Main

EIP-2935: A Step to Achieving Stateless Execution

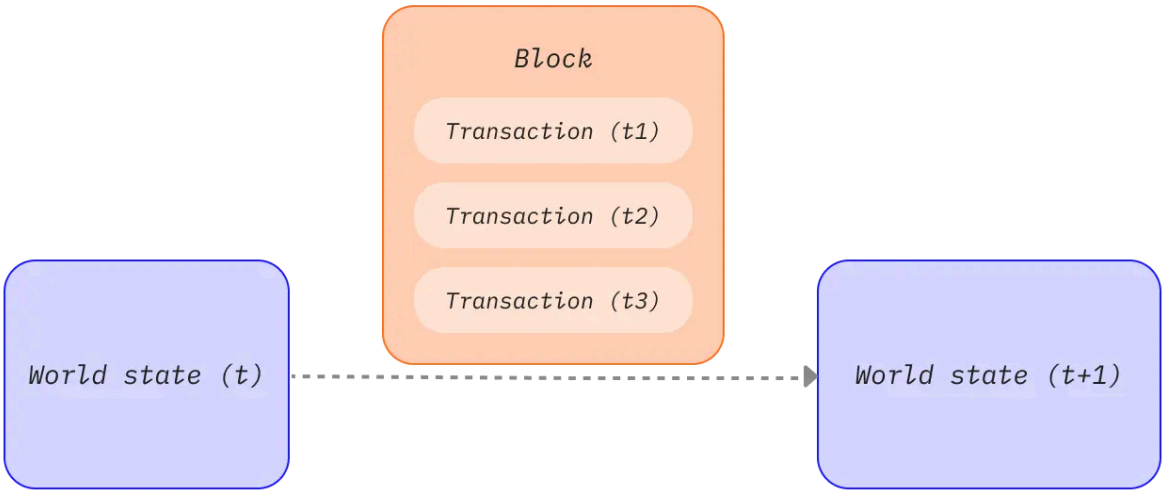
Introduction

What blockchains store and refer to while processing transactions is called *state*. On Ethereum, state is the property that facilitates node consensus. Every full node needs to store and update this state during every period of valid blocks. State, as important as they are to blockchains, come with a downside; they grow with time. They are a major problem in blockchains, such as Bitcoin and Ethereum, because the increase in size comes with an equal increase in hardware requirements for nodes. The space threshold phases out some nodes over time, leading to centralization. EIP-2935 proposes to bring statelessness to Ethereum, relieving nodes of the burden of *size*. EIP-2935 is an improvement proposal that tries to achieve statelessness by storing and serving the last 8192 block hashes from the state for stateless execution in Ethereum.

