Gasper-Siesta: Reducing Ethereum's Commit Latency

Siddhant Sharma

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-68
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-68.html

May 9, 2024

Gasper-Siesta: Reducing Ethereum's Commit Latency

by Siddhant Sharma

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

N. Crooks

Professor Natacha Crooks
Research Advisor

05/09/2024

(Date)

* * * * * * *

RPopa

Professor Raluca Ada Popa
Second Reader

05/08/2024

(Date)

Gasper-Siesta: Reducing Ethereum's Commit Latency

Siddhant Sharma
siddhantsharma@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

ABSTRACT

Ethereum is the world's largest permissionless blockchain that supports a large, decentralized validator set that is resilient to Byzantine faults and dynamic participation. However, Ethereum's consensus protocol, Gasper, is plagued by requiring consecutive honest leaders to commit transactions or blocks. Thus, malicious leaders and even unsuspecting software bugs can easily weaken liveness of the system.

We reduce the commit latency of Gasper by relaxing the requirement of consecutive honest leaders to *any* honest leaders, allowing non-consecutive honest leaders to commit transactions. Our modifications, named Gasper-Siesta, maintains quadratic word complexity for message communication and retains properties that Ethereum strives to maintain. Furthermore, our changes to the Ethereum consensus protocol are fairly limited in code, showing the feasibility of integrating the modifications into the Ethereum consensus protocol in production.

1 INTRODUCTION

Ethereum is the world's largest permissionless blockchains with a large, decentralized validator set that is resilient to dynamic participation, network partitions, and Byzantine actors. Ethereum combines traditional state machine replication (SMR) with an execution environment that allows network participants and clients to send requests that get ordered by consensus and executed by the Ethereum Virtual Machine (EVM). Thus, Ethereum provides users with the abstraction of a centralized, highly-available service. Ethereum guarantees that a set of replicas linearize a sequence of transactions despite bad actors, since it solves BFT SMR. Furthermore, with its emphasis on decentralization and open-participation, Ethereum also scales to a large number of network participants, where participants are clients sending requests as well as replicas participating in BFT SMR.

Ethereum uses a proof-of-stake consensus protocol to solve BFT SMR by combining two different gadgets: a fork-choice rule and a finality gadget. The fork-choice rule, LMD-GHOST, is used to produce new blocks. On the other hand, Casper is the finality gadget used to finalize existing blocks produced via LMD-GHOST. The combination of both gadgets is used in Ethereum today and is known as Gasper[6]. Two key techniques of Gasper are *chaining* and *leader-speaks-once*. Many other consensus protocols also build on the same properties, such as HotStuff[25], DiemBFTv4[24], and Jolteon[12].

Chaining is a technique used to ensure safety of the protocol while reducing expensive cryptography and reusing messages. BFT SMR protocols use voting phases to create *quorum certificates* (QCs) to prove honest replicas signed a given proposal. To improve throughput, protocols often pipeline quorum certificates to use votes on the second or third phase of a block from round v to certify a block in round $v+1$. This helps minimize expensive cryptographic operations and reuse messages. Leader-speaks-once (LSO) is a technique that helps minimize the effect of Byzantine leaders on the progress of the protocol. In this design, leaders propose a block in a given round or view and are rotated out. Other BFT and crash tolerant protocols, such as PBFT[7] and Paxos[17], use stable leaders and only rotate when leaders do not make progress or behave faulty.

Today, Ethereum uses k-finality, where $k = 2$. k-finality refers to a BFT SMR protocol requiring k consecutive honest leaders to commit or finalize a set of transactions or block. Thus, liveness requires 2, or $k + 1$ consecutive honest leaders. However, this leads to significant performance issues in practice. For example, on May 11, 2023, Ethereum faced a mainnet liveness issue with finality [16]. There were 2 incidents where Ethereum failed to finalize blocks for 4 and 9 epochs respectively. Although the incident was caused by a bug in attestation processing in a validator implementation, the chain was unable to finalize blocks due to the fact a high number of slots were missed. Furthermore, the bug lead to significant financial loss, with validators losing an estimated 28 ether on inactivity leak penalties and 50 ether due to missed vote bonuses. The inactivity leak is a small, dynamic penalty that "fines" validators small portions of its stake if the network does not send or sign attestations across several epochs. To make matters worse, the chain recovered due to failover methods that allowed validators to run different validator implementations and revive finality. The consensus protocol was not robust to such validator bugs and failed to restore finality on its own.

We aim to reduce Ethereum's commit latency by relaxing Casper FFG. By allowing Casper FFG to finalize over any $k + 1$ honest leaders, we can strengthen liveness and resilience of Ethereum. We show that we can modify Casper to satisfy a property called any honest leaders (AHL), introduced by BeeGees[1].

Definition 1 (AHL). After GST, if an honest leader in view V proposes a block B, it is guaranteed to commit if possibly non-contiguous views $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ have honest leaders.

With a stronger finality rule that does not impose the consecutive honest leaders invariant on finality, Ethereum would have been more resilient to finality and replica implementation bugs such as the incident discussed above. Ethereum could have finalized quicker and not relied on alternative failover mechanisms to ensure stability. Our work extends the contributions of BeeGees[1] to Ethereum by modifying Gasper. We extend the core concepts that were applied to HotStuff to the Casper finality gadget. We design Gasper-Siesta to guarantee safety for finality gadgets and liveness of the fork-choice rule. We show that Gasper-Siesta will in practice reduce commit latency and is feasible to implement in the Ethereum ecosystem.

2 PRELIMINARIES

2.1 BFT SMR Preliminaries

BFT SMR assumes n network participants and f Byzantine actors such that $n \geq 3f + 1$. Since Ethereum is a proof-of-stake BFT SMR protocol, we can assume that honest participants of the network hold over $\frac{2}{3}$ of the network's total stake. BFT SMR protocols follow two properties:

Definition 2 (BFT SMR). A Byzantine fault tolerant state machine replication protocol satisfies safety and liveness in a linearized log of client requests.

(1) *Safety* All honest replicas commit the same block B at the same slot j.
(2) *Liveness* All client requests eventually receive a response and all honest replicas eventually commit each request.

We also formalize chained leader-speaks-once (CLSO), inspired by [2] and [1]. Ethereum currently implements chained leader-speaks-once, but does not provide the AHL property for CLSO discussed above.

Definition 3 (CLSO). A CLSO BFT SMR protocol advances in views or rounds and satisfies 3 properties.

(1) Every view change changes the leader.
(2) Blocks cannot be committed within a single view.
(3) Honest leaders eventually lead a view, infinitely many times. Essentially, honest leaders will lead an infinite number of views.

2.2 Ethereum Preliminaries

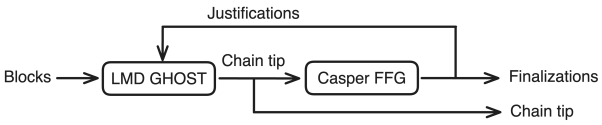

Figure 1: Relationship between LMD-GHOST and Casper. Inspired by Goldfish[8].

