=nil;'s zkSharding for Ethereum

Ilia Shirobokov
=nil; Foundation
i.shirobokov@nil.foundation

Ilya Marozau
=nil; Foundation
ilya.marozau@nil.foundation

Vitaly Kuznetsov
=nil; Foundation
v.kuznetsov@nil.foundation

James A. Henderson
=nil; Foundation
james.henderson@nil.foundation

June 27, 2024
v0.51

1 Introduction

The distributed ledger ecosystem continues to evolve, bringing about an increase in both the complexity of decentralized applications and the requirements for system throughput. The most robust solutions, which have proven their security through years of stable operation, now face the challenge of evolving at a pace that does not compromise the decentralization inherent in the original protocol. The Ethereum Network is the main example of these challenges, suffering from network congestion and high transaction fees. To mitigate these issues, Layer 2 solutions have been introduced. These protocols extend the original protocol to enhance scalability while inheriting the security of the Layer 1 network.

The current strategy to address the scaling issues leans heavily on the concept of modularity through rollups and data availability and consensus layers. While this approach has shown promise, existing solutions introduce significant drawbacks. Rollups are segregated by design, leading to fragmentation in terms of security, liquidity, and data consistency. Furthermore, the need to redeploy applications from Ethereum to a Layer 2 solution exacerbates liquidity fragmentation. Additionally, rollups are not scalable in themselves and require an additional rollup-on-top-of-rollup to achieve scalability.

This document introduces zkSharding, a Layer 2 architecture capable of scaling the Layer 1 network as needed without causing fragmentation. This is made possible through several key components:

- Parallel execution of transactions across different shards by distinct sets of validators, enabling a high throughput of up to 60,000 transactions per second;
- Zero Knowledge state transition proofs that secure the system, allowing validator sets to operate independently on shards and verify other shards in a stateless manner;
- An efficient consensus algorithm that facilitates cross-shard communication, thus reducing transaction processing times.

As illustrated in Figure 1, the state of zkSharding is partitioned into the Consensus Shard and several execution shards. The Consensus Shard 's role is to synchronize and consolidate data from the execution shards. It uses Ethereum both as its Data Availability Layer and as a verifier for state transition proofs, similar to zkRollups operations.

Execution shards function as "workers", executing user transactions. These shards maintain unified liquidity and data through a cross-shard messaging protocol, eliminating any fragmentation amongst them. Each shard is supervised by a committee of validators. There is a periodic rotation of these validators across shards. In addition, updates to a shard's state are verified to the Consensus Shard using VM state transaction proofs

The zkSharding architecture serves as a foundation for =nil; – a zk-powered Layer 2 solution for scaling Ethereum.

Section 2 introduces fundamental definitions and the models within which the protocol operates. Section 3 discusses how individual shards achieve consensus and process transactions within Byzantine Fault Tolerant (BFT) settings. Section 4 explains the collaboration among shards to ensure global

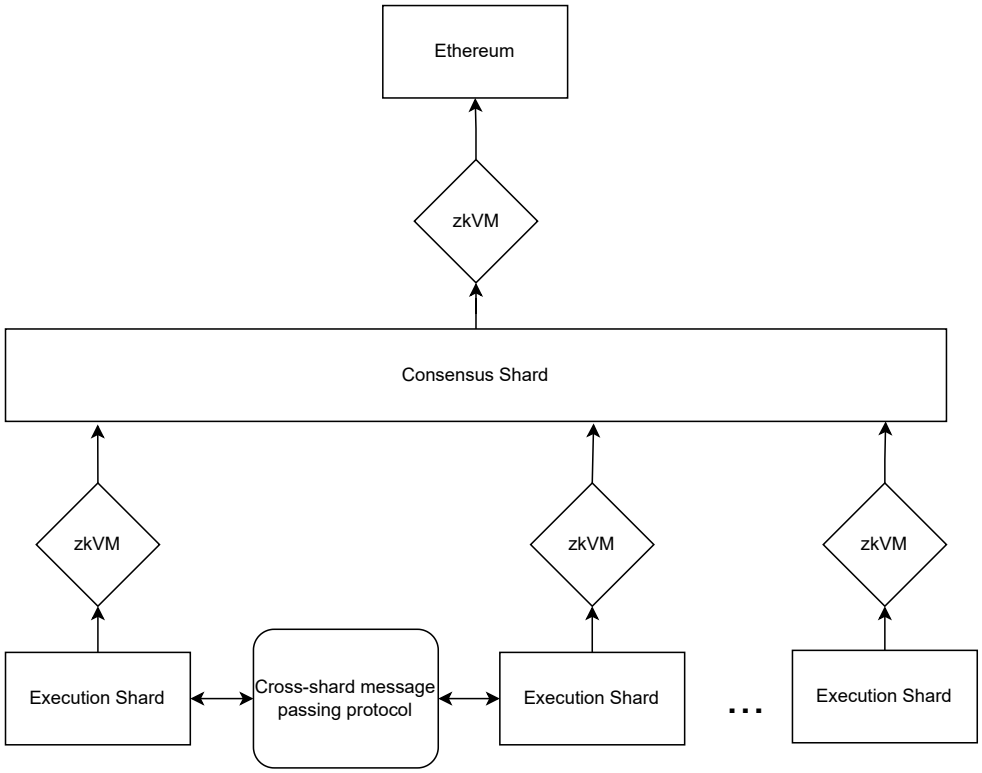

Figure 1: *zkSharding Architecture*

security guarantees. Section 4.5 explores techniques aimed at reducing transaction processing times in sharded configurations. Section 5 describes the sequencing of transactions for zkSharding, including how zkSharding's Data Availability transactions are ordered on Ethereum. Section 6 delves into the Data Availability mechanism utilized by zkSharding and provides an overview of Layer-1 finalization. Section 7 examines the state transition proof mechanism employed for dual purposes: Layer-2 state finalization and the inheritance of security from Layer-1.

2 Preliminaries

Network Model. The system operates under a *partial synchrony* model. In this model, after an unknown Global Stabilization Time (GST), the network achieves synchrony with a known maximum delay Δ. This approach recognizes that synchrony might be temporarily disrupted, potentially due to attacks, but it is expected to eventually stabilize. A distinction is made between the maximal network delay Δ post-GST for worst-case scenarios and an actual network delay δ for average or optimistic case scenarios.

Adversary Model. It is assumed that up to f of the shard's committee members are malicious. Hence, the committee size n is at least $3f + 1$. The adversary can diverge from the specified protocol in any way.

Protocol Properties. A protocol has *optimistic responsiveness* if in an optimistic case it takes $O(\delta)$ to make a decision. In other words, protocol operates at the speed of the network.

A protocol is considered *safe* if at all times, for every pair of correct nodes, the output log of one is a prefix of the other.

A protocol provides *liveness* if, after GST, all non-faulty nodes repeatedly output growing logs.

Consensus Algorithm Background. A *view* in consensus protocols refers to a specific configuration or state of the network. The protocol operates in a sequence of *views*, where each view has a designated leader.

The protocol is decomposed into two subprotocols: *view-synchronization* and *in-view operation*. The view-synchronization subprotocol, also called *pacemaker*, is used by parties to enter a new view and spend a certain amount of time in the view. The *in-view operation* subprotocol is used by parties to commit a

block. This decomposition, which is used by HotStuff [1] and its successors, allows us to analyze *safety* and *liveness* properties of the protocol separately.

Proposer-Builder Separation (PBS). This framework divides the role of single validator into two roles: proposer and builder. Block builders are responsible for constructing the actual contents of a block, including ordering and veriffication transactions. Block proposers are responsible for proposing (validation and propogation) new blocks to be added to the network.

2.1 Multi-Threshold BFT

It is expected to have in partially synchronous systems more emphasis on safety than liveness: The system is designed to be safe always, while liveness is guaranteed only after a certain time. Moreover, both attacks on safety and liveness require committee reformations, but the former also requires a state reformation. The inconvenience is that popular BFT SMR protocols, such as PBFT [2], Tendermint [3], and HotStuff [1], have the same threshold for safety and liveness, a third of the committee size. However, it is possible to decouple the safety and liveness thresholds [4]. This section describes some properties of the Multi-Threshold BFT protocols.

