return true if one block is a certificate for another
    Ancestors ← {B ∈ DAG s.t. hash(B) ∈ B_cert.ancestors}
    res ← |{B.author : B ∈ Ancestors s.t. IsVote(B, B_leader) = true}|
    return res ≥ 2f + 1

function IsLink(B_old, B_new)                      ◇ Return true if one block is reachable from another
    return ∃ sequence of blocks B_1, ..., B_k ∈ DAG, k ∈ ℕ, s.t.
      B_1 = B_old, B_k = B_new and
      ∀j ∈ [2, k] : hash(B_{j−1}) ∈ B_j.ancestors

function GetReachableBlocks(L)                     ◇ Return all blocks reachable from leader block
    return {B ∈ DAG s.t. IsLink(B, L) = true}

function IsCertifiedLink(B_anchor, B_leader)       ◇ Return true if block can reach certificate for another block
    r ← B_leader.round
    Certificates ← DAG[r + 2]                                           // All blocks in certifying round
    return ∃B ∈ Certificates s.t. IsCert(B, B_leader) = true & IsLink(B, B_anchor) = true

function IsAck(B_ack, B)                           ◇ Return true if block acks a tx data of past block
    return hash(B) ∈ B_ack.acks
```

validator v who created both a block in B'.ancestors and a certifying block at round k. Since validators reference their own previous blocks via self, B' has a path through v's block chain to the certificate for

Algorithm 3: Committer of Leader Blocks and Sequencer for Transaction Data

```
procedure TryDecide(r_decided, r_highest)        ◇ Update sequence of commit/skip decisions for leaders
    sequence ← [ ]
    for r ∈ [r_highest − 1 down to r_decided + 1] do
        status ← TryDirectDecide(r)
        if status = Undecided(r) then
            status ← TryIndirectDecide(r, sequence)
        sequence ← status||sequence
    for status ∈ sequence in ascending round-order do
        if status = Undecided(r) then
            break
        if status = Commit(L) then
            SequenceTransactionsFromDAC(L)
        r_decided ← r_decided + 1                                   // Update last decided round

function TryDirectDecide(r)        ◇ Return direct commit/skip decision or undecided state
    if IsDirectlySkippedSlot(r) = true then
        return Skip(r)
    L ← DirectlyCommittedLeaderBlock(r)
    if L ≠ Undecided(r) then
        return Commit(L)
    return Undecided(r)

function TryIndirectDecide(r, sequence)        ◇ Return indirect commit/skip decision or undecided state
    Anchors ← {s ∈ sequence s.t. s.round ≥ r + 3}
    for A ∈ Anchors in ascending round-order do
        if A = Undecided(r) then
            return Undecided(r)
        if A = Commit(B_anchor) then
            leader ← GetPredefinedLeader(r)
            Proposals ← {B ∈ DAG[r] s.t. B.author = leader}          // All blocks by leader in proposer slot
            for L ∈ Proposals do
                if IsCertifiedLink(B_anchor, L) then
                    return Commit(L)
            return Skip(r)
    return Undecided(r)

procedure SequenceTransactionsFromDAC(L)        ◇ Sequence tx data for blocks with L being their DAC
    Reachable ← GetReachableBlocks(L)
    for B ∈ Reachable do
        if IsDataAvailCert(L, B) then
            Sequence(B)                          // Sequence transaction data of block B if not sequenced already

function IsDataAvailCert(B_DAC, B_check)        ◇ Return true if one block is DAC for another
    Reachable ← GetReachableBlocks(B_DAC)
    res ← |{B.author : B ∈ Reachable s.t. IsAck(B, B_check) = true}|        // Number of unique acknowledgments
    return res ≥ 2f + 1
```

B. □

The proof for the following lemma is different from the one provided in [2] since we use a more general condition for direct skipping of a proposer slot, see Algorithm 2.

Lemma 9. *If an honest validator commits a leader block of round r, then no honest validator decides to directly skip the proposer slot of round r.*

Proof. If an honest validator v directly or indirectly commits a leader block L of round r, then there exists a set W_L of $2f + 1$ validators that vote in round $r + 1$ for L. The set W_L includes $f + 1$ honest validators. Toward a contradiction assume that another validator v' directly skipped the proposer slot of round r. That means at the time of skipping, v' had a local DAG such that for each (if any) leader block L' of round r there were $2f + 1$ non-voters for L'. We consider two cases:
1. There was at least one leader block L' in the local DAG of v'. This means within the $2f + 1$ validators, that non-vote for L', there was at least one honest validator from W_L. Therefore, $L' \neq L$. Moreover, the local DAG of v' has to contain additionally L because of this honest non-voter. This means that there should be $2f + 1$ non-voters for L which is not possible due to quorum intersection.
2. There was no leader block of round r in the local DAG of v' and the local DAG contained blocks from $2f + 1$ validators in the voting round $r + 1$. However, one of these $2f + 1$ validators is an honest validator from W_L, implying that L was a part of the local DAG. This leads to a contradiction with the assumption of point 2. □

Lemma 10. *If an honest validator directly commits a leader block of round r, then no honest validator will decide to skip the proposer slot of round r.*

Proof. For the sake of contradiction, assume that an honest validator v directly commits a leader block B of round r, and an honest validator v' skips the proposer slot of round r. By Lemma 9 v' can not skip directly. It remains to consider the case when the validator v' skips indirectly.

Let B' be a block which was an anchor, when v' decided to indirectly skip the proposer slot from round r. Then the block B' is from round $r' > r + 2$. The validator v directly committed B, therefore $2f + 1$ blocks from different validators certify B in round $r + 2$. By Lemma 8 there is a path from B' to certificate for B, which means that B shouldn't have been skipped. □

Lemma 1 (Unique certified leader)**.** *At most one leader block from a given leader slot can be certified.*

Proof. For contradiction's sake, assume that two block proposals for a slot gather a quorum of votes. The intersection of these quorums contains at least one honest validator that voted for two blocks for one slot, which is a contradiction. □

Corollary 1. *No two honest validators commit distinct leader blocks for the same slot.*

Lemma 11. *All honest validators have a consistent state for each proposer slot, i.e., if two validators have decided the state of the slot, then both either commit the same block or skip the slot.*

Proof. Assume that at some moment validators v_1 and v_2 have inconsistent states. Pick a slot from the latest round r, such that one validator commits the leader block in the corresponding proposer slot, and the other skips this slot. Without loss of generality, v_1 commits the leader block, and v_2 skips it. By Lemmas 9 and 10 both validators made their decisions indirectly. Let r_1 and r_2 be the rounds, from which the decisions were made by v_1 and v_2 correspondingly. Then the validator v_1 skips all slots in rounds $[r + 3, r_1)$ and commits in round r_1, and v_2 skips all slots in rounds $[r + 3, r_2)$ and commits in the round r_2. Since r was assumed to be the last round with different decisions for v_1 and v_2, we conclude that $r_1 = r_2$. The decision to indirectly commit or skip the leader block in round r depends only on the causal history of a committed leader block in round $r_1 = r_2$, but this committed leader block have to be the same for both v_1 and v_2 by Corollary 1. This leads to a contradiction. □

Corollary 2. *All honest validators have a consistent sequence of committed leader blocks, i.e. a sequence of committed leader blocks for one honest validator is a prefix of another or other way round.*

The following theorem is adopted for Starfish since to sequence transaction data validators use committed leader blocks and find for which blocks in the causal past they serve as DACs.

