TL;DR

- =nil; is a sharded zkRollup - a new L2 concept for dynamic and secure scaling of Ethereum through a protocol-level parallel transaction execution across shards.
- Equipped with zkSharding, =nil; offers horizontal scaling without compromising the benefits of a single execution layer, namely unified liquidity and economic security.
- =nil; provides applications full composability with Ethereum through transparent and verifiable Ethereum data access.
- =nil; introduces a Type-1 zkEVM compiled with zkLLVM.
- Fast proof generation is guaranteed by open market competition through the =nil; Proof Market - a permissionless proof generation marketplace.

Introduction to zkSharding

Today, layer-2 solutions trade-off scalability for state fragmentation. We introduce a layer-2 (L2) design, =nil;, that pushes the limits of Ethereum scalability without compromising the benefits of a unified execution environment. The solution combines a dynamic sharding mechanism with verifiable access to Ethereum data, secured by zero knowledge technologies. Key elements include:

- zkRollup with Sharding: The core of =nil; is a provable sharding protocol, enabling horizontal scalability without compromising security or efficiency. This approach addresses some of the current limitations of vertical scaling (L3, L4, etc.), namely 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.

Through zkSharding =nil; benefits from advantages of both monolithic and modular designs including:

1. Scalability
2. No scalability limitations as the execution is parallel. Throughput is estimated to be around 60k ERC-20 transfers per second with roughly 400 nodes.
3. Competitive proof generation via the Proof Market provides the fastest L1-finality and cheapest proof generation costs.
4. No scalability limitations as the execution is parallel. Throughput is estimated to be around 60k ERC-20 transfers per second with roughly 400 nodes.
5. Competitive proof generation via the Proof Market provides the fastest L1-finality and cheapest proof generation costs.
6. Unified Execution Environment
7. Unified execution environment guarantees no security/liquidity fragmentation as each shard is a part of the whole cluster.
8. Reduction of a need to migrate liquidity from Ethereum as =nil; provides transparent access to its data for applications via enforcing each validator to maintain a full Ethereum state as a part of deployment allowing applications to access data right from =nil;'s zkEVM.
9. Unified execution environment guarantees no security/liquidity fragmentation as each shard is a part of the whole cluster.
10. Reduction of a need to migrate liquidity from Ethereum as =nil; provides transparent access to its data for applications via enforcing each validator to maintain a full Ethereum state as a part of deployment allowing applications to access data right from =nil;'s zkEVM.
11. Security
12. State transitions secured by 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.
13. Decentralized from day one thanks to the decentralized proof generation enabled by =nil; Proof Market.
14. State transitions secured by 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.
15. Decentralized from day one thanks to the decentralized proof generation enabled by =nil; Proof Market.

16. Functionality
17. A Type-1 zkEVM, fully EVM bytecode-equivalent zkEVM compiled via zkLLVM.
18. An environment tailored for applications that have high demands related to time, memory, and algorithmic complexity by boosting a single-shard consistency and introducing an enforced per-shard applications co-location to reduce the latency. Examples include decentralized exchanges, proof markets, decentralized sequencers/builders, shared state applications (aka autonomous worlds etc).
19. A Type-1 zkEVM, fully EVM bytecode-equivalent zkEVM compiled via zkLLVM.
20. An environment tailored for applications that have high demands related to time, memory, and algorithmic complexity by boosting a single-shard consistency and introducing an enforced per-shard applications co-location to reduce the latency. Examples include decentralized exchanges, proof markets, decentralized sequencers/builders, shared state applications (aka autonomous worlds etc).

Dynamic composable scaling

On the lower level, the state of =nil; is partitioned into the primary 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.

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.

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(1/S)
4. The hash of B(1/S) is recorded on the main shard within block B(1/M)
5. A state transition proof for B(1/S) is produced and verified by the main shard in block B(2/M)
6. A state transition proof for B(2/M) 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 a user's transaction can trigger a creation of new transactions on other shards.

With all accounts being distributed among shards, 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, external bridges.