Ethereum is a permissionless, decentralized blockchain that allows anyone to participate in BFT SMR with low hardware and network requirements. Since Ethereum aims to be resilient to dynamic participation changes, the blockchain provides two ledgers: dynamic availability and finalized prefix ledger. The dynamic availability ledger is essentially a longest-chain protocol that is always live. The ledger always attempts to produce blocks and extend the chain. However, the dynamic availability ledger does not guarantee safety at all times: it is safe unless there is a network partition. On the other hand, the finalized prefix ledger is a prefix of the dynamic availability ledger that is always safe. However, it is only live under high-enough network participation ($> \frac{2}{3}$ of network stake).

Ethereum chooses to allow leaders to propose blocks for *slots*, and uses *epochs* as checkpoints for finalizing blocks. In traditional BFT SMR literature, slots are essentially leader proposals and epochs are when blocks within the past view are committed. More formally, for an epoch j, there are 32 slots. Slots increase monotonically and last for 12 seconds. We can find the global slot number within an epoch with the formula $i = 32j + k$, where j is the epoch number and k is the index of the slot within the epoch. The 12 seconds in a slot are divided into blocks of 4 seconds. The first 4 seconds are used for a leader to propose and broadcast a block. The next 4 seconds are used for replicas to vote on the leader's proposal. The last 4 seconds are used for BLS signature aggregation [3] to batch signatures together into the block for finalization and slashing checks.

Ethereum uses a committee structure that is inspired by a sharding-focused scalability design. Each slot consists of several committees of validators or replicas. The leader of a slot proposes a block. To decide which block to vote on, other replicas in the committees determine whether they should vote on this "leaf" block in the chain and if it satisfies all correctness rules to extend their last seen block. It is important to note that there is 1 leader, but many committees. Thus, committees without the leader wait to receive the block broadcasted by the leader and decide whether they should vote on the newly produced block. Committees aggregate their votes using BLS signature aggregation once the voting phase is complete.

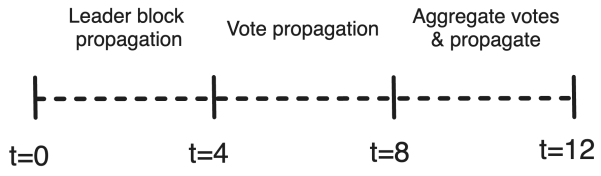

Figure 2: Anatomy of an Ethereum slot.

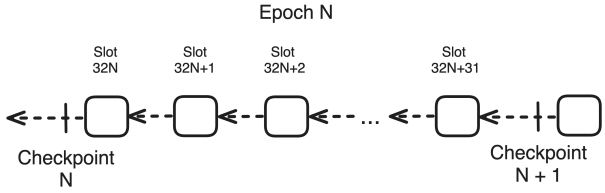

Figure 3: Relationship between epochs, slots, and checkpoints

2.2.1 Network Model. Ethereum follows an partial synchrony networking model in practice, but in theory can perform correctly in asynchrony. Following the slot timing described above and shown in Figure 2, replicas expect to receive a block from the leader within the first 4 seconds, and so forth. However, the 12 second period per slot is not a strict requirement: the chain can move faster if blocks and votes are gossiped quicker than the maximum 4 seconds allocated per phase of the slot. Thus, Ethereum has a weak synchrony network model that uses a known maximum delay in network partitions. It is important to note that Ethereum is designed to tolerate missed slots, or slots that do not have blocks proposed or did not accrue enough votes to be finalized into the chain. To formalize this, Ethereum can assume periods of asynchrony until an unknown but bounded global stabilization time (GST)[11], and synchronously thereafter in practice. This distinction is crucial to our safety and liveness arguments for Gasper-Siesta.

2.2.2 LMD-GHOST and Fork Choice Rule. Last Message Driven Greediest Heaviest SubTree (LMD-GHOST) allows validators to determine what the latest block in the chain is with the most support. LMD-GHOST is a modification of Sompolinsky and Zohar in [22]. Given a view V and attestations A, the weight of a block B is the sum of the stake of validators that attested to B or descendants. Views are equivalent to the chain's state at a given slot number and attestations are votes. LMD-GHOST returns the heaviest leaf block in the subtree, where the weight is computed using the expression above. For our work, LMD-GHOST is black-boxed, as we do not necessarily modify the fork-choice rule.

Algorithm 1 LMD-GHOST Fork Choice Rule

```
1: procedure LMD-GHOST(V)
2:     B ← B_gen                                  ▷ Start from genesis
3:     A ← most recent attestations from validators
4:     while B is not leaf in V do
5:         B ← arg max_{B' child of B} w(V, B', A)
6:     return B
```

2.2.3 Casper and Finality. Casper FFG[5] finalizes blocks produced by validators and voted on by LMD-GHOST. Similar to many other consensus protocols, such as PBFT[7], Casper uses 2 phases of voting to finalize blocks. The first phase of voting is known as *justification* and the second phase is known as *finalization*. The justification phase consists of a replica broadcasting its local best checkpoint, where best is defined as the latest or highest known checkpoint. The replica waits to hear from other participants to determine if other replicas also believe it is the highest known checkpoint. If over $\frac{2}{3}$ of the network stake agree, the checkpoint is now *justified*. The finalization phase consists of replicas broadcasting their latest or highest known justified checkpoint to other replicas. If over $\frac{2}{3}$ of the network stake of participants agree with the replica, the checkpoint is considered *finalized*.

Formally, Casper FFG takes a justified subset of a view V, denoted as $J(V)$, and tries to produce a finalized subset denoted as $F(V)$ such that $F(V) \subset J(V) \subset V$. Finalization occurs between checkpoints A and B where there is over $\frac{2}{3}$ of network stake attesting to the transition from $A \rightarrow^J B$, where $\rightarrow^J$ indicates the finalization threshold is achieved. This is known as a *supermajority* link.

Casper aims to guarantee two core properties: *accountable safety* and *plausible liveness*. Accountable safety ensures two checkpoints on different branches cannot be finalized unless a provable set of validators acted against the protocol. This is similar to most safety properties. Plausible liveness ensures new checkpoints can be finalized if the block production rule creates new blocks.

Casper FFG can be generalized into k-finality that requires k-consecutive epochs to be finalized to commit an epoch. For example, if epochs 100, 101, 102, and 103 were justified in an instantiation of 4-finality, we can finalize epoch 100. This generalization follows the same accountable safety and plausible liveness properties as "vanilla" Casper. However, Ethereum in production, only checks to finalize over the 4 most recent epochs of checkpoints under 2-finality. This is a choice made by the consensus spec and subsequent validator implementations to help recent epochs with delayed attestations still get finalized, but not reach far into the history of the chain. With k-finality, Casper FFG falls into the same class of consensus protocols as HotStuff and requires k consecutive honest leaders to commit. However, we aim to reduce commit latency by generalizing k-finality to any k honest leaders.

As noted in the accountable safety property, Casper FFG slashes for two main rules to slash for incorrect behavior.

- *Double voting*: A validator cannot broadcast votes $s_1 \rightarrow t_1$ (from source block s_1 to target block t_1) and $s_2 \rightarrow t_2$ such that $h(t_1) = h(t_2)$. Therefore, a validator can only vote one block per target epoch.
- *Surround voting*: A validator cannot publish votes $s_1 \rightarrow t_1$ and $s_2 \rightarrow t_2$ such that $h(s_1) < h(s_2) < h(t_2) < h(t_1)$. Effectively, a validator's votes cannot forget existing history by voting around it.

