The Espresso Sequencer: HotShot Consensus and Tiramisu Data Availability

Espresso Systems

September 7, 2023

Abstract

Layer-2 (L2) rollups are a popular approach for scaling Layer-1 (L1) blockchains. Rollups move the transaction processing off-chain, while the L1 only checkpoints the rollup state. This design leaves open the question of which transactions are rolled up and in what order. Unfortunately, all of the current rollups use their own centralized *sequencers* for ordering transactions. This leads to two caveats: (i) the centralized sequencer is a single point of failure, and (ii) applications from different L2 ecosystems are harder to interoperate.

We introduce Espresso Sequencer, a decentralized network that can be shared by all of the L2 rollups. Our design consists of two key components, HotShot Consensus and Tiramisu data availability that are modularly separated to handle the two key tasks of a sequencer — ordering transactions and ensuring data availability. HotShot is an optimistically responsive, communication-efficient consensus protocol in a proof-of-stake setting that is resistant to bribing adversaries and scalable to a large number of nodes. Our layered Tiramisu data availability protocol combines the use of verifiable information dispersal and small random committees to ensure data availability with linear communication complexity. Both of our protocols allow the use of a content distribution network at the networking layer that unlocks Web2 performance in the optimistic case while still providing strong Web3 security guarantees in the pessimistic case.

Contents

1 Introduction

Ethereum is a popular blockchain system supporting general purpose smart contracts enabling many exciting applications including decentralized finance (DeFi) [130, 36, 118], non-fungible tokens (NFTs) [117], decentralized gaming [29], etc. However, while the system offers good decentralization and economic security, it suffers from a poor transactions throughput of ∼25 transactions per second and a latency of ∼13 minutes.

More recently, *layer-2 or L2* solutions [66, 119, 109] such as rollups have been introduced to improve the transaction processing performance on top of Ethereum. The key idea in a rollup is to move transaction processing off-chain while only maintaining a state checkpoint contract on Ethereum. To achieve this, an *L2 sequencer* collects and orders user transactions. The ordered transactions are executed by the rollup and an updated state is stored on the L1 using a rollup contract on the L1. The correctness of the state transition is guaranteed through a validity proof in zkRollups [119], or a fraud proof in optimistic rollups [66]. This approach amortizes the gas cost for the execution and storage over all of the batched transactions, thus enabling much higher throughput and lower gas fees. Moreover, if the users trust the L2 operator, they can optimistically confirm their transactions before seeing the rollup contract state change, thus improving latency as well.

The need for a shared decentralized sequencer. Unfortunately, all the current L2 rollups today use their own *centralized* sequencers to collect and order L2 transactions. This leads to two fundamental caveats that undermine the major benefits that Ethereum brought in the first place. First, the centralized sequencer is a single point of failure. If it goes down, the entire rollup ceases to function. This is not just theoretical but has happened in practice [133]. Even if it does not crash, we end up placing an enormous amount of trust for it to not equivocate on the ordering, censor or block users' transactions or simply increase fees to monopolistic levels. Thus, L2 users no longer share the same level of security as they had on Ethereum. Second, one of the most attractive features of Ethereum is its composability and interoperability over multiple applications. However, a different sequencer for each L2 system makes interactions across these L2 ecosystems much harder. With a shared sequencer we can support atomic execution of a set of transactions belonging to multiple rollups, in order to leverage some cross-rollup arbitrage opportunity without risk. In practice, a block builder not respecting the atomicity requested by a user can be made publicly accountable and slashed. This is clearly not possible when sequencers are different for each rollup. A shared sequencer also opens up a new design space of cross rollup applications such as flashloans by introducing participants willing to take calculated risks in exchange for fees. In such setting, the additional participant carries all the consequence of a misbehaving builder but the rollup security is not affected. This leads us to the following natural question:

> *Can we build a decentralized but highly performant sequencer that can be shared by all the L2 rollups?*

We answer this question affirmatively through the Espresso sequencer network. To understand the properties of our sequencer, we first need to understand what it means to be *decentralized but highly performant.* At its core, *decentralization* implies that we need consensus among a set of sequencer nodes while having the ability to tolerate realistic and powerful adversaries, and *performant* implies that the sequencer can handle high throughput with a low commit latency.

A sequencer faces fewer design constraints than an L1. Design constraints for state machine replication (SMR) systems such as an L1 include the following three aspects. Each of these is a potential bottleneck for an SMR.

(Data broadcast) Nodes must broadcast the data (transactions, commands, etc.) to all the other nodes.

(Consensus and ordering) Nodes must reach consensus on the order of transactions.

(Execution) Nodes must execute these transactions on their respective state machines.

A sequencer, by contrast, does not execute transactions. Instead, execution is typically guaranteed by ZK or fraud proofs produced by the rollups themselves. A consequence is that a sequencer also need not *validate* transactions, and so transaction data need not be broadcast to all the nodes. Instead, it suffices that a sequencer guarantee only *availability* of transaction data, meaning this data can be retrieved in the future by anyone who needs it. Thus, a sequencer only needs to solve this relaxed data availability problem and the consensus problem. The Espresso sequencer addresses both of these problems modularly through HotShot Consensus and Tiramisu Data Availability protocols.

1.1 HotShot: A High-Throughput Low-Latency Scalable Consensus

The distributed computing literature has witnessed a large number of consensus protocols secure under different network conditions, under different types of adversaries, and obtaining a subset of other desirable properties (e.g., dynamic availability, accountability, responsiveness). Compared to all of these approaches, in designing HotShot, a key design principle is to secure against a strong adversary and simultaneously aim for high efficiency in the optimistic case. This is motivated by both theory and practice, as attacks and network outages are usually temporary events, and we can set incentives to ensure that the benefits to the adversary of a long-lived attack do not justify its cost. More precisely, from a security standpoint, a key goal is for the network to be as secure as the Ethereum network itself. This implies that we need to achieve the high decentralization of running the protocol on a large number of nodes (or Ethereum stakers). Moreover, the protocol needs to be safe and live despite bribery attacks against the stakers [19, 84, 116]. In terms of performance, on the other hand, our goal is to obtain a protocol with very high throughput, very low latency, and ideally with responsiveness. Throughput corresponds to the total transactions that can be sequenced, latency corresponds to the delay between the submission of a transaction to the network and its sequencing, and the responsiveness property signifies a sequencer that can move as fast as the network allows it, e.g., does not have fixed block times. In other words, our performance complements that provided by Ethereum and, at the same time, attempts to reach that of a centralized sequencer.

In more detail, our consensus protocol adapts a permissioned Byzantine Fault Tolerant (BFT) protocol to the proof-of-stake setting. This enables us to support dynamic network participants that can freely join and leave protocols by bonding or unbonding stake. The protocol should satisfy safety and liveness so long as more than two-thirds of the total amount of stake is controlled by honest nodes. As mentioned earlier, to achieve full decentralization, we need to support tens of thousands of staked consensus nodes. Traditional approaches usually combine a permissioned BFT protocol with proof-of-stake via committee sampling, where a small random committee will represent the entire set of staked users to run the consensus. However, this type of scheme usually suffers from adaptive attacks, where an adversary, that can control only a small amount of stake, can still corrupt the elected committee and break the security of consensus. There do exist solutions (e.g., Algorand [59], YOSO [58]) secure against adaptive adversaries. The best known defense is to hide the elected committee until they published their votes. Thus, an adaptive adversary cannot change the committee's behavior after the fact. However, a bribery attack is still a practical concern for this solution, where a malicious adversary, even without knowing who to corrupt, can advertise payouts for certain verifiable malicious behaviors [12], e.g., the attacker can create a smart contract that pays elected committee members to censor specific transactions. Thus, to achieve the desired efficiency and scalability without losing bribery resistance, we let all staked nodes participate in the consensus protocol, and use HotStuff-2 [82] as the underlying BFT protocol, which achieves linear communication complexity given a pacemaker module that enables nodes to enter the same view for a sufficient amount of time. [Kartik: in the future, need to smoothen this part.] However, some pacemaker instantiations such as the one in the original HotStuff [125] incur quadratic communication complexity and sub-optimal latency. We thus leverage the view sync protocol from Naor and Keidar [90] to achieve linear communication complexity in the optimistic case. This allows our consensus to scale to thousands of physical nodes, which also makes bribery highly costly. Meanwhile, the protocol achieves an important property called *optimistic responsiveness*, meaning that it can achieve close performance to centralized systems when network conditions are favorable. Moreover, we build efficient primitives (e.g., aggregated quorum certificates, stake table and decentralized random beacons) to adapt the permissioned HotStuff-2 protocol to the fully decentralized setting without performance deterioration.

1.2 Tiramisu: The Three-Layered Data Availability Solution

State machine replication (SMR) requires that a data payload B of size $|B|$ be broadcast to all n nodes on the network. Observe that $O(|B|\,n)$ communication is a trivial lower bound on this problem, and it can easily be solved with $O(|B|\,n^2)$ communication by using a consensus protocol to agree on the contents of B.

However, as described earlier in Espresso's sequencer network, our goal is only to ensure *availability* of data, not to *broadcast* the data. This relaxation of SMR can be achieved with vastly improved communication complexity $O(|B|)$.[1]

Our data availability solution, Tiramisu, combines three novel ideas to ensure DA security without sacrificing efficiency. These ideas can be viewed as layers, mirroring the layers of the timeless Italian dessert.

At the base, the *Savoiardi* layer, uses a newly-introduced variant of the verifiable information dispersal (VID) scheme from [10] to gurantee data availability. The idea is to encode the data block into erasure-coded chunks and send one chunk per node. The nodes that receive valid chunks will return an ack signature, and the consensus leader can proceed as long as it receives ack signatures from a quorum of nodes. The Savoiardi VID layer is highly communication efficient and secure. To corrupt it, even a bribing adversary needs to control more than a 1/3-fraction of the stake in the system, which is as hard as breaking the HotShot consensus protocol.

A disadvantage of Savoiardi is its performance. A client who wishes to recover the full payload must download shares from many storage nodes and spend computation resources to decode the payload. To remedy this, we introduce the *Mascarpone* layer. This layer enables fast reconstruction and access to subsets of data. We utilize the HotShot random beacon to elect a small random committee per epoch. Each node in this committee receives the entire data block (not just chunks of it). A valid DA certificate, and thus a valid HotShot block, must include a signature by a threshold, e.g. 80% of the committee. This ensures that with high probability, one honest node will be part of the quorum and can provide fast access to the data and aid reconstruction. The random selection ensures that the committee can be small, but unfortunately, this also makes it more vulnerable to briberies. The Savoiardi layer does not have this vulnerability and provides the strongest security. Combining Savoiardi and Mascarpone gives both strong security and fast reconstruction.

Finally, we can optimistically increase the performance of the system by adding a content delivery network (CDN), which we call the Cocoa layer. The CDN can cache and efficiently distribute data, and can be thought of as an efficient broadcast layer. It can deliver performance on the level of traditional Web2 infrastructure, however it is a centralized component that does not deliver any security guarantees. This is not a security issue, as the CDN is entirely optional and is backed up by the Mascarpone and Savoiardi layers. However, if the CDN is online, it can significantly improve the performance of those layers and the system overall, by quickly disseminating data and providing fast access to it. The Cocoa layer works well with our *optimistically responsive* design principle where the network works faster when all components are fully working and online, but still gives high-security guarantees in the presence of an adversary.

1.3 Architecture Overview

In this section, we briefly describe the main components of the Espresso Sequencer, as well as their interactions with users, L2 rollups and the L1 blockchain (e.g., Ethereum).

The Espresso Sequencer is a decentralized network of thousands of heterogeneous nodes providing *log replication* (a.k.a. transaction sequencing) as a service. Internally, it consists of two core components:

1. The *HotShot Consensus layer* that provides a sequential ordering of the submitted transactions (but no execution). It is a permissionless, leader-based, multi-shot BFT protocol that extends HotStuff-2 [82] to the proof-of-stake setting. We refer to Section 2 for more details.

2. The *Tiramisu DA layer* that guarantees the availability of the data submitted by clients. More specifically, a set of storage nodes are responsible for the availability and retrievability of raw transaction data. Since the sequencer is not responsible for execution, we note that the data need not be broadcast to every full node; this is essential for achieving better throughput. We refer to Section 3 for more details.

As shown in Figure 1, the Espresso Sequencer actively interacts with users, L2 rollups and the L1 blockchain.

- **Layer 2 (L2) rollups.** These are off-chain execution engines (VMs) that accept users' transactions and deterministically process them after being ordered and finalized by the

[1]Strictly speaking, this improved communication complexity is $O(|B| + n)$, but in any realistic setting we expect n to be $O(|B|)$.

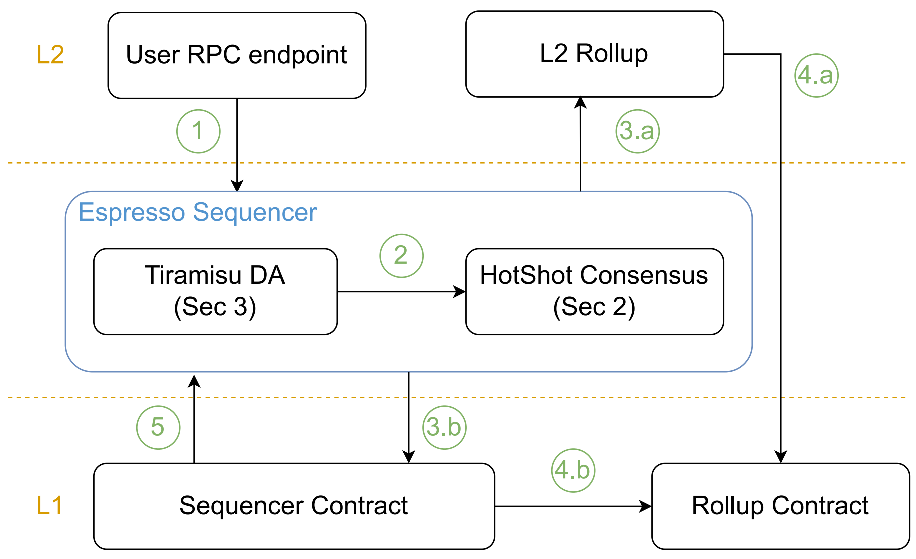


Figure 1: Main components in the HotShot protocol and the typical transaction flow.

Espresso Sequencer. Their execution logic could be anything from app-specific to fully-featured smart contract platforms (like EVM rollups). Furthermore, the prover network as a subcomponent will periodically update the state commitments in the rollup contract on L1, along with a *validity proof* (for zkRollups) or a potential *fraud proof* (for optimistic rollups).

- **Layer 1 (L1) blockchain.** The Espresso Sequencer checkpoints the ledger state to this blockchain. Its primary function is to serve as an always-online, minimally trusted verification light client for the HotShot Consensus layer. When the L1 is more mature with wider adoption and higher economic cost for forking/reorg, these checkpoints also provide a defense in depth on long-range attacks on HotShot consensus. Internally, there is one *sequencer contract* that logs the finalized ledger produced by the HotShot sequencers; and one *rollup contract* per L2 rollup that reads the ledger state from the sequencer contract and maintains its rollup-specific states.

The life cycle of the transactions submitted by L2 rollups is described as follows (see Figure 1).

1. Users submit raw transactions to the Tiramisu DA mempool through an RPC endpoint/gateway. The RPC endpoint/gateway can be run anywhere by anyone. Each transaction also attaches a namespace ID to indicate which L2 rollup it belongs to.

2. A leader in the DA layer collects the transaction data for possibly different rollups, and disperses the data to the Tiramisu DA layer to obtain a certificate for data availability. Note that the formation of a DA certificate does not necessarily require every storage node to obtain the full data. Meanwhile, a leader in the HotShot Consensus layer broadcasts a proposal containing the *short* vector commitment to the transaction data. The nodes in HotShot Consensus would vote and finalize the block commitment. Note that the nodes participating HotShot Consensus are also storage nodes in the DA layer, and the finalization of a block commitment would also signify the data availability of the corresponding transaction data.

3. When a block is considered finalized in the HotShot consensus protocol, the nodes in the Espresso Sequencer would propagate the new block commitment (plus quorum certificates) to the sequencer contract on L1 (Step 3.b in Figure 1), after which the sequencer contract verifies the certificate and updates the checkpointed ledger state. Note that the sequencer contract maintains information about the current Espresso configuration and stakers and thus is able to verify the committed new state. Concurrently, the DA nodes are disseminating the multi-rollups block content to the L2 rollup provers who can extract their own set of ordered transactions (Step 3.a). We emphasize our design decision of moving data dissemination out of the critical path of consensus for lower latency and higher throughput. The dissemination of the full block to clients (e.g., the rollup provers) does not necessarily complete even if the block has already been finalized, and only a minimally necessary bandwidth is used to ensure data availability on the critical path.

4. The prover upon receiving the new block, will compute an updated state and its commitment

and generate a state-transition proof which would be posted on the rollup contract on L1 as a verifiable state checkpoint (Step 4.a). To verify the L2 state-transition proof, the rollup contract will also read the updated Espresso ledger state from the sequencer contract (Step 4.b).

Finally, the Espresso network nodes periodically read from the L1 sequencer contract the updated version of the stake table [Kartik: should we specify an approximate duration or so?] to obtain a global view for the set of staking participants (Step 5). This usually happens during transition between epochs where the set of staking participants reconfigures. These checkpoints provide a defense in depth against long-range attacks on HotShot consensus.

1.4 Key Contributions

In this section, we present key contributions of HotShot Consensus and Tiramisu DA that address the security and scalability challenges for building a decentralized shared sequencer.

1.4.1 HotShot Consensus

The nodes in HotShot run a protocol to agree upon a *short vector commitment* to the raw transaction data, thus finalizing the sequential order of the submitted transactions. In the context of decentralized sequencers, we highlight the following aspects that facilitate a scalable consensus.

Dynamic participation. Different from HotStuff [125], HotStuff-2 [82] and other traditional BFT protocols that only work in a *permissioned* setting where the set of consensus participants is fixed apriori, we extend to the proof-of-stake (PoS) setting where the participants can dynamically join and leave the consensus protocol through the transfer of their staking power. To this end, we adapt HotStuff-2 to the PoS setting, and the adapted protocol satisfies safety and liveness so long as the accumulative adversarial weight behind corrupted staking keys is below 1/3 of the total staking weight. One particular challenge is for dealing with epoch transitions where the set of staking nodes changes. For instance, in the extreme case, over two consecutive epochs, the set of staked nodes can be completely different and the locking mechanism in HotStuff/HotStuff-2 no longer works. We address the issue by extending the HotStuff-2 protocol to add a constant number of rounds at the end of each epoch to help securely transfer the view information from one staking set to another. We refer to Section 2.4.2 for more details.

Bribery resistance. A common approach to adapt a permissioned protocol to the proof-of-stake setting is to elect a small random committee that represents the entire set of stakers and run consensus among the committee members. However, an adaptive adversary can corrupt the committee and break the consensus after knowing the election result. Algorand [59] addresses this issue by leveraging a secret election protocol periodically such that the committee members are unknown to adversaries until the elected members broadcast their proofs of being on the committee. Note that via a slashing mechanism, the elected members can speak only once for broadcasting their winning ticket and consensus messages, thus it's hard for an adversary to corrupt the committee and send equivocated messages after knowing the election result. [[Binyi: Do we want to remove the committee-based schemes/algorand part as it was explained previously in intro?][Kartik: no strong opinion] Unfortunately, as noted in [12], Algorand's approach is insufficient in the presence of a bribing adversary, who can advertise for guaranteeing the payout to consensus nodes for certain verifiable malicious behaviors. The adversary does not even need to know the internal state or the identity of the bribed nodes. [12, 22] also note that the existence of such bribery attacks can lead to safety and liveness violations for many longest-chain PoS and BFT-based PoS consensus protocols.

Therefore, in Section 2.2, we introduce a new threat model to capture bribery attacks. In particular, we consider a virtual adversary who can monitor not only the messages published in the network, but also the "buffered" messages that honest nodes prepare to send. The adversary can adaptively corrupt an honest node anytime after seeing new messages published in the network; moreover, after seeing the list of buffered messages, the adversary can adaptively corrupt honest nodes and modify/drop the buffered messages to be sent. The power of adaptively corrupting honest nodes from their *unsent buffered messages* is the key to simulating bribery attacks. Different from traditional adaptive corruption, bribery attacks usually incur a continuous unrecoverable cost from an adversary with a limited budget, thus we further assume a finite but unknown bound on the number of briberies.