Theorem 1 (Totality, Agreement and Total Order)**.** *If one honest validator delivers the transaction data of block B then all other honest validators will also deliver the transaction data of block B. Moreover, transactions are executed in the same order for all honest validators.*

Proof. The decision to sequence the transaction data of block B after committing the leader block L depends on whether the causal history of block L contains acknowledgments from $2f + 1$ validators. This history of a given block is the same for all validators. By Corollary 2, the sequence of committed leader blocks is the same for all honest validators; thereby, the order of the transaction data will be also the same. In other words, all honest validators will deliver transactions in the same order.

We note that each validator delivers at most one block from a given round from a given validator since we remove the equivocating blocks, see Appendix C.3. □

E.2 Safety of Starfish-C

The safety properties established in the previous subsection extend directly to the compressed block format of Section 6. The key observation is that Lemma 8 applies only to leader blocks, and in the compressed format, leader blocks retain full `ancestors` arrays containing n hash references to the latest blocks from distinct validators (with at least $2f + 1$ from round $r - 1$), while non-leader blocks require to reference blocks from the same validators, forming a block chain. Thus the quorum intersection argument in the proof of Lemma 8 applies unchanged.

The remaining safety lemmas do not depend on the structure of non-leader `ancestors` fields: Lemmas 9 and 1 use quorum intersection on votes and certifiers respectively, Lemma 11 follows from the previous lemmas, and Theorem 1 follows from the consistent leader sequence.

F Liveness Analysis

In this section, we prove a liveness property, which completes the proof that Starfish is a BAB. In Starfish, we distinguish two liveness properties. The first concerns the liveness of the sequence of leader blocks, ensuring that leader blocks proposed by honest validators are committed after GST. This is formalized in the first results, where progress in consensus relies solely on block headers. The second property, the liveness of transaction data, Theorem 3, is induced by the liveness of committed leader blocks together with the guarantees provided by the DACs.

F.1 Liveness of Starfish

Let $r_{\max}$ be the largest round among honest validators at the moment of GST. The first property that we prove is about synchronization of honest validators after GST, which was not shown in prior work on uncertified DAGs.

Lemma 4 (Synchronicity after GST)**.** *All honest validators enter any round $r > r_{\max}$ within time Δ of each other. Moreover, all honest validators create their round-r blocks within time Δ of each other.*

Proof. Consider round r higher than any round achieved by honest validators at the moment of GST. Consider the first honest validator v_1 that enters round r. Say that it happened at the moment t_1. By condition **B2**, this validator sends the blocks it knows to all other validators. Then at the moment $t_1 + \Delta$ all other validators have enough blocks to advance to the next round from any round $r' < r$, due to **A2**. Condition

A1 is fulfilled since for any round $r'r$ they create their blocks since the condition **C3** is satisfied. Indeed, for any round $r' < r$ the validator v_1 has at least $2f + 1$ blocks, and for any other validator v these blocks (except those which were already known to v from v_1's point of view) have been sent to v at the moment t_1. So, at the moment $t_1 + \Delta$ every honest validator has created and disseminated its blocks for all rounds until $r - 1$ and advanced to round r. In other words, all honest validators will enter round r in time interval $[t_1, t_1 + \Delta]$.

Let v_2 be the first honest validator that creates a block B in round r and it happens at the moment t_2 (it could be that $v_2 \neq v_1$). The creation of this block could not have been triggered by condition **C3**, since v_2 is the first among honest validators and could have obtained at most f round-r blocks from Byzantine validators. If the creation of the block was triggered by timeout condition **C2**, then all other validators will also create their blocks during the time interval $[t_2, t_2 + \Delta]$. Indeed, the difference between their entry (to the round r) times is at most Δ, and the difference between the moments when their timeouts expire is also at most Δ. If some honest validator creates its block before its timeout expires, then this moment is still in the time interval $[t_2, t_2 + \Delta]$, since v_2 is the first one by definition. If the creation of the block was triggered by condition **C1**, then not later than at moment $t_2 + \Delta$ all other validators will receive the block from v_2 together with the unknown history. They will be able to create their own blocks since the condition **C1** will be satisfied for them. □

Lemma 12 (Timely delivery)**.** *For any round $r > r_{\max}$, all blocks of honest validators from round $r - 1$ are delivered to every honest validator before its round-r timeout expires.*

Proof. If GST happens at the moment t_{GST}, and the last block of honest validator from round $r - 1$ was created at moment t_{last}, then at the moment $t_{deliver} = \max(t_{GST}, t_{last}) + \Delta$ all blocks of honest validators from round $r - 1$ will be delivered to every validator v. Let t_v be a moment when the validator v enters round r. Since rr_{max}, this happens after GST, so $t_v \geq t_{GST}$. Also by Lemma 4, $t_v \geq t_{last} - \Delta$. Recall that the timeout $\delta_{TO} = 2\Delta$, see Table 3. The two latter inequalities imply $t_v + \delta_{TO} \geq \max(t_{GST}, t_{last}) + \Delta = t_{deliver}$. Hence, all blocks of honest validators from round $r - 1$ will be delivered to every honest validator v before the round-r timeout of v expires. □

Lemma 13 (Consistent leader reference)**.** *For any round $r > r_{\max}$, if all honest validators receive the leader block of round $r - 1$ before their round-r timeout expires, then all round-r blocks created by honest validators reference this leader block.*

Proof. If the round-r block creation for some honest validator v was triggered by the condition **C1** then the created block of round r by v references the leader block of round $r - 1$. If the block creation is instead triggered by the timeout condition **C2**, the validator is assumed to have already received the leader block before the timeout and will reference it.

It remains to explain the case when condition **C3** triggered the block creation for some validator v. For this case, we consider $f + 1$ honest validators, which are the fastest to create a block in round r. Denote the set of these $f + 1$ honest validators by W. For these validators the block creation couldn't have been triggered by condition **C3** since there are not enough round-r blocks for it. Therefore, their block creation was triggered by **C1** or **C2** and have referenced the leader block by the above arguments. In the case that the block creation for validator v was triggered by **C3**, i.e., v received $2f + 1$ round-r blocks. There is at least one block B from W among them, and this block references the leader of the previous round. The validator v received the round-$(r - 1)$ leader block (or block header) together with B and will reference it in its block. □

Lemmas 12 and 13 imply the following corollary.

Corollary 3. *Suppose an honest validator occupies the proposer slot of round $r-1$ such that $r > r_{\max}$. Then all round-r blocks of honest validators reference the leader block of honest validator from round $r-1$.*

Using similar arguments as in the proof of Lemma 13, one can prove the following statements about voting blocks.

Lemma 14. *Suppose that for some round r, each honest validator has received before its r-round timeout expires $2f+1$ blocks voting for a leader block of round $r-2$. Then all round-r blocks created by honest validators reference the $2f+1$ blocks voting for the leader block of round $r-2$.*

Proof. If the round-r block creation for some honest validator v was triggered by the condition **C1** then the block references the $2f+1$ blocks voting for the leader block of round $r-2$ by definition. If it was triggered by the timeout condition **C2** then validator v has, by assumption, already received $2f+1$ blocks voting for the leader block of round $r-2$ and will reference them.

