Week 7 Research Track - Verkle Trees

Guest speaker

Guillaume, Ethereum Geth team

Ignacio, EF research team

Josh, EF research team

Verkle slides

Summary notes

- Edited by Chloe Zhu
- Online version: https://ab9jvcjkej.feishu.cn/docx/C6VMdpDDHoRq2XxGmKicy0E3nng

EF stateless consensus team & Website info

- Team member
  - Guillaume, team lead
  - Ignacio
  - Josh
  - Kevaundray
- Verkle website: https://verkle.info/

Motivation of the Verkle

- Stateful applications are complex
  - State in systems creates many challenges, eg. state access, efficiency, storage
  - State (usually) only grows with time
- For Ethereum, it puts some pressure on core values
  - Currently, to be able to validate blocks, the prerequisites incl.
    - Download all the state, which takes time
    - Save the state somewhere -> a full note requires ~ 1T + 300GB disk
  - Handling state isn't zk-friendly
- Towards a better stateless world

    - A new node can join the network
        - Without the need to sync all the state
        - No requirement of disk storage for the EL client
    - Become Zk-friendly L1, by
        - Removing complex data structure (MPT)