=nil;

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

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

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

Mikhail Komarov
=nil; Foundation
nemo@nil.foundation

April 17, 2024

Contents

Chapter 1

Introduction

Increasing complexity of distributed applications which are required to work within untrusted environments, bringing an increase in requirements for the overall such database cluster 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 1.1, 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 **??** introduces fundamental definitions and the models within which the protocol operates. Section **??** discusses how individual shards achieve consensus and process transactions within Byzantine Fault Tolerant (BFT) settings. Section 2 explains the collaboration among shards to ensure global security guarantees. Section **??** explores techniques aimed at reducing transaction processing times in sharded configurations. Section **??** describes the sequencing of transactions for zkSharding, including how zkSharding's Data Availability transactions are ordered on Ethereum. Section 3 delves into the Data Availability mechanism utilized by zkSharding and provides an overview of Layer-1 finalization. Section **??** examines the state transition proof mechanism employed for dual purposes: Layer-2 state finalization and the inheritance of security from Layer-1.

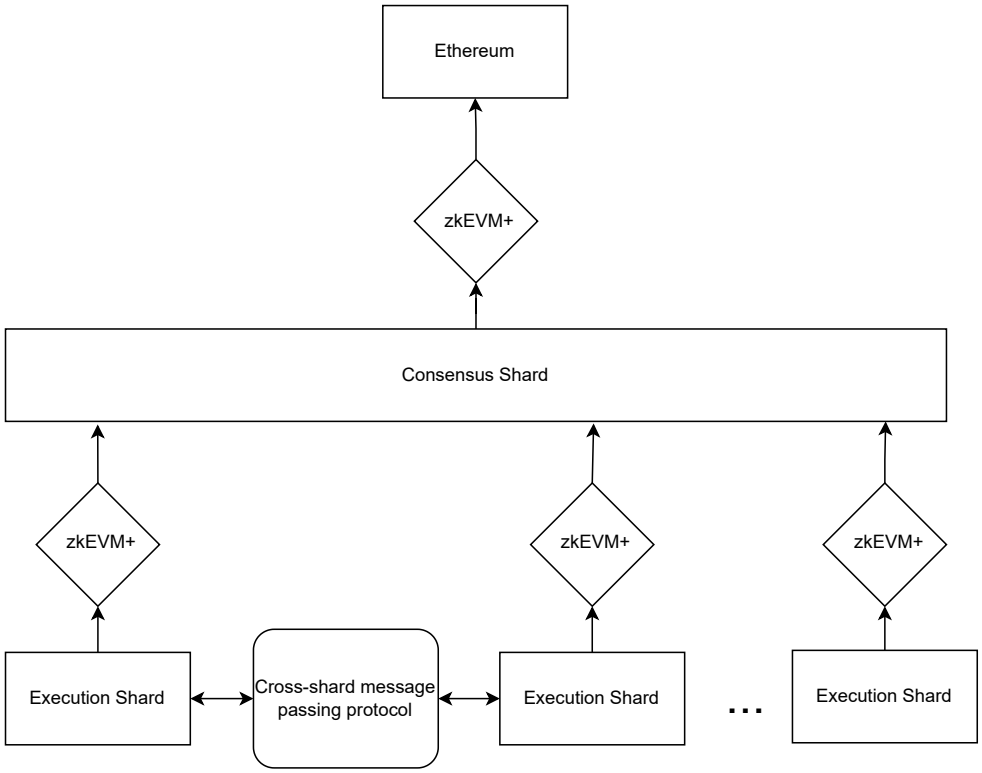

Figure 1.1: *zkSharding Architecture*

1.1 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). The ramework divides a 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.

1.1.1 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 **??**.

1.2 Intra-shard Replication

An *account* is a minimal data unit from the sharding algorithm perspective. An account is characterized by its address (public key) 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 each shard is discussed in this section.

1.2.1 Shard structure

One of the key properties of the BFT SMR is that the state of the system can be restored from the log of committed transactions.

Remark. Split and merge conditions also describe a way to split/merge the state of the shard so that the state can be restored. It is discussed in more detail in Section **??**.

A way to achieve this property is to use a blockchain data structure[6] for a shard's transactions commit log. The blockchain is a sequence of replication packets (or simply blocks) $\{B_k\}_{k=0}^{K}$ along with a validity predicate `isValid`. Here and in subsequent sections, the terms 'block' and 'replication packet' are used interchangeably.

In more detail, following notation from [7], each packet B_k contains a sequence of transactions T_k and a reference to the previous packet $h_{k-1} \coloneqq H(B_{k-1})$:

$$B_k \coloneqq (T_k, h_{k-1}).$$

Ilia S.
It's not empty, it contains some state,

The first packet B_0 is called the *genesis state* and is defined as $B_0 \coloneqq (\emptyset, \perp)$, where $\perp$ is a special value.

The validity predicate `isValid` is defined as follows:

$$\texttt{isValid}(B_0) \coloneqq \texttt{true},$$
$$\texttt{isValid}(B_k) \coloneqq \texttt{true} \iff \texttt{isValid}(B_{k-1}) \wedge \texttt{isValidBlock}(B_k),$$

where `isValidBlock` is a predicate that checks the validity of a block defined on an application level.

Remark. The block (or packet) validity predicate `isValidBlock` also contains protocol-level checks:

- *Message passing check:* the block processes all necessary outgoing messages from neighboring shards
- *Split/merge conditions check:* the block satisfies split/merge conditions
- *State consensus check:* the block is committed by a designated committee

It is discussed in more detail in Section **??**.