Brief Overview of Ethereum's Current Structure

Blocks

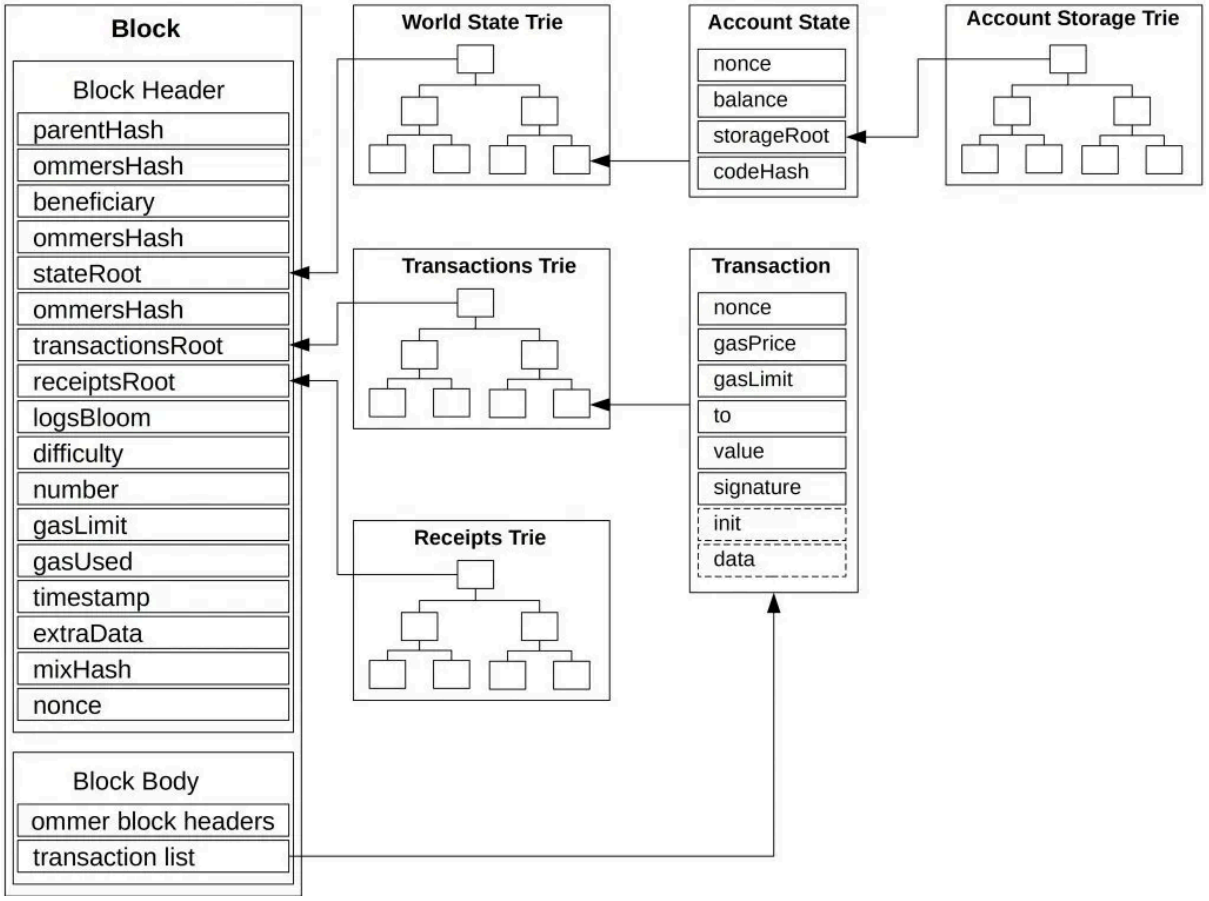
Blocks are batches of transactions with a reference (hash) of the previous block included in the chain. Hashes in each block are derived from the block data itself cryptographically. This inclusion links the chains together with hashes, and batching implies that all participants in the network agree and synchronize the state at once. Further, the agreement on batched blocks signals Ethereum to update its global state in each participant in the network as a node.



Alt: State change in Ethereum

Once a new block is handled and broadcast with a validator on the network, the rest of the participants add it to their storage and update their global state. The validator of each block is selected randomly by the Randomize Decentralized Autonomous Organization (RANDAO) parameter. The block structure is strictly ordered, and block creation and consensus processes are specified under Ethereum's proof-of-stake protocol.

A block contains several fields in its header and body. For instance, the block header includes slot, proposer_index, parent_root, state_root, and body fields. Block body includes randao_reveal, eth1_data, graffiti, proposer_slashings, attester_slashings, attestations, deposits, voluntary_exits, sync_aggregate, and execution_payload. Each field has different parameters in it, and handles different needs.



Alt: Ethereum's block structure in general terms

Ethereum's 12-second block creation period, also called a "slot", comes from enabling enough time for network participants to synchronize with new blocks and agree on the consensus. During this period:

1. The block proposer randomly selects a validator,
2. Validator bundles the transactions together and executes them to determine a new global state,
3. They include this information in a new block and broadcast it to the rest of the validator set through a gossip protocol,

4. Other validators re-execute the transactions to ensure the validity and agree with the global state change as a consensus.
5. If the validator verifies the new block as valid, they add it to their storage.