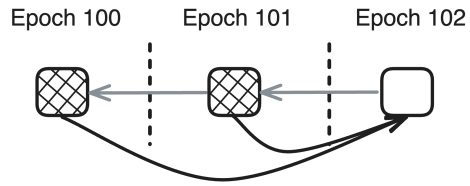

Figure 4: Example of Casper FFG double voting. Epochs 100 and 101 are justified, whereas epoch 102 is not

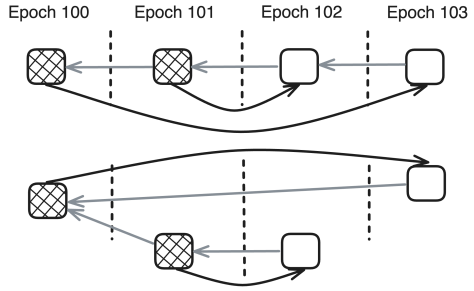

Figure 5: Examples of Casper FFG surround voting. In the first example, we see the vote from epoch 100 to 103 surround the vote from 101 to 102. In the second example, we see the same surrounding, despite when epoch 103 extends epoch 100 directly.

The Ethereum spec implements checks for double voting, but surprisingly does not check for surround voting. Surround voting is complex to detect and requires traversing and maintaining large amounts of historical state (up to 54,000 epochs of votes). Thus, it is left to various researchers and client implementations such as Lighthouse[21] and Prysm[20]. Lighthouse uses an approach called min-max slashing[23], inspired by [18], to efficiently store attestations and search over them for slashing at runtime.

Despite using different terminology, Casper FFG closely borrows concepts from PBFT. The justification and finalization phases of

Casper FFG are analogous to prepare and commit phases of PBFT. Casper FFG voting or attestations showcase some of the differences between the two protocols. Casper FFG follows *chaining*, requiring blocks to extend an existing parent. Thus, votes/attestations must extend a parent block to a descendant block. Furthermore, k-finality implies Casper FFG can only commit a set of operations once k-consecutive honest leaders have completed the justification phase. PBFT does not use a notion of k-finality.

3 RELATED WORK

PBFT [11] is a foundational work that Gasper draws from. Casper follows PBFT-style communication and message structure to commit new blocks. Ethereum's Gasper consensus protocol follows a similar paradigm to Algorand's *BA*⋆ protocol[14]. Algorand was one of the first protocols to deploy proof-of-stake in production to thousands of users. Gasper draws from the tree and DAG based fork-choice rule for choosing the head of the chain similar to Algorand. *BA*⋆ also uses a two-phase protocol to commit, but does not follow the slot-epoch design of Ethereum.

HotStuff [25] extends PBFT to the partially synchronous network model that allows consensus to be driven at the pace of network delays and linear communication complexity. HotStuff and Casper are conceptually similar in their commit mechanisms, but have different communication complexity due to the paradigm of replica to replica attestation gossip in Ethereum, compared to leader to replica communication in HotStuff. Tendermint[4] is another popular BFT SMR consensus protocol used in blockchain settings. Tendermint has a similar commit rule to PBFT. Tendermint and PBFT do not use the notion of k-finality to commit blocks or sets of transactions: once a transaction has over $\frac{2}{3}$ of votes, it is considered final. All of the aforementioned works extend Dwork et. al [11], which is the first solution to partial synchrony that uses a one-chain commit, which can unfortunately lead to deadlocks. Gasper-Siesta draws closely from BeeGees[1], as we apply their techniques for any-honest leaders to finalize blocks to the Casper setting.

Other variations of HotStuff include Jolteon[13] and Fast-HotStuff [15]. Jolteon and Fast-HotStuff take similar approaches to achieving linear message complexity in the optimal case for commits, improving on PBFT. HotStuff-2 [19] is an improvement to HotStuff by Malkhi et. al that also makes similar improvements as Jolteon and Fast-HotStuff. All three protocols take 2 phases of messages to commit in the normal cases. This is an improvement from the 3 phases required in traditional HotStuff. However, HotStuff-2 and Jolteon and Fast-HotStuff differ in their view-change protocols. HotStuff-2 follows similar techniques to Tendermint to achieve a view-change where leaders proceeds if it has a lock from the highest preceding view or waits a delay period. Jolteon uses timeout certificates to view-change more efficiently than HotStuff. It does not aim for linear communication complexity as the view-synchronization is practically bound to quadratic complexity. Fast-HotStuff adds a pre-commit phase to achieve 2 rounds of communication in the optimal case.

4 GASPER-SIESTA

To understand the challenges of relaxing the Casper commit rule, we motivate readers with an example. In a naive implementation where we attempt to relax CLSO from k consecutive honest leaders to k non-contiguous leaders across views, safety violations can occur under periods of network asynchrony. Consider the following example, where without loss of generality, we assume that Ethereum uses a model of one slot per epoch:

Epoch 1: Leader L_1 proposes block B_1. Assume B_1 is justified by at least $\frac{2}{3}$ of replicas and thus finalizes parent block B_0.

Epoch 2: Leader L_2 proposes block B_2 extends B_1. Assume B_2 is justified by at least $\frac{2}{3}$ of replicas, but leader L_3 is partitioned and does not see B_2's justification.

Epoch 3: Leader L_3 proposes block B_3 extending B_1. Assume B_3 is also justified by at least $\frac{2}{3}$ of replicas, but leader L_4 is partitioned and does not observe B_3 or it's justification. Note that B_3's justification is *not* a safety violation, as it is not finalized.

Epoch 4: Leader L_4 proposes block B_4 extending B_2. However, B_4 cannot be justified, as justifying B_4 requires a portion of validators to surround vote over B_3. However, no further blocks can be finalized, as finalizing a block on B_3's fork of the chain requires surround voting around justified block B_2. This is also a safety concern in naive AHL. Consider a replica that has not seen the block in epoch 3 due to network partition. It may believe epoch 2 can be finalized as there are 2 honest leaders, but it does not know of epoch 3 conflicting with epoch 2. However, no validators acted maliciously under the protocol and network assumptions.

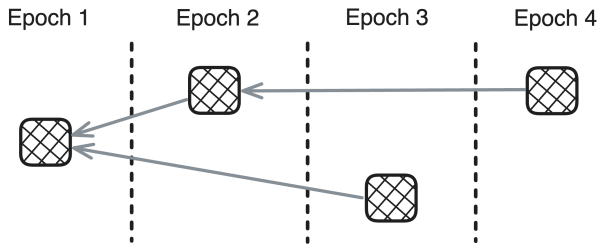

Figure 6: Example of Casper ping-pong effect for finalization with naive any honest leaders.

Solving the safety and liveness concerns of relaxing the consecutive honest property of Gasper requires careful inspection of historical attestations. The core intuition behind Gasper-Siesta uses the same insights as BeeGees[1]. By inspecting previous attestations, we can determine whether epochs not in the non-contiguous commit path could have been committed. If we notice that another epoch may have been committed, we can abort any conflicting commits. This approach avoids the so-called ping-pong effect shown in the example above. Furthermore, we also make use of the QC materialization approach used by [1] to strengthen liveness, and apply that to Gasper's protocol.