1.2.2 Local consensus

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 replication packet. 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 1.1: *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 replication packet 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.

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 packet 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`

Chapter 2

Dynamic 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. 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 subset of tables (accounts), defined by the deterministic function

$$F_S : (pk, M_{shards}) \rightarrow id_{shard}$$

In this context, pk represents the account's public key, while M_{shards} refers to the history and metadata of shards stored in the Consensus Shard.

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 1.2.

In accordance with the parameters outlined in the Consensus Shard, each shard has a maximum block capacity. The dynamic sharding behavior is influenced by two events:

- *Split Conditions are met*: If, during the previous N replication cycles, a shard's block occupancy approaches its capacity, the shard is split into two. The exact values of N and the fullness threshold are defined by the protocol's parameters.
- *Merge Conditions are met*: Conversely, if during the last N replication cycles, the replication packet occupancy of two shards remains substantially below capacity (with at least half the block remaining vacant), then a merge of the two shards is initiated.

2.0.1 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.

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

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.

2.0.2 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.

Each committee has additional tasks beyond just maintaining their shard. They are responsible for tracking a specific type of events, namely cross-shard messages, within *near* shards. Near shards are determined based on the Hamming distance in shard identifiers.

More specifically, block is considered to be valid if it propogates all *necessary* messages. Each outgoing message has a destShard field, which uniquely determines nextShart identifier:

$$\texttt{nextShard} = \texttt{NextHop}(\texttt{destShard}, \texttt{shardId})$$

If nextShard is equal to the current shard, then the message is considered to be *necessary*, and block proposer must include it into the block along with a merkle proof of inclusion to the set of outgoing messages of the neighboring shard. Validation of the included messages is a part of isValidBlock function mentioned in Section 1.2.1.

2.0.3 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, 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.

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

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 2.0.2) 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 1.1.1, 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 3.1) 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 2.0.4. 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.

2.0.4 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 3.1.

Despite the introduction of mechanisms such as the two-level consensus protocol (Section 2.0.3), Multi-Threshold BFT (Section 1.1.1), and shard committees determined by neighborhood size (Section 2.0.3), 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.

2.0.5 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 $\{pk_1, pk_2\}$ are consistently located within the same shard. In other words, $F_S(pk_1, M_{shards}) = F_S(pk_2, M_{shards})$ for every possible value of M_{shards}.

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, 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.

Define the *co-location depth* of an address A, denoted as $|S_A|$, to be the cardinality of the set encompassing all addresses co-located with A. Formally, $|S_A| = |\bigcup S \mid S \cap A|$.

Limitations are split into two parts:

- Economic restrictions. The cost of each co-location depends on the resulting co-location depth of the address.
- Ownership or domain restriction. Only addresses controlled by the same key (potentially a multi-sig key) may be co-located.

Domain restrictions are introduced to prevent attacks on popular applications by malicious users co-locating their addresses with a popular one, thereby preventing the addition of new modules by developers.

Economic Restrictions

To initialize a previously unmentioned address, it is necessary to send an initialization transaction to this address, containing initial code, initialization values, and a fee.

The basic address deployment cost is defined as the sum of the following parameters:

- Basic transaction fee
- Address activation cost
- Application bytecode size
- Constructor call cost
- Reserved data size by the application
- Initial values costs (as part of transaction additional data)

The co-location's economic restrictions form part of the address creation costs. These costs for address A are defined by the following formula:

$$\texttt{address\_creation\_fee} = b \cdot k^{|M_A|},$$

where b represents the basic address creation cost, and k is the co-location multiplier. Both parameters are set in the configuration and can be updated by governance.

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.

Ownership Restrictions

The requirement to bind addresses in some manner introduces ownership restrictions. These can be implemented in two ways:

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

- Binding solely through an explicit map of co-located addresses.
- A functional definition of co-located addresses entails the ability to derive one address from another. This is possible given that addresses are public keys of some digital signature scheme, which allows for such derivation.

In general, it is not feasible to define a "master" key from a derived key. Therefore, an explicit map of bindings must be stored in any case. Furthermore, cryptographic derivation does not add security because forging the map of co-located addresses would require compromising the protocol itself. Thus, the explicit map of the co-located addresses is solely utilized.

The concept of a *domain* is introduced. A domain is a set of co-located addresses defined by the master key.

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.

In this scenario, the overall address creation costs become:

$$\texttt{address\_creation\_fee} = b \cdot k^{|M_A|} + e,$$

where e is the execution cost of the co-location manager.

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
    };

    // 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.

