Lifefin: Escaping Mempool Explosions in DAG-based BFT

Jianting Zhang, Purdue University
Sen Yang, Yale University
Alberto Sonnino, Mysten Labs/UCL
Sebastián Loza, UTEC
Aniket Kate, Purdue University/Supra Research

***Abstract*—Directed Acyclic Graph (DAG)-based Byzantine Fault-Tolerant (BFT) protocols have emerged as promising solutions for high-throughput blockchains. By decoupling data dissemination from transaction ordering and constructing a well-connected DAG in the mempool, these protocols enable zero-message ordering and implicit view changes. However, we identify a fundamental liveness vulnerability: an adversary can trigger *mempool explosions* to prevent transaction commitment, ultimately compromising the protocol's liveness.**

In response, this work presents Lifefin, a generic and self-stabilizing protocol designed to integrate seamlessly with existing DAG-based BFT protocols and circumvent such vulnerabilities. Lifefin leverages the Agreement on Common Subset (ACS) mechanism, allowing nodes to escape mempool explosions by committing transactions with bounded resource usage even in adverse conditions. As a result, Lifefin imposes (almost) zero overhead in typical cases while effectively eliminating liveness vulnerabilities.

To demonstrate the effectiveness of Lifefin, we integrate it into two state-of-the-art DAG-based BFT protocols, Sailfish and Mysticeti, resulting in two enhanced variants: Sailfish-Lifefin and Mysticeti-Lifefin. We implement these variants and compare them with the original Sailfish and Mysticeti systems. Our evaluation demonstrates that Lifefin achieves comparable transaction throughput while introducing only minimal additional latency to resist similar attacks.

I. Introduction

For several decades, the Byzantine Fault-Tolerant (BFT) state machine replication problem has been a prominent topic of study in the distributed systems literature. In this problem, a group of nodes executes a BFT protocol to establish a consistent order for transactions continuously submitted by clients [1]–[8]. This process typically involves a *data dissemination* task where transactions are propagated among nodes and an *ordering* task where nodes decide on an order for the propagated transactions.

BFT protocols have been widely adopted by several in-production blockchains [9]–[15] for achieving consensus on transaction ordering. Modern deployed BFT protocols [8], [16]–[19] typically operate in the partial synchrony model [20] and as they scale, they decouple data dissemination from ordering: nodes continuously propagate intact transactions in batches or blocks, while the ordering task is performed separately using lightweight metadata (that is, hashes of batches/blocks) [21]. This decoupling offers two key advantages. First, it improves bandwidth utilization, as nodes can disseminate data even during asynchrony periods, which are known to hinder ordering due to the famous FLP impossibility result [22]. Second, it enhances communication efficiency in the ordering task, as nodes only need to process lightweight metadata instead of full transactions.

Most prior decoupled BFT protocols [1], [8], [16], [17], [19], [23]–[29] rely on leader-driven ordering, where a designated leader node in each consensus instance proposes a transaction order and *explicitly* coordinates agreement among nodes. Although the decoupled design reduces communication overhead in ordering, the leader-driven approach exposes several practical limitations. First, leader nodes experience uneven communication overhead and become potential bottlenecks, since they need to exclusively disperse order proposals to all nodes. Second, if order proposals fail due to network delays or faulty leaders, these protocols require a costly view-change mechanism to elect new leaders and/or need to manage large backlogs [8].

DAG-based BFT protocols [21], [30]–[40] introduce an alternative decoupled design that eliminates the limitations of leader-driven approaches. Specifically, these protocols operate in rounds: in each round, every node proposes a vertex, including transactions and connections to vertices from the previous round. The proposed vertices and connections are then disseminated to form a directed acyclic graph (DAG), where connections serve as votes to the connected vertices. The ordering task is performed separately: nodes check whether some selected leader vertices meet the committing rules (i.e., have enough votes/connections in subsequent rounds), and if they do, nodes order and commit them in a deterministic order. Due to the encoded information in the DAG, once a leader vertex is committed, all its causal history can also be ordered and committed.

The key innovation of DAG-based BFT lies in the structured *DAG-based memory pool (mempool)*, which nodes maintain to track uncommitted vertices. This design offers two advantages. First, nodes can independently order and commit transactions by interpreting the DAG, where edges represent votes for leader vertices—a property known as *zero-message ordering*. Second, in the presence of faulty leaders, nodes avoid costly view-change procedures; a subsequent valid leader vertex can naturally supersede failed proposals through its causal history, enabling *implicit view-change* (§ II-A).

This paper demonstrates that existing partially synchronous DAG-based BFT protocols suffer from a fundamental liveness vulnerability that arises from a gap between theory and practice: their liveness guarantees rely on infinite resources in theory, while nodes have bounded resources in practice. This practical, resource-bounded model is realistic but often

overlooked in theoretical analyses. Specifically, nodes can exhaust their resources maintaining an ever-growing set of uncommitted vertices in the DAG-based mempool during periods of asynchrony. Once resource exhaustion occurs, nodes cannot generate vertices to support the latest leader, causing the protocol to stall and preventing transactions from being committed, even after the network returns to synchrony.

More specifically, in DAG-based BFT, nodes create new vertices without waiting for the commitment of earlier ones. However, before the Global Stabilization Time (GST), a leader vertex is not guaranteed to gather enough votes to be committed. As uncommitted vertices accumulate, nodes risk running out of resources while trying to maintain the mempool. Meanwhile, leader vertices require sufficient votes, manifested as new subsequent vertices, for successful ordering under the zero-message ordering mechanism. As a result, resource exhaustion blocks correct nodes from generating new vertices or collecting enough votes, ultimately halting the protocol's progress. We refer to this resource exhaustion issue as *mempool explosion* (§ III-A). Intuitively, once a mempool explosion occurs, the DAG-based BFT protocol can no longer guarantee liveness.

To exploit this liveness vulnerability, this paper introduces a class of *inflation attacks* (§ III-B). In an inflation attack, the adversary participates in the data dissemination process by proposing vertices in every round but deviates from the ordering process by refraining from voting for leader vertices. As new vertices are continually generated without being committed, correct nodes exhaust their resources maintaining the DAG-based mempool. We show that by preventing *even a single* correct node from participating in the protocol for a period of time, the adversary triggers a mempool explosion and ultimately compromises liveness. We note that, to make the attack succeed, we make *no new* assumptions on either the threat model (i.e., we still assume f out of $3f+1$ nodes are Byzantine and $2f+1$ nodes are *always* honest) or the standard partial synchrony network model (§ III).

In response to the liveness vulnerability and inflation attacks, we propose Lifefin[1], a generic and self-stabilizing protocol that can be seamlessly integrated into existing partially synchronous DAG-based BFT protocols to "save the lives" of these fish named protocols (§ IV). The key insight is that if nodes can complete the ordering task with finite resource usage, they can ensure enough resources are available for ordering, thereby avoiding mempool explosions. The core concept behind Lifefin is a single-shot agreement that uses a bounded set of recognized DAG messages to commit the backlogs in the DAG-based mempool. Specifically, Lifefin is built on the Agreement on Common Subset (ACS) paradigm, which allows n nodes with n inputs to agree on a common subset V (where $|V| \geq n-f$, with f being the number of faulty nodes). In Lifefin-enhanced DAG-based BFT protocols, nodes continue to efficiently disseminate, order, and commit vertices following the underlying DAG protocol. When nodes suspect an inflation attack or are near resource exhaustion, they autonomously switch to the ACS mechanism, committing vertices with bounded resources. Moreover, Lifefin carefully selects vertices from the ACS output to order and incorporates a smooth transition to prevent inconsistencies caused by asynchrony or the switch between the ACS mechanism and the underlying protocol. As a result, Lifefin introduces (almost) zero overhead in typical cases while effectively eliminating liveness vulnerabilities and inflation attacks.

We instantiate Lifefin in two representative DAG-based protocols: Sailfish [33] (§ V) and Mysticeti [37] (§ VI), and implement Lifefin on top of them. We conduct extensive experiments in geo-distributed environments across five regions to evaluate the performance of Lifefin. The results demonstrate that Lifefin achieves comparable throughput while introducing only a minor latency overhead to effectively resist the exploited inflation attacks.

We summarize our contributions as follows:

- We identify a fundamental liveness vulnerability (§ III-A) in DAG-based BFT protocols and exploit it with a family of attacks (§ III-B).
- We present Lifefin (§ IV), a generic and self-stabilizing protocol that can be integrated into existing DAG-based BFT protocols to fundamentally eliminate the liveness vulnerability and effectively resist the exploited attacks.
- We instantiate Lifefin in Sailfish [33] (§ V) and Mysticeti [37] (§ VI), along with rigorous security analyses. As Sailfish and Mysticeti represent the two categories of DAG-based BFT protocols—certified and uncertified—we emphasize that Lifefin is broadly applicable and can be integrated into any DAG-based protocol.
- We implement Lifefin on Sailfish and Mysticeti, resulting in two enhanced variants: Sailfish-Lifefin and Mysticeti-Lifefin. We evaluate these variants and compare them with the original Sailfish and Mysticeti protocols (§ VII), showing that Lifefin achieves comparable performance while offering effective recovery from the attacks.

II. Preliminaries

We make the following system and network assumptions.

Threat and system models. We consider a system with $n = 3f+1$ nodes $\{N_1, \cdots, N_n\}$ of which up to f nodes are Byzantine and can be corrupted by a static adversary. The Byzantine nodes can behave arbitrarily but are computationally bounded. The remaining $2f+1$ nodes are correct and *always* follow the protocol. Note that we assume the same threat model as the prior DAG-based protocols [21], [31], [33], [37], [41].

We use a public-key infrastructure and digital signatures for message authentication. We use $\langle m \rangle_i$ to denote a message m digitally signed by node N_i with its private key. We consider the bounded model [42], [43], where each correct node has bounded memory/resource. Under this model, nodes can only store limited unprocessed messages and cannot accept or create new messages once their resources are exhausted.

[1]Lifefin is a made-up compound of "life" and "fin" (i.e., fish fin), suggesting the lifeline for fishes.

Network model. We consider the partial synchrony model of Dwork et. al. [20], where there exists an unknown time called Global Stabilization Time (GST) and a known time bound Δ, such that any message sent by correct nodes after GST is guaranteed to arrive within time Δ.

This paper focuses on the Byzantine Fault-tolerant (BFT) state machine replication problem, where a group of nodes runs a BFT protocol to establish a consistent order for transactions that are continuously submitted by clients. A BFT protocol is designed to guarantee properties [44]:

- **Safety:** If a correct node delivers a transaction tx before tx', then no correct node delivers tx' without first delivering tx, i.e., all correct nodes have the same prefix ledger.
- **Liveness:** If a transaction tx is sent to all correct nodes, then all correct nodes will eventually deliver tx.

A. DAG-based BFT

The directed acyclic graph (DAG)-based BFT protocols are proposed to enhance performance and have been adopted by modern blockchains, such as Sui [9]. Based on the method used to disseminate DAG vertices, DAG-based protocols can be categorized into *certified* DAG [21], [31]–[33], [41] and *uncertified* DAG [36]–[38]. We defer detailed descriptions of these types of DAGs to § V-A and § VI-A and summarize their common features below.

Separating data dissemination and ordering. DAG-based BFT protocols decouple data dissemination from metadata ordering. Specifically, the DAG-based BFT processes in logical rounds. In each round r, every node creates and disseminates a vertex consisting of transactions and a quorum of connections to round $r-1$ vertices. The connected vertices then form a DAG and are maintained in the nodes' mempool until they are later ordered and committed.

The DAG-based mempool keeps growing along with nodes moving to new rounds, allowing them to perform the ordering task separately by interpreting the DAG. Specifically, the DAG-based BFT protocol designates leader nodes to propose *leader vertices*. For instance, in partially synchronous DAG-based BFT protocols, leader vertices are predefined, e.g., using a deterministic method to select leader nodes based on the round number. By interpreting the DAG, nodes search for leader vertices that were connected (i.e., voted for) by enough vertices in their subsequent rounds, and then order the satisfactory leader vertices along with their causal histories with a deterministic algorithm.

To achieve a secure separation between data dissemination and ordering, DAG-based BFT defines two core quorums:

- **Dissemination Quorum** q_d: The minimum number of correct nodes required for data dissemination to proceed.
- **Committing Quorum** q_c: The minimum number of correct nodes required for the ordering task to proceed.

Various quorum settings. In DAG-based BFT protocols, q_d is set to $2f+1$. Specifically, a node processing round r waits for q_d vertices from round r before advancing to round $r+1$, and a round $r+1$ vertex must connect to at least q_d vertices from round r. A leader vertex can be committed directly if at least q_c vertices from the future rounds are connected to it. q_c can be set differently. Earlier DAG-based BFT protocols [21], [30], [31] rely on reliable broadcast (RBC) [45] to generate certified vertices (i.e., each vertex includes a quorum of signatures) and require several voting rounds between consecutive leader rounds, such that q_c is set by $f+1$ loosely. A recent work, Sailfish [33], sets a larger $q_c = 2f+1$ to reduce confirmation latency (S V-A). Meanwhile, a class of uncertified DAG-based protocols [36], [37] forgoes RBC to achieve lower latency but enforces a stricter committing rule. Briefly, a leader vertex in round r is directly committed only when at least $2f+1$ vertices in round $r+2$ have $2f+1$ paths reaching it (§ VI-A). For these uncertified DAG-based protocols, $q_c = 2f+1$. The dissemination and committing quorums stipulate the formation of a DAG, enabling nodes to maintain a robust DAG-based mempool to securely order transactions with an efficient ordering process.

Two gains with one DAG-based mempool. Unlike previous decoupled BFT protocols, the DAG-based BFT protocol organizes uncommitted vertices as a DAG in the mempool, enabling two key features for the ordering task:

- **Zero-message ordering:** Nodes can order transactions in the formed DAG-based mempool without extra communication for ordering.
- **Implicit view-change:** There is no need for explicit view-change mechanisms for handling faulty leaders.

Once the DAG of vertices is formed, each node can order transactions *locally* by interpreting the DAG in the mempool, with connections serving as votes for leader vertices. This enables zero-message ordering. Additionally, unlike previous leader-driven BFT protocols that require extra communication rounds to handle faulty leaders, DAG-based BFT implements an *implicit* view-change mechanism. Specifically, since the relevant quorums q_d and q_c govern the formation of the DAG, uncommitted leader vertices can be indirectly committed by leader vertices in subsequent rounds. This eliminates the need for an explicit view change to select new leader vertices, even if the originally selected leaders cannot be committed due to network delays or Byzantine behaviors.