It remains to explain the case when condition **C3** triggered the block creation. In this case, we consider $f+1$ honest validators, which are the fastest to create a block in round r. The block creation couldn't have been triggered by condition **C3** since there are not enough round-r blocks for it. Therefore, they have referenced $2f+1$ blocks voting for the leader block of round $r-2$ by the above arguments. Denote the set of these $f+1$ honest validators by W. The block creation for validator v was triggered by **C3**, i.e., v received $2f+1$ round-r blocks. There is at least one block B from W among them, and this block references $2f+1$ blocks voting for the leader block of round $r-2$. The validator v received them together with B and will reference them in its block. □

Corollary 3 and Lemmas 12 and 14 imply the following corollary:

Corollary 4. *Suppose an honest validator occupies the proposer slot of round $r-2$ such that $r > r_{\max}+1$. Then all round-r blocks of honest validators will be certificates for the leader block of the honest validator from round $r-2$.*

Lemma 5 (Honest leaders committed)**.** *Any leader block created by an honest validator in round $r \geq r_{\max}$ will be marked* TO-COMMIT *by the direct decision rule.*

Proof. Consider a leader block created by an honest validator v in round $r \geq r_{\max}$. By Corollary 4 all honest validators will certify this leader block in round $r+2$. These $\geq 2f+1$ certificates will be obtained by every validator, and every honest validator will mark the leader block of round r TO-COMMIT by the direct decision rule. □

The next two lemmas are analogues of Lemmas 11 and 12 in Mysticeti [2] with minor adjustments. We provide the proofs for completeness.

Lemma 15. *The round-robin leader schedule ensures that in any $n+2 = 3f+3$ consecutive rounds, there are three consecutive rounds in which their leaders are honest.*

Proof. There are $3f+1$ groups of three consecutive rounds. Due to the round-robin schedule, each of the honest validators must be a leader in exactly 3 of these groups. As there are at least $2f+1$ honest validators, due to the pigeonhole principle, one group must contain $\lceil \frac{3(2f+1)}{3f+1} \rceil = 3$ honest leaders. □

Lemma 6 (All leaders decided)**.** *After GST, any undecided leader block will eventually be decided, i.e., marked as* TO-COMMIT *or* TO-SKIP.

Proof. Consider an arbitrary undecided leader block from round r. By Lemma 15 there will be 3 consecutive rounds k, $k+1$, $k+2$ with honest leaders in the corresponding proposer slots for some k such that $\max(r, r_{\max}) \leq k \leq \max(r, r_{\max}) + n - 1$. The leader blocks in these 3 rounds will be marked TO-COMMIT by Lemma 5. Let us iterate over all undecided leader blocks before round k in the order of decreasing rounds and apply the indirect decision rule. For each undecided block, the corresponding anchor will be marked TO-COMMIT, so by the indirect decision rule the undecided block will be marked TO-COMMIT or TO-SKIP. □

Lemmas 5 and 6 imply the liveness of the consensus.

Theorem 2. *Any leader block created by an honest validator in round $r \geq r_{\max}$ will be committed.*

Lemma 16. *Transaction data of any block created by an honest validator will eventually get a data availability certificate that will be committed.*

Proof. A block B created by an honest validator at t_1 will be received by every honest validator at latest after $\max(t_{GST}, t_1) + \Delta$. Each honest validator will acknowledge the data availability of the transaction data in a next block. Let t_2 be a moment when a slowest honest validator includes the acknowledgment. Every leader block, created after $t_2 + \Delta$ will be a DAC for the transaction data of B. By Lemmas 5 and 6, such a leader block will be eventually decided and committed; thereby, the transaction data of block B will be sequenced at latest after deciding such a leader block. □

Lemma 17. *If the transaction data of a block has a data availability certificate in the view of one honest validator, then the transaction data of that block will be eventually available by every honest validator.*

Proof. Since validators in Starfish employ the push-based dissemination strategy, every other honest validator will recognize the same DAC in time Δ after the next broadcast unknown history event is triggered. At least $f + 1$ honest validator, contributing to the DAC, will share their encoded shards at the time when they make acknowledgments. The parameters of the $(n, f+1)$ Reed-Solomon code allow for the successful reconstruction of the original transaction data. □

Theorem 3 (Validity)**.** *Transaction data of every block by an honest validator will be eventually sequenced.*

Proof. Consider a block B created by an honest validator v. By Lemma 16, this block will get a DAC L that is decided and committed. When attempting to sequence the transaction data for this block, one needs to check data availability for all blocks that got DACs in the sequence of committed leader blocks ended by L. By Lemma 17, the transaction data for all such blocks will be eventually available by every honest validator. □

F.2 Liveness of Starfish-C

The liveness properties established above extend to the compressed block format of Section 6. Recall from Section 6 that Starfish-C uses modified pacemaker conditions **A1***, **B1***, **B2***, and **C2*** (see definitions in Section 6). Condition **A2** remains unchanged.

These modifications affect the synchronization bounds after GST.

Lemma 18 (Synchronicity after GST for Starfish-C)**.** *All honest validators enter any round $r > r_{\max}$ within a time interval of length 3Δ and create their blocks in round r within a time interval of length 3Δ.*

Proof. The proof follows the same structure as Lemma 4, with adjustments for the modified pacemaker conditions. Consider the first honest validator v_1 that enters round r at moment t_1. By condition **B2***, the unknown history broadcast is delayed by 2Δ, so the blocks (or a block containing the aggregated signature $\langle r-1 \rangle_{\text{agg}}$) are sent at $t_1 + 2\Delta$ and delivered to all validators by $t_1 + 3\Delta$. At this point, all other validators can satisfy **A1*** and advance to round r.

For block creation, consider the first honest validator v_2 that creates a block in round r at moment t_2. If block creation is triggered by timeout condition **C2***, then all other validators create their blocks within $[t_2, t_2 + 3\Delta]$, since the difference between round entry times is at most 3Δ. If triggered by condition **C1**, the block is sent immediately via **B1***, but the unknown history is delayed by 2Δ, so other validators receive the complete information by $t_2 + 3\Delta$ and can create their blocks. □

Lemma 12 (Timely delivery) holds for Starfish-C with the adjusted parameters. The proof relies on the inequality $t_v + \delta_{TO} \geq \max(t_{GST}, t_{last}) + \Delta$, where t_v is the round entry time and t_{last} is the creation time of the last honest block from the previous round. From Lemma 18, we have $t_v \geq t_{last} - 3\Delta$. With $\delta_{TO} = 4\Delta$ (per **C2***), we get $t_v + 4\Delta \geq t_{last} + \Delta$, so the inequality holds. The larger timeout compensates for the weaker synchronization bound.

The remaining liveness lemmas (Lemmas 13–17) and Theorems 2–3 hold for Starfish-C, as they depend on Timely delivery which is satisfied with the modified parameters.

F.3 Latency of Starfish after GST

The following latency analysis applies to Starfish. Now we discuss the situation when the actual transmission latency after GST is upper bounded by $\delta < \Delta$, where the upper limit Δ is known to the validators, and δ is unknown. At first we note that in this case Lemma 4 holds with upper limit δ instead of Δ. We rewrite it with the corresponding change.

Lemma 19. *All honest validators enter any round $r > r_{\max}$ within a time interval of length δ and create their blocks in round r within a time interval of length δ.*

Lemma 20. *Assume that the leaders of rounds $r-1$, r, $r+1$ are honest, and $r > r_{\max}$. Then round-r leader block will be marked* TO-COMMIT *by all honest validators not later than 4δ after the moment of the creation of this leader block.*

In case when all validators are honest and $r > r_{\max} + 2$ the round-r leader block will be committed not later than after 4δ.

