zkSharding Solution Brief

January 23, 2025

1 Overview

=nil;'s zkSharding introduces a scalable sharded-zkRollup solution for Ethereum that leverages sharding and SNARKs to address the network's scalability challenges while maintaining decentralization and security. Unlike conventional rollups, zkSharding partitions the network into parallel execution shards, where each shard processes transactions independently while maintaining unified liquidity and state. The contracts deployed on these shards communicate *asynchronously* [**?**], enabling them to send messages to contracts on other shards without pausing execution to await the message results. This approach allows zkSharding to achieve significantly higher throughput without fragmenting liquidity or increasing the complexity of cross-shard interactions. In this document, we explain how we achieve these objectives in detail.

At a high level, zkSharding is composed of three interconnected components. These components are not structured as hierarchical layers, but rather work collaboratively, each serving a distinct role:

1. **L1 (Ethereum):** Acts as the settlement and data availability layer. Shards submit aggregated state proofs and data commitments here for finalization and security.

2. **Main Shard:** Coordinates execution shards, manages cross-shard communication, and ensures system-wide state synchronization and integrity.

3. **Execution Shards:** Handle user transactions and smart contracts in parallel, each maintaining its own state and processing a subset of transactions.

zkSharding's primary objective is to enable scalable computation within a decentralized framework. By dividing workloads across multiple shards, zkSharding increases transaction throughput without centralizing control. L1 acts as an additional layer of security by verifying zkSharding's state transition validity through submitted validity proofs. These proofs allow zkSharding to integrate with Ethereum's canonical blockchain. In this way, it ensures efficiency and security while preserving decentralization. Next, we give an overview of the role of L1 in zkSharding system before diving into the design of zkSharding (See Figure 1).

1.1 L1 Support in zkSharding

From L1's perspective (see Figure 1), zkSharding is a blackbox which periodically submits cryptographic proofs and data blobs to verify its operations without revealing internal processes. L1 provides key contracts for state validation, deposits, withdrawals, and data availability. While L1 provides zkSharding with a reliable settlement layer to verify zkSharding's state and a data availability layer, zkSharding enables L1 to achieve higher throughput by offloading transaction processing to zkSharding. The L1 network is unaware of the detailed execution inside zkSharding, but it plays a crucial role in maintaining zkSharding's trustworthiness and security.

For more insights into zkSharding's internal mechanics and how it achieves high throughput, refer to the next sections. Now, we introduce the key functionalities that L1 provides to support zkSharding.

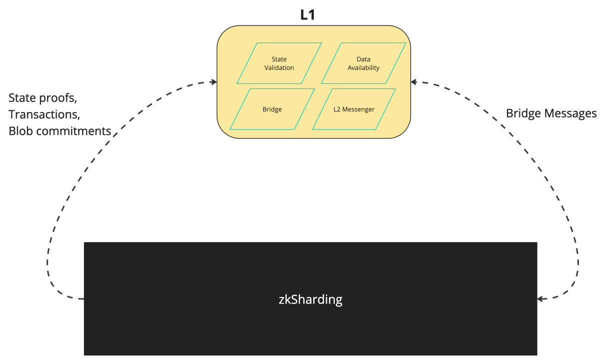

Figure 1: *Visualizing zkSharding as a* blackbox

L1 Data Availability: Proto-Danksharding (EIP-4844) [?] is a proposal aimed at reducing rollup costs when posting data to Ethereum's L1. zkSharding leverages this by utilizing Ethereum's data availability layer. zkSharding submits executed transactions as a *compressed* batch, denoted as $\mathcal{B}^*$, in the form of a data blob. This blob is committed to Ethereum's execution layer via a KZG commitment [?], which represents the data as a polynomial for efficient storage and verification. Batch compression is utilized to maximize the number of transactions that can be efficiently stored within a single blob, ensuring optimal use of available space.

State Validity Contract: The State Validity Contract is crucial for ensuring the correctness of zkSharding's new state after processing transactions across all shards. It verifies that the new state correctly reflects the processed transactions, represented by $\mathcal{B}$, from *all shards*. When zkSharding submits a new state root s to L1, it also provides a proof π to prove the correctness of this new state. The contract checks this proof against the KZG commitment Com_{da} to the compressed batch $\mathcal{B}^*$ of $\mathcal{B}$.