III. DAG-based BFT: Theory meets Practice

A. Observation

Mempool explosions. We identify an overlooked cost of achieving zero-message ordering and implicit view-change features in the DAG-based BFT: a node needs to maintain unlimited data in its DAG-based mempool during periods of asynchrony, potentially causing a *mempool explosion*. Specifically, nodes are allowed to create new vertices even when prior vertices remain uncommitted (cf. implicit view-change). However, there is no guarantee that leader vertices can receive enough votes for the commitment due to the nature of asynchrony. Thus, nodes may eventually exhaust their resources for maintaining continuously generated uncommitted vertices.

Liveness issue in practice. Based on this observation, we claim that existing partially synchronous DAG-based BFT

protocols cannot guarantee liveness when nodes operate with finite resources, i.e., under the bounded model.

To understand why liveness is compromised, consider that a correct node that exhausts its resources will be unable to create new vertices. Any correct node that runs out of resources will fail to fulfill both data dissemination and ordering tasks. As a result, any leader vertex in the latest round will not receive sufficient connections (i.e., votes) from q_c subsequent vertices and will remain uncommitted.

We emphasize that this liveness issue persists even after the network recovers to synchrony, as it arises from the failure of vertex creation rather than message delays. In the following section, we demonstrate how this liveness vulnerability can be exploited in two representative, partially synchronous DAG-based BFT protocols. We introduce a class of attacks, termed *inflation attacks*, designed to exhaust nodes' resources to the point where an exhausted correct node cannot participate in the protocol, even after GST. Our estimates show that these inflation attacks can compromise the protocol's liveness in as little as 11 minutes (see § III-B for more evaluation details).

It is important to note that this liveness issue cannot be trivially mitigated by nodes halting the data dissemination task to prevent the generation of unlimited data. This is because existing DAG-based BFT protocols rely on the continuous creation of new vertices to commit and order leader vertices and their causal history (cf. zero-message ordering). If a correct node stops creating new vertices, it inherently violates the protocol's liveness guarantee. To address this liveness issue at its core, this paper proposes Lifefin, a generic and self-stabilizing protocol that can be seamlessly integrated into existing DAG-based BFT protocols. We provide an overview of Lifefin in § IV and present two concrete instantiations of Lifefin on two representative DAG-based BFT protocols in § V and § VI, respectively.

Note that non-DAG decoupled BFT protocols, such as Hotstuff [1] and Autobahn [8], are not fundamentally vulnerable to the liveness issue, as they do not rely on continuously growing data for ordering. However, fully eliminating this issue and preventing mempool explosions still requires explicit resource management, which, to the best of our knowledge, remains unaddressed in the design and implementation of these protocols. We elaborate on this point in Section D.

B. Inflation Attack

Attack description The goal of an inflation attack is to cause a mempool explosion by exhausting nodes' resources before correct nodes can complete the ordering task. To achieve this, Byzantine nodes actively propagate DAG vertices from correct nodes in every round, enabling the data dissemination task to proceed. However, they simultaneously prevent correct nodes from completing the ordering task. As a result, correct nodes are forced to maintain ever-growing uncommitted DAG vertices in their mempools, eventually leading to resource exhaustion and mempool explosion.

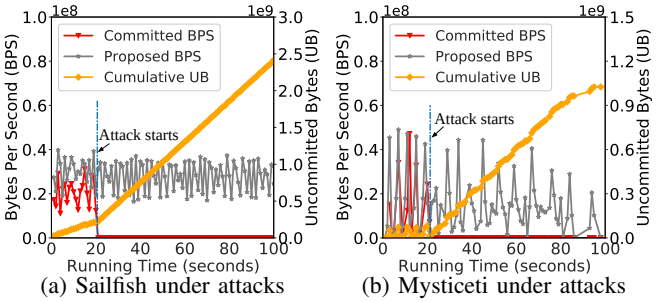
To make the inflation attack succeed, a straightforward approach is to execute a distributed denial-of-service (DDoS) attack to *prevent leader nodes from proposing leader vertices each round*. As a result, no leader vertices will be committed, and consequently, no vertices in the DAG can be committed under the protocol's rules. This attack will work across all partially synchronous DAG-based protocols, as their leader nodes are predetermined. However, continuously DDoSing the leader node in every round is costly, as it requires changing the attacked target (i.e., leader node) in each round.

In this paper, we propose and implement a more efficient inflation attack targeting state-of-the-art DAG-based BFT protocols, including Sailfish [33] and Mysticeti [37]. The attack allows the adversary to compromise the liveness by making *arbitrarily one correct node* temporarily unavailable. Specifically, recall from § II-A that these protocols use a committing quorum of $q_c = 2f + 1$. Under this setting, the adversary only needs to prevent a single correct node (not necessarily a leader) from participating in the ordering task—for example, by launching a temporary DDoS attack or inducing a misconfiguration to make it temporarily unavailable. Since the f Byzantine nodes do not vote for leader vertices, the remaining $2f$ correct nodes will never collect enough (i.e., $2f < q_c$) votes for committing leader vertices. Thus, the ordering task keeps failing, causing uncommitted DAG vertices to accumulate and eventually triggering a mempool explosion.

Note that this DDoS attack or misconfiguration is temporary and only used to trigger the mempool explosion. That is, it will stop once one correct node exhausts its resources. As an exhausted node cannot create new vertices to vote for leader vertices, the adversary can either repeatedly trigger mempool explosions or stall the protocol's progress to new rounds—both could ultimately violate liveness. We defer the concrete attack description to Appendix D. In addition, it is important to note that a correct node being temporarily DDoSed is not considered one of the f Byzantine nodes. This aligns with the partial synchrony model, where correct nodes may have delayed message delivery (due to DDoS) before GST.

Attack evaluation and estimation. We implement the proposed inflation attack on Sailfish and Mysticeti and evaluate its effectiveness. Specifically, we deploy $n = 10$ nodes, along with a set of clients submitting transactions at a rate of 50,000 tx/s to the nodes, where each transaction consists of 512 random bytes (see § VII-C for more experiment setups). We focus on the following metrics: (i) the committed bytes per second (BPS) that depicts transaction throughput; (ii) the proposed BPS that depicts the proposed vertices per second; (iii) the cumulative uncommitted byte (UB) that depicts the uncommitted vertices in the mempool. Figure 1 illustrates our evaluation results: the attack starts at 20s, after which the throughput (depicted by the committed BPS) drops to 0, while nodes continue proposing new DAG vertices (depicted by the proposed BPS), leading to sharply increasing uncommitted vertices (depicted by cumulative UB). This demonstrates it is possible to launch an inflation attack to exhaust nodes' resources and eventually cause mempool explosions.

With the data observed in our later experiments (§ VII), we estimate the time cost to cause a successful inflation



(a) Sailfish under attacks (b) Mysticeti under attacks

Fig. 1: DAG-based BFT protocols under inflation attacks: after the attack starts (at 20s), the committed BPS (indicating ordering) drops to 0 while the proposed BPS (indicating data dissemination) continues, causing a sharp increase of uncommitted data maintained in the DAG-based mempool.

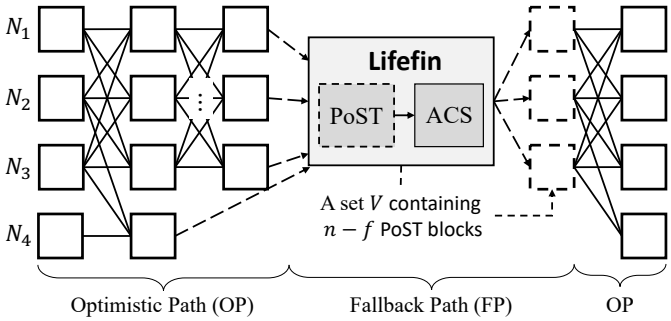


Fig. 2: The overview of Lifefin: In the optimistic path, nodes disseminate, commit, and order DAG vertices using the underlying DAG protocol. While suspecting inflation attacks or resource exhaustion, nodes switch to the fallback path and trigger the ACS protocol to commit and order DAG vertices and their causal histories with bounded resources.

attack. For example, Mysticeti can achieve over 400 KTps, indicating that a node needs to handle around 0.2GB of data per second. Assume a correct node uses 128GB of memory (which is suggested by the Sui blockchain [46]) to maintain its mempool, then it would cost the adversary approximately 11 minutes to exhaust a correct node's memory resource.

IV. The Lifefin Protocol

We now present Lifefin, a generic and self-stabilizing solution that offers liveness to DAG-based protocols, where *generic* means Lifefin can be seamlessly integrated into existing DAG-based protocols without significant changes, and *self-stabilizing* means nodes can autonomously handle adverse cases without external intervention once attacks occur.

A. Building Blocks

Agreement on common subset. Lifefin uses agreement on common subset (ACS) as a building block. In an ACS, each node N_i invokes $acs\_propose_i(m)$ to propagate its proposal m and outputs a common set of proposals V via $acs\_decide_i(V)$. An ACS protocol satisfies the following properties [44]:

- **Validity:** If a correct node N_i outputs $acs\_decide_i(V)$, then $|V| \geq n-f$ and V includes proposals from at least $n-2f$ correct nodes who invoke $acs\_propose$.
- **Agreement:** If a correct node N_i outputs $acs\_decide_i(V)$, then every other correct node N_j outputs $acs\_decide_j(V)$.
- **Termination:** If every correct node N_i calls $acs\_propose_i(m)$, then N_i outputs $acs\_decide_i(V)$.

B. System Overview

Lifefin's key insight is that, unlike DAG-based BFT requiring unlimited resources for ordering, single-shot agreement protocols (such as ACS realized using FIN [44]) operate with bounded resources. To elaborate, the single-shot agreement protocols do not rely on continuously growing data to output and commit proposals as they avoid zero-message ordering. Instead, nodes only take as input at most n proposals, each requiring a bounded resource. As a result, nodes can always preserve sufficient resources to complete the agreement process, thereby escaping mempool explosions.

Based on this observation, Lifefin incorporates an ACS protocol into the underlying DAG-based protocol. As the ACS protocol can decide on a set of proposals (i.e., vertices in DAG), nodes in Lifefin can commit these decided vertices with bounded resources. By carefully applying the underlying rules of the original DAG protocols to the committed vertices, Lifefin ensures that all correct nodes order historical vertices of the DAG consistently. Below we demonstrate the main components of Lifefin: *ACS-based fallback* and *Proof-of-STuck (PoST)*, shown in Figure 2.

ACS-based fallback. In a Lifefin-empowered DAG protocol, nodes primarily run the underlying DAG protocol in the optimistic path, but activate an ACS-based fallback mechanism to switch to the fallback path when necessary. Since the ACS mechanism is a single-shot agreement and only takes as input proposals from up to n nodes, it allows each node to reach an agreement with bounded resources. Specifically, both message and communication complexities are bounded, as also shown in our adopted ACS protocol [44, Section 5.2].

To elaborate, a correct node switches to the fallback path *locally* once its fallback conditions are triggered. There are two fallback conditions considered in the paper, and the node can configure the relevant parameters specifically: (i) the node's uncommitted data exceeds a predefined limit (e.g., 50GB), or (ii) the node receives a PoST block and fails to commit DAG vertices for a large timeout period (e.g., 1 hour). Upon entering the fallback path, each node stops processing DAG vertices from the underlying DAG-based protocol and $acs\_proposes$ its last vertex to participate in the ACS protocol. Once correct nodes collectively trigger the fallback mechanism, they wait to acs_decide a set V containing at least $n-f$ vertices. From the ACS properties, all correct nodes achieve a consistent output for V. With a deterministic rule (we will detail it in § V-B and § VI-B for different DAG protocols) to select a leader vertex from V, nodes eventually commit the leader vertex and its

causal history, completing the ordering task. After that, nodes return to the optimistic path, advancing to a new round to propose new vertices that reference the decided vertices in V. Since nodes can restart the optimistic path autonomously with the guaranteed termination of ACS, Lifefin is self-stabilizing.

Proof-of-STuck. The ACS protocol does not inherently ensure the validity of proposals: malicious nodes can $acs\_propose$ invalid vertices that contain nonexistent causal history or are from non-latest rounds. If an invalid leader vertex is selected from the ACS-based fallback mechanism, nodes may fail to retrieve the uncommitted history of vertices or risk committing vertices inconsistently.

To elaborate, in uncertified DAG-based BFT protocols [36]–[38], DAG vertices are not guaranteed to be available. If Byzantine nodes reference ancestors that are never shared with others, correct nodes will not be able to retrieve the missing ancestors of the ACS output, introducing new liveness issues.

Moreover, in both certified and uncertified DAG-based BFT protocols, if Byzantine nodes use non-latest round DAG vertices as their ACS proposals, the ACS outputs will contain a subset of old round DAG vertices. As a result, some correct nodes may have already committed previous leader vertices via the underlying DAG protocol, while others commit a newly selected leader vertex (from the ACS output) that indirectly skips committing them. To understand why this safety issue exists, take Bullshark [31] (which has been deployed in production) as an example, where a round r leader vertex v_L^r is directly committed with $f+1$ DAG vertices in round $r+1$ connecting to it. Assume these $f+1$ DAG vertices in round $r+1$ are created by a set of nodes $\mathcal{N}_1$ consisting f Byzantine nodes and one correct node N_c. The remaining $2f$ correct nodes (denoted by a set $\mathcal{N}_2$) have their round $r+1$ DAG vertices not connecting to v_L^r. In this case, if Byzantine nodes from $\mathcal{N}_1$ use their round r (not the latest round $r+1$) DAG vertices as ACS proposals, and correct nodes from $\mathcal{N}_2$ use their round $r+1$ DAG vertices as ACS proposals, then the ACS output V might consist of $2f+1$ DAG vertices that do not connect to v_L^r (i.e., N_c's round $r+1$ DAG vertex is excluded in V). As a result, correct nodes from $\mathcal{N}_2$ will skip committing v_L^r, leading to inconsistency.

A straightforward solution for the above issues is to validate ACS proposals (e.g., synchronize missing ancestors, check their freshness, and validate the corresponding DAG vertices) during the ACS protocol. However, this approach might require substantial modifications to the adopted ACS protocol, subject to the DAG structures.

To address these limitations, Lifefin introduces the *Proof-of-STuck (PoST)* block. At a high level, a PoST block encapsulates verifiable information about the DAG state observed by a node, indicating that the node is stuck committing vertices in its latest round. PoST blocks are constructed by a simple `Propose` and `Vote` scheme. Specifically, once switching to the fallback path, a node broadcasts a PoST block and collects signatures from a quorum of $2f+1$ nodes. These $2f+1$ signatures collectively form a certificate of validity, ensuring that malicious nodes cannot propose invalid information. The counter-signed PoST blocks are then used as the proposals of the ACS protocol.