A block and transaction finality means it can't be changed without a significant ETH burn in a distributed network. Ethereum has a *"checkpoint blocks"* approach to manage finalization. The first block in each epoch is assumed to be the checkpoint of that slot. Validators vote for this assumption to make it a valid checkpoint. If two-thirds of the total staked ETH with validator votes elect a pair of checkpoints, the checkpoints are upgraded to *justified*. The previous justified checkpoints are upgraded after the next checkpoints upgrade, and they become finalized. If a malicious actor wants to revert a finalized block, it needs to commit to losing at least one-third of the total supply of staked ETH. Although a mechanism called inactivity leak exists to restore finality by imposing large penalties on validator funds and cutting down the attester rewards in the case of a permanent failure of large numbers of validators.

When processing the block, Ethereum applies a hash function to take the blocks' data and reduce it to unique strings. In the hash function, every input produces a unique output. Hash values in the blocks are the only immutable part. It is changeable with one-third of the total staked ETH burned. However, this amount comes from recalculating the Merkle trie's hashes until a checkpoint. The output of the hashing process for each block is returned with the blockHash parameter. The blockHash parameter includes all of the data in the block.

Blocks' hash values and other needed parameters are stored in Merkle tries. Ethereum uses Merkle trie architecture with different versions, such as Merkle Patricia Trie.

Merkle and Merkle Patricia Tries

Trie structures, especially Merkle Tries, are fundamental to blockchain storage. Without Merkle tries, blockchains become single blocks that are hard to process every time. With Merkle tries, or generally trie architectures, blockchains achieve completeness with basic shards. This enables them to become network participants with low hardware requirements.

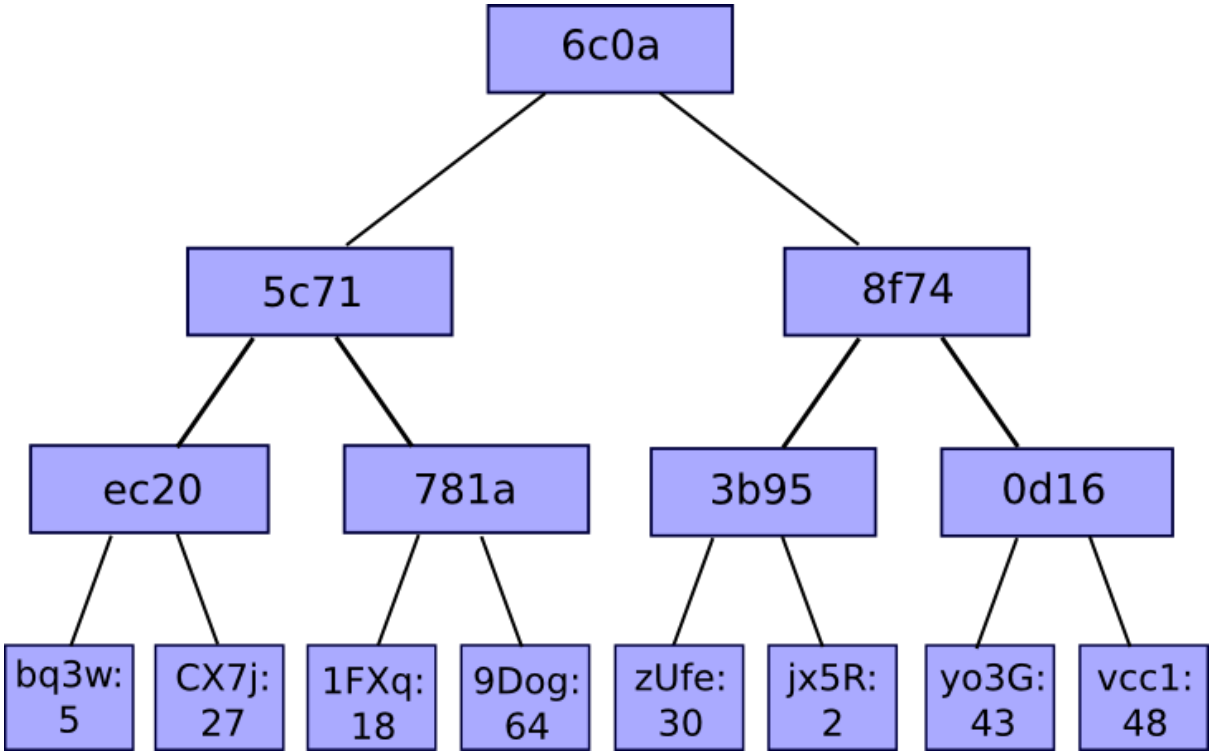
Merkle Tries are a structured way of hashing large numbers of chunks and splitting them into parts. With Merkelization, data gets organized in pairs; each is hashed together. The Merkelization process repeats until a single Merkle root is obtained.