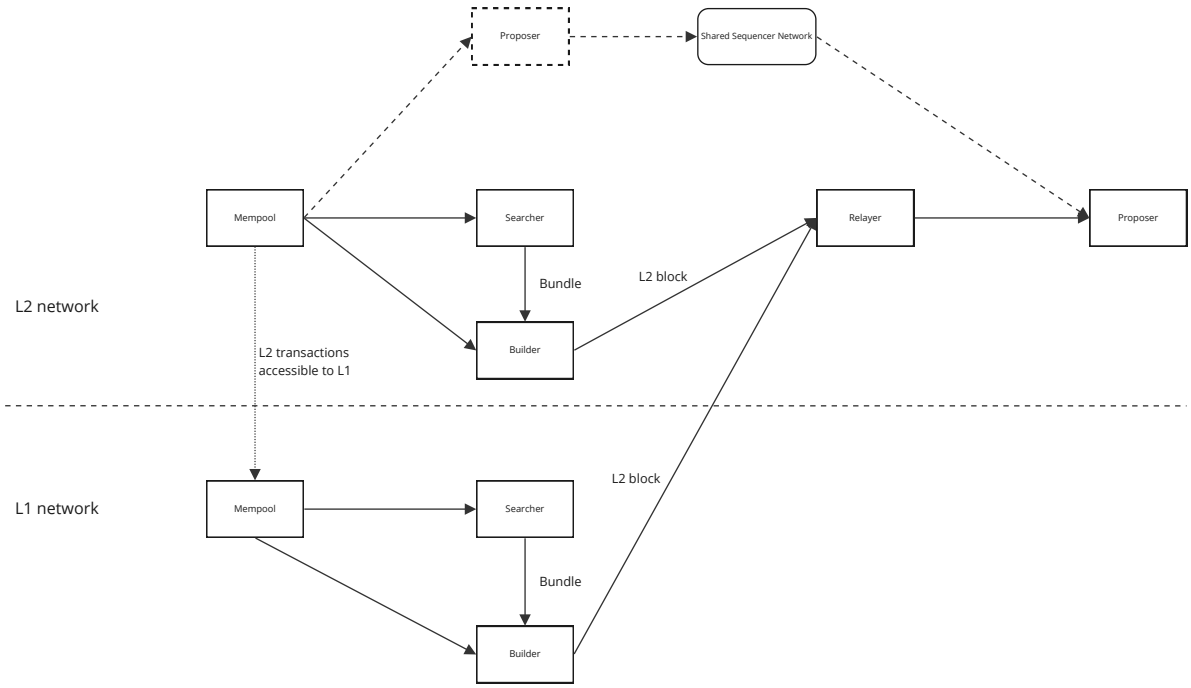

Figure 2.1: *Sequencing model*

2.1 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 4.1 for an illustration of the high-level schema.

2.1.1 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 4.1 (dotted line).

Remark. Split and merge events can represent a significant shift for builders and searchers. Any changes to the entire network, including the involved members, will be communicated through the Consensus Shard. In practice, without maintaining a long queue of pre-prepared blocks, the merge event will not

introduce significant changes to the constructed builder's blocks (e.g., shardID). Relayers will need to reapprove signatures in the new committee after the event to continue delivering blocks to the proposers.

2.1.2 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.

Chapter 3

Data Availability

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

3.0.1 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 4). 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).

image here

3.0.2 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.

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:

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

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.

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.

3.0.3 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.

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 high-level concept is illustrated in Figure 3.

insert image from Petr and rename figure

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.

3.1 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 $\mathcal{F}$:

$$\mathcal{F}(S_i, T, S_{i+1}, PI) \rightarrow (\pi) \tag{3.1}$$

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 $\mathcal{V}$:

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

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:

$$\mathcal{F}(S_i^0, \ldots, S_i^{k-1}, S_{i+1}^0, \ldots, S_{i+1}^{k-1}, T^0, \ldots, T^{k-1}, PI^0, \ldots, PI^{k-1}) \to (\pi) \tag{3.2}$$

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*.

3.1.1 Proof Generation Protocol

Define three types of proofs for the protocol:

- π_S is a state transition proof defined by Equation 3.1.
- 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 3.1 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.

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.

Global State Transition Proof

Note that Equation 3.2 can be represented as:

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

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.

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.

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 3.0.1.

Chapter 4

L1-Composability

Ensuring compatibility between Ethereum and its associated Rollups/Layer 2 (L2) solutions is indispensable for the development of a sophisticated infrastructure, ecosystem, and applications. This compatibility not only facilitates the efficient migration of existing applications from Ethereum networks to Rollups but also minimizes associated efforts and costs. A crucial aspect of this compatibility is the ability to write on the Layer 1 (L1) from the Layer 2 network. This paper presents two distinct approaches at the protocol level within the `=nil;`'s zkSharding framework to achieve this essential functionality.

4.1 Background

4.1.1 Notations

1. Tx – native network transaction;
2. Txe – transaction, or with at least one external call, or with at least one Ethereum transaction, or with a target to Ethereum address;
3. H(d) – digest function of input "d";
4. Pb – public key;
5. Pk – private key.

4.1.2 External call

The external call, referred to as "extcall" denotes the segment of code wherein an invoke to the Ethereum application occurs. This invocation can transpire explicitly or implicitly through a function containing the "extcall" keyword. Various methods exist to initiate it, but the one that is both straightforward and compatible with Ethereum Virtual Machine (EVM) bytecode involves utilizing an "unaligned address" akin to the approach employed in RISC-V and ARM architectures with Thumb support.

By default, function addresses in Ethereum are represented as 20 bytes in little-endian format. Notably, the 21st byte can be designated with a value of 1 to signify the occurrence of an external call. This adjustment doesn't necessitate the introduction of new opcodes; rather, it requires processing logic within the Virtual Machine (VM). To illustrate, consider the following example code:

```
function someFunction public extcall {
    // 20-bytes USDC contract address on Ethereum
    IUSDC usdcContractOnEth = 0x2200000000000000000000000000000000000011;
    // as it's excall compiler will set 21-bytes address:
    // 0x012200000000000000000000000000000000000011
    usdcContractOnEth.unlock(....); // extcall 1
    usdcContractOnEth.transfer(....); // excall 2
}

interface IUSDC {
```

```
    function unlock() extern extcall; // extcall function keyword
}
```

This capability is feasible and aligns with EVM principles by storing data for the call opcode on a 32-byte size stack. In this scenario, when the EVM casts to "address," it effortlessly disregards the surplus data. For VMs equipped with extcall support, distinguishing the call type becomes straightforward by examining the address using: $addr \wedge 2^{161}$. To convert extcall address to a normal 20-byte address one can perform the following operation: $addr_{target} = addr_{extcall} \wedge (2^{161} - 1)$.