By introducing PoST blocks, Lifefin decouples the ACS protocol from the underlying DAG-based protocol, making it easily integrated into any existing DAG-based protocol with any ACS protocol, regardless of the underlying DAG structure. Specifically, a PoST block contains a certificate that validates its authenticity. When performing the ACS, nodes simply verify each PoST proposal using its certificate rather than its history DAG structure, which may differ in different DAG-based protocols. In the following sections, we will elaborate on how to securely and seamlessly integrate Lifefin into two representative DAG-based protocols: Sailfish [33] (§ V) and Mysticeti [37] (§ VI).

Security intuition behind Lifefin. Lifefin enables nodes to output a set of PoST blocks V with bounded resources. Since nodes can always maintain enough resources to perform the ordering task through the ACS-based fallback mechanism, Lifefin-empowered DAG-based protocols can avoid mempool explosions and prevent the exploited inflation attacks. Regarding the liveness of the resulting protocol, since V contains at least $n-f \geq q_d$ vertices, nodes can return to the optimistic path by creating new vertices that reference the blocks in V. For the safety of the protocol, Lifefin carefully selects vertices from the ACS output for commitment and ensures a smooth transition to avoid different nodes committing inconsistent vertices. In brief, Lifefin may select a new leader vertex to replace the originally predefined one, but it always follows the underlying committing rules to order and commit DAG vertices. We present the following proposition, which is used for the sketch proof of safety, and defer its full proof to Appendix A and Appendix C.

Proposition 1. *In a Lifefin-empowered DAG-based BFT protocol, if r^* is the highest round number among the outputted PoST blocks V of a fallback instance, then the predefined round r^* leader vertex must not be committed before the fallback instance terminates.*

V. Sailfish-Lifefin

A. The Sailfish Protocol

Sailfish is a recent *certified* DAG-based protocol designed to optimize latency for early-stage DAG-based protocols [21], [31]. Figure 3 illustrates its overview. We first review its DAG construction and committing rules.

DAG construction. The DAG in Sailfish consists of regularly created vertices identified by the round number and the creator. In each round r, each node N_i contributes to the construction of the DAG using a reliable broadcast (RBC) [47], [48] to propagate a certified vertex v_i^r containing a list of transactions, $2f+1$ references (hash digests) to round $r-1$ vertices, and a quorum $2f+1$ of signatures.

Leader vertices. Each round r consists of a *predefined* leader vertex v_L^r. The predefined leader vertex v_L^r is created by a designated leader L_r, which is selected via a method based

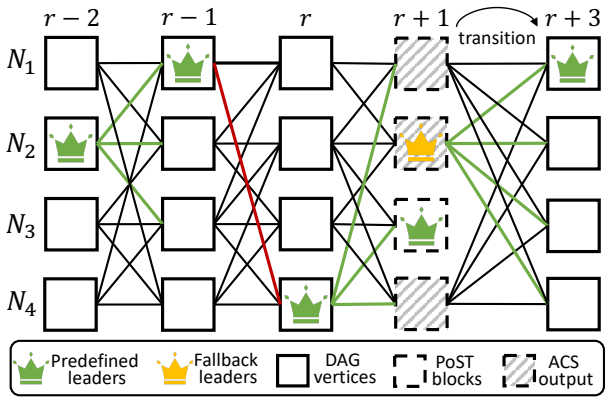


Fig. 3: Sailfish-Lifefin's round-based DAG from a node's local view: a leader vertex connected by green lines (—) is committed via the direct committing rule; a leader vertex connected by red lines (—) is committed via the indirect committing rule. All DAG vertices are disseminated via RBC.

on the round number r (e.g., round-robin). In Sailfish, the creation of leader vertices adheres to the following restriction:

Restriction 1. *When creating a new leader vertex v_L^{r+1}, L_{r+1} includes either the round r leader vertex v_L^r in its references or a $2f+1$ quorum of messages from other nodes indicating that they do not reference v_L^r in their round $r+1$ vertices.*

Committing rules. Driven by the leader vertices, each node employs the underlying committing rules to order and deliver DAG vertices (Figure 5). There are two committing rules used by the protocol: direct commit and indirect commit.

(1) Direct committing rule: A leader vertex v_L^r is directly committed by node N_i as soon as N_i observes $2f+1$ first messages of the RBC for round $r+1$ vertices with connections to v_L^r (Figure 5, Line 21-27). For instance, in Figure 3, round $r-2$ leader vertex v_L^{r-2} and round r leader vertex v_L^r are committed via the direct committing rule.

(2) Indirect committing rule: Due to the asynchrony and Byzantine behaviors, some leader vertices may not receive enough (i.e., $2f+1$) connections in time from the subsequent round and fail to be directly committed. Sailfish deploys an indirect committing rule to commit these failure leaders (Figure 5, Line 30-37). Specifically, once a round r leader vertex v_L^r is directly committed, N_i checks if there is a path between round r leader vertex and round $r-1$ leader vertex. If this is the case, round $r-1$ leader vertex is indirectly committed, and the mechanism is recursively restarted from round $r-1$ until it reaches a round $r'<r$ in which the round r' leader vertex $v_L^{r'}$ was previously directly committed. In Figure 3, round $r-1$ leader vertex v_L^{r-1} is indirectly committed through the directly committed v_L^r.

B. Securing Sailfish with Lifefin

Lifefin can be integrated into Sailfish to escape mempool explosions. During this process, nodes construct PoST blocks and perform an instance of ACS to commit the backlogs.

Variables:
$DAG_i[]$ — An array of sets of vertices (indexed by rounds)
$r_{fb} \leftarrow 0$ ▷ the round number decided by the last ACS instance, initialized by 0
$mode \leftarrow OP$ ▷ start with the optimistic path
struct PoST block SB
$SB.view$ - the view number of the fallback mechanism
$SB.vertex$ - the last vertex created by N_i
$SB.creator$ - the node that creates SB
$SB.cert$ - a quorum $2f+1$ signatures for SB
1: **procedure** create_new_vertex(r)
2: **if** $mode = FP$ **then**
3: **return** $\perp$
4: · · · ▷ return a round r DAG vertex if processing in the optimistic path
5: **upon** switch_fallback **do** ▷ trigger the fallback
6: **if** $mode = OP$ **then**
7: $mode \leftarrow FP$ ▷ switch to the fallback path
8: $r \leftarrow \max\{r \mid \exists v \in DAG_i[r] \wedge v.creator = N_i\}$
9: create_new_post(r)
10: **procedure** create_new_post(r) ▷ PoST `Propose`
11: $SB_i.view \leftarrow r_{fb}$ ▷ indicate the fallback view
12: $SB_i.vertex \leftarrow v$ s.t $v \in DAG_i[r] \wedge v.creator = N_i$
13: $SB_i.creator \leftarrow N_i$
14: multicast SB_i
15: **upon** receiving SB_j **do**
16: **if** is_post_valid(SB_j) **then** ▷ PoST `Vote`
17: send $\langle SB_j \rangle_i$ to N_j ▷ sign SB_j and send it back
18: **upon** receiving $2f+1$ $\langle SB_i \rangle_*$ **do**
19: $SB_i.cert \leftarrow$ a certification formed by $2f+1$ $\langle SB_i \rangle_*$
20: $acs\_propose_i(SB_i)$

Fig. 4: Sailfish-Lifefin: vertex creation for node N_i, where gray codes were implemented by Sailfish [33, Figure 2]

PoST construction (Figure 4, Line 5-20). The construction of PoST blocks is performed in an event-driven manner (Line 5), that is, node N_i switches to the fallback path to create its PoST block SB_i whenever one of the following conditions is satisfied during the protocol execution (§ IV-B): (i) its uncommitted data exceeds a predefined limit, or (ii) it receives a PoST block and fails to commit DAG vertices for a large timeout period T_{st}. During the fallback path, N_i no longer participates in the underlying DAG construction (Line 3).

N_i leverages a simple `Propose` and `Vote` scheme to construct SB_i (Line 10-17). Specifically, N_i propagates a SB_i containing a fallback view number (note that the protocol might trigger fallback multiple times, and the fallback view enables nodes to terminate fallback instances in sequence) and its last created vertex (representing where N_i is stuck in the DAG). Upon receiving SB_i, every other node N_j checks the validity of SB_i via the is_post_valid(SB_i) function (Line 16). In particular, is_post_valid(SB_i) checks whether N_j has the same fallback view, whether SB_i is the first PoST block from N_i within the view, and whether $SB_i.vertex$ is the most recently valid vertex seen in N_j's local view DAG_j or possesses a round larger than that of N_i's last vertex seen in DAG_j. If SB_i is valid, N_j acknowledges (i.e., certifies) it with a signature (Line 17).

Upon $2f+1$ nodes certifying SB_i, N_i initiates an ACS instance using SB_i as its proposal (Line 18-20). The $2f+1$ signatures of SB_i constitute a proof of validity for the ACS proposal, allowing nodes to forgo the verification of the validity of PoST blocks during the ACS instance. Thus, Lifefin can be implemented with any ACS protocol without the need for modifications to the ACS protocol itself.

```
Variables:
    currentRound ← 1; committedRound ← 0; leaderStack ← {}
    lastReferences— A set of parent vertices used to build a new block
    SB_L - the leader PoST block selected by the ACS instance
21: upon receiving a set S of ≥ 2f + 1 first messages for round r + 1 vertices do
22:     try_commit(r, S)
23: procedure try_commit(r, S)
24:     v ← get_leader_vertex(r)
25:     votes ← {v' ∈ S | path(v', v)}
26:     if votes ≥ 2f + 1 ∧ committedRound < r then
27:         commit_leader(v)                                   ▷ direct commit
28: procedure commit_leader(v)
29:     leaderStack.push(v)
30:     r ← v.round − 1
31:     v' ← v
32:     while r > committedRound do
33:         v_s ← get_leader_vertex(r)
34:         if path(v', v_s) then
35:             leaderStack.push(v_s)                          ▷ indirect commit
36:             v' ← v_s
37:         r ← r − 1
38:     committedRound ← v.round
39:     order_vertices()                  ▷ any deterministic ordering algorithm
40: upon acs_decide_i(V) do
41:     if ∀SB ∈ V, r_fb = SB.view then
42:         finalize_fallback(V)
43: procedure finalize_fallback(V)
44:     r_fb ← max{SB.vertex.round | SB ∈ V}
45:     if ∃SB' ∈ V : SB'.vertex.creator = L_{r_fb} then
46:         SB_L ← SB'
47:     else
48:         SB_L ← deterministically select a SB' ∈ V s.t. SB'.vertex.round = r_fb
49:         r ← r_fb − 1
50:         v_L ← ⊥                  ▷ the last round leader vertex chosen in OP
51:         if r > committedRound ∧ ∃v' ∈ ∪_{SB∈V}{SB.vertex.references} s.t. v'.creator = L_r then
52:             v_L ← v'
53:         if v_L ≠ ⊥ then
54:             commit_leader(v_L)                             ▷ direct commit
55:     L_{r_fb} ← SB_L.creator
56:     commit_leader(SB_L)                                    ▷ direct commit
57:     lastReferences ← V
58:     currentRound ← r_fb + 2                                ▷ graceful transition
59:     mode ← OP                                              ▷ switch to the optimistic path
```

Fig. 5: Sailfish-Lifefin: vertex commit for node N_i, where gray codes were implemented by Sailfish [33, Figure 3]

Commit DAG from ACS (Figure 5, Line **40-59**). The ACS instance will decide on a common set of PoST blocks V, based on which nodes consistently commit the pending DAG vertices (Line 40-42). At a high level, nodes determine a leader vertex SB_L in the updated round r_{fb} from V and commit its causal history using the underlying committing rules.

Obviously, there should be no conflicting order between SB_L and any previous round $r'<r_{fb}$ leader vertex that may have been committed using the underlying DAG mechanism ahead of the fallback mechanism. To achieve this, SB_L is selected from the PoST blocks in V with the *highest* round number. In the context where V contains a predefined round r_{fb} leader vertex, SB_L is set consistent with the predefined leader vertex (Line 45-46). However, there is no guarantee that V contains the predefined round r_{fb} leader vertex, given that $|V| \leq n$ and the leader node $L_{r_{fb}}$ could be malicious or stuck in a round $r'<r_{fb}$ (and therefore $L_{r_{fb}}$ does not create the round r_{fb} leader vertex). For example, in Figure 3, the ACS output V excludes the predefined round $r+1$ leader vertex v_3^{r+1} (where $r_{fb} = r+1$). In this case, SB_L is chosen with a deterministic algorithm (Line 48), e.g., based on digest.

However, such a newly selected leader vertex SB_L might have no path to the previous round $r_{fb}-1$ leader vertex since the creation of SB_L does not follow Restriction 1. This will lead to inconsistent commitments among nodes. For example, in Figure 3, round r leader vertex v_L^r can be directly committed by some nodes who received all $2f+1$ round $r+1$ vertices (i.e., v_1^{r+1}, v_3^{r+1}, and v_4^{r+1}) connecting it, while the other nodes who have not received v_3^{r+1} skip it through the indirect committing rule applying on SB_L. Consequently, nodes need to identify the round $r_{fb}-1$ predefined leader vertex $v_L^{r_{fb}-1}$ to which the blocks in V have a path (Line49-52).

After selecting SB_L, N_i commits $v_L^{r_{fb}-1}$ (if exists) and SB_L via the underlying commit_leader() function (Line 53-56). Then, N_i converts to the optimistic path and is configured to propose new vertices (Line 57-59). It is worth noting that we implement a graceful transition where nodes advance to $r_{fb}+2$ after committing blocks in the fallback path (Line 58). This can prevent nodes from creating equivocation round $r_{fb}+1$ vertices (some nodes might construct their PoST blocks in round $r_{fb}+1$ that are not included in the ACS output V).

Security proof. We give a proof sketch here and defer the formal security proof to Appendix A. For safety, we only need to consider the commitment of round r_{fb} leader vertex as Lifefin only might change the leader vertex in that round. In Sailfish-Lifefin, the fallback leader vertex SB_L has the highest round number r_{fb} among the ACS output V. According to Proposition 1, no correct nodes will commit the predefined round leader vertex $L_{r_{fb}}$ before the fallback instance terminates. Moreover, after the fallback instance terminates, all correct nodes will commit the unique SB_L in round r_{fb}. Consequently, all correct nodes will commit the same leader vertex in round r_{fb}. For liveness, the timeout T_{st} ensures that all correct nodes trigger the ACS mechanism if the protocol fails to commit new transactions after GST. By the ACS properties, every correct node eventually has $|V| \leq n-f$ referred vertices to move to a new round and propose new vertices, indicating that Sailfish-Lifefin can continuously process new transactions.

