Portal Network & Verkle

Summary

The goal of this research is to describe how Verkle trie (see EIP-6800) can be efficiently stored in a Portal Network.

Firstly, I will give quick overview of what Portal Network and Verkle Tries are and how they work, with the focus on parts that are most relevant to this research. Afterwards, I will specify my proposed solution.

Portal Network overview

Portal Network (https://www.ethportal.net/) can be explained as decentralized Ethereum archive node. It's designed as a peer-to-peer network, where each node holds only small portion of data, while the entire Portal Network stores entire historical Ethereum data. Each Portal Network node decides which content to store locally based on the distance between its Node-Id

and Content-Id

.

The Portal Network is actually several peer-to-peer networks (beacon, history and state). Each of these networks stores a specific subset of the data stored by an Ethereum full node. The focus of this research is the state network, whose specification for the Merkle Patricia Trie (MPT) can be found here: link.

In short, Portal Network stores every trie node and smart contract bytecode from Ethereum state that ever existed. Each entry (trie node/bytecode) is represented with an unique Content-Key

. The hash of the Content-Key

is 32-byte long Content-Id

.

In the case of the MPT, there are 3 types of content:

- Account Trie Node - the trie node of the main state trie
- It is uniquely identified by the path from the root of the trie and the hash of the trie node
- The value is the rlp encoding of the trie node itself
- It is uniquely identified by the path from the root of the trie and the hash of the trie node
- The value is the rlp encoding of the trie node itself
- Contract's Storage Trie Node - the trie node that belongs to the smart contract's storage trie
- It is uniquely identified by the address of the smart contract, path from the root of the storage trie, and the hash of the trie node
- Same as above, value is the rlp encoding of the trie node itself
- It is uniquely identified by the address of the smart contract, path from the root of the storage trie, and the hash of the trie node
- Same as above, value is the rlp encoding of the trie node itself
- Contract bytecode - the bytecode of the smart contract
- It is uniquely identified by the address of the smart contract and the hash of the bytecode
- The value is the bytecode itself
- It is uniquely identified by the address of the smart contract and the hash of the bytecode
- The value is the bytecode itself

In order to prevent bad actors from polluting the network with invalid data, the gossiped content contains the Merkle proof, so receiver can verify that the content is canonical to the Ethereum chain. This proof is not stored locally in order to improve performance.

Verkle Trie overview

The Verkle Trie is the new structure for storing Ethereum state. Current plan is for Ethereum to switch from Merkle Patricia Trie to Verkle Trie two forks from now (sometime in 2025).

The details of why Ethereum wants to switch to the Verkle trie, and how Verkle trie works can be found in the EIP-6800. However, we are going to provide overview of its properties that are relevant for this research.

Pedersen commitment

Each trie node is uniquely represented with a Pedersen commitment, which are based on Elliptic curves. The curve that is used is Banderwagon (the variation of the Bandersnatch curve).

The Pederson commitment is calculated using following formula:

C = a_0 B_0 + a_1 B_1 + ... + a_{255} B_{255} = Commit(a_0, a_1, ... , a_{255})

where B_i

(basis) are already known, fixed points on the elliptic curve and a_i

are values we are being committed to. The values a_i

have to be from the scalar field of the elliptic curve, which in our case has less than 2^{253}

elements.

The Pedersen commitment itself is a point on the elliptic curve. The Pedersen hash is a value in a scalar field derived from the Pederson commitment.

Structure overview

Verkle Trie is a trie structure with a branching factor of 256, and is used for storing key-value pairs where both key and value are arrays of 32-bytes.

Verkle trie has 2 types of nodes:

- Inner node - contains up to 256 Pedersen hashes, representing the 256 children nodes
- Extension node - contains the path from the root of the trie (31-byte long, called stem) and up to 256 values (for each value of the last byte, suffix)
- Each value is 32-byte long array
- Each value is 32-byte long array

[

image - Verle trie

811×401 100 KB

](https://ethresear.ch/uploads/default/original/3X/7/a/7a12397bf7f94f89c116dc88c4ccfb5d07e0c52c.png)

Representation of a walk through a Verkle tree for the key 0xfe0002abcd..ff04

: the path goes through 3 internal nodes (children: 254, 0, 2) and an extension node. Image taken from here.

Extension node structure

Value that is stored in a trie is 32-bytes, which can't be uniquely mapped to the elliptic curve scalar field (less then 2^{253}

elements). To circumvent this problem, the 256 values of the Extension node are split into two groups of 128 each, and each value is split into two 16-bytes values: v_i = v_i^{low} + v_i^{high}

.