The analysis of the work of [4] provides a framework to design and update protocols to have optimal safety a nd liveness thresholds in both partially synchronous and synchronous settings. For the purposes of this paper, attention is focused on the partially synchronous setting, and a brief summary of the relevant results are provided.

- In the partially synchronous model there exists a BFT SMR protocol with a *safety threshold* of $f_s \geq n/3$ and a *liveness threshold* of f_l, that must satisfy just the following condition:

$$f_l \leq \frac{n - f_s}{2}.$$

- The protocol is based on a Sync HotStuff protocol [5], therefore it has the same 2-vote structure. There are two main differences:
  - Different quorum size, necessary to create a Quorum Certificate, which is equal to $n - f_l$.
  - The protocol is not optimistically responsive (however, for a local consensus, where committee size is not large, it should not become a problem, based on the performance evaluation in [5]).

Remark. By giving up responsiveness, the protocol can become significantly more safe. For example, the liveness threshold can be set to $n/4$, while having a safety threshold of $n/2$, exactly like in the synchronous setting. This improvement is crucial for the sharding protocol, where shards' safety threshold should be strictly greater than the safety threshold of the whole cluster. It is discussed in more detail in Section 4.

3 Intra-shard Replication

An *account* is a minimal data unit from the sharding algorithm perspective. An account is characterized by its address and its associated source code. Notably, there's no distinction between user wallets and other applications.

The state of the cluster is split into parts called *shards*. The shards operate semi-independently, handling only a portion of the accounts of the zkSharding database.

Each shard is maintained by a subset of validators called *committee*. The committee is responsible for the integrity of the shard's state. Since committee members might be malicious, this situation falls within the context of the Byzantine Fault Tolerance (BFT) State Machine Replication (SMR) problem. A BFT SMR protocol for a shard is discussed in this section.

3.1 Local consensus

3.1.1 Pacemaker Module

The pacemaker module is a liveness component of the consensus protocol. It ensures that parties eventually arrive at a view with an honest leader and spend a sufficient amount of time in the view to commit a block. The problem pacemaker solves is called *Byzantine View Synchronization problem* and is thoroughly researched in literature. As mentioned above, the pacemaker module is a bottleneck of the consensus protocol. Hence, an efficient pacemaker is crucial for the overall performance of the consensus protocol.

HotStuff-2 uses RareSynch [8] and Lewis-Pye [9] adaptatioin of a pacemaker protocol, which has theoretically optimal worst case performance, however its average case performance coinsides with the worst case performance, having $O(n^2)$ communication complexity in the average case and $O(n\Delta)$ latency. Previously, a simple yet efficient (in average case) pacemaker protocol was proposed in Naor-Keidar [10]. It has constant expected latency $O(1)$, and worst case latency is $\Omega(n\Delta)$, which was already optimal, [11]. However, it improves the communication complexity from $O(n^2)$ to $O(n)$ in the average case. Worst-case complexity is still $O(n^3)$, but with randomized leader selection, the probability of cascading leader failure is small.

Protocol	**Latency**		**Message Complexity**	
	Avg	**Worst**	**Avg**	**Worst**
Cogsworth[12]	$O(1)$	$O(n\Delta)$	$O(n^2)$	$O(n^3)$
Naor-Keidar[10]	$O(1)$	$O(n\Delta)$	$O(n)$	$O(n^3)$
RareSync/LewisPye[8, 9]	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$

Table 1: *Comparison of Pacemaker Protocols*

As a part of consensus algorithm, zkSharding employs Naor-Keidar pacemaker protocol, named Cogsworth [12]. Algorithm 1 provides a high-level description of the simplified version of the protocol. According to the protocol, a party enters a new view if one of the following conditions is met:

- The block from the previous view was committed.
- A timeout certificate was received.
- A view-change certificate was received.

Algorithm 1: Pacemaker Protocol (Cogsworth)

Step *Wish*:
- **Non-Leader:** If there is no progress, send the leader of view $r+1$ a message `(WISH, r+1)`.
- **Leader:** Collects $f+1$ `(WISH, r+1)` messages and broadcasts an aggregate.

Step *Ready*:
- Upon receiving `WISH` aggregate from any leader, it responds with `(READY, r+1)`.
- Upon timeout, it forwards the `WISH` aggregate to fallback leaders of views $r+2, \ldots, r+f+1$, one by one, to collect READY responses, until there is progress.

Step *Advance*:
- **Leader:** $2f+1$ `(READY, r+1)` messages and broadcasts a `READY` aggregate.
- **Non-Leader:** Upon receiving a `READY` aggregate from any leader, it enters view $r+1$. Upon timeout, it forwards the `WISH` aggregate to fallback leaders of views $r+2, \ldots, r+f+1$, one by one, to collect `READY` responses, until there is progress.

3.1.2 In-View Protocol

An in-view protocol is a protocol that parties execute once they enter the same view and is used to commit a block. It is a safety component of a consensus protocol.

A *quorum certificate* (QC) is a proof that a replicaiton packets was signed by a quorum (2/3 of the committee) of validators. In one view v, there can be at most one QC for a block B_k, denoted as $C_v(B_k)$, and at most one QC for a block QC from the same view, denoted as $C_v(C_v(B_k))$.

Here, a basic (not pipelined) version of the in-view protocol proposed in [7] is described. Pipelining is a technique to amortize the number of rounds required to commit a block. In a steady state, the protocol has two vote phases. A high-level description of the protocol, after honest nodes enter the same view v, is given in Algorithm 2.

Algorithm 2: In-View Protocol (view v, height k)

Step *Enter*:
  if *in view* $v-1$ *a block* B_{k-1} *was committed* **then**
    Leader: Go to **Propose** step.
    Non-leader: Go to **Vote 1** step.
  else
    Leader: Wait for Δ time, then go to **Propose** step.
    Non-leader: Send `lockedValue` to the leader, then go to **Vote 1** step.
Step *Propose*:
  Leader proposes a block B_k and broadcasts `propose`(v, B_k) to all nodes.
  `// ` B_k `is either a locked block with the highest view among all lockedValue messages, or a new block created by the leader.`
Step *Vote 1*:
  Upon receiving `propose`:
  if *isSafe*(B_k) = *true* **then**
    Vote for B_k by threshold signing `vote`(v, h_k), where $h_k = H(B_k)$, and send it to the leader.
Step *Prepare*:
  Leader aggregates $2f+1$ votes into a QC $C_v(B_k)$ and broadcasts `prepare`$(v, C_v(B_k))$.
Step *Vote 2*:
  Upon receiving `prepare`:
  Vote for $C_v(B_k)$ by threshold signing `vote`$(v, C_v(B_k))$, and send it to the leader.
  `lockedView` $:= v$
  `lockedValue` $:= (B_k, C_v(B_k))$
Step *Commit*:
  Leader aggregates $2f+1$ votes into a QC $C_v(C_v(B_k))$ and broadcasts `commit`$(v, C_v(C_v(B_k)))$.

Function `isSafe`(B_k):
  if *isValidBlock*$(B_k) \wedge$ *lockedView* $< v$ **then**
    return `true`
  return `false`

4 Global Sharding Protocol

Consensus Shard. The first shard, known as the Consensus Shard, holds essential data about the protocol's consensus and its current parameters. It also contains information about other shards and the hashes of the most recent replication packets from all shards. Finally, it is responsible for mapping accounts to their corresponding execution shards. In essence, the Consensus Shard serves a dual purpose:

- It sets the protocol's rules and parameters.
- It ensures synchronization across all other shards, including verifying state transition proofs from these shards.

Execution Shards. Execution shards are responsible for processing user transactions. Each shard manages a predefined subset of accounts. The assignment of accounts to shards is determined by a mapping, which is stored on the Consensus Shard:

$$F_S : a \mapsto \texttt{shardId}$$

In this context, a represents the account's address, while F_S directly maps this key to a shard identifier. This mapping can be manually updated by users' requests, subject to constraints maintained by the colocation manager, as detailed in the section 4.5.

Each shard is maintained by a specific group of validators (*committee*). These validators run a "local" consensus algorithm to ensure the shard's state consistency. Details about the shard's local consistency are provided in Section 3.