VI. Mysticeti-Lifefin

A. The Mysticeti Protocol

Mysticeti is an *uncertified* DAG-based protocol that replaces RBC with best-effort broadcast to disseminate DAG vertices, achieving better confirmation latency (Figure 6). We first review Mysticeti's DAG construction and committing rules.

DAG construction and leader vertices. The Mysticeti DAG is composed of vertices that are created in rounds. Specifically, for every round r, N_i propagates a vertex consisting of a list of transactions and $2f+1$ references to round $r-1$ vertices via best-effort broadcast; once receiving $2f+1$ round r vertices, N_i moves to round $r+1$ and propagates a new vertex. All leader vertices are predefined and pre-ordered based on their round numbers, where a leader node L_r is designated to propose a round r leader vertex v_L^r.

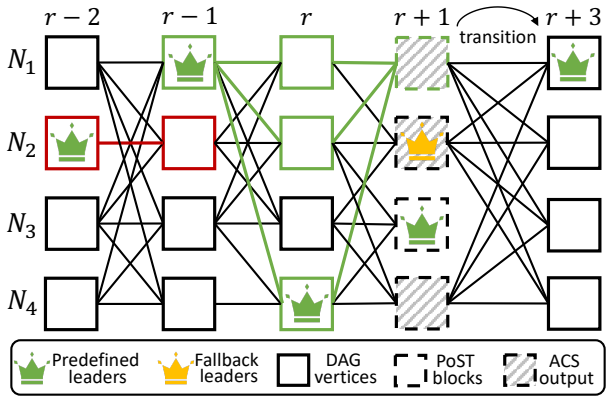


Fig. 6: Mysticeti-Lifefin's DAG from a node's view: The red pattern (□ and —) represents a skip pattern on v_2^{r-2}. The green pattern (□ and —) in round r represents a certificate pattern on v_1^{r-1}, and vertex v_1^{r+1} is called a certificate for v_1^{r-1}. All DAG vertices are disseminated via best-effort broadcast.

DAG patterns. Vertices in Mysticeti are not explicitly certified, that is, nodes do not need to acknowledge DAG vertices with their signatures during the DAG construction process. To prevent equivocation, Mysticeti implements an implicit certification, where nodes interpret DAG vertices with two patterns: (i) *skip pattern*, identifying a round r vertex that is not explicitly connected by at least $2f+1$ vertices of round $r+1$. For instance, the leader vertex v_2^{r-2} in Figure 6 is identified as a *skip pattern*; (ii) *certificate pattern*, identifying a round r vertex that is connected by at least $2f+1$ vertices from round $r+1$. A vertex is considered *certified* if a certificate pattern identifies it. Additionally, any subsequent vertex (in round $>r+1$) that includes such a pattern in its causal history is called a *certificate* for the vertex. For instance, the vertex v_1^{r-1} in Figure 6 is identified by a *certificate pattern* (from round r), and vertex v_1^{r+1} is a certificate for v_1^{r-1}.

Committing rules. Mysticeti commits leader vertices by identifying DAG patterns. While running the underlying DAG protocol, nodes check whether a DAG pattern can be applied to each leader vertex, deciding the leader vertices on two final statuses: to-commit or to-skip.

Figure 8 presents the core commit rules of Mysticeti. Specifically, every time N_i receives a new valid vertex, it invokes the try_decide() function to try to decide previous leader vertices (Line 25-37). Similarly, Mysticeti's committing rules consist of a direct decision rule and an indirect decision rule.

(1) Direct decision rule: N_i first tries to directly decide round r leader vertex v_L^r (Line 29), which leads to three results: (i) v_L^r is to-commit, if N_i observes $2f+1$ round $r+2$ vertices are certificates for v_L^r, that is, each round $r+2$ vertex references $\geq 2f+1$ round $r+1$ vertices that connect v_L^r. (ii) v_L^r is to-skip, if N_i observes a *skip pattern* over the vertex. That is at least $2f+1$ round $r+1$ vertices that do not reference v_L^r. (iii) otherwise, v_L^r is marked as undecided. For instance, in Figure 6, round $r-1$ leader vertex v_L^{r-1} is directly decided as to-commit since all round $r+1$ vertices are certificates for v_L^{r-1}.

(2) Indirect decision rule: For any leader vertex v_L^r marked as undecided via the direct decision rule, N_i applies the indirect decision rule over it (Line 31). Specifically, N_i searches for the first subsequent leader vertex $v_L^{r'}$ (where $r'>r+2$) that has been marked as either to-commit or undecided. Depending on the status of $v_L^{r'}$ and the relevant DAG pattern, v_L^r is marked as one of the statuses: (i) to-commit, if $v_L^{r'}$ is to-commit and causally references a *certificate pattern* over v_L^r, (ii) to-skip, if $v_L^{r'}$ is to-commit but does not reference a *certificate pattern* over v_L^r, and (iii) undecided if $v_L^{r'}$ is undecided.

Using these two decision rules, all correct nodes will consistently mark the predefined leader vertices as either to-commit or to-skip. By applying any deterministic ordering algorithm on the to-commit leader vertices, nodes eventually order all DAG vertices consistently.

Variables:
$DAG_i[]$ — An array of sets of vertices (indexed by rounds)
$r_{fb} \leftarrow 0$ ▷ the round number decided by the last ACS instance, initialized by 0
$mode \leftarrow OP$ ▷ start with the optimistic path
struct PoST block SB $\{\cdots\}$ ▷ same as Sailfish-Lifefin
$postBuffer \leftarrow \{\}$ ▷ PoST blocks missing history data

1: **procedure** create_new_vertex(r)
2: **if** $mode = FP$ **then**
3: **return** $\perp$
4: $\cdots$ ▷ return a round r DAG vertex if processing in the optimistic path
5: **upon** switch_fallback **do**
6: **if** $mode = OP$ **then**
7: $mode \leftarrow FP$
8: $r \leftarrow \max\{r \mid \exists B \in DAG_i[r] \wedge B.creator = N_i\}$
9: create_new_post(r)
10: **procedure** create_new_post(r) ▷ PoST `Propose`
11: $SB_i.view \leftarrow r_{fb}$ ▷ indicate the fallback view
12: $SB_i.vertex \leftarrow v$ s.t $v \in DAG_i[r] \wedge v.creator = N_i$
13: $SB_i.creator \leftarrow N_i$
14: multicast SB_i
15: **upon** receiving SB_j **do**
16: **if** is_post_valid(SB_j) **then** ▷ PoST `Vote`
17: **if** $(SB_j.vertex.round - currentRound \leq 1) \wedge$ miss_history(SB_j) **then**
18: $postBuffer \leftarrow postBuffer \cup \{SB_j\}$
19: synchronize_post(SB_j) ▷ sync missing blocks and then call back
20: **else**
21: send $\langle SB_j \rangle_i$ to N_j ▷ sign SB_j and send it back
22: **upon** receiving $2f+1$ $\langle SB_i \rangle_*$ **do**
23: $SB_i.cert \leftarrow$ a certification formed by $2f+1$ $\langle SB_i \rangle_*$
24: $acs\_propose_i(SB_i)$

Fig. 7: Mysticeti-Lifefin: vertex creation for node N_i, where gray codes were implemented by Mysticeti [37]

B. Securing Mysticeti with Lifefin

The integration of Lifefin with Mysticeti closely mirrors its integration with Sailfish, involving the construction of PoST blocks and an ACS instance.

PoST construction (Figure 7, Line **5-24)**. Upon the fallback condition satisfies (§ IV-B), node N_i creates a PoST block SB_i consisting of a fallback view and its last created vertex (Line 10-14). N_i waits for $2f+1$ nodes to counter-sign SB_i.

Recall that in Mysticeti, the DAG vertices are not explicitly certified. A malicious node may construct a PoST block with a phantom vertex whose history vertices are never shared with correct nodes. When N_i receives a PoST block SB_j,

```
Variables:
    currentRound ← 1; committedRound ← 0;
    lastReferences— A set of parent vertices used to build a new block
    SB_L - the leader block selected by the ACS instance
25: procedure try_decide()
26:     sequence ← []
27:     for r ∈ [currentRound down to committedRound + 1] do
28:         if ∃v ∈ DAG_i[r] : v.creator = L_r then
29:             status ← try_direct_decide(v)                    ▷ direct decide
30:             if ¬status.is_decided() then
31:                 status ← try_indirect_decide(v, sequence)    ▷ indirect decide
32:             sequence ← status||sequence
33:     decided ← []
34:     for status ∈ sequence do
35:         if ¬status.is_decided() then break
36:         decided ← decided||status
37:     finalize_decided_vertices(decided)
38: upon acs_decide_i(V) do
39:     if ∀SB ∈ V, r_fb = SB.view then
40:         finalize_fallback(V)
41: procedure finalize_fallback(V)
42:     r_fb ← max{SB.vertex.round | SB ∈ V}
43:     if ∃SB' ∈ V : SB'.vertex.creator = L_{r_fb} then
44:         SB_L ← SB'
45:     else
46:         SB_L ← deterministically select a SB' ∈ V s.t.
    SB'.vertex.round = r_fb
47:     L_{r_fb} ← SB_L.creator
48:     lastReferences ← V
49:     currentRound ← r_fb + 2                                  ▷ graceful transition
50:     try_decide()
51:     mode ← OP                                    ▷ switch to the optimistic path
```

Fig. 8: Mysticeti-Lifefin: vertex commit for N_i, where gray codes were implemented by Mysticeti [37, Algorithm 3]

there is no guarantee that it can retrieve the causal history of SB_j. To avoid the ACS instance outputting PoST blocks containing phantom vertices, in Mysticeti-Lifefin, nodes require an additional verification for PoST blocks (Line 17-19). In particular, if N_i receives a SB_j wrapping a vertex in a round equal to or smaller than N_i's next round, it must synchronize the causal history of SB_j before certifying it. Note that the resources used for synchronization are bounded, as N_i only needs to synchronize missing vertices within a limited range of round numbers (see Claim 20 for more details). This ensures N_i will not exhaust its resources after triggering the fallback mechanism, enabling N_i to terminate the fallback mechanism with bounded resources. As we prove in Claim 17 of Appendix B, such designs are sufficient to guarantee any PoST block certified by $2f + 1$ nodes retains its causal history with at least one correct node (i.e., ensuring data availability).

Commit DAG from ACS (Figure 8, Line **38-51)**. Committing vertices after the ACS instance in Mysticeti-Lifefin is straightforward. Nodes use the same rule (cf. § V-B) to select a leader vertex SB_L from the ACS output V, and SB_L replaces the predefined leader vertex from the same round. Subsequently, nodes follow the underlying committing rules to commit the pending vertices. In contrast to Sailfish-Lifefin, finalizing the fallback mechanism in Mysticeti-Lifefin eliminates the need to commit at least one predefined leader vertex after the ACS instance. This is because Mysticeti's underlying DAG protocol does not require a leader vertex to connect to its predecessor from the previous round. Consequently, changing leader vertices does not cause conflicting orders.

A proof for Mysticeti-Lifefin is similar to Sailfish-Lifefin, and we defer a formal security proof to Appendix B.

VII. Evaluation

In Lifefin, nodes continue to efficiently disseminate, order, and commit vertices following the underlying DAG protocol in typical cases, thereby maintaining the original protocol's performance. The fallback mechanism is only triggered under adverse conditions. Thus, our evaluation focuses solely on scenarios when fallback occurs and aims to answer the following questions:

- **Performance**: How does Lifefin perform in terms of throughput and latency once fallback occurs? (§ VII-A)
- **Scalability**: How well does Lifefin scale with the increasing number of nodes n? (§ VII-B)
- **Inflation Resistance**: Can Lifefin effectively resist inflation attacks? (§ VII-C)

Implementation details. We implement Lifefin in Rust, respectively integrating it into Sailfish [49] and Mysticeti [50] to derive Sailfish-Lifefin and Mysticeti-Lifefin. We adopt FIN [44] as our ACS protocol with RBC adopted by Narwhal [21] due to its practicality. We use Tokio [51] for networking and ed25519-dalek [52] for signatures. Code is available, but the link is omitted for blind review.

Baselines. We use Sailfish and Mysticeti as baselines since we aim to compare the performance difference of Lifefin under typical and adverse cases. In the following experiments, Sailfish and Mysticeti can represent the performance of Lifefin under the optimistic path, while Sailfish/Mysticeti-Lifefin reflect its performance when fallback occurs.

Optimization. During our initial evaluation, we observed that Mysticeti-Lifefin exhibited latency on the order of several seconds. We identified the cause as Mysticeti's original synchronization mechanism, where a node *randomly* fetches missing vertices, introducing unbounded latency. In our implementation, we eliminate this overhead by replacing it with a deterministic synchronization mechanism that fetches missing vertices from all nodes. This ensures correct nodes can receive missing vertices within one round-trip. Although this approach may introduce redundant messages, we emphasize it does not meaningfully affect efficiency as fallbacks are rare in practice.

Experimental setup. We evaluate all systems on AWS, using c5a.4xlarge EC2 instances spread across 5 regions (us-east-1, us-east-2, us-west-1, eu-west-1, and eu-west-2). Each instance provides 16 vCPU, 32GB RAM, and up to 10 Gbps of bandwidth and runs Linux Ubuntu server 20.04.

We deploy one client per node to send raw transactions, each of which is made up of 512 random bytes. To evaluate the performance of Lifefin under the adverse cases, in the following experiments, we set nodes to trigger the fallback mechanism to commit DAG vertices in a specific round. We focus on several performance metrics, including *throughput*, which represents the number of transactions committed per second (Tps), and *end-to-end (e2e) latency*, which is measured by the time from when clients send transactions to when nodes

commit the transactions. We use the default leader timeout set by the baselines, i.e., 5s in Sailfish and 1s in Mysticeti. We run each experiment for 240 seconds. In the following evaluation, we stop the transaction-workload experiments as soon as we observe a notable latency rise or verify that Lifefin introduces minimal overhead. We also allow clients to begin issuing transactions after a short setup period (40s), which may introduce pending latency to the e2e latency since transactions cannot be immediately packaged into vertices. We limit our experiments due to cost considerations.

A. Performance

We first evaluate the performance under different transaction workloads by running $n = 10$ nodes.

Crash-free performance. Figure 9b shows the throughput and e2e latency of all systems under different transaction workloads without crash faults. Sailfish-Lifefin and Mysticeti-Lifefin respectively achieve comparable throughput and latency as their vanilla Sailfish and Mysticeti, showing that Lifefin maintains decent performance. For instance, Sailfish-Lifefin achieves 245KTps with a 7.6s e2e latency while Sailfish achieves 258KTps with a 7.3s e2e latency. Moreover, in some cases, Sailfish-Lifefin achieves an even better latency than Sailfish. This is because the ACS-based fallback mechanism eliminates the timeout (i.e., leader timeout) used to commit predefined leader vertices, enabling nodes to commit leader vertices in realistic message delays.