More specifically, the State Validity Contract verifies that the new state root s was created by correctly applying the transactions from the commitment Com_{da} to the last verified state st' with the root s'. Conceptually, the state update process in zkSharding can be represented by the state transition function F, which takes the current state st' and a batch $\mathcal{B}$ of transactions from all shards, and produces a new state st with the root s by executing all the transactions correctly.

The State Validity Contract receives s, s', and the proof π as inputs and also accesses the associated polynomial commitment Com_{da}. It then verifies the following:

- there exists $\mathcal{B}$ such that $\mathsf{F}(\mathsf{st}', \mathcal{B})$ outputs a new state st with the root s, and
- $\mathcal{B}^*$ committed in Com_{da} is the compressed version of $\mathcal{B}$.

Deposit/Withdrawal Contracts: Deposit contract secures asset transfers from Ethereum to zkSharding. When users deposit assets like ETH or ERC-20 tokens, they are locked on L1 and reflected within zkSharding for use on L2. The Withdrawal Contract handles asset transfers from zkSharding back to L1. Once zkSharding processes the withdrawal and generates a proof, the contract verifies it and releases the assets on Ethereum, ensuring a smooth exit to L1.

We have additional operational contracts deployed on L1, but here we focus on the key ones that ensure the secure functioning of zkSharding.

2 zkSharding Architecture

As previously discussed, zkSharding operates across three interconnected components. Each component serves a distinct function, yet they work together to ensure scalability and security. In this section, we explore how they are linked and describe the architecture that enables state synchronization. We also introduce the key actors who play critical roles in the system.

2.1 Actors

The core participants in zkSharding's architecture are ***validators***, who play vital roles across different layers. zkSharding launches with a centralized set of validators. Validators can operate in multiple roles:

Validators are responsible from building and maintaining the main shard and execution shards. When operating the main shard, they participate in running the *global* consensus algorithm which helps to the synchronization of states across all execution shards. In the execution shards, validators are responsible for executing shard-specific transactions and maintaining its local consensus. Beyond this, validators can be the part of the synchronization committee $\mathcal{SC}$. The members of $\mathcal{SC}$ are responsible for interfacing zkSharding with L1. They manage tasks like submitting data, proofs, and transactions to the L1 contracts outlined in Section 1.1. The committee is re-elected each epoch based on protocol parameters, and only validators opting for this role participate. To maintain clear role separation, a validator cannot simultaneously serve on both the main shard and the $\mathcal{SC}$ committee.

In addition to validators, zkSharding has a role ***prover***. Each prover P is part of a dedicated proving network and is responsible for executing zkSharding's proving algorithm (see Section 4).

Furthermore, there is the role of ***relayers***, who ensure the reliable transmission of contract calls from Ethereum to zkSharding.

2.2 Components

In this section, we describe the structure and functionality of the main shard and execution shards.

2.2.1 Main Shard:

Main shard functions as a specialized blockchain that manages operational transactions crucial to the integrity of zkSharding. The key operational transactions are as follows:

- **Randomness Generation:** To support unpredictability outcome of some protocols in zkSharding, the main shard incorporates a randomness generation mechanism.
- **Bridge-Related Operations:** The main shard is responsible for managing operations related to bridging assets and data between zkSharding and Ethereum.
- **Finalizing Execution Shard Blocks:** Once an execution shard completes a block, its header is submitted to the main shard, which verifies and finalizes the block header by including it in the global consensus (see Section 3.2)

The main shard functions similarly to a referee in a game. Its role is to verify the block headers from execution shards to validate that each shard adheres to network rules. This setup ensures that all execution shards maintain a synchronized view. It is achieved by the global consensus provided by the main shard.

The global consensus in the main shard coordinates and synchronizes the states of all execution shards. It provides a consistent and unified system state. This internal process is distinct from L1 consensus, which validates and finalizes the state of both the main shard and execution shards by verifying state validity proofs. While L1 guarantees the validity of zkSharding's state, it is not involved in the internal synchronization or consensus mechanisms of zkSharding, which focus on managing shard execution and coordination, In a nutshell, L1 consensus offers an additional layer of protection through Ethereum's strong security guarantees. This design combines the decentralized scalability benefits of sharding with Ethereum's proven security. To summarize the consensus layers shown in Figure 2:

- Execution shards run local consensus with their validators to maintain synchronization of their state in the shard.

