HammerHead: Score-based Dynamic Leader Selection

Giorgos Tsimos
Mysten Labs*
University of Maryland, College Park

Anastasios Kichidis
Mysten Labs

Alberto Sonnino
Mysten Labs
University College London

Lefteris Kokoris-Kogias
Mysten Labs
IST Austria

***Abstract*—Recent advancements on DAG-based consensus protocols allow for blockchains with improved metrics and properties, such as throughput and censorship-resistance. Variants of the Bullshark [18] consensus protocol are adopted for practical use by the Sui blockchain, for improved latency. However, the protocol is leader-based, and is strongly affected by crashed leaders that can lead to various performance issues, for example, decreased transaction throughput.**

In this paper, we propose HammerHead, a DAG-based consensus protocol, that is inspired by Carousel [8] and provides Leader-Utilization. Our proposal differs from Carousel, which is built for a chained consensus protocol; in HammerHead chain quality is inherited by the DAG. HammerHead needs to preserve safety and liveness, despite validators committing leader vertices asynchronously. The key idea is to update leader schedules dynamically, based on the validators' scores during the previous schedule.

We implement HammerHead and show a minor improvement in performance for cases without faults. The major improvements in comparison to Bullshark appear in faulty settings. Specifically, we show a drastic, 2x-latency improvement and up to 40% increased throughput when crash faults occur (100 validators, 33 faults).

***Index Terms*—Blockchains, Consensus, Byzantine Atomic Broadcast, Leader Utilization**

I. Introduction

Advances in Byzantine Fault Tolerant State-Machine-Replication (SMR) dictated by the needs of blockchain technology to have high throughput and censorship resistance (also referred in the literature as Chain Quality [8]) has resulted in a surge of research around DAG-based consensus [3], [10], [12], [15]–[18], [22]. These protocols are now being deployed in production environments. For instance, Bullshark [18] has been adopted by the Sui blockchain [21] and is on the roadmap of Aptos [20] due to its lower latency and non-reliance on setting up and maintaining a common coin. This, however, comes with the caveat that Bullshark is a leader-based protocol which results in performance deterioration when some candidate leaders inevitably crash or are taken down for maintenance and software update.

This phenomenon has been already seen in Sui's production deployment. For example on August 29th, between 15:30 and 17:30 UTC, suddenly 10% of the validators started being less responsive. This resulted in the p95 latency going up from 3 seconds to 4.6 seconds and even the p50 latency increasing from 1.9 seconds to 2.2 seconds. This is especially alarming because at that point the system was under low load (only 130 tx/sec) so the lost capacity did not affect the latencies. Furthermore, in real blockchains, validators vary in stake and thus leader election frequency. Some high-stake validators act as leaders more often than others, but when they briefly fail or undergo maintenance, performance suffers, causing stress for node maintainers who must work tirelessly to restore them. This pressure arises because missing many leader spots affects overall performance. We present HammerHead, a protocol that aims to ease this burden by promptly removing these major validators from the leader schedule temporarily and swiftly reintegrating them when they recover, ensuring seamless operation. These findings confirm our intuition that the cost of not having a leader-aware SMR [8] is significant even in DAG-based consensus protocols.

To resolve this challenge we design a leader-aware SMR for DAG-based consensus protocols. Inspired by Carousel, we also rely on on-chain metrics to achieve high leader-utilization. However, doing this on a DAG instead of a chain is not trivial. Firstly, unlike chained consensus protocols, DAGs do not commit blocks in the same view for all nodes. As a result, we cannot simply rely on the consensus protocol for agreement but need to open the black-box and adapt the way the DAG is interpreted to get safety and liveness. Additionally, DAG-based consensus protocols provide chain quality by design. Hence we need not aggressively diversify on who is the leader. HammerHead runs locally on each validator and does not require any extra protocol message or cryptographic tool.

To achieve Leader Utilization, HammerHead relies on the classic parent-based voting scheme adopted by many DAG-based protocols (such as Bullshark [18], Tusk [10], Dag-Rider [15], Fino [16]) to retrieve information regarding which parties are the fastest and most active during the current leader schedule. In every round, the fastest $2f+1$ parties to vote for a leader's proposal increase their respective scores. The scores accumulated during each schedule epoch are used during the calculation of the next leader schedule. Specifically the f validators with the lowest score (corresponding to the least active validators, either due to crashes or Byzantine behavior) lose their schedule slots, which are allocated to the set of f validators with the best score instead. A schedule change is triggered after a predetermined number of rounds has passed, but only upon an observable commit of the DAG in order to

*Work done when the author was an intern at Mysten Labs.

preserve the safety of the system.

Main challenges. The main technical challenges lie with maintaining all the properties of Byzantine Atomic Broadcast, while also guaranteeing liveness. A major difference from static leader schedules is that now different honest validators might be operating under different schedules. What we show is that all honest validators eventually allocate the same interval of rounds to the same schedule and thus have agreement on the DAG and on the schedule changes. HammerHead also solves different challenges than Carousel [8]. Carousel targets chained consensus protocols where the safety of the protocol is guaranteed even when honest validators disagree on the identity of leader: only liveness may suffer and need to be eventually restored. In contrast, HammerHead operates on DAG-based protocols where disagreement on the leader's identity may lead to safety violations.

Real-world system. We provide a *production-ready* and *fully-featured* (crash-recovery, monitoring tools, etc) implementation of HammerHead that has been adopted by the Sui blockchain: HammerHead runs within the Sui mainnet since version `mainnet-v1.9.1`[1]. Our evaluation shows that HammerHead (i) introduces no throughput loss and even provides small latency gains when the protocol runs in a faultless setting, (ii) drastically improves both latency and throughput in the presence of crash-faults, and that unlike Bullshark that deteriorates with more faults, HammerHead maintains performance (up to 2x latency reduction and 40% throughput improvement for 100-validator deployments suffering 33 faults); and (iii) does not suffer from any visible throughput degradation despite crash-faults.

Contributions. We make the following contributions:

- We present HammerHead, the first[2] reputation-based leader-election mechanism for DAG-based consensus protocols.
- We formally prove that HammerHead achieves Safety, Liveness, and Leader Utilization.
- We provide a production-ready and fully-featured implementation of HammerHead and demonstrates its benefits through extensive benchmarks.

II. Preliminaries

A. Model

Network. We assume a set Π of n parties (or validators; both are used interchangeably throughout this work) $\{p_1, \ldots, p_n\}$ and an *adaptive* adversary $\mathcal{A}$ that can corrupt up to $f < n/3$ of the parties arbitrarily, at any point. A party is *crashed* if it halts prematurely at some point during execution. Parties that deviate arbitrarily from the protocol are called *Byzantine* or bad. Parties that are never crashed or Byzantine are called *honest*. Parties are communicating over a partially synchronous network [11], in which there exists a special event called Global Stabilization Time (GST) and a known finite time bound Δ, such that any message sent by a party at time x is guaranteed to arrive by time $\Delta + \max\{\mathsf{GST}, x\}$.