Ethereum prefers Merkle Patricia Tries, a dual Merkle trie structure. It uses Binary Tries for basic data handling, like transaction data, since this approach is more efficient for such situations. However, Ethereum uses the more complex Merkle Patricia Tries in complex cases, such as handling state.

In a state trie data storage scenario, Ethereum leverages a key-value map. The keys represent addresses, and the values represent account declarations. State tries are more dynamic than binary tries. Thus, new accounts can be frequently inserted, and keys can be

frequently inserted and deleted. This process needs a quickly updatable data structure that doesn't require recomputing the entire data set.

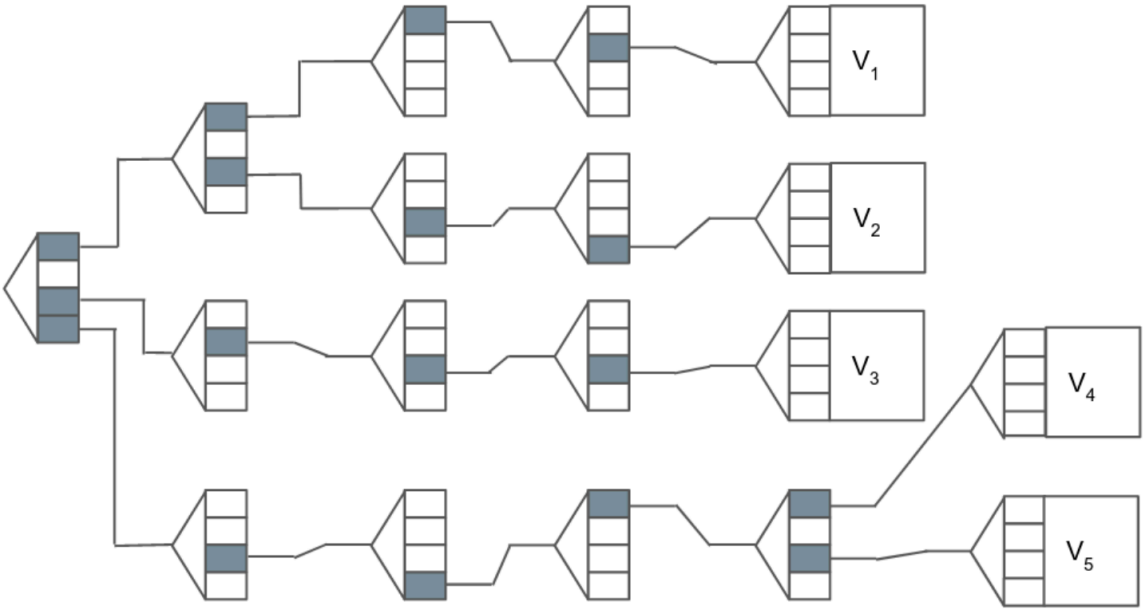
The root of the trie depends only on the data, not the order. Making updates to data in different orders won't change the root itself.



Alt: Binary Merkle Trie

Ethereum uses a modified Merkle Patricia Trie, which has some features from PATRICIA (Practical Algorithm to Retrieve Information Coded in Alphanumeric) and Merkle Trie with modifications along it. This architecture is deterministic and cryptographically verifiable. The only way to generate a state root in this structure is by computing it from each individual piece of state. Two identical states can be easily proven by comparing the root hash and the hashes that led to it.