Optimistic responsiveness. The new threat model poses significant challenges for obtaining a performant consensus protocol that scales to tens of thousands of nodes. In particular, with bribery attacks, we exclude the consensus schemes that utilize small committees. On the other hand, the system should minimize the negative effect on liveness and safety in the presence of a bribery attack: (i) the consensus should always satisfy safety regardless of bribery, and (ii) the protocol should be able to drive progress after the network condition goes back to normal and the bribing adversary runs out of budget and stops. This demotivates the intent of an adversary to launch any bribery attacks in the first place. Thus we can safely assume that most of the time, the system will be in a good scenario without any bribery attempts.

To construct a scalable consensus protocol with great performance in good scenarios, we adapt HotStuff-2 to the PoS setting and introduce the following optimizations:

- **Consensus on short commitments instead of full data.** We devise a modular data-availability and consensus separation to enable better throughput and latency. The consensus layer is only responsible for agreeing upon a short commitment to the data block, whereas the availability of the full data is delegated to a DA layer running in parallel. Since the Tiramisu DA (described in the next subsection) is highly efficient and can achieve availability without disseminating data to everyone, the throughput/latency of the consensus protocol can be significantly improved.
- **Instantiating QC with aggregate signatures and bit-vectors.** Compared to a fully permissioned setting, it is non-trivial to build succinct and efficiently verifiable quorum certificates in the proof-of-stake setting because the set of voters is dynamic over time. To address the issue, the dynamic stake table maps identities to signature verification keys and stake amounts, and we instantiate the table using a persistent Merkle tree for efficient lookup/update and public key sampling. The stake key registration also requires proof-of-possession to prevent rogue-key attacks (the proof-of-possessions step can be removed if the registered keys are account addresses on L1). Given the stake key table, in Appendix B, we develop an efficient instantiation of quorum certificates. At a high level, a quorum certificate consists of an aggregate signature plus a short bit-vector indicating the set of voters. The nodes verify QCs by checking the signature after obtaining the aggregated signature verification key from the stake key table and the bit vector. Asymptotically the QC size is still linear to the number of staking nodes, but the linear part is indeed one-bit per node. Thus the QC is concretely efficient (e.g., 1.25KB for 10000 nodes). Moreover, the QC construction satisfies accountability which is essential for slashing.
- **The use of a CDN at the network layer.** Obtaining low latency requires parties to communicate with each other faster than relying on a P2P network. However, if we have tens of thousands of consensus nodes, directly connecting to all of them and maintaining these connections is not efficient for all consensus nodes. We leverage a CDN at the network layer akin to what's used in Web2 architectures. A CDN is powerful hardware with high bandwidth connections that enable efficient information routing at low latency. Such an infrastructure allows us to scale much better even with a large number of nodes both from the standpoint of computation and communication. For communication, data can be disseminated to all consensus nodes faster. From a computation standpoint, computationally-heavy tasks such as signature aggregation can be offloaded to this hardware. Moreover, the optimistic responsiveness property of our protocol synergizes with this modification at the networking layer to achieve even better performance.
- **Efficient view synchronization.** In HotStuff-2, liveness is guaranteed only if the participants are in the same view for a sufficient amount of time. HotStuff achieves this using a pacemaker module that suffers from quadratic communication complexity and non-optimal latency. To address the issue, we use the view synchronizer from Naor and Keidar [90] to achieve linear communication complexity and constant latency in the optimistic case.
- **Efficient decentralized random beacon.** The Espresso Sequencer requires a reliable source of randomness to generate the seed periodically for electing the consensus/viewsync leader as well as for sampling the small DA committee. In Appendix B.1, we instantiate a highly efficient random beacon in a decentralized fashion using aggregate signatures and delay function [15], which might be of independent interest.

1.4.2 Tiramisu DA

The Tiramisu DA layer plays a crucial role in ensuring data availability before reaching consensus. A DA leader, after collecting the raw data of the block, disperses the data over the DA network and obtains a certificate for availability. To achieve desirable throughput and low latency, it is essential to minimize both total communication and the number of roundtrips. The DA layer consists of 3 main components:

- **Savoiardi: bribery-resilient DA.** A verifiable information dispersal (VID) protocol where the leader encodes the input data into erasure-coded chunks and distributes the chunks among all storage nodes via a *dispersal* protocol. We emphasize that each storage node stores only a single chunk of size $O(|B|/n)$ rather than the entire block data, and the consistency of the chunk to the full data can be efficiently verified by the chunk recipient. After the dispersal protocol, the leader obtains a certificate to ensure that valid VID shares are available from a threshold of storage nodes. Later, anyone with a copy of the certificate can retrieve the original full data block by running a VID *retrieve* protocol. Our VID protocol is a combination of AVID-1 from [10] and DispersedLedger [123], which might be of independent interest. The dispersal protocol is highly efficient: It requires only one round of communication between the leader and each storage node where the leader sends the VID share and the storage node returns a signature after checking the consistency. Moreover, the total communication over the network is only a constant factor larger than the original block raw data, plus a tiny amount of payload commitment data sent to each node. Note that, however, the VID retrieval protocol requires the retriever to fetch encoded payload shares from many nodes and then decode those shares, which is slower than, say, downloading the full raw payload from a single source. Fortunately, the VID retrieval protocol is only triggered as a last resort when the other two layers, CDNs and random DA committee described next, are unavailable. This incurs high unrecoverable bribing costs (e.g., money rewarded to the bribed users) without causing any catastrophic effect on the security and performance of the system. Thus we expect that there is no incentive to launch committee bribery attacks, and we can quickly retrieve the data most of the time.
- **Mascarpone: a small DA committee for fast consensus.** A small set of random DA committee nodes that is refreshed periodically. The DA leader sends the full data block to everyone in the committee and obtain a certificate once a threshold of committee nodes acknowledge and return signatures. With high probability, a sufficient number of committee nodes are honest (under a static adversary) and thus guard the data availability of the data. Unfortunately, a small committee is susceptible to bribery attacks. For example, the adversary can bribe the committee to sign but not store the data. In such situations, we would resort to the Savoiardi layer for availability.
- **Cocoa: Web2 performance from a CDN.** A CDN layer consisting of a set of interconnected powerful servers that help route information efficiently. In the optimistic condition where the CDN is fully operational, and the network is not under attack, the CDN provides performance comparable to Web2 centralized network. While robust, CDNs are inherently centralized and thus cannot be considered as the sole infrastructure for blockchain protocols. The protocol, therefore, works fine without the CDN layer, and it is an optional component that increases performance.

In summary, using the fact that sequencer nodes do not execute the state machine, we achieve improved throughput and latency by constructing a highly efficient DA layer. This is possible because not every node needs to store the complete transaction data. In particular, our DA protocol minimizes overall network communication to the original block data size multiplied by a small constant, and enables comparable performance to centralized networks in optimistic conditions. Additionally, our DA layer remains secure against bribery attacks.

The connection between HotShot Consensus and Tiramisu DA. For liveness, the nodes in HotShot consensus need to ensure data availability before voting for a vector commitment proposal. Otherwise, it is unclear how a new leader can build a new block of transactions from the existing blocks. In particular, a node would vote only if (i) the proposal attaches a certificate that a threshold of parties in the random DA committee receives the full data, and (ii) the node itself receives a VID share for the commitment. This guarantees that the transaction data is always available after finalizing its order. Note that a bribing adversary might corrupt the entire random DA committee to sign but not store the data, hence the protocol has to trigger the VID decoding algorithm to retrieve the data. However, the adversary might be better off just corrupting the consensus leader in this case for stalling liveness. Thus, we expect that the VID retrieving protocol will never be triggered and the data can be retrieved quickly from CDN nodes or the small DA committee in the optimistic condition where the leader is honest and the DA committee is not bribed.

1.5 Related Work

The problem of state machine replication [78, 79, 104] studies how multiple deterministic distributed machines can agree on a common shared state, even if some of the machines are adversarial or *Byzantine*. The last four decades have seen a lot of progress towards designing Byzantine Fault Tolerant (BFT) protocols for SMR [31, 80, 87, 86, 59, 40]. While reviewing all the work in this space is beyond the scope of this manuscript, we will describe aspects that are closely related to HotShot.

Achieving high throughput and low latency. Several works in the past two decades have focused solely on designing protocols that can process a large number of transactions with low latency. This includes Nakamoto style protocols [88, 27, 51, 99, 100, 38, 69, 122, 126, 72] as well as classical BFT protocols under different network conditions such as synchrony [34, 3, 4, 45, 59, 111, 101, 106, 2], partial synchrony [31, 24, 125, 33, 32, 57, 112], and asynchrony [5, 61, 81, 108, 57]. To improve throughput, a recent line of work attempts to separate data dissemination from consensus on transactions [37, 41, 40, 108, 107, 100, 122]. Similarly, in terms of latency, the notion of optimistic responsiveness was introduced and shown to hold for many protocols [99, 101, 106, 125, 33, 31]; in a nutshell, a responsive protocol finalizes transactions at the speed of the network (independent of any pessimistic upper bound on network delay Δ).

Similar to many of these protocols, HotShot aims to obtain a high throughput and low latency too. However, given HotShot's use as a sequencer brings in unique constraints. First, to improve throughput, we separate data dissemination from consensus. Moreover, since the sequencer nodes are not responsible for execution of the state machine, HotShot only needs to ensure *availability* of data but the nodes themselves do not need access to the *all* of the transactions. Due to this, data availability is less of a bottleneck compared to traditional SMR systems. Moreover, our goal is for HotShot to be scalable to a large number of nodes.[2] Consequently, the amount of data transferred during consensus is an important metric relating the performance of the system. Thus, we base HotShot on the HotStuff-2 protocol [125, 82] and thus, inherit favorable properties such as low communication (linear in the optimistic case, and quadratic in the worst case), low latency for finality and optimistic responsiveness. Moreover, we implement an accountable variant of HotStuff-2 to ensure the ability to detect and slash misbehaving parties in the case of a safety violation when the actual number of faults $f > n/3$.

Data availability. At a high level, data availability protocols must ensure that once the data is shared among the nodes of the network, it can be reliably recovered later. Erasure codes is a fundamental tool to achieve such a goal, and in particular Reed Solomon codes [121] are used as a building block to either (i) ensure the data has been correctly shared in a probabilistic way via *fraud proofs* [127, 6, 7], (ii) rely on a *Verifiable Information Dispersal (VID)* protocol that guarantees the data can be recovered if enough participants confirm their data share is valid[10, 93, 123, 9, 105, 92, 28, 124, 64]. Typically, these works proceed in two steps where in the first step a different erasure code share is shared with each of the parties (ensuring availability of data with communication complexity $O(|B|)$ bits) and in the second step, each of the parties share their data with all of the parties (making data available to all parties, with communication complexity $O(|B|\,n)$). Given our need to ensure only the availability of data, our protocol performs only the first step requiring $O(|B|)$ communication. Any party, e.g., a rollup, who needs access to the data can always reconstruct the data using shares collected from other nodes. This is also similar to Ethereum's Danksharding proposal [96].

The approach using erasure codes obtains low communication complexity; however, accessing the data still requires communicating with many nodes. Thus, our solution contains two additional layers of data availability: a small data availability committee and a content delivery network (CDN) that store the entire data. While we do not rely on these additional layers for security (since

[2] E.g executable on all Ethereum validators.

the small committee can be bribed and the CDN may malfunction), under optimistic conditions, they do provide easy and efficient data access responsively.

Bribery resistance. Prior works have classified adversaries as either static, adaptive, or mobile. While static corruption is a weak adversarial assumption for open blockchain settings, practical solutions are not known under strong mobile adversaries [98, 128, 20, 54]. Thus, many works aim to secure themselves under an adaptive adversary, although many of them still incur a high communication complexity [21, 30, 70, 129, 3, 45, 1, 68]. Algorand [59] improves upon this to simultaneously achieve low communication complexity using small committees, in addition to security against an adaptive adversary. However, as also mentioned in [12], we note that an adversary can utilize bribery attacks to blindly corrupt parties, making solutions based on small committees insecure. Our solution, on the other hand, obtains the desirable performance parameters while being secure under an adaptive and bribing adversary.

Use of a CDN. We leverage a hybrid network composed by a Content Delivery Network (CDN) coupled with a P2P network. In the optimistic scenario, all messages and data will be exchanged through the CDN, thus guaranteeing close to optimal throughput and latency. While the idea of combining CDN and P2P networks to improve content distribution has been explored in previous works [131, 65], their purpose is to boost the performance of centralized systems. In our case the goal is to maximize network performance most of the time while retaining liveness and censorship resistance even when the CDN is disrupted or compromised.

[Kartik: have heterogenous h/w (even other types) been used in other blockchains?]

Rollups. Since they started to be conceptualized [66, 120] a few years ago, L2 rollups have become a reality and are gaining a lot of traction in the Ethereum ecosystem. Users have now a wide range of alternatives [75, 94, 109, 114, 97] for making payments or interacting with their favorite dApps while cutting significantly on their gas expenses. Among the most popular rollups projects, the question of decentralization has drawn increasing attention, especially due to some practical issues [133]: Aztec is inviting the community to participate in the process of decentralizing their network [74], while other rollups project are including the topic of decentralization in their roadmap [76, 113]. In parallel, the community has been discussing and working on a more generalized approach, of which the present work is an example: instead of decentralizing each rollup individually, the idea is to rely on a *shared* and *decentralized* sequencer that reliably orders and stores the transactions for all rollups combined. The numerous benefits, as well as the delicate tradeoffs, of using a single sequencer for all rollups (e.g.: reuse of infrastructure, cross-chain MEV capture and redistribution, interoperabilitly) are being discussed actively [110, 35] and constitute a promising area of research. On the practical side, several concrete initiatives [110, 77] are aiming for bringing shared sequencers into the hands of rollups and end users in a near future. Regarding MEV [39] handling, which is a central concern when designing a shared sequencer, the team behind SUAVE [55] is building a network for decentralizing block building in order to make MEV capture and distribution more fair.

2 HotShot Consensus

In this section we specify the HotShot consensus layer. We first define the network and adversary model in Section 2.2, then proceed to explain how our consensus protocol is built on top of HotStuff in Sections 2.3 and 2.4. We specify the protocol executions in Section 2.4.5. Finally, we defer the concrete instantiations of the primitives described in this section to Appendix B.

2.1 Notation

The HotShot protocol proceeds as a succession of *epochs*.

- $e \in \mathbb{N}$: an epoch.
- Γ: number of blocks committed in each epoch.
- $\mathbf{N}_e$ (boldface): the total amount of stake in the system in some epoch $e \in \mathbb{N}$.
- $\mathbf{t} = \frac{1}{3}$ (boldface): the maximum fraction of the Byzantine stake.
- n: number of consensus nodes.

- t: number of Byzantine consensus nodes.

2.2 Threat Model

Types of adversaries. We distinguish between the following types of adversaries: *Static* adversaries have to choose the set of corrupt nodes at the beginning of the protocol, while *adaptive* adversaries are allowed to take control of new nodes during the execution of the protocol. We say an adversary is *passive* if it has access to the memory of a (corrupted) node but does not change the behaviour of such node. *Active* adversaries on the contrary are able to make corrupted nodes behave arbitrarily. Bribing adversaries [12] leverage economic incentives in order to be able to identify nodes with critical responsibility in the protocol (e.g. being the leader for the next round) in a passive manner. *Mobile* adversaries[128] have the ability to corrupt and uncorrupt nodes during the execution of the protocol. *Rational* adversaries[60, 52] aim only at maximizing their profit, while *non-rational* adversaries considered in this work are willing to invest economic resources in order to break some core property of the protocol such as safety or liveness.

Motivation: Bribery resistance. A common recipe to adapt a permissioned protocol to the permissionless setting is to use a small committee that runs the steps of the permissioned protocol. Each committee member has a probability of being selected proportional to its resources (e.g. stake in PoS blockchains). However, PoS protocols that follow this approach can only withstand a *mildly adaptive adversary* [99] who can launch targeted attacks but with a delay longer than the rotational period. A fully adaptive adversary with instant corruption can break these protocols by corrupting all committee nodes immediately after the election result becomes publicly known. To counter fully adaptive corruption, Algorand [59] introduced the notion of *player replaceability* by leveraging on secret elections in every sub-step of its Byzantine agreement where the lottery tickets are locally computed, therefore committee members are unpredictable and unknown to adversaries until winners reveal their publicly verifiable winning tickets for that step. More importantly, elected members only speak once by broadcasting their winning tickets and consensus-related messages. The next sub-step will be carried out by another randomly elected committee, leaving no window for targeted attacks.

Unfortunately, as observed by Bagaria et al. [12], player replaceability is insufficient in the presence of bribing adversaries who can launch a campaign that guarantees payouts for certain verifiable malicious behaviors. For example, attackers can create a smart contract that pays elected committee members to censor certain types of transactions. The hallmarks of bribing adversaries compared to adaptive adversaries are:

- Adaptive yet passive corruption: adversaries do not need to know the identities of the target to launch attacks. Furthermore, corruption is passive as the bribed nodes will voluntarily render controls or behave maliciously as instructed to collect bribes. An important characteristic of passive corruption is that it does not require any *predictability* of the protocol to succeed. As a result, player replaceability as a countermeasure for proactive adaptive attacks becomes less useful.
- Increasing cost for continuous bribery: the budget for bribing participants is finite and once it gets consumed, it becomes inaccessible in the future. [Dahlia: I am not sure this is clear enough. Perhaps say the opposite? Namely: Bribery budget does NOT run out unless utilized.] [Philippe: I think it is still better to retain this idea of continuous expenditure, as it intuitively explains why we are able to deal with bribery attacks. However I removed the mention to MEV as indeed it is confusing because it feels like the adversary can renew his bribery budget using MEV gains.]

It has been noted that the existence of such bribery attacks can lead to safety and liveness violations for many PoS consensus protocols [12, 22]. Meanwhile, vote buying and bribery platforms exist in decentralized governance systems.[3] In general, we believe that consensus nodes are rational in practice, and bribery attacks might be possible so long as it poses little risk to the users being bribed. Unfortunately, formal definitions and analyses of real-world bribery attacks are lacking. Therefore, we consider a new threat model that captures not only the adaptive adversaries but also the possibility of bribery.

[3]Example bribery platforms (a.k.a. incentive marketplaces) in DeFi governance: `https://hiddenhand.finance/`, `https://docs.votium.app/`, and `https://stakedao.org/`

Network model. Our network condition follows the standard GST-flavored *partial synchrony model* [47] where there is an unknown Global Stabilization Time (GST) before which the network is asynchronous and after which the network is synchronous with a known delivery bound Δ. We assume three types of network communications: (i) reliable and authenticated point-to-point connections between peers; (ii) gossip communication over the network; and (iii) sending a message from a node to a list of receivers through a shared communication channel. Note that sending a message to n nodes via a dedicated communication channel is practically much cheaper than naïvely sending the message to n nodes via point-to-point channels or gossip networks. E.g., we can leverage powerful CDN nodes to cache and deliver the message. We defer the details of the network layer to Section 4.

Protocol execution. We consider a system with dynamic overall participation from nodes represented using a publicly-accessible *stake table* that maps nodes' public keys to their stakes in the network. Each node generates a secret/public key pair, where the public key is published on the stake table and uniquely identifies the node. Moreover, the public key carries a non-negative *weight* proportional to and deterministically computable from the node's stake.

At the beginning of each epoch e, the adversary can reconfigure the set as well as the stake weight of participating nodes. Each epoch consists of multiple *views* and each view associates a unique dedicated leader to drive progress. [Gus: Please clarify: each view has 0 or 1 blocks?? Can we phrase it as a hierarchy: epochs, views, blocks?] Epochs, blocks and views are denoted by monotonically increasing epoch number e, block height h and view number v respectively. Execution of a consensus protocol can be modeled as a general state machine, whose overall states at any moment consist of the summation of internal states of each process/node and a *transcript* that records all network messages delivered so far.[4] We refer to the internal states of all faulty nodes plus the transcript as the adversarial view of the execution. Each honest node's behavior is well-specified by its internal state and the messages it received, whereas each Byzantine node can behave arbitrarily. Observe that every action taken by an honest node (e.g., sending out messages or committing to an agreed value) will only be triggered by the arrival of a new message. Next, we specify how the adversary can corrupt nodes adaptively.