This approach doesn't mandate the introduction of any supplementary opcodes or alterations in functionality. It ensures seamless compatibility with existing EVM bytecode.

4.1.3 External transaction

There are three criteria by which a transaction on nil;'s cluster should be marked as external:

1. It has at least one user's signed Ethereum transaction in the "data" field. They can be several and the order will be preserved by batching them. It must be mentioned that $\forall Tx_i, Tx_j => i = j + 1, Tx_{i_{nonce}} = Tx_{j_{nonce}} + 1$. Otherwise, the transaction will be considered invalid.
2. The transaction contains a call to the function marked with "extcall".
3. The target address is the Ethereum address.

It is noteworthy to emphasize that when a call is made to an extcall function, the callee should be explicitly marked as an extcall as well, even if the execution path might not eventually trigger it. This is particularly relevant in scenarios where an external function is encapsulated within a branch statement that evaluates to false during execution.

4.1.4 Composability with Ethereum

Ethereum, as designed, lacks inherent support for write composability with L2 solutions and doesn't readily offer technical avenues for seamless integration. A pragmatic and feasible strategy involves the explicit emission of Ethereum transactions, generating them explicitly at the protocol level.

However, undertaking this approach presents several challenges that necessitate resolution. Among these challenges, one of the most formidable is the handling of Ethereum Externally Owned Account (EOA) signatures.

4.1.5 Atomicity

One of the fundamental attributes in traditional databases is the atomicity of a transaction, ensuring that the system is in a complete and consistent state either after the full transaction is applied or when it is not applied. While write composability can assure local atomicity within a given context, achieving global atomicity faces challenges in the Ethereum network due to the asynchronous nature of external transactions.

The asynchronous implies that immediate feedback is unavailable, necessitating the adoption of pull or callback approaches for transaction status confirmation and result, while execution of the transaction continues without considering external transaction status. It's crucial to acknowledge that ensuring global atomicity in such an environment is complex. While there exists an approach rooted in deferred calls to secure global atomicity, its practical implementation is often both expensive and challenging.

4.1.6 Ethereum authentication schema

Ethereum employs the ECDSA secp256k1 standard signature for transaction verification, which consists of two numbers: (r, s). This signature, when paired with the public key and message, is required for signature verification. Notably, the ECDSA scheme possesses the unique property to reconstruct the public key from the values (r, s).

In the Ethereum context, when referencing msg.sender in EVM, it doesn't rely on the "from" field of the transaction or the direct use of the public key hash. Instead, it performs a process of public key recovery from the signature. This involves verifying the signature, reconstructing the public key, and subsequently deriving the address by hashing the public key. The resulting address is then stored in

tx.origin and *msg.sender* by default. Note that we do not cover the aspects of parameter v in secp256v1 as it's not important for the schema. In Algorithm 1 the detailed structure of EAS is presented.

This method dictates that an Ethereum transaction must be signed with the user's private key for the process to be successful. That narrows the potential set of viable solutions. Consequently, the reliance on the user's private key for transaction signing is a fundamental requirement for the write composability.

Algorithm 5: High-level Ethereum Authentication Schema (EAS)

1. Alice has EOA with dedicated P_k and P_b as per $secp256k1$ (defines G, n, m);
2. Alice creates transaction with the target contract she wants to invoke;
3. EIP-155 specifies the content that should be signed: $(nonce, gasprice, startgas, to, value, data, chainid, 0, 0)$; *last two zeros – (r, s) pair that not yet defined on the current step
4. Signing process:
   - 4.1 h = H($nonce$ || $gasprice$ || $startgas$ || to || $value$ || $data$ || $chainid$ || 0 || 0);
   - 4.2 random k selected;
   - 4.3 $r = (G * k)_x \% n$;
   - 4.4 $s = (h + r * P_k) * k^{-1} \% n$;
   - 4.5 Output: (r, s).
5. Bob restore Alice's public key:
   - 5.1 Input: $T_{x_A} =>$ ($nonce$, $gasprice$, $startgas$, to, $value$, $data$, $chainid$, r, s);
   - 5.2 h = H($nonce$ || $gasprice$ || $startgas$ || to || $value$ || $data$ || $chainid$ || 0 || 0);
   - 5.3 secp256k1 curve: $R_y = TonelliShanks(r^3 + 7, m)$;
   - 5.4 $R = (r, R_y)$;
   - 5.5 $t_1 = -h * r^{-1} \% n$;
   - 5.6 $t_2 = s * r^{-1} \% n$;
   - 5.7 $P_b = (G * t_1 + R * t_2) \% m$;
6. Bob verifies signature:
   - 6.1 $u = h * s^{-1} \% n$;
   - 6.2 $v = r * s^{-1} \% n$;
   - 6.3 $C = (G * u + P_b * v) \% m$;
   - 6.4 $C.x == r$;
7. Bob restores Alice's Ethereum address:
   - 7.1 $Addr = keccak256(P_b)$;
   - 7.2 In EVM: $msg.sender = address(LE(Addr[0:20]))$;

4.2 Overview

Both presented approaches are concluded with their essential advantages and disadvantages. That should be mentioned the technical feasibility was confirmed on the artificial simulation of the solutions.