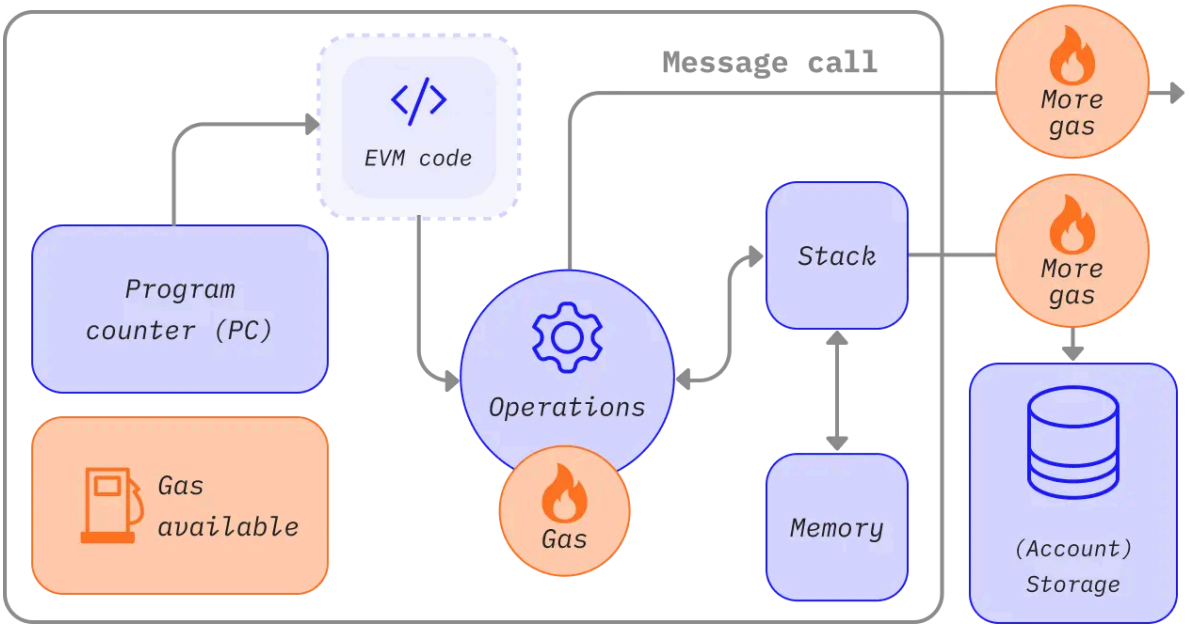
Ultimately, modifying the state with different values is impossible because it will result in a different state root hash. All trie structures in Ethereum's execution layer use a Merkle Patricia Trie. The network has three tries: State Trie, Storage Trie, and Transaction Trie. Besides, every block has its own Receipts Trie. While Merkle Patricia Tries are efficient in many ways, Ethereum is motivated to substitute them with Verkle Tries to achieve statelessness.



Alt: Merkle Patricia Trie

Gas

Gas is the property needed in EVM for execution. Since EVM uses and needs computational resources to execute, the return of these efforts is paid with gas to ensure the security of the EVM. Gas fee is calculated as the required amount of gas multiplied by the cost per unit. It is a dynamic value and depends on the type of operation.



Alt: https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

EIP-1559 introduced some significant changes to the transaction fee mechanism. Paying for gas to include transactions in the blocks worked with an auction-like method in the past. With EIP-1559, there is a minimum limit called the base fee and a tip called the priority fee introduced. Blocks are proposed to start from the transaction with the highest priority fee. After the block process, the base fee gets burned, and the priority fee is used to incentivize validators.

State

Blockchain state can be defined as a set of data (or variables) that describe a certain system at a specific period. The internet has state almost from the beginning, but it was stored on a single server. With Web3, a global state maintained on an open and distributed network secured through decentralized methods was introduced. Everyone can view and verify the state of the distributed network at any time they want.