The message-passing and adversary models. To enable a unified model that can capture both proactive corruption (e.g., targeted hacking) and passive corruption (e.g., bribery attacks), we consider the following message-passing model: We devise a natural notion of *outbox buffer* to denote a global message queue. An honest node, after being triggered by the arrival of a new message, can post messages to the buffer with the intended receiver(s), the sender and a timestamp specified. Meanwhile, the adversary can inject arbitrary messages into the outbox buffer. There are three stages before delivering the messages from the outbox buffer:

Corruption stage: After viewing the outbox buffer, the adversary can adaptively corrupt a new set of Byzantine nodes so long as the cumulative adversarial weight behind corrupted keys (in the active staking set) is below the threshold $\mathbf{tN}_e$, where $\mathbf{t} = 1/3$ and $\mathbf{N}_e$ denotes the total stake weight in the current epoch e. Note, however, that the adversary *cannot* drop/modify the messages sent by the nodes that were honest previously but become Byzantine after the corruption stage. That is, the new Byzantine set will take effect only in the next round. Additionally, note that the adversary *cannot uncorrupt* a Byzantine node that it already corrupted previously. This constraint is necessary to exclude the attack described in Remark 2.2. [[Binyi: Relate the proactive adversary in https://eprint.iacr.org/2022/698.pdf to our security model.]

Bribery stage: After the corruption stage, the adversary can pick a subset S of *newly corrupted nodes* and drop/modify the buffer messages sent by S. Let $|S|$ denote the number of nodes in S, we say that the adversary conducts $|S|$ briberies in the bribery stage. Note that the difference between bribery and corruption is that the bribed nodes can immediately behave maliciously, while the non-bribed corrupted nodes can only behave maliciously in the next round. Also note that the number of briberies per epoch is bounded by $\mathbf{tN}_e$ as the adversary cannot uncorrupt parties during an epoch.

Network delay stage: The (network) adversary in control of message delivery can finally shuffle/re-order (but never drop without delivering) messages in the outbox buffer while respect-

[4]Multi-casting to m nodes will generate m entries on the transcript upon their delivery. The transcript also includes the initial configuration and public parameters of the protocol.

ing the network condition.[5] The messages will be removed from the buffer upon successful delivery.

[[Binyi: Remove Figure 3 for now, as the flow in Figure 3 should be changed, the static adversary box should go first.]

Remark 2.1. *Our model captures the attacking power that an adversary can bribe without knowing a party's identity or its published messages. This is because the adversary is determining the new corruption set upon seeing the buffered messages that have not been sent yet. A subset of this newly corrupted set would be the bribed set that can behave arbitrarily in this view, while the rest can start behaving badly only from the next view.*

Remark 2.2. *As noted by [8], it is highly non-trivial to model parties that are temporarily corrupted but can be uncorrupted later. E.g., after uncorruption, the adversary can still remember the internal state of the previously corrupted parties (e.g., the long-term secret keys). On the other hand, the uncorrupted party might forget the meta-data stored previously by the adversary. Both issues add complexity to the security analysis. Note that compared to our threat model, Algorand [59] enforces a strictly stronger assumption where the adaptive adversary can never uncorrupt either, and there is no possibility for bribing.*

Remark 2.3. *There exist stages where the participants of two consecutive epochs* e *and* $e+1$ *are actively staking simultaneously, e.g., during the period where the epoch* e *finishes but epoch* $e+1$ *has not committed any blocks yet. In this period, we assume that the accumulative adversarial weight behind corrupted keys in epoch* e *is below the threshold* $\boldsymbol{tN}_e$*, and meanwhile, the weight behind corrupted keys in epoch* $e+1$ *is below the threshold* $\boldsymbol{tN}_{e+1}$*.*

Safety and liveness. A consensus protocol in our model is used to enable a set of staking nodes to agree upon the sequential order of a list of submitted transactions. We say that the protocol provides *Safety* if different honest staking nodes never commit to different values at the same position; we say that the protocol provides *Liveness* if each submitted transaction will eventually be committed by all honest staking nodes in some epoch $e \in \mathbb{N}$.

2.3 Overview of HotStuff

In this section, we present a high-level overview of the core of the HotStuff-2 protocol in the permissioned setting; we refer readers to HotStuff [125] and HotStuff-2 [82] for a detailed exposition. While both of these works are compatible with HotShot, we will assume HotStuff-2 as the underlying protocol. The protocol operates in a view-by-view manner. It consists of two parts, a steady-state protocol and a view synchronization (or pacemaker) protocol for advancing views. In this section, we will only focus on the steady-state protocol that drives a commit decision in a view when all correct nodes overlap in the view for sufficiently long, and a designated leader for a view known to all nodes is correct; we will describe details related to view synchronization in Section 2.4.3.

We will first describe some data structures and terminology. The permissioned setting consists of n nodes out of which up to t are Byzantine.

Block format. The protocol forms a chain of values. We use the term *block* to refer to each value in the chain. We refer to a block's position in the chain as its *height*. A block B_k at height k has the following format

$$B_k := (b_k, h_{k-1})$$

where b_k denotes a proposed value at height k and $h_{k-1} := H(B_{k-1})$ is a hash digest of the predecessor block. The first block $B_0 = (b_0, \perp)$ has no predecessor. Every subsequent block B_k must specify a predecessor block B_{k-1} by including a hash of it. We say a block is *valid* if (i) its predecessor is valid or $\perp$, and (ii) its proposed value meets application-level validity conditions and is consistent with its chain of ancestors (e.g., does not double-spend a transaction in one of its ancestor blocks).

Block extension and equivocation. We say B_l *extends* B_k, if B_k is an ancestor of B_l ($l > k$). We say two blocks B_l and $B'_{l'}$ *equivocate* one another if they are not equal and do not extend one another.

[5]"Respecting the network condition" means 1) timely delivery during synchrony based on the sent timestamp and the known delay bound; 2) eventual delivery during asynchrony.

Let v be the current view number and node L_v be the leader in this view. Perform the following steps.

1. **Propose (leader only).** The leader L_v broadcasts $\langle \mathsf{propose}, B_k, v, C_{v'}(B_{k-1})\rangle_{L_v}$.
   Here, $B_k := (b_k, h_{k-1})$ is the block that should extend the highest certified block B_{k-1} with certificate $C_{v'}(B_{k-1})$ known to leader.
2. **Vote (all nodes).** Upon receiving the first valid proposal $\langle \mathsf{propose}, B_k, v, C_{v'}(B_{k-1})\rangle_{L_v}$ in view v: If $C_{v'}(B_{k-1})$ is ranked no lower than the locked block, then send $\langle \mathsf{vote}, B_k, v\rangle$ as a threshold signature share to L_v. Update lock to B_{k-1} and the certificate to $C_{v'}(B_{k-1})$ to leader L_{v+1}. [Gus: What does it mean to "update" a certificate to a leader? How do I deduce L_{v+1}?]

   Commit rule (all nodes). Commit block B_k iff there exists an $l \geq k$ such that $C_{v'}(B_l)$ and $C_{v'+1}(B_{l+1})$ are formed. [Gus: Why is this item not enumerated as step 3?]

Figure 2: View protocol for nodes in view v. [Gus: Suggestion: delete this figure and integrate it into the main body. This figure duplicates content from the main body and it is difficult to understand without context from the main body.]

Certificates and certified blocks. In the protocol, nodes vote for blocks by signing them using an aggregate signature. We use $C_v(B_k)$ to denote a set of signatures on $h_k = H(B_k)$ by $2t + 1$ nodes in view v. We call $C_v(B_k)$ a certificate or a quorum certificate (QC) for B_k from view v. Certified blocks are ranked by the views in which they are certified, i.e., a certificate $C_v(B_k)$ is ranked higher than $C_{v'}(B'_{k'})$ if $v > v'$.

Locked blocks. At any time, a party locks the highest certified block to its knowledge. During the protocol execution, each party keeps track of all signatures for all blocks and keeps updating its locked block. Looking ahead, the notion of a locked block will be used to guard the safety of a commit.

Steady-state protocol. Figure 2 presents the core steps executed in the steady state protocol. Each view has a designated leader who is responsible for driving consensus on a sequence of blocks. For now, we will assume that we know a designated set of leaders. The goal of a view leader is to extend the highest certified block in the current sequence with a new block and get it certified.

- A leader in view v forms a block B_k at height k that contains its proposed value, the view number, and the highest-ranked certificate the leader knows. The leader sends a proposed block to all the nodes.
- Each party keeps the highest certificate it ever received. A party votes for block B_k if it extends the highest certificate, by sending a signed vote to the next leader.
- A block becomes certified if $2t + 1$ nodes vote for it. The certificate $C_v(B_k)$ formed from $2t + 1$ signed votes is aggregated by the leader of the next view.

Committing a block. A block B_k is said to be committed if there exists an $l \geq k$ such that $C_{v'}(B_l)$ and $C_{v'+1}(B_{l+1})$ are formed, and B_l extends B_k. In other words, either for B_k or for one of its successors, two blocks at consecutive heights are certified in consecutive views.

Intuition for safety and liveness. Safety within a view is ensured by a simple quorum intersection argument – no two distinct blocks can be certified when the number of Byzantine nodes is bounded by t. Safety across views is guaranteed by the locking mechanism – if a node commits a block in a view v, then a quorum of $\geq t + 1$ honest nodes are locked on this block. This condition ensures that a lock and consequently a commit on an equivocating block is not possible in higher views. The liveness argument is more subtle. At a high level, it relies on the ability of an honest leader to eventually propose a block that extends the highest certified block, which would be voted for by all honest nodes.

2.4 Towards HotShot Consensus

HotShot extends the HotStuff protocol to a proof-of-stake (PoS) setting. In this section, we describe the management of keys in HotShot (Section 2.4.1), the reconfiguration process as the set of stakers change (Section 2.4.2), the view synchronization protocol used by HotShot (Section 2.4.3), the source of randomness used for electing nodes (Section 2.4.4), and we put all of these pieces together to describe the protocol (Section 2.4.5).

2.4.1 Handling Staking Keys

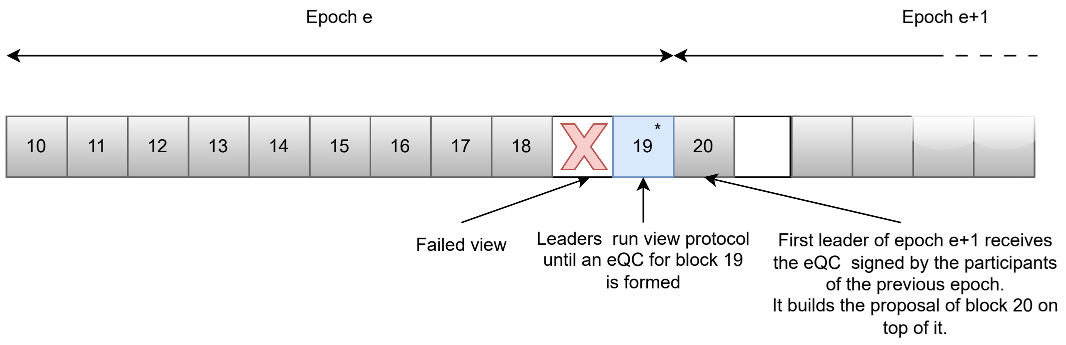
A central component of a PoS protocol is the stake table that maps public keys or participant identities to stake in the system. In HotShot, nodes rely on the stake table to know who is allowed to vote on blocks. Instead of letting nodes manage the stake table as some shared state, we instantiate this stake table as part of the *Sequencer* contract on the L1 introduced in Section 1.3.

There are several advantages to this approach. First stake transfers are handled outside HotShot which ensures our solution remains a pure Byzantine atomic broadcast without an execution layer. Beyond performance, this design decision allows to leverage restaking protocols such as EigenLayer [49] in order allow any owner of L1 staking tokens (ETH tokens on Ethereum, for example) to participate in HotShot. In practice nodes monitor the L1 blockchain in order to know the state of the stake stable. By doing so, at the beginning of each epoch, all honest nodes can agree on the version of the stake table that will be used for the next epoch. Note that for efficiency purposes nodes will maintain a local copy of the stake table optimized for fast access during QC validation. Finally note that the presence of the stake table on Ethereum prevents long range attacks [44, 56]. The reason is that when a new node joins and wants to catchup with the sequencer chain, it can ask the participants of the last epoch, derived from Ethereum, for the most recent state. After collecting identical replies from nodes with greater than $\mathbf{tN}_e$ fraction of stake, the node can safely proceed to download the sequence of blocks until being able to recompute locally the state.

2.4.2 Reconfiguration Across Epochs

Extended quorum certificate. We extend the notion of *quorum certificate* (QC) for supporting stake reconfiguration in HotShot. For a block B_k, an *extended Quorum Certificate (eQC)* of B_k consists of 3 consecutive QCs for B_k in 3 consecutive views. The first two consecutive QCs ensures that the block B_k is committed in the corresponding epoch (according to the committing rule in Section 2.3); and the last QC guarantees that a quorum of honest nodes in the epoch has committed the block B_k. [Gus: How are the first two QCs different from the final QC? Why not just say "all 3 QCs ensure that the block is committed in the corresponding epoch"?]

Reconfiguration mechanism. While in HotStuff-2 the set of participants is fixed, in our setting we want to allow participants to dynamically join and leave the protocol. In order to achieve such a goal, we rely on the stake table that tracks the voting power of each member. While the stake table is updated every time a block is appended to the chain, we cannot refresh the set of participants after every block. Indeed, before any decision is made, consensus must first be reached regarding who is allowed to vote. That is why a snapshot of the stake table is taken periodically at the beginning of every epoch e. This snapshot will define the set of participants for the next epoch $e+1$ and corresponds to the state of the table w.r.t. the first block of epoch e. For the first epoch of the protocol, the set of participants is hardcoded in the genesis block.



Epoch e
Epoch e+1
10
11
12
13
14
15
16
17
18
*
19
20
Failed view
Leaders run view protocol until an eQC for block 19 is formed
First leader of epoch e+1 receives the eQC signed by the participants of the previous epoch. It builds the proposal of block 20 on top of it.

Figure 3: **Epochs transitioning.** In the example, each epoch contains $\Gamma = 10$ blocks. After block 18 is committed, a view fails. At this point the leaders keep running the view protocol in order to build the eQC for block 19. This eQC is sent to the first leader of epoch $e + 1$. Assuming this leader is honest, it will check the eQC in order to ensure block 19 has been committed and will add its own block numbered 20.

For the set of staking nodes in epoch $e + 1$, to agree upon the first block B to be committed in epoch $e + 1$, they have to first agree upon the last committed block in epoch e that B can extend to. To achieve this (see example in Figure 3), at the stage where the staking set for epoch e tries to commit to a last block B' in epoch e, the nodes will keep running the view protocol of HotStuff-2 until an extended QC of B' is formed. And the nodes in epoch e will send the eQC to the nodes in epoch $e + 1$ and ensure that a quorum of honest nodes in epoch $e + 1$ receive the eQC. We specify the mechanism for achieving this in Sect. 2.4.3. After obtaining the eQC, an honest leader in epoch $e + 1$ would be able to generate a valid proposal accepted by the voters as they are convinced that B' is the unique block being committed in the last epoch.

2.4.3 View Synchronization

Liveness can only be guaranteed after GST so long as honest nodes remain in the same view for a sufficient length of time [89]. HotShot's view synchronization module augments HotStuff's [125] Pacemaker module to ensure this is the case. HotShot employs Naor-Keidar [90] to achieve view synchronization. The Naor-Keidar protocol has expectedly and optimistically linear communication complexity and constant latency even under Byzantine faults, allowing HotShot to scale to large, heterogeneous networks. It is a partially synchronous protocol that proceeds through three rounds of voting during which nodes send their view synchronization votes to relay nodes, which are consensus nodes chosen independently of consensus leader nodes that collect view synchronization votes and form certificates.

The view synchronization module runs asynchronously alongside the consensus module. In HotShot, a node will not locally enter a new view unless they have sufficient evidence that enough honest nodes in the network agree on the same view number. This evidence can take one of three forms:

- A quorum certificate for the previous view. (Or an extended QC from the previous epoch if the new view is the first view in a new epoch.)
- A timeout certificate for the previous view.
- A view synchronization certificate for the current view.

The consensus module handles forming quorum certificates and timeout certificates. Timeout certificates (not described in the simplified description earlier), also used in HotStuff, are similar to QCs but state a node's timeout was triggered before receiving any valid message from the leader. The view synchronization module handles forming view synchronization certificates, which are certificates formed during the Naor-Keidar protocol. View synchronization certificates are only needed if neither of the other certificates exist.

A node's local view synchronization process can be triggered in two ways. First, it can be triggered directly from the consensus module when the node reaches the end of a view but does not have sufficient evidence to enter the next view. Second, a node can also join a view synchronization protocol already in progress in the network by receiving valid view synchronization messages. This second trigger allows a node to participate in view synchronization even if they did not locally observe a view timeout. Once the view synchronization module has seen a valid view synchronization certificate it sends a ViewChange event to the consensus module. The consensus module updates its state accordingly.

ViewSync across epochs. As described in Sect. 2.4.2, after the nodes in epoch e have committed the last block and obtained the eQC, they need to send the eQC to sufficiently many honest nodes in epoch $e+1$ and drive view synchronization in epoch $e+1$. We achieve this by running a variant of the Naor-Keidar protocol: Suppose the nodes in epoch e obtain eQC at view v, and denote $R_1, R_2, \ldots, R_n$ as n pseudorandom relay nodes in epoch $e+1$ for view $v+1$. The nodes in epoch e first send the eQC to relay R_1, and R_1 runs as a random relay in the Naor-Keidar protocol over the set of nodes in epoch $e+1$. After obtaining the viewsync certificate, R_1 sends the viewsync certificate back to the nodes in epoch e to convince them that sufficiently many honest nodes have received the eQC and advanced to view $v+1$. For a node in epoch e, if it doesn't receive a viewsync certificate after timeout, it sends eQC to relay R_2, and R_2 will run the same algorithm as R_1 did, and so on. The scheme has linear expected communication complexity, but has quadratic complexity in the worst case. The main difference of this scheme from the Naor-Keidar protocol is that the set of nodes that triggers the viewsync (i.e. the nodes in epoch e) can be different from the set of nodes that runs the viewsync protocol (i.e. the nodes in epoch $e+1$).

2.4.4 Decentralized Random Beacon

We leverage a decentralized random beacon (DRB) as a dependable source of randomness for selecting our consensus leader, view synchronization relay nodes, and data availability committees. Note that selecting the consensus leader or view synchronization relay in a deterministic way (e.g. via a round-robin approach) may cause linear latency given a static adversary. This is why we use a secure source of randomness to ensure that moving to the next view as well as driving consensus progress takes constant expected time in an optimistic scenario that has no adaptive corruptions.

In our threat model, the conventional security requirements of unpredictability and unbiasability are not sufficient. Indeed, allowing bribery attacks on the (DRB) may allow an adversary to bias the seed in order to get elected most of the time over many consecutive views. At this point even a static adversary might be in condition to keep controlling the seed without spending additional resources and thus stall liveness indefinitely. Additionally, we need to take into consideration rewards collected by the leader whether explicitly or under the form of MEV extraction: If an adversary can manipulate the beacon via bribery attacks and get elected more often, then it will also increase its income for producing blocks. In some scenario the adversary might even be able to reuse part of this unfair profit to fund more bribery attacks. Without a bribery resistant DRB to handle leader election, we are left with some deterministic (e.g. round robin) solution. In practice this can be problematic as it will be required to go over every unit of stake during an epoch. [Gus: Why? Example: why can't an unfinished round robin simply resume in the next epoch?] This means that the length of epoch cannot be bounded a priori which may harm user experience as registering a new staking key may take a long and/or variable time.

On the other hand, maintaining low communication complexity becomes quite a challenge when dealing with numerous participants. A naive solution where random shares are broadcast to the network would lead to excessive communication, thus we require a leader to collect all random shares for linear communication. However, this opens the space for a malicious leader to bias the output by arbitrarily choosing which shares to include. To address these challenges, we adopt a "commit-reveal" approach, utilizing a *Delay Function* that requires a certain amount of time to compute, even in parallel. By committing the input to the delay function before the leader obtains the output, which serves as the beacon value, we achieve both desired outcomes. Details can be found in Appendix B.1.

Finally, we observe that in theory we could also leverage the Ethereum random beacon [48]: While its current implementation based on RANDAO [103] is not bribery resistant, possible improvements based on VDF like in our solution may make it suitable for our needs.

2.4.5 Putting Pieces Together

In this section, we combine the designs presented in Section 2.4.1, Section 2.4.2, Section 2.4.3 and Section 2.4.4, and describe the HotShot protocol in Figure 4.