Threat model. The adversary is computationally bounded. Pairwise points of communication between any two honest parties are considered *reliable*, i.e. any honest message is *eventually* (after a finite, bounded number of steps) delivered. However, until GST the adversary controls the delivery of all messages in the network, with the only limitation that the messages must be eventually delivered. After GST, the network becomes synchronous, and messages are guaranteed to be delivered within Δ time after the time they are sent, potentially in an adversarially chosen order.

B. Building Blocks

HammerHead leverages the *reliable broadcast* primitive.

Definition 1 (Reliable Broadcast)**.** *Each party P_i broadcasts messages by calling $r\_bcast_i(m, r)$, where m is a message and $r \in \mathbb{N}$ is a round number. Each party P_j outputs $r\_deliver_j(m, r, i)$, where m is a message, r is a round number, and $i \in [n]$ the index of party P_i who called the corresponding $r\_bcast_i(m, r)$. A Reliable Broadcast protocol achieves the following properties:*

> ***Agreement.*** *If an honest party P_i outputs $r\_deliver_i(m, r, k)$, then all other honest parties P_j eventually output $r\_deliver_j(m, r, k)$.*
> ***Integrity.*** *For every round $r \in \mathbb{N}$ and for every $k \in [n]$, an honest party P_i outputs $r\_deliver_i(m, r, k)$ at most once, regardless of m.*
> ***Validity***. *If an honest party P_k calls $r\_bcast(m, r)$, then eventually every honest party P_i outputs $r\_deliver_i(m, r, k)$.*

C. Problem Definition

Our result focuses on achieving *Byzantine Atomic Broadcast (BAB)*, while also satisfying additional properties. In order to keep notation clear between reliable and atomic broadcast, we refer to the BAB broadcast and deliver events as $a\_bcast(m, r)$ and $a\_deliver(m, r, p_j)$ respectively, where m is some message, $r \in \mathbb{N}$ is a round number and p_j is a party out of the n parties.

Definition 2 (Byzantine Atomic Broadcast)**.** *Each party P_i broadcasts messages by calling $a\_bcast_i(m, r)$, where m is a message and $r \in \mathbb{N}$ is a round number. Each party P_j outputs $a\_deliver_j(m, r, i)$, where m is a message, r a round number and $i \in [n]$ the index of party P_i who called the corresponding $a\_bcast_i(m, r)$. A* Byzantine Atomic Broadcast *protocol satisfies the properties of Reliable Broadcast and:*

> ***Total Order*** *If an honest validator P_i outputs $a\_deliver_i(m, r, k)$ before $a\_deliver_i(m', r', k')$, then no honest party P_j outputs $a\_deliver_j(m', r', k')$ before $a\_deliver_j(m, r, k)$.*

[1] https://github.com/MystenLabs/sui/releases/tag/mainnet-v1.9.1

[2] Developed concurrently with Shoal [17], see Section VII.

Throughout the text we will interchangeably refer to the properties of Agreement and Integrity, defined by BAB as *Safety*. Similarly, we define *Liveness* as the property that every honest party in the protocol will eventually commit a new anchor. We will later prove that HammerHead satisfies both Safety (i.e. BAB) and Liveness.

An additional property of interest to this work is *Leader Utilization*, introduced in Spiegelman et al. [17].

Definition 3 (Leader-Utilization)**.** *A BAB protocol achieves* Leader Utilization *if, in crash-only executions, after GST, the number of rounds r for which no honest party commits a vertex formed in r is bounded.*

D. The Bullshark Protocol

We present and build HammerHead on top of the partially synchronous Bullshark [18]. For clarity, we provide a description of the Bullshark protocol but we refer the reader to the original [18] or the shorter, partially synchronous-only version [19] of the protocol for more information. The aim of the protocol is to achieve Byzantine Atomic Broadcast (per Definition 2) via an efficient DAG-based protocol (as in *Directed Acyclic Graph*).

The DAG is an abstraction of the communication layer between parties. Every party constructs its own local DAG with respect to the communication it has observed throughout the protocol's execution. However, since each vertex of the DAG corresponds to a message that is reliable-broadcasted (by a specific party), the properties of the reliable broadcast guarantee non-equivocation as well as that the same vertices are eventually added to each local DAG.

Each party attempts to commit incoming vertices in the respective position in the DAG, depending on whether each incoming vertex satisfies the commit rules. Every even round of the protocol has a corresponding leader out of the participating nodes. The vertex for that round originating from the leader is defined as an "anchor". Each incoming message (vertex) contains a set of at least $n - f$ edges towards vertices from the previous round. For a party to commit the anchor a of round i in the DAG, the party has to receive at least $f + 1$ messages of round $i + 1$ with edges towards a — this is the "commit rule". If there exists an edge from vertex v to anchor a we say that v *voted for* a.

A notion that appears repeatedly throughout this line of work is that of "quorum intersection". In this case, the argument is as follows: *If an honest party commits anchor a, it must have observed at least $f+1$ votes for a in the next round $i+1$. Any anchor for round $i+2$ will have at least $n-f$ edges towards the previous round. Since $(n - f) + (f + 1) > n$, then any anchor for round $i + 2$ will have a path to a. By induction, any anchor for any round $\geq i + 2$, will have a path to a.* Quorum intersection guarantees that although parties might have a different view of the DAG at any given point, they eventually agree on the same view. Specifically, *if from some future anchor b, there is no path to a previous anchor a, then no honest party committed a and parties can disregard a.*

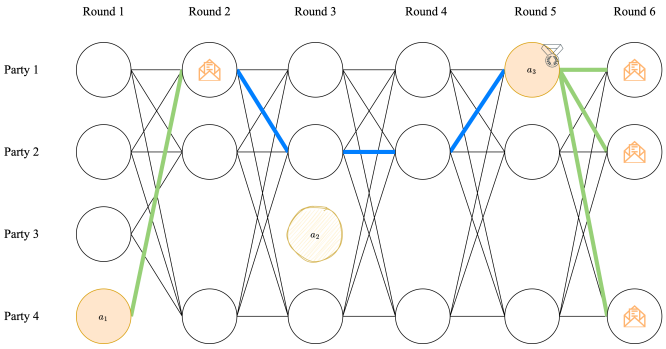

Fig. 1. An example of the commit rule execution in Bullshark. This is from the point of view of, say, Party 2 for the first 4 rounds. The first direct commit occurs in Round 5, where anchor a_3 receives sufficient votes to be committed. Party 2 has not received from Party 3, and a_2 is not in Party 2's local view of the DAG. Anchor a_1 was also not committed in Round 1. However, a_1 is in the causal history of a_3, so once a_3 is committed, a_1 will also be (indirectly) committed in the DAG.