4.1 Validators Rotation Procedure

At the end of each epoch, the whole validator set generates a new seed for the next epoch using a Verifiable Secret Sharing (VSS) scheme [13, 14]. The seed is used by the validators to generate a new committee for each shard.

$$\texttt{assignment} : \texttt{shardIds} \to 2^{\texttt{validators}}$$

The exact mechanism of assignment update:

1. For each `shardId`, an array of the following values is sorted

$$\texttt{PRF}_{\texttt{seed}}(\texttt{shardId}||\texttt{validatorId})$$

2. The validators corresponding to the first n values are used to form a new committee for the shard.

One could rotate leaders in a round-robin fashion, $\texttt{leader\_id} = \texttt{view} \mod n$, but this would be vulnerable to DoS attacks, since an adversary easily can obtain the leader schedule. A leader election protocol based on Verifiable Random Functions (VRFs), as proposed in [15], is utilized:

$$\begin{aligned} \texttt{seed} &= \texttt{VRF}_{\texttt{prev\_leader}}(\texttt{height}, \texttt{view}) \\ \texttt{leader\_id} &= \texttt{PRF}_{\texttt{seed}}(\texttt{view}) \mod n \end{aligned}$$

where `PRF` is a pseudorandom function. The leader of the previous view provides the seed as a result of evaluating the VRF. Every node can verify that the seed is correct by evaluating the VRF with the public key of the previous leader.

4.2 Cross-Shard Communication

As previously highlighted, all accounts are distributed among shards. At an initial glance, this might seem similar to the data fragmentation issue found in the application-specific rollups approach. However, the key difference is in how cross-shard communication is handled: it's integrated directly into the overall protocol, rather than being managed by separate bridges.

In zkSharding, processing of transactions across shards is mediated by cross-shard transactions (CSTs). To constrain the manipulation and exploitation of CST processing, ZkSharding employs a directed acyclic graph (DAG) architecture called the shardDAG that combines protocol rules, rewards and penalties to incentivise and define a verifiable order in which CSTs must be processed. Within each shard, this order roughly corresponds to processing CSTs in the order in which they were created, completing any existing CSTs before new transactions can be initiated.

The strategy for achieving constraints and to guarantee eventual transaction processing is shown in Fig. 2.

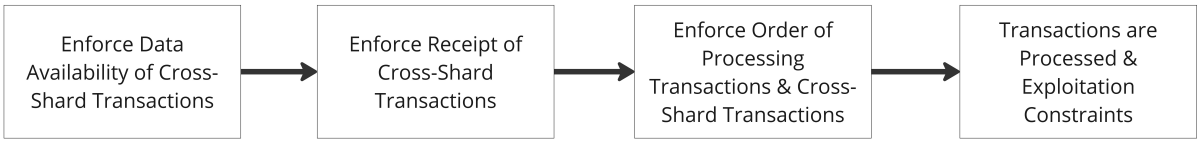

Figure 2: *Achieving constraints on transaction exploitation and guaranteeing eventual transaction processing requires enforceable protocol rules for ordering of transaction and CST processing. An enforceable order requires enforcing shards to receive CST data from other shards. Enforcing receipt of CST data requires enforcing that the CST data is available. Thus, exploitation constraints and eventual transaction processing rests upon data availability of CSTs.*

A key outcome of the shardDAG is the guarantee that once a block containing cross-shard transactions is included in a (valid) consensus block, then those cross-shard transactions are guaranteed to (eventually) be processed and included in their destination shards. Therefore, all transactions are guaranteed to be processed once they have begun processing in an initial shard.

Figure 3 illustrates how the shardDAG induces an order of transaction and cross-shard transaction processing via a simple example shardDAG on the left. In this example shard B block 2 is in the process of being created using hashes to its prior block shard B block 1, as well as shard C block 1 and consensus block 0. Coloured parts of blocks indicate CSTs with destination shard B. In this example is assumed that all cross-shard transactions are pending (i.e. not included in shard B block 1 or earlier). Tracing the

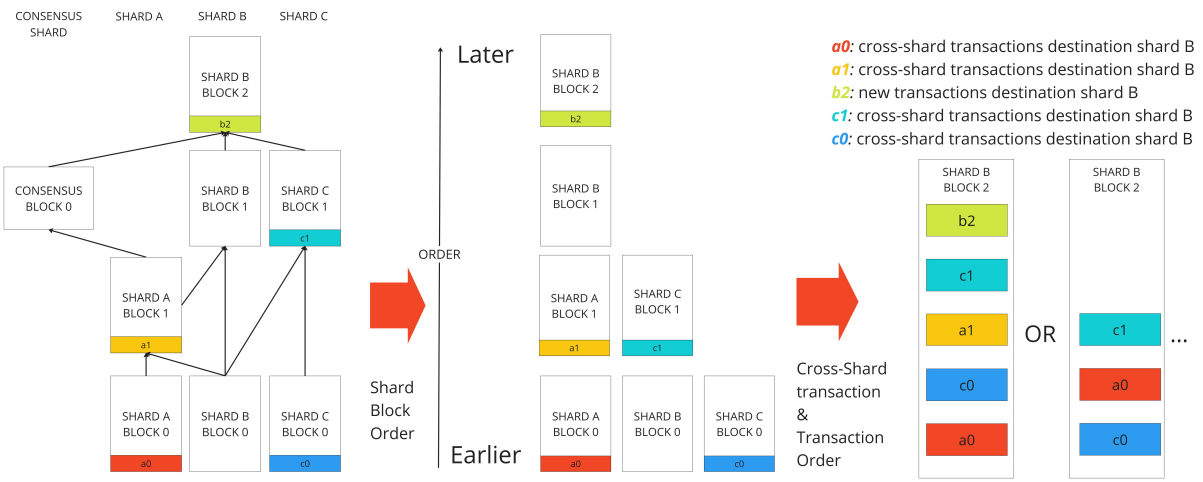

Figure 3: *Left: Shard B block 2's shardDAG subgraph. Centre: A partial order of shard blocks in shard B block 2's subgraph. Right: The order of CST and transaction inclusion in a block must be consistent with the partial ordering of shard blocks. Two examples of valid shard B block 2 are shown containing ordered CSTs.*

DAG edges allows most (but not all) pairs of blocks to be ordered with respect to each other. The centre of the figure shows a partial order of shard blocks in shard B block 2's subgraph. The order in which CSTs are included in shard B block 2 must be consistent with the partial ordering of shard blocks. Two example valid shard B block 2's are shown. Notice that any new transactions ($b2$) can only be included after any CSTs, i.e. transactions that are partially processed are prioritised over new transactions. Shard block proposers can select the order of cross-shard transactions amongst blocks that are equally ordered e.g. the red and blue CSTs. If block capacity restricts inclusion, only the earliest cross-shard transactions are included in shard B block 2, any remaining unprocessed CSTs will be processed in later blocks.

4.2.1 Shard DAG Formation

To form a shardDAG shard blocks include

- A hash to the previous (valid) shard block in the same shard, as in a typical blockchain.
- A set of hashes to other shards blocks in other shards. Up to $N-1$ hashes are allowed, where N is the number of shards. At most one hash per shard is allowed, and no hash can fall in the subgraph of another hash, to eliminate redundant data.
- a hash to a (valid) consensus block, equal to or later than the most recent consensus block already included in prior shard blocks.

The subgraph of a shard block b includes all shard blocks that can be reached starting from b (including b itself) by traversing edges in the shardDAG, including edges to and from consensus blocks. An example shardDAG subgraph is shown in Fig. 4. The black block's header contains a list of hashes (thick solid arrows) to its prior shard block in shard 1, another shard block in shard 2, and a single hash to a consensus block. The dotted arrow indicates a hash that is not allowed because the lower block is in the subgraph of a higher shard block, and the edge is therefore redundant. Thin arrows trace the subgraph of the black block, beyond the blocks explicitly included in its header via hashes.

4.2.2 Block Creation

Validators only propose and sign shard blocks that comply with the shardDAG protocol rules listed below. In constructing the shardDAG we refer to a parameter F that controls the branching of the DAG. One potential constraint on F is to choose F to be at least the maximum number of malicious shards in the network, limiting the ability of malicious shards to collude with each other and not interact with non-malicious shards. However, for performance reasons it may be preferable to use a smaller value of F, and potentially vary F amongst the different conditions described below. For simplicity, in the descriptions below the single parameter F is retained for all conditions.