4.2.1 External transactions on-demand

A high-level and step-by-step description of the processing of external transactions by demand is presented below (Figure 1).

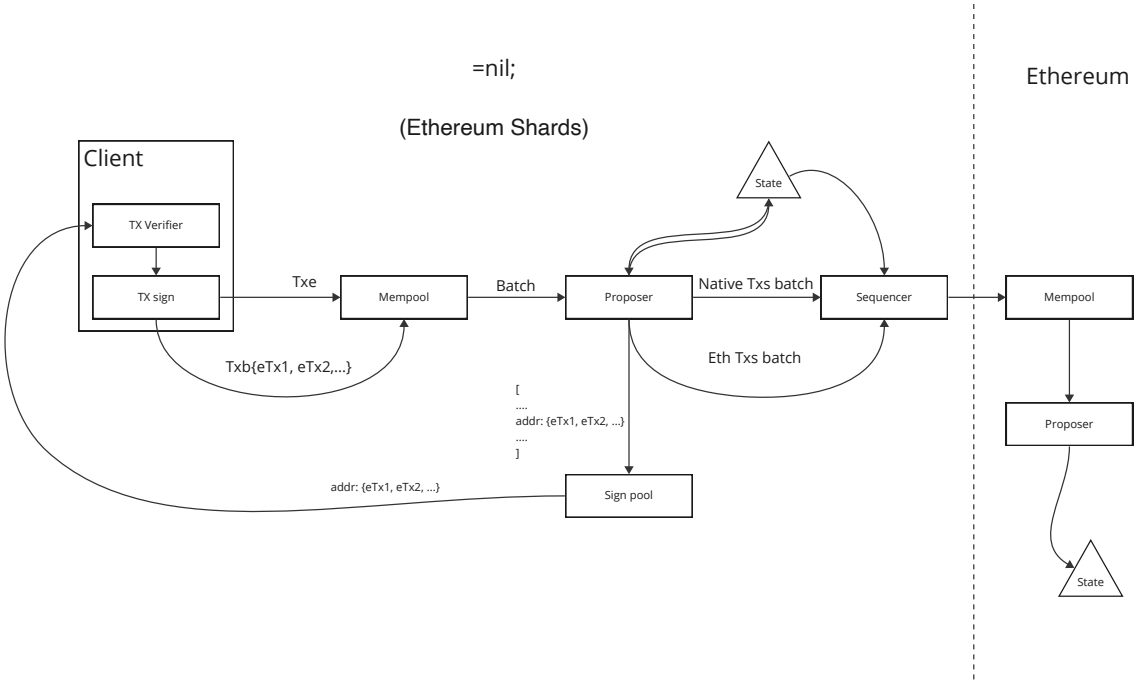

Figure 4.1: *On-demand transactions processing design*

We cover the most comprehensive case that will trigger the sub-sceneries with other external transaction types.

1. A user creates an external transaction on the nil;'s cluster. That transaction invokes a function marked with extcall, from the contractnatively deployed on the cluster. Formally, the transaction may look like this:

```
{
    chainID: <nil chain ID>,
    from: <user addr>,
    to: <nil 20-bytes contract address>,
    type: "extcall",
    ....
    data: <function selector>||<input params>
}
```

2. The transaction is sent to the mempool;
3. Builders on their own or with the help of searchers derive and batch transactions from mempool. Then batch is sent to the proposer by auction or with the help of the relayers;
4. Proposer validates the transaction, and there can be several cases:
   - 4.1 The transaction is marked as native – no Ethereum transaction will be generated;
   - 4.2 The extcall transaction – that is expected that there will be generated some number of Ethereum transactions. User Ethereum nonce should be provided;
   - 4.3 Ethereum address as the trarget (field "to") – there will be generated a transaction on Ethereum. User Ethereum nonce and transaction signature should be provided. Note that it's not mandatory for the user to provide the complete transaction, as RLP-serialized data can be easily reproduced from some basic metainfo and data from the user that will reduce the cost of native transactions, while validators can construct transactions for cheap;
   - 4.4 Batch of signed Ethereum transactions from the user – the proposer is not required to perform any specific actions but rather to propagate the ordered batch to the sequencer.

5. For the extcall transaction, each time during validation proposer detects 21-bytes address, it creates a special record in temporary memory, that may contain: new nonce, input data, target contract, function selector, callback address (if exists);
6. When native block validation is completed, the proposer sends to the signature pool a specially composed structure with prepared Ethereum transactions generated during validation process;
7. The user identified block confirmation containing their external transaction and by request gets a batch of Ethereum transactions from the signature pool;
8. There are several options to ensure the correctness of input data in L1 transactions. However, the simplest and most dependable method involves compelling other attestators to verify the Ethereum transaction data either. Through this process, users can assess the number of signatures from the block committee that have endorsed the batched transactions;
9. The user signs Ethereum transactions from the signature pool and constructs a new native transaction, consolidating L1 transactions into a batch within it. Subsequently, this transaction is sent to the mempool;
10. Upon detection of such a transaction, the proposer straightforwardly propagates the batch to the sequencer.