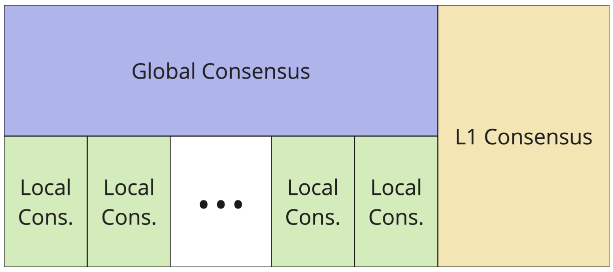

Figure 2: *Consensus layers in zkSharding: Local consensus aligns execution shard states, global consensus by the main shard synchronizes them, and L1 consensus on Ethereum finalizes the state of both main and execution shards for overall system security.*

- The main shard runs global consensus, ensuring synchronization and consistency of states across all execution shards.
- L1 verifies and finalizes the states of both the main shard and execution shards, ensuring overall zkSharding finality.

2.2.2 Execution Shard

In zkSharding, there are multiple execution shards, each managing its own set of accounts. An account is the fundamental data unit in each shard and includes the address, balance, storage root, and a hash of the contract's source code. Notably, all accounts in zkSharding are represented by smart contracts. By structuring all accounts as smart contracts, zkSharding centralizes operational logic across the system. It simplifies message handling and state updates.

Each execution shard operates a dedicated mempool that temporarily stores external messages sent by users, dApps, or other external sources. These messages are queued in the mempool until the associated smart contract processes them. During contract execution, if a contract generates additional internal messages to other contracts, these bypass the mempool and are instead routed directly to their target contracts on either the same or another shard. This direct routing avoids unnecessary queuing, creating a more efficient and streamlined message processing flow across execution shards.

zkSharding deploys *enshrined token design* [?] in execution shards which offers efficient approach to managing token transaction across execution shards. With enshrined tokens, the core token functions (like transfer, balance checking, and approvals) are directly built into the core protocol of execution shards rather than being implemented through smart contracts. This allows these operations to benefit from protocol-level optimizations. In this way, the protocol itself handles the complexity of moving tokens between execution shards.

Execution shards communicate through a *cross-shard communication protocol.* It guarantees that messages between shards are eventually executed. In this protocol, even execution shards which are not message's destination play a *supporting* role by storing message-related data, thereby helping to enforce eventual execution on the destination shard. To enable efficient communication between execution shards, zkSharding allow smart contracts deployed on different execution shards to interact without halting execution. The `asyncCall` function [?] is integral to this feature. It enables a contract on one shard to call functions on contracts located on other shards directly, without waiting for an immediate response. This mechanism produces a message that is processed by the destination shard asynchronously, allowing parallelism across shards and improving network scalability.

A unique data structure of our execution shards is the **ShardDAG** [?]. It is a structure that connects blocks from different execution shards as well as blocks from the main shard. This shared structure imposes a global ordering of transactions that mitigates Maximum Extractable Value (MEV) attacks [?, ?] especially for cross-shard transactions and guarantees ultimate cross-shard transaction processing.

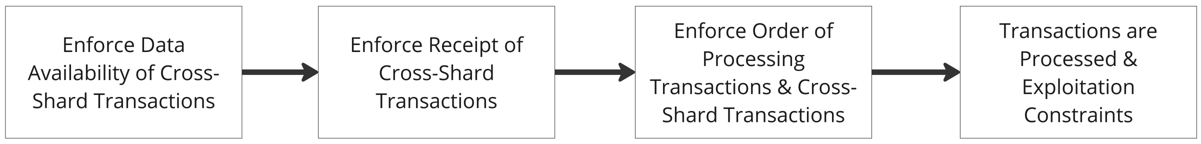
Enforce Data Availability of Cross-Shard Transactions
Enforce Receipt of Cross-Shard Transactions
Enforce Order of Processing Transactions & Cross-Shard Transactions
Transactions are Processed & Exploitation Constraints

Figure 3: *The ShardDAG rules enforce a strategy that ensures secure cross-shard transactions: (1)* ***Child Condition*** *enforces data availability by requiring broadcast to more than F shards; (2)* ***Parent Condition*** *guarantees receipt by integrating data from multiple shards; and (3)* ***Main-Parent Condition*** *maintains order with the main shard. Together, these rules ensure secure processing and reduce exploitation risks.*

ShardDAG: It is inspired by DAG-based blockchains [?, ?], and operates as follows: When a validator generates a block B for execution shard $\mathbb{S}_i$, in addition to the transactions, it adds the following elements to connect B to other blocks to form the DAG:

- The hash of the latest block from the same shard $\mathbb{S}_i$, as in a typical blockchain.
- Hashes of blocks from other shards, following the ShardDAG rules [?], which act as acknowledgments that the execution shard has received data from those other shards.
- The hash of the latest main shard block to ensure that cross-shard transactions are eventually processed and recognized by the entire network.