Stake registration and update. Staking keys (and amounts) can be registered (and updated) into the stake table $\mathcal{T}$. At the time of key registration/stake transfer, the stake amount of the key is unchanged temporarily. The stake table updates are only activated after every Γ block periods. More precisely, Γ is the number of blocks per epoch during which the stake table configuration is unchanged. For an epoch $e \in \mathbb{N}$, we use $\widetilde{\mathcal{T}}[e]$ to denote the set of staking keys that participate in voting for epoch e, which is the staking set after executing all of the staking key registration/updates before epoch $e-1$.

Random beacon seeds. Looking ahead, the consensus protocol needs to continuously sample random leaders for driving progress. Hence we need a mechanism to generate random beacon seeds for each epoch. More specifically, we define the random beacon seed for each epoch $e+1 \in \mathbb{N}$ as the delay function output on input the QC of the first committed block in epoch e. As explained in Section 2.4.4, the DRB seed is unpredictable and unbiasable even against bribery; moreover, since the staking table configuration for epoch $e+1$ is finalized at the end of epoch $e-1$, the leaders in epoch $e+1$ are randomly distributed and it takes constant expected time to choose an honest leader that drives the consensus progress under the optimistic scenario where the adversary does not adaptively corrupt nodes during an epoch.

Workflow. At the beginning of the protocol, each node in the genesis staking set initializes its local data structures and variables, which include: (i) the list of stake table snapshots for the recent and the current epochs; (ii) the node's QC signing key; and (iii) the list of committed blocks. Besides, the node also stores auxiliary information for view synchronization and random leader selection.

The consensus protocol is run by the set of active staking nodes and proceeds by *views*. The nodes enter new views according to mechanism specified in Section 2.4.3. In each view v, the nodes agree with a *view leader* sampled randomly from the active stake key table. The sampling randomness is derived from the view number v and the random beacon seed for the current epoch. The view leader will pick a *short commitment* to the transaction data available in the DA layer, and broadcast a proposal for the commitment. The proposal also attaches the DA committee certificate for the commitment as well as a certified block that the commitment will extend to. Note that there are three cases:

- **Case 1:** The leader's proposed block is the first block in a new epoch e. Then we require the certified block to be the last *committed* block in the previous epoch $e-1$, and the certificate should be the extended QC (Section 2.4.2) of the committed block.
- **Case 2:** The leader's proposed block is not the first block in an epoch and the certified block is not equal to the proposed block. Then the certified block should be in the same epoch and with a quorum certificate that has the highest view known to the leader.
- **Case 3:** The leader's proposed block is the last block in an epoch and the certified block is equal to the proposed block, then the certified block should be from the previous view $v-1$.

For each of the other staking nodes, upon receiving a proposal, it will check the validity of the proposal and send votes to the leader of the next view. The proposal is valid only if (i) the DA committee certificate is valid and the node itself receives a VID share for the commitment, and (ii) the QC of the certified block follows the requirement, that is, it is a valid extended QC if the certified block is the last committed block in the last epoch, otherwise, it is a valid QC that has no lower rank than the node's locked block or it is a valid QC for the proposed block in the previous view. To validate a QC in epoch e, the validation algorithm builds the aggregated verification key given the staking keys of voters for the QC, it then checks that (1) the aggregate signature inside the QC is correctly verified, and (2) the total amount of stake corresponding to the QC is greater than $2\mathbf{N}_e/3$ where $\mathbf{N}_e$ is the total amount of stake for epoch e. We refer to Appendix B.3 for more details.

If the proposal is valid, the node will update its locked block. For the leader of the next view $v+1$, it will collect the votes sent by nodes. If it receives enough votes, it will update its local information and proceeds to the $v+1$ view.

Parameters: Let Γ denote the number of blocks after which the list of stake table snapshots $\tilde{\mathcal{T}}$ is updated, i.e., every Γ blocks form an *epoch*. Denote $\tilde{\mathcal{T}}[e]$ as the snapshot used for epoch e. Denote $\texttt{seeds}[e]$ as the decentralized random seed used in epoch e.

Initialization: For every node in the initial staking set, initialize the stake table snapshots $\tilde{\mathcal{T}}$, random beacon seeds $\texttt{seeds}$, and the list of committed blocks to default values.

View protocol: Let $v \in \mathbb{N}$ be the current view number. Perform the following steps.

1. **Stake table update.** For every node, let h be the block height of its last committed block before view v. If $h \bmod \Gamma = 0$, i.e., it comes to an end of an epoch, the node will update the stake table given the stake updates from L1, and append the stake table for the new epoch to the list of stake table snapshots.
2. **Seed update.** If a node commits a first block in epoch e, it computes the beacon seed $\texttt{seeds}[e+1]$ for epoch $e+1$ as $\texttt{DF.Eval}(C_{v'}(B_{(e-1)\Gamma+1}))$, i.e., the delay function on input the QC of the first committed block in epoch e. Note that the delay function hardness parameter is set so that the node is guaranteed to know $\texttt{seeds}[e+1]$ before the end of epoch e. For newcomers in epoch $e+1$, they can extract the seed from a committed block in epoch e.
3. **ViewSync.** Let $e = \lceil (h+1)/\Gamma \rceil$ be the current epoch. Let L_v denote the leader of view v sampled from the stake key table $\tilde{\mathcal{T}}[e]$ using randomness $H_1(\texttt{seeds}[e], v)$ (where H_1 is a hash function modeled as a random oracle). The nodes enter view v if one of the following happens:
   - L_v receives a valid QC (or extended QC from the last epoch). The leader will send the (e)QC to other nodes, who would enter view v and vote after receiving the certificates.
   - If L_v does not receive a valid QC but receives a timeout certificate (TC) from the last view v, similarly, the leader sends the TC to other nodes who then enter view v afterward.
   - If any honest node reaches the end of view $v-1$ but doesn't receive any evidence to enter view v, it triggers the Naor-Keidar protocol [90] for view synchronization.[a]
   - For the special case where view $v-1$ is the last view in epoch $e-1$, the honest nodes in epoch $e-1$ will trigger the variant of the Naor-Keidar protocol and send the eQC to the nodes in epoch e (Sect. 2.4.3).
4. **Propose.** The leader L_v broadcasts $\langle \mathsf{propose}, B_k, v, C^*_{v'}(B_{k'}) \rangle_{L_v}$. Here,
   - B_k is the proposed block. As its data, it contains a short commitment to a block of raw transaction data, and an optimistic DA certificate dac from the small DA committee.
   - $C^*_{v'}(B_{k'})$ is the certificate for the block $B_{k'}$ to be extended. There are three cases:
     - **Case 1:** B_k is the first block in an epoch (i.e., $k \bmod \Gamma = 1$). Then $k' = k-1$ and B_{k-1} is the last block committed in the previous epoch and $C^*_{v'}(B_{k-1})$ is the extended QC for block B_{k-1} (See Section 2.4.2).
     - **Case 2:** B_k is not the first block in any epoch and $k' = k-1$. Then $C^*_{v'}(B_{k-1}) = C_{v'}(B_{k-1})$ is the QC for the highest certified block B_{k-1} known to leader.
     - **Case 3:** $k' = k$ and B_k is the last block in an epoch (i.e., $k \bmod \Gamma = 0$). Then $v' = v-1$ and $C^*_{v'}(B_k) = C_{v'}(B_k)$ is the previous QC for block B_k.
   - If L_v obtains $\texttt{seeds}[e+1]$ (Section 2.4.4), it also includes it in the proposal.
5. **Vote.** Upon receiving $\langle \mathsf{propose}, B_k, v, C^*_{v'}(B_{k'}) \rangle_{L_v}$, a node sends vote $\langle \mathsf{vote}, B_k, v) \rangle$ to L_v if:
   - the node received the VID share for the block commitment inside B_k, and the DA committee certificate dac inside B_k is valid;
   - the QCs in $C^*_{v'}(B_{k'})$ are valid given the stake table $\tilde{\mathcal{T}}[\lceil (k')/\Gamma \rceil]$ for height k';
   - if $k \bmod \Gamma = 1$, then $k' = k-1$ and $C^*_{v'}(B_{k-1})$ is the extended QC of B_{k-1}; if $k \bmod \Gamma = 0$ and $k' = k$, then $v' = v-1$ and $C^*_{v'}(B_{k'}) = C_{v-1}(B_k)$ is the previous QC for B_k; otherwise $C^*_{v'}(B_{k'})$ is the QC for B_{k-1}, which is ranked no lower than the node's locked block;
   - the random beacon seed field in the proposal is either empty or $\texttt{DF.Eval}(C_{v'}(B_{(e-1)\Gamma+1}))$ (i.e., the valid seed for epoch $e+1$).

   If the conditions hold, the node updates lock to B_{k-1} if $k \bmod \Gamma \neq 1$ and $k' = k-1$. The node also sends a vote $\langle \mathsf{vote}, C^*_{v'}(B_{k'}), v) \rangle$ to leader L_{v+1} where $C^*_{v'}(B_{k'})$ is included as part of the vote. If the node does not receive any valid message from the leader L_v after timeout, the node sends a timeout vote $\langle \mathsf{vote}, \perp, v) \rangle$ to L_{v+1}.

   Commit rule. Commit block B_k iff $\exists l \geq k$ such that $C_{v'}(B_l)$, $C_{v'+1}(B_{l+1})$ are formed.

[a]The NK protocol setups multiple random relays so that at least one of them is honest. The nodes first send votes to the first random relay, who aggregates votes and sends back the threshold signature as a viewsync certificate. If a node does not receive a response after a timeout, it sends vote to the next random relay and runs similar procedures, etc. We refer to Section 5.2 of [90] for more details of the algorithm specification.

Figure 4: The HotShot Consensus Protocol. [[Binyi: For simplicity, no need to include view sync logic.]

2.5 Security Sketch

[Kartik: After EthCC, (i) we should discuss the responsiveness and also have lemmas for communication complexity etc after taking into account other protocols like random beacons into consideration, (ii) update proofs to be for the pipelined version, (iii) how does view synchronization play across epochs when the set of nodes change?, (iv) discuss permission-gated download requests in the DA section, (v) cite bloxroute and others, (vi) ensure change of Ne is fine, (vii) our diagrams use different tools – needs to be consistent, (viiI) make the overall content coherent, (ix) discuss unknown budget vs ast.] We analyze the safety and liveness of HotShot. The proof idea is similar to that of HotStuff [125] and HotStuff-2 [82], except that we need to additionally address the challenges of (i) dynamic stake changes, (ii) bribing adversaries, and (iii) that the adaptive adversary can corrupt nodes gradually. The solution to challenge (i) involves modifications to the original HotStuff protocol, while challenges (ii) and (iii) are easier to handle and only involve minor changes to the security proof.

Lemma 2.1. *If a staked node commits a block B_k in view v, then any block that obtains a QC in a view number no less than v extends or equals B_k.*

Proof. Consider any block $B_{k'}$ that obtains a QC in view $v' \geq v$, where the QC is w.r.t. the staked nodes set in the epoch $e' = \lceil k'/\Gamma \rceil$. First, by definition $B_{k'}$ obtains a QC in view $v' \geq v$ thus $B_{k'}$ cannot be in an epoch $e' < e$.

Next, we consider the case where $B_{k'}$ is in an epoch $e' > e$. We will argue that in view v', all blocks in epoch e should have already been committed and no fork is possible: Note that over two consecutive epochs a and $a+1$, the set of staked nodes can be completely different, so the locking mechanism in HotStuff no longer works. Fortunately, there is a unanimous global view of the staking set for the new epoch through a source of truth (e.g., Ethereum), so that everyone can agree upon the set of staking nodes in epoch $a+1$. Moreover, our construction would collect all the necessary QCs at the end of each epoch for committing the last block of the epoch. This enables the participants in epoch a to transfer the safety property to the stake set in epoch $a+1$. More precisely, the honest nodes in epoch $a+1$ would only vote for a block that extends the last committed block in epoch a, and safety holds as the last committed block in epoch a is unique.[6]

Thus, $B_{k'}$ must extend the last committed block of epoch e, and the last committed block would extend or equal B_k if the lemma holds for any blocks that obtain QC in epoch e. In sum, it's sufficient to consider the case where $B_{k'}$ and B_k are in the same epoch.

Finally, we prove that the lemma holds when $B_{k'}$ and B_k are in the same epoch (i.e., $e' = e = \lceil k/\Gamma \rceil$). The proof idea follows that from Lemma 4.1 of [82]. However, the proof in [82] is only for the non-pipelined version of HotStuff-2, thus we reprove it here for completeness: First, if $v' = v$, $B_{k'}$ must equal or extend B_k by the quorum intersection argument in view v. Suppose any blocks that obtain QCs from view v to $v'-1$ extend or equal B_k. For the inductive case $v' > v$, since B_k is committed in view v, by the commit rule, it must be that a subset S of the staked nodes in epoch e with stake weight at least $2\mathbf{tN}_e + 1$ have locked a block that equals or extends B_k at the end of view v (where $\mathbf{tN}_e$ is the Byzantine stake threshold for epoch e). By the induction hypothesis and by the locking rule, at the end of view $v'-1$, the honest nodes in S will still lock a block that equals or extends B_k. In view v', let w denote the stake weight controlled by the Byzantine nodes in set S, and let w' denote the stake weight of nodes in set S that are bribed by the adversary. By definition of the threat model, it holds that $w + w' \leq \mathbf{tN}_e$ and thus the non-bribed honest nodes in set S have weight at least $\mathbf{tN}_e + 1$. Moreover, since the adversary never uncorrupts a node, the set of non-bribed honest nodes in S *still remember the highest locked block* in view $v'-1$. Now assume for contradiction that the proposed $B_{k'}$ in view v' does not equal or extend B_k. By the inductive hypothesis, the block extended by $B_{k'}$ can only have a QC with view less than v. Recall that $B_{k'}$ and view v' are still in epoch e by assumption, and there are non-bribed honest nodes with weight at least $\mathbf{tN}_e + 1$ having locked blocks with view numbers no less than v. Thus $B_{k'}$ will not get enough votes in view v' to obtain a QC, contradiction. □

Remark 2.4. *The restriction that an adversary cannot uncorrupt Byzantine nodes per epoch is essential for the above proof. Otherwise, the uncorrupted honest nodes may not remember important meta-data (e.g., the highest locked block) they had before, and there would not be enough honest*

[6]Note that the honest nodes in epoch a would still be online/staked until the first block in epoch $a+1$ is committed, thus before committing the first block in epoch $a+1$, the adversary cannot corrupt a quorum set of nodes in epoch a to forge a last committed block in epoch a. This avoids the possibility of a long-range attack, where the adversary corrupts the set of nodes that were staked in the past and quickly creates a fork to history.

nodes guarding the safety. Moreover, the adversary might remember the long-term secret keys of the uncorrupted nodes and forge messages on behalf of them. We refer to Remark 2.2 for more discussion on the choice of the adversary model.

Theorem 2.2 (Safety)**.** *Every two staked nodes commit the same block for every block height h.*

Proof. Assume for contradiction that the two nodes commit to two different blocks B_h and B'_h at height h. Let B_k (and $B_{k'}$) be the first certified block that equals or extends B_h (and B'_h) and subsequently obtains 2 consecutive QCs, respectively. Let v and v' be the views that B_k and $B_{k'}$ being committed. WLOG assume $v \leq v'$, and $k' \geq k$ if $v = v'$. By Lemma 2.1, $B_{k'}$ must extend or equal B_k, which implies $B_h = B'_h$, contradiction. □

Theorem 2.3 (Liveness)**.** *Assuming all honest nodes are synchronized in consecutive views v, $v+1$, $v+2$ for a sufficiently long period of time after GST, and the leaders L_v, L_{v+1}, L_{v+2} in three views are honest, the honest nodes would commit a new block in view $v+2$.*

Proof. Consider time after GST. Let B_h (with block height h) be the block proposed by the honest leader L_v in view v, and let $e = \lceil h/\Gamma \rceil$ denote the epoch for block B_h. We consider two cases:

- $h \bmod \Gamma = 1$, that is, B_h is the first block to be committed in epoch e, and B_h extends the last committed block B_{h-1} in epoch $e-1$. In this case, with similar arguments in the proof of Lemma 2.1, there would be unanimously agreed staking sets for epoch e and $e-1$. Moreover, the honest nodes in epoch $e-1$ would send the eQCs for the block B_{h-1} to the leaders in epoch e until the first block in epoch e has been committed. Thus it's guaranteed that the honest leader L_v would be able to receive B_{h-1} and propose B_h. Finally, all honest staking nodes in epoch e would accept and vote for B_h given that it extends the last committed block B_{h-1}.[7]
- B_h is not the first block to be committed in an epoch. In this case, given enough synchronization time, the honest nodes would send their highest locked block to L_v, and the leader would receive all of them in view v (as GST has passed). Hence the proposal B_h would extend the highest locked block over all honest nodes, and all honest nodes would vote on this block.

In both cases, the quorum certificate $C_v(B_h)$ will be formed. With similar arguments for views $v+1$ and $v+2$, there will be blocks B_{h+1} and B_{h+2} extending B_h such that the QCs $C_{v+1}(B_{h+1})$ and $C_{v+2}(B_{h+2})$ are formed in view $v+1$ and $v+2$ respectively. (A special case is when B_h is the last committed block in an epoch. In this scenario, we have $B_{h+2} = B_{h+1} = B_h$.) By the commit rule, the honest nodes will commit B_h at the end of view $v+2$, which completes the proof. □

Remark 2.5. *The restriction that an adversary cannot uncorrupt Byzantine nodes per epoch is essential for the liveness proof. Otherwise, in the case where B_h is not the first block in an epoch, the set of honest nodes that send the highest locked block can be completely different from the set of honest nodes that vote for the block. It might be the case that the nodes in the latter set have higher locked blocks and won't vote for the block that the leader L_v proposed.*

Remark 2.6. *The assumption of three consecutive honest leaders is reasonable. The total weight of corruption is bounded per epoch, and the adversary cannot immediately corrupt one of the honest leaders in three views after knowing the leader's identity if the adversary runs out of the corruption budget (i.e. $\boldsymbol{N}_e/\boldsymbol{t}$ where $\boldsymbol{t} = 1/3$ is the threshold of corruption ratio). Moreover, since the leaders are chosen at random, the probability that three consecutive leaders are honest is at least $(1-\boldsymbol{t})^3$.*

Remark 2.7. *The assumption that all honest nodes will be in the same view for a sufficient amount of time after GST is reasonable. After GST, if the honest leader L_v receives a valid QC or a timeout certificate from the previous view, or receives a valid extended QC for the last epoch, L_v sends the certificate to other nodes, and the honest nodes enter the new view after seeing the certificate. If no evidence certificate is received after a timeout, an honest node will trigger the Naor-Keidar protocol [90].[8] Note that the NK protocol is never run across different epochs and the set of participants is fixed (i.e., the epoch staking set) in which the malicious nodes account*

[7]The index of the last committed block of an epoch is implicit and thus a malicious leader in epoch $e+1$ cannot extend an older block.

[8]As noted in Sect. 2.4.3, the honest node that triggers the viewsync protocol can be from the previous epoch, but the NK protocol is still run over the nodes in the current epoch.

for weight less than $\mathbf{t}\boldsymbol{N}_e$. *Thus the honest nodes will be in sync given the security of the Naor-Keidar protocol. Note that the communication complexity of the NK protocol could be quadratic if the adversary adaptively corrupts a linear number of random relays. Fortunately, the adversary can never uncorrupt nodes and the accumulative adversarial weight is bounded by* $\mathbf{t}\boldsymbol{N}_e$ *per epoch, hence the total number of adaptive corruptions over the entire epoch is still linear, and the total communication complexity for viewsync is still quadratic in the worst case. This means that the amortized complexity can stay linear if the number of views per epoch is large enough.*

[[Binyi: todo: discuss communication complexity and latency under different adversarial scenarios.]

3 Tiramisu Data Availability

3.1 Overview

In a conventional consensus protocol each node votes to finalize a new block only after it has seen the entire data payload for that block. This naive strategy limits the throughput and latency of the protocol because a large payload must be received by a supermajority of nodes in order to finalize a block. If there are n nodes in the network and the payload has size $|B|$ then the total network communication for this strategy is $O(n\,|B|)$, which conflicts with our goal of $O(|B|)$ communication. In order to unlock further improvements in throughput and latency we must relax the conventional requirement that each voting node see the entire payload pre-finalization.