Performance under crash faults. Figure 10 illustrates performance comparison under varying numbers of crash faults $f = 1$ and 3. All systems experience performance degradation as the number of crash faults increases. Sailfish-Lifefin and Mysticeti-Lifefin, respectively, have a similar trend to Sailfish and Mysticeti regarding performance degradation, showing matched resilience to crash faults.

Fallback performance. To evaluate the performance of the ACS-based fallback mechanism of Lifefin, we conduct experiments to measure the committing latency of the fallback leaders (Figure 11). Thanks to the certified DAG structure design deployed by Sailfish, Sailfish-Lifefin commits fallback leader vertices fast, slightly higher than committing predefined leader vertices. In contrast, in Mysticeti-Lifefin, committing fallback leader vertices introduces latency than committing predefined leader vertices. This overhead arises because its uncertified DAG structure requires nodes to synchronize missing vertices before they can finalize the fallback and propose new DAG vertices. Nevertheless, the latency overhead is acceptable and can be significantly amortized in practice, since nodes proceed in the optimistic path for most of the time while fallback occurs rarely. Figure 9 and Figure 10 prove such minimal overheads after the fallback-path latency is amortized by the optimistic-path latency.

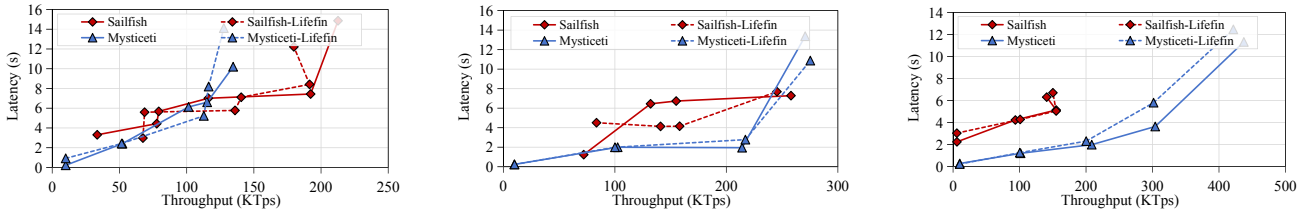
B. Scalability

We then evaluate Lifefin's scalability by running varying numbers of nodes $n = 4, 10$, and 20 (Figure 9), by following the same settings in Autobahn [8]. For both Sailfish and Mysticeti, Sailfish-Lifefin and Mysticeti-Lifefin match their throughput and latency under the same workload and n. For instance, under $n = 20$, both Sailfish and Sailfish-Lifefin achieve 155KTps with a 5.1s e2e latency; similarly, Mysticeti maintains a 304KTps with a 3.6s e2e latency while Mysticeti-Lifefin achieves 302KTps with a 5.8s e2e latency. Moreover, Mysticeti-Lifefin (and Mysticeti) can achieve a better throughput than Sailfish-Lifefin (and Sailfish) as the network scales, due to its elimination of certifying DAG, including removing RBC and certificate verification of DAG vertices. From Figure 9, we can show that incorporating Lifefin preserves the scalability of the underlying DAG-based protocol.

C. Inflation Resistance

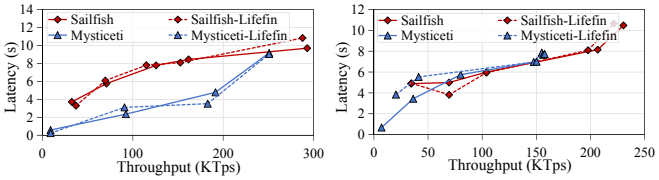
We finally evaluate whether Lifefin can efficiently resist inflation attacks. Due to the potential risks caused by attacking AWS, we chose to conduct these experiments on a local machine in a controlled way. The experiment settings are the same as those of Figure 1. Specifically, we perform the experiments on a Ubuntu server with 48 CPU cores and 128GB of RAM and set the transaction input rate relatively low ($50,000$ tx/s). We begin the attack by making one correct node unresponsive. Then, we let nodes trigger the fallback mechanism at round 80. We use a small round number to quickly trigger the fallback to avoid crashing our server in our experiments; in practice, the fallback triggering conditions rely on a large timeout (i.e., 1 hour) as discussed in § IV-B. Finally, we let the unresponsive correct node recover and become responsive 20 seconds after the fallback mechanism is triggered.

We run the protocols for approximately 100 seconds. Figure 12a and Figure 12b respectively illustrate the performance of Sailfish-Lifefin and Mysticeti-Lifefin under inflation attacks, where we focus on three metrics: committed BPS, proposed BPS, and cumulative UB. Specifically, committed BPS (the left y-axis) represents the committed data size (in committed vertices) per second, i.e., bytes throughput. Proposed BPS (the left y-axis) indicates the data size per second in vertices proposed by nodes, including the committed and uncommitted vertices. Cumulative UB (the right y-axis) represents the cumulative bytes that have been proposed but not committed by nodes. Experiments show that after the attack starts at around 20 seconds, the throughput (i.e., committed BPS) drops to 0 while nodes still propose new vertices (depicted by proposed BPS). During this period, the cumulative UB increases sharply as no vertices are committed. However, after the protocol triggers the fallback mechanism, nodes stop creating new DAG vertices, causing the cumulative UB to remain unchanged (i.e., during the 56s–76s interval in Figure 12a and the 34s-54s interval in Figure 12b). This shows our fallback mechanism can prevent the adversary from exhausting nodes' resources. After all correct nodes recover alive and become responsive, the fallback mechanism can be terminated, and the uncommitted data is handled. Compared to the original DAG-based protocols (cf. Figure 1), Lifefin-empowered DAG protocols



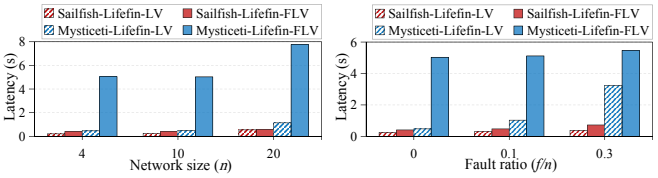
(a) $n = 4$ (b) $n = 10$ (c) $n = 20$

Fig. 9: Throughput vs. end-to-end latency under various network sizes (no crash failures), where Sailfish-Lifefin and Mysticeti-Lifefin operate in adverse cases with fallback triggering while Sailfish and Mysticeti operate in good cases



(a) $n = 10,\ f = 1$ (b) $n = 10,\ f = 3$

Fig. 10: Throughput vs. end-to-end latency under varying crash failures, where Lifefin operates in adverse cases



(a) Various n with $f = 0$ (b) Various fault f with $n = 10$

Fig. 11: Leader vertex latencies when fallback occurs: LV = predefined leader vertex, FLV = fallback leader vertex

effectively resist inflation attacks by preventing the adversary from exploiting mempool explosions.

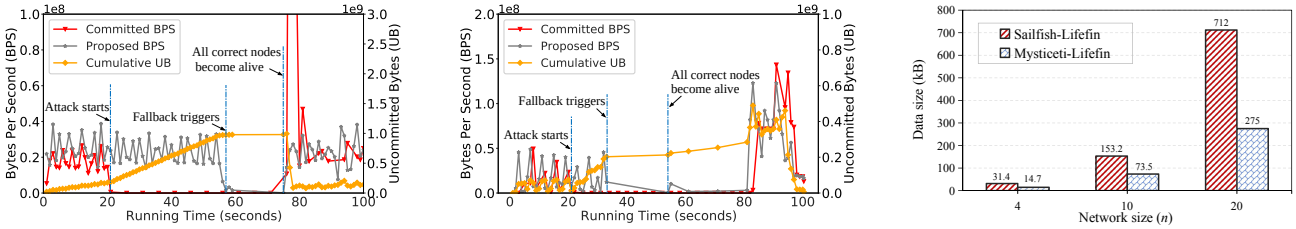
Figure 12c illustrates sizes of fallback data under varying n, where the fallback data mainly includes PoST blocks, some auxiliary data used to verify PoST blocks, and messages used for FIN protocol. This experiment demonstrates that Lifefin can commit DAG vertices with fixed and relatively small resources, enabling nodes to escape mempool explosions. For instance, in Sailfish-Lifefin, running $n = 20$ nodes requires <1MB of memory. Even running $n = 100$ nodes, given the quadratic storage complexity of the ACS mechanism, the overhead is expected to grow to a lightweight 18MB.

VIII. Related Work

Lifefin is designed to be generic and directly applicable to existing DAG-based BFT protocols [21], [30]–[34], [36]–[38], [41], [53]–[57] to circumvent mempool explosions. It applies to both certified and uncertified DAG-based protocols. Notable examples of certified DAG-based protocols include Bullshark [31], [58], Sailfish [33], and Shoal++ [41]. These protocols use reliable or consistent broadcast mechanisms to certify some or all DAG vertices [59]. Certification requires multiple message delays per DAG round but simplifies commit rules by ensuring that equivocating DAG vertices cannot occur. Notable examples of uncertified DAG protocols include Cordial Miners [36] and Mysticeti [37]. These protocols operate on an uncertified DAG, where each vertex represents a block disseminated on a best-effort basis to all peers [60]. Forgoing certification of vertices allows for reducing latency, bandwidth, and CPU requirements, but at the cost of a more complex commit rule and block synchronization module [37].

Asynchronous DAG-based BFT. The inflation attacks we exploit in this paper focus on partially synchronous DAG-based protocols (which predetermine leaders) due to their popularity [31]–[34], [36], [37], [41], [53]–[57]. As a result, asynchronous DAG-based BFT protocols (such as DAG-Rider [30], Tusk [21], and Mahi-mahi [38]), which select leaders randomly, are immune to our proposed specific attacks. However, the asynchronous DAG-based BFT protocols do not fundamentally address the liveness vulnerability since they still rely on continuously growing data to commit DAG vertices—the root cause of mempool explosions. Moreover, asynchronous DAG-based protocols cannot be integrated into the existing partially synchronous DAG-based protocols. In contrast, Lifefin is generic and can be seamlessly integrated into existing DAG-based protocols regardless of the network model assumption and data structure (certified and uncertified). In addition, they introduce randomization using a global perfect coin [30], [61]–[63] for every leader vertex. This approach significantly increases latency. For instance, DAG-Rider requires at least 12 message delays per block commitment; in contrast, state-of-the-art partially synchronous protocols commit blocks in just 3 message delays [37]. Lifefin minimizes this overhead by operating within a partially synchronous protocol and resorting to ACS protocols only in rare, adversarial scenarios.

Dual-mode protocols. The family of protocols most conceptually similar to Lifefin are dual-mode consensus protocols. These protocols, exemplified by Ditto [17] , Bullshark [31], Flexico [64], and Bolt-Dumbo [65], combine two consensus protocols: a partially-synchronous protocol optimized to reduce latency during periods of synchrony and a second protocol optimized for asynchronous conditions. Among these dual-mode protocols, only the dual-mode Bullshark [31] is designed for DAG structure, but its full implementation does



(a) Sailfish-Lifefin under attacks, $n = 10$ (b) Mysticeti-Lifefin under attacks, $n = 10$ (c) The fallback data size

Fig. 12: The resistance evaluation of Lifefin under inflation attacks. The evaluations without Lifefin are given in Figure 1

not exist (The Sui team implemented only the partially synchronous version of Bullshark [58], [66]). In addition, Bullshark's asynchronous mode still relies on unbounded DAG-based agreements and does not satisfy the bounded model. Moreover, Bullshark is designed for the certified DAG and cannot be applied to many uncertified DAG-based BFT protocols (including Mysticeti [37] that has been deployed in production).

Study on bounded models. The bounded model has been studied in distributed protocols. Prior works [42], [43], [67] prove the impossibility of solving sequential repeated reliable broadcast under the bounded model. Works [42], [68] circumvent the impossibility by assuming a (weaker) transient fault model or a (stronger) failure detector [69]. This paper shows that the bounded model also plays a role in the liveness guarantee of DAG-based BFT protocols due to the new features introduced by maintaining a DAG-based mempool. Prior theoretical analyses of distributed protocols often overlook such bounded-resource considerations, leaving substantial room to uncover similar vulnerabilities in future designs.

IX. Conclusion

In this work, we identify a fundamental liveness issue in existing DAG-based BFT protocols and show how it can be exploited by the adversary. In response, we propose a generic and self-stabilizing solution called Lifefin. Theoretical and experimental analyses show Lifefin can efficiently eliminate the liveness issue while only introducing acceptable latency.

Acknowledgment

The authors would like to thank Nibesh Shrestha for his insightful discussion on the Sailfish implementation. This work is supported in part by a research gift by Supra Labs.

References

[1] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *ACM PODC*, 2019, pp. 347–356.

[2] R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 35–48.

[3] S. Liu, W. Xu, C. Shan, X. Yan, T. Xu, B. Wang, L. Fan, F. Deng, Y. Yan, and H. Zhang, "Flexible advancement in asynchronous bft consensus," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 264–280.

[4] G. Sun, M. Jiang, X. Z. Khooi, Y. Li, and J. Li, "Neobft: Accelerating byzantine fault tolerance using authenticated in-network ordering," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 239–254.

[5] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, and M. Sadoghi, "Dissecting bft consensus: In trusted components we trust!" in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 521–539.

[6] F. Suri-Payer, M. Burke, Z. Wang, Y. Zhang, L. Alvisi, and N. Crooks, "Basil: Breaking up bft with acid (transactions)," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 1–17.

[7] W. Chen, D. Xia, Z. Cai, H.-N. Dai, J. Zhang, Z. Hong, J. Liang, and Z. Zheng, "Porygon: Scaling blockchain via 3d parallelism," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 1944–1957.

[8] N. Giridharan, F. Suri-Payer, I. Abraham, L. Alvisi, and N. Crooks, "Autobahn: Seamless high speed bft," *ACM SOSP*, 2024.

[9] T. S. Team, "The sui blockchain," https://sui.io/, accessed: 2024.

[10] T. A. Team, "Aptos networks," https://aptosfoundation.org/, accessed: 2024.

[11] T. S. Research, "The supra blockchain," https://supra.com/, accessed: 2025.

[12] T. M. Team, "The monad blockchain," https://www.monad.xyz/, accessed: 2025.

[13] T. C. Team, "Cosmos," https://cosmos.network/, accessed: 2025.

[14] T. I. team, "Iota is a decentralized blockchain infrastructure to build and secure our digital world," https://www.iota.org, 2025.