Ethereum includes accounts, balances, deployed smart contracts, and associated storage in the global state. Ethereum's state grows with the additions and changes to these parameters. State growth becomes problematic when the hardware cost of hosting a full node becomes prohibitive. In light of these challenges, Vitalik Buterin introduced the concept of stateless Ethereum in 2017. The methods proposed to fix the state growth in Ethereum include data and state expiry.

Differences Between Data Expiry and State Expiry

Data Expiry

Data or history expiry means pruning away the unneeded data from clients after a certain period. With the “weak subjectivity checkpoints”, clients could find their way in syncing from the genesis and pruning historical waste data.

Subjectivity refers to the nodes of a network relying on social information to reach a consensus on the current state. In this light, *weak subjectivity* refers to a chain that can progress objectively with some initial seed of information retrieved socially. However, weak subjectivity checkpoints imply that all nodes on the network responsible for consensus belong in the canonical chain. Weak subjectivity checkpoints help Ethereum PoS to have the recent state (weak subjectivity checkpoint) from a trusted source to sync from.

EIP-4444 provides the practical path to achieve data expiry through weak-subjectivity. The proposal does not aim to change how Ethereum nodes handle state data; rather, it modifies access to historical data.

State Expiry

State Expiry seeks to eliminate state from individual nodes if it hasn’t been accessed recently. Expiry may be implemented by termination being a factor of rent or time. *Expiry by rent* means imposing a surcharge on accounts for storing the state. On the other hand, *expiry by time* means rendering accounts inactive if they remain inactive for a certain period.

Both Data and State Expiry are open research areas. Current approaches to achieve these proposed statuses involve passing them through centralized networks/providers. So far, weak statelessness and state expiry have been partially achieved in Ethereum.

Stateless Ethereum

Introduction to Statelessness and Stateless Ethereum

Stateless Ethereum introduces new concepts to the core protocol. Ideally, statelessness doesn't imply that state does not exist. Rather, it means clients can choose the state they want to maintain. When clients receive validated blocks, they also receive the corresponding witness for that block. Witnesses for each block consist of all the data required to execute the transactions contained in that block.

EIP-161 introduced the first state-reducing approach. Ethereum had a DoS (Denial of Service) attack, and this vulnerability allowed it to create empty accounts that increased the global state of Ethereum. EIP-161 proposed to remove empty accounts with zero (0) value at their nonce that came from this attack, with low costs. The proposal was executed, resulting in a state revert.

Another effort towards statelessness was proposed via EIP-4788. This proposal sought to expose the roots of the beacon chain blocks in the EVM. Similar to the approach of the Eth1-Eth2 bridge connecting the Beacon chain (consensus layer) and execution layer, this proposal allows trust-minimized access between EVM and the consensus layer.

Because each execution block contains the parent beacon block's root, missed slot events wouldn't need to change the previous block root. Therefore, the prerequisites of execution narrow down to reserving a small space for a trust-minimized oracle in each execution block. The proposal suggests storing a small history of block roots in a root contract to improve the oracle's efficiency.

Statelessness in Ethereum is possible either as weak or strong statelessness.

Weak Statelessness

Weak statelessness is a status that doesn't eliminate the need for state storage in all nodes. Instead, it differs in how nodes verify the changes to the Ethereum state. It puts the responsibility for state storage on block proposers and requires other nodes to verify the blocks without storing the full state data.

Block proposers need to store full state data. However, verifier clients do not have to store the state data on the network. Instead, they can store the state root (a hash of the entire state). Weak Statelessness requires Proposer-Builder Separation (PBS) and Verkle tries implementation.

Proposers create witnesses using the state data to prove the state changes, and validators verify the proofs against the state root hash.

Strong Statelessness

Strong statelessness eliminates the need for any node to store the state data. It works by incorporating sent transactions and witnesses aggregated by block proposers. Block proposers store the only state they are working on, generating witnesses for relevant accounts. The proposal still needs more development and includes requirements like Access Lists or EIP-2930.