The intuition of the protocol is to flesh out the full DAG, starting from the backbone of ordering the anchors. After that, vertices that are in between anchors can be deterministically ordered as well, eventually resulting to the same final DAG between all parties. Anchors are ordered recursively; if a party commits anchor a_i, the party checks if anchor a_{i-2} can also be commited. If so, the party orders anchor a_{i-2} before a_i and checks recursively for the predecessor of a_{i-2}, until it reaches a previously ordered anchor.

Another notion that is used throughout this work is that of the *causal history* of a vertex. In short, the causal history of vertex v on the DAG, is the set of all vertices reachable from v. Once the anchors have been ordered, their causal histories can be ordered as well, according to a deterministic rule. We show an example of the commit rule in Bullshark in figure 1.

III. The HammerHead Protocol

We propose a protocol that satisfies both Safety and Liveness, while operating on a dynamically changing schedule of leaders. Our protocol is inspired by Carousel as far as how we identify the well-performing validators and giving them more chances of being leaders. Unlike Carousel, we need not worry about chain quality but we need to take extra steps to make sure that the protocol is safe and live although it is running over a DAG (see Section VII for a more detailed comparison).

The protocol starts with an initial schedule S_0, which is a fair round-robin unbiased of the results of the previous epoch. The schedule can be initialized by randomly permuting all validators based on their stake; For example, if each validator u holds stake $\mathsf{stake}(u)$ and the total number of rounds that require leaders is TR, we initialize the schedule with each validator u being the leader of $TR \times \mathsf{stake}(u) / \sum_u \mathsf{stake}(u)$ rounds in order and then randomly permute them.

To compute a new schedule S' to switch from schedule S, we initialize a table pos with all validators. In pos there are two columns per validator, one with the initial number of

slots they have on the previous schedule S and another with the number of slots they will have on the schedule S'. Each validator goes through all the rounds where S is active and computes a data structure $\mathsf{scores}(\cdot)$ mapping each validator to their reputation score. Every validator starts with a reputation score of 0. Upon committing a sub-dag in Bullshark we update the reputation score of each validator, using some deterministic rule, in order to guarantee agreement across views. Since all validators observe the same sequence of committed sub-dags, they all attribute the same scores to validators.

We propose the deterministic rule for updating reputation scores to be that *each validator receives 1 point each time they vote for a leader's proposal (i.e., there is a parent link from the block of the validator at round r to the leader, according to schedule S, of round $r-1$).* The reputation score of each validator is increased by the number of points they accumulate.

The first subtle challenge to preserving Safety is that when we commit a sub-dag in Bullshark this happens through a subjective view of the DAG. This means that two validators might see a different subset of votes or they might even commit sub-dags at vastly different points in time. In order to resolve the first challenge we introduce a delay at the calculation of the reputation score. More specifically, although committing the leader is subjective what is consistent is that (a) every validator will eventually commit that same leader and (b) when the leader is committed the subDAG that gets committed is the same. Leveraging these two observations we calculate the reputation score up to but excluding the committed leader.

Furthermore, we separate the execution of the BAB in schedule epochs, each of which lasts approximately T leaders[3]. Once the epoch ends the validators compute a new leaders' schedule S' as follows: They select a set B that contains at most f validators (by stake); this set contains the validators with the lowest reputation scores. They also select a set G of equal size to B ($|G| = |B|$); this set contains the validators with the highest reputation scores. Any ties for either of the sets are deterministically resolved. The new schedule S' is computed by round-robin replacing each B validator with a G validator from the previous schedule S. To do the replacement we perform the following:

- Pick a validator P_b from B
- Find a slot they are leaders in S
- Pick a validator P_g from G
- Set $pos[v_g, 1] \leftarrow pos[v_g, 1]+1$; $pos[v_b, 1] \leftarrow pos[v_b, 1]-1$ and replace P_b with P_g in the new schedule S'

Once the S' is calculated, the new schedule takes effect immediately.

The second and most critical challenge of HammerHead appears during the schedule switch. This is because validators may not commit a leader immediately, but through recursion over the DAG and after an unbounded number of rounds before GST. Nevertheless, we show that if we carefully apply the schedules through and induction and without skips we can avoid any Safety violations.

Finally, Liveness is also at risk as validators in Bullshark only wait to see the block proposal of the leader every time. However, if validators are not synchronized and each one has a different belief of who is the leader of round r (because they are in a different schedule) then no leader might succeed in committing. An easy solution to this would be to forfeit responsiveness and make every round last Δ. Fortunately, with HammerHead we avoid this and create a responsive protocol by opening up the Bullshark algorithm and ensuring that after GST all honest validators will be in sync (or the adversary will have to keep them out of sync by committing subDAGs, effectively providing Liveness as well).

A. Protocol Specification

Our protocol is described in algorithm 1 and algorithm 2. It operates on top of a DAG-based BAB protocol, such as Bullshark [18].The main idea is to change the leader scheduling from static to adaptive, based on reputation scores. We already explained how the scores are computed by each validator in our practical application. However, our proposal is not specific to the calculation of the schedule and could work with any *deterministic* schedule-change rule. Our protocol differs from the original Bullshark protocol in specific parts.

Specifically, since schedules are being updated after committing an anchor by observing vertices that voted for an anchor; this means that schedule changes may need to occur retroactively. Thus, there may be cases where validators were operating under a previous schedule for a few rounds, perhaps because they were unable to commit anchors. Once a validator commits a new anchor, they update their view accordingly and observe the new schedule. Thus, they need to retroactively apply the new schedule for the time-period in which they where operating under the previous schedule, while the new schedule was already active.

We provide a crude logical flowchart of our protocol's execution in figure 2.

B. Protocol Correctness

We aim to prove the correctness of our construction. We will show that HammerHead satisfies the properties of BAB, as well as Liveness and Leader Utilization. First, we prove the following two claims, which can be proven in a similar fashion as in [15].

Claim 1. *When an honest party P_i adds a vertex u to its DAG_i, the entire causal history of u is already in DAG_i.*

Proof. The claim can be proven via strong induction on the vertices added by an honest party. Fix an honest party. Let $v_1, v_2, \dots v_M$ be the set of vertices added in the DAG by that party throughout the protocol's execution, ordered from first to last. The base case of the induction is when the DAG is still empty, and the property holds trivially. Let vertex v_i be added in the DAG and assume that when v_i is added, the property holds for all vertices in the set $\{v_1, \dots, v_i\}$, i.e. all their causal

[3]It might be slightly larger because the leader after the T-th commit are crashed.