Proof. Suppose a leader block was created by an honest validator at the moment t in round r. By Lemma 19 all other honest validators will create their round-r blocks not later than at $t + \delta$. At the moment $t + 2\delta$ all round-r blocks created by honest validators will be obtained by everyone. Every honest validator will be able to enter round $r + 1$ at the moment $t + 2\delta$. Every honest validator will create its block in round $r + 1$ at the moment $t + 2\delta$ or earlier. Indeed, at the moment $t + 2\delta$ every honest validator will have received the round-r leader block and all other round-r blocks from honest validators. Since the leader of round $r - 1$ is honest by Corollary 3 every round-r block from honest validator will reference the leader of the round $r - 1$. Hence, the condition **C1** will be satisfied at $t + 2\delta$, and every honest validator will create its block in round $r + 1$ at the moment $t + 2\delta$ or earlier.

All blocks from round $r + 1$ created by honest validators will be obtained by everyone at $t + 3\delta$ or earlier. So, at the moment $t + 3\delta$ or earlier all honest validators will enter the round $r + 2$. Since the leader of the round $r + 1$ is honest, the leader block of round $r + 1$ will be received at $t + 3\delta$ or earlier. All honest blocks from round $r + 1$ will reference the leader block from round r by Corollary 3. So, at the moment $t + 3\delta$ the

condition **C1** in round $r+2$ will be satisfied for all honest validators, so they create their blocks at $t+3\delta$ or earlier. By Corollary 4 these blocks will certify the leader of the round r.

Every honest validator will receive at least $2f+1$ certificates at the moment $t+4\delta$ or earlier and mark the round-r leader block TO-COMMIT.

The leader block can be marked TO-COMMIT but not committed if some of the previous leader blocks are UNDECIDED. If all validators are honest, then from Lemma 5, we conclude that the leaders of round $r-2$, $r-1$, r will be marked TO-COMMIT. Similarly to Lemma 6 all previous leaders will be decided after that, and round-r leader will be committed. □

Lemmas 21 to 24 serve as useful tools for estimating latencies in various scenarios.

Lemma 21. *After GST, the time interval between the creation of blocks by an honest validator v in round r and $r+1$ is*

1. *at most 2δ if the leaders of rounds r and $r-1$ are honest.*
2. *at most $2\delta + 2\Delta$ otherwise.*

Proof. If a block in round r was created at the moment t, then all other honest validators will also create their blocks not later than at $t+\delta$ by Lemma 19. Then, at the moment $t+2\delta$ or earlier, the validator v will receive round-r blocks from all honest validators. It is enough to enter round $r+1$. Then, at the moment $2\delta+2\Delta$, the timeout will expire, and the validator v will create a round-$(r+1)$ block if it was not created before.

If the leaders of round r and $r-1$ are honest, then condition **C1** is satisfied at the moment $t+2\delta$ by Corollary 3, and the block will be created at moment $t+2\delta$ or earlier. □

Lemma 22. *After GST, the time that an honest validator v spends in round r is*

1. *at most 3δ if the leaders of rounds $r-1$ and $r-2$ are honest.*
2. *at most $2\delta + 2\Delta$ otherwise.*

Proof. If an honest validator v enters round r at moment t, then all other honest validators broadcast their blocks from round $r-1$ and enter round r not later than at $t+\delta$ by Lemma 19. The validator v and all other validators obtain all blocks from honest validators not later than at $t+2\delta$. If leaders of round $r-1$ and $r-2$ are honest, then condition **C1** is satisfied, and all honest validators, including v, create their blocks at $t+2\delta$ or earlier. All validators receive round-r blocks from honest validators not later than at $t+3\delta$, and can advance to the next round.

If at least one of the leaders of rounds $r-1$ and $r-2$ is Byzantine, then the validator v may need to wait until its timeout expires. Anyway, at the moment $t+2\Delta$ or earlier, the validator v creates and broadcasts its round-r block. Applying Lemma 19, other validators will do the same at latest at $t+2\Delta+\delta$.

Therefore, at moment $t+2\Delta+2\delta$ or earlier, all honest validators receive round-r blocks from all honest validators and advance to round $r+1$.

□

The following statement is key for establishing bounds for the latency for a long enough range of rounds, e.g. average and worst-cast end-to-end latencies with 0 or f Byzantine nodes. It says that the DAG progresses in rounds for honest validators at pace at least one round per time δ when there is no Byzantine validators in

the schedule. Any Byzantine validator in the leader schedule can affect the DAG contruction by additional 2Δ or even 4Δ depending whether it creates its block in the respected round.

Lemma 23. *Let $t_{last,r}$ denote the moment when the last honestly created block[25] of round r was created. Then after GST, $t_{last,r+1} - t_{last,r}$ is upper bounded by*

1. *at most δ if the leaders of rounds r and $r-1$ are honest.*
2. *at most $\delta + 2\Delta$ if the leader of round r or $r-1$ is Byzantine.*
3. *at most δ if the leader of round $r-1$ is Byzantine and the leader of round r is honest, and the leader of round $r-1$ fails to create its block in round $r-1$.*

Proof. If the last round-r block from an honest validator was created at the moment t then at the moment $t+\delta$ all honest validators will obtain all round-r blocks from honest validators. It is enough to enter round $r+1$. If leaders of rounds r and $r-1$ are honest then the condition **C1** is satisfied, and they can create and broadcast their blocks from round $r+1$.

If at least one of the leaders of round r and $r-1$ is Byzantine, then they may need to wait their timeouts. In that case all honest validators create their blocks from round $r+1$ not later than at $t+\delta+2\Delta$.

If the leader of round $r-1$ is Byzantine and *failed* to create its block and leader of round r is honest, then round-r blocks from honest validators create a skip pattern for the proposer round $r-1$ and the condition **C1** is satisfied at $t+\delta$. □

Lemma 24. *If all validators are honest, then after GST, the time interval between the creation of two (consecutive) leaders' blocks is at most 2δ.*

Proof. If the leader block of round r was created at the moment t, then at the moment $t+\delta$ all other blocks from round r are created, and at moment $t+2\delta$ they are received by every validator. Then the leader block for round $r+1$ will be created not later than at $t+2\delta$ since the condition **C1** is satisfied. □

Theorem 4 (Worst-case latency, all honest validators)**.** *Suppose all validators are honest. After GST, the worst-case consensus latency is at most 9δ. The worst-case end-to-end latency is at most 11δ.*

Proof. Assume that a given transaction was included in a block B created at the moment t. Then, at the moment $t+\delta$, all validators will receive the block B. By Lemma 21, at the moment, not later than $t+3\delta$, all validators will create their blocks with acknowledgement for the block B. At the moment, not later than $t+4\delta$, all validators will have in their local DAGs $\geq 2f+1$ acknowledgements for the block B. The next leader block L with $\geq 2f+1$ acknowledgements for B in its history will be created in some round, say r. By Lemma 21, all blocks from round r will be created not later than at $t+6\delta$. By Lemma 23 the block of a slowest validator from round $r+1$ and $r+2$ will be created not later than at $t+7\delta$ and $t+8\delta$. All honestly created blocks from round $r+2$ will certify the block L, and these blocks will be obtained by all validators not later than at $t+9\delta$. So, the leader block L will be committed not later than at $t+9\delta$, and the transaction data of block B will be sequenced.