However, some changes and properties can be used to achieve statelessness. EIP-2935 proposes to serve historical block hashes from the state to allow stateless execution.

Introduction to EIP-2935

EIP-2935 aims to save historical block hashes in the blockchain's state in special storage slots called HISTORY_STORAGE_ADDRESS. This process will allow stateless execution by facilitating easy access to the necessary historical information in stateless clients.

EIP-2935 proposes to store the last 8192 historical block hashes in a system contract to use as a part of the block processing logic. The problem this proposal is trying to solve is the EVM's assumption that the client has the recent block hash. This assumption-based approach does not apply to future Ethereum and especially stateless clients.

Including the previous block hashes in the state will allow bundling these hashes with hash functions in the sight of a witness. A witness will later be provided to the stateless client to verify the execution and achieve stateless execution. This proposal is called pre-Verkle speciation because it can be implemented before the adaptation to Verkle tries in the core protocol, and it facilitates partial statelessness.

Extending the range of blocks that blockHash serves to 8192 blocks will allow a soft transition to executional methods. Rollups can benefit from this longer history window by directly querying this contract because the blockHash data is stored in this contract. Besides, this EIP will facilitate validation of proofs related to the 8192 block amount of HISTORY_SERVE_WINDOW against the current state.

Understanding EIP-2935

EIP-2935 allows Ethereum clients, especially stateless ones, to easily access recent block hashes. To make this work, it introduces four new parameters:

- **BLOCKHASH_SERVE_WINDOW**: Tells clients how many past block hashes they should keep around. The default value is 256.
- **HISTORY_SERVE_WINDOW**: This parameter defines how many blocks' worth of hashes are stored. The default value is 8191.
- **SYSTEM_ADDRESS**: A special address (originally introduced in EIP-4788) that acts as the "caller" for writing block hashes into storage.
- **HISTORY_STORAGE_ADDRESS**: The contract address where these block hashes are stored.

The proposal's specifications direct the last HISTORY_SERVE_WINDOW block hashes to be stored in a ring buffer storage with a length of HISTORY_SERVE_WINDOW.

Ring Buffer

EIP-2935 uses an additional feature called a ring buffer for temporary storage. A ring buffer is a property that allows a network to reuse the same storage for different data. The storage is limited in the ring buffer to the same storage location each time. The ring buffer on EIP-2935 is used only to serve the requisite HISTORY_SERVE_WINDOW, as EIP-4788 and beacon state accumulators allow proof against any ancestor.

Fork Choice Rules and Specifications

After the fork, when the network starts with these EIP considerations, the HISTORY_STORAGE_ADDRESS parameter will be referred to as SYSTEM_ADDRESS with block.parent.hash input (default 32 byte), gas limit of 30.000.000, and 0 value. This process will trigger the set() function of the history contract.

The post-proposal fork process differs from the regular fork process. This set() operation is a general system operation, but the call follows these conventions:

- Must execute to completion
- Doesn't count against the block's gas limit
- Doesn't follow the EIP-1559 burn mechanism
- If no code exists at HISTORY_STORAGE_ADDRESS, the call must fail without fatal errors.

This process must fill the block period HISTORY_SERVE_WINDOW to meet the ring buffer trigger period. The history contract will only contain the parent hash of the fork block, which serves as a reference hash and the new hashing starting point.

A block hash history contract will have two operations: get() and set(). The set() operation is activated if the caller of the contract in a transaction is equal to the SYSTEM_ADDRESS that was introduced with EIP-4788. If the caller and the SYSTEM_ADDRESS are not equal to or do not fulfill the conditions, get() operation is activated.

get() operation is used in EVM to locate block hashes. It allows callers to provide the block number they are querying, if calldata input is not 32 bytes (which means that it is a valid block.parent.hash) and if the request is outside of the range ([block.number-HISTORY_SERVE_WINDOW, block.number-1]), it reverts the transaction.

set() takes input block.parent.hash as calldata when a caller calls the contract with a transaction. It also sets the storage value as calldata[0:32] at the block.number-1 % HISTORY_SERVE_WINDOW.