Algorithm 1 Data structures and basic utilities for party p_i

Local variables:
struct *vertex* v: ▷ The struct of a vertex in the DAG
v.*round* - the round of v in the DAG
v.*source* - the party that broadcast v
v.*block* - a block of transactions information
v.*edges* - a set of at least $n - f$ vertices in v.*round* $- 1$ ▷ Provide fairness
$DAG_i[]$ - An array of sets of vertices
activeSchedule - auxiliary info related to the schedule change. Input to the deterministic GETLEADER$(\cdot)$ function.

1: **procedure** PATH(v, u) ▷ Check if exists a path from v to u in the DAG
2: **return** exists a sequence of $\mathsf{k} \in \mathbb{N}$, vertices $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_\mathsf{k}$ s.t.
$\mathsf{v}_1 = \mathsf{v}$, $\mathsf{v}_\mathsf{k} = \mathsf{u}$, and $\forall j \in [2..k]$: $\mathsf{v}_\mathsf{j} \in \bigcup_{\mathsf{r}\geq 1} \mathsf{DAG_i[r]} \wedge \mathsf{v}_\mathsf{j} \in \mathsf{v}_{\mathsf{j}-1}.edges$

3: **procedure** GETANCHOR(r)
4: p $\leftarrow$ GETLEADER(r, activeSchedule) ▷ Any public deterministic function
5: **if** $\exists$v $\in$ DAG[r] s.t. v.$source$ = p **then**
6: **return** v
7: **return** $\perp$

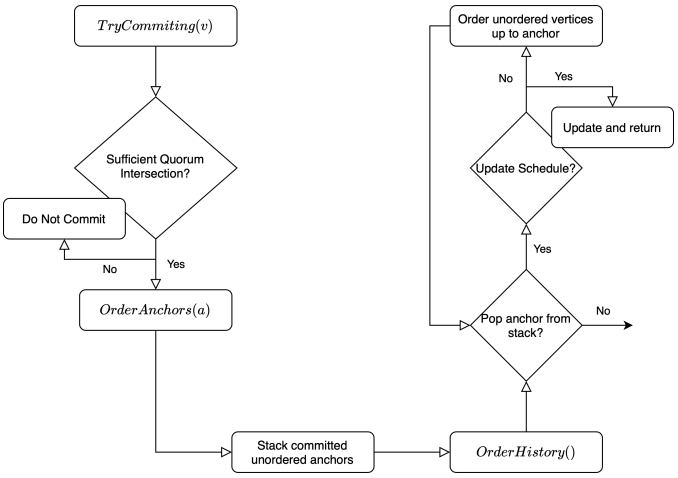

Fig. 2. HammerHead flowchart of operations upon receiving a new vertex u. Party calls TryCommiting(u), which in turn might lead to committing an anchor a, if the party now observes at least $f+1$ for a. In case a new anchor a is committed, party calls OrderAnchors(a), which in turn, constructs a stack of all uncommitted anchors in the causal history of a, with a being at the bottom. Then, upon calling OrderHistory(), anchors are being popped from the stack, until the stack is empty, or until the algorithm returns. In case an anchor triggers a schedule update, then the schedule is updated and the algorithm returns. Else, for each anchor, the party deterministically orders all unordered vertices in its causal history and loops back to popping a new anchor from the stack.

histories are in the DAG. If vertex v_{i+1} is added in the DAG, then $v_{i+1}.edges$ contains vertices that must be already in the DAG (by assumption, since else v_{i+1} would not be added). The combination of all these vertices and all of their causal history form, by definition, the causal history of v_{i+1}. Since all the vertices are already in the DAG, by the strong induction hypothesis, all their causal histories are also in the DAG, and thus v_{i+1}'s causal history is already in the DAG. □

Claim 2. *If an honest party P_i adds a vertex u to its DAG_i, then all honest parties eventually add u to their DAG.*

Proof. Since P_i added u in the DAG, the rules for a vertex to be valid hold for u. From Reliable Broadcast, the rules will hold for u eventually for every honest party. A vertex is added in the DAG if it is in the causal history of a committed anchor. Since P_i added u, then u is in the causal history of an anchor that is committed by P_i. From quorum intersection, this anchor will be in the causal history of any later anchor that will eventually be committed by every other honest party. Thus, u will eventually be added to every honest party's DAG. □

We can thus easily prove that any two honest parties who add vertex u, will have the same causal history for u.

Corollary 1. *If an honest party P_i adds a vertex u to its DAG_i , then every honest party will (i) add u and (ii) upon adding u will have the same causal history for u in its DAG.*

Proof. From Claim 2 if P_i adds u, then every honest party P_j eventually adds u. From Claim 1, P_j will have the entire causal history of u in DAG_j upon adding u. □

Proposition 1 (Schedule Agreement)**.** *Assume that all honest validators eventually switch every schedule according to the schedule switch rule. Then, if an honest validator p_i switches to schedule S, eventually every honest validator will switch to schedule S.*

Proof. Via strong induction. Base case: From S_0 to S_1;
Let S_0 be the very first schedule of the epoch and assume that P_i is the first honest validator who switches from S_0 to say S_1. According to Alg. 2, P_i must have committed some anchor for round $r_i \geq \mathsf{T}$, else the triggering of schedule switch would not occur. Say that another honest validator P_j has so far committed up to round r_j, then $r_j \leq r_i$; Otherwise, if $r_j > r_i$, then according to Alg. 2 P_j would have switched to the next schedule by r_i, which is also S_1 according to the view of P_j, from corollary 1 up to r_i.

So, given that $r_j \leq r_i$, P_j will commit some anchor $a_{r'_j}$ for some round $r'_j > r_i$. Then, from quorum intersection, since P_i committed anchor, say a_{r_i}, in round r_i, there will be a path from $a_{r'_j}$ to a_{r_i}. So, P_j will order a_{r_i}, meaning that P_j will switch schedules. From corollary 1, P_j will switch exactly to S_1, since it will observe the same causal history

Algorithm 2 HammerHead: algorithm for party p_i.

Local variables:
orderedVertices ← {}
lastOrderedRound ← 0 ▷ or lastCommittedRound
orderedAnchorsStack ← initialize empty stack

8: **procedure** TRYCOMMITTING(v)
9: **if** v.*round* mod 2 = 1 or v.*round* = 0 **then**
10: return
11: anchor ← GETANCHOR(v.*round-2*)
12: votes ← v.*edges*
13: **if** $|\{v' \in votes : PATH(v', anchor)\}| \geq f + 1$ **then**
14: ORDERANCHORS(anchor)

15: **procedure** ORDERANCHORS(v)
16: anchor ← v
17: orderedAnchorsStack.*push*(anchor)
18: r ← anchor.*round* − 2
19: **while** r > lastOrderedRound **do**
20: prevAnchor ← GETANCHOR(r)
21: **if** PATH(anchor, prevAnchor) **then**
22: orderedAnchorsStack.*push*(prevAnchor)
23: anchor ← prevAnchor
24: r ← r − 2
25: lastOrderedRound ← v.*round*
26: ORDERHISTORY()