This desire for improved performance is in tension with our desire for security against powerful bribery attacks on the network. If we allow a block to be finalized before its full payload has reached a quorum of nodes then a bribing adversary might corrupt the small number of nodes that saw the entire payload pre-finalization. In that case, the block will be finalized but its payload is forever lost—a catastrophic failure of liveness.

This tension is the essence of the *data availability (DA) problem.* In this section we propose *Tiramisu*—an efficient, three-layer solution to the DA problem. Each layer represents a point on the security-performance tradeoff curve. (See Figure 5.)

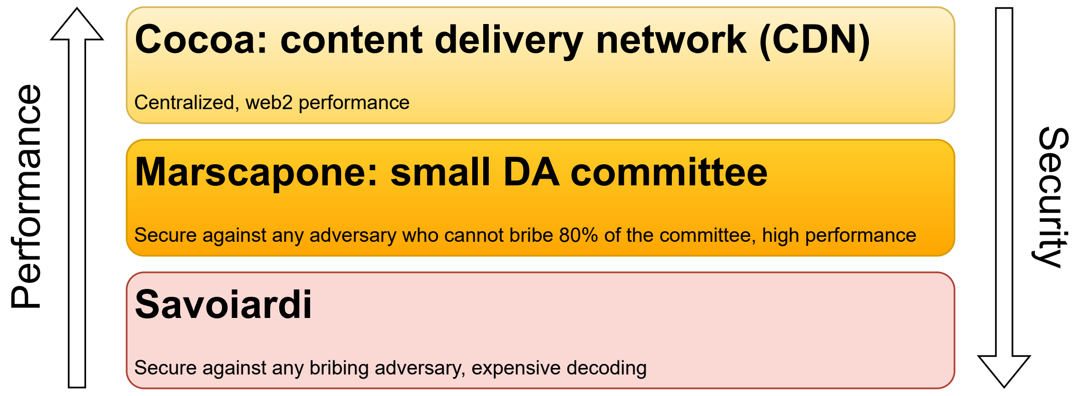

Performance
Cocoa: content delivery network (CDN)
Centralized, web2 performance
Marscapone: small DA committee
Secure against any adversary who cannot bribe 80% of the committee, high performance
Savoiardi
Secure against any bribing adversary, expensive decoding
Security

Figure 5: Tiramisu: a three-layer solution for data availability (DA).

Base layer. *Savoiardi: bribery-resilient DA.* In rare cases the network operates under *pessimistic* conditions: the network is under active attack by a powerful active or bribing adversary, and the other layers of Tiramisu might cease to function for the duration of this attack.

In this case we rely on a solution similar to Ethereum's Danksharding proposal [96]. The proposer of a new block encodes its payload under an erasure code. Then, instead of broadcasting the entire size-$|B|$ payload to all n nodes ($O(n\,|B|)$ communication), the proposer partitions the encoded payload into small shares and distributes only these small shares among a $O(|B|)$-size subset of nodes in the network ($O(|B|)$ communication). A full description of Savoiardi is given in Appendix A.

Security. This erasure-coded payload has the property that the entire payload can be recovered from any sufficiently large subset of shares. Thus, even if a powerful adversary corrupts many nodes, the remaining honest nodes are able to recover the payload.

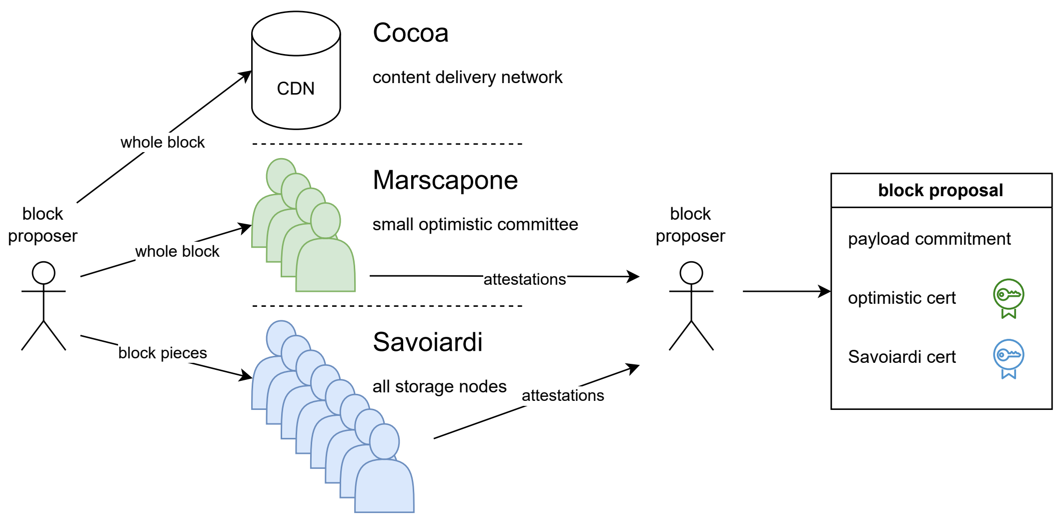


Figure 6: Tiramisu flow. Block proposer disperses block among storage nodes, aggregates attestations into certificates, builds a candidate HotShot block.

Performance. A disadvantage of Savoiardi is that no single node has the entire payload. A client who wishes to recover the full payload must download shares from many storage nodes and spend computation resources to decode the payload. Thus, while Savoiardi is the ultimate defense against powerful adversaries, for performance reasons its use should be avoided in all but the worst network conditions.

Middle layer. *Mascarpone: a small DA committee for fast consensus.* Most of the time we expect the network to operate under *optimistic* conditions: the network is not under attack by a powerful active or bribing adversary.

In such cases the network can avoid Savoiardi's expensive payload recovery process and instead rely on a small, constant-size DA committee. The block proposer or other participant uploads a new block's payload to the committee, and the block is finalized after receiving attestations from a quorum of committee members[9].

Security. Members of the committee are selected at random, and the committee is replaced at the beginning of each epoch. Randomness for this selection is sourced from the decentralized random beacon (DRB) as described in Section 2.4.4. The size of the committee is chosen so that if a passive adversary corrupts at most a $\mathbf{t}$ fraction of all stake then a at least one committee member is honest with overwhelming probability. Thus, the Mascarpone layer is secure against all but the most powerful adversaries. It can be compromised only by an adaptive adversary who can quickly corrupt the committee before it is replaced with a new one, or a bribing adversary who can corrupt the committee immediately.

Top layer. *Cocoa: Web2 performance from a CDN.* We can further improve performance under optimistic network conditions via a centralized content distribution network (CDN).

This solution is simple: the block proposer uploads a new block's payload to the CDN, and anyone who wants the block payload may request it from the CDN much like a web2 streaming service consumer might download a video.

The CDN can also serve as an ultra-fast channel for passing messages between the block proposer and other nodes for use cases such as collecting attestations for the quorum certificate described in Section 4.2.

A centralized CDN might be an easy target for an attacker, or it could experience occassional downtime even in the absence of any malicious actor. As such, the Cocoa of a CDN is well-supported by the Mascarpone layer below it when the CDN us unavailable but network conditions are otherwise optimistic.

[9]This is the step that may fail in the presence of a bribing adversary, but will succeed in optimistic conditions as mentioned above.

3.1.1 Communication complexity

Tiramisu achieves asymptotically optimal $O(|B|)$ total network communication, summed over all three layers of the protocol:

Base layer (Savoiardi). The Savoiardi erasure-code dispersal scheme achieves $O(|B|)$ as described in Appendix A.6.

Middle layer (Mascarpone). The full block payload is sent to each member of the Mascarpone committee. The size of this committee is constant. (Say, 10–200 nodes.) As such, total network communication for this layer is $O(|B|)$.

Top layer (Cocoa). Like Mascarpone, the full block payload is sent to only a constant number of nodes. In Cocoa, the payload is sent to only a single node—the CDN. Thus, total network communication for this layer is $O(|B|)$.

3.2 Syntax

It is convenient to describe Tiramisu as a protocol for *verifiable information dispersal (VID).* VID schemes have the following syntax:

`Commit` : $B \mapsto C$**.** The algorithm `Commit` takes as input a block payload B and produces as output a succinct commitment C.

`Disperse` : $B \mapsto \mathit{cert}(C)$**.** The protocol `Disperse` is run by a *payload sender* and multiple *storage nodes.* It takes as input a block payload B from the sender and produces as output either "fail" or a retrievability certificate $\mathit{cert}(C)$ to the sender.

`Retrieve` : $(C, \mathit{cert}(C)) \mapsto B$**.** The protocol `Retrieve` is run by a *client* and the storage nodes. It takes as input a commitment C and a retrievability certificate $\mathit{cert}(C)$ for C from the client and produces as output either "fail" or a block payload B to the client.

There are many different combinations of security properties a VID scheme might possess. All VID schemes need some form of *correctness* (an honest party always retrieves an honestly dispersed block) and *availability* (an honest party always retrieves some block, even in the presence of a malicious sender). The reader is referred to Ref. [93, Section 1.4] for discussion of VID security properties.

The security properties required for Tiramisu are:

Commitment binding. `Commit` is a deterministic binding commitment to block payload B.

Correctness. If an honest sender initiates `Disperse` for block payload B then eventually it terminates and obtains a retrievability certificate $\mathit{cert}(\texttt{Commit}(B))$.

Availability. If an honest client initiates `Retrieve` for valid commitment C and retrievability certificate $\mathit{cert}(C)$ then eventually it terminates and obtains a block payload B with $\texttt{Commit}(B) = C$. (This property holds even if $\mathit{cert}(C)$ was produced by a malicious sender.)

3.3 Construction

In this section we describe Tiramisu VID operations `Tiramisu.Commit`, `Tiramisu.Disperse`, `Tiramisu.Retrieve`. These operations use the `Savoiardi` VID subprotocol as a black box, and are quite simple otherwise. Thus, we provide only an informal description of these operations in terms of the `Savoiardi` black box. A more formal description of the `Savoiardi` black box is given in Appendix A. (See Figure 6.)

3.3.1 Commit

Tiramisu places no additional restrictions on `Commit` beyond that which is required for the `Savoiardi` black box. Thus, for our purpose in this section it suffices to leave `Tiramisu.Commit` unspecified with the understanding that `Tiramisu.Commit` = `Savoiardi.Commit`. To ground the present discussion it might help to think of `Tiramisu.Commit` merely as a hash of the block payload B.

3.3.2 Disperse

`Tiramisu.Disperse`(B) is a one-round interactive protocol between the block sender and storage nodes. There are three types of storage nodes: oridnary storage nodes (Savoiardi), the *small DA committee* nodes (Mascarpone), and the *content delivery network (CDN)* node (Cocoa).

Block sender. Given a block payload B, do the following:

1. Compute the payload commitment $C \leftarrow \texttt{Commit}(B)$.
2. Initiate all three layers of Tiramisu concurrently:

   Savoiardi. Execute `Savoiardi.Disperse`(B) with all storage nodes.

   Mascarpone. Upload (B, C) to the small DA committee.

   Cocoa. Upload (B, C) to the content delivery network (CDN).

All storage nodes (Savoiardi). Specified in `Savoiardi.Disperse` in Appendix A.

Small DA committee nodes (Mascarpone). On receiving (B, C) from the sender, do the following:

1. Verify $C = \texttt{Commit}(B)$. If not then abort.
2. Respond to the block sender with a signed attestetion of C.
3. Gossip (B, C) to the other nodes in the small DA committee.

Content delivery network (CDN) (Cocoa). The job of the CDN is simple: on receiving (B, C) from from the sender, be ready to serve (B, C) to anyone who requests it. The CDN provides no formal security to Tiramisu, so there is no need for the CDN to respond to the sender after receiving (B, C).

Of course, there might be other reasons for the CDN to respond to the sender. For example, it might help for performance reasons to stipulate that the CDN acknowledge receipt of (B, C). We consider such a response to be an implementation detail.

Block sender. Await responses from storage nodes, then compose a certificate of availability:

1. Concurrently:

   Mascarpone. Wait for attestations of C from a quorum of small DA committee nodes. The Mascarpone certificate for C consists of an aggregation of these attestations.

   Savoiardi. Wait for attestations of C from a quorum of storage nodes as per `Savoiardi.Disperse`. The Savoiardi certificate for C consists of an aggregation of these attestations.

2. The Tiramisu certificate $\mathit{cert}(C)$ for C consists of both the Mascarpone and Savoiardi certificates for C.

3.3.3 Retrieve

`Tiramisu.Retrieve`$(C, \mathit{cert}(C))$ is a one-round interactive protocol between the client and the storage nodes. The client does the following.

1. Request the block payload from the CDN (Cocoa). If the CDN replies with a payload B with $\texttt{Tiramisu.Commit}(B) = C$ then return B.
2. Request the block payload from the small DA committee (Mascarpone). If the small DA committee replies with a payload B with $\texttt{Tiramisu.Commit}(B) = C$ then return B.
3. Execute `Savoiardi.Retrieve`(B) with the storage nodes (Savoiardi).

3.4 How HotShot uses Tiramisu

HotShot's use of Tiramisu is simple. During each view, the HotShot consensus leader will attempt to finalize a single block B. A block cannot be finalized without an accompanying certificate of availability. Such a certificate is obtained by the leader by executing $\texttt{Tiramisu.Disperse}(B)$ with the storage nodes. The resulting block commitment $C = \texttt{Tiramisu.Commit(B)}$ and its corresponding certificate $\texttt{cert}(C)$ are included on-chain. As discussed previously, the payload B is too big to fit on-chain, but the security properties of `Tiramisu.Disperse` ensure that B is available to anyone who wants it.

In this section we clarify certain details on HotShot's use of Tiramisu.

3.4.1 Liveness

Recall that `Tiramisu.Disperse` succeeds only if both

1. The block proposer collected sufficiently many signatures from the small DA committee to produce a Mascarpone certificate, and
2. `Savoiardi.Disperse` successfully produced a Savoiardi certificate.

Savoiardi is guaranteed to succeed by the properties of the Savoiardi scheme and the HotShot threat model. But Mascarpone could fail if a sufficiently powerful adversary corrupts the optimistic DA committee so that it is unable to produce a certificate. In this case, no block can be finalized for this HotShot view—a temporary failure of liveness.

Because the optimistic DA committee is selected at random and frequently refreshed, the only way an adversary can reliably cause this the committee to fail is to execute an expensive adaptive or bribery attack. As discussed in Section 2.2, these attacks exhaust the adversary's budget, after which liveness recovers immediately.

To put this attack in perspective, observe that the adversary could cause a similar liveness failure more cheaply by attacking only the *leader* for this view, so this avenue for attack cannot further weaken HotShot liveness.

3.4.2 Forcing expensive data recovery

Assume that the Cocoa layer of Tiramisu (CDN) is not functioning for some reason, as there can be no problem retrieving payload data when the CDN is functioning. Of all the layers of Tiramisu, Savoiardi is the only layer that is secure against an adaptive or bribing adversary. Thus, we expect there must exist an attack whereby the adversary forces the network to rely on the expensive Savoiardi scheme to recover the block payload.

The only such attack is to corrupt the small DA committee so that it produces a certificate, yet is unwilling or unable to deliver the payload upon request. In Section 3.4.1 we observed that liveness attacks exhaust the adversary's budget. The same principle applies to attacks that force expensive data recovery.

3.4.3 Optimization: no need for Savoiardi certificate

For ease of exposition it is convenient to stipulate that the certificate $\mathit{cert}(C)$ produced by `Tiramisu.Disperse` include both the Mascarpone (small DA committee) and Savoiardi certificates. But in HotShot the Savoiardi certificate is not strictly required. Because most or all HotShot nodes are also Savoiardi storage nodes, these nodes can cast a vote in HotShot for a candidate block even if they have seen only (i) the Mascarpone certificate, and (ii) their own Savoiardi share of the encoded block. Under this scheme, if the HotShot vote passes then it must also be the case that a quorum of Savoiardi storage nodes has each seen its Savoiardi share, so there's no need to compile these attestations into a certificate.

3.4.4 Policy on payload dissemination, retrieval, and retention

HotShot uses Tiramisu to ensure only that it is possible *for some parties* to retrieve the block's full payload *at the time that block is finalized.*

Why only some parties? This question is about *dissemination* and *retrieval*: who is *required* to download the full payload, and who is *allowed* to download the full payload?

Because HotShot does not execute the transactions it sequences, there is no need for most nodes on the network to download the full payload of every block. Retrieval of the full payload is required only for those few powerful network nodes in the DA committee (Mascarpone) or the CDN (Cocoa). A fortunate consequence of this lax requirement is that there is no concern that HotShot must accommodate slow nodes in order to achieve security. Indeed, we expect that most HotShot nodes will *not* download the payload for most blocks.

Who is allowed to download the full payload? This question is relevant because if there is no restriction on who may initiate payload retrieval then perhaps the system is open to a denial-of-service attack whereby a malicious actor floods the network with requests for data. This problem is not unique to HotShot—it is relevant to any permissionless system, including all public blockchains. A solution to this problem is outside the scope of this document. For our purpose it suffices to observe that different protocols enact different retrieval policies to suit their needs. A common approach is that each network node begins with a permissive policy that sends data to anyone who requests it, but the node watches for signs of abuse and may unilaterally decide to stop servicing requests.

Why ensure availability only at the time the block is finalized? This question is about *retention*. The availability of historical payload data that was finalized in the past is a separate question from the availability of current payload data for a new candidate block. An explicit retention policy is beyond the scope of this document. For our purpose it suffices to observe that retrieval of historical data could be provided by other actors such as an archival service or a decentralized data storage platform.

4 Capitalizing on Responsiveness with a CDN

In this section, we describe a core technique applied across HotShot Consensus (Section 2) and Tiramisu DA solution (Section 3): the hybrid networking layer with *Content Delivery Network* (CDN) and *peer-to-peer* (P2P) for optimistic performance and resilient fallback. While the usage of CDN seems like an engineering optimization, we will demonstrate its high congruence with our protocol and explain how it can turbocharge our practical performance. The remainder of this section starts with describing existing popular choices of networking for blockchains whose limitations motivate our "CDN+P2P" design. Then, we establish the basics and key properties of a CDN. Finally, we elaborate on why it is a perfect match for our responsive protocol and how we utilize it concretely.

4.1 CDN + P2P: A Hybrid Blockchain Networking

The networking layer is a key component for the performance and security of a blockchain. Most blockchain protocols run over a decentralized P2P network [46], usually composed of nodes with varying computing power, storage capacity and bandwidth that can join and leave at any time. The heterogeneity, the multi-hop routing, and the high churn rate of a P2P network introduce latency which makes the propagation of transactions and blocks slow. For example, it was observed that the Bitcoin P2P network is not bandwidth efficient [91] and that safety can be compromised when trying to increase throughput [43]. While there are blockchains (e.g., DiemBFT, Aptos[10]) that maintain a full-mesh network where each consensus node has a full membership view and direct connection with all other nodes, thereby obtaining low latency, it is impractical to scale these blockchains to a much larger network size due to their quadratic number of connections.

In contrast, modern networking by Web2 businesses aims to serve large amounts of data (e.g. videos) to millions of users with sub-second latency by relying on a CDN infrastructure. A CDN consists of a set of interconnected servers which can route information efficiently to geographically distributed clients, leveraging powerful hardware, high bandwidth connections, and advanced caching techniques [102]. While CDNs are robust and offer close to optimal service, they are inherently centralized and thus cannot be considered as a standalone alternative for blockchain networking.

Despite the intricate internals and engineering subtleties of a CDN, it's sufficient to use a simplified, abstract model for our purposes. We think of CDN as a powerful but centralized hub

[10]`https://github.com/aptos-labs/aptos-core/blob/5b06b64cd7823bd8150274237778ec49bf18da14/network/README.md`. In practice, Aptos only has around 100 validators whereas HotShot is targeting tens of thousands of nodes

interposed between all nodes offering fast message delivery and delegated computation. The delegated computation is enabled by the powerful hardware of CDN servers and could be especially handy in leader-based protocols where part of the leader's task (such as signature aggregation) is computationally intensive and would benefit from delegation.