[15] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability," *arXiv preprint arXiv:1906.08936*, 2019.

[16] T. L. Team, "State machine replication in the libra blockchain," https://developers.diem.com/docs/technical-papers/state-machine-replication-paper/, 2020.

[17] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *FC*. Springer, 2022, pp. 296–315.

[18] I. Doidge, R. Ramesh, N. Shrestha, and J. Tobkin, "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)*. IEEE, 2024, pp. 470–482.

[19] M. M. Jalalzai and K. Babel, "Monadbft: Fast, responsive, fork-resistant streamlined consensus," *arXiv preprint arXiv:2502.20692*, 2025.

[20] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, apr 1988.

[21] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *ACM EuroSys*, 2022, pp. 34–50.

[22] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *JACM*, pp. 374–382, 1985.

[23] S. Duan, H. Zhang, X. Sui, B. Huang, C. Mu, G. Di, and X. Wang, "Dashing and star: Byzantine fault tolerance with weak certificates," in *ACM EuroSys*, 2024, pp. 250–264.

[24] J. Zhang, Z. Luo, R. Ramesh, and A. Kate, "Optimal sharding for scalable blockchains with deconstructed smr," *Proceedings of the VLDB Endowment*, 2025.

[25] D. Malkhi and K. Nayak, "Hotstuff-2: Optimal two-phase responsive bft," *Cryptology ePrint Archive*, 2023.

[26] D. Kang, S. Gupta, D. Malkhi, and M. Sadoghi, "Hotstuff-1: Linear consensus with one-phase speculation," *arXiv preprint arXiv:2408.04728*, 2024.

[27] B. Y. Chan and E. Shi, "Streamlet: Textbook streamlined blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 1–11.

[28] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai, "Fast-hotstuff: A fast and resilient hotstuff protocol," *arXiv preprint arXiv:2010.11454*, 2020.

[29] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, "State machine replication in the libra blockchain," *The Libra Assn., Tech. Rep*, vol. 7, 2019.

[30] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All You Need is DAG," in *PODC'21: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021.

[31] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *ACM CCS*, 2022, pp. 2705–2718.

[32] A. Spiegelman, B. Aurn, R. Gelashvili, and Z. Li, "Shoal: Improving dag-bft latency and robustness," in *International Conference on Financial Cryptography and Data Security*. Springer, 2024, p. 92–109.

[33] N. Shrestha, A. Kate, and K. Nayak, "Sailfish: Towards improving latency of dag-based bft," in *IEEE S&P*, 2025.

[34] D. Malkhi, C. Stathakopoulou, and M. Yin, "Bbca-chain: One-message, low latency bft consensus on a dag," *arXiv preprint arXiv:2310.06335*, 2023.

[35] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino, B. Williams, and L. Zhang, "Sui lutris: A blockchain combining broadcast and consensus," in *ACM CCS*, 2024.

[36] I. Keidar, O. Naor, O. Poupko, and E. Shapiro, "Cordial Miners: Fast and Efficient Consensus for Every Eventuality," in *37th International Symposium on Distributed Computing (DISC 2023)*, 2023.

[37] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian, "Mysticeti: Reaching the limits of latency with uncertified dags," in *Network and Distributed Systems Security Symposium (NDSS)*, 2025.

[38] P. Jovanovic, L. K. Kogias, B. Kumara, A. Sonnino, P. Tennage, and I. Zablotchi, "Mahi-mahi: Low-latency asynchronous bft dag-based consensus," *arXiv preprint arXiv:2410.08670*, 2024.

[39] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and M. Tian, "Obelia: Scaling dag-based blockchains to hundreds of validators," *arXiv preprint arXiv:2410.08701*, 2024.

[40] J. Zhang and A. Kate, "No fish is too big for flash boys! frontrunning on dag-based blockchains," *Cryptology ePrint Archive*, 2024.

[41] B. Arun, Z. Li, F. Suri-Payer, S. Das, and A. Spiegelman, "Shoal++: High throughput dag bft can be fast and robust!" in *Usenix NSDI*, 2025.

[42] C. Delporte-Gallet, S. Devismes, H. Fauconnier, F. Petit, and S. Toueg, "With finite memory consensus is easier than reliable broadcast," in *International Conference On Principles Of Distributed Systems*. Springer, 2008, pp. 41–57.

[43] D. Dolev and M. Spielrien, "Possibility and impossibility of reliable broadcast in the bounded model," *arXiv preprint arXiv:1611.05161*, 2016.

[44] S. Duan, X. Wang, and H. Zhang, "Fin: practical signature-free asynchronous common subset in constant time," in *ACM CCS*, 2023, pp. 815–829.

[45] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.

[46] Sui, "Sui validator node configuration," https://docs.sui.io/guides/operator/validator-config, 2025.

[47] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Good-case latency of byzantine broadcast: A complete categorization," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 331–341.

[48] S. Das, Z. Xiang, and L. Ren, "Asynchronous data dissemination and its applications," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2705–2721.

[49] Nibesh Shrestha, "Sailfish codebase," https://github.com/nibeshrestha/sailfish, 2025.

[50] Alberto Sonnino, "Mysticeti codebase," github.com/asonnino/mysticeti, 2025.

[51] Tokio, "Tokio - an asynchronous rust runtime," https://tokio.rs/, accessed: 2025.

[52] D. Cryptography, "Dalek elliptic curve cryptography," https://github.com/dalek-cryptography/ed25519-dalek, accessed: 2025.

[53] D. Malkhi and P. Szalachowski, "Maximal extractable value (mev) protection on a dag," *arXiv preprint arXiv:2208.00940*, 2022.

[54] X. Dai, Z. Zhang, J. Xiao, J. Yue, X. Xie, and H. Jin, "Gradeddag: An asynchronous dag-based bft consensus with lower latency," in *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2023, pp. 107–117.

[55] X. Dai, Z. Zhang, Z. Guo, C. Ding, J. Xiao, X. Xie, R. Hao, and H. Jin, "Wahoo: A dag-based bft consensus with low latency and low communication overhead," *IEEE Transactions on Information Forensics and Security*, 2024.

[56] X. Dai, G. Wang, J. Xiao, Z. Guo, R. Hao, X. Xie, and H. Jin, "Lightdag: A low-latency dag-based bft consensus through lightweight broadcast," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 998–1008.

[57] X. Dai, W. Li, G. Wang, J. Xiao, H. Chen, S. Li, A. Y. Zomaya, and H. Jin, "Remora: A low-latency dag-based bft through optimistic paths," *IEEE Transactions on Computers*, 2024.

[58] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: the partially synchronous version," arXiv preprint arXiv:2209.05633, 2022.

[59] M. Raikwar, N. Polyanskii, and S. Müller, "SoK: DAG-based Consensus Protocols," in *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2024, pp. 1–18.

[60] B. Ford, "Threshold Logical Clocks for Asynchronous Distributed Coordination and Consensus," *CoRR*, vol. abs/1907.07010, 2019.

[61] E. Blum, J. Katz, C.-D. Liu-Zhang, and J. Loss, "Asynchronous byzantine agreement with subquadratic communication," in *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18*. Springer, 2020, pp. 353–380.

[62] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, 2000, pp. 123–132.

[63] J. Loss and T. Moran, "Combining asynchronous and synchronous byzantine agreement: The best of both worlds," 2018.

[64] S. Ren, C. Lee, E. Kim, and S. Helal, "Flexico: An efficient dual-mode consensus protocol for blockchain networks," *PLoS ONE*, 2022.

[65] Y. Lu, Z. Lu, and Q. Tang, "Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT," in *CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[66] T. S. team, "Sui," https://github.com/mystenLabs/sui, 2024.

[67] A. Ricciardi, "Impossibility of (repeated) reliable broadcast," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, p. 342.

[68] S. Dolev, R. I. Kat, and E. M. Schiller, "When consensus meets self-stabilization," *Journal of Computer and System Sciences*, vol. 76, no. 8, pp. 884–900, 2010.

[69] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[70] Alberto Sonnino, "Hotstuff," https://github.com/asonnino/hotstuff/tree/3-chain, 2025.

[71] ——, "Jolteon," https://github.com/asonnino/hotstuff, 2025.

[72] The Diem Team, "Diem," https://github.com/diem/diem, 2025.

[73] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[74] Neil Giridharan, "Autobahn codebase," https://github.com/neilgiri/autobahn-artifact, 2025.

[75] Jianting Zhang, "Arete codebase," https://github.com/EtherCS/arete, 2025.

APPENDIX

In this section, we give a detailed security analysis for Sailfish-Lifefin proposed in § V.

Compared to the vanilla Sailfish protocol, Sailfish-Lifefin introduces an ACS-based fallback mechanism to accomplish the ordering task with a constant size of data. To be more specific, nodes handle at most n PoST blocks during the ACS instance and leverage the underlying committing rules to commit DAG vertices according to the ACS output. The modification is that Sailfish-Lifefin may select a new round r^* leader vertex $SB_L^{r^*}$ to replace the predefined round r^* leader vertex $v_L^{r^*}$ after each ACS instance. Thus, we adhere to the security analysis of Sailfish [33, Appendix B] while considering the subtle modification introduced by Lifefin.

Following Sailfish, we say that a leader vertex v_i is directly committed by node N_i if N_i invokes the commit_leader(v_k) function (Figure 5, Line 28). In contrast, we say that a leader vertex v_l is indirectly committed if v_l is added to $leaderStack$ by N_i (Figure 5, Line 35). Additionally, we say N_i consecutively directly commits leader vertices v_k and v'_k if N_i directly commits v_k and v'_k in rounds r and r' respectively and does not directly commit any leader vertex between r and r'.

A. Safety

We first analyze the safety of Sailfish-Lifefin. Safety indicates that all correct nodes commit transactions in the same order.

Claim 1. *For every two correct nodes N_i and N_j, at any given time t and round r, if there are two DAG vertices v_k and v'_k s.t. $v_k \in DAG_i[r] \wedge v'_k \in DAG_j[r] \wedge v_k.creator = v'_k.creator$, then $v_k = v'_k$.*

Proof: For the sake of contradiction, if $v_k \neq v'_k$, then $v_k.creator$ (wlog called N_k) must create two different vertices in round r. Note that in Sailfish-Lifefin, a node N_i adds a vertex v_k to its local DAG $DAG_i[r]$ after delivering v_k through RBC or ACS. However, both events require $2f+1$ nodes to acknowledge v_k. By the standard quorum intersection argument, there exists at least one correct node equivocated if $v_k \neq v'_k$, which is a contradiction. □

Claim 2. *Given any round number r^* that is decided by an ACS instance, at most $2f$ vertices can be created in round r^*+1.*

Proof: For the sake of contradiction, assume there exists a set of $2f+1$ round r^*+1 vertices before nodes commit the ACS instance. Since the ACS output consists of at least $2f+1$ vertices (ACS validity), by a quorum intersection argument, there exists at least $f+1$ nodes that have created their round r^*+1 vertex but creating their PoST block with a round $r' < r^*+1$ vertex. Since at most f nodes are malicious, this leads to a contradiction. Additionally, due to the graceful transition, nodes will create new vertices in round $r+2$ after the ACS instance. Consequently, there are at most f vertices created in round r^*+1. □

Proof of Proposition 1. By Claim 2 that at most $2f$ vertices are created in round r^*+1, before the fallback terminates, the predefined round r^* leader vertex cannot collect $\geq 2f+1$ votes, and thus cannot be directly committed according to Sailfish's direct committing rule. Moreover, recall that the indirect committing rule necessitates a round $r' > r^*$ leader vertex to be directly committed; in that case, the fallback instance must terminate. The proof is done.

Claim 3. *If a node N_i directly or indirectly commits a leader vertex v in round r, and a node N_j directly or indirectly commits a leader vertex v' in round r, then $v = v'$.*

Proof: Note that Sailfish-Lifefin may replace a predefined leader vertex v_L^r in round r^* with $SB_L^{r^*}$ after an ACS instance. If $r \neq r^*$, by Claim 1, we know that both N_i and N_j have the same predefined leader vertex, i.e., $v = v'$. If $r = r^*$ and $v_L^r = SB_L^{r^*}$ (i.e., the ACS instance does not update the predefined leader vertex), we are trivially done. Otherwise, we show that no correct nodes directly or indirectly commit v_L^r while $v_L^r \neq SB_L^{r^*}$ in round $r = r^*$.

By Claim 2, we know that there are at most $2f$ vertices created in round $r+1$. Therefore, v_L^r cannot be directly committed by the condition in which a node receives $2f+1$ vertices in round $r+1$ (Figure 5, Line 27). Additionally, v_L^r cannot be directly committed through the ACS-based fallback mechanism since $v_L^r \neq SB_L^{r^*}$. Therefore, nodes will not directly commit v_L^r. Moreover, since after the ACS instance, nodes are forced to connect blocks in the ACS output (which excludes v_L^r), there is no path from any subsequent leader vertex to v_L^r. By the code of the commit_leader function, v_L^r will not be indirectly committed. The proof is done. □

Claim 4. *If a correct node N_i directly commits a leader vertex v_k in round $r = r^*-1$ (where round r^* is the round decided by an ACS instance), then all correct nodes must directly commit v_k in round r.*

Proof: According to Figure 5, there are two cases for each node N_i to directly commit a round $r = r^*-1$ leader vertex. If N_i directly commits v_k through the ACS-based fallback mechanism (Figure 5, Line 54), then by the agreement property of ACS, all correct nodes must perform finalize_fallback to directly commit v_k. If N_i directly commit v_k through receiving $2f+1$ round $r+1$ vertices (Figure 5, Line 27), then there exist at least $f+1$ correct $r+1$ vertices connecting v_k. According to the validity property of ACS where the output V contains at least $f+1$ correct PoST blocks, by a quorum intersection argument, v_k must be connected by at least one correct PoST block from the ACS output V. In this case, all other correct nodes that do not commit v_k through receiving $2f+1$ round $r+1$ vertices must directly commit v_k through the ACS-based fallback mechanism. □

Claim 5. *If a correct node N_i directly commits a leader vertex v_k in round $r \neq r^*-1$ (where round r^* is the round decided by an ACS instance), then for every leader vertex v_l in round r' such that $r' > r$, there exists a path from v_l to v_k.*

Proof: According to Figure 5, if $r \neq r^*-1$, v_k in round r can be directly committed under two paths respectively: 1) in the

optimistic path (Line 27 where $r \neq r^*$), and 2) in the fallback path (Line 56 where $r = r^*$).