27: **procedure** ORDERHISTORY()
28: **while** ¬orderedAnchorsStack.*isEmpty*() **do**
29: anchor ← orderedAnchorsStack.*pop*()
30: t ← activeSchedule.initialRound + T ▷ T : schedule-change frequency
31: **if** $t \leq$ anchor.round **then**
32: activeSchedule ← UPDATESCHEDULE(anchor)
33: **return**
34: verticesToOrder ← $\{v \in \bigcup_{r>0} DAG_i[r] \mid PATH(anchor, v) \wedge v \notin orderedVertices\}$
35: **for every** v ∈ verticesToOrder in some deterministic order **do**
36: **order** v ▷ output $a\_deliver_i(v.block, v.round, v.source)$
37: orderedVertices ← orderedVertices ∪ {v}

38: **procedure** UPDATESCHEDULE(v)
39: **for** all rounds from activeSchedule.initialRound up to v.round **do**
40: Add 1 to each validator's scores(·) that voted for previous round's leader
41: Compute schedule: the updated schedule according to scores(·)
42: **return** schedule

and thus compute the same schedule as P_i.

Assume that the statement holds for all schedules from S_0 up to S_k. We prove that this holds also for S_{k+1}.
Let P_i be the first honest validator who switches from S_k to S_{k+1}. Then, for each other honest validator, who is in some schedule $S_r : r < k + 1$, we can use the induction hypothesis, which means that each will switch to S_k at some point. According to Alg. 2, P_i must have committed some anchor, say a_{r_i} for round $r_i \geq \mathsf{T} + S_k$.initialRound. Say that another honest validator P_j has so far committed up to round r_j, then $r_j < r_i$, for the same reason as in the base case. Eventually, P_j will switch to S_k and after that, P_j will commit some anchor $a_{r'_j}$ for some round $r'_j > r_i$. Then, from quorum intersection, since P_i committed a_{r_i} in round r_i, there will be a path from $a_{r'_j}$ to a_{r_i}. So, P_j will order a_{r_i}, which means that P_j will switch schedules and, from corollary 1, it will switch to S_{k+1}. □

Claim 3. *Let t be some timestep after GST. If an honest party reliably brodacasts (or delivers) a message m at time t, then all honest parties deliver m by time $t + \Delta$.*

Proof. As also explained in [18], this is satisfied, since before delivering the message, any honest party would multicast it to all other parties and since it is after GST, the message would arrive within Δ time. □

Lemma 1 (View Synchronization)**.** *Let $t_{sync} = GST + \Delta$. Let $S_{\max}$ be the latest schedule any honest party has advanced to before GST. Then, by time t_{sync}, all honest parties can advance up to schedule $S_{\max}$.*

Proof. By time t_{sync} all parties deliver all pre-GST messages. From Claim 1 and the fact that some honest party switched to schedule $S_{\max}$ before GST, it is guaranteed that the causal histories of (…the anchors that upon commit, force switching to…) schedule $S_{\max}$ and all the intermediate schedules, are in DAG_i for every honest party P_i. □

Lemma 2 (View Distance)**.** *After GST, if an honest party enters schedule S then all honest parties will be at some schedule $S' \geq S$ within Δ time.*

Proof. From reliable broadcast, if an honest party delivers sufficient messages to enter schedule S it will broadcast this information to all honest parties, let's say wlog at time t and t is after GST. These messages will be delivered by all honest parties, the latest at $t + \Delta$. An honest party will either ignore the messages because it is already at $S' \geq S$ or enter S. □

Now we will show Liveness in two cases. First we assume that there is no adversarial behaviour and show that honest parties will move from S to $S + 1$ in a bounded number of steps after GST. Then we will show that the only way for the adversary to prevent all parties from collectively advancing schedules is to keep some honest parties ahead. However, to keep those parties ahead the adversary will need to keep advancing schedules, providing Liveness as well.

Lemma 3 (Schedule switch)**.** *Let S be a schedule. After GST, if all honest parties are in schedule S after round $S.\mathsf{initialRound} + \mathsf{T}$, then all honest parties will switch to the next schedule.*

Proof. After GST honest parties are at most Δ away from each other (Lemma 2). Also, all parties are at schedule S. Then, within a bounded amount of time, the parties who are ahead, will be in round $\geq S.\mathsf{initialRound} + \mathsf{T}$. They either commit the new anchor and switch schedules, or they cannot. If they switch, then by Lemma 2 all honest parties will switch within Δ. Else, within Δ all honest parties will be caught up and they will all be able to commit, so they all switch together. □

Lemma 4 (Liveness)**.** *Let $S_{\max}$ be the latest schedule any honest party has advanced to before GST. Within a bounded number of steps some honest party will enter $S' = S_{\max} + 1$*

Proof. From view synchronization, every honest party will be at $S_{\max}$ at $\mathsf{GST} + \Delta$. Now there are two cases. First, if some honest party moves to $S_{\max} + 1$ then by view distance all honest parties will move to $S_{\max} + 1$ within Δ. So Liveness is proven. Else all honest parties will be at $S_{\max}$ and from Bullshark Liveness will succesfully advance $\geq \mathsf{T}$ rounds. Thus, from Schedule switch, they will all switch to schedule $S_{\max} + 1$. □

The following claim is used to prove Total Order.

Claim 4. *Let honest parties P_i, P_j commit anchors a_1^i, a_1^j in rounds r_1^i, r_1^j respectively such that they are on the same schedule. Similarly let P_i, P_j commit anchors a_2^i, a_2^j in rounds $r_2^i > r_1^i, r_2^j > r_1^j$ respectively. Then, P_i and P_j order the same vertices in the same order between rounds $\max\{r_1^i, r_1^j\}$ and $\min\{S, r_2^i, r_2^j\}$, where S is the round when the next schedule change occurs.*

Proof. Assume there is an overlap of rounds, else the claim trivially holds. Then, assume wlog that $r_1^i \leq r_1^j < r_2^i \leq r_2^j$. Once a party commits its second anchor, it backtracks from it until it reaches a previously committed anchor and then moves forward ordering all vertices until it either reaches a schedule change, or the latest committed anchor. Let the next schedule change occur in round $S > \max\{r_1^i, r_1^j\}$; if $S \leq \max\{r_1^i, r_1^j\}$, the claim again holds trivially, since there is no round overlap. Thus, if $S > \max\{r_1^i, r_1^j\}$, both parties will order vertices at least between rounds $\max\{r_1^i, r_1^j\}$ and $\min\{S, r_2^i, r_2^j\}$. During this time they will both operate in the same schedule by proposition 1. Since from corollary 1 they both observe the same DAG then, during that span they will order the same vertices in the same order. □

Lemma 5 (HammerHead-BAB)**.** *HammerHead satisfies Byzantine Atomic Broadcast per Definition 2.*

Proof. (*Total Order*) Follows from inductively applying Claim 4 for every pair of honest parties, over all pairs of consecutive commits.