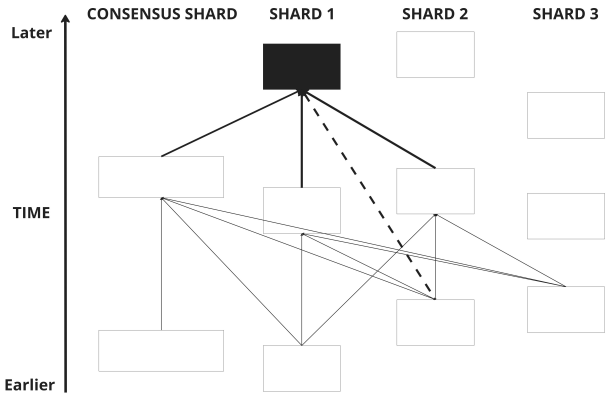

Figure 4: *Illustration of a subgraph in the shardDAG. Hashes included in the black block's header are indicated by thick arrows. The dashed arrow is not a valid hash because it is redundant. The black block's entire subgraph is traced by thin arrows.*

4.2.3 Valid Block Conditions

For a given shard block b to be valid, it must satisfy all the below conditions:

- **Ordering Condition**: For a shard block b to be valid, b's set of processed CSTs C, and processed transactions T must be processed in an order consistent with the order induced by the shardDAG. Specifically, C and T must conform to the following ordering rules. Let U be the set of all (unprocessed) cross-shard transactions in b's subgraph whose destination is b's shard, but which have not been processed in blocks earlier than b. Let V be the set of all (unprocessed) cross-shard transactions in b's subgraph whose destination is b's shard, but which have not been processed in b or earlier, i.e. $V = U \setminus C$. The following must be satisfied for C and T to be valid, ordered sets of CSTs and transactions:
  - C are the earliest CSTs in U. Specifically, for each transaction $c \in C$ there does not exist some other transaction $v \in V$, for which v is ordered earlier than c in the shardDAG.
  - The ordering of processing of elements of C is consistent with the shardDAG ordering of blocks in which each $c \in C$ was first created.
  - If a single shard block creates multiple CSTs with shard b's shard as their destination, then the order of their creation is preserved in the order of processing.
  - C must only contain CSTs that were created within shard blocks in b's subgraph.
  - T are processed after C.

  No rules apply to ordering within sets of transactions and CSTs that are not ordered with respect to each other in the shardDAG, block producers are expected to (but not required to) order based on MEV.
- **Parent Condition**: For a given shard block b with parent shard block a from the same shard, b's subgraph must contain shard blocks created by $> F$ shards that are not present in the subgraph of a. Within b's block header, at most one hash to another shard block is allowed per shard, and no hash can fall in the subgraph of another hash (regardless of shard), to eliminate redundant data.
- **Consensus-Parent Condition**: For a given shard block b with parent consensus block c, there must not be more than X prior shard blocks from b's shard that also have c as a consensus block parent. X is to be chosen once relative timings of consensus and shard blocks becomes clearer.

The purpose of the parent and consensus-parent conditions are to force shards to acknowledge the receipt of shard block headers and outboxes of CSTs. The purpose of the ordering condition is to force shards to obey a clearly defined order for the processing of transactions and CST that each shard has acknowledged receiving. This ordering of transaction and CST processing can be verified and penalties can be applied for breaches of correct ordering.

The following is a central concept in the function of the shardDAG. When shard A creates a shard block that includes the hash of another shard block H, this inclusion acts as an acknowledgement that shard A has received the headers and outboxes of CSTs for H and H's entire subgraph in the shardDAG, up to a limit at which it has been established that this data is no longer required.

An honest validator should not sign a consensus block until it possess the shard block headers and outboxes of CSTs contained within the consensus block. If a shard block includes a consensus block, or a

shard block in its shard block's subgraph, but the creator does not possess the required data, then the shard block risks processing transactions and CSTs in the incorrect order and therefore incurring penalties, and potentially initiating a rollback.

4.2.4 Shard Block Finalisation Condition

The consensus shard incorporates the shardDAG by including sets of shard blocks in each consensus block. Let D be the subset of the shardDAG that has been included in consensus blocks. Before a shard block $b \in D$ can be finalised within the consensus chain it must satisfy the following condition

- **Child Condition**: Within D, there must be more than F shards that have one or more blocks whose subgraph contains b.

Note that is not the only condition for a shard block to be finalised, various other conditions must also be met. The purpose of this child condition is to force shards to distribute their shard block headers and outboxes of CSTs *before* they are included in a consensus block, thus constraining the ability of shards to withhold and delay CST processing.

4.2.5 Local ShardDAG Construction

Each validator constructs its own local shardDAG as follows

- Newly received shard blocks undergo basic validity checks, like checking signatures, checking for more than F shard block hashes etc.
- Shard blocks that meet the above basic validity checks are held in a buffer.
- Shard blocks are moved from the buffer to the shardDAG
  - When a shard block in a buffer has all of its parents (its shard block hashes) already included in the shardDAG, and
  - When the validator has received and verified the shard block's outbox of cross-shard transactions.

The above ensures that the local shardDAG only includes valid shard blocks and the local shardDAG is not missing any shard blocks such that there are no 'holes' in the shardDAG where edges do not have a block on both ends.

4.3 Global Replication Protocol

The safety of the system is limited by the safety of the weakest shard committee. To address this concern, sharded protocols enhance the sizes of shard committees, thereby achieving acceptable safety guarantees [14, 16]. A similar strategy is employed in zkSharding concept, ensuring that full sharding (encompassing storage, communication, and computation) is not compromised. That is, within one epoch, validators need to store, process, and communicate with only a small part, approximately $\frac{(\log N)^R}{N}$ fraction of the whole system.

As previously stated, the set of shard identifiers `shardIds` incorporates a metric, the Hamming distance `dist`. The metric structure of this set is utilized to define a shard's committee: it comprises all validators assigned to the neighborhood of the shard.

$$\texttt{committee(shardId)} = \bigcup_{\substack{\texttt{s} \in \texttt{shardIds} \\ \texttt{dist(s,shardId)} \leq R}} \texttt{assignment(s)},$$

Where $R \geq 0$ serves as a protocol parameter that determines the neighborhood's size, the exact value of R is not specified; however, it is selected to ensure the committee size is sufficiently large to afford acceptable safety guarantees. A standard methodology is employed to estimate the probability of a 1% attack, as detailed in Appendix A.

Remark. By forming committees in this *local* manner, compatibility between the consensus protocol and the message routing protocol (see Section 4.2) is achieved. Validators of neighboring shards, tasked with tracking cross-shard messages, must retrieve necessary messages from these neighboring shards. Thus, including them in the consensus committees of neighboring shards addresses the data availability issue.

The safety analysis, as detailed in A.1, indicates that the probability of a safety attack is non-negligible if the safety threshold for a shard is set equal to the safety threshold of the entire system. To address this issue, other widely recognized protocols [16, 14] either lower the safety threshold of the entire system

or transition to a synchronous network model. A different solution is proposed here: the adoption of the Multi-Threshold BFT [4] consensus protocol, as described in 2.1, which serves to elevate the safety threshold of the shard. The safety threshold, essentially governed by the quorum size, also functions as a safety parameter within the consensus protocol.

Additionally, the zkSharding protocol relies on state transaction proofs (see Section 7) to enable all validators in the system to verify the state of each shard in a stateless manner. However, state transition proofs take time to generate, and standard consensus mechanisms above are used to provide the best security guarantees in the meantime, before the state transition proof is generated.

Therefore, global consensus protocol is a two-level protocol:

- **Local consensus protocol** is a Multi-Threshold BFT consensus protocol, variation of a Sync HotStuf, run by a committee of a shard.
- **Global consensus protocol** is a HotStuff-2 consensus protocol run by the whole validator set.

After running the local consensus protocol, each committee leader proposes a block digest along with a quorum certificate to the Consensus Shard's leader. The Consensus Shard's leader collects all block digests and quorum certificates and proposes a block to the Consensus Shard's committee (whole validator set). The Consensus Shard's committee runs a consensus protocol to finalize shards' latest states.

As mentioned, the probability of a safety attack is adjusted by the protocol's safety parameters and is set to be sufficiently low. However, if the attack still happens, the state of the corrupted accounts is rolled back to the last known good state; for details, see 4.4. Attack detection is facilitated through finalization via state transition proofs, which, once generated, are submitted to the Consensus Shard. If the proof is not valid or the Consensus Shard doesn't receive a state transition proof in a predetermined amount of time (fixed number of successful consensus rounds in the Consensus Shard), the committee size of the shard is increased, by increasing the neighborhood size R. The consensus protocol is then rerun with the new committee size.