In order to overcome the "blockchain trilemma"[26] at the network level, we combine a CDN with a P2P network for a hybrid infrastructure. The idea is to rely on the CDN most of the time under good networking conditions in order to get the best possible throughput and latency. In the rare case of CDN failures or severe performance degradation, we fall back to the P2P network that is less performant but more robust and will guarantee liveness until the CDN recovers. This hybrid approach allows us to get the best of both worlds reflected by the list of properties below:

- **Optimal optimistic performance (high throughput, low latency).** Under optimistic network conditions with a functioning CDN, we enjoy optimal Web2 performance. The low latency comes partially from the deployment of a distributed set of *Point of Presence* (PoP) [11] that significantly reduces routing distances between a client and the requested data. The optimized caching mechanism prevalent in CDN solutions achieves high *cache hit ratio* even for dynamic content, further reducing the average delivery time. The high throughput derives from intelligent load balancing in software and high bandwidth in hardware.
- **Security (safety and liveness).** Modern CDN incorporates numerous mitigation techniques for DDoS attacks, traffic hijacking, spoofing attacks, and man-in-the-middle attacks, and thus should offer great protection and is less vulnerable to eclipse attacks [63] as the P2P counterpart. Even with a failed CDN, our nodes gracefully fall back to the resilient P2P network to continue making progress and perform as well as most blockchains today (from the networking perspective).
- **Censorship resistance.** We emphasize that CDN and propagation through P2P operate *in parallel* and any data posted to CDN *could* be disseminated to P2P independently. This is necessary for censorship resistance in case of a corrupted, package-dropping CDN, thanks to the eventual delivery over the slower yet more resilient P2P path.
- **Uncompromising scaling.** Mature CDN clusters can horizontally scale and handle internet-scale traffic without compromising performance or security.
- **Maximizing utilities of responsiveness and linearity.** This is the most relevant and compelling reason for our adoption of CDNs. Responsive protocols can progress at network speed in the optimistic case without waiting for any a priori timeout, thus can benefit the most from a blazing fast underlying network. Even though a faster network is advantageous for any consensus in general, its amplification would be offset by the fixed timeout during view change per round in Tendermint [23] for example. Furthermore, the linear communication complexity of HotStuff-2 and transitively our HotShot consensus matters less if we are using a P2P network or if the network size is small (e.g., only a hundred). In P2P networks, communication still incurs a logarithmic cost due to the expected number of hops before reaching the destination; and broadcast via flooding algorithm lowers effective throughput due to redundant messages. Fortunately, CDN delivers data with expected-constant hops (fewer than that in P2P), linearity can truly showcase its merit as the practical complexity remains linear with small concrete parameters and the saving is even more pronounced as we scale to larger networks with tens of thousands of nodes.

We remark that bloXroute [71] is a relevant prior work on improving the networking layer for general blockchains. We share similarities in our hybrid approach of using centralized (potentially distributed) infrastructure for performance and P2P network for resilience and security – the bloXroute relay network corresponds to our CDN layer which incorporates lots of optimization techniques from modern internet infrastructure; where the P2P network in both designs exists as a resilient fallback. However, our networking is designed with our specific consensus in mind whereas bloXroute aims to be a blockchain-agnostic network layer. Therefore we can further customize our CDN for computation like signature aggregation beyond message delivery whereas content transmitted in bloXroute's CDN-equivalent network is encrypted and cannot be computed over. Moreover, bloXroute provides stronger neutrality guarantees and mitigates sender-based and content-based censorship by the CDN which comes at a slight latency penalty for all packets; whereas we use CDN only in the optimistic case and thus do not require such strong properties. It

[11] See Fastly's network of PoPs: `https://www.fastly.com/network-map/`

remains as future works to provide a more qualitative comparison and an evaluation of our system when switching to bloXroute.

4.2 Integrating a CDN with the Espresso Sequencer

With the properties of CDN and its synergy with the HotShot consensus explained in Section 4.1, we elaborate on the concrete integration across our design stack in this section.

P2P network. Our P2P network is an instance of GossipSub [115]. GossipSub is well suited for blockchain P2P protocols as it optimizes bandwidth usage while mitigating common adversarial behaviors such as Sybil, censorship and eclipse attacks. At a high-level GossipSub organizes the network in a set of meshes where each node is directly connected to each other. Communication between meshes and nodes not belonging directly to any mesh is done via gossiping where a message is sent to a random subset of the peers. Additionally, each node maintains a score function of its peers aiming at detecting malicious behavior and reconfiguring the network accordingly. As our protocol is a PoS blockchain and leader based, we extend the GossipSub networking infrastructure by: (1) authenticating the Distributed Hash Table[12] information about peers, in particular, IP addresses of a node are signed with the corresponding staking key and (2) we ensure that IP addresses corresponding to nodes (and more importantly leaders) are well spread across the meshes.

CDN. Our CDN is currently instantiated with a master server, whose duties are listed below, and a set of point of presence (PoP) servers:

- Assist the leader for expensive computations such as quorum certificate construction (see Appendix B.3.3). [Philippe: Should we sketch the solution consisting of building the aggregate signature bottom up in an optimistic fashion?]
- Deliver broadcast messages such as leader proposals or point-to-point messages such as partial signatures or vote back to the leader).
- Assign PoPs to node clusters based on their geographical location and other parameters. [Philippe: Maybe in order to distribute VID shares more effectively. Do we want to attempt a specification of this?].

Broadcast messages are sent to the master server and further forwarded to all nodes. Nodes initially connect to a dedicated naming server to obtain their assigned PoP, and then primarily interact with this PoP onward.

Message delivery. Under a functional CDN, all messages are delivered by CDN and are optionally sent to P2P if the sender wants extra censorship protection and redundancy. Switching to P2P network does not require explicit agreements, but happens fluidly as more nodes detect the failure of the CDN and start primarily using P2P for all messages. The two networking options exhibit different throughput/latency profiles; failure and degradation in any or both of them will never violate safety and failure of the CDN will at most stall liveness temporarily under our threat model.

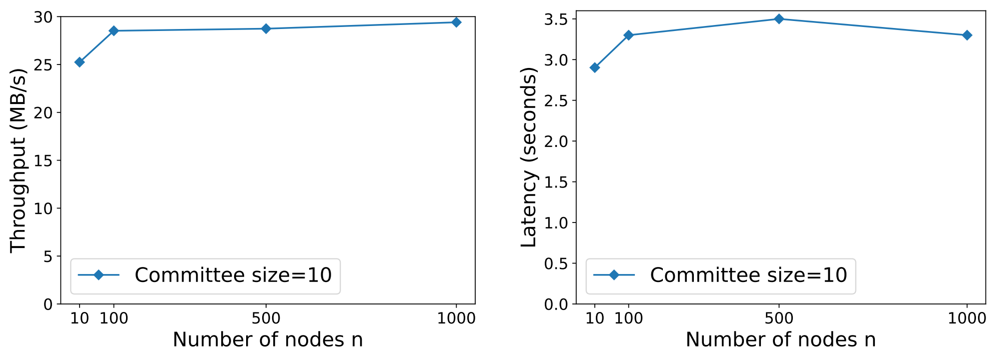
5 Preliminary Evaluation

In this section, we present a (very) preliminary evaluation of our current implementation.

Implementation details. We have an open-sourced implementation of a version of HotShot and Tiramisu in Rust.[13] Our HotShot implementation adapts the HotStuff protocol and currently assumes a fixed set of permissioned sequencer nodes of size n whose public keys are known to each other. The nodes engage in running the steady state protocol where leaders switch in a round-robin manner and engage in view synchronization if they go out of sync. In particular, our current implementation does not obtain staking keys from Ethereum, implement epochs, perform any reconfiguration. The quorums are thresholds of size $> \frac{2}{3}n$ instead of being weighted by stake, and we currently do not aggregate these signatures. The Tiramisu implementation consists of the Cocoa and Mascarpone layer but does not include Savoiardi. The set of nodes participating

[12] We use Kademlia[83].

[13] Code available here: https://github.com/EspressoSystems/HotShot/tree/3c3b8a6f9a4a29c35f51fd13085b844490dff652



(a) Throughput (b) Latency

Figure 7: Throughput and latency at varying n and a fixed Mascarpone committee

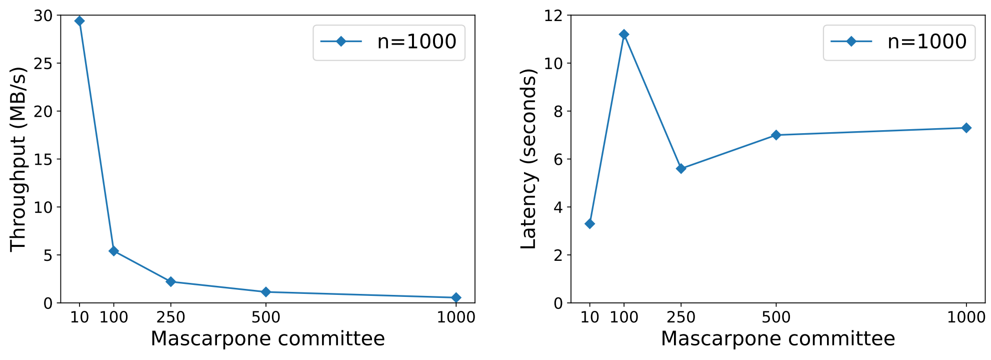
in the Mascarpone layer are fixed and known to all other nodes. In terms of networking, all communication is currently routed through the optimistic CDN layer and there is no fallback P2P network. Moreover, the CDN is not currently used for any expensive computation.

Experimental setup. All our experiments were conducted over Amazon AWS in the us-east-2 region [11]. Our experiments use two CDN instances each fronted by Nginx [95]: one for HotShot messages and one for Tiramisu messages. Each CDN is an m6a.xlarge EC2 instance. Each has 4 vCPUs, 16 GB of memory, and up to 12.5 Gbit/s of bandwidth. Nodes poll the CDNs using HTTP/1.1 at 100ms intervals. Nodes comprising the HotShot and Tiramisu network are each run as ECS Fargate tasks with 2 vCPUs and 16 GB of memory. AWS does not release bandwidth specifications for ECS Fargate tasks. This node hardware is not more powerful than the requirements of Ethereum nodes [50]. The ECS tasks are equally distributed across 3 availability zones in the us-east-2 AWS region.

Methodology. For each run, 10 nodes were selected as clients as separate async tasks in Rust to submit pre-generated transactions to all other nodes. The run lasts until 100 blocks are committed by *all* the nodes in the network. We compute the average view time as the ratio of the total run time and the number of views required in the run. The reported latency is three times the average view time, and the throughput is the ratio of the sum of sizes of all data blocks divided by the run time. We conduct three such runs for each data point and report the mean values across the three runs and the committed blocks.

To find the maximum throughput of our system under different configurations we varied the size of transactions being submitted to the network while keeping the interval of transaction submission the same. The leader aggregates the transactions in its queue to produce the next block. We experimentally chose small transaction sizes to start and doubled the transaction size so long as network throughput kept increasing. Once we reached an inflection point where throughput began to decrease, we chose transaction sizes equal to the median between the inflection point and the next closest point on either side of the inflection point. We recursively continued this binary search-inspired approach on either side of the inflection point until transaction sizes increased or decreased less than 2 orders of magnitude of the original inflection point transaction size. From there we found the new inflection point and maximum throughput.

Evaluation. With the given experimental setup, we evaluate the performance of the system when varying two key parameters: number of nodes n, and the Mascarpone committee size. Figure 7 presents the throughput and latency of the system when the size of the Mascarpone committee is fixed and number of nodes vary. As we can observe, the mean throughput is consistently in the range of ∼25-30MB/sec and the mean latency stays consistent in the range of ∼2.9-3.5secs. Thus, our performance does not degrade as the number of nodes in the network increases. Interestingly, the throughput for a ten-node network is considerably lower than when the node sizes are greater than 100. We note that this is an artifact of the current implementation since a fixed set of ten nodes are simultaneously responsible for generating transactions, performing their role for data availability, and for consensus. When n increases, most nodes involved in consensus are not engaged in the other two tasks.



(a) Throughput (b) Latency

Figure 8: Throughput and latency at varying Mascarpone committee sizes and a fixed n

Figure 8 presents the performance of the system with a fixed number of nodes and varying Mascarpone committee size. As can be seen, in our current implementation, as the committee size increases, the throughput decreases proportionally. This strongly, and not surprisingly, indicates that data dissemination in the Mascarpone layer is currently the bottleneck. The latency varies drastically as the committee size increases; this is an artifact of adjusting block sizes for a run to optimize for the throughput. These are known issues and will be addressed in the subsequent iteration of the implementation.

References

[1] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected $o(1)$ rounds, expected $o(n^2)$ communication, and optimal resilience. Cryptology ePrint Archive, Paper 2018/1028, 2018. `https://eprint.iacr.org/2018/1028`.

[2] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected o (1) rounds, expected communication, and optimal resilience. In *International Conference on Financial Cryptography and Data Security*, pages 320–334. Springer, 2019.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Dfinity consensus, explored. *Cryptology ePrint Archive*, 2018.

[4] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.

[5] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

[6] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:1809.09044*, 160, 2018.

[7] Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. Fraud and data availability proofs: Detecting invalid blocks in light clients. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*, pages 279–298. Springer, 2021.

[8] Andreea B Alexandru, Erica Blum, Jonathan Katz, and Julian Loss. State machine replication under changing network conditions. *Cryptology ePrint Archive*, 2022.

[9] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Asynchronous verifiable information dispersal with near-optimal communication, 2022. `https://eprint.iacr.org/2022/775`.

[10] Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. Succinct erasure coding proof systems, 2021. `https://eprint.iacr.org/2021/1500`.

[11] AWS. General purpose network performance. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html#general-purpose-network-performance`, Accessed: 2023-08-11.

[12] Vivek Bagaria, Amir Dembo, Sreeram Kannan, Sewoong Oh, David Tse, Pramod Viswanath, Xuechao Wang, and Ofer Zeitouni. Proof-of-stake longest chain protocols: Security vs predictability. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, pages 29–42, 2022.

[13] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 390–399, 2006.

[14] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *Public Key Cryptography—PKC 2003: 6th International Workshop on Practice and Theory in Public Key Cryptography Miami, FL, USA, January 6–8, 2003 Proceedings 6*, pages 31–46. Springer, 2002.

[15] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I*, pages 757–788. Springer, 2018.

[16] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in Cryptology—EUROCRYPT 2003:*

International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4–8, 2003 Proceedings 22, pages 416–432. Springer, 2003.

[17] Dan Boneh and Chelsea Komlo. Threshold signatures with private accountability. In *Advances in Cryptology–CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part IV*, pages 551–581. Springer, 2022.

[18] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *Advances in Cryptology—ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings 7*, pages 514–532. Springer, 2001.

[19] Joseph Bonneau. Why buy when you can rent? bribery attacks on bitcoin-style consensus. In *International Conference on Financial Cryptography and Data Security*, pages 19–26, 2016.

[20] Silvia Bonomi, Antonella Del Pozzo, Maria Potop-Butucaru, and Sébastien Tixeuil. Optimal mobile byzantine fault tolerant distributed storage. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 269–278, 2016.

[21] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, 1984.

[22] Jonah Brown-Cohen, Arvind Narayanan, Alexandros Psomas, and S. Matthew Weinberg. Formal barriers to longest-chain proof-of-stake protocols. In *Proceedings of the 2019 ACM Conference on Economics and Computation*, page 459–473. Association for Computing Machinery, 2019.

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

[24] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[25] Vitalik Buterin. Reed-solomon erasure code recovery in $n \log^2 n$ time with ffts, August 2018. `https://ethresear.ch/t/reed-solomon-erasure-code-recovery-in-n-log-2-n-time-with-ffts`.

[26] Vitalik Buterin. Why sharding is great: demystifying the technical properties. https://vitalik.ca/general/2021/04/07/sharding.html, 2021.

[27] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[28] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

[29] Wei Cai, Zehua Wang, Jason B Ernst, Zhen Hong, Chen Feng, and Victor CM Leung. Decentralized applications: The blockchain-empowered software system. *IEEE access*, 6:53019–53033, 2018.

[30] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, 1993.

[31] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[32] Benjamin Y Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. *Cryptology ePrint Archive*, 2023.

[33] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *Cryptology ePrint Archive*, 2018.

[34] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. *Cryptology ePrint Archive*, 2018.

[35] Jon Charbonneau. Rollups aren't real. `https://joncharbonneau.substack.com/p/rollups-arent-real`, Accessed: 2023-05-11.

[36] Yan Chen and Cristiano Bellavitis. Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights*, 13:e00151, 2020.

[37] Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, and Manuel Vidigueira. Every bit counts in consensus. *arXiv preprint arXiv:2306.00431*, 2023.

[38] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*, pages 23–41. Springer, 2019.

[39] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

[40] 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.

[41] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. Cryptology ePrint Archive, Paper 2021/777, 2021. `https://eprint.iacr.org/2021/777`.

[42] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2518–2534. IEEE, 2022.

[43] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10. IEEE, 2013.

[44] Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE access*, 7:28712–28725, 2019.

[45] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[46] Maya Dotan, Yvonne-Anne Pignolet, Stefan Schmid, Saar Tochner, and Aviv Zohar. Survey on blockchain networking: Context, state-of-the-art, challenges. *ACM Computing Surveys (CSUR)*, 54(5):1–34, 2021.

[47] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35:288–323, 1988.

[48] Ben Edgington. Upgrading ethereum: A technical handbook on ethereum's move to proof of stake and beyond, 2022.

[49] EigenLabs. Eigenlayer: shared security to hyperscale ethereum. `https://www.eigenlayer.xyz/`, Accessed: 2023-07-18.

[50] Ethereum.org. Run a node. `https://ethereum.org/en/developers/docs/nodes-and-clients/run-a-node/#requirements`, Accessed: 2023-08-08.

[51] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. {Bitcoin-NG}: A scalable blockchain protocol. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, pages 45–59, 2016.

[52] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Communications of the ACM*, 61(7):95–102, 2018.

[53] Dankrad Feist and Dmitry Khovratovich. Fast amortized kzg proofs, 2023. `https://eprint.iacr.org/2023/033`.

[54] Orr Fischer and Merav Parter. Distributed congest algorithms against mobile adversaries. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 262–273, 2023.

[55] Flashbots. The future of mev is suave. `https://writings.flashbots.net/the-future-of-mev-is-suave/`, Accessed: 2023-05-11.

[56] Peter Gaži, Aggelos Kiayias, and Alexander Russell. Stake-bleeding attacks on proof-of-stake blockchains. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 85–92. IEEE, 2018.

[57] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 296–315. Springer, 2022.

[58] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. Yoso: You only speak once: Secure mpc with stateless ephemeral roles. In *Annual International Cryptology Conference*, pages 64–93, 2021.

[59] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[60] Adam Groce, Jonathan Katz, Aishwarya Thiruvengadam, and Vassilis Zikas. Byzantine agreement with a rational adversary. In *Automata, Languages, and Programming: 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II 39*, pages 561–572. Springer, 2012.

[61] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

[62] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*, pages 147–176. Springer, 2021.

[63] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on Bitcoin's Peer-to-Peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144. USENIX Association, August 2015.

[64] James Hendricks, Gregory R Ganger, and Michael K Reiter. Verifying distributed erasure-coded data. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 139–146, 2007.

[65] Hai Jiang, Jun Li, Zhongcheng Li, and Xiangyu Bai. Efficient large-scale content distribution with combination of cdn and p2p networks. *International Journal of Hybrid Information Technology*, 2(2):4, 2009.

[66] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium*, pages 1353–1370, 2018.

[67] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*, pages 177–194. Springer, 2010.

[68] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.

[69] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference*, pages 357–388. Springer, 2017.

[70] Valerie King and Jared Saia. Breaking the o (n 2) bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):1–24, 2011.

[71] Uri Klarman, Soumya Sankar Basu, Aleksandar Kuzmanovic, and Emin Gün Sirer. bloxroute: A scalable trustless blockchain distribution network whitepaper, 2018. `https://bloxroute.com/wp-content/uploads/2018/03/bloXroute-whitepaper.pdf`.

[72] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th usenix security symposium (usenix security 16)*, pages 279–296, 2016.

[73] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1751–1767, 2020.

[74] Aztec Labs. Request for proposals: Decentralized sequencer selection. `https://discourse.aztec.network/t/request-for-proposals-decentralized-sequencer-selection/350/1`, Accessed: 2023-05-11.

[75] Offchain Labs. Arbitrum. `https://arbitrum.io/`, Accessed: 2023-05-11.

[76] Offchain Labs. Arbitrum decentralization update. `https://offchain.medium.com/arbitrum-decentralization-update-39f093768c42`, Accessed: 2023-05-11.

[77] Settler Labs. Astria. `https://www.astria.org/`, Accessed: 2023-05-11.

[78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978.

[79] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[80] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[81] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2159–2173, 2022.

[82] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[83] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers*, pages 53–65. Springer, 2002.