(*Agreement*) Directly from corollary 1. Let some honest party P_i call $a\_deliver(\mathsf{v}.block, \mathsf{v}.round, \mathsf{v}.source)$. Then, by construction this means that P_i committed some anchor a that contained v in its causal history. From corollary 1, all honest parties will eventually commit a and will observe the same causal history, so they all eventually order v and output $a\_deliver(\mathsf{v}.block, \mathsf{v}.round, \mathsf{v}.source)$.

(*Validity*) Let honest P_i call $a\_bcast(m, r)$, then we need to show that eventually every honest party outputs $a\_deliver(m, r, k)$. From Liveness, eventually party P_i will include m in a vertex v_i and reliably broadcast v_i. From Reliable Broadcast's validity, all honest parties will eventually receive v_i and add it to their DAG. From Liveness, there will eventually be a round after GST for which the leader, say P_j, is honest. Then, by construction, P_j will commit an anchor for which v_i will be in its causal history. Thus, eventually all honest parties will order v_i and by result will call $a\_deliver(m, r, k)$.

(*Integrity*) Integrity follows trivially from the Integrity of Reliable Broadcast. □

Lemma 6 (Leader Utilization)**.** *HammerHead satisfies Leader Utilization per definition 3. Specifically, the number of rounds r for which no honest party commits a vertex formed in r is bounded by $O(\mathsf{T}) \cdot f$.*

Proof. After GST, a crashed node will not cast votes. As a result from the calculation of reputation scores, it will be in the B set the latest $O(\mathsf{T})$ rounds after it crashed and will not get in the G set for as long as it is crashed. Therefore, the number of rounds for which no honest party commits a vertex is bounded by $O(\mathsf{T})$ for each of the up to f crashed leaders. □

IV. Implementation

We implement a networked multi-core HammerHead validator in Rust forking the Narwhal-Bullshark implementation of Sui[4]. We select this codebase because it is the only production-ready implementation of a DAG-based consensus protocol deployed in the real world (at the best of our knowledge). It uses Tokio[5] for asynchronous networking, fastcrypto[6] for elliptic curve based signatures. Data-structures are persisted using RocksDB[7]. We use QUIC[8] to achieve reliable authenticated point-to-point channels. By default, this Narwhal-Bullshark implementation uses traditional round-robin to elect leaders; we modify its leader election module to use HammerHead instead. Implementing our mechanism requires adding less than 600 LOC (+ 400 LOC of tests), and does not require any extra protocol message or cryptographic tool. Contrarily to most prototypes, our implementation is *production-ready* and *fully-featured* (crash-recovery, monitoring tools, etc). It runs at the heart of the Sui mainnet since version `mainnet-v1.9.1` [9]. We open source our implementation of HammerHead[10].

V. Evaluation

We evaluate the throughput and latency of HammerHead through experiments on Amazon Web Services (AWS). We then show its improvements over the baseline round-robin leader-rotation mechanism of Bullshark [18]. We aim to demonstrate the following claims.

- **C1:** HammerHead introduces no throughput loss and even provides small latency gains when the protocol runs in ideal conditions (faultless setting).
- **C2:** HammerHead drastically improves latency and throughput in the presence of crash-faults; and its benefit increases with the number of faults.
- **C3:** HammerHead does not suffer from any visible throughput degradation despite (crash-)faulty validators. Note that evaluating BFT protocols in the presence of Byzantine faults is an open research question [1].

Experimental setup. We deploy our fully-featured HammerHead testbed on AWS, using `m5d.8xlarge` instances across 13 different AWS regions: N. Virginia (us-east-1), Oregon (us-west-2), Canada (ca-central-1), Frankfurt (eu-central-1), Ireland (eu-west-1), London (eu-west-2), Paris (eu-west-3), Stockholm (eu-north-1), Mumbai (ap-south-1), Singapore (ap-southeast-1), Sydney (ap-southeast-2), Tokyo (ap-northeast-1), and Seoul (ap-northeast-2). Validators are distributed across those regions as equally as possible. Each machine provides 10Gbps of bandwidth, 32 virtual CPUs (16 physical corest) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and runs Linux Ubuntu server 22.04. HammerHead persists all data on the NVMe drives provided by the machine (rather than the root partition). We select these machines because they provide decent performance and are in the price range of 'commodity servers'.

In the following graphs, each data point is the average of the latency of all transactions of the run, and the error bars represent one standard deviation (errors bars are sometimes too small to be visible on the graph). We instantiate several geo-distributed benchmark clients submitting transactions at a fixed rate for a duration of 10 minutes; each benchmark client submits at most 350 tx/s and the number of clients thus depends on the desired input load. The transactions processed by both systems are simple increments of a shared counter. The leader-reputation schedule is recomputed every 10 commits and excludes the 33% less performant Validators [11] . When referring to *latency*, we mean the time elapsed from when the client submits the transaction to when it receives confirmation of the transaction's finality. When referring to *throughput*, we mean the number of *distinct* transactions over the entire duration of the run.

In addition to our codebase, we also open-source all orchestration and benchmarking scripts as well as measurements data[12] to enable reproducible evaluation results. Appendix A provides a tutorial to reproduce our experiments.

Benchmark in ideal conditions. Figure 3 compares the performance of the baseline Bullshark and HammerHead running with 10, 50, and 100 honest validators. Regardless of the committee size, the performance of Bullshark is similar to HammerHead. We observe a peak throughput around 4,000 tx/s (for committee sizes of 10 and 50) and 3,500 tx/s (for a committee size of 100) for both systems. The latency of Bullshark is slightly higher than HammerHead, at 3 seconds while HammerHead provides a latency of 2.7 seconds. This small latency gains is due to HammerHead's added benefit to focus on electing performant leaders. Leaders on more remote geo-locations that are typically slower are elected less often, the protocol is thus driven by the most performant parties. These observations validate our claim **C1** stating that HammerHead introduces no throughput loss and provides small latency gains when the protocol runs in ideal conditions.

Benchmark with faults. Figure 4 compares the performance of Bullshark and HammerHead when a committee of 10, 50, and 100 validators respectively suffers 3, 16, and 33 crash-faults (the maximum that can be tolerated).