4.1 Modifying Casper

4.1.1 Fast-Vew Change Commit. The Casper commit rule can be categorized into two modes for simplicity. The first is the normal steady state of consecutive honest leaders. We dub this the fast

Algorithm 2 Utilities used in pseudocode. For brevity, we blackbox implementations.

procedure *blockToPropose*
    Generate block to propose for a leader in slot j
procedure $createAttestation(\beta)$
    Generate valid attestation for given replica voting on β
procedure $isValidBlock(\beta)$
    Check if β is a valid block for leader in slot j
procedure $isValidAttestation(\alpha)$
    Check if α is valid attestation for current epoch and slot
procedure $isJustified(\beta, A)$
    Check if block β is justified by attestations in A
procedure $isKJustified(JC)$
    Check if last k checkpoints are justified on the same chain
procedure $existsConflictingVotes(FC, JC)$
    Check if there exists conflicting votes against the current justified chain of checkpoints
    If there is at least 1 Gwei of conflicting stake, there may have been another block finalized, so we must abort in such scenarios
procedure $existsValidChain(start, end)$
    Check if there is a chain between given start and end blocks

Algorithm 3 Fast-View Change Commit Rule Pseudocode

$B \leftarrow \{\}$ ▷ Store blocks
$A \leftarrow \{\}$ ▷ Store attestations
$JC \leftarrow \{\}$ ▷ Store justified checkpoints
$FC \leftarrow \{\}$ ▷ Store most recent finalized checkpoint
upon $timer = 0$ **do**
    if $isLeader()$ **then**
        $\beta \leftarrow blockToPropose()$
        send β to all
upon $timer = \Delta$ **do**
    while $timer < 2\Delta$ **do**
        upon receiving β from L_e **do**
            if $isValidBlock(\beta)$ **then**
                $B \leftarrow B \cup \{\beta\}$
                $\alpha \leftarrow createAttestation(\beta)$
                $A \leftarrow A \cup \{\alpha\}$
                send α to all
upon $timer = 2\Delta$ **do**
    while $timer < 3\Delta$ **do**
        upon receiving α from R_i **do**
            if $isValidAttestation(\alpha)$ **then**
                $A \leftarrow A \cup \{\alpha\}$
upon $timer = 3\Delta$ **do**
    $\beta \leftarrow B[-1]$
    if $isJustified(\beta, A)$ **then**
        $JC \leftarrow JC \cup \beta$
    if $j \mod e == 0 \wedge isKJustified(JC)$ **then**
        $FC \leftarrow JC[-1]$

view change case. In this scenario, the leader proposes a block B by broadcasting it to replicas within committees in the slot. The replicas, vote on the block and let it be justified. Since the fast view change takes place under synchrony, replicas send their votes in support of B and extend the previous justified checkpoint block B'. Therefore, we can finalize B' safely. We focus on the slow view change, as it is the focus of our modifications and improvements. Below, we show case the commit rule under the fast-view change and contiguous views with honest leaders.

4.1.2 Slow-View Change Commit. The slow-view change case of Casper does not follow the contiguous honest leaders across epochs paradigm. Thus, we must guarantee safety whilst allowing progress from honest leaders. To achieve this, we must ensure no blocks that can violate safety can be finalized. We carefully design Gasper-Siesta to allow for blocks to be justified if they do not violate Casper slashing rules, but *not* finalize if any other block could have been finalized across non-contiguous views.

We keep track of the previous attestations from replicas voting on proposals to create equivocation proofs that show whether certain blocks could have finalized across non-contiguous views. If we have any equivocation proofs, or conflicting votes with over $\frac{2}{3}$ of network stake on each vote, in an attempted commit, we must abort the commit in the current epoch. We also recognize, similar to BeeGees[1], that in the partial synchrony model, we cannot assume blocks will be justified before GST. Thus, we prior to committing a block, we check for other potential committed blocks during periods of asynchrony. We achieve this using the helper function `existsConflictingVotes(FC, JC)`. This function allows replicas to iterate through the slots of history between the finalized checkpoint and the current justified checkpoint. We iterate through every slot to ensure there were no forks of the chain that may have been justified in a conflicting manner. If no conflicting blocks or forks could have been finalized, we are safe to commit! If we detect *any* conflicting stake used to attest conflicting blocks, we must abort our commit. This is because up to $\frac{1}{3}$ of network stake can be inactive or not attest to a slot transition due to network asynchrony, and an additional $\frac{1}{3}$ can be Byzantine stake. When we sum this stake, we see this already reaches the $\frac{2}{3}$ of network stake, so any additional stake voting for such a transition may have finalized in another chain's view.

In our pseudocode, Implementing this change requires changes to the core data structures and functionality of Ethereum validators, discussed in section 4.3. We iterate over the history and past attestations in lockstep to check whether there is conflicting history. We abstract this logic away in `existsConfictingVotes(FC, JC)`, where we get the local view of the block proposed at a slot. We use the set A to index the attestations that specify the block at the current slot as the target. The set A tracks all of the proposals from leaders and responses from replicas at a given slot. We also attach the local view of the replica that send an attestation to check the attestations are voting along the same view (no equivocation). If a slot is missed, we also check to make sure no other blocks were proposed for that slot that a replica may not have seen due to asynchrony.

We also check to ensure there exists a valid chain between the given start and end block. Validators try to commit between their current finalized checkpoint, previous justified checkpoint, and the

current justified checkpoint. We use `existsValidChain(start, end)` to check there is a valid chain and attestations between these blocks. We also check for `isKJustified(JC)` to ensure there are $k+1$ blocks justified for finalization.

Algorithm 4 Slow-View Change Commit Rule Pseudocode

$B \leftarrow \{\}$ ▷ Store blocks
$A \leftarrow \{\}$ ▷ Store attestations
$JC \leftarrow \{\}$ ▷ Store justified checkpoints
$FC \leftarrow \{\}$ ▷ Store most recent finalized checkpoint
upon $timer = 0$ **do**
  if $isLeader()$ **then**
    $\beta \leftarrow blockToPropose()$
    send β to all
upon $timer = \Delta$ **do**
  while $timer < 2\Delta$ **do**
    upon receiving β from L_e **do**
      if $isValidBlock(\beta)$ **then**
        $B \leftarrow B \cup \{\beta\}$
        $\alpha \leftarrow createAttestation(\beta)$
        $A \leftarrow A \cup \{\alpha\}$
        send α to all
upon $timer = 2\Delta$ **do**
  while $timer < 3\Delta$ **do**
    upon receiving α from R_i **do**
      if $isValidAttestation(\alpha)$ **then**
        $A \leftarrow A \cup \{\alpha\}$
upon $timer = 3\Delta$ **do**
  $\beta \leftarrow B[-1]$
  if $isJustified(\beta, A)$ **then**
    $JC \leftarrow JC \cup \beta$
  if $j \mod e == 0$ **then**
    $justified \leftarrow isKJustified(JC)$
    $abort \leftarrow existsConflictingVotes(FC, JC)$
    if $justified \wedge \neg abort \wedge existsValidChain(FC, JC)$ **then**
      $FC \leftarrow JC[-1]$

4.2 Properties and Proofs

4.2.1 Correctness Proofs. First, we discuss the safety proof of Gasper-Siesta. As discussed above, safety in CLSO BFT SMR protocols is found in Definition 2.2. We start by defining properties of a justifed block.