The ShardDAG enforces rules that are designed to maintain security, data availability, and orderly processing of transactions. The key rules include :

- **Child Condition:** An execution shard block is not finalized until it has received acknowledgments from over F other shards, where F represents the system's tolerance for potentially adversarial shards. This condition helps ensure cross-shard data is sufficiently distributed, preventing single-shard control over transaction flow and supporting broad data availability.
- **Parent Condition:** An execution shard block must have a subgraph that includes blocks from more than F other shards relative to its predecessor. This condition encourages regular integration of cross-shard data, reducing the risk of shards bypassing or isolating cross-shard transactions.
- **Main-Parent Condition :** A shard block should not reference the same consensus block for more than X consecutive blocks, where X depends on the execution shard's block time. This helps shards stay aligned with updates from the main shard. This promotes synchronization across the network.

See Figure 3 for a detailed illustration of how these rules enforce constraints for succesfull and orderly execution of transactions.

3 Transaction Lifecycle: From Execution to L1 Finality

In this section, we describe the lifecycle of a transaction in zkSharding, while highlighting specific solutions introduced to ensure both security and efficiency.

In zkSharding, there are two fundamantal messages: external and internal . External messages originate from external actors and are sent directly to the mempool of the contract's execution shard for processing. Internal messages, on the other hand, are generated during contract execution and do not enter the mempool, as we explained in Section 2.2.2. We differenciate internal messages, for the sake of clarity in this section, as intra-shard transactions (ISTs) and cross-shard transactions (CSTs) although ISTs and CSTs share the same structure and follow the similar execution path in zkSharding.

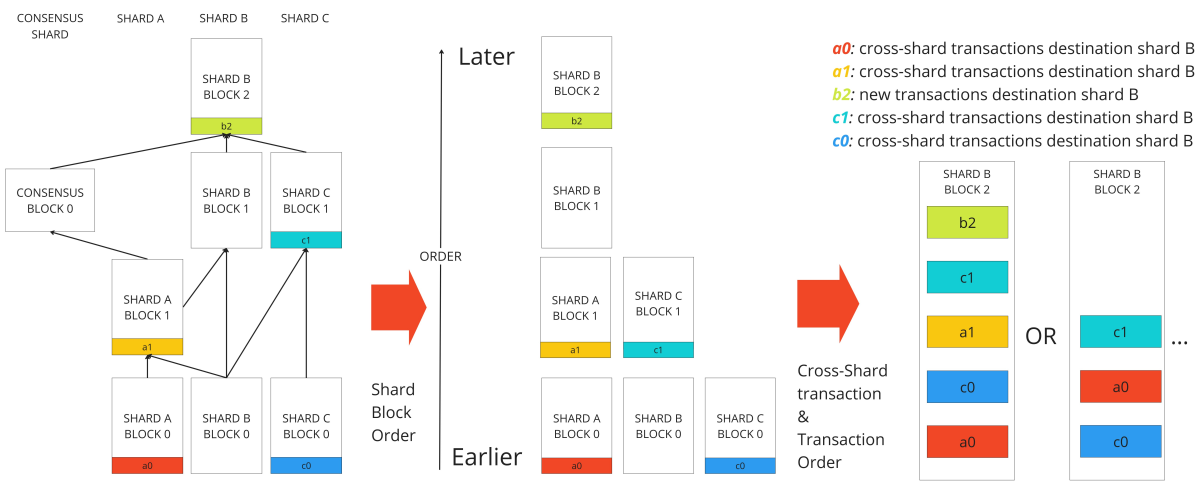
CONSENSUS SHARD
SHARD A
SHARD B
SHARD C
SHARD B BLOCK 2
b2
CONSENSUS BLOCK 0
SHARD B BLOCK 1
SHARD C BLOCK 1
c1
SHARD A BLOCK 1
a1
SHARD A BLOCK 0
a0
SHARD B BLOCK 0
SHARD C BLOCK 0
c0
Later
ORDER
Earlier
Shard Block Order
Cross-Shard transaction & Transaction Order
a0: cross-shard transactions destination shard B
a1: cross-shard transactions destination shard B
b2: new transactions destination shard B
c1: cross-shard transactions destination shard B
c0: cross-shard transactions destination shard B
OR
...