4.4 Fixing Errors

The protocol outlines the following stages for state change finalization:

- Local consensus is achieved.
- The latest state of the execution shard is provided to the Consensus Shard.
- The state transition proof of the execution shard is submitted to the Consensus Shard.
- The state transition proof for the zkSharding protocol is submitted to Layer 1. Further details on state transition proofs are discussed in Section 7.

Despite the introduction of mechanisms such as the two-level consensus protocol (Section 4.3), Multi-Threshold BFT (Section 2.1), and shard committees determined by neighborhood size (Section 4.3), there remains a slight chance for malicious nodes to gain control over one of the execution shards. This can occur before state transition proofs are submitted to the Consensus Shard, i.e., within minutes of real time. To address the potential consequences of such attacks, a rollback mechanism has been integrated into zkSharding.

Rollbacks introduce additional complexity to the protocol and incur a large communication overhead. However, the expected overhead is negligible due to the low probability of a safety attack. The protocol follows the following steps before triggering a rollback:

1. As was mentioned, the shard's state finalization cannot be completed, the corresponding committee size is temporarily increased, and the consensus protocol is rerun on an unsafe state change.
2. If an attack is detected:
   - 2.1 Malicious validators who signed an invalid block are slashed.
   - 2.2 Since the safety of the system is attacked, then the whole validator committee must correct the errors via state rollback.

Regarding the rollback:

- The most straightforward and robust possibility is to roll back the system to the last known verified state. This solution, however, does not consider the fact that most of the accounts are not affected by the attack, making redundant the rollback of their states.

- A more sophisticated approach is to roll back only the affected accounts. The problem with this approach is that the error propagation speed is higher than the speed of state transition proofs generation, A big part of the system has to regenerate the latest state transition proofs. This approach is more complex but suffers from the same problem as the first one.

zkSharding uses the first approach with full rollback. The details on this will be provided in the future versions of the document.

4.5 Co-location

Cross-shard communication could extend the processing time of applications located on different shards. For scenarios demanding the swiftest possible transaction processing (i.e. increased consistency), the protocol incorporates a *co-location* technique. [1]

Co-location ensures that two accounts $\{a_1, a_2\}$ are consistently located within the same shard. In other words, $F_S(a_1) = F_S(a_2)$.

The relationship of co-location between addresses A and B is represented as $A \cap B$. The property of transitivity is inherent in the co-location relation, such that if $A \cap B$ and $B \cap C$, it logically follows that $A \cap C$.

Scalability Concerns. Co-location creates an opportunity for concentrating applications on one shard, potentially undermining the sharding concept. From the perspective of the common good, this approach is counterproductive. However, for individual actors, co-locating applications with those that are most used may seem advantageous.

To mitigate this, economic limitations on co-location are proposed. In essence, an address is permitted to be co-located with at most N other addresses, subject to economic constraints that may influence the actual number of feasible co-locations.

Economic restrictions are applied during the operation of account creation, i.e., when a user activates the account with a transaction for the first time. Typically, this transaction includes funding and initial values.

The details on the restrictions will be provided in the future versions of the document.

Binding Map. An explicit map can be implemented via an application on top of the consensus shard, termed a *co-location manager*. Since each validator is required to track the consensus shard, access to this map is always available.

Co-location manager contains the following operations:

```
class CoLocationManager {
    // Data structure to store co-location domains
    domains: map[address -> array[address]];

    // Function to attempt co-locating two addresses
    co-locate: function(pair<address, address>) -> bool {
        // Checks and updates domains to include the co-location if possible
        // Returns true if co-location is successful, false otherwise
    };

    // Function to release the co-location relationship between two addresses
    release: function(pair<address, address>) -> bool {
        // Updates domains to remove the co-location relationship
        // Returns true if the operation is successful, false otherwise
    };

    // Function to check if two addresses are co-located
    is_co_located: function(pair<address, address>) -> bool {
        // Returns true if the addresses are in the same co-location group, false otherwise
    };
```

[1] Obviously, enhancing shard performance leads to improved cross-shard performance, but it cannot enable transaction processing within the timings of one replication packet.

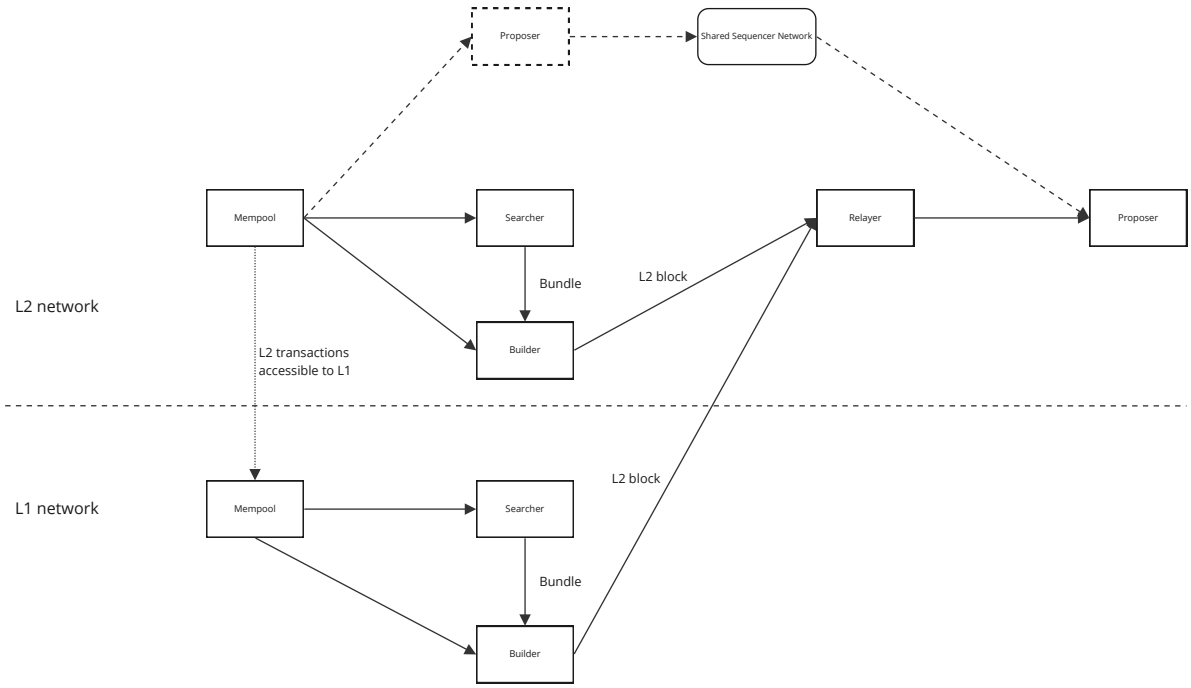

Figure 5: *Sequencing model*

```
        // Function to calculate the fee for address creation or co-location based on co-location
        calculate_fee: function(address) -> number {
            // Calculates and returns the fee based on the co-location depth of the address
        };

        // Function to retrieve the co-location group for a given address
        get_co_location_group: function(address) -> array[address] {
            // Returns the array of addresses that are co-located with the given address
        };
    }
```

Validators of the shard are tasked with tracking the co-location manager and processing accounts related to the shard.

Additionally, co-location enables the emulation of a synchronous mode for contract execution. This means that original Ethereum applications can be redeployed and run on top of `=nil;` without needing to be updated for the asynchronous execution environment of the sharded system. However, this functionality is a feature of the `=nil;` product and not inherent to the zkSharding architecture, so its details are beyond the scope of this document.

5 Sequencing

Utilizing the PBS model, a network of distinct builders and searchers emerges, engaging in competition to construct the most lucrative blocks that outbid others in the Realyer auction. L2 transactions adhere to a structure fully compatible with Ethereum, offering the potential to harness the capabilities of L1 builders and searchers. This approach enhances protocol stability and liveness while maintaining sovereignty. Refer to Figure 5 for an illustration of the high-level schema.