Definition 4 (Conflicting Blocks). Blocks B and B' are conflicting if B and B' do not extend each other. Formally, $\neg\,(B \leftarrow B' \vee B' \leftarrow B) \vee B \neq B'$.

Lemma 1. *For any two justified blocks B and B' where $B.epoch = B'.epoch$, then $B = B'$.*

Proof. Consider the contradiction where $B! = B'$. Since we believe both blocks are justified, this means both blocks received at least $\frac{2}{3}$ of the total network's stake of votes. However, by quorum intersection, we know that the intersection of the two attestations must have at least $\frac{1}{3}$ of honest stake, otherwise the network is $\frac{1}{3}$-slashable by [6]. Therefore, there is a contradiction.

Next, we can define properties of extension and equivocation proofs.

Definition 5 (Block Extension). A block B' extends B, denoted by $B \leftarrow B'$, if B' is a descendant of B. Thus, there must exist a path of blocks between B and B'.

Definition 6 (Equivocation Proof). We define π_j to be an equivocation proof for epoch j. An equivocation proof contains attestations α and α' for epoch j that contain conflicting history.

Lemma 2. *If there exists a justified block B and a valid attestation α, s.t. $\alpha.block = B$, where $B'.epoch > B.epoch$, and $B \nleftarrow B'$, there is an equivocation proof π_j that B' extends.*

Proof. Consider the contradiction where B' does not extend an equivocation proof π_j. Let B_l be the lowest epoch ancestor of B' where $B_l.epoch > B.epoch$ and $B \nleftarrow B_l$. Then, let B_p be B_l's direct parent. We know that by definition, B_p cannot conflict with B. Thus, we are left with three cases.
Case 1: $B \leftarrow B_p$. Then, $B \leftarrow B_p$ and $B_p \leftarrow B_l$, and it is implied $B \leftarrow B_l$. However, this is a contradiction since $B \nleftarrow B_l$.
Case 2: $B_p \leftarrow B$. Then, $B_p.epoch < B.epoch$ by virtue of blocks only extending blocks from previous epochs. We know B is justified, so by quorum intersection, at least $\frac{1}{3}$ of the network stake voted on B, and one of the participants must be honest. However, this is a contradiction since B_l's parent is B_p, which is in a lower epoch than B.
Case 3: $B.epoch = B_p.epoch$. If $B = B_p$, then $B \leftarrow B_l \leftarrow B'$. However, this is a contradiction, as we know they conflict. In the case where $B! = B_p$, since B is justified, B_l must contain an attestation pointing to B as the source. Similarly, since B_l's parent is B_p, B_l must also contain an attestation with a B_p as the source. However, this forms a contradiction.

Lemma 3. *If block B is committed by an honest replica, then for every justified block B' where $B.epoch < B'.epoch < B^*.epoch$, $B \leftarrow B'$.*

Proof. Consider the contradiction where a justified block B' exists s.t. $B.epoch < B'.epoch < B^*.epoch$ and $B \nleftarrow B'$. We know since B' is justified it must have an attestation where B' is the target. Through lemma 1, we know B' must have an ancestor block that contains an equivocation proof for epoch $B.epoch$. Since $B \leftarrow B^*$ and $B \nleftarrow B'$, we know $B \nleftarrow B^*$. By Lemma 1, we know B^* must also have an ancestor block with an equivocation proof for epoch $B'.epoch$. However, this is a contradiction because an honest replica committed B, as it checked for a conflicting ancestor of B^* and did not find a conflicting vote, yet an equivocation proof exists for B'.

Lemma 4. *If an honest replica commits block B after receiving justified block B_j, then for every valid attestation with target block B' s.t. $B'.epoch > B_j.epoch$, then $B \leftarrow B'$.*

Proof Sketch. We sketch this property via induction. In the base case, we consider the case where we inspect the epoch after our current justified checkpoint, denoted as $e = B.epoch + 1$. We know that every block in an epoch after B must extend B, given by Lemmas 1 and **??**. Inspecting B_j's parent block B_p, we know that since B_j is justified, it must have attestations that specify B_j as the target epoch. Since we know epoch B is consecutive and 1 epoch before B_j,

$B = B_p$. Therefore, we know $B \leftarrow B'$. We can also loosely inducton this for cases where B and B' are not across contiguous epochs. We can use the same argument of justification to show that all parents of B' that are justified must also extend B, by Lemmas 1 and 2. We know that epochs after B must extend B, so we see that $B \leftarrow B'$.

Lemma 5. *If an honest validator commits block B, then every justifed block B' s.t. $B.epoch < B'.epoch$, must contain B in it's chain $chain(B')$. We can also express this as $B \leftarrow B'$.*

Proof Sketch. Using lemmas 1, 2, 3, we know a justified block B' s.t. $B.epoch < B'.epoch$ must extend B. By Lemma 4, any attestation where $B.epoch < B'.epoch$ must also extend B. Thus, any justified block in an epoch after B' must also extend B.

Lemma 6. *For any blocks B and B' committed by an honest validator, either $B \leftarrow B'$ or $B' \leftarrow B'$.*

Proof Sketch. By Lemma 1, we know $B \neq B'$. Therefore, if $B.epoch < B'.epoch$, Lemma 5 tells us $B \leftarrow B'$. Else, also by Lemma 5, we know $B' \leftarrow B$.

4.2.2 Plausible Liveness Proof. Similar to the work in [6], we see that Gasper-Siesta is a modification to the finality rule that Ethereum uses. However, it does not modify the underlying fork-choice rule protocol, LMD-GHOST, so we only concern ourselves with proving that the plausible liveness properties of Gasper-Siesta does not change from Casper. As long as satisfy this property, we can show liveness of the chain.

Definition 7 (Ethereum Formalities). Let function $LJ(B)$ be defined as a function to retrieve the last justified block in the chain $chain(B)$. Let function $EBB(B, j)$ be defined as a function to retrieve the boundary block, or most recent block in epoch j, of chain $chain(B)$.

Definition 8 (Chain Stability). Drawing from [6], we say a chain $c = chain(B)$ at an epoch j is *stable* if the last justified checkpoint at the boundary block of epoch j is block B' from epoch $j-1$. More formally, $LJ(EBB(B, j)) = (B', j-1)$.

To show that Gasper-Siesta satisfies plausible liveness, we prove the following lemma.