To compute end-to-end latency, we must add the time the transaction waits to be included in the block, which is at most 2δ by Lemma 21. □

Theorem 5 (Average latency, all honest validators)**.** *Suppose all validators are honest. After GST, the average consensus latency is at most 7δ. The average end-to-end latency is at most 7.5δ.*

[25] Block created by a slowest honest validator

Proof. We divide the end-to-end latency into three subintervals I1, I2 and I3. I1 is the time interval from when a transaction gets available to a validator until it gets included in a block by the validator. I2 is the time interval for a block B from the time of its creation till the time creation of last round-r block such that all validators at or before round r acknowledged the availability of the transaction data of block B. The last interval I3 is from the end of I2 till a leader block serving as a DAC is committed by every honest validator.

By Lemma 23, the slowest validator creates blocks with a delay between these blocks at most δ. Since transactions are arriving continuously in time, the average time for interval I1 can be upper bounded by 0.5δ.

Let $t_{first,r}$ denote the moment when a first round-r block is created or the time creation of the block of a (round-r) fastest honest validator (since all validators are assumed to be honest in this statement). All blocks that are created at or before $t_{first,r}-\delta$ are guaranteed to receive a quorum of acknowledgments from round-r blocks. Moreover, by Lemma 19, the slowest validator creates round-r block at moment $\leq t_{first,r}+\delta$.

Now we are ready to upper bound the duration of I2 on average. By Lemma 23, one can show that the average value of $t_{first,r} - t_{first,r-1}$ is bounded by δ. Therefore, the duration of I2 is on average at most $\delta + (t_{first,r}+\delta) - (t_{first,r}-\delta) = 3\delta$.

Regarding I3, round-$(r+1)$ leader block L is guaranteed to be a DAC for the respected transaction data as all validators at or before round r made acknowledgment. All honestly created blocks at round $r+3$ will certify L. Thus, applying Lemma 23, I3 is bounded by the progress of slowest blocks in the DAG from round r to round $r+3$ and the delivery of round-$(r+3)$ blocks, i.e. 4δ.

To sum up, the average end-to-end latency is bounded by 7.5δ. □

Theorem 6 (Worst-case latency, f Byzantine validators)**.** *If $f < \frac{n}{3}$ validators are Byzantine, then after GST, the worst case end-to-end latency of any transaction sent to an honest validator is upper bounded by $4f\Delta + n\delta + O(\Delta)$. If Byzantine validators fail to create their blocks in rounds when they are leaders in the their proposer slots, then the latency can be estimated as $2f\Delta + n\delta + O(\Delta)$*

Proof. Assume that a given transaction is sent to an honest validator v at the moment t. We divide time interval until this transaction is sequenced by every validator into three subintervals I1, I2, I3. I1 is the time until there is a quorum of acknowledgments for this transaction in the local DAG of every honest validator. I2 is the time from the end of I1 until there are three consecutive leaders from honest validators. Finally, I3 is the time until the third leader in the sequence of the three consecutive leaders is committed ensuring that the transaction is sequenced.

First estimate the duration of I1. By Lemma 21, at moment $\leq t+2\delta+2\Delta$ the validator v creates the block B which includes the transaction. At moment $\leq t+3\delta+2\Delta$ all honest validators receive the block B. Applying again Lemma 21, at moment $\leq t+5\delta+4\Delta$ all honest validators create blocks with acknowledgement for B. At the moment $\leq t+6\delta+4\Delta$ all honest validators have in their local DAGs a quorum of acknowledgements for the transaction data of block B. Any honest leader committed after this moment will sequence the block B.

Second we bound the duration of I2. By Lemma 15 there will be 3 consecutive honest leaders in $n + O(1)$ rounds, which will be marked TO-COMMIT by Lemma 5. After this all previous leaders that were UNDECIDED will be marked TO-COMMIT or TO-SKIP. The time interval until three consecutive honest leaders is $n\delta + 4f\Delta + O(\Delta)$ by Lemma 23. Indeed, two honest leaders in the scheduler allow the slowest honest validator to create blocks with delay between them at most δ; one Byzantine leader among a window of two rounds can increase this to $\delta + 2\Delta$; if Byzantine leader does not create its block it can affect the latency in only one round, resulting in $\delta + 2\Delta$.

Finally it remains to bound the duration of I3. It takes up to $O(\Delta)$ to commit one leader among the three consecutive honest leaders.

Therefore, the total time to sequence the transaction can be bounded by $n\delta + 4f\Delta + O(\Delta)$ in the worst case if f Byzantine nodes are active and $n\delta + 2f\Delta + O(\Delta)$ if they fail to produce their blocks when they are leaders. □

Theorem 7 (Worst-case latency, 1 Byzantine validator)**.** *Suppose the number of validators is at least 6. If one validator is Byzantine, then after GST the worst-case end-to-end latency of a transaction sent to the honest validator is at most* $14\delta + 4\Delta$*. If the Byzantine validator fails to create its block, the latency is bounded by* $14\delta + 2\Delta$*.*

Proof. Assume a given transaction was received by honest validator v at moment t_1. At the moment t_1 the last block created by v was from round r. The validator v creates its block B for round $r+1$ which includes the given transaction at the moment t_2. Applying Lemma 21, if the Byzantine validator was a leader in proposer rounds r or $r-1$ then $t_2 \leq t_1 + 2\delta + 2\Delta$, otherwise $t_2 \leq t_1 + 2\delta$.

At the moment $t_3 = t_2 + \delta$ all honest validators receive the block B and add it to their local DAGs. All blocks created by honest validators that contain acknowledgement for B are from round $\geq r + 2$. Let $\hat{r}$ be a minimal round, such that all honest validators created blocks with acknowledgement for B in round $\hat{r}$ or earlier. Note that $\hat{r} \geq r + 2$. Consider a validator $\hat{v}$ that created its block with acknowledgement for B in round $\hat{r}$. At the moment t_3 it already has created the block for round $\hat{r} - 1$. By Lemma 19 all honest validators created their blocks in round $\hat{r} - 1$ at $t_3 + \delta$. Denote the moment of creation of last round $\hat{r}$ block by honest validator as t_4. By Lemma 23 $t_4 \leq t_3 + 2\delta$ if the leaders of rounds $\hat{r} - 1$ and $\hat{r} - 2$ are honest; if at least one of them is Byzantine, then $t_4 \leq t_3 + 2\delta + 2\Delta$.

At the moment $t_5 = t_4 + \delta$ all validators will receive from every honest validator a block with acknowledgement for the block B. Therefore, any committed leader block created by an honest validator after t_5 will be a DAC for block B. Say that the last leader block created before t_5 was from round $r' \geq \hat{r}$. Then by Lemma 19 the slowest honest validator will create his block in round r' at $t_6 \leq t_5 + \delta$.

We consider different cases depending on whether the Byzantine node is a leader at rounds $r' - 1$, r', $r' + 1$ or $r' + 2$:

1. The Byzantine node is not a leader at rounds $r' - 1$, r' or $r' + 1$, $r' + 2$. In this case, the blocks of honest validators from round $r' + 3$ will be certificates for the honest leader block of round $r' + 1$. Moreover, by arguments from Lemma 6, marking the three constitutive leader blocks as TO-COMMIT results in the sequencing the transaction data of block B. By Lemma 23, the slowest honest validator will create its block at round $r' + 3$ at moment not later than $t_6 + 3\delta$. After time δ, all honest validators will receive all honestly created round-$(r' + 3)$ blocks. All together the resulting end-to-end latency is at most $11\delta + 4\Delta$.