[84] Patrick McCorry, Alexander Hicks, and Sarah Meiklejohn. Smart contracts for bribing miners. In *IACR Cryptology ePrint Archive*, 2018.

[85] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 245–254, 2001.

[86] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. Cryptology ePrint Archive, Paper 2016/199, 2016. `https://eprint.iacr.org/2016/199`.

[87] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

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

[89] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv: Distributed, Parallel, and Cluster Computing*, 2019.

[90] Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr, 2020. `https://arxiv.org/abs/2002.07539`.

[91] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Sasha Fedorova, and Ivan Beschastnikh. Bandwidth-efficient transaction relay for bitcoin. *arXiv preprint arXiv:1905.10518*, 2019.

[92] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. *arXiv preprint arXiv:2002.11321*, 2020.

[93] Kamilla Nazirkhanova, Joachim Neu, and David Tse. Information dispersal with provable retrievability for rollups. *arXiv preprint arXiv:2111.12323*, 2021.

[94] Aztec Network. Aztec. `https://aztec.network/`, Accessed: 2023-05-11.

[95] Nginx. Deliver modern applications at scale with nginx. `https://www.nginx.com/`, Accessed: 2023-08-08.

[96] Valeria NikolaenkoDan and Dan Boneh. Data availability sampling and danksharding: An overview and a proposal for improvements. `https://a16zcrypto.com/posts/article/an-overview-of-danksharding-and-a-proposal-for-improvement-of-das/`, Accessed 2023-07-12.

[97] Optimism. Optimism. `https://www.optimism.io/`, Accessed: 2023-05-11.

[98] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 51–59, 1991.

[99] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. Cryptology ePrint Archive, Paper 2016/917, 2016. `https://eprint.iacr.org/2016/917`.

[100] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM symposium on principles of distributed computing*, pages 315–324, 2017.

[101] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*, pages 3–33. Springer, 2018.

[102] Al-Mukaddim Khan Pathan, Rajkumar Buyya, et al. A taxonomy and survey of content delivery networks. *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report*, 4(2007):70, 2007.

[103] Youcai Qian. Randao: A dao working as rng of ethereum. `https://github.com/randao/randao`, 2018. Accessed 26-July-2023.

[104] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[105] Peiyao Sheng, Bowen Xue, Sreeram Kannan, and Pramod Viswanath. Aced: Scalable data availability oracle. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*, pages 299–318. Springer, 2021.

[106] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 839–857, 2020.

[107] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness, 2023.

[108] 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.

[109] Starkware. Starknet. `https://www.starknet.io/`, Accessed: 2023-05-11.

[110] Espresso Systems. The espresso sequencer. Accessed: 2023-05-11.

[111] DFINITY Team et al. The internet computer for geeks. *Cryptology ePrint Archive*, 2022.

[112] TD Team et al. State machine replication in the diem blockchain, 2021.

[113] Scroll Tech. Scroll. `https://scroll.io/`, Accessed: 2023-05-11.

[114] Polygon Technology. Polygon zkevm. `https://polygon.technology/polygon-zkevm`, Accessed: 2023-05-11.

[115] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. Gossipsub: Attack-resilient message propagation in the filecoin and eth2.0 networks, 2020.

[116] Anton Wahrstätter, Liyi Zhou, Kaihua Qin, Davor Svetinovic, and Arthur Gervais. Time to bribe: Measuring block construction market. *arXiv preprint arXiv:2305.16468*, 2023.

[117] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447*, 2021.

[118] Sam M Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J Knottenbelt. Sok: Decentralized finance (defi). *arXiv preprint arXiv:2101.08778*, 2021.

[119] Barry Whitehat. zkrollup, 2018. `https://github.com/barryWhiteHat/roll_up`, Accessed: 2023-05-11.

[120] Barry Whitehat, Alex Gluchowski, Yondon Fu, and Philippe Castonguay. Rollup rollback snark side chain 17000 tps, 2018. `https://ethresear.ch/t/roll-up-roll-back-snark-side-chain-17000-tps/3675`, Accessed: 2023-05-11.

[121] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[122] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:1909.11261*, 2019.

[123] 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, Renton, WA, April 2022. USENIX Association.

[124] 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.

[125] 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.

[126] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 90–105. IEEE, 2020.

[127] Mingchao Yu, Saeid Sahraei, Songze Li, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. Coded merkle tree: Solving data availability attacks in blockchains. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers*, pages 114–134. Springer, 2020.

[128] Moti Yung. The" mobile adversary" paradigm in distributed computation and systems. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 171–172, 2015.

[129] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.

[130] Dirk A Zetzsche, Douglas W Arner, and Ross P Buckley. Decentralized finance. *Journal of Financial Regulation*, 6(2):172–203, 2020.

[131] Ge Zhang, Wei Liu, Xiaojun Hei, and Wenqing Cheng. Unreeling xunlei kankan: Understanding hybrid cdn-p2p video-on-demand streaming. *IEEE Transactions on Multimedia*, 17(2):229–242, 2014.

[132] Haibin Zhang, Sisi Duan, Chao Liu, Boxin Zhao, Xuanji Meng, Shengli Liu, Yong Yu, Fangguo Zhang, and Liehuang Zhu. Practical asynchronous distributed key generation: Improved efficiency, weaker assumption, and standard model. *Cryptology ePrint Archive*, 2022.

[133] @zksync. Post-mortem: Zksync liveness issue. Twitter, May 2023. `https://twitter.com/zksync/status/1642277357368090626`, Accessed: 2023-05-11.

A Savoiardi Verifiable Information Dispersal

HotShot uses a variant of a VID scheme due to Alhaddad-Duan-Varia-Zhang (ADVZ) that those authors call *AVID-1* [10]. Our variant of AVID-1 is called *Savoiardi* and differs from the original scheme in the following ways:

1. Reduce the asymptotic communication burden of AVID-1 `Disperse` from quadratic to linear in the number of storage nodes. The quadratic communication of AVID-1 `Disperse` is due to the all-to-all messaging among storage nodes in the "echo" and "ready" steps of that scheme. HotShot eliminates these steps, thus achieving linear communication for `Disperse`. (See Section A.7 for discussion.)
2. Augment `Commit`(B) to include a constant-size vector commitment to certain polynomial evaluations as described below. The purpose of this augmentation is to enable the use of quasi-linear algorithms to batch-compute KZG proofs. (See Section A.5 for discussion.)

Next, we informally describe Savoiardi—see Algorithm 9 for pseudocode. Let n be the number of storage nodes, let r be the rate of the erasure code (example: $r = 1/4$), let $m = rn$ be the number of fragments into which the block payload B is split. Without loss of generality we assume that the block payload B consists of a list of scalars in some suitable prime field, and the size of this list is a multiple of m so that B has size km for some k.

A.1 Commit

View the block payload B as k sublists of m scalars each. For $i = 1, \ldots, k$ view the m scalars of sublist i as coefficients for a degree-$(m-1)$ polynomial p_i and let $\hat{p}_i$ denote the KZG [67] commitment to p_i. For each $j = 1, \ldots, n$ let

$$e_j = (p_1(j), \ldots, p_k(j))$$

denote the k-tuple of evaluations of these polynomials at j. Let vc denote an arbitrary constant-size vector commitment scheme. `Commit`(B) is defined as the pair (h, v) where

$$h = \mathrm{hash}(\hat{p}_1, \ldots, \hat{p}_k)$$
$$v = \mathrm{vc}(e_1, \ldots, e_n)$$

(If desired, the bit length of `Commit`(B) could be further reduced by hashing the pair (h, v).)

A.2 Disperse

`Disperse`(B) is a one-round interactive protocol between the block sender and the storage nodes. For $j = 1, \ldots, n$ the sender sends the following data to storage node j:

1. Polynomial commitments $\hat{p_1}, \ldots, \hat{p_k}$ and the vector commitment v. (This data is the same for each storage node.)
2. An evaluation tuple $e_j = (p_1(j), \ldots, p_k(j))$ and a vc opening v_j for e_j to v.
3. A constant-size aggregate KZG witness w_j of the polynomial evaluations relative to the polynomial commitments.

The KZG witnesses $w_1, \ldots, w_n$ are computed as follows:

1. Compute the pseudorandom scalar

$$t = \text{hash-to-field}(\texttt{Commit}(B)) \tag{1}$$

2. Compute the polynomial p as a pseudorandom linear combination

$$p = \sum_{i=1}^{k} t^i p_i. \tag{2}$$

3. Each w_j is a KZG witness for the polynomial evaluation $p(j)$. Batch-compute all KZG witnesses $w_1, \ldots, w_n$ in quasi-linear time via the Feist-Khovratovich algorithm [53].

```
Disperse(B)
    ▷ Sender
1 :  $p_1, \ldots, p_k \leftarrow$ interpret $B$ as polynomials
2 :  $\hat{p}_1, \ldots, \hat{p}_k \leftarrow$ KZG commitments to $p_1, \ldots, p_k$
3 :  for $j = 1, \ldots, n$
4 :    $e_j \leftarrow (p_1(j), \ldots, p_k(j))$ evaluate polynomials
5 :  endfor
6 :  $h \leftarrow \mathrm{hash}(\hat{p}_1, \ldots, \hat{p}_k)$
7 :  $v \leftarrow \mathrm{vc}(e_1, \ldots, e_n)$
8 :  $t \leftarrow$ hash-to-field$(h, v)$
9 :  $p \leftarrow \sum_{i=1}^{k} t^i p_i$   (random lin combo)
10 : $(w_1, \ldots, w_n) \leftarrow$ batch-KZG-prove$(p)$
11 : for $j = 1, \ldots, n$
12 :   $v_j \leftarrow$ open $v$ at $e_j$
13 :   Send to storage node $j$:
14 :     $\hat{p}_1, \ldots, \hat{p}_k, v$ and $e_j, w_j, v_j$
15 : endfor
    ▷ Storage node $j$
16 : Receive $\hat{p}_1, \ldots, \hat{p}_k, v$ and $e_j, w_j, v_j$
17 : Verify vector opening $v_j$ of $v$ at $e_j$
18 : $h \leftarrow \mathrm{hash}(\hat{p}_1, \ldots, \hat{p}_k)$
19 : Commit$(B) \leftarrow (h, v)$
20 : $t \leftarrow$ hash-to-field(Commit$(B)$)
21 : $\hat{p} \leftarrow \sum_{i=1}^{k} t^i \hat{p}_i$
22 : $p(j) \leftarrow \sum_{i=1}^{k} t^i p_i(j)$
23 : KZG-verify$(\hat{p}, j, p(j), w_j)$
24 : Store $\hat{p}_1, \ldots, \hat{p}_k, v$ and $e_j, w_j, v_j$
25 :   indexed by Commit$(B)$
26 : Send to Sender: sign(Commit$(B)$)
    ▷ Sender
27 : Wait for $q$ valid sigs $s_1, \ldots, s_q$
28 :   of message Commit$(B)$ from storage nodes
29 : $s \leftarrow$ aggregate sigs $s_1, \ldots, s_q$
30 : return certificate of retrievability
31 :   $cert$(Commit(B)) $= (s,$ Commit$(B))$
```

```
Commit(B)
1 :  $p_1, \ldots, p_k \leftarrow$ interpret $B$ as polynomials
2 :  $\hat{p}_1, \ldots, \hat{p}_k \leftarrow$ KZG commitments to $p_1, \ldots, p_k$
3 :  $h \leftarrow \mathrm{hash}(\hat{p}_1, \ldots, \hat{p}_k)$
4 :  for $j = 1, \ldots, n$
5 :    $e_j \leftarrow (p_1(j), \ldots, p_k(j))$ evaluate polynomials
6 :  endfor
7 :  $v \leftarrow \mathrm{vc}(e_1, \ldots, e_n)$
8 :  return commitment $(h, v)$
```

```
Retrieve($c$, $cert(c)$)
    ▷ Client
1 :  Check validity of $cert(c)$
2 :  Retrieve $\hat{p}_1, \ldots, \hat{p}_k, v$ from somebody
3 :  $h \leftarrow \mathrm{hash}(\hat{p}_1, \ldots, \hat{p}_k)$
4 :  Verify $c = (h, v)$
5 :  $t \leftarrow$ hash-to-field$(h, v)$
6 :  $\hat{p} \leftarrow \sum_{i=1}^{k} t^i \hat{p}_i$
7 :  Send to all storage nodes: “retrieve $c$, $cert(c)$”
    ▷ Storage node $j$
8 :  Receive “retrieve $c$, $cert(c)$”
9 :  Retrieve $e_j, w_j, v_j$ and send to Client
    ▷ Client
10 : Receive $e_j, w_j, v_j$ from storage node $j$
11 : Verify vector opening $v_j$ of $v$ at $e_j$
12 : $p(j) \leftarrow \sum_{i=1}^{k} t^i p_i(j)$
13 : KZG-verify$(\hat{p}, j, p(j), w_j)$
14 : Store $j, e_j$
15 : Retrieve from $m$ storage nodes $j_1, \ldots, j_m$
16 : for $i = 1, \ldots, k$
17 :   $p_i \leftarrow$ Interpolate from $p_i(j_1), \ldots, p_i(j_m)$
18 : endfor
19 : $B \leftarrow$ Interpret $p_1, \ldots, p_k$ as a block payload
20 : return $B$
```

Figure 9: VID

On receiving this data from the sender, each storage node j checks the integrity of its data. If the integrity check succeeds then the storage node stores its data for later use and replies to the sender with a signature of $\texttt{Commit}(B)$ to indicate its success.

The integrity check proceeds as follows for storage node j:

1. Verify the vc opening v_j for e_j relative to v.
2. Compute t as in (1) and the commitment $\hat{p}$ and evaluation $p(j)$ according to

$$\hat{p} = \sum_{i=1}^{k} t^i \hat{p}_i \tag{3}$$

$$p(j) = \sum_{i=1}^{k} t^i p_i(j) \tag{4}$$

3. Run KZG verification to check that the witness w_j is consistent with $\hat{p}$, j, and $p(j)$.

The sender waits for signatures of $\texttt{Commit}(B)$ from at least q storage nodes. (The choice of q is discussed in Section A.4.) The certificate of retrievability for block payload B consists of an aggregation of these q signatures and $\texttt{Commit}(B)$.

A.3 Retrieve

$\texttt{Retrieve}(c, \mathit{cert}(c))$ is a one-round interactive protocol between the client and storage nodes. The client fetches the polynomial commitments $\hat{p}_1, \ldots, \hat{p}_k$ and vector commitment v from somewhere—possibly from one of the storage nodes—and checks correctness of these commitments by verifying $c = (\text{hash}(\hat{p}_1, \ldots, \hat{p}_k), v)$. Next, the client computes the scalar t as per (1) and polynomial commitment $\hat{p}$ as per (3).

The client extracts the identities of at least q storage nodes from $\mathit{cert}(c)$ and sends a request to each such storage node for its block data for commitment c. Storage node j retrieves its data tuple e_j and witnesses w_j, v_j and sends this data to the client.

On receiving this data from a storage node j, the client checks the integrity of the data:

1. Verify the vector opening v_j for e_j with respect to v.
2. Compute $p(j)$ as per (4) and verify the KZG-witness w_j with respect to $\hat{p}$, j, and $p(j)$.

The client waits for at least m successful retrievals from storage nodes $j_1, \ldots, j_m$. For each $i = 1, \ldots, k$ the client recovers the degree-$(m-1)$ polynomial p_i from the m evaluations $p_i(j_1), \ldots, p_i(j_m)$ via interpolation. The coefficients of $p_1, \ldots, p_k$ are precisely the data in the block payload B.

A.4 Storage quorum size

The number q of storage nodes in a certificate of retrievability is chosen so that the `Disperse` sender and `Retrieve` client are both guaranteed to succeed even in the presence of up to f malicious storage nodes. Thus, we require $m + f \leq q \leq n - f$. There are many choices of f, m, q that meet this constraint. For example, the overhead from erasure encoding is inversely proportional to the erasure code rate $r = m/n$, which is maximized at $m = n - 2f$, implying $q = n - f$. Alternatively, a smaller choice of r enables larger f or smaller q.

A.5 On the need for a vector commitment

We defined $\texttt{Commit}(B)$ in Section A.1 to include a commitment v to the vector $(e_1, \ldots, e_n)$ of polynomial evaluation tuples. Why? For each j the pseudorandom scalar t must depend on the evaluations $p_1(j), \ldots, p_k(j)$ as otherwise a malicious sender could produce a valid KZG witness for incorrect evaluations.

An alternative that avoids the need for a vector commitment is to define a different scalar t_j for each storage node j as $t_j = \text{hash}(\texttt{Commit}(B), p_1(j), \ldots, p_k(j))$ and compute p, $\hat{p}$, and $p(j)$ differently for each storage node using t_j instead of t.

Unfortunately, computation of these polynomials (and their KZG proofs) precludes the use of Feist-Khovratovich and introduces a quadratic dependence on the number n of storage nodes for the sender's run time. As quasi-linear runtime is a priority for HotShot, we prefer the additional communication overhead of the vector opening v_j over $O(n^2)$ run time for the sender.

A.6 Asymptotic complexity

Let $|B|$, $|open|$ denote the size of the block payload B and vector openings v_j, respectively. Total communication over all nodes for the payload B (without overhead) is $O(|B|)$. Overhead per node is $O(k + |open|)$. Recall that kn is $O(|B|)$. Thus, if $|open|$ is constant then Savoiardi achieves optimal asymptotic communication complexity $O(|B|)$.

Asymptotic computational complexity for both `Disperse` and `Retrieve` includes many costs, such as computation and verification of a vector commitment. But these costs are dominated by the discrete Fourier transforms (DFTs) computed in these protocols. Field arithmetic is cheaper than group arithmetic, so we account for these two costs separately.

Disperse. For each $i = 1, \ldots, k$ the polynomial evaluations $p_i(1), \ldots, p_i(n)$ cost $O(n \log n)$ for a total cost of $O(|B| \log n)$ field operations. The batch KZG proof costs $O(n \log n)$ group operations.

Retrieve. The k polynomial interpolations $p_1, \ldots, p_k$ each cost $O(n \log^2 n)$ [25] for a total cost of $O(|B| \log^2 n)$ field operations.

	Field ops	Group ops
`Disperse`	$\|B\| \log n$	$n \log n$
`Retrieve`	$\|B\| \log^2 n$	1

Table 1: Dominant DFT costs of two main procedures in Savoiardi.

A.7 Minimal termination guarantee

Why does AVID-1 `Disperse` of Ref. [10] have "echo" and "ready" steps, and why can these steps be safely eliminated in HotShot? These steps are necessary to achieve strong termination guarantees for AVID-1. Specifically, AVID-1 achieves both *termination* (if the sender is honest then all honest storage nodes complete `Disperse`) and *agreement* (if any honest storage node completes `Disperse` then all honest storage nodes complete `Disperse`).

As observed in Ref. [93], such strong termination guarantees are not needed in protocols such as HotShot. Instead, it suffices that only an honest sender for `Disperse` is guaranteed to complete the protocol and obtain a valid retrievability certificate . This weaker guarantee can be achieved without all-to-all messaging among storage nodes and so we may safely eliminate these steps in HotShot.

A.8 Strong availability guarantee

Some VID protocols offer only a weak availability guarantee: if an honest client initiates `Retrieve`$(C, \mathit{cert}(C))$ then eventually it terminates and obtains some block payload B'. However, there is no guarantee that `Commit`$(B') = C$. This weak availability guarantee allows for a much simpler and faster VID protocol that can be instantiated with any erasure code and any hash function. Examples of state-of-the-art VID protocols of this type include AVID-M [123] and the unnamed protocol of Ref. [9].

The weak availability guarantee implies that a maliciously dispersed payload might not be discovered during `Disperse`. Instead, discovery must wait until `Retrieve`, where the client can re-compute `Commit`(B') to check consistency with commitment C.

In the event where a HotShot adversary corrupts *both* the VID `Disperse` sender *and* the entire HotShot optimistic DA committee so that the DA committee stores no data and the retrieved block payload B' is inconsistent with the commitment C, the data-availability can be lost. Fortunately, the event that `Commit`$(B') \neq C$ can be an evidence for identifying and penalizing a corrupt `Disperse` sender.

- Example: someone could assemble a subset S of storage node shares that recovers B' and create SNARK proof that S is inconsistent with C.

- Example: A quorum of storage nodes could each attest that B' is consistent with its own share but inconsistent with C.

However, the above mitigations are complex and expensive. A complex, expensive, and rarely-used mitigation process is especially vulnerable to mistakes. It is not clear that the performance benefits of weak availability VID is worth this risk.

