=nil;'s zkSharding for Ethereum

Mikhail Komarov
=nil; Foundation
nemo@nil.foundation

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

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

November 7, 2023

Abstract

1 Introduction

The imperative to scale Ethereum is driven by its growing user base and the increasing complexity of applications built on its platform. 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 several drawbacks:

- **Fragmentation**: Rollups are separated from each other by design. This leads to fragmentation in terms of security, liquidity, and data consistency.
- **Updates in zkEVMs**: The dynamic nature of zkEVMs brings regular updates that may lead to potential security vulnerabilities.
- **Applications Redeployments**: Users have to redeploy their applications from Ethereum to L2, which in the same time causes liquidity fragmentation which leads us to the first point.

In addressing Ethereum's scalability and fragmentation challenges, we propose a new layer-2 (L2) concept, *zkSharding*. This solution merges Zero-Knowledge proofs with a sharding mechanism and blends a bunch of other =nil; technologies into it.

Key aspects include:

- **"zkRollup with Sharding" for Horizontal Scalability**: The core of zkSharding is a blend of zkRollups and sharding, enabling extensive horizontal scalability without compromising security or reducing efficiency. This approach counters the limitations of vertical scaling (L3, L4, etc.), significantly reducing data and liquidity fragmentation.
- **Direct Ethereum data access**: The ability to call Ethereum's original data from L2 applications allows us to reuse already deployed applications. Direct access to L1 data from L2 ensures a more unified and seamless environment.
- **Type-1 zkEVM Compiled via zkLLVM from in-Production EVM**: zkEVM's, being often implemented from scratch to replicate the actual state transition executor logic (EVM) pose security risks as the circuit is very large, very complicated and hardly auditable. zkSharding relies on a zkEVM circuit compiled by zkLLVM circuit compiler from a production-grade EVM without manual circuit reimplementation which reduces security risks.

We envision zkSharding as a step forward in blockchain modularity, aligning with Ethereum's principles and offering a scalable, integrated solution.

=nil; zkSharding solution provides the follwing properties:

1. **Scalability**:
   (a) No scalability limitations as the execution is parallel. Throughput around 60k transfers per second with around 400 nodes.
   (b) Competitive marketplace-based proof generation guarantees fastest L1-finality and chepest generation costs.

2. **Unified Liquidity/Security**:
   (a) No security/liquidity fragmentation as each shard is a part of the wholistic cluster.
   (b) Reduction of a need to migrate liquidity from Ethereum as `=nil;` provides transparent access to its' data for applications.
3. **Security**:
   (a) State transitions secured by `=nil;`'s zkEVM compiled via zkLLVM. It provides auditable security (e.g. constraints security) as the code is easily inspectable since zkEVM circuits are compiled from a production-used EVM implementation in high-level language and not written manually.
   (b) Liveness security guaranteed with native staking or restaking existing staking pools (e.g. Lido) TVL.
   (c) Decentralized from day one thanks to combination of Ethereum staking and Proof Market. See Section 2.5 for details.
4. **Functionality**:
   (a) A Type-1 zkEVM[1], fully EVM bytecode-equivelent.
   (b) An environment tailored for applications that have high demands related to time, memory, and algorithmic complexity. Examples include Decentralized Exchanges (DEXes) or Proof Market[2].

1.1 Overview

As illustrated in Figure 1, the state of `=nil;` zkSharding is partitioned into the main shard and several secondary shards. The main shard's role is to synchronize and consolidate data from the secondary shards. It uses Ethereum both as its Data Availability Layer and as a verifier for state transition proofs, similar to typical zkRollups operations. For a comprehensive understanding of the sharding approach, refer to Section 3.1.

Secondary 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 main shard using zkEVM (see 3.3.1 for a detailed explanation).

Users have direct access to Ethereum data, thanks to the Ethereum Data Provider linked to every shard. This data is validated using zkBridge.

To illustrate the transaction flow from initiation by a user to confirmation on Ethereum, consider the following steps:

1. The user signs a transaction tx and dispatches it to the network.
2. Validators in shard S, where the user's wallet is located, place tx into the mempool.
3. These validators then create a new block B_S^1.
4. The hash of B_S^1 is recorded on the main shard within block B_M^1.
5. A state transition proof for B_S^1 is produced and verified by the main shard in block B_M^2.
6. A state transition proof for B_M^2 is sent to Ethereum for verification and coupled with the necessary data for ensuring data availability.
7. Once this process is complete, tx achieves confirmation by Ethereum.

This outline assumes that the user's transaction does not activate the cross-shard messaging protocol. However, in this case the transaction flow remains the same with a difference that user's transaction triggers a creation of new transactions on other shards.