2. The Byzantine node is a leader at round $r' - 1$. We note that in this case waiting for the transaction to be included in block B takes at most 2δ (but not $2\delta + 2\Delta$) since $r' \geq \hat{r} \geq r + 2$. In the worst case, transactions from B will be sequenced when all honestly created leader blocks from round r', $r' + 1$ and $r' + 2$ are marked as TO-COMMIT. This will happen after all honestly created blocks from round $r' + 4$ are delivered. By Lemma 23, the slowest honest validator will create its block at round $r' + 4$ at moment not later than $t_6 + 4\delta + 2\Delta$. After time δ, all honest validators will receive all honestly created round-$(r' + 4)$ blocks. All together the resulting end-to-end latency is at most $12\delta + 4\Delta$.

3. The Byzantine node is a leader at round r'. In this case $t_6 - t_1 \leq 7\delta$ since $r' \geq \hat{r} \geq r + 2$. In the worst case, transactions from B will be sequenced when all honestly created leader blocks from round $r' + 1$, $r' + 2$ and $r' + 3$ are marked as TO-COMMIT. This will happen after all honestly created blocks from round $r' + 5$ are delivered. By Lemma 23, the slowest honest validator will create its block at round $r' + 5$ at

moment not later than $t_6 + 5\delta + 4\Delta$. After time δ, all honest validators will receive all honestly created round-$(r' + 5)$ blocks. All together the resulting end-to-end latency is at most $13\delta + 4\Delta$.

4. The Byzantine node is a leader at round $r' + 1$. In this case $t_6 - t_1 \leq 7\delta$ since $r' \geq \hat{r} \geq r + 2$. In the worst case, transactions from B will be sequenced when all honestly created leader blocks from round $r' + 2$, $r' + 3$ and $r' + 4$ are marked as TO-COMMIT. This will happen after all honestly created blocks from round $r' + 6$ are delivered. By Lemma 23, the slowest honest validator will create its block at round $r' + 6$ at moment not later than $t_6 + 6\delta + 4\Delta$. After time δ, all honest validators will receive all honestly created round-$(r' + 6)$ blocks. All together the resulting end-to-end latency is at most $14\delta + 4\Delta$.

5. The Byzantine node is a leader at round $r' + 2$. In this case $t_6 - t_1 \leq 7\delta$ since $r' \geq \hat{r} \geq r + 2$. In the worst case, transactions from B will be sequenced when all honestly created leader blocks from round $r' - 1$, r' and $r' + 1$ are marked as TO-COMMIT. This will happen after all honestly created blocks from round $r' + 3$ are delivered. By Lemma 23, the slowest honest validator will create its block at round $r' + 3$ at moment not later than $t_6 + 3\delta + 2\Delta$. After time δ, all honest validators will receive all honestly created round-$(r' + 3)$ blocks. All together, the resulting end-to-end latency is at most $11\delta + 2\Delta$.

If the Byzantine validator fails to create its own blocks, then the timeout will affect the resulting latency at most once. □

F.4 Happy case latency for Starfish-C after GST

We now show that the latency lemmas from the previous subsection extend to Starfish-C in the "happy case" where all validators are honest and actual transmission latency is $\delta < \Delta$.

The key observation is that condition **B1*** broadcasts the validator's own block immediately—only the unknown history broadcast is delayed by 2Δ. When all validators are honest, every validator receives all n blocks from each round via the immediate own-block broadcasts. Consequently, no validator ever needs to receive unknown history from another validator, and the delayed broadcast mechanism is never triggered.

Lemma 25. *Suppose all validators are honest and actual message latency is at most $\delta < \Delta$. Then after an initial synchronization phase comprising at most n rounds after $r_{\max} + 1$, for each subsequent round r, all honest validators enter round r within a time interval of length δ and create their round-r blocks within a time interval of length δ.*

Proof. We show that synchronization occurs within n rounds after $r_{\max} + 1$.

Initial spread. By Lemma 18, all validators create their round-$(r_{\max} + 1)$ blocks within an interval of length 3Δ. Let X_r denote the block creation spread in round r, with $X_{r_{\max}+1} \leq 3\Delta$.

Spread bound. We show $X_{r+1} \leq \max(X_r, \delta)$. Suppose round-$r$ blocks are created in $[c, c + X_r]$, with the "slow" validator v_s creating at $c + X_r$.

For round $r + 1$ with leader $\ell \neq v_s$: The leader's block exists by time $c + X_r$. Fast validators (those who created near time c) enter by time $c + \delta$ (waiting for **A1***) and receive the leader's block by $c + \delta$, so they create by $c + \delta$. The slow validator v_s enters at $c + X_r$ (bottleneck is **A2**: its own round-r block). Since $X_r \geq \delta$ (otherwise we are at the floor), v_s has received the leader's block by entry time and creates at $c + X_r$. Thus $X_{r+1} \leq X_r$.

Forced synchronization. When the slow validator v_s becomes the round-$(r + 1)$ leader, all other validators must wait for its block to satisfy **L1**. Since v_s's block is created at $c + X_r$, all validators receive it by $c + X_r + \delta$ and create by then. The first validator creates no earlier than $c + X_r$ (when the leader block exists), giving spread $\leq \delta$. With round-robin leadership, each validator leads within n rounds.

Steady state. Once $X_r \leq \delta$, we show $X_{r+1} \leq \delta$. If round-r blocks are created in $[c, c+\delta]$: The slow validator v_s (creating at $c+\delta$) has **A2** satisfied at $c+\delta$. It receives the first $2f+1$ blocks by $c+\delta$ (since these exist by $c+\delta$ and v_s's own block is among them). So v_s enters by $c+\delta$. The leader's block is created by $c+\delta$ and delivered to v_s by entry time. Thus v_s creates at $c+\delta$. Fast validators create earlier. The spread is $(c+\delta) - c = \delta$. □

Lemma 26. *In the happy case, Lemma 21 holds for Starfish-C: the time interval between the creation of blocks by an honest validator in rounds r and $r+1$ is at most 2δ if the leaders of rounds r and $r-1$ are honest.*

Proof. The proof of Lemma 21 relies on: (i) Lemma 19 for synchronization, (ii) immediate broadcast of blocks, and (iii) condition **C1** being satisfied when leaders are honest. By Lemma 25, synchronization holds with bound δ. Since all validators are honest, every validator receives all n round-r blocks within 2δ of creating its own block, no unknown history is needed, and **C1** is satisfied at time $t + 2\delta$. The proof applies unchanged. □

Lemma 27. *In the happy case, Lemma 23 holds for Starfish-C: $t_{last,r+1} - t_{last,r} \leq \delta$ if the leaders of rounds r and $r-1$ are honest.*

Proof. The proof of Lemma 23 uses immediate broadcast of the last round-r block and condition **C1** satisfaction when leaders are honest. Since **B1*** broadcasts own blocks immediately and all validators receive all blocks when all are honest, the proof applies unchanged. □

Corollary 5. *In the happy case, the delayed unknown history broadcast in **B1*** and **B2*** sends no additional data, and Starfish-C achieves $O(\kappa n^2)$ communication complexity per round.*

Proof. By Lemma 26, validators create blocks at intervals of at most 2δ when leaders are honest (which holds when all are honest). Since $\delta < \Delta$, we have $2\delta < 2\Delta$. When the 2Δ timer in **B1*** or **B2*** fires, the validator checks whether the recipient needs the unknown history. In the happy case, by time 2Δ after a block is created, all validators have already received all n blocks from that round and subsequent rounds via the immediate own-block broadcasts of **B1***. Thus, the recipient's DAG already contains all previously unknown blocks, and no additional data is sent. Only own blocks are exchanged via immediate broadcast: n validators each send their $O(\kappa)$-sized block to $n-1$ peers, yielding $O(\kappa n^2)$ communication per round. □

F.5 Main Theorem

The latency analysis above, combined with the safety proofs in Appendix E, establishes the following main result.