5.1 Shards' sequencing

While each shard manages its own mempool of transactions, there are no restrictions on access for any network members. PBS participants have the autonomy to decide which shard to collaborate with. The associated risks that builders might choose to exclusively handle shards with high-gain applications (e.g., DeFi) are not significant. Several reasons substantiate the market stability of this model:

- When numerous searchers and builders view to propose a single block to a relayer on a shard, the likelihood of placing the highest bid with equivalent efficiency is directly proportional to the number of effective builders. The expected gain can be expressed as $E(G) = 1/n \cdot p$, where n represents the number of effective builders and p is the expected gain. In the secondary shard with a lower gain, denoted as k, where the competition is less intense, the expected gain could surpass that of the Consensus Shard in the case of smaller competition, i.e., $E(G_k) > E(G_n)$, where $n < p/k$.
  On the other side, gas prices rise for underloaded shards, and the gain k will increase over time if the block construction is delayed due to the inactivity of builders;
- A merge occurs when high gas prices result from the inactivity of builders.

Following the principle of PBS and a separated mempool, the possibility arises to utilize independent sequencers tailored to specific needs. The integration and utilization are seamlessly designed, allowing validators to choose such a system over independent builders to enhance specific aspects of the shard. For instance, reducing MEV on a shard with a highly liquid decentralized exchange could potentially significantly decrease slippage, although not mandatory, but likely increasing transaction costs. An example of this integration is illustrated in Figure 5 (dotted line).

5.2 Consensus Shard

The Consensus Shard distinguishes itself from others with its essential requirements of speed and liveness. However, a set of associated risks has emerged along with the reasons for their association:

- No potential MEV issues, due to specifics of the transactions.
- No market competition for gas prices due to exclusivity.
- With few shards quite a lot of unutilized block gas.

This requires an approach where proposers both construct and verify blocks, eliminating the need for a separate role. Instead, a committee will oversee the construction process, and transaction costs will be covered by the protocol fee. The estimation of gas prices for the Consensus Shard is derived from the market price.

6 Data Availability

The Data Availability (DA) layer for L2 solutions outlines the method for storing information essential to recover L2 data in emergency situations.

6.1 Synchronization on L1

To facilitate L2 data availability on the L1 network, the Synchronization Committee is introduced. They ensure data availability on Ethereum for the entire zkSharding solution.

Synchronization Committee participants hold a distinct role in zkSharding. The committee is formed by validators who opt for this additional role. The committee operates in epochs defined by the protocol parameters, with a new committee elected each epoch. An account cannot be an active validator and a member of the Synchronization Committee simultaneously; this separation is enforced by the committee election algorithm. Therefore, although committee members use the same stake, these funds are eligible for slashing for only one role at any given time.

Following a period of time defined by the protocol parameters, the committee generates a state difference for the shard between time T and T + p. The protocol is operated via application on top of the Consensus Shard. A selected node proposes the hash of the state difference and the Synchronization Committee votes on it. Upon achieving $\frac{2}{3} + 1$ votes, the state difference, its hash, and the aggregated signature are composed into an Ethereum data availability transaction. In case of achieving $\frac{1}{3}$ votes "against" the proposed difference, the leader is slashed.

If multiple Ethereum transactions are prepared, the committee may decide to compose them into an L1 block. This enables participation in Relayer auctions and achieves soft finality faster by including the block in the nearest current epoch slot (Figure 6). Alternatively, in case of too few Ethereum DA or state-proof transactions, they can be sent directly to the builders/searchers (as bundle or transaction).

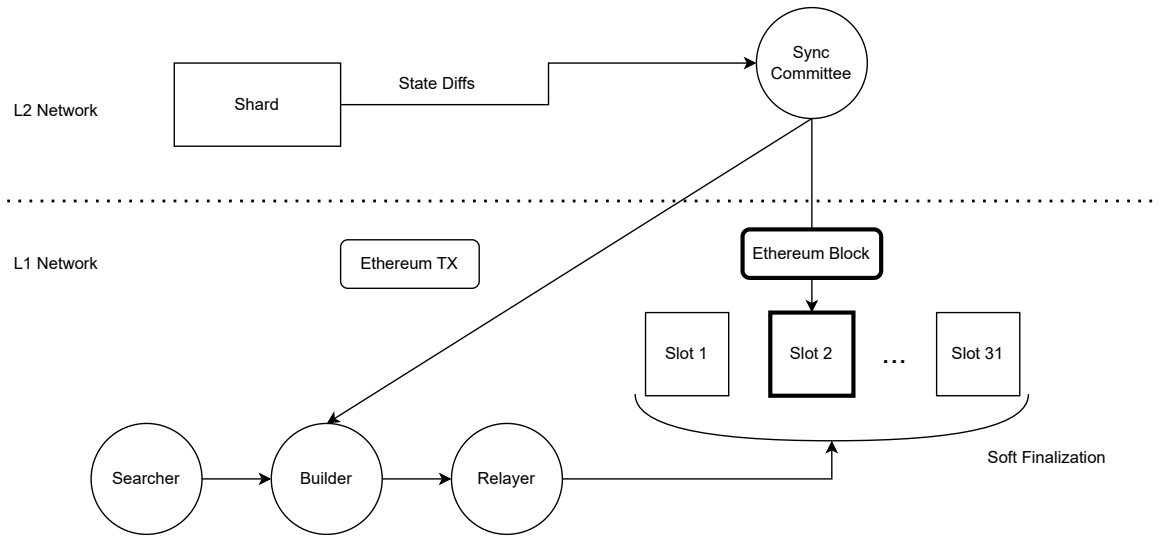

Figure 6: *Synchronization on L1*

6.2 Consensus Shard

The Consensus Shard periodically submits its snapshot to Layer 1 (L1) in the form of the state differentials. These state differentials represent the modified segments of the global state resulting from the application of L2 transactions to the previous state. The purpose of these differentials is to aid in reconstructing a complete L2 state by integrating sequential historical changes in case of rollback events.

Given that the Consensus Shard is responsible for storing and synchronizing the latest state roots committed by execution shards, the transactions it handles are highly specific and persistent in their computations and storage usage.

6.2.1 Finalization

Probabilistic (soft) finalization is attainable due to the high reliability of Ethereum validators. The achievement of probabilistic finalization happens when the Consensus Shard's data availability transaction is verified in the L1 slot.

On the other hand, hard finalization is only accomplished after the verification process. The state change proof, coupled with fully finalized state differences, is necessary for the finalization of Layer 2 defined as follows:

$$Finalization_t = \left\{ \begin{array}{ll} true, & \text{if } V_{zk}(proof_t) \wedge V_{diff}(Diff_t), where: V - verificationfunction \\ false, & \text{otherwise} \end{array} \right\}$$

6.2.2 Data organization and store

Data on L1 is stored in a specifically deployed contract, tasked with accepting L2 state differences, verifying signatures, and persistently storing the data on the chain. Each submitted L2 Consensus Shard state difference is stored in Ethereum *calldata* while metadata on storage as a sequential chain, and the structure appears as mapping:

```
head: hash32;
mapping (hash32 => struct) {
    signature : hash32,
    da_hash : hash32,
    period : uint32 (>= 1),
    prev_da : hash32,
    zk_proof_hash : hash32,
    zk_verification_passed : bool
}
```

State transition proof and state differential hashes will serve as the means for navigating the data stored in *calldata*. The verification status can only be set after the successful validation of the state transition proof. The term "period" refers to the number of blocks that are consolidated. Since this number is not fixed and is defined by the protocol parameters, it must be explicitly stored.

6.2.3 Transactions cost impact

The primary content in the data availability transaction submitted to L1 consists of the state roots submissions from execution shards to the Consensus Shard. As mentioned earlier, the influence of transactions on Consensus Shard storage remains relatively constrained. Although the fundamental idea of sharding revolves around limitless horizontal scaling, the calculations are presently centered on achieving the current target of 60,000 transactions per second (TPS) with 400 execution shards.

Execution shards submit relatively lightweight transactions to the Consensus Shard, primarily focused on submitting the latest state root after each new block. Simultaneously, the Consensus Shard is expected to submit data availability during intervals measured in blocks an interval adjustable by the protocol parameters. GPT For the calculations, a value equal to half of the Ethereum slot time, which is 6 seconds, was chosen. The system produces one block per second.