[1] https://vitalik.ca/general/2022/08/04/zkevm.html
[2] proof.market

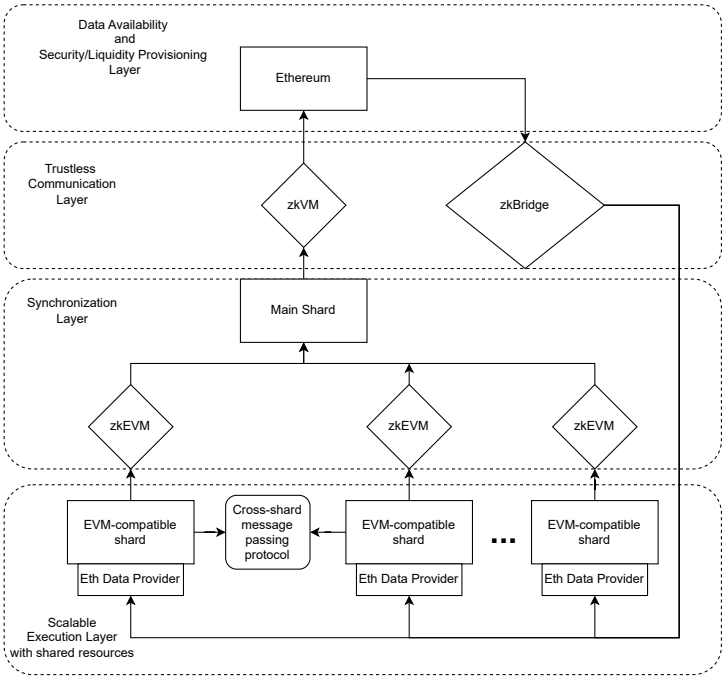

Figure 1: *=nil; zkSharding overview*

2 Foundations for =nil; zkSharding

This section describes the precursor set of technologies leading to the product description in Section 3.

2.1 zkLLVM

zkLLVM ([1]) is a circuit compiler designed to translate high-level mainstream languages, such as C++ and Rust, into representations suitable for provable computation protocols, namely circuits for proof systems.

The architecture of zkLLVM offers distinct advantages over other methods of circuit development:

1. It allows users to directly compile their algorithm, without needing a custom Domain Specific Language (DSL) and duplicating source code.
2. It omits any intermediary layer, such as a specific zkVM, between the original algorithm and the resulting circuits. This absence translates to no additional overhead in the circuit size (and consequently, the proving time).
3. Due to direct access to the inner circuit representation, zkLLVM facilitates the generation of optimized low-level verifier code tailored for specific virtual machines. For example, =nil; utilizes the zkLLVM transpiler[3] for the EVM Placeholder verifier.
4. As an LLVM-based compiler, zkLLVM boasts compatibility with any LLVM IR-based extension.

zkLLVM assumes a crucial role in the novel zkEVM construction approach, detailed further in Section 3.3.1.

2.2 Placeholder Proof System

Initiated in 2021, Placeholder [2] is a modular IVC (Incrementally Verifiable Computation) that relies on a PLONK-inspired arithmetization-based proof system. Its inherent modularity provides the flexibility for various adjustments based on the specific use case:

- **Underlying Fields**: For algorithms that do not involve cryptographic operations, there is often no requirement for 256-bit fields. To optimize both the proving and verification processes, Placeholder can be adapted to work with 64-bit fields.

[3] https://github.com/NilFoundation/zkllvm-transpiler

- **Commitment Schemes**: The choice between commitment schemes, such as those based on hashes or bilinear pairings, allows for trade-offs between the necessity for a trusted setup and associated verification expenses.
- **Lookup Techniques**: Different techniques, like Plookup or Baloo, can be employed depending on the size of the lookup table and the frequency of lookup requests, thus reducing the lookup overhead.
- **Gate Generation Techniques**: Efficient IVC is achieved using an approach similar to Protostar for gate definition. Nonetheless, this might not always be the optimal choice, especially when there is no need for recursive proof verification.

Such modularity enables Placeholder to reduce the confirmation times on the L1-layer and speed up zkBridge data provision. This is achieved through faster IVC-based zkVM and Ethereum consensus proof generation.

2.3 zkBridge

zkBridge is a concept that emerged in early 2021 from collaborative efforts between the `=nil;` Foundation, Ethereum Foundation, and Mina Foundation [3]. Later in the same year, the Solana Foundation joined this initiative [4].

This concept materialized into a series of trustless bridging projects amongst the Mina, Ethereum, and Solana networks.