Figure 4: *ShardDAG Order*

Each internal message contains several fields that help process it robustly, but the key distinction between ISTs and CSTs lies in two specific fields: the sender contract address S and the recipient contract address $\mathtt{to}$. To differentiate between shards, we use superscripts to denote which shard the accounts belong to, e.g., S^i and $\mathtt{to}^i$ indicate that the sender or recipient account belongs to execution shard $\mathbb{S}_i$. For a given transaction $\mathtt{tx} = (S^i, \mathtt{to}^j)$, if $i = j$, meaning both the sender and recipient belong to the same shard, we classify it as an *intra-shard transaction (IST)*. If $i \neq j$, where the sender and recipient belong to different shards, we classify it as a *cross-shard transaction (CST)*. In short:

- ISTs are processed entirely in the execution shard where they are initiated.
- CSTs involve interaction between different shards and require cross-shard coordination.

Once either an internal or external message arrives to the execution shard, it waits to be included in a block by a block producer, who connects it to the blockchain, making it a canonical part of the execution shard. Below, we describe this process in the context of a block producer of $\mathbb{S}_i$.

3.1 Reaching Local Consensus

- **Transaction ordering:**. The block producer begins by applying the ShardDAG ordering rules (see Figure 4). For this, it first analyzes the local shardDAG subgraph for shard $\mathbb{S}_i$. This subgraph contains unprocessed transactions $\{(S^j, \mathtt{to}^i)\}$ where $j = i$ or $j \neq i$ and their dependencies across different execution shards. The block producer then determines the order of these transactions with the help of the subgraph. In case additional block space is available, it also selects a set of transactions from the mempool. In the end, it obtains the list of transactions $\mathcal{T}$ to be processed. The ShardDAG ordering rules enforce a *parent-child relationship* between transactions to be processed in a valid order. Specifically, CSTs with earlier dependencies must be processed before later transactions. This ordering prevents inconsistencies in cross-shard interactions.
- **Block Creation:** The proposer executes all transactions in $\mathcal{T}$ in the context of the latest state using the state transition function. If these executions result in new cross-shard transactions (CSTs), such as $(S^i, \mathtt{to}^j)$ where $i \neq j$, the proposer adds them to a special data structure called the *outbox* $\mathcal{O}_i$. $\mathcal{O}_i$ tracks all CSTs originating from $\mathbb{S}_i$ that are waiting to be processed by their respective destination shards[1].

[1]If the block capacity is full and the mempool still contains unprocessed messages that should be included based on the ShardDAG order, they can be added to the block's outbox for future inclusion in later blocks [?]

After executing the transactions, the block producer creates a block B. We note that if the execution of some CSTs and ISTs fails, they enter a refund mechanism that allows failed transactions to be retried with additional fees or canceled via the mailbox. When bulding B, the block producer respects to the Parent and Main-Parent condtions of ShardDAG rules. Therefore, it retrieves any new outgoing CSTs from other execution shards and includes the hashes of the originating blocks in B, thereby linking B to the corresponding blocks from other execution shards. B includes the list of executed transactions $\mathcal{T}$, the updated state st, state root s of st, the outbox $\mathcal{O}_i$, which contains CSTs that are yet to be processed by other shards and block header bh. This newly created block is then proposed to the network as part of the local consensus process.

- **Local Consensus:** After receiving the block B, the validators initiate the Multi-Threshold BFT consensus mechanism [?] to validate and finalize the block *locally* within execution shard $\mathbb{S}_i$. Each validator verifies the block's correctness by [2]
  - verifying the correctness of the transaction order in B to ensure that the order of transactions adheres to the rules of the shardDAG,
  - verifying if the block follows the parent condition and main-parent condition introduced by ShardDAG rules and,
  - checking if $\mathsf{F}(\mathsf{st}', \mathcal{T}) \to \mathsf{st}$ where st' is the last locally finalized state.