Bullshark suffers a massive degradation in both throughput and latency. For committee sizes of 10 and 50 suffering respectively 3 and 16 faults, the throughput of Bullshark drops by 25% and its latency increases by 2-3x compared to ideal conditions. In contrast, HammerHead only suffers a slight latency degradation (at most 0.5 second) due to a smaller pool of leaders to elect from. Notably, HammerHead does not suffer

[4]https://github.com/mystenlabs/sui
[5]https://tokio.rs
[6]https://github.com/MystenLabs/fastcrypto
[7]https://rocksdb.org
[8]https://github.com/quinn-rs/quinn
[9]https://github.com/MystenLabs/sui/releases/tag/mainnet-v1.9.1
[10]https://github.com/asonnino/sui/tree/hammerhead (commit `03c96a3`)

[11]Mainnet Sui uses more conservative parameters: it recomputes the schedule every 300 commits and only excludes the bottom 20% of validators.
[12]https://github.com/asonnino/hammerhead-paper/tree/main/data

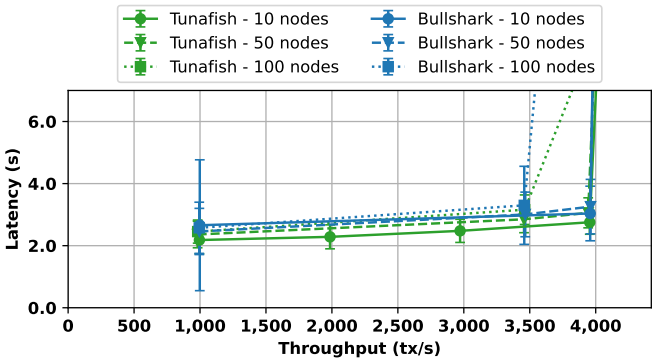

Fig. 3. HammerHead and Bullshark latency-throughput performance with 10, 50, and 100 validators (no faults).

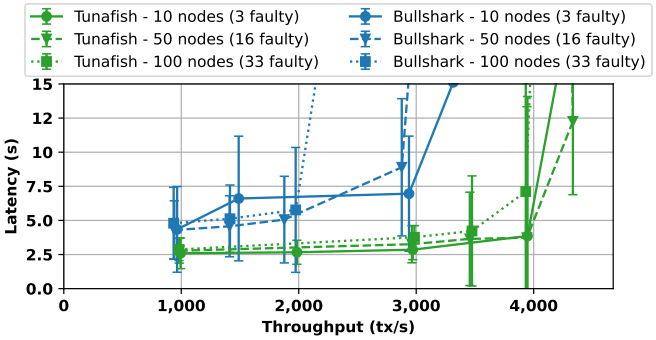

Fig. 4. HammerHead and Bullshark performance with 10, 50, and 100 validators when experiencing their respective maximum number of tolerable faults.

from any throughput degradation: it does not elect crashed leaders, the protocol continues to operate electing leaders from the remaining active parties, and is not overly affected by the faulty ones. This validates our claim **C2**.

The performance benefits of HammerHead are even more drastic for larger committees: for a committee size of 100 suffering 33 faults, the throughput of Bullshark drops by over 40% and its latency increases 2x compared to ideal conditions. In contrast, HammerHead once again does not suffer from any throughput degradation and has only a slight latency increase. We thus observe that HammerHead provides a 2x latency reduction and a throughput increase ranging from 25% (small committees) to 40% (large committees) with respect to Bullshark. This validates our claim **C3**.

VI. Production Deployment

We closely collaborated with the Sui team to integrate HammerHead into the Sui Mainnet. We present an overview of the roadmap for the complete integration of HammerHead into the Sui mainnet along with timelines to provide an insight on the size of such task. Despite the apparent simplicity of its algorithm, HammerHead brings substantial modifications to critical blockchain components, necessitating a comprehensive roadmap before its production deployment. This journey involved over four months of engineering effort.

The first milestone involved the implementation of the core reputation mechanism, which computes a reputation score for each validator and incorporates it into various non-critical aspects of the system, including a separate control system overseeing transaction submissions to consensus. This feature was seamlessly integrated ahead of the Sui mainnet launch as it could not compromise the safety of the system. Following this, we conducted extensive testing over several months to ensure that our chosen reputation metrics accurately reflected real-world performance. We have maintained continuous monitoring since the inception of the Sui mainnet, a period spanning approximately four months. Next, we harnessed these reputation scores to fine-tune the leader schedule. This involved rigorous testing within private deployments, followed by rigorous evaluation in the devnet and testnet environments, encompassing both load and failure testing. This phase consumed approximately 1.5 months. With the confidence gained from successful private and test deployments, HammerHead was made publicly available and ran for a month in the devnet and testnet environments. Afterwards, it was fully integrated into the mainnet codebase., albeit gated through a protocol configuration and initially turned off. Finally, the switch was initiated and HammerHead was incorporated as a pivotal component of mainnet version 1.9.1[13], corresponding to Sui protocol version 23, marking the culmination of this meticulous integration process.

VII. Related Work

Carousel [8] presents the first reputation-based leader-rotations mechanisms for SMR protocols providing Leader Utilization. It specifically targets chained consensus protocols [2], [4], [6], [7], [9], [14], [23] and its main challenge thus lies in achieving Chain Quality [13], which entails limiting the number of committed blocks proposed by Byzantine validators. In contrast, HammerHead is tailored for DAG-based consensus protocols [3], [10], [12], [15]–[18], [22] and thus encounters distinct challenges. Unlike chained consensus protocols, DAG-based protocols do not preserve safety when validators disagree on the identity of the leader. As a result, HammerHead cannot simply leverage the state of every view to recompute the reputation scores because different validators may commit the same block in different views. This distinction necessitates that we open the black-box of the DAG and adapt our interpretation of it to ensure both safety and liveness. On the positive side, using a DAG eliminates the need to be concerned about Chain Quality as HammerHead directly inherits it from underlying DAG, even if all leaders are malicious. Consequently, HammerHead forgoes the need to ensure that honest leaders make sufficiently frequent proposals.

One extreme scenario we also explored is that of the classic static leader that pre-blockchain BFT protocols used (e.g., PBFT [5]), however, the risk of having a leader that performs just slow enough to not cause a gap in the schedule (and a subsequent "schedule change") is too great for the slight

[13]https://github.com/MystenLabs/sui/releases/tag/mainnet-v1.9.1

benefits of having a above average performance leader more often. We leave an open question if we can have a small subset of active leaders or a more adaptive reputation scoring mechanism to exploit the most performant nodes as leaders more often.

The concurrent work Shoal [17] is the closest system to HammerHead. Shoal's primary objective is to lower the latency associated with DAG-based consensus, employing various strategies that include a leader-reputation mechanism like HammerHead. Similar to HammerHead, Shoal's leader-reputation mechanism maintains a record of scores for each validator and employs a deterministic rule to recalibrate the mapping from rounds to leaders based on these scores. Shoal conceptually leaves open the choice of this deterministic rule and its implementation assigns higher scores to committed leaders and lower scores to leaders that were skipped. Conversely, HammerHead assigns scores based on the frequency of votes for leaders, discouraging Byzantine actors from withholding their votes for honest leaders. Shoal and HammerHead however mostly diverge in their areas of emphasis. Shoal takes a broader perspective, focusing on reducing the latency of DAG-based consensus through additional techniques like consensus pipelining and prevalent responsiveness [17], while HammerHead entirely focuses on leader-reputation, offering detailed algorithms and formal security proofs.