The data access approach detailed in Section 3.4 harnesses the zkBridge concept to minimize trust assumptions.

2.4 Crypto3

Crypto3[4] is a modular cryptographic suite developed in C++17. It is designed to support the research and development of innovative primitives and protocols by creating new constructions or combining existing ones. The suite includes a variety of constructions, such as basic algebraic structures (like finite fields and elliptic curves) and cryptographic primitives (for instance, symmetric and asymmetric encryption, threshold signatures, and algebraic hash functions).

Additionally, Crypto3 serves as a C++ Software Development Kit (SDK) for zkApp development, making it easier to use zkLLVM. For example, zkBridge solution developed with `=nil;` technologies utilizes Crypto3 as the SDK for zkLLVM.

2.5 Proof Market

Proof Market[5] is a decentralized platform. It is designed to outsource zkProof generation tasks. The foundational premise behind Proof Market is straightforward. It aims to aggregate the hardware-intensive demand for proof generation from numerous protocols. Then, it reallocates these tasks to independent entities. These entities include professional hardware operators, equipment owners, and circuit/hardware designers.

This arrangement fosters a dynamic marketplace. In this marketplace, hardware operators can directly cater to the needs of proof consumers. Proof Market can maintain a consistent and robust demand due to its substantial user base. This sustained demand draws in large-scale hardware operators. They are keen on maximizing the utilization of their resources.

Proof Market presents a competitive landscape for proof producers. This environment compels them to relentlessly pursue the optimization of proof generation overheads. Several strategies can achieve this. Producers might scale up their infrastructure. This scaling can benefit from economies of scale. Alternatively, they can refine their hardware for optimal proof generation efficiency. A producer who offers competitive pricing and rapid proof generation can dominate the market. This dominance incentivizes other operators to enhance their competitive offerings.

By leveraging Proof Market for proof generation, `=nil;` zkSharding achieves a significant advantage. It can ensure decentralized proof generation from the day one. At the same time, it distinguishes between the roles of validators and proof generators.

[4] https://github.com/NilFoundation/crypto3
[5] proof.market

3 Proposal

3.1 Dynamic Sharding

Considerations concerning application-level Ethereum sharding (via L2s, L3s, L4s, etc.) gave rise to the concept of protocol modularity, encapsulated by the principle: one application, one database. Initially, this concept was perceived to possess the potential to augment the blockspace of Ethereum for specific applications. However, concurrent with its potential advantages, it also introduced several challenges:

1. **Fragmented Liquidity:** Undermines the efficiency and viability of applications. The currency presented on one application cannot be used with another without bridging.
2. **Security:** Compromised security due to fragmentation, as inter-application interactions now necessitate reliance on various bridges. Although zkBridges offer a robust solution, they are not feasible for hundreds of distinct applications.
3. **Efficiency:** Decreased efficiency in cross-application interactions, given that each bridge interaction entails additional fees.
4. **Reduced Throughput:** As interactions between disparate applications demand the use of bridges. These bridges introduce added latency and affect the overall transactions per second ratio.

As detailed in subsequent sections, the sharding approach addresses these challenges. Each shard interacts with others within the framework of a unified protocol, ensuring seamless data access without the need for dedicated bridges between shards.

`=nil;` introduces a notion of protocol-level enshrined sharding for byzantine fault-tolerant environment-deployed distributed databases (in particular for Ethereum).

In the subsequent sections, we elaborate the foundational concepts and components of `=nil;` zkSharding.

3.1.1 Primary Shard

The first shard, known as the "primary shard," holds essential data about the protocol's consensus and its current parameters. It also contains data about other shards, and the hashes of the most recent replication packets (blocks) from all shards. In essence, the main 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.

3.1.2 Shards

First, we define a table (account) as 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.

Secondary shards handle user transaction processing. Each of these shards manages a subset of tables (accounts), defined by the deterministic function $F_S : (pk, M_{shards}) \to id_{shard}$. In this context, pk represents the account's public key, while M_{shards} is shards history and metadata stored in the main shard.

Every shard is maintained by a specific group of validators (*committee*). These validators run a "local" consensus algorithm to ensure consistency in the shard's state.

In accordance with parameters outlined in the main shard, each individual shard has a maximum replication packet capacity. 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, then the shard is split into two shards. 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.

3.1.3 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.

Additionally, it's possible to provide a fraud proof for not relaying a cross-shard transaction. As a result, validators who ignore this responsibility can be slashed.

3.1.4 Colocation

Cross-shard communication might extend the time needed to achieve outcomes reliant on a range of applications located on different shards. For scenarios demanding the swiftest possible transaction processing (aka increased consistency), the protocol incorporates a colocation technique.