Client application

There will be specific changes in the client application, primarily evident in two distinct modules for signatures and transaction construction: one designed for Ethereum and the other tailored for nil;'s transactions. Even if the signature standard remains consistent, variations in the transaction signing process and content may arise.

Another crucial modification involves the verification module. The client application must verify the correctness of input data for Ethereum transactions generated by the proposer during the validation process of an extcall. The Ethereum transaction from the proposer should undergo verification before being signed to mitigate the risk of fraudulent activities.

In addition, it makes sense for the user verification mechanism to await the finalization of the corresponding block. This precaution is warranted due to the absence of atomicity in transactions, meaning that even if a rollback occurs in nil;'s cluster, the transactions submitted on Ethereum may not be reverted. Consequently, waiting for block finalization adds an extra layer of assurance in maintaining consistency and reliability in user verification processes.

Signature pool

The signature pool is a temporary storage for the records of Ethereum transactions that may look like:

```
H(<nil address>||<external transaction hash>) : [
    txs: {eTx1', eTx2', ...}, // non user-signed transactions -> (r, s) = (0, 0)
    sig: <aggregated signature of the committee>,
    preserve_commitment: ...,
    block_id: ....
]
```

preserve_commitment (prc) is a special field, that will allow removing record only when a user signs the transactions and send an aggregated external transaction. It can be constructed in the following way:

$$prc = H(H(txs)||(user_{commitment})),$$

where

$$user_{commitment} = H((Pb \times rand_{nonce}))$$

In this approach, the removal of an entity from the pool occurs only when the user signs the transactions and sends them back in the same order within the aggregated transaction. Additionally, the user includes their $rand_{nonce}$, confirming their intention to eliminate the record.

There is no necessity for a separate network pool to store these records, as they can technically be housed in the same area as the transaction pool. It is evident that the load on the mempool will not significantly increase, given the relatively small number of Ethereum transactions. Moreover, temporary storage costs are incurred to prompt users to sign transactions from the pool. Both these considerations suggest that a standard native mempool can effectively be employed to manage the signature pool.

Tradeoffs

Key advantages of this design include:

1. Ethereum transactions order preservation – enabling the development of sophisticated solutions. As an illustration, complex operations such as transferring funds from Ethereum to `=nil;` cluster can be seamlessly executed with just a single function call;
2. Arbitrary function calls on L1 – as users sign transactions explicitly, on the Ethereum side everything will look as if would user directly creates and sends it;
3. Simple implementation – the design doesn't require difficult decisions or architecture, while the implementation of the client side will not be overcomplicated;
4. An unlimited number of Ethereum transactions per one extcall function – as validator creates transactions and increases nonce, the potential number of Ethereum transactions limited only by native gas limit.

1. Active client application participation – is not as significant a concern in practice as it might initially appear, particularly for wallets and clients other than hardware devices. In any scenario, transactions are retained in the signature pool until the proposer confirms the submission to L1;
2. Separation of pools – despite the fact that division can be done virtually it may require additional development efforts to support.

4.2.2 External transactions pre-process

A high-level and step-by-step description of pre-processed transactions is presented below (Figure 2).

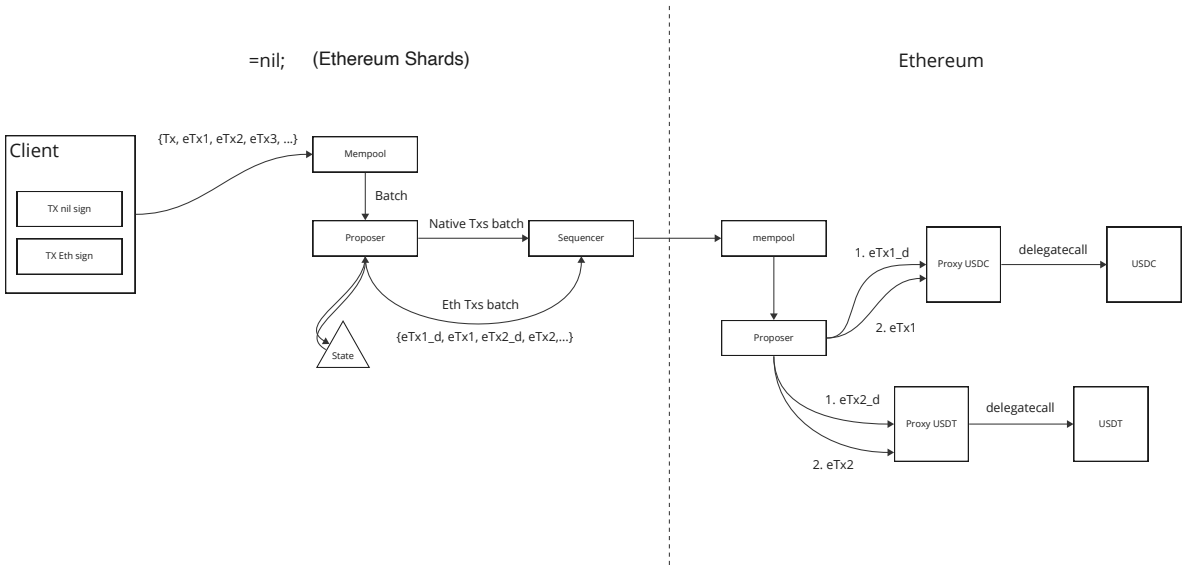

Figure 4.2: *Pre-processed transactions design*

The approach is called transaction pre-processing, as additional effort will be required from the user side. Below, we cover the scenario:

1. User generate an extcall transaction;
2. The client is required to preprocess a sequence of consecutively signed Ethereum transactions destined for a predetermined address; This address is determined in advance, often based on the anticipated execution flow, such as using an execution flow tree;
3. Builders on their own or with the help of searchers derive and batch transactions from mempool. Then batch is sent to the proposer by auction or with the help of the relayers;

4. Proposer validates the transaction. Each time validator detects extcall, it creates a special data transaction on Ethereum and "binds" (by ordering) this transaction with the corresponding L1 call transaction. In the Figure, it is shown as (eTx_d, eTx);
5. Batched Ethereum transaction sent to sequencer;
6. Sequencer aggregates Ethereum batch and submits it to L1 builders;
7. Ethereum proposers validate transactions, starting with the data transaction. In this process, the transaction puts data into the corresponding proxy contract;
8. Following the data transaction, the call transaction with the user's signature will be processed in sequence. This call is directed to the dedicated proxy where the data was previously saved;
9. The proxy contract function employs a call or delegatecall to the target contract. This function is invoked with the data stored earlier by the data transaction, signed with the committee's signature.

Before submitting a native transaction with an external call, the client must prepare a set of consecutive signed Ethereum transactions. The addresses for these transactions will be derived from the Control Flow Processor (CFP) from the source function. It is important to note that the use of extcall is not allowed under explicit branching statements (recursion, loops, conditions). Despite the fact that a revert implies an implicit flow break, it is still permitted within the main flow. The CFP will disregard implicit main flow branching. The overall approach enables the pre-construction of Ethereum transactions without obvious reduction, ensuring proper identification of the target address.

It is essential to note that changes to the proxy address must occur transparently without immediate replacement. This is because the proxy address is derived during the Ethereum state S_i, while the actual proxy call takes place during state S_j, where i<j. In the need of an upgrade of the proxy contract on Ethereum, the address on Nil's cluster will need to be updated after L1 finalization. Complete replacement on Nil will occur only after the finalization of the cluster. "state_source" stated before the last completed replacement will not be accepted.

The CFP will determine the number of Ethereum transactions to be prepared and the target address for each of them. The client constructs a batch of sequenced transactions with substituted addresses and signs them. Subsequently, the client prepares a joined native transaction that may appear as follows:

```
{
    chainID: <nil chain ID>,
    from: <user addr>,
    to: <nil 20-bytes contract address>,
    type: "extcall",
    ....
    data: <function selector>||<input params>
    state_source: <blockID>, // state from which the proxy address derived
    eth_tx: [
        eTx1, eTx2, ..., eTxn
    ]
}
```

Data transaction

In order to sign transactions on the L1, a data transaction must precede and store data on the proxy contract. This prerequisite arises because, for the transaction to be signed, data must be known in advance. The challenge arises from the fact that the input data is only ascertainable when the nil;'s contract invokes an external call during the execution process.

To overcome this limitation, the proposer initiates a new Ethereum transaction, termed a "data transaction". Within this transaction, a specific function on the proxy contract, such as *updateData* is invoked with already computed extcall data as a parameter. Subsequently, the proposer batches the data transaction along with the corresponding call transaction, ensuring that the update data call to the proxy occurs immediately before the data update. Through this approach, clients can proactively sign Ethereum transactions in advance, even without knowledge of the actual input parameters.

Proxy contract

Proxy contracts play a pivotal role in the execution of user-signed transactions by utilizing data previously saved by the proposers' data transaction. There are essentially two approaches to invoke target contracts from proxy: through the use of *delegatecall* and *call*. When employing *delegatecall*, the target contract is executed within the context of the proxy contract, with the proxy credentials remaining unaltered. Conversely, the *call* method allows the target contract to operate within its own context, while the sender credentials undergo a transition from an externally owned account (EOA) to those of the proxy. The most important credentials for such calls one should take into account are *msg.sender* and *msg.value*.

It is noteworthy that, as of the present, there exists no technical mechanism to invoke functions from another contract while preserving the target context without making modifications to the sender data. That means there must be made a choice between preserving own credentials or the target context within a call.

The code snippet below is an illustration of a proxy contract utilizing a delegatecall. In the initial data transaction, information is stored in the internal storage under a specific key, where the key may simply be a hash of the user's address. Before data is saved, a validation check is performed to ascertain if the externally owned account (EOA) is a valid nil;'s validator and possesses the authorization to save data. Subsequently, a user-signed transaction triggers the "transfer" function. Upon retrieval of data from storage using the user's address, the function initiates a delegatecall to the USDC contract. It is crucial to note that this call does not alter the context of the target contract.

```
contract USDC_proxy is IUSDC {
     ... <original usdc contract data fields> ...
    mapping(bytes32 => bytes) input_storage;

    //uint256 -- blockID after which the validator right expires.
    // if < curBlockID -- update reverted
    mapping (address => uint256) nil_validators;

    address usdc;

    function upateData(bytes32 key, bytes calldata _input) public {
        revert(ifUpdateNotAllower(msg.sender)); // Optionaly
        input_storage[key] = _input;
    }

    function transfer() {
        bytes memory raw_input = input_storage[keccak256(msg.sender)];
        (... input ...) = parseBytes(raw_input);
                (bool success, bytes memory data) = usdc.delegatecall(
        abi.encodeWithSignature("<abi>", ... input ...); // or call/staticcall
    }
}
```