VIII. Conclusions

This paper introduces HammerHead, a novel leader-aware SMR custom-designed for DAG-based consensus protocols. Drawing inspiration from Carousel and harnessing on-chain metrics, HammerHead achieves high leader utilization. To achieve this it addresses the unique challenges posed by DAG structures, where block commitments lack synchronization across all nodes, by reinterpreting the DAG to ensure both safety and liveness. HammerHead's dynamic leader schedule adjustment, based on validator activity and reliability, optimizes leader selection while preserving system safety. This approach ensures sustained performance and throughput even in the presence of crash faults, outperforming existing leader-based protocols like Bullshark. In summary, HammerHead's implementation showcases its robustness in various scenarios, emphasizing the critical importance of leader-awareness in such systems.

Acknowledgements

This work is supported by Mysten Labs. We thank the Mysten Labs Engineering teams for valuable feedback broadly, and specifically to Laura Makdah for helping implementing the early reputation score system for validators and Dmitry Perelman for managing the overall implementation effort.

References

[1] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: Bft systems made robust. In *25th International Conference on Principles of Distributed Systems*, 2022.

[2] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 1(1), 2019.

[3] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. Sui lutris: A blockchain combining broadcast and consensus. Technical report, Technical Report. Mysten Labs. https://sonnino. com/papers/sui-lutris. pdf, 2023.

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

[5] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, 1999.

[6] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

[7] Junchao Chen, Suyash Gupta, Alberto Sonnino, Lefteris Kokoris-Kogias, and Mohammad Sadoghi. Resilient consensus sustained collaboratively. *arXiv preprint arXiv:2302.02325*, 2023.

[8] Shir Cohen, Rati Gelashvili, Lefteris Kokoris Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. Be aware of your leaders. In *International Conference on Financial Cryptography and Data Security*, pages 279–295. Springer, 2022.

[9] Shir Cohen, Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Proof of availability and retrieval in a modular blockchain architecture. In *Financial Cryptography and Data Security*, pages 36–53, 2024.

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

[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[12] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1187–1201, 2022.

[13] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.

[14] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International Conference on Financial Cryptography and Data Security*, pages 296–315. Springer, 2022.

[15] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, page 165–175, 2021.

[16] Dahlia Malkhi and Pawel Szalachowski. Maximal extractable value (mev) protection on a dag. *arXiv preprint arXiv:2208.00940*, 2022.

[17] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.

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

[19] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: The partially synchronous version. *arXiv preprint arXiv:2209.05633*, 2022.

[20] The Aptos team. https://aptoslabs.com, 2023.

[21] The Sui team. http://sui.io, 2023.

[22] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, 2022.

[23] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

APPENDIX

We provide the orchestration scripts[14] used to benchmark the codebase evaluated in this paper on AWS .

Deploying a testbed. The file '~/.aws/credentials' should have the following content:

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY_ID
aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
```

configured with account-specific AWS *access key id* and *secret access key*. It is advise to not specify any AWS region as the orchestration scripts need to handle multiple regions programmatically.

A file 'settings.json' contains all the configuration parameters for the testbed deployment. We run the experiments of Section V with the following settings:

```
{
    "testbed_id": "${USER}-tunafish",
    "cloud_provider": "aws",
    "token_file": "/Users/${USER}/.aws/credentials",
    "ssh_private_key_file": "/Users/${USER}/.ssh/aws
        ",
    "regions": [
        "us-east-1",
        "us-west-2",
        "ca-central-1",
        "eu-central-1",
        "ap-northeast-1",
        "ap-northeast-2",
        "eu-west-1",
        "eu-west-2",
        "eu-west-3",
        "eu-north-1",
        "ap-south-1",
        "ap-southeast-1",
        "ap-southeast-2"
    ],
    "specs": "m5d.8xlarge",
    "repository": {
        "url": "https://github.com/AUTHOR/REPO.git",
        "commit": "tunafish"
    }
}
```

where the file '/Users/$USER/.ssh/aws' holds the ssh private key used to access the AWS instances, and 'AUTHOR' and 'REPO' are respectively the GitHub username and repository name of the codebase to benchmark.

The orchestrator binary provides various functionalities for creating, starting, stopping, and destroying instances. For instance, the following command to boots 2 instances per region (if the settings file specifies 13 regions, as shown in the example above, a total of 26 instances will be created):

```
cargo run --bin orchestrator -- testbed \
    deploy --instances 2
```

The following command displays he current status of the testbed instances

```
cargo run --bin orchestrator testbed status
```

Instances listed with a green number are available and ready for use and instances listed with a red number are stopped. It is necessary to boot at least one instance per load generator, one instance per validator, and one additional instance for monitoring purposes (see below). The following commands respectively start and stop instances:

```
cargo run --bin orchestrator -- testbed start
cargo run --bin orchestrator -- testbed stop
```

It is advised to always stop machines when unused to avoid incurring in unnecessary costs.

Running Benchmarks. Running benchmarks involves installing the specified version of the codebase on all remote machines and running one validator and one load generator per instance. For example, the following command benchmarks a committee of 100 validators (none faulty) under a constant load of 1,000 tx/s for 10 minutes (default), using 3 load generators:

```
cargo run --bin orchestrator -- benchmark \
    --committee 100 fixed-load --loads 1000 \
    --dedicated-clients 3 --faults 0
    --benchmark-type 100
```

The parameter `benchmark-type` is set to 100 to instruct the load generators to sequence all transactions through the consensus engine. We select the number of load generators by ensuring that each individual load generator produces no more than 350 tx/s (as they may quickly become the bottleneck).

Monitoring. The orchestrator provides facilities to monitor metrics. It deploys a Prometheus instance and a Grafana instance on a dedicated remote machine. Grafana is then available on the address printed on stdout when running benchmarks with the default username and password both set to `admin`. An example Grafana dashboard can be found in the file 'grafana-dashboard.json'[15].

Troubleshooting. The main cause of troubles comes from the genesis. Prior to the benchmark phase, each load generator creates a large number of gas object later used to pay for the benchmark transactions. This operation may fail if there are not enough genesis gas objects to subdivide or if the total system gas limit is exceeded. As a result, it may be helpful to increase the number of genesis gas objects per validator in the 'genesis_config' file[16] when running with very small committee sizes (such as 10).

[14] https://github.com/asonnino/sui/tree/hammerhead (commit `03c96a3`)

[15] https://github.com/asonnino/sui/blob/hammerhead/crates/orchestrator/assets/grafana-dashboard.json

[16] https://github.com/asonnino/sui/blob/03c96a3648f40f89bd78930b837aa1393bab73ec/crates/sui-swarm-config/src/genesis_config.rs\#L360