HISTORY_STORAGE_ADDRESS will be deployed via EIP-4788, which is referred to above as the way of accessing block hashes at EVM directly through the consensus layer.

HISTORY_STORAGE_ADDRESS will have the nonce value as 1 and be exempt from EIP-161's zero nonce cleanup standard.

BLOCKHASH Transition Logic

A concern about this EIP is how to best transition the BLOCKHASH resolution logic after the fork. Two major approaches are being considered:

- Wait for HISTORY_SERVE_WINDOW blocks for the entire relevant history to persist,
- Store all the last HISTORY_SERVE_WINDOW block hashes on the fork block.

Developers choose the first option. It is more practical because it does not require any temporary changes.

What is the difference between EIP-2935 and similar EIPs?

Similar EIPs have been proposed before to allow and achieve stateless execution. EIP-2935 differs from these prior proposals because it targets the complexities highlighted below.

- **Trie Structure Approach:** EIP-2935 doesn't use a trie-like structure with multiple layers and instead opts for a single list. On the contrary, EIP-4444 proposed pruning historical data in execution clients older than a year. Other EIPs with similar ambitions have Verkle tries as requirements.
- **Writing the EIP in EVM Code:** EIP-2935 doesn't require a change in the EVM.
- **Serial Unbounded Storage of Hashes:** Serial unbounded hash storage is inefficient. This EIP overcomes this problem by bundling the hashes.

Security Considerations

Since EIP-2935 uses smart contracts, it risks branch poisoning attacks. However, the size of the state root makes any update attempt slow and a poisoning attack considerably costlier.

What Could it Bring?

- **Making Trustless Oracle Systems Faster:** In Uniswap v2-based oracles, anyone with Ethereum node access can generate a proof of Uniswap's storage and submit it for on-chain validation. The average price is determined between the current block and the supplied proof's block, with validated proof up to 256 blocks, since the blockHash supports up to 256 blocks. Benefiting from EIP-2935, this process can be improved by allowing access to older block hashes, which means proofs can be validated over a longer period.
- **Allowing Contracts to Consider Past State Assertions, Trustlessly:** EIP-2935 improvement creates the possibility to look at blockchain data from inside the EVM, trustlessly. A client can query the history, get it hashed, and verify it with other nodes. The solution could make light clients efficient and easy to implement.

- **Bridging between L1 <> L2:** To verify a message from L2, L1 needs to know about L2 state roots and block hashes. However, L1 in its current state can't access arbitrary block hashes due to gas limits and architectural constraints. EIP-2935 enables L1 to verify the arbitrary historic data with the ability to probe inclusion proofs for old events. The access and verification power will improve, and the bridging performance.

Conclusion

EIP-2935 represents a crucial step toward achieving the long-term goal of stateless Ethereum. Storing the last 8192 block hashes in a dedicated contract within the state addresses a fundamental limitation in the current design. Ethereum's assumption that clients inherently have access to recent block hashes. In the context of stateless execution, where clients no longer maintain the full state, this assumption becomes outdated. EIP-2935 resolves this by introducing a lightweight, efficient, and non-intrusive mechanism that enables stateless clients to access necessary historical data without modifying the EVM or relying on complex data structures like Verkle tries.

Beyond stateless execution, this proposal unlocks broader benefits across the Ethereum ecosystem. It enhances the capabilities of trustless oracles, extends the usability of light clients, and strengthens interoperability between Layer 1 and Layer 2 solutions by enabling reliable verification of older state data. Its implementation relies on a clean and gas-efficient design, using ring buffer storage and system-level contracts that avoid the need for hardcoding into the EVM, offering both simplicity and scalability.

As Ethereum continues its evolution toward greater decentralization, scalability, and accessibility, EIP-2935 serves as a foundational improvement. It bridges the gap between the current stateful architecture and the envisioned stateless future and lays the groundwork for more robust, efficient, and permissionless infrastructure across the blockchain ecosystem.