  Once a supermajority of validators agrees on the block's validity, the block is finalized locally. This local consensus ensures that the block is securely added to $\mathbb{S}_i$'s chain while maintaining consistency with the shardDAG ordering rules.
- **Block Propogation:** After B is finalized locally, validators are responsible for distributing $\mathcal{O}_i$ and bh to the destination shards $\mathbb{S}_j$, as well as to other shards to ensure data availability of CSTs. They are incentivized to propagate the data to help reaching global consensus at the main shard level, as required by the child condition. Remember that execution shards that receive the CST data link their next block to bh. Even shards that do not process the CSTs must store the data off-chain, as they may be involved in forming DAG edges or verifying the ShardDAG rules, which is essential for their own block finality.

Validator sends bh to the main shard after finalizing it locally.

3.2 Reaching the Global Consensus

When the main shard validators receive a block header bh from an execution shard, they perform the following checks before including bh in a main shard block:

- **Local Finality Check:** Confirm that bh has been signed by a supermajority of validators, indicating that it has achieved local finality.
- **Validation of ShardDAG Rules:** Verify that the *Parent Condition*, *Child Condition*, and *Main-Parent Condition* are all satisfied.

If all checks pass, bh is included in a main shard block and finalized.

3.3 Reaching the L1 Finality

With the transactions now executed in an execution shard block and included in the main shard, the next step involves the synchronization committee $\mathcal{SC}$ to extract the transaction data, coordinate with provers to generate proofs, and ultimately submit the verified data to L1 for final settlement and verification. Here is how $\mathcal{SC}$ executes the process:

[2] The list of checks is not exhaustive. We give the critical ones ensuring the safety and security of cross-shard communication.

- The **observer** in $\mathcal{SC}$ monitors the growth of the main shard's state and execution shard states. Once the state changes reach a certain threshold, the observer initiates the process of preparing data for L1 submission.
- When the observer signals that a snapshot of data between time T (just after the last proven state) and $T+n$ is ready, the **aggregator** in $\mathcal{SC}$ compiles the data executed between T and $T+n$ in *all* shards into a batch. Each batch of $\mathbb{S}_i$ consists of transactions $\mathcal{T}_i$ executed between time T and $T+n$. In the end, aggregator composes $k+1$ bathches where k is the number of execution shard into one batch $\mathcal{B}$. The aggregator sends $\mathcal{B}$, the last verified state on L1 st', and other necessary data to the verifier in $\mathcal{SC}$. It also forwards $\mathcal{B}$ to the proposer in $\mathcal{SC}$. By grouping all transactions into a batch, we achieve better cost-efficiency during the proving process.
- The **verifier** generates a special transaction called *proof order* to outsource the proof generation process to the prover network. The proof order specifies the batch and state updates that need to be proven, along with a payment for generating the proof. Prover network runs the proving algorithm which receives as a private input $\mathcal{B}$, current state st' and as a public input the new state root s and previous verified state root s'. In simple terms, provers generate a succinct proof π that verifies the correctness of the new state with the public inputs (See Section 1.1 to understand the proving statement).
  After receiving π, the verifier gives π to the proposer if π is verified. The provers joining the proving process get their fee.
- The **Proposer** runs the compression algorithm for $\mathcal{B}$ which is an optimized algorithm for zkSharding's specific data types (e.g., block headers, state roots, and transaction summaries). This reduces the size of the data submitted to L1. Once the batch is compressed, the proposer submits it to L1 through an EIP-4844 blob transaction. This transaction includes the compressed batch data. The blob transaction ensures that the data is permanently stored and its KZG commitment Com_{da} is accessible for future verification. Addionally, the proposer submits π and public inputs to the L1 state validity contract and the main shard.

Once the L1 State Validity Contract verifies π as described in Section 1.1, it updates the verified state root to s and finalizes it as part of L1 through Casper FFG. It means that once a state is finalized through Casper FFG, it is irreversible and considered as a canonical part of Ethereum and also zkSharding.

4 Proof Sytem in zkSharding

Our zkEVM [?] operates at the bytecode level by directly interpreting EVM bytecode. It ensures high compatibility with existing Ethereum dApps and smart contracts, although it may produce slightly different state roots than the standard EVM due to the use of SNARK-friendly optimizations. Projects like Scroll [?] and Hermez [?] by Polygon also use this method.