1) If $r \neq r^*$, v_k is chosen in the optimistic path, and there exists a set $\mathcal{Q}$ of $2f+1$ round $r+1$ vertices that connect v_k. Let $\mathcal{H} \in \mathcal{Q}$ be the set of vertices created by correct nodes. Note that $|\mathcal{H}| \geq f+1$. We then complete the proof by showing that the statement holds for any $r' > r$.

Case $r' = r+1$: Since $r \neq r^* - 1$, v_l must be the predefined leader vertex (i.e., the creation of v_l satisfies the constraints illustrated in § V-A). If $v_l \in \mathcal{H}$, we are trivially done. Otherwise, the vertices in $\mathcal{H}$ are from round $r+1$ correct non-leader nodes. Since $|\mathcal{H}| \geq f+1$ and nodes creating vertices of $\mathcal{H}$ do not send a round r no-vote message, by the quorum intersection argument, the round r' leader node cannot form a $\mathcal{NVC}_r$ for v_l such that v_l does not connect v_k. Therefore, if v_l exists, it must connect v_k and there exists a path from v_l to v_k.

Case $r' > r+1$: Since Sailfish adopts RBC to propagate vertices, all vertices in $\mathcal{H}$ ($|\mathcal{H}| \geq f+1$) will eventually be delivered by all correct nodes and added to $DAG[r+1]$. Furthermore, according to the construction of the DAG, a round $r+2$ vertex must connect at least $2f+1$ round $r+1$ vertices. By a quorum intersection argument, each round $r+2$ vertex (including round $r+2$ leader vertex) must have a path to v_k. Similarly, each round $r+3$ vertex must have a path to a round $r+2$ vertex, and thus has a path to v_k. By transitivity, all vertices in $r'' > r+3$ have a path to v_k. This implies v_l in round $r' > r+1$ must have a path to v_k.

2) If $r = r^*$, v_k is included in the ACS output V. By the design of Sailfish-Lifefin, all vertices from the next round $r+2$ must connect to all vertices (including v_k) in V and thus have a path to v_k. If $r' = r+2$, then we are done. Otherwise, similar to the second case of 1), we can prove that v_l must have a path to v_k. □

Claim 6. *If a correct node N_i directly commits a leader vertex v_k in round r and a correct node N_j directly commits a leader vertex v_l in round $r' \geq r$, then N_j (directly or indirectly) commits v_k in round r.*

Proof: If $r = r'$, by Claim 1, $v_k = v_l$ and we are trivially done. When $r' > r$, there are two cases depending on the relation between r and r^*, where round r^* is the round decided by an ACS instance.

Case $r = r^* - 1$: By Claim 4, all correct nodes (including N_j) directly commit v_k in round r.

Case $r \neq r^* - 1$: By Claim 5, there exists a path from v_l to v_k. By the code of the commit_leader function, after directly committing v_l, N_j indirectly commits leader vertices v_m in smaller rounds such that there exists a path from v_l to v_m until it reaches a round $r'' < r'$ in which it previously directly committed a leader vertex. If $r'' < r < r'$, N_j will indirectly commit v_k in round r. Otherwise, by inductive argument, N_j must have indirectly committed v_k through directly committing round r'' leader vertex. □

Claim 7. *Let v_k and v'_k be two leader vertices consecutively directly committed by a party N_i in rounds r_i and $r'_i > r_i$, respectively. Let v_l and v'_l be two leader vertices consecutively directly committed by N_j in rounds r_j and $r'_j > r_j$, respectively. Then, N_i and N_j commit the same leader vertices between rounds $\max(r_i, r_j)$ and $\min(r'_i, r'_j)$ and in the same order.*

Proof: If $r'_i < r_j$ or $r'_j < r_i$, then there are no rounds between $\max(r_i, r_j)$ and $\min(r'_i, r'_j)$, and we are trivially done. Otherwise, assume wlog that $r_i \leq r_j < r'_i$. By Claim 3 and 6, both N_i and N_j will (directly or indirectly) commit the same leader vertex in the round $\min(r'_i, r'_j)$. Assume $\min(r'_i, r'_j) = r'_i$. Given that Sailfish-Lifefin propagates vertices via RBC, both DAG_i and DAG_j will contain r'_k and all vertices that have a path from v'_k.

Recall from the implementation of the commit_leader function, after committing the leader vertex v'_k, nodes indirectly commit leader vertices in smaller rounds until they reach a round in which they previously directly committed a leader vertex. Consequently, both N_i and N_j will be able to indirectly commit all leader vertices from r'_i to r_j (i.e., from $\min(r'_i, r'_j)$ to $\max(r_i, r_j)$ under our general assumption). Furthermore, thanks to deterministic code of commit_leader, both nodes will commit the same leader vertices between $\min(r'_i, r'_j)$ to $\max(r_i, r_j)$ in the same order. □

By applying Claim 7 inductively to any two pairs of correct nodes, we derive the following corollary.

Corollary 1. *correct nodes commit the same leader vertices in the same order.*

Lemma 1 (Safety)**.** *Sailfish-Lifefin satisfies safety.*

Proof: By Corollary 1, nodes will commit the same leader vertices in the same order. With a deterministic order algorithm implemented in the order_vertices function, nodes consistently order the committed leader vertices as well as their causal histories. Given the same transaction payload in each DAG vertex, all correct nodes commit transactions in the same order. The safety property is guaranteed. □

B. Liveness

We now analyze the liveness of Sailfish-Lifefin. Liveness indicates that the system can continuously commit new transactions. We consider nodes to have limited resources to maintain pending DAG vertices. When a node exhausts its resources, it cannot receive and create DAG vertices.

Recall that a node triggers N_i the ACS-based fallback mechanism if one of the following conditions gets satisfied:

- **Condition 1:** N_i's uncommitted vertices exceeded a limit;
- **Condition 2:** N_i receives a PoST block and cannot commit DAG vertices for a timeout T_{st};

After triggering the fallback mechanism, N_i constructs its PoST block SB_i. We call SB_i a certified PoST block if it contains $2f+1$ signatures.

Claim 8. *If a correct node N_i triggers the ACS-based fallback mechanism at time t, then all correct nodes receive N_i's certified PoST block SB_i by time $\max\{GST, t\} + 3\Delta$.*

Proof: Recall that N_i adopts a `Propose` and `Vote` scheme to construct SB_i, which requires at most 2Δ to collect $2f+1$ signatures after GST. It then requires at most Δ to disseminate the certified PoST block after GST. The proof is done. □

Claim 9. *If a correct node N_i receives a certified SB_j from node N_j at time t, then after that N_i either triggers the ACS-based fallback mechanism or fails to commit DAG vertices for at most T_{st}.*

Proof: Assume the most recent time that N_i commits DAG vertices is $t' > t$. By time $t' + T_{st}$, if N_i does not commit any DAG vertices, by **Condition 2**, N_i triggers the fallback mechanism at time $t' + T_{st}$. Otherwise, if N_i commits DAG vertices during the time from t' to $t' + T_{st}$, t' updates. By iteration, the statement keeps holding. □

Claim 10. *Sailfish-Lifefin can continuously commit new DAG vertices even though correct nodes have limited resources to maintain pending DAG vertices.*

Proof: By **Condition 1**, a correct node N_i triggers the fallback mechanism once its resource usage for uncommitted vertices exceeds a quota. In practice, the quota can be set appropriately such that N_i always has available resources to proceed in an ACS instance, that is, it can handle at most n PoST blocks where n is the number of nodes. Once N_i triggered the fallback mechanism, by Claim 8, all correct nodes eventually received its PoST block. By Claim 9, there are two cases:

Case All correct nodes trigger the ACS mechanism: By the termination property of ACS, all correct nodes can output a set of PoST blocks V and commit pending DAG vertices based on V. Additionally, by the validity property of ACS, every correct node eventually has $|V| \leq n - f$ referred vertices to move to a new round and propose new vertices. After all correct nodes switch to the optimistic path, the underlying Sailfish protocol ensures that they can continuously generate new DAG vertices.

Case The protocol keeps committing DAG vertices: the statement trivially holds. □

Lemma 2 (Liveness)**.** *Sailfish-Lifefin satisfies liveness.*

Proof: By Claim 10, Sailfish-Lifefin can continuously generate and commit new DAG vertices. As the protocol allows nodes to create new round DAG vertices only when there exist at least $2f + 1$ vertices for the last round, at least $f + 1$ correct nodes can continuously generate and commit new DAG vertices. As a result, transactions will eventually be delivered and committed if they are sent to all correct nodes, that is, the liveness property is guaranteed. □ In this section, we give a detailed security analysis for Mysticeti-Lifefin proposed in § VI.

Recall that to order DAG vertices, the Mysticeti protocol iteratively marks *all* predefined leader vertices as either to-commit or to-skip status. By applying the decision rules to interpret the DAG pattern of each leader vertex, Mysticeti allows nodes to order leader vertices consistently. When integrating Lifefin into Mysticeti, to order DAG vertices, the only modification is that Mysticeti-Lifefin may select a new leader vertex $SB_L^{r^*}$ to replace the predefined leader vertex $v_L^{r^*}$ from the same round r^* after each ACS instance. Thus, we follow the security proof of Mysticeti [37, Section V and Appendix C] to complete our proof while considering the subtle modification introduced by Lifefin.

C. Safety

We first analyze the safety of Mysticeti-Lifefin. Safety indicates that all correct nodes commit transactions in the same order.

Claim 11. *If a correct node commits a leader vertex v in round r, then no correct nodes decide to directly skip v.*

Proof: We prove it by contradiction. Recall that a node N_i decides to directly skip a leader vertex v in round r if N_i observes a *skip pattern* for v, that is, at least $2f + 1$ round $r + 1$ vertices do not reference v. If another correct node N_j committed v, by the decision rules, there exists at least one *certificate pattern* for v, that is, at least $2f + 1$ round $r + 1$ vertices reference v. By a quorum intersection argument, at least one correct node proposes two vertices in round $r + 1$ that, respectively, reference and do not reference v, which is a contradiction. □

Claim 12. *For any round r leader vertex v, if $2f+1$ vertices in round r' from distinct nodes certify v, then all leader vertices in round $r'' > r'$ will have a path to a certificate for v.*

Proof: Recall that a round r' vertex v' is said to certify a round r vertex v (where $r' > r+1$) if v' includes a *certificate pattern* in its causal history. v' is also called a certificate for v. We prove the statement by contradiction. Assume there exists a leader vertex v'' in round $r'' > r'$ that does not reference a certificate for v. Since v'' must reference at least $2f + 1$ vertices from the previous round, if $r'' = r' + 1$, by a quorum intersection argument, a correct node equivocated in round v', which is a contradiction. Thus, all vertices in round $r' + 1$ reference a certificate for v. By transitivity, all vertices (including leader vertices) in round $r'' > r'$ have a path to a certificate for v. □

Claim 13. *If a correct node N_i directly commits a leader vertex v in round r, then no correct nodes decide to skip v.*

Proof: We prove it by contradiction. Assume that a correct node N_i directly commits a round r leader vertex v while another correct node N_j decides to skip v. We consider all cases where N_j decides to skip v.

Case N_j decides to directly skip v: This is a contradiction of Claim 11.

Case N_j decides to indirectly skip v: According to the indirect decision rule, there exists a committed leader vertex v' in round $r' > r+2$ that does not reference (or have no path to) a certificate for v. However, since N_i directly commits v,

there are $2f+1$ certificates for v, leading to a contradiction due to Claim 12. □

Claim 14. *Given any round number r^* that is decided by an ACS instance, at most $2f$ vertices from distinct nodes can be created in round r^*+1.*

Proof: For the sake of contradiction, assume there exists a set $\mathcal{Q}$ of $2f+1$ round r^* vertices. Let $\mathcal{H} \in \mathcal{Q}$ be the set of vertices created by correct nodes. Then we have $|\mathcal{H}| \geq f+1$. During the ACS instance, since a node uses its last vertex to construct the PoST block, there will be $|\mathcal{H}| \geq f+1$ PoST blocks include a vertex with a round $r' \geq r^*+1$. However, by the validity of ACS, there are at least $f+1$ correct PoST blocks included in the ACS output V. Given that r^* is set as the highest round number among V (Figure 8, Line 42), by a quorum intersection argument, there exists a correct node constructing two PoST blocks in one ACS instance, which is a contradiction. Additionally, due to the graceful transition, nodes will create new vertices in round r^*+2 after the ACS instance. Consequently, there are at most $2f$ vertices created in round r^*+1. □

Proof of Proposition 1. By Claim 14 that at most $2f$ vertices are created in round r^*+1, before the fallback terminates, the predefined round r^* leader vertex cannot collect $\geq 2f+1$ votes, and thus cannot be directly committed according to Mysticeti's direct committing rule. Moreover, recall that the indirect committing rule necessitates a round $r' > r^*$ leader vertex to be directly committed; in that case, the fallback instance must terminate. The proof is done.

Claim 15. *If a node N_i marks a leader vertex v in round r as to-commit or to-skip, and a node N_j marks a leader vertex v' in round r as to-commit or to-skip, then $v = v'$.*

Proof: Recall that Mysticeti-Lifefin may change a predefined leader vertex v_L to a new leader vertex SB_L in round r^* decided by an ACS instance. If $r \neq r^*$, we are trivially done as both v and v' are predefined. If $r = r^*$ and $v_L = SB_L$ (i.e., the ACS instance does not change the leader vertex), we are also trivially done.

We now consider the case where $r = r^*$ and $v_L \neq SB_L$. Let wlog $v = v_L$ and $v' = SB_L$. This indicates that N_i has decided v before terminating the ACS instance. By the decision rules, the earliest round that a node can decide a leader vertex is round $r+1$ (i.e., the node directly skips it after observing that $2f+1$ round $r+1$ vertices do not reference it). However, by Claim 14, both v and v' cannot be decided in round $r+1$. N_i has to wait for $2f+1$ round $r+2$ vertices to decide v. However, upon receiving $f+1$ round $r+2$ vertices that reference PoST blocks, N_i must be able to terminate the ACS instance and update round r leader vertex to SB_L, leading to a contradiction. □

Claim 16. *All correct nodes decide a consistent status for a leader vertex of each round, i.e., if two correct nodes N_i and N_j decide the status of a leader vertex, then both either commit or skip the same leader vertex.*

Proof: Let $[v^i]_{i=0}^{k}$ and $[v^i]_{i=0}^{l}$ denote the status of the leader vertices for N_i and N_j, such that k and l are respectively the indices of the highest committed round number. By definition, N_i directly commits round k leader vertex, and N_j directly commits round l leader vertex. Assume wlog that $k \leq l$. We now use induction to prove statement $P(r)$ for $0 \leq r \leq k$: if N_i and N_j both decide the round r leader vertex, then both either commit or skip the same leader vertex.