Theorem 8. *Starfish and Starfish-C are Byzantine Atomic Broadcast protocols in the partial synchrony model. Let M denote the average total transaction data per round and κ the security parameter (hash/signature size).*

Communication complexity per round:

- Transaction data*: Both Starfish and Starfish-C achieve $O(Mn)$.*
- Metadata*: Starfish incurs $O(\kappa n^4)$. Starfish-C achieves $O(\kappa n^3)$ in the worst case and $O(\kappa n^2)$ when all validators are honest (happy case).*
- Amortized per transaction byte*: Starfish achieves $O(\kappa n^4/M + n)$, which is $O(n)$ for $M = \Omega(\kappa n^3)$. Starfish-C achieves $O(\kappa n^3/M + n)$ in the worst case ($O(n)$ for $M = \Omega(\kappa n^2)$) and $O(\kappa n^2/M + n)$ in the happy case ($O(n)$ for $M = \Omega(\kappa n)$).*

Average latency (all honest validators):

- Starfish*: average block latency* $\leq 7\delta$*, average end-to-end latency* $\leq 7.5\delta$.
- Starfish-C*: average block latency* $\leq 8\delta$*, average end-to-end latency* $\leq 8.5\delta$.

Worst-case latency (after GST): *The worst-case block and end-to-end latencies for Starfish depend on the number and type of Byzantine validators:*

	n **honest**	**1 failure**[1]	**1 Byzan.**[2]	f **failures**	f **Byzan.**
block lat.	9δ	$12\delta + 2\Delta$	$12\delta + 4\Delta$	$n\delta + 2f\Delta + O(\Delta)$	$n\delta + 4f\Delta + O(\Delta)$
e2e lat.	$+2\delta$	$+2\delta$	$+2\delta$	$+2\delta$	$+2\delta$

For Starfish-C, the timeout increases from 2Δ *to* 4Δ*, so the "1 failure" column becomes* $12\delta + 4\Delta$ *and the "*f *failures" column becomes* $n\delta + 4f\Delta + O(\Delta)$.

G Discussion of Table 1

The average block and transaction latencies are computed under the following *Assumption*:

- All validators are honest[26].
- All messages are delivered within time interval δ.
- Any block of round r references all blocks from round $r-1$.
- All validators are perfectly synchronized, i.e., they enter the round r at the same time.
- Transactions are coming to validators continuously at a uniform rate.

Recall that we include in this comparison certified DAG-based protocols such as Bullshark [37], Shoal [36], Shoal++ [1], Sailfish [33] and uncertified DAG-based protocols such as Cordial Miners [23], Mysticeti [2] and Starfish.

We provide the results for one leader per round, even though some protocols from the table allow to have multiple leaders per round. For example, Mysticeti protocol can support n leaders per round and under the above assumption this could improve the latency. However, without such strict and unrealistic assumption we didn't find quantitative results confirming that this could improve the latency in the Byzantine environment. On a practical side, the up-to-date production Mysticeti code [27] uses one leader per round.

We start the analysis with Cordial Miners. Note that the latencies are different for the leader blocks and non-leader blocks. In addition, the latencies are different for non-leader blocks with different round offsets. Specifically, the block latency for the leader block from round r is 3δ, for blocks from round $r-1$ it is 4δ, for blocks from round $r-2$ it is 5δ, for non-leader blocks from round $r-3$ it is 6δ. Average block latency is

$$\frac{3\delta + 4\delta \cdot n + 5\delta \cdot n + 6\delta \cdot (n-1)}{3n} = 5\delta - \frac{\delta}{n}.$$

We omit the terms of order $O(n^{-1})$ and write 5δ. Since blocks are created at pace one block at time δ and the transactions are coming continuously, we add 0.5δ for the average time for a transaction to be included in a block. For one leader failure, one needs to add 6Δ for the latency as 2Δ is the timeout of a validator before the creation of a block and one such leader at round r triggers waiting the timeout of all honest validators at rounds $r+1$, $r+2$ and $r+3$.

[1] A Byzantine failure is a Byzantine validator that fails to create its blocks

[2] For the results with 1 Byzantine validator, it is required $n \geq 6$.

[26] For the last column of the table, we consider one faulty validator

[27] `https://github.com/MystenLabs/sui/releases`, release: MAINNET-V1.42.2

Now we analyse Mysticeti with pipelined leaders and a single leader at round. Unlike Cordial Miners, the block latency for non-leader block is 4δ. Therefore, the average block latency is $\frac{3\delta+4\delta(n-1)}{n} = 4\delta - \delta/n$. Similarly, 0.5δ is added to the transaction latency. For one leader failure, we added 4Δ while the Mysticeti paper does not explicitly explain when the rounds are advanced and blocks are created. Nevetherless, it was mentioned that validators at round r are waiting for the leader block of round $r-1$ and votes of the leader of round $r-2$, thereby one failure can trigger waiting the timeout twice. We update the condition **L2** with including the skipped pattern[28] for leader of round $r-2$, which allows validators in Starfish to wait the timeout only at one round.

Let us turn to analysing the latency for Starfish. Unlike Mysticeti, one needs to form a DAC for transaction data of block B in a committed leader block in order to sequence transaction data of B. This requires one more round, resulting in 5δ block latency and 5.5δ transaction latency. Now assume that the leader of round r is a failure and the leaders of the next rounds are honest. Consider the non-leader block B of the honest validator from round r. Block B will be acknowledged in round $r+1$, leader block L of the round $r+2$ will see these acknowledgments in its history, block L will receive votes in round $r+3$, certificates in round $r+4$, and then it will be marked TO-COMMIT. The faulty leader will be marked TO-SKIP, since it will not receive any votes. So the time from the creation of the block B until the execution of its transactions is 5δ plus the time which was spent by honest validators on waiting. In the round $r+1$ honest validators waited for their timeout 2Δ, but in all other rounds, they did not need to wait. In round $r+2$ they see a skip pattern for the faulty leader, in other rounds the block creation depends only on the blocks from honest validators.

The values for Bullshark, Shoal and Sailfish are taken from the Sailfish paper [33], where a similar analysis was performed and leader and non-leader block latencies are computed. For Bullshark, a leader block is committed in time 8δ if one uses RS-based RBC from [12]. Non-leader blocks can incur extra 4δ or 8δ latency depending on the round offset, resulting in average $\frac{8\delta+12\delta n+16\delta(n-1)}{2n} = 14\delta + O(1/n)$. Since blocks are created in Bullshark with such RBC at pace one block per 4δ, it results in extra 2δ delay for the transaction latency. In Shoal, non-leader blocks can incur extra 4δ, which results in block latency 12δ and transaction latency 14δ. Depending on whether communication efficient RBC is used, Sailfish requires 3δ (5δ) to commit leader blocks and extra 2δ (4δ) for non-leader blocks. This results in average block latency 5δ (9δ) and average transaction latency 6δ (11δ).

Shoal++ achieves similar results as Sailfish, except two things. It was suggested to use multiple DAGs in parallel to improve the transaction latency (reducing waiting time for a transaction until included in a block). While k can be infinitely large, Shoal++ suggested to use a moderate value of $k = 3$, so the additional time for the transaction latency was computed in Shoal++ [1] as $1.5\delta/k = 0.5\delta$. In addition, the waiting time in case of a faulty validator is larger than the one in Sailfish due to the lack of non-vote certificates.