Colocation 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}.

3.2 `=EVM++;`

`=EVM++;` is an EVM-based virtual machine, enhanced for more efficient execution of general-purpose algorithms. It's tailored for optimized state transition proof generation through zkLLVM.

The design of the virtual machine encompasses three core properties:

- Binary compatibility with original EVM applications. This means that any EVM applications input into `=EVM++;` produces an output state identical to one executed on an EVM machine.
- Execution capability for general-purpose C++/Rust code. This allows to build applications that usually cannot be deployed on blockchain, such as shared sequencers.
- Efficient state transition proof generation.

To achieve the first property, `=EVM++;` extends the basic EVM architecture and its implementation. For example, the opcodes in `=EVM++;` form a superset of those in EVM.

Efficiency in general-purpose C++ code execution is realized via:

1. Opcodes for reading and writing memory blocks smaller than 256 bits, coupled with the introduction of new integer types. This adjustment reduces the overhead tied to using exclusively 256-bit words in general-purpose programs.
2. Reducing certain EVM stack limitations:
   - Infinite stack depth, with a free stack depth limit set at 1024 elements. When this limit is exceeded, an additional 1024-element memory chunk is added to the stack. Every subsequent stack chunk demands more gas than its predecessor.
   - New commands to access elements from any stack depth.
3. The introduction of new control flow commands that support relative offsets and function calls, similar to EIP-615[6].
4. Program memory is divided into heap and stack, governed by the EVM applications's `stack_ratio` parameter. For pure EVM applications, `stack_ratio` = 100, meaning they only utilize the stack.

The third property is achieved through synchronizing the opcode sets of `=EVM++;` with zkLLVM.

3.3 Security Mechanisms: zkRollup with Ethereum Stake and Data Availability

3.3.1 zkEVM via zkLLVM: Type-1 Secure, Auditable and Performant zkEVM

As highlighted in Section 1, `=nil;` zkSharding derives its security from the Ethereum Network by using state transition zero-knowledge proofs, similar to zkRollups. For this to work, every shard must submit

[6] https://eips.ethereum.org/EIPS/eip-615

their updates, along with their proofs, to the main shard. Afterwards, the main shard confirms its state transition to Ethereum.

To achieve this objective, two essential components are required:

- An efficient zkVM for `=EVM++;`, constructed using zkLLVM. Given that `=EVM++;` is an extension of EVM, `=nil;`'s zkVM can be viewed as a type-1 zkEVM.
- Proof aggregation based on Placeholder's IVC. This allows the splitting of the complex zkVM circuit into multiple proofs, which can be computed in parallel.

The general structure of `=nil;`'s zkEVM is not much different from other zkEVM implementations. It includes two types of proofs: the State Proof and the EVM Proof. The State Proof ensures that operations are performed correctly. However, it doesn't validate the exact location of the written or read data. The EVM Proof, on the other hand, confirms that the State Proof selected the correct data location.

However, earlier versions of zkEVM have common limitations:

- Circuit definition is resource-intensive: Creating a circuit takes a lot of time (even for experienced engineers) and significant resources.
- Circuits can get complex: With the evolution of rollups, new features are introduced, leading to changes in existing circuits. This added complexity increases both risks and costs.
- Manually crafted circuits may have security issues: Such circuits depend on human input and often use new, not fully tested libraries.
- Circuit audits are challenging: The complexity and difficulty of auditing circuits means audits remain both difficult and costly.

Furthermore, these limitations are not just a "one-time setup" challenge. They recur with every EVM update.

`=nil;`'s zkEVM, built on zkLLVM, removes human intervention from the circuit definition process. This solves the previously mentioned challenges.

3.3.2 Data Availability

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

The choice of DA varies between the main shard and the secondary shards:

- The main shard employs Ethereum as its DA.
- Secondary shards have the option to use Ethereum or opt not to have a distinct DA.

This arrangement is established by launching two kinds of shards at the start: those with a separate DA and those without. In subsequent phases, only shards of the same DA category can be merged. This means that during its creation, each account must be mapped to a specific DA category.

Additionally, this framework can be expanded to include other types of DA.

Ethereum as DA layer. The comprehensive way to store a sufficient amount of data for restoring the L2 state is keeping it as transaction `calldata` on Ethereum. That leads to the benefits of lower prices for storage because of its lack of persistence. It also allows keeping the shards' data archived while preserving some metadata for more straightforward access and search. It can be considered the finalized shard's state when the distinct transaction with the shard's state has been finalized on Ethereum.