Lemma 7. *If at least $\frac{2}{3}$ of the network stake is honest, then it is always possible for a new block B to be finalized with honest validators that follow the protocol after GST.*

Proof Sketch. Consider an arbitrary epoch j and arbitrary slot within the epoch i, in a network after GST. We do not consider the case before GST, as we can make no guarantees about the underlying protocol functioning correctly under asynchrony. Due to plausible liveness, we can plausibly assume the proposer of slot i produces block B after running LMD-GHOST honestly on its previous slot. To show plausible liveness holds true for the next epoch, we must show $chain(B)$ is stable at the beginning of epoch j_{k_1}, where $j < j_{k_1}$. We know that the network must restore some synchrony to make progress under LMD-GHOST, so in an arbitrary j_{k_1} where we partial synchrony is restored, we know at least $\frac{2}{3}$ of the network stake is honest, so they can attest to B or arbitrary descendant B^* s.t. $B \leftarrow B^*$. Thus, in epoch j_{k_1}, it is plausible for a descendant B^* to include the attestations s.t. B is justified, implying $chain(B^*)$ is also stable in epoch j_{k_1}. Therefore, we have a supermajority link between B's parent block in epoch j_{k_0} and B in epoch j, shown as $(B', j_{k_0}) \rightarrow^J (B, j)$

Now, we can relax our assumptions of synchrony and show plausible liveness under the $k = 1$ finalization. Assuming that there is asynchrony after epoch j_{k_1}, we want to show that *any* epoch j_{k_1} s.t. $j < j_{k_1}$ can plausibly finalize if that epoch is after GST and has $\frac{2}{3}$ honest network stake and an honest proposer. Similar to our above analysis, after GST, we know that at least $\frac{2}{3}$ of honest network stake must vote on an honest leader's proposal B^*. The honest validator's proposal receives at least $\frac{2}{3}$ of honest stake's attestations, as incorrect blocks would not receive incorrect votes. Thus, we know that B' can be finalized, as $(B', j_{k_0}) \rightarrow^J (B, j) \rightarrow^J (B^*, j_{k_1})$. Therefore, we satisfy plausible liveness in the simple $k = 1$ finalization case. We can induct and show this applies to any arbitrary k non-contiguous honest leaders.

4.2.3 Message Complexity. Gasper-Siesta has similar message complexity across the fast- and slow-view commit cases. This is due to the slot and networking design of Ethereum. In the fast-view commit, the leader sends a single block to all replicas in committees. This step sends $O(1)$ messages to $O(n)$ replicas. In the voting phase, each replica sends $O(1)$ messages to all other replicas, leading to $O(n^2)$ message complexity. In the slow-view commit, we also expect message complexity of $O(n^2)$ words, as we have leaders broadcast messages to replicas, and then replicas broadcast messages to each other. To be more precise, we expect $O(kn^2)$ words to be sent before we commit across both scenarios, where k is the number of epochs between justification and finalization. In fast-view commits, we expect k to be 2, and in slow-view commit we expect it to be larger, but in the same order of magnitude. We do more analysis on this in section 4.4.

We see both cases have the same message complexity due to the broadcasting of attestations in the gossip layer between committees. Due to the committee and sharding-oriented design of Ethereum, replicas must gossip their votes to all other replicas in the voting phase, leading to $O(n^2)$ words sent.

4.3 Implementing in Ethereum's Consensus Specifications

Etheruem core developers have developed a Python implementation of the Ethereum consensus protocol, known as the Ethereum consensus specifications[9]. The consensus specifications provide a common abstraction of the core data structures and functionality of the blockchain at a high level. Various parts of a validator's core duties, such as signing attestations, producing blocks, slashing, and networking, are defined in the specification. Most importantly, the specification provides a common abstraction to generate test correctness and functionality of new modifications to the protocol. This is useful for our work, as we can test the correctness of our Gasper-Siesta changes for finality in non-contigious epochs with honest leaders. Furthermore, it provides a simple method for consensus client developers to integrate our work into their replica implementations, such as Lighthouse and Prysm.

Implementing the Gasper-Siesta changes into the Ethereum consensus specifications was a fairly small code diff ontop of the existing specification. Our modifications changed some of the core data structures used by the specification, outlined in Appendix A. We also modify some of the core slot, epoch, and attestation processing functionality of the consensus specification. We summarize the changes here and discuss why we deemed the changes necessary. We were motivated to minimize the code delta as much as possible to showcase the feasibility and simplicity of our protocol modifications. Furthermore, maintaining a small delta motivates Ethereum consensus client implementations, such as Lighthouse[21] and Prysm[20] to implement these changes to make Ethereum more resilient.

We update the `JUSTIFICATION_BITS_LENGTH` constant of the protocol to align with the finality window we want to make Ethereum follow. In theory, Gasper-Siesta can finalize between any window of epochs: there is no bound on the window size for finality. However, to ensure validators are not overburdened with managing too much state over large epoch windows. A `JUSTIFICATION_BITS_LENGTH` of 256 indicates Ethereum can finalize across at most 256 epochs, approximately 27 hours. We believe this is a realistic balance between a sufficiently large window to finalize over in the case of network asynchrony or replica failures and not storing too much historical state. We also add a `HISTORICAL_FINALITY_WINDOW` constant to track the same concept. We choose to decouple the two constants is because `JUSTIFICATION_BITS_LENGTH` already exists in the protocol, and reusing across our new changes can be misleading.

We modify `Attestations` and `BeaconState` classes in the consensus specification to track additional state required for our Gasper-Siesta commit rule. `Attestation` represents a vote object and `BeaconState` represents a replica's state. We add an additional attribute to `Attestation` to track the voting replica's view of the chain when creating the vote. We do so to allow replicas to determine whether a block may have been committed between the replica's view of the previous justified checkpoint to the current block produced. In periods of network asynchrony, a replica may not have the full network view of the chain. Thus, this leaves us with two options. The first is to attach each replica's view of the chain to each replica. This leads to relatively-bloated attestation messages, but self-contained messages. The second approach is to use a synchronization step during the vote propagation and processing phase of the slot that uses gossip to acquire the necessary slots that a replica is missing. However, this can add additional latency to the protocol and require extra network infrastructure at the peer-to-peer gossip layer. We modify the `BeaconState` container to store the historical attestations and historical view of the chain. The traditional Ethereum consensus spec does not store the "true" chain history with forks and branches, as it models a correct validator's performance. Thus, we had to modify the consensus specification to store a full chain via a `ChainHistory` abstraction. Each `ChainHistory` object can be viewed as a tuple of the current block root, parent block root, current slot, and parent slot. A replica uses the data stored in the `BeaconState` and `Attestations` to determine whether its most recent justified checkpoint can be finalized across any honest leaders. We add an attribute, `historical_chain`, that tracks this generalized version of the chain history. We can store up to 1 `ChainHistory` object per slot across each of the epochs in our finality window. We store an attribute, `historical_attestations`, that tracks the votes our replica has seen over the same finality window. For brevity, we do not discuss the existing attributes in the consensus specification for both data structures.