Tradeoffs

1. Potentially faster than on-demand schema, because you won't need to perform nil's double transaction;
2. More robust liveness guarantees as user is not involved;
3. Complex client – it requires advanced CFP to precisely compute the number of extcall that will raise during execution;
4. Proxy contracts maintaining – all proxy contracts will have to be properly managed to be up to date with relevant target addresses and validators;
5. Double Ethereum transactions – for each call transaction, a dedicated data transaction is needed, which increases the final cost for users;

6. Proxy-contracts – target contracts require the implementation of dedicated proxies, thereby introducing heightened maintenance and development expenses; Utilizing any arbitrary Ethereum contract without a proxy is not feasible under this design;

7. Limited extcalls – no way to utilize extcalls with branching, which limits development flexibility and comprehensiveness of solutions;

8. Credentials limitation – any target contract whose logic is bound to *msg.sender* rather than *tx.origin* will not work properly;

9. Risks of malicious behavior – as the raw transactions are publicly available, malicious actors can submit them directly to Ethereum before the protocol commits it or sends data transactions. Despite not providing the actor with direct income gain – it has reputational and protocol risks.

4.2.3 Extacall feedback model

In order to obtain and process results from an extcall, an asynchronous callback mechanism can be employed. Contracts within the nil network that incorporate extcall functionality include specific functions designed to handle raw results from the target contract on Ethereum. Upon deployment, such a nil;'s contract initializes a lookup table for specific callback handlers, each identifiable by a simple ID.

When an extcall is expected to return a result or trigger an event, the contract explicitly initiates a special event, as demonstrated in the code snippet below. Subsequently, the EthDataProvider filters feedback data from the Ethereum network and invokes the appropriate feedback function within the nil cluster. An illustrative example of such a nil;'s application is presented below:

```
struct Handler {
    address handler_address;
    string abi;
};

mapping (bytes32=>Handler) lookup;

function someFunc(address addr) public extcall {
    addr.call(....); // extcall that is expected to return result to handle
    emit FeedbackRequest(
        address(this), // source contract
        keccak256(handler_address, abi), // handler id
        keccak256(addr, userEthAddress, userNonce) // unique id of extcall by which to filter
        );
}

function processResult(bytes32 handlerId, bytes callback data) {
    lookup[handlerId].call(lookup[handlerId].abi, data);
}
```

4.3 External transactions associated risks

For any novel and unique solution, a crucial aspect involves conducting a thorough risk assessment, which should be articulated alongside the proposed solutions. In this section, we delve into some key risks that demand consideration concerning the technology.

4.3.1 Inconsistency of Ethereum transactions

The term inconsistency means the conflicts of transactions on the L1 network from the same user. The risk of inconsistency among Ethereum transactions is substantial, primarily because proposers face challenges in accurately determining the latest nonce of an externally owned account (EOA). The options available, such as obtaining it from an internal Ethereum light client or including it in the nil's transaction, both carry inherent risks. Users must be mindful of providing the most up-to-date nonce, and they must refrain

from committing transactions on Ethereum before transactions from nil;'s cluster are submitted to L1. Failing to observe this sequence may lead to inconsistencies in transactions.

Furthermore, in both models, users are unable to assess the exact number of Ethereum transactions that will be submitted to L1 from L2 with their signature, making it unpredictable to determine the nonce value for the standalone user transaction in advance. The occurrence of duplicated nonces is problematic, as a transaction with identical or lower nonce is deemed incorrect and subsequently not processed. This becomes especially precarious when the rejected transaction is one originating from the cluster.

4.3.2 High final cost for users

The ultimate cost of Ethereum transactions can become a significant financial loss for users, primarily due to the necessity of setting a gas price higher than the current on-market rate. This precaution is essential because the sequencer may decline to include the transaction in a block, risking rejection from the builder if the market price rises more than one is set in a transaction. As the delay from signing Ethereum transaction to pushing it to the builder may be long enough, a higher gas price needs to be established in advance, thereby inflating the final cost for users. This cost increese is particularly high in the preprocessing model, where Ethereum transactions are duplicated.

4.3.3 Uncomputable timeframes

In both design cases, the timeframe between sending a nil's transaction and making a call on the Ethereum target application is entirely unpredictable. Several significant factors contribute to this uncertainty. Firstly, batched transactions are not guaranteed to be included in the subsequent sequencer block. Secondly, the timing of when a block will be accepted by the relayer or builder (depending on the sequencer model) is uncertain. In the case of the on-demand design, this timeframe also encompasses the time required for user signing and transaction verification.

These factors collectively imply that despite the relative predictability of the finalization of the cluster, the finalization of external transactions is unfeasible to precompute.

4.3.4 Unrevertable transactions

In the current design, achieving full atomicity is not feasible. Consequently, there exists a risk wherein the nil;'s cluster state may undergo reversion while the transaction on Ethereum successfully executes. This vulnerability is particularly critical in the preprocess model, where a validator could potentially exploit maliciousdata as input for a user transaction signed in advance. Even if the state of the nil;'s cluster is reverted, the malitius data and call transactions on L1 can still proceed successfully, posing a substantial risk to users, especially in scenarios where the execution of a malicious transaction on Ethereum prevails over slashing risks. The on-demand model mitigates this vulnerability by allowing users to await block finalization on the cluster before signing the proposed transaction, thus circumventing the mentioned pitfall.

Bibliography

[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.

Appendix 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.