To guarantee the security of each secondary shard, its validator committee is obligated with proving its state transitions to the primary one to ensure no fraud has happened within a smaller validator group. Each shard validators committee has additional tasks beyond shard maintenance. Validators are responsible for tracking specific types of events, namely cross-shard messages, within "near shards". Near shards are determined based on the Hamming distance in shard identifiers.

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

=nil;s zkEVM is a Type-1 zkEVM compiled with zkLLVM. To understand the differences between more traditional zkEVMs and =nil;'s zkEVM, we need to discuss limitations associated with the circuit definition process that underlies zkEVMs. zkEVM circuit is a critical part, responsible for a state transition proof to be considered correct, being usually defined with some custom zkDSL or simply a library. Such a circuit definition way brings issues related to:

- Security

: Issues due to the size of a circuit and manual replication of EVM logic.

- Auditability

: Limited auditability and inspectability due to the complexity and non-explicitness of zkDSLs used.

- Upgradeability

: Maintenance and upgradeability complexity due to manual constraints definition requirements. In case any EVM change occurs - the majority of zkEVM circuits would be required to be re-done and re-audited from scratch.

- Compatibility

: Complexity of the implementation of the actual bytecode-compatible (aka Type-1) zkEVM circuit often results in limitations for applications due to the differences in zkEVM and the actual EVM behavior.

=nil; zkEVM is effectively addressing all of these challenges by being:

- Secure

: A circuit should be automatically generated from the same high-level code used in actual production-running Ethereum nodes to ensure no algorithm differences are present.

- Auditable

: A circuit should be represented in a high-level programming language (aka C++ or Rust) which should be written in the way to be easily readable by an average developer.

- Upgradeable

: A circuit should be defined the way so any change within EVM should be easily translatable/compilable to a zkEVM circuit proving/defining exactly the same behavior. No full re-compilation or re-audit necessity should arise from such an upgrade.

- Bytecode Compatible (aka Type-1)

: Circuit compilation from high-level languages brings full bytecode and EVM behavior compatibility drastically reducing time to market for EVM applications and development time/efforts necessary to achieve such compatibility.

zkEVM compiled via zkLLVM is secure by design, leveraging evmone to ensure complete consistency with the Ethereum's production-used EVM. The zkLLVM (C++ or Rust) automatically compiles down to the circuit, meaning human error is removed from the circuit definition process.

Moreover, because =nil; zkEVM is compiled via zkLLVM, it is naturally more flexible (and hence, future proof) than manually defined circuits as it is easily adjustable and circuit generation is automatic. It is also more auditable, meaning its security doesn't come at the cost of including the latest EIPs added to Ethereum.

zkRollup with Ethereum's Security and Data Availability

As the primary shard and the secondary shards are different in regards to their dedicated tasks - secondary shards focus on transaction processing while the primary shard focuses on data synchronization - they have different approaches to data availability (DA), which helps recover state data in emergency situations. This means:

- 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, external DA solution 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.

Transparent Ethereum Data Access

One of our primary goals is to optimize for application composability and prevent liquidity fragmentation, so naturally the zkSharding approach would be incomplete without trustless access to Ethereum state. This means =nil; offers full composability and transparent integration with Ethereum through the Data Provider module.

The Data Provider operates independently from the shard's data storage, synchronizes its information with an external database and injects Ethereum's fingerprint of the last monitored database state (represented by Ethereum's block hash) into the shard's block. The most recent state of this database receives validation from the confirmation module, which uses a zkBridge with Ethereum's Casper FFG consensus proof.

What's next:

=nil; and zkSharding are a culmination of products that =nil; Foundation has developed over the last 4 years. Its aim is to be the first composable, scalable and universal Ethereum L2 zkRollup solution. We are excited to share more implementation details over the next several months. Make sure to follow our Twitter to stay up-to-date with our progress!

For the technically inclined, we've developed a separate, comprehensive primer that delves into the details of =nil; and zkSharding. This primer is your gateway to understanding the intricacies behind this approach, equipped with all the technical details and preliminaries you need.

Dive into our technical primer now and join the conversation onDiscord and Telegram. Let's explore the limitless possibilities of zkSharding together!