Listing 1: Attestation pseudocode in Ethereum

```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    # LMD GHOST vote
    beacon_block_root: Root
    # FFG vote
    source: Checkpoint
    target: Checkpoint

class Attestation(Container):
    aggregation_bits: Bitlist[
        MAX_VALIDATORS_PER_COMMITTEE]
    data: AttestationData
    signature: BLSSignature
```

We also update 3 core functions in the consensus specification that are crucial for processing slots, attestations, and update finality. We modify the `process_slot` function to manage the state of our `BeaconState.historical_chain`, which stores the historical chain of the current replica. As we see a new slot, we phase out stale `ChainHistory` objects stored in the historical chain of the replica. After removing stale chain history, we prepend the new chain history object received by the validator. We do the state removal when processing a new epoch for the `historical_attestations` in a replica's state in the `process_epoch` function. We update `process_attestation` to insert new attestations into `historical_attestations` of a replica. We do careful bookkeeping to ensure attestations are added correctly and old attestations are discarded. Our main contribution is to the commit rule, found in `weigh_justification_and_finalization`. We maintain the core logic in the function that updates justification checkpoints by weighing the attestations and network state, as this is not changed by our Gasper-Siesta commit rule. We only replace the finality logic to relax $k+1$ consecutive honest leaders to $k+1$ non-contiguous honest leaders. To achieve this, we iterate between the previous justified checkpoint of a replica to the current slot. In each step, we check if any other block may have been committed in the epochs between the previous justified checkpoint and the current slot. If there may have been such an epoch or block, we abort the finalization process and update our local state accordingly. If we do not detect such an epoch or block, we check if there are $k+1$ justified checkpoints between the previous checkpoint and the current epoch. If both criteria are satisfied, we know there are $k+1$ non-contiguous justified checkpoint, allowing our previous justified checkpoint to be finalized!

Listing 2: Validator state pseudocode in Ethereum

```
class BeaconState(Container):
    slot: Slot
    justification_bits: Bitvector[
        JUSTIFICATION_BITS_LENGTH]
```

```
    previous_justified_checkpoint:
        Checkpoint
    current_justified_checkpoint:
        Checkpoint
    finalized_checkpoint: Checkpoint
    historical_attestations: List[List[
        Attestation, MAX_ATTESTATIONS],
        SLOTS_PER_EPOCH *
        HISTORICAL_EPOCH_FINALITY_WINDOW]
    historical_chain: List[ChainHistory,
        SLOTS_PER_EPOCH *
        HISTORICAL_EPOCH_FINALITY_WINDOW]
```

Implementing our modifications in the protocol lead to roughly 75 lines of code delta within the Ethereum consensus specification. This includes the core data structure changes, core function changes, and helper functions. We also implemented tests for our changes to ensure parity with existing Ethereum functionality with respect to upgrading the network to use our modifications and correctness of our data structures. Most importantly, we also test for correctness of finalization under various scenarios, including the existing Ethereum tests as well as tests ensuring finalization across non-contiguous epochs. We implemented our work in Python and use existing libraries within the official Ethereum consensus specifications. The official Ethereum spec also allows potential consensus client implementations to easily generate test vectors to verify correct implementations of Gasper-Siesta across clients.

4.4 Expected Commit Latency Improvement

Theorem 9. *With randomized leader election, Gasper-Siesta commits a block in expected $\frac{(k+1)n}{n-f}$ epochs after GST and with omission faults only.*

Proof: Inspired by [1], we can prove this theorem holds. Gasper-Siesta needs $k+1$ non-contiguous epochs to commit as a CLSO protocol. The probability of an honest leader being selected is given by $p = \frac{n-f}{n}$. We know leaders are selected independently, so we can assign $k+1$ random variables to each leader, denoted by $L_i \in \{L_1, L_2, \ldots, L_k\}$. The expected number of rounds until we select the i-th honest leader is given by $\mathbb{E}[L_i] = \frac{1}{p} = \frac{n}{n-f}$. Over $k+1$ rounds, we see $L = \sum_{i=0}^{k} \mathbb{E}[L_i] = \frac{k+1}{p} = \frac{(k+1)n}{n-f}$.

Theorem 10. *With randomized leader election, after GST, the expected number of rounds to commit a block in traditional Gasper is $L = \frac{(1-p^k)}{(1-p)p^k}$, where $p = \frac{n-f}{n}$ and k is the number of consecutive honest leaders required to commit.*

This result is shown in [10].

Using the parameter $k = 3$, similar to what is used in Ethereum in production today, we can compute the difference in expected commit latency for Gasper and Gasper-Siesta. We can assume the probability of an honest node being selected is at least $\frac{2}{3}$; otherwise, the network would not function as a valid proof-of-stake network. Thus, using the formulas from above, the expected commit latency of Gasper is 19 epochs and the expected commit latency of Gasper-Siesta is 4.5 epochs, in partially synchronous network conditions. Thus, Gasper can take up to 2 hours in production under partial synchrony conditions, whereas Gasper-Siesta takes about 28 minutes.

One may argue in practice Ethereum and Gasper do not take 19 epochs to finalize on average. However, our contribution bounds the *worst case* commit latency to be significantly better than Gasper today. Drawing from BeeGees[1], we see that AHL protocols have a worst case commit latency of 18 rounds, whereas CHL protocols with $k = 3$ have a worst case commit time of 76 rounds. Gasper-Siesta should compare to Gasper in the same fashion.

5 CONCLUSION

Our contribution, Gasper-Siesta, combines the novel techniques from [1] to the Ethereum proof-of-stake consensus protocol to reduce the expected commit latency. We use novel techniques to relax the consecutive honest leaders property of the Casper finality gadget to any honest leaders across potentially non-contiguous epochs. We prove safety, liveness, and message complexity properties of our contribution. We also do analysis to show the expected commit latency of our work comapred to traditional Casper and showcase how we improve average and worst-case commit latency. We also implement our work in the official Ethereum consensus specification, showcasing the small protocol changes required to implement our protocol modifications as well as provide a baseline for well-maintained Ethereum consensus clients. We hope our work can make production Ethereum more resilient to various Byzantine scenarios and improve average-case epoch commitment to make the protocol more stable in the long-run.

ACKNOWLEDGMENTS

To Natacha Crooks and Neil Giridharan, for their guidance and advice throughout the course of this work.

REFERENCES

[1] Ittai Abraham, Natacha Crooks, Neil Giridharan, Heidi Howard, and Florian Suri-Payer. 2023. BeeGees: stayin' alive in chained BFT. arXiv:2205.11652 [cs.DC]
[2] Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. 2021. Optimal Good-Case Latency for Rotating Leader Synchronous BFT. In *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France (LIPIcs, Vol. 217)*, Quentin Bramas, Vincent Gramoli, and Alessia Milani (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:19. https://doi.org/10.4230/LIPICS.OPODIS.2021.27
[3] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact Multi-signatures for Smaller Blockchains. In *Advances in Cryptology – ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part II* (Brisbane, QLD, Australia). Springer-Verlag, Berlin, Heidelberg, 435–464. https://doi.org/10.1007/978-3-030-03329-3_15
[4] Ethan Buchman. 2016. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. https://api.semanticscholar.org/CorpusID:59082906
[5] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. arXiv:arXiv:1710.09437
[6] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. 2020. Combining GHOST and Casper. arXiv:2003.03052 [cs.CR]
[7] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.
[8] Francesco D'Amato, Joachim Neu, Ertem Nusret Tas, and David Tse. 2023. Goldfish: No More Attacks on Ethereum?! arXiv:2209.03255 [cs.CR]
[9] Ethereum Core Developers. [n. d.]. Ethereum/consensus-specs: Ethereum Proof-of-stake consensus specifications. https://github.com/ethereum/consensus-specs