Every execution shard transaction will result in a change to its account nonce value (32 bytes), balance (32 bytes), and storage (32 bytes state root hash). It's worth noting that the Merkle State Trie path, in this case, is considered to have an average depth of 3. The probability of choosing two identical 3-byte prefixes for 400 keys is approximately 0.475%, which can be safely accepted as the worst case.

$$\texttt{size} = \texttt{shards} \cdot (\texttt{nonce} + \texttt{storage} + \texttt{balance} + \texttt{merkle\_path}) = 400 \cdot 32 \cdot 6 = 76800 \ \texttt{bytes}$$

The total data size, excluding metadata, is 76.8 kilobytes. The metadata and aggregated committee signatures are relatively small and can be disregarded.

Given the high entropy nature of the data, the ratio of zeros to non-zeros in the data packet is approximately $\frac{1}{256}$. This results in:

$$Gas_{zero} = \frac{1}{256} \cdot 76800 \cdot 4(\texttt{gas\_cost}) = 1200$$
$$Gas_{non-zero} = \frac{255}{256} \cdot 76800 \cdot 16(\texttt{gas\_cost}) = 1224000$$
$$Gas_{total} = Gas_{zero} + Gas_{non-zero} = 1225200$$

At the current ETH cost of $2900 and a gas price of 15 gwei, the approximate cost of the Data Availability transaction is $53.2962.

It is important to note that this calculation does not account for the diffs period. The rationale behind this omission is that the commitments size of the diffs remains consistent, given that the changes involve the same accounts.

It is important to note that this calculation does not account for the state diffs period. The rationale behind this omission is that the commitment size of the diffs remains consistent, given that the changes involve the same accounts.

However, the estimated additional cost fee for L2 transactions related to DA can be calculated as $1225200/(6 \cdot 60000) = 3.403$ gas or $0.00014 per user transaction. In comparison, it will be 6 times higher ($0.00084) in the case of submission each main submission. It's crucial to acknowledge that increasing the period for state diffs may compromise stability, particularly in the event of a Consensus Shard revert where the entire state diff period must be reverted across all shards.

6.3 Execution Shards

The paper does not outline specific requirements for data availability in the execution shards. Nevertheless, it is recommended to bolster the shard's security by storing snapshots on a reliable off-chain solution. For instance, each execution shard can autonomously merge state differences over a designated period, compress the data, and then submit it to the Ethereum network (as "calldata" or by EIP-4844) or a dedicated Data Availability layer solution.

6.3.1 Continuous state difference merge (CSDM)

EIP-4844 introduces substantial improvements for all L2 solutions on the Ethereum network. To harness the full capabilities of this new standard and unlock significant cost reduction potential for zkSharding, the CSDM mechanism is introduced to enhance data availability for execution shards.

In alignment with the Ethereum philosophy of verification, this data availability mechanism also incorporates complete persistence of the shard's state on temporary storage.

The process is divided into three parts: initialization, state difference saving, and merge. During the initialization stage, execution shards store the full state in a blob at time T for a duration of n periods. Subsequently, at regular intervals of time p (blocks), the execution shard saves the state difference, D, between $T + pk$ and $T + p(k + 1)$. The merge operation takes place at time $T + n$ when the shard executes the following operation:

$$S_{T+n} = \hat{Y}(...(\hat{Y}(\hat{Y}(S_T, D_{T+p}), D_{T+2*p})..., D_{T+n}),$$

where $\hat{Y}$ is the state merge function defined as $S_{T+k} = \hat{Y}(S_T, D_{T+k})$.

The rationale behind the mechanism can be substantiated by examining the evidence of saving state differences during the time. Suppose there is a throughput of 60,000 TPS, 256 million unique accounts (statistics from Ethereum), 1 second block generation time (1 BPS, blocks per second).

Given the BPS and TPS figures, it is asserted that the minimum number of changed accounts will be at least 60,000, since only externally owned accounts (EOA) can create transactions.

In each new block, the distribution of unique accounts (transactions) follows a normal (Gaussian) distribution as a natural process. If updates occur every 90 days, the total number of changes will be $seconds * TPS = 466560000000$. Utilizing a generally calculated standard deviation, the probability of changing all 256 million accounts is very high (94.51%), and the probability of changing 90% is even higher (99.93%). This renders it entirely reasonable to fully update and resave the state, as the changes during this time effectively create an entirely new state.

7 State Transition Proofs

A *state transition proof* is a cryptographic construct that validates a state transition from S_i to S_{i+1} due to one or more transactions, without the need to rerun these transactions.

The formal representation of a state transition proof can be defined as a function $\mathcal{F}$:

$$\mathcal{F}(S_i, T, S_{i+1}, PI) \rightarrow (\pi) \tag{1}$$

where:

- S_i is the state before the transactions.
- T represents the transaction or batch of transactions.
- S_{i+1} is the state resulting from applying the transactions.
- $PI = [C_T, C_{S_i}, C_{S_{i+1}}]$ is a public input with a succinct representation of S_i, S_{i+1}, and T.
- π is the zero-knowledge proof verifying the correctness of the transition from S_i to S_{i+1} without knowing T.

This proof π is subsequently verified by a verifier function $\mathcal{V}$:

$$\mathcal{V}(PI, \pi) \rightarrow \{\text{true, if the transition is valid; false, otherwise}\}$$

These definitions can be applied both to the zkSharding whole system and to particular shards. In the first case, S_i represents the "world" state of zkSharding and includes whole sharded database. In the second case, S_i represents the state of the particular shard. However, a more precise definition of $\mathcal{F}$ for zkSharding's world state can be provided. For k shards, the state transition proof's formal representation is as follows:

$$\mathcal{F}(S_i^0, \ldots, S_i^{k-1}, S_{i+1}^0, \ldots, S_{i+1}^{k-1}, T^0, \ldots, T^{k-1}, PI^0, \ldots, PI^{k-1}) \rightarrow (\pi) \tag{2}$$

There are two issues that prevent the implementation of state transition proof generation by a single validator node:

- State transition proofs are computationally intensive tasks that take time. The larger the state change, the more time it takes.

- To provide a proof for the whole zkSharding state, the prover must obtain the state of the entire sharded system.

For these reasons, the function $\mathcal{F}$ is implemented as a multi-party protocol. Participants of the protocol are called *proof producers*.

7.1 Proof Generation Protocol

Define three types of proofs for the protocol:

- π_S is a state transition proof defined by Equation 1.
- π_A is an aggregation proof that aggregates two state transition or aggregation proofs:

$$\mathcal{F}_A(\pi_1, \pi_2) \to \pi_A$$

- π_O is an output proof defined by the function:

$$\mathcal{F}_O(\pi_{A,1}, \pi_{A,2}) \to \pi_O$$

The output proof is required for cases when proof verification costs on the execution layer depend on the proof system parameters. For example, KZG-based proofs verification is generally cheaper than FRI-based ones on the Ethereum Virtual Machine. Note that $\mathcal{F}_O$ is required only for cost optimizations and may be represented as $\mathcal{F}_A$ for the simplicity of implementation.

Now, equation 1 can be represented as:

$$\mathcal{F} = \mathcal{F}_O\left(\mathcal{F}_A^{\log k-1}\left(\mathcal{F}_A(\mathcal{F}_S(\ldots)), \mathcal{F}_A(\mathcal{F}_S(\ldots))\right), \mathcal{F}_A^{\log k-1}\left(\mathcal{F}_A(\mathcal{F}_S(\ldots)), \mathcal{F}_A(\mathcal{F}_S(\ldots))\right)\right),$$

The *Proof Distribution Protocol* (or *PDP*) is responsible for assigning proof producers to particular *slots*. A *slot* is a task for generating one specific proof with defined input. Separating the proof aggregation algorithm from the proof producer assignment logic allows for independent updates of both algorithms.

Algorithm 3: Proof Distribution Protocol: Slots Assignment

1. An event occurs: a new block is sent to the Consensus Shard.
2. A part of the block reward is locked as a reward for proof producers. The part is calculated as the sum of rewards for each slot related to the block, with slot rewards defined by adjustable protocol parameters.
3. PDP defines the list of open slots, each defined as follows:

```
slot_id: uint
shard_id: uint
block_seq_no: uint
batch_seq_no: uint
proof_type: enum
max_fee: float
```

4. Proof producers provide proofs for the slots, requesting any fee lower or equal to `max_fee`. While multiple proof producers can provide the proof for the same slot, only the proof with the lowest fee is chosen.
5. The rewards are paid to proof producers according to the requested fee.