A.9 Related work

As mentioned previously, Savoiardi is a variant of AVID-1 [10] with weaker termination guarantees. A similar state-of-the-art protocol is *Semi-AVID-PR* due to Nazirkhanova-Neu-Tse [93]. Note that we can alternatively use *Semi-AVID-PR* as our VID protocol, which has features like a transparent setup and the support to fast discrete-log-based curves. The tradeoff is that *Semi-AVID-PR* has higher verification/communication complexity compared to AVID-1 when the number of storage nodes is large.

In Section A.8 we cited Refs. [123, 9] as examples of state-of-the-art VID protocols with a weak availability guarantee.

B Instantiations

In this section, we specify the instantiations of the building blocks and data structures used in the HotShot consensus protocol.

B.1 Random Beacon

Our random beacon is designed to generate unpredictable and unbiased randomness in a decentralized manner. It should periodically provide fresh randomness as input to our leader election protocol, data availability committee sampling, and view synchronization for consensus nodes.

A naive solution is to source entropy from a block hash or a signature of the recent block, and hash them into the desired distribution. But this solution is prone to bias and predictability by an adversary who is able to influence all the entropy sources.

Recall that we assume the adversary cannot control more than $\mathbf{f}$ units of stake, so we can source our beacon from signatures whose total weight is at least $\mathbf{f}+1$, including at least one signature that was honestly generated. However, broadcasting those signatures would lead to excessive communication overhead, so we require a leader to collect all signatures to achieve a linear communication. To prevent the signature collecting leader from biasing the output by arbitrarily choosing which signatures to include, we adopt a "commit-reveal" approach, enforcing that the beacon source has to be fixed and committed before anyone learns the final beacon value. Such magic is achieved by a *delay function* whose output can be obtained only after a certain amount of time has elapsed, even by a well-resourced party capable of massive parallel computation.

Delay Function. A *delay function* (`DF`) is a function that, given a delay parameter $\mathbf{D}$, requires $\mathbf{D}$ sequential steps to compute, and cannot be accelerated with parallel processors. For simplicity, we won't explicitly mention the delay parameter $\mathbf{D}$ and write $y \leftarrow \texttt{DF.Eval}(x)$ for invoking a delay function.

Construction As demonstrated in Algorithm 1, our protocol has the following properties:

- Recall that every block leader already computes a quorum certificate, which includes an aggregate signature with a total weight of at least $2\mathbf{f}+1$. We can reuse this signature as the source entropy for our beacon.
- The new beacon value in block $\mathbf{h}$ is sourced from the quorum certificate of block $\mathbf{h}-\mathbf{k}$, where we assume that block $\mathbf{h}-\mathbf{k}$ is already finalized. Therefore it cannot be predicted $\mathbf{k}$ blocks ahead.
- The time needed to compute the delay function `DF` output, should be significantly larger than the view timeout to ensure the unbiasability. On the other hand, it shall be less than the time to process $\mathbf{k}$ blocks so that the new beacon value will be available once needed.

- We require every party to compute this delay function to prevent the adversary from stalling liveness by DDoSing the designated beacon computing party. Therefore, compared to those solutions with a verifiable delay function, we could simply verify the beacon value by checking the equality, and omit the time consuming proof generation.

Algorithm 1: Decentralized Random Beacon protocol

```
      ▷ When block $\mathbf{h} = \Gamma, 2\Gamma, \ldots$ is decided
 1 :  as everyone
 2 :    Let qc.sig be the quorum certificate aggregate signature of block $\mathbf{h}$
 3 :    Begin computing $candidate\_seed \leftarrow$ DF.Eval(qc.sig) asynchronously
        We assume that the computation will be done before block $\mathbf{h}+\mathbf{k}$
      ▷ On block $\mathbf{h}+\mathbf{k}$
 4 :  as a leader
 5 :    Let $candidate\_seed$ be the output of DF computed in block $\mathbf{h}$
 6 :    Propose a block that contains $candidate\_seed$
      as a node
 7 :    Fetch $candidate\_seed$ from the leader's block proposal
 8 :    Let $candidate\_seed'$ be the output of DF computed in block $\mathbf{h}$
 9 :    Proceed to accept the block if $candidate\_seed = candidate\_seed'$
      ▷ When block $\mathbf{h}+\mathbf{k}$ is decided
10 :  as everyone
11 :    Update the random seed to be the most recent finalized $candidate\_seed$
```

B.2 Random committee sampling

Our Tiramisu data availability solution stipulates that a random committee of fixed size N be selected for the Mascarpone layer in each epoch $e \in \mathbb{N}$. This committee is selected according to Algorithm 2 using the seed produced by the random beacon protocol described in §B.1. All keys being sampled are from the stake table snapshot for epoch e, which was finalized at the end of epoch $e-2$, thus the random committee will have enough number of honest nodes with high probability under the optimistic scenario that the adversary does not adaptively corrupt or bribe the committee nodes. Note again that, the time interval for taking stake table snapshots Γ is at least as large as $\mathbf{k}$, which is the time interval needed to compute a beacon output. Finally, we require the sampled committee to be known by nodes with enough time upfront so that data is available to the members when the leader proposes a new block.

Algorithm 2: SampleRandomCommittee(curBlock,N)

```
      // $\texttt{root}_{\texttt{active}}$ corresponds to the lastest stake table snapshot which is at least $\Gamma$ blocks ahead
 1 :  for $i \in [1..N]$
 2 :    $j \leftarrow$ SampleRandom($seed||i, |keys\_list|$)
 3 :    $key \leftarrow \mathcal{T}$.Sample($\texttt{root}_{\texttt{active}}, j$)
 4 :    $committee.append(key)$
 5 :  endfor
 6 :  $res \leftarrow$ set($committee$)// Remove duplicates
 7 :  return $res$
```

B.3 Quorum Certificates

B.3.1 Background

A *quorum certificate* (QC), as its name suggests, is a certificate of a sufficient quorum of distinct parties voted for a message or statement. It plays an important role in the HotShot's reliable data dissemination and transaction ordering protocols, and its succinctness is essential for achieving desirable communication/computational complexity for the underlying protocols. For example, the following scenarios use a quorum certificate:

Consensus. A consensus leader, after receiving a sufficient number of votes on its commitment proposal, will assemble the votes into a QC, and send this QC to other consensus nodes to precommit/commit the commitment proposal.

Data availability. The Savoiardi protocol stipulates that a block proposer disperse payload shares among storage nodes. Storage nodes respond to the proposer with a signature acknowledging receipt of the share. The proposer assembles the votes into a QC (in this case a DA certificate), and finally it *reliably broadcasts* the DA certificate to the entire network. An honest party, after receiving the DA certificate, can be confident that the corresponding block data will eventually be available to all honest parties.

Checkpoint. Recall that HotShot interacts with L1 via a *sequencer contract*, which checkpoints the state of the HotShot consensus trustlessly. The sequencer smart contract takes as input a QC of a committed block (which includes both the QC for consensus and the DA certificate), verifies the QC against the block commitment, and appends the block commitment if validation succeeds.

A naive approach to construct QC is to let parties sign the message, and the aggregator concatenates the list of signatures as the QC. This approach, however, incurs linear-sized proof and verification time, which is infeasible when the quorum size is large (e.g. 1000). An alternative is to use a threshold signature scheme where any quorum of parties can collaboratively construct a single signature to be verified, and we can set QC to be the threshold signature. This scheme, however, requires an expensive *distributed key generation* (DKG)[132, 62, 42, 73] protocol to enable the parties to secretly share a signing key. Note that the DKG needs to be run periodically as the stake key table is dynamic and can be updated frequently. Moreover, traditional threshold signatures do not allow you to extract the signers' identity from a QC, which is a desirable property if you want to reward participants or slash them for bad behaviors. An *accountable threshold signature scheme*[17, 85, 13, 18, 14] achieves this property but introduces additional complexity.

In this section, we describe two simple QC constructions that suit our needs. The schemes work well as long as the quorum size is at the same order of magnitude as 10K. In Section B.3.2, we specify the APIs and the efficiency/security requirements of the QC scheme. In Section B.3.3, we describe the constructions: the first scheme is used in consensus, while the second is used in the sequencer smart contract.

B.3.2 Definition

Intuitively, we require a quorum certificate (QC) to be short and allow efficient generation and verification. Optionally, we also desire QC to be *accountable*, meaning that we can verifiably slash a party if it signs under two QCs for two conflicting messages (e.g., two conflicting blocks).

Let $\mathcal{T} = \{(\texttt{pk}_i, w_i)\}_{i\in[n]}$ denote a dynamic stake key table where n is the total number of registered keys and $\mathbf{N}_e = \sum_{i=1}^{n} w_i$ is the total number of stakes. Each public key $\texttt{pk}_i$ maps to a corresponding secret key $\texttt{sk}_i$. We formally define the syntax of the algorithms that are related to QC.

- $\texttt{sig} \leftarrow \texttt{QC.Sign}^{\mathcal{T}}(\texttt{sk}, m)$ takes as input a secret key $\texttt{sk}$ and a message m, outputs a signature $\texttt{sig}$. We can understand $\texttt{sig}$ as a weighted vote for message m from a party that holds secret key $\texttt{sk}$.
- $\texttt{qc} \leftarrow \texttt{QC.Assemble}^{\mathcal{T}}(\{\texttt{sig}_i\}, S)$ takes as input a list of signatures, a subset $S \subseteq [n]$, and outputs a QC or $\perp$.
- $b \leftarrow \texttt{QC.Check}^{\mathcal{T}}(\texttt{qc}, m)$ takes as input a QC $\texttt{qc}$, a message m, outputs a bit indicating whether the QC is valid.
- $\{\texttt{pk}_i\} \leftarrow \texttt{QC.Trace}^{\mathcal{T}}(m, \texttt{qc})$ takes as input a message m and a QC $\texttt{qc}$, outputs a list of public keys that participate in the signing of $\texttt{qc}$.

We measure the efficiency of a QC scheme by the running time of the above algorithms, as well as the QC size.

Let T be the quorum size threshold of the QC scheme. We informally state the properties that a quorum certificate scheme should satisfy:

- *Completeness:* Any subset of parties with total stake at least T can assemble a valid QC (for any message) that passes the QC check.

- *Unforgeability:* Any subset of parties with total stake less than T, even if they can query a QC oracle with input subset $S \subseteq [n]$ (with a total stake at least T) and message m and obtain the corresponding valid QC, cannot forge a *new different* valid QC.

Optionally, an accountable QC scheme further satisfies the following:

- *Accountability:* For any set of parties S that participates in the generations of a QC $\mathtt{qc}$ for a message m, the algorithm $\mathtt{QC.Trace}(\mathtt{qc}, m)$ will output the public keys corresponding to the set S.
- *Non-frameability:* For any honest party $i \in [n]$, the set $[n] \setminus \{i\}$ cannot forge a pair $(\mathtt{qc}, m)$ such that $\mathtt{pk}_i$ is in the output of $\mathtt{QC.Trace}(\mathtt{qc}, m)$.

B.3.3 Construction

The QC constructions use an *Aggregate Signature* scheme as a building block. An aggregate signature scheme $\Pi_{\mathsf{agg}} = (\mathtt{Sign}, \mathtt{Verify}, \mathtt{Aggregate})$ is similar to a regular signature scheme but adds an additional API $\Pi_{\mathsf{agg}}.\mathtt{Aggregate}$. The aggregation API takes as input a list of signatures and the corresponding public keys and outputs a single signature $\mathtt{sig}_{\mathsf{agg}}$. An example of instantiation is the BLS aggregate signature [16].

Aggregate signature with signers' identity vector. Next, we describe our first QC construction. Intuitively, the QC is an aggregate signature plus a bit-vector indicating the set of signers under the QC.

```
$\mathtt{sig} \leftarrow \mathtt{QC.Sign}^{\mathcal{T}}(\mathtt{sk}, m)$:
    1. obtain stake table commitment $\mathsf{cm}_{\mathcal{T}}$ from oracle $\mathcal{T}$
    2. return $\mathtt{sig} := \Pi_{\mathsf{agg}}.\mathtt{Sign}(\mathtt{sk}, (\mathsf{cm}_{\mathcal{T}}||m))$
$\mathtt{qc} \leftarrow \mathtt{QC.Assemble}^{\mathcal{T}}(\{\mathtt{sig}_i\}, S)$:
    1. if the total stake weight $\sum_{i \in S} w_i$ is less than the quorum threshold $T$, return $\perp$
    2. extract public key set $\mathtt{pk}_S := \{\mathtt{pk}_i\}_{i \in S}$ from stake table $\mathcal{T}$
    3. compute $\mathtt{sig}_{\mathsf{agg}} \leftarrow \Pi_{\mathsf{agg}}.\mathtt{Aggregate}(\{\mathtt{sig}_i\}, \mathtt{pk}_S)$
    4. set bit-vector $\vec{v}_S \in \{0,1\}^n$ such that $\vec{v}_S[i] = 1$ if and only if $i \in S$
    5. return $\mathtt{qc} := (\mathtt{sig}_{\mathsf{agg}}, \vec{v}_S)$
$b \leftarrow \mathtt{QC.Check}^{\mathcal{T}}(\mathtt{qc}, m)$:
    1. parse $\mathtt{qc} = (\mathtt{sig}_{\mathsf{agg}}, \vec{v}_S)$
    2. extract set $S$ from vector $\vec{v}_S$
    3. if the total stake weight $\sum_{i \in S} w_i$ is less than the quorum threshold $T$, return 0
    4. extract public key set $\mathtt{pk}_S := \{\mathtt{pk}_i\}_{i \in S}$ from stake table $\mathcal{T}$
    5. obtain stake table commitment $\mathsf{cm}_{\mathcal{T}}$ from oracle $\mathcal{T}$
    6. return $b \leftarrow \Pi_{\mathsf{agg}}.\mathtt{Verify}(\mathtt{pk}_S, \mathtt{sig}_{\mathsf{agg}}, (\mathsf{cm}_{\mathcal{T}}||m))$
$\{\mathtt{pk}_i\} \leftarrow \mathtt{QC.Trace}^{\mathcal{T}}(\mathtt{qc}, m)$:
    1. parse $\mathtt{qc} = (\mathtt{sig}_{\mathsf{agg}}, \vec{v}_S)$
    2. extract public key set $\mathtt{pk}_S$ from stake table $\mathcal{T}$ and vector $\vec{v}_S$
    3. obtain stake table commitment $\mathsf{cm}_{\mathcal{T}}$ from oracle $\mathcal{T}$
    4. compute bit $b := \Pi_{\mathsf{agg}}.\mathtt{Verify}(\mathtt{pk}_S, \mathtt{sig}_{\mathsf{agg}}, (\mathsf{cm}_{\mathcal{T}}||m))$
    5. if $b = 1$ return $\mathtt{pk}_S$; else return $\perp$
```

A QC in the above scheme consists of a single signature plus an n-bit vector, which is concretely small when n has a moderate size. E.g., the bit vector is 125B when $n = 1000$, and is 1.25KB when $n = 10000$. The QC verification involves a public key aggregation step and a single regular signature verification, which is practical when running on a regular desktop. However, the public key aggregation step is too expensive to be executed on the sequencer smart contract, thus we propose a modified scheme below that achieves gas-efficient verification.

SNARKed aggregate signature. Intuitively, in our second scheme, a QC consists of an aggregate signature, an aggregated public key, and a SNARK proof showing that the public key was aggregated correctly. Let $\Pi_{\mathsf{SNARK}} := (\cdot, \mathtt{Prove}, \mathtt{Verify})$ be a SNARK scheme, we first describe a SNARK circuit $\mathcal{C}_{\mathsf{agg}}$ for key aggregation.

<u>$\mathcal{C}_{\mathsf{agg}}(\mathtt{pk}_{\mathsf{agg}}, T, \mathsf{cm}_{\mathcal{T}}, \vec{v}_S; \mathcal{T})$:</u>

- *Public input:* the aggregated public key $\mathtt{pk}_{\mathsf{agg}}$, the quorum size threshold T, and the stake table commitment $\mathsf{cm}_{\mathcal{T}}$, and the signers' identity vector $\vec{v}_S$
- *Secret witness:* the stake key table $\mathcal{T}$
- *The SNARK constraints:*
  1. constrain $\mathtt{pk}_S = \{\mathtt{pk}_i\}_{i\in S}$ (and $\{w_i\}_{i\in S}$) to be the subset of public keys (and stake weights) in stake table $\mathcal{T}$ that matches the set S in vector $\vec{v}_S$; note that $\mathcal{T}$ binds to the commitment $\mathsf{cm}_{\mathcal{T}}$
  2. constrain that the weight sum $\sum_{i\in S} w_i$ exceeds the quorum threshold T
  3. constrain that $\mathtt{pk}_{\mathsf{agg}}$ equals the aggregation of the key set $\mathtt{pk}_S$

Remark B.1. *As noted in Remark B.2, $\vec{v}_S$ can be moved to the secret witness if the QC scheme is only used in the sequencer smart contract.*

The second QC scheme $\mathtt{QC}_2$ has the same instantiations for APIs $\mathtt{Sign}, \mathtt{Trace}$ as in the first QC scheme $\mathtt{QC}_1$. We only describe the modifications in the algorithms $\mathtt{Assemble}$ and $\mathtt{Check}$. Note that in the description below, the stake table commitment $\mathsf{cm}_{\mathcal{T}}$ and the quorum threshold T are known to the public.

<u>$\mathtt{qc} \leftarrow \mathtt{QC}_2.\mathtt{Assemble}^{\mathcal{T}}(\{\mathtt{sig}_i\}, S)$:</u>

```
1. compute $\mathtt{qc}' \leftarrow \mathtt{QC}_1.\mathtt{Assemble}^{\mathcal{T}}(\{\mathtt{sig}_i\}, S)$
2. return $\perp$ if $\mathtt{qc}' = \perp$, else parse $\mathtt{qc}' := (\mathtt{sig}_{\mathsf{agg}}, \vec{v}_S)$
3. extract public key set $\mathtt{pk}_S := \{\mathtt{pk}_i\}_{i\in S}$ from table $\mathcal{T}$ and vector $\vec{v}_S$
4. compute aggregated public key $\mathtt{pk}_{\mathsf{agg}}$ from key set $\mathtt{pk}_S$
5. compute SNARK proof $\pi_{\mathsf{SNARK}}$ for circuit $\mathcal{C}_{\mathsf{agg}}(\mathtt{pk}_{\mathsf{agg}}, T, \mathsf{cm}_{\mathcal{T}}, \vec{v}_S; \mathcal{T})$
6. return $\mathtt{qc} := (\mathtt{qc}', \mathtt{pk}_{\mathsf{agg}}, \pi_{\mathsf{SNARK}})$
```

<u>$b \leftarrow \mathtt{QC}_2.\mathtt{Check}^{\mathcal{T}}(\mathtt{qc}, m)$:</u>

```
1. return 0 if $\mathtt{qc} = \perp$
2. parse $\mathtt{qc} = (\mathtt{sig}_{\mathsf{agg}}, \vec{v}_S, \mathtt{pk}_{\mathsf{agg}}, \pi_{\mathsf{SNARK}})$
3. obtain stake table commitment $\mathsf{cm}_{\mathcal{T}}$ from oracle $\mathcal{T}$
4. set SNARK public input $\mathtt{PI} := (\mathtt{pk}_{\mathsf{agg}}, T, \mathsf{cm}_{\mathcal{T}}, \vec{v}_S)$
5. return $b := \Pi_{\mathsf{agg}}.\mathtt{Verify}(\mathtt{pk}_{\mathsf{agg}}, \mathtt{sig}_{\mathsf{agg}}, (\mathsf{cm}_{\mathcal{T}} || m)) \wedge \Pi_{\mathsf{SNARK}}.\mathtt{Verify}(\pi_{\mathsf{SNARK}}, \mathtt{PI})$
```

Remark B.2 (An optimization)**.** *An acute reader may observe that the bit-vector $\vec{v}_S$ is attached in the QC only for achieving accountability, where a challenger can extract a bit vector identifying the subset of misbehaving nodes. Since we use the first QC scheme $\mathtt{QC}_1$ in the consensus, accountability can already be achieved. Thus for the sequencer smart contract application, we can apply the following optimizations: (i) remove the bit-vector $\vec{v}_S$ from the $\mathtt{QC}_2.\mathtt{qc}$; (ii) move $\vec{v}_S$ from public input to secret witness in the key aggregation SNARK circuit.*