[10] Steve Drekic and Michael Z. Spivey. 2021. On the number of trials needed to obtain k consecutive successes. , 109132 pages. https://doi.org/10.1016/j.spl.2021.109132
[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (apr 1988), 288–323. https://doi.org/10.1145/42282.42283
[12] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2021. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. arXiv:arXiv:2106.10362
[13] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2021. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. *CoRR* abs/2106.10362 (2021). arXiv:2106.10362 https://arxiv.org/abs/2106.10362
[14] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757
[15] Mohammad M. Jalalzai, Jianyu Niu, and Chen Feng. 2020. Fast-HotStuff: A Fast and Resilient HotStuff Protocol. *CoRR* abs/2010.11454 (2020). arXiv:2010.11454 https://arxiv.org/abs/2010.11454
[16] Offchain Labs. 2023. Post-mortem report: Ethereum Mainnet Finality (05/11/2023). https://medium.com/offchainlabs/post-mortem-report-ethereum-mainnet-finality-05-11-2023-95e271dfd8b2
[17] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/
[18] Diederik Loerakker. 2019. Protolambda/ETH2-surround: Experimenting with different surround-vote matching optimizations for ETH 2.0. https://github.com/protolambda/eth2-surround
[19] Dahlia Malkhi and Kartik Nayak. 2023. Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT. Cryptology ePrint Archive, Paper 2023/397. https://eprint.iacr.org/2023/397 https://eprint.iacr.org/2023/397.
[20] OffchainLabs. [n. d.]. Prysmaticlabs/prysm: GO implementation of Ethereum Proof of Stake. https://github.com/prysmaticlabs/prysm
[21] SigmaPrime. 2018. SIGP/Lighthouse: Ethereum Consensus Client in rust. https://github.com/sigp/lighthouse
[22] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8975)*, Rainer Böhme and Tatsuaki Okamoto (Eds.). Springer, 507–527. https://doi.org/10.1007/978-3-662-47854-7_32
[23] Michael Sproul. 2020. Implementing a min-max slasher. https://hackmd.io/@sproul/min-max-slasher
[24] The Diem Team. 2021. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf
[25] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus in the Lens of Blockchain. arXiv:1803.05069 [cs.DC]

A ETHEREUM CONSENSUS SPECIFICATION DATA STRUCTURE DELTAS

We specify our modification to the official Ethereum consensus specification's data structures to test the Gasper-Siesta commit rule. Ethereum follows a custom serialization format, known as simple serialize (SSZ). SSZ is a deterministic serialization protocol that efficiently serializes and Merklizes data. Thus, the `Container` keywords indicate a Python class can be serialized using SSZ. All attributes in a class is serializable using SSZ as a primitive or compound type.

To summarize the data structure changes, we update some constants of the protocol. Then, we add an abstraction to store the chain history for slots and the blocks produced. We use this to model the relaxation of the commit rule and verify whether our finalized block is safe. Finally, we modify attestations and validator's state to store this additional metadata.

Listing 3: Update constants to denote how many epochs we look back to finalize.

```
# 256 epochs is approx 27 hours
JUSTIFICATION_BITS_LENGTH = 256
HISTORICAL_FINALITY_WINDOW = 256
```

Listing 4: An abstraction to store an explicit chain of slots and each slot's block roots.

```
class ChainHistory(Container):
    root: Root
    parent_root: Root
    slot: Slot
    parent_slot: Slot
```

Listing 5: Extending attestations to store the replica's justification chain.

```
class Attestation(phase0.Attestation):
    justification_chain: List[
        ChainHistory, SLOTS_PER_EPOCH *
        HISTORICAL_FINALITY_WINDOW]
```

Listing 6: Modification to validator's state to run Gasper-Siesta commit rule.

```
class BeaconState(phase0.BeaconState):
    historical_attestations: List[List[
        Attestation, MAX_ATTESTATIONS],
        SLOTS_PER_EPOCH *
        HISTORICAL_EPOCH_FINALITY_WINDOW]
    historical_chain: List[ChainHistory,
        SLOTS_PER_EPOCH *
        HISTORICAL_EPOCH_FINALITY_WINDOW]
```

Listing 7: Modification to slot processing function.

```
def process_slot(state: BeaconState) ->
    None:
    # Cache state root
    previous_state_root = hash_tree_root(
        state)
    state.state_roots[state.slot %
        SLOTS_PER_HISTORICAL_ROOT] =
        previous_state_root
    # Cache latest block header state
        root
    if state.latest_block_header.
        state_root == Bytes32():
        state.latest_block_header.
            state_root =
            previous_state_root
    # Cache block root
    previous_block_root = hash_tree_root(
        state.latest_block_header)
    state.block_roots[state.slot %
        SLOTS_PER_HISTORICAL_ROOT] =
        previous_block_root
```

```
+    state.historical_chain[1:] = state.historical_chain[:(SLOTS_PER_EPOCH * HISTORICAL_EPOCH_FINALITY_WINDOW) - 1]
+     state.historical_chain[0] = ChainHistory(
+         block_root=previous_block_root,
+         parent_root=previous_state_root,
+         slot=state.slot,
+         parent_slot=state.slot - 1 if state.slot > 0 else 0,
+     )
```

Listing 8: Modification to slot processing function.

```
def process_attestation(state: BeaconState, attestation: Attestation) -> None:
    data = attestation.data
    assert data.target.epoch in (get_previous_epoch(state), get_current_epoch(state))
    assert data.target.epoch == compute_epoch_at_slot(data.slot)
    assert data.slot + MIN_ATTESTATION_INCLUSION_DELAY <= state.slot <= data.slot + SLOTS_PER_EPOCH
    assert data.index < get_committee_count_per_slot(state, data.target.epoch)

    committee = get_beacon_committee(state, data.slot, data.index)
    assert len(attestation.aggregation_bits) == len(committee)

    pending_attestation = PendingAttestation(
        data=data,
        aggregation_bits=attestation.aggregation_bits,
        inclusion_delay=state.slot - data.slot,
        proposer_index=get_beacon_proposer_index(state),
    )

    if data.target.epoch == get_current_epoch(state):
        assert data.source == state.current_justified_checkpoint
        state.current_epoch_attestations.append(pending_attestation)
    else:
        assert data.source == state.previous_justified_checkpoint
        state.previous_epoch_attestations.append(pending_attestation)

+    slots_ago = state.slot - attestation.data.slot
+    assert slots_ago <= HISTORICAL_EPOCH_FINALITY_WINDOW * SLOTS_PER_EPOCH
+    state.historical_attestations[slots_ago].append(attestation)

    # Verify signature
    assert is_valid_indexed_attestation(state, get_indexed_attestation(state, attestation))
```

Listing 9: Modification to slot processing function.

```
+def get_conflicting_attestation_stake(state: BeaconState, slot: Slot, block_root: Root) -> Gwei:
+    """
+    Return the total stake of validators that made conflicting attestations for the given slot and block root.
+    """
+    conflicting_stake = Gwei(0)
+    attestation_index = state.slot - slot
+    assert attestation_index < SLOTS_PER_EPOCH * HISTORICAL_EPOCH_FINALITY_WINDOW, f"Attestation is too old to get conflicting historical conflicting stake: {state.slot}, {slot}"
+    for attestation in state.historical_attestations[attestation_index]:
+        if attestation.data.target.root != block_root or not is_in_justified_checkpoint_chain(state, attestation):
+            conflicting_stake += sum(state.validators[index].effective_balance for index in get_attesting_indices(state, attestation))
+    return conflicting_stake
```