Also, in order to distinguish absent from zero values, when v_i

is accessed, the marker is set on the 129th bit of the v_i^{low}

, resulting in: v_i^{low+access} = v_i^{low} + 2^{128}

Two commitments of each group of 128 values is calculated separately (C_1

and C_2

):

\begin{align*} C_1 &= Commit(v_0^{low+access}, v_0^{high}, v_1^{low+access}, v_1^{high}, ... , v_{127}^{low+access}, v_{127}^{high}) \ C_2 &= Commit(v_{128}^{low+access}, v_{128}^{high}, v_{129}^{low+access}, v_{129}^{high}, ... , v_{255}^{low+access}, v_{255}^{high}) \end{align*}

They are then combined with the stem to calculate commitment of the entire Extension node:

C = Commit(1, stem, C_1, C_2)

Other important details

There are many more details regarding Verkle trie, but only important ones for the rest of this research are given below:

- Unlike MPT, smart contract's bytecode and storage values are stored within the same trie
- contract's storage slots are mapped to 32 bytes, which are used as a key of the Verkle Trie
- bytecode is split into chunks of 31 bytes
- contract's storage slots are mapped to 32 bytes, which are used as a key of the Verkle Trie
- bytecode is split into chunks of 31 bytes
- Values that are likely to be accessed together (e.g. balance and nonce, consecutive contract's storage slots, bytecode chunks) are likely going to end up in the same Extension node
- It's possible to update node's commitment if we know previous commitment, which children changed, and previous and new commitment of those children
- Most EL clients use this and only store commitments of the Inner nodes (not inner nodes themselves)
- Same approach doesn't work well for archive nodes
- Most EL clients use this and only store commitments of the Inner nodes (not inner nodes themselves)
- Same approach doesn't work well for archive nodes

Portal Network & Verkle Trie

The simplest approach would be do the same for Verkle Trie as we are doing in the case of Merkle Patricia Trie: store every trie node

. However, this leads to much higher storage requirement for the Portal Network.

When block is executed, updated trie nodes (except the root and the first layer nodes) would have only few children values updated while majority would stay the same. If we do naive approach (store every trie node as it is), Portal Network as a whole would need significantly more storage. This issue is significantly more impactful with Verkle trie in comparison with MPT because of the bigger branching factor (256 vs. 16).

However, we can do this in a smarter way. Let's consider Inner node C

with children C_0, C_1, ... ,C_{255}

. Instead of storing it as a single node, we can split it into 2-layer trie:

\begin{alignat*}{3} C &=& C'0 + C'_1 &+…+ C{15} \ C'0 &=& C_0B_0 + C_1B_1 &+…+ C{15}B_{15} &&= Commit(C_0, …,C_{15}, 0,…,0) \ C'1 &=& C{16}B_{16} + C_{17}B_{17} &+…+ C_{31}B_{31} &&= Commit(0,…,0, C_{16},…,C_{31}, 0, …,0) \ &&&\quad… \ C'{15} &=& C{240}B_{240} + C_{241}B_{241} &+…+ C_{255}B_{255} &&= Commit(0,…,0, C_{240}, …,C_{255}) \ \end{alignat*}

[

image - Inner node split

768×287 10.4 KB

](https://ethresear.ch/uploads/default/original/3X/9/e/9ec60b31ab7d4af8127fe23d8f077b2ab9430f6c.png)

This would result in the root node of such trie having the same commitment as it would have otherwise. But this way, we would have a trie with a branching factor of 16. Similar approach can be applied to the Extension node as well.

It’s also worth noting that this splitting can be done again (leading to 4-layer trie with branching factor of 4), or even multiple times (8-layer binary trie), or even in alternative way (3-layer trie with branching factors [4,8,8]

). Deeper tries would probably require less total storage, but they would make lookup queries more expensive.

Other considered solutions

Path based Content-Id

Another alternative solution that seems plausible is to make Content-Key

and/or Content-Id

dependent only on the path from the root, not the trie commitment itself.

Benefits of this approach would be that the same Portal Network node can optimize its storage in order not to duplicate values that don’t change over time (e.g. by storing only diffs).

However, this approach has a main downside, which is that content space is not uniformly distributed. The trie nodes associated with frequently used smart contracts (e.g. WETH, DEX pools, Layer-2, Oracles, …) will have significantly more updates than the average trie node. This non-uniformity breaks linear relationship between node’s radius and the amount of data that it needs to store, which is the major assumption about Portal Network design.