Then, we provide a column with amortized communication complexity. It is measured in the number of bytes we need to transmit over the network per 1 byte of transactions. The complexity is computed for the worst-case scenario with f Byzantine nodes and sufficiently large transaction data. Protocols on certified DAGs can achieve that by employing RS-based RBC from [12]l however, the resulting latencies are higher that provided by Starfish. Alternatively, for Sailfish and Shoal++, it is possible to use a faster RBC, but this leads to a quadratic communication complexity in the worst case. For the protocols with uncertified DAGs, such as Mysticeti and Cordial Miners, the latencies can be lower, however, the amortized communication complexity is quadratic. Our proposed protocol, Starfish, simultaneously has linear communication complexity and relatively low latencies. h

[28] A similar skipped pattern was proposed in Mysticeti, but was not integrated to the block creation logic

References

[1] Arun, B., Li, Z., Suri-Payer, F., Das, S., Spiegelman, A.: Shoal++: High throughput dag bft can be fast! In: Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2025)

[2] Babel, K., Chursin, A., Danezis, G., Kokoris-Kogias, L., Sonnino, A.: Mysticeti: Reaching the limits of latency with uncertified dags. In: Proc. of the 2025 the Network and Distributed System Security Symposium (NDSS) (2025)

[3] Baird, L.: The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep **34**, 9–11 (2016)

[4] Blackshear, S., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Li, X., Logan, M., Menon, A., Nowacki, T., Sonnino, A., et al.: Sui lutris: A blockchain combining broadcast and consensus. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (2024)

[5] Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In: International Workshop on Public Key Cryptography. pp. 31–46. Springer (2003)

[6] Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: International conference on the theory and application of cryptology and information security. pp. 514–532. Springer (2001)

[7] Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM (JACM) **32**(4), 824–840 (1985)

[8] Cachin, C., Tessaro, S.: Asynchronous verifiable information dispersal. In: 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05). pp. 191–201. IEEE (2005)

[9] Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: OsDI. vol. 99, pp. 173–186 (1999)

[10] Chursin, A.: Adelie: Detection and prevention of byzantine behaviour in dag-based consensus protocols (2024), https://arxiv.org/abs/2408.02000

[11] Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: a dag-based mempool and efficient bft consensus. In: Proceedings of the 17th European Conference on Computer Systems (EuroSys). pp. 34–50 (2022)

[12] Das, S., Xiang, Z., Ren, L.: Asynchronous data dissemination and its applications. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 2705–2721 (2021)

[13] Doidge, I., Ramesh, R., Shrestha, N., Tobkin, J.: Moonshot: Optimizing block period and commit latency in chain-based rotating leader bft. In: 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 470–482 (2024). https://doi.org/10.1109/DSN58291.2024.00052

[14] Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) **35**(2), 288–323 (1988)

[15] Gao, S.: A new algorithm for decoding Reed-Solomon codes. In: Communications, Information and Network Security, The Springer International Series in Engineering and Computer Science, vol. 712, pp. 55–68. Springer (2003)

[16] Gao, Y., Lu, Y., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. p. 1187–1201. CCS '22, Association for Computing Machinery, New York, NY, USA (2022)

[17] Gelashvili, R., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A., Xiang, Z.: Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In: Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers. p. 296–315. Springer-Verlag, Berlin, Heidelberg (2022)

[18] Giridharan, N., Suri-Payer, F., Abraham, I., Alvisi, L., Crooks, N.: Autobahn: Seamless high speed bft. In: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. pp. 1–23 (2024)

[19] Gągol, A., Leśniak, D., Straszak, D., Świętek, M.: Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In: Proceedings of the 1st ACM Conference on Advances in Financial Technologies (AFT). pp. 214–228 (2019)

[20] Golan Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M., Seredinschi, D.A., Tamir, O., Tomescu, A.: Sbft: A scalable and decentralized trust infrastructure. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 568–580 (2019). https://doi.org/10.1109/DSN.2019.00063

[21] IOTA Foundation: IOTA 2.0: Core Concepts - Consensus. https://wiki.iota.org/learn/protocols/iota2.0/core-concepts/consensus/introduction/ (Accessed 2023), accessed on October 29, 2023

[22] Keidar, I., Kokoris-Kogias, E., Naor, O., Spiegelman, A.: All you need is dag. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing (PODC). pp. 165–175 (2021)

[23] Keidar, I., Naor, O., Shapiro, E.: Cordial miners: A family of simple, efficient and self-contained consensus protocols for every eventuality. In: Proc. of the 37th International Symposium on Distributed Computing (DISC) (2023)

[24] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. p. 45–58. SOSP '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1294261.1294267

[25] Lin, S.J., Al-Naffouri, T.Y., Han, Y.S., Chung, W.H.: Novel polynomial basis with fast fourier transform and its application to reed–solomon erasure codes. IEEE Transactions on Information Theory **62**(11), 6284–6299 (2016)

[26] Malkhi, D., Nayak, K.: Hotstuff-2: Optimal two-phase responsive bft. Cryptology ePrint Archive (2023)

[27] Malkhi, D., Stathakopoulou, C., Yin, M.: Bbca-chain: One-message, low latency bft consensus on a dag. In: Proc. of the 2024 Financial Cryptography and Data Security (FC) (2024)

[28] Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of bft protocols. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. p. 31–42. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978399, https://doi.org/10.1145/2976749.2978399

[29] Qiu, L., Xiao, J., Shao, Z.: Mechanized safety and liveness proofs for the Mysticeti consensus protocol under the LiDO-DAG framework. In: Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP) (2026), to appear

[30] Raikwar, M., Polyanskii, N., Müller, S.: Sok: Dag-based consensus protocols. In: 2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 1–18. IEEE (2024)

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

[32] Shapiro, E.: Grassroots cryptocurrencies: A foundation for a grassroots digital economy. arXiv preprint arXiv **2202** (2022)

[33] Shrestha, N., Kate, A., Nayak, K.: Sailfish: Towards improving latency of dag-based bft. In: Proceedings of the 2025 IEEE Symposium on Security and Privacy (SP) (2025)

[34] Shrestha, N., Yu, Q., Kate, A., Losa, G., Nayak, K., Wang, X.: Optimistic, signature-free reliable broadcast and its applications. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS) (2025). https://doi.org/10.1145/3719027.3765220

[35] Singleton, R.: Maximum distance q-nary codes. IEEE Transactions on Information Theory **10**(2), 116–118 (1964)

[36] Spiegelman, A., Aurn, B., Gelashvili, R., Li, Z.: Shoal: Improving dag-bft latency and robustness. In: Proc. of the 2024 Financial Cryptography and Data Security (FC) (2024)

[37] Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: Dag bft protocols made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2705–2718 (2022)

[38] Tonkikh, A., Arun, B., Xiang, Z., Li, Z., Spiegelman, A.: Raptr: Prefix consensus for robust high-performance bft. arXiv preprint arXiv:2504.18649 (2025)

[39] Welch, L.R., Berlekamp, E.R.: Error correction for algebraic block codes. U.S. Patent 4,633,470 (1986)

[40] Yu, Q., Losa, G., Shrestha, N., Wang, X.: Angelfish: Consensus with optimal throughput and latency across the leader-dag spectrum. arXiv preprint arXiv:2509.15847 (2025)