In our zkEVM, we use FRI-based placeholder proof system [?] which uses lookup argument [?]. During the implementation of Plookup and its practical use, we encountered some technical issues that were not mentioned in the original solution. Therefore, we propose practical improvements [?] for writing large PLONK circuits with a complex logic.

zkSharding's zkEVM consists of multiple subcircuits , each of which is handled by separate provers. A straightforward approach would be to have each prover P_i generate an independent FRI-based proof π_i for each subcircuit $\mathcal{C}_i$, resulting in M proofs (where M is the number of subcircuits). These proofs would then need to be aggregated into a succinct proof π. However, FRI lacks homomorphic properties, making this aggregation process computationally expensive, which contradicts FRI's main advantage of prover efficiency.

To address this, we designed Distributed FRI (DFRI) [?], which uses FRI batching techniques to enable efficient proof aggregation across multiple provers. In DFRI, the committed polynomials from each prover are combined using a collaboratively generated random challenge. This batching mechanism allows us to aggregate the proofs more efficiently while maintaining the security and integrity of the proof system. DFRI also ensures accountability by making dishonest behavior from any prover detectable, which is crucial for maintaining liveness in distributed systems like zkSharding.

From a performance perspective, DFRI maintains the same proof size as a single FRI proof, as compared to the straightforward approach which would involve having M proofs without infeasable aggregation layer. This is achieved through coordination between provers during the proving process by slightly increasing the communication overhead. Additonally, DFRI does not extend the overall proving time comparing to the single FRI-based prover. Overall, DFRI helps retain the efficiency of FRI while enabling secure, distributed proof generation, making it a critical innovation for zkSharding's scalability and security.

5 Local Fee Model

In zkSharding, gas price calculation follows a Local Fee Model at the shard level[3]. Each shard maintains its own distinct base fee, designed to regulate gas demand and promote balanced load distribution across the network.

5.1 zkSharding Transaction Fee Mechanism

The zkSharding transaction fee mechanism consists of several key components:

- **Shard-Specific Base Fees**: Each shard operates with its own independent base fee, enabling granular control over gas demand and encouraging an even workload across the network.
- **Adjustment Mechanism**: The model incorporates a modified EIP-1559 base fee adjustment mechanism designed for quicker adjustments during periods of high congestion, enabling shards to dynamically adapt their base fees to prevailing network conditions. In addition to faster responsiveness, the mechanism allows greater flexibility around the target gas usage, aiming to enhance the predictability of cross-shard transactions and transaction fees when the system is in a balanced state.
- **Cross-shard Transaction Premium**: A "take it or leave it" premium is applied to manage congestion in cross-shard transactions (CSTs). This mandatory fee ensures their viability by incentivizing validators to prioritize CST inclusion and directly rewarding them for doing so.
- **Cross-shard Transaction Base Fee Discount**: Cross-shard transactions are charged a discounted shard base fee to ensure cost efficiency for users. The goal is to align the cost of CSTs with that of intra-shard transactions, creating a consistent and predictable fee structure across the network without disproportionately penalizing cross-shard activity.
- **Limited Transaction Gas Space**: The available gas for transactions within a block is dynamically adjusted based on the amount of gas used for cross-shard transactions in the previous block. This design incentivizes validators to prioritize cross-shard transactions, as doing so increases their potential tip and message premium rewards.
- **EMA-Based Smoothing**: To address volatility in L1 blob base fees, the model uses an Exponential Moving Average (EMA). This smooths out fee spikes by distributing their impact over time, preventing sudden cost increases for users.
- **Transformation of L1 Fees into Gas**: L1 fees, such as those for data availability and proof verification, are converted into gas units to maintain compatibility with Ethereum's

[3]ETH is used as a payment token for =nil;

existing tools and frameworks. These gas units are excluded from the base fee adjustment mechanism, ensuring a clear separation between L1 fee and L2 fee dynamics.

6 Conclusion

zkSharding presents a robust solution to Ethereum's scalability challenges by leveraging sharding and SNARK-based proofs. By dividing the network into execution shards and synchronizing their state via the main shard, zkSharding ensures efficient, decentralized processing without fragmenting liquidity. Ethereum serves as the settlement and data availability layer, further enhancing the system's security. Through techniques like DFRI and a multi-layer consensus model, zkSharding achieves a balance between scalability, security, and decentralization, making it a promising framework for Ethereum's future.