Remark. Step 4 of Algorithm 4 can start as soon as at least two proofs are generated at Step 3. For clarity, this is omitted in the algorithm description.

Algorithm 4: Proof Distribution Protocol: Intra-shard State Transition

1. Validators confirm the block B that contains a set of transactions T.
2. Based on the protocol parameters, T is split into k batches $[T_0, \ldots, T_{k-1}]$.
3. PDP opens k slots for π_S proofs for the given batches.
4. PDP consequently opens $k-1$ slots for π_A to aggregate the k π_S proofs into two proofs as a binary tree.
5. PDP opens 1 slot for π_O proof.

7.1.1 Global State Transition Proof

Note that Equation 2 can be represented as:

$$\mathcal{F}(S_i^0, S_{i+1}^0, \overline{T}, \overline{PI}),$$

where 0 is the sequence number of the Consensus Shard, and $\overline{T}, \overline{PI}$ contain data about transactions that call the verification function $\mathcal{V}(PI^i, \pi^i)$ for $i \in [1, k]$.

In other words, the global state transition proof can be obtained from the Consensus Shard's state transition proof, with state differences that include verification of other shards' state transition proofs. Thus, the algorithm is as follows:

1. Proof producers generate π_O^i for each execution shard.
2. Validators send π_O^i proofs to the Consensus Shard.
3. The Consensus Shard verifies the proofs.
4. Proof producers generate π_O for the Consensus Shard.

Later, the global state transition proof is transferred to Layer 1 by the Synchronization Committee as described in Section 6.1.

References

[1] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus in the lens of blockchain," 2019.

[2] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, (New Orleans, LA), USENIX Association, Feb. 1999.

[3] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," 2016.

[4] A. Momose and L. Ren, "Multi-threshold byzantine fault tolerance," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, (New York, NY, USA), p. 1686–1699, Association for Computing Machinery, 2021.

[5] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 106–118, 2020.

[6] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, 2008.

[7] D. Malkhi and K. Nayak, "Extended abstract: Hotstuff-2: Optimal two-phase responsive bft." Cryptology ePrint Archive, Paper 2023/397, 2023. https://eprint.iacr.org/2023/397.

[8] P. Civit, M. A. Dzulfikar, S. Gilbert, V. Gramoli, R. Guerraoui, J. Komatovic, and M. Vidigueira, "Byzantine consensus is $\theta(n^2)$: The dolev-reischuk bound is tight even in partial synchrony! [extended version]," 2022.

[9] A. Lewis-Pye, "Quadratic worst-case message complexity for state machine replication in the partial synchrony model," 2022.

[10] O. Naor and I. Keidar, "Expected linear round synchronization: The missing link for linear byzantine smr," 2020.

[11] M. K. Aguilera and S. Toueg, "A simple bivalency proof that t-resilient consensus requires t+1 rounds," *Information Processing Letters*, vol. 71, no. 3, pp. 155–158, 1999.

[12] O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman, "Cogsworth: Byzantine view synchronization," 2020.

[13] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pp. 427–438, 1987.

[14] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), p. 931–948, Association for Computing Machinery, 2018.

[15] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malki, O. Naor, D. Perelman, and A. Sonnino, "State machine replication in the libra blockchain," 2019.

[16] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 583–598, 2018.

A Protocol Security Proof

A.1 Committee Selection Security

Assuming that validator assignment is random and nonintersecting, the probability of a single shard safety is given by a simple combinatorial argument:

$$p_{\texttt{local\_fail}} \coloneqq \mathbb{P}\left(X \geq \lfloor m \cdot f \rfloor\right) = \sum_{x=\lfloor m \cdot f \rfloor}^{m} \frac{\binom{t}{x}\binom{n-t}{m-x}}{\binom{n}{m}}$$

where we have used the following notation:

- n – total nodes
- F – safety threshold fraction in the network
- $t = n \cdot F$ – total faulty nodes
- m – shard size
- f – safety threshold fraction on a shard
- X – number of faulty nodes in a shard

It can be shown that if $F \geq f$, then $p_{\texttt{local\_fail}} \geq 1/2$. Therefore there is an inherent need to set safety thresholds on main shard and local shards differently.

B ShardDAG Deletion

The purpose of a sharded blockchain is to allow the system to scale to larger throughput than is possible in non-sharded blockchains. It is therefore necessary to consider the scaling properties of the shardDAG. Summarizing the shardDAG system described above, validators are required to

- Receive and store shard block headers from all shards.
- Receive and store shard block outboxes of cross-shard transactions from all shards.
- Send shard block headers that have been stored.
- Send shard block outboxes of cross-shard transactions that have been stored.

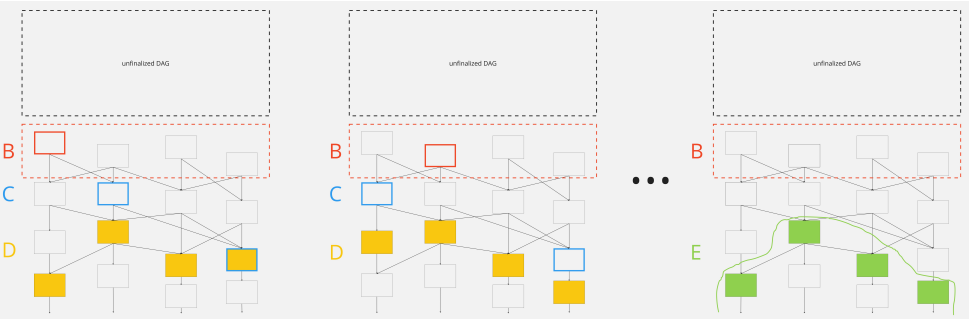

Figure 7: *An illustration of the sampling procedure for determining shardDAG data that can be deleted. A set of entire ZK finalised shard blocks B (red dashed rectangles) are sampled. Left: the red shard block contains cross-shard transactions created within shard blocks C (blue bordered shard blocks), hence the red shard has already processed all cross-shard transactions in blocks that fall under any blue block. These form D, noting that an element of C can be present in D iff it falls under another element of C, here indicated by the yellow block with blue border. Centre: Similar to left, but for a different element of B. Right: E is composed of the lowest shard blocks for each shard in all D's. All cross-shard transactions created by shard blocks in E's subgraph have been processed and ZK finalised, thus E's data can be deleted.*

Over time storing the outboxes of cross-shard transactions would accumulate into a large database and become a burden on validators. Fortunately, each outbox of cross-shard transactions produced by other shards need only be stored until it can be confirmed that its contents have been included in a shard block in each of their destination shards, and those shard blocks are finalised and cannot be rolled back, i.e. included in a consensus block, and all required ZK proofs have been created and validated.

Validators do not receive or store processed transactions in each shard block, meaning that validators cannot track precisely when old cross-shard transaction data is no longer needed and can be deleted. Instead, periodic sampling can be used to efficiently determine when old cross-shard transaction data is no longer needed. To do so, validators can use the following process illustrated in Fig. 7.

1. Periodically sample a set B of ZK finalised shard blocks (that cannot be rolled back), one for each shard. It is necessary to obtain each block's list of processed transactions and cross-shard transactions.

2. For each shard block b in B, find the set of shard blocks C originating each processed cross-shard transactions.

3. For each C, construct the set of shard blocks D that are the highest shard blocks that are in the subgraph of an element of C, but are not themselves an element of C unless it falls under another element of C. According to the protocol ordering rules, any cross-shard transactions contained within shard blocks that fall under C have already been processed. D defines a cut across the shardDAG that distinguishes shard blocks that must have, and may not have been completely processed.

4. Across each D for all shards, extract the lowest shard block for each shard to construct the set of shard blocks E.

5. The validator can delete shard block headers and cross-shard transaction data for shard blocks that are in the subgraph of E, including the elements of E (but excluding data from the validator's own shard which is required for participating in that shard).

To function efficiently step 2 requires that shard blocks contain hash data linking each processed cross-shard transaction to the shard block that created it.

The sampling frequency should be short enough that validator storage requirements are manageable. ZK block proofs are expected to require approximately 10 minutes to produce, thus the sampling period should likely be some multiple of this. Storing an hour, or few hours worth of data is insignificant compared to storing an entire history, and sampling a single block from each shard this often is not a significant burden.