Base case $r = k$: if N_i directly commits round k leader vertex v_k, by Claims 11 and 15, if N_j decides the round k leader vertex, then N_j must commit v_k.

Inductive hypothesis and step: assume that $P(r)$ is true for $m+1 \leq r \leq k$, we now prove $P(m)$. Like in the base case, if N_i directly commits a round m leader vertex v_m, then N_j commits v_m if it decides the round m leader vertex. Additionally, if N_i directly skips v_m, by (inverse) Claim 11, N_j skips v_m. We now analyze the only left case where N_i and N_j indirectly decide v_m. Let m' denote the first round with a round number higher than the decision round of v_m. There exist rounds $m_i \geq m'$ and $m_j \geq m'$ such that N_i commits a leader vertex v_i in m_i while skipping all leader vertices in $[m', m_i)$ and N_j commits a leader vertex v_j in m_j while skipping all leader vertices in $[m', m_j)$. Since $m_i \leq k$, by the induction hypothesis, we have $m_i = m_j$ and $v_i = v_j = v$. It means that N_i and N_j use the same leader vertex to indirectly decide v_m. Thus, N_i and N_j have the same DAG pattern for v_m with the same causal history of v and decide round m leader vertex consistently. □

Claim 17 (Data availability)**.** *For any round r leader vertex v that is marked to-commit, all correct nodes can derive the whole causal history of v.*

Proof: Let r^* denote the round number selected by an ACS instance. If $r \neq r^*$, we are trivially done as Lifefin does not affect the underlying committing rules (i.e., a node votes for a leader vertex only if it receives the whole causal history of the leader vertex). We now consider the case of $r = r^*$. Note that for any PoST block SB_i constructed by a node N_i, $SB_i.vertex$ must reference $2f+1$ vertices in round $r' = SB_i.vertex.round - 1$, indicating that at least $2f+1$ nodes are processing in a round $\geq r'$. Thus, by a quorum intersection argument, the $2f+1$ signatures in SB_i contain at least one from a correct node N_j processing in a round $\geq r'$. By the construction of PoST blocks (Figure 7, Line 17), N_j synchronizes the whole causal history of SB_i before signing it. Thus, for any v selected by the ACS instance, there exists at least one correct node storing the causal history of v, and thus all correct nodes can derive the causal history of v. □

Lemma 3 (Safety)**.** *Mysticeti-Lifefin satisfies safety.*

Proof: Since Mysticeti-Lifefin enforces nodes to decide a leader vertex in each round, by Claim 16, all correct nodes will commit the same leader vertices in the same order. By Claim 17, correct nodes can collect causal histories of all committed leader vertices. With a deterministic order algorithm, all correct nodes order DAG vertices in the same order.

Given the same transaction payload in each DAG vertex, all correct nodes commit transactions in the same order. The safety property is guaranteed. □

D. Liveness

We now analyze the liveness of Mysticeti-Lifefin. Liveness indicates that the system can continuously commit new transactions. We consider nodes to have limited resources to maintain pending DAG vertices. When a node exhausts its resources, it cannot receive and create DAG vertices.

Recall that a node triggers N_i the ACS-based fallback mechanism if one of the following conditions gets satisfied:

- **Condition 1:** N_i's uncommitted vertices exceeded a limit;
- **Condition 2:** N_i receives a PoST block and cannot commit DAG vertices for a timeout T_{st};

After triggering the fallback mechanism, N_i constructs its PoST block SB_i. We call SB_i a certified PoST block if it contains $2f+1$ signatures.

Claim 18. *If a correct node N_i triggers the ACS-based fallback mechanism at time t, then all correct nodes receive N_i's certified PoST block SB_i by time $\max\{GST, t\} + 5\Delta$.*

Proof: Recall that N_i adopts a `Propose` and `Vote` scheme to construct SB_i. Once receiving an uncertified SB_i, a node N_j might take two communication steps (i.e., 2Δ) to synchronize the missing causal history (Figure 7, Line 19). Thus, N_i requires at most 4Δ to collect $2f+1$ signatures after GST. It then requires at most Δ to disseminate the certified PoST block after GST. The proof is done. □

Claim 19. *If a correct node N_i receives a certified SB_j from node N_j at time t, then after that N_i either triggers the ACS-based fallback mechanism or fails to commit DAG vertices for at most T_{st}.*

Proof: Assume the most recent time that N_i commits DAG vertices is $t' > t$. By time $t' + T_{st}$, if N_i does not commit any DAG vertices, by **Condition 2**, N_i triggers the fallback mechanism at time $t' + T_{st}$. Otherwise, if N_i commits DAG vertices during the time from t' to $t' + T_{st}$, t' updates. By iteration, the statement keeps holding. □

Claim 20. *Mysticeti-Lifefin can continuously commit new DAG vertices even though correct nodes have limited resources to maintain pending DAG vertices.*

Proof: By **Condition 1**, a correct node N_i triggers the fallback mechanism once its resource usage for uncommitted vertices exceeds a quota. The quota is configured to ensure that N_i retains sufficient resources to participate in an ACS instance. This includes: (i) synchronizing any missing DAG vertices from the last ACS instance, which accounts for at most $f * (r+1-\max\{r_{fb}, r_{gc}\})$ DAG vertex as a node N_i only needs to synchronize the causal history of PoST blocks in round $r' \leq r+1$ (cf. Figure 7, Line 17-19), where r is N_i's stuck round, r_{fb} is the last fallback round, and r_{gc} is the round whose previous vertices are garbage-collected; (ii) managing up to n PoST blocks required for the current ACS instance, where n is the number of nodes. Once N_i triggered the fallback mechanism, by Claim 18, all correct nodes eventually received its PoST block. By Claim 19, there are two cases:

Case All correct nodes trigger the ACS mechanism: By the termination property of ACS, all correct nodes can output a set of PoST blocks V and mark a newly selected leader vertex SB_L as to-commit. By the data availability Claim 17, all correct nodes can derive the whole causal history of SB_L and commit pending DAG vertices based on V. Additionally, by the validity property of ACS, every correct node eventually has $|V| \leq n-f$ referred vertices to move to a new round and propose new vertices. After all correct nodes switch to the optimistic path, the underlying Mysticeti protocol ensures that they can continuously generate new DAG vertices.

Case The protocol keeps committing DAG vertices: the statement trivially holds. □

Lemma 4 (Liveness)**.** *Mysticeti-Lifefin satisfies liveness.*

Proof: By Claim 20, Mysticeti-Lifefin can continuously generate and commit new DAG vertices. As the protocol allows nodes to create new round DAG vertices only when there exist at least $2f+1$ vertices for the last round, at least $f+1$ correct nodes can continuously generate and commit new DAG vertices. As a result, transactions will be eventually delivered and committed if they are sent to all correct nodes, that is, the liveness property is guaranteed. □ In this section, we describe a specific implementation of inflation attacks on two state-of-the-art protocols: Sailfish and Mysticeti.

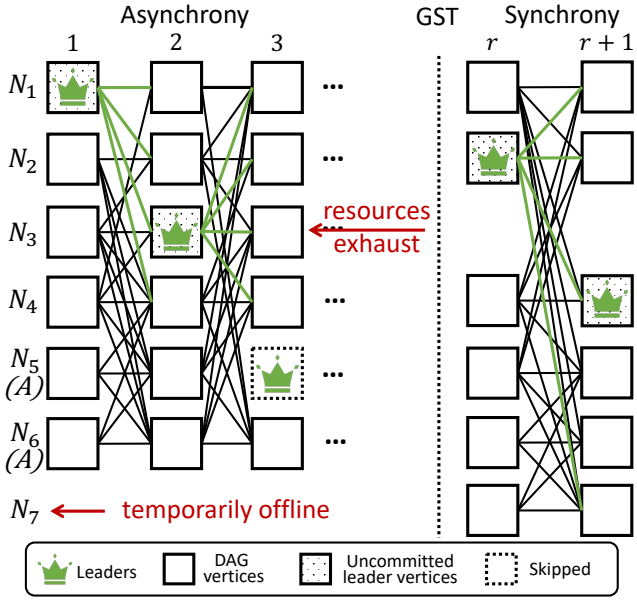


Fig. 13: Overview of the inflation attack in DAG-based BFT, where N_5 and N_6 are attackers and N_7 is temporarily offline: attackers skip proposing their leader vertices and do not vote for leader vertices. By doing this, attackers exhaust N_3's resources before the network recovers to the synchrony

As illustrated in Figure 13, the adversary controls f nodes

out of $3f + 1$ nodes. For simplicity, we use $f = 2$ as an example, and therefore the dissemination quorum $q_d = 5$ and the committing quorum $q_c = 5$. Recall that in DAG-based BFT protocols, a leader vertex can only be committed if it is connected by q_c subsequent vertices (see § II-A). In other words, a leader vertex will remain uncommitted if it is connected by at most $q_c - 1$ subsequent vertices. Such uncommitted vertices will continue to consume nodes' resources. The inflation attack that we implemented on Mysticeti and Sailfish utilizes this property by deliberately creating scenarios where leader vertices remain uncommitted. This forces nodes to allocate resources to these uncommitted vertices, eventually causing mempool explosions.

More specifically, the adversary first makes a correct node temporarily unavailable by launching a DDoS attack or causing a misconfiguration error on it. Then, the adversary uses the following approach to prevent any leader vertex from being committed.

In a round where a correct node is selected to propose the leader vertex, Byzantine nodes (i.e., N_5 and N_6) create vertices in the next round that deliberately do not connect to the leader vertex. Given that the adversary controls f nodes and at least one correct node is temporarily offline due to the DDoS attack, the correct leader vertex can only be connected by at most $2f$ vertices. Because of the requirement $q_c = 2f+1$ for Mysticeti and Sailfish, the correct leader vertex cannot be committed.

In a round where a Byzantine node is selected to propose the leader vertex, the adversary deliberately skips this task, resulting in no leader vertex being created for that round. Consequently, the vertices proposed in previous rounds still remain uncommitted.

Note that during the above process, the data dissemination task is always successful as there are $3f > q_d$ nodes generating new vertices. As a result, correct nodes require maintaining an ever-growing uncommitted DAG in their mempool, and eventually exhaust their resources.

If a correct node exhausts its resources before GST (e.g., N_3 in Figure 13), it becomes incapable of performing any tasks, even if the network recovers to synchrony. Then, the adversary can adopt one of the following strategies to sustain consensus failure, depending on how the protocol handles the mempool explosion. In particular, if the exhausted node can recover by upgrading resources (such as memory or storage), the adversary can repeatedly trigger mempool explosions using the same approach as discussed above. Note that this time, the adversary no longer needs to launch DDoS attacks since there is already one correct node (i.e., the node exhausting its resources) that becomes unavailable. On the other hand, if there is no solution for handling mempool explosions, the adversary can simply stop creating new vertices. In this case, since only $2f$ correct nodes are available, by the dissemination quorum $q_d = 2f + 1$, the system will stall indefinitely. Both of the above adversarial behaviors ultimately compromise the liveness of DAG-based BFT protocols.

In the main body, we show how a mempool explosion can compromise liveness of partially synchronous DAG BFT protocols. In this section, we discuss whether the adversary can exploit the mempool explosion on non-DAG decoupled BFT protocols to compromise these protocols' liveness.

E. Mempool explosions on linear-chain BFT protocols

The mempool explosion can be exploited by the adversary on most implementations of linear-chain protocols [70]–[72], including the HotStuff family of protocols [1], [17], [25]–[29]. However, the vulnerability depends on the specific implementation of their pacemaker module [1].

In these implementations, the following scenario can occur: during a prolonged period of asynchrony, the protocol may experience multiple rounds that time out. In each round, the leader successfully propagates a proposal to the entire committee but fails to collect enough votes to form a certificate (due to the asynchrony or the exploited attacks). As a result, validators retain uncertified blocks while waiting to determine which one will be used for progress. This approach, however, is unsustainable under prolonged periods of asynchrony, as it requires unbounded resources.

To withstand this vulnerability, these implementations could adopt a strategy where validators discard uncertified blocks [17] after a fixed number of timeout rounds [1], [17]. However, this solution presents a challenge, as validators typically commit to persisting and serving any block they sign [29], [70]. A promising direction for future work is to explore whether validators could retain only the latest two or three rounds of blocks (depending on the protocol [1], [17], [28]). This would ensure that each validator's storage requirements grow only linearly with the size of the committee.

We finally note that all implementations of these linear-chain BFT protocols that separate data dissemination from consensus [70], [71] are inherently sensitive to the mempool explosion attacks. Validators continuously generate batches of transactions intended for inclusion in future blocks. This forces committee members to hold onto these uncommitted batches, and, to the best of our knowledge, there is currently no literature exploring a cleanup strategy for such batches. Once again, these protocols could mitigate the issue by retaining only a fixed number of batches per round and producing empty batches upon detecting a prolonged loss of liveness. However, this approach is delicate, as it complicates formal liveness arguments and may introduce unforeseen performance degradation during temporary and brief losses of network synchrony.

F. Mempool explosions on multi-chain BFT protocols

Recent decoupled BFT protocols (such as Autobahn [8], Star [23], and Arete [24]) adopt a multi-chain structure to separate data dissemination from consensus. In these protocols, nodes can propagate multiple blocks of transactions in parallel and append the blocks to multiple chains. The consensus task is performed separately by any existing linear-chain BFT protocols (such as Hotstuff [1] and PBFT [73]), where nodes take as input the hash digests of disseminated blocks and agree on a global order across multiple chains.

Our inflation attacks can be extended to these multi-chain BFT protocols. Specifically, the adversary proactively participates in the data dissemination task by continuously proposing new blocks, but it prevents the protocol from agreeing on the global order by making arbitrarily one correct node unavailable for a while. This can be achieved by the adversary launching a DDoS attack or causing a misconfiguration error on the correct node. As a result, correct nodes will experience a mempool explosion with the ever-growing uncommitted blocks, compromising the protocol's liveness.

To mitigate this vulnerability, these protocols could force nodes to stop producing new blocks once their resources are nearly exhausted and to always preserve enough resources for the ordering task only (i.e., for a BFT instance in the ordering task). However, this solution also relies on the specific implementation of the employed BFT protocol and faces the same issues as discussed in Section E. To the best of our knowledge, none of these multi-chain BFT protocols consider the management of limited resources for their liveness guarantees in both theoretical analysis and implementations [74], [75]. It is worth emphasizing that traditional garbage collection mechanisms cannot circumvent this issue, as they rely on consensus to determine which data can be discarded, but consensus is never reached in the above cases. An independent and urgent follow-up topic for both academia and industry is to explore similar liveness vulnerabilities and design efficient mitigations for these BFT protocols.