The overall structure of the DA transaction can be separated into two parts – payload for the restoration process and metadata that is used for identifying the distinct shard, signature, and details of the payload. That is considered that the payload can be archived as direct access to it in general cases is not required.

Shard state can be represented with the following structure:

```
struct ShardDescription {
    uint32_t shardID;
    uint32_t forkID;
    uint8_t rev_shard[32];
    uint8_t signature[32];
    uint256_t epoch;
    std::pair<uint32_t, uint32_t> epoch_part;
```

```
    uint256_t payload_size;
    uint8_t payload_hash[32];
    uint8_t payload[];
};
```

The main shard commits to L1 each epoch, which we consider could be around 5 minutes. The number of blocks per second on the shard will equal one. It leads to the estimation of 300 blocks that will be validated per one Ethereum epoch. Each sub-shard will try to keep its state updated on the main shard, where we imply the number of transactions in the block will equal the number of shards running at the same time.

To restore the main shard state it's sufficient to keep only sub-shards' transition states for each epoch. That leads to the statement of the sufficiency of commiting to L1 only the difference between transitions from the previous main shard epoch to the current.

For further estimations, it's supposed that the number of shards running simultaneously is equal to 100. The expected size of each transition difference is around 96 bytes (state root, aggregated signature, transaction hash, other values are small enough to ignore them in this analysis), while the empty transaction size is around 109 bytes. The total expected size for one commitment of the main shard to the L1 network is

$$100 \cdot (109 + 96) = 20500 \texttt{ bytes}$$

which is 328000 gas (upper estimation). That is the approximate amount of data that will be transferred to the L1 network by the main shard. That data will be stored in an archived way, leading to a smaller size. As well for more precise estimation, that must be summed up with a size of proof and some metadata for the L1 transaction. That is expected the size of the proof could reach 200 kilobytes. That will lead to $200000 + 20500 = 220500$ bytes or 3528000 gas (upper estimation).

3.4 Ethereum Data Access

Direct access to Ethereum data is based on the Data Provider module within the protocol's node. The Data Provider operates independently from the shard's data storage and synchronizes its information with an external database. The most recent state of this database must receive validation from the confirmation module. As an Ethereum confirmation module, we use zkBridge with Ethereum consensus proof.

The fingerprint of the last monitored database state (represented by Ethereum's block hash) is added into the shard's block. This is done with a confirmation that can be authenticated using `=EVM++;`.

3.4.1 Calls to Ethereum Application

From a developer's perspective, accessing Ethereum data does not differ from any other calls within the original Ethereum network. This is achieved by making a distinction between the original Ethereum addresses and `=nil;` zkSharding addresses at the `=EVM++;` level. Each `=EVM++;` instruction that works with the `address` type checks whether the address belongs to the Ethereum Network. If it does, the `=EVM++;` retrieves data from the Data Provider module instead of the original database.

This means that the source code for original Ethereum applications and `=nil;` zkSharding applications does not differ:

```
// Imagine that the applications Caller does not have the source code for the
// applications Receiver, but it knows the address of applications Receiver
// and the function to call.
function testCallFoo(address _addr) public view {
    // You can send ether and specify a custom gas amount
    (bool success, bytes memory data) = _addr.call(
        abi.encodeWithSignature("foo(string,uint256)", "call foo", 123)
    );

    emit Response(success, data);
}
```

Note that `foo` must be a `view` or `pure` function. If it is not, the transaction will be reverted.
The same approach can be applied to other external databases and networks.

References

[1] N. Kaskov and M. Komarov, "zkllvm circuit compiler," 2023.

[2] A. Cherniaeva, I. Shirobokov, and M. Komarov, "Placeholder proof system," 2023.

[3] A. Cherniaeva, I. Shirobokov, and M. Komarov, "In-evm mina state verification," April 2021.

[4] A. Cherniaeva, I. Shirobokov, and M. Komarov, "In-evm solana light-client state verification," May 2021.

[5] M. Komarov, "`=nil; ÒROP DATABASE *`," 2023.

[6] A. Sofronov, M. Komarov, and I. Shirobokov, "`=nil; DROP DATABASE *` deployment sustainability model," October 2023.

[7] I. Shirobokov and V. Kuznetsov, "Proof market general description," October 2022.

[8] D. Kales, F. Nieddu, and R. Walch, "`=nil;` foundation zkml investigation report," 2023.

[9] E. Shapiro, I. Meckler, and J. Drake, "Full verification of the mina blockchain on evm," March 2021.

[10] I. Shirobokov, "=nil; staking market," 2023.

[11] I. Marozau, "=nil; staking market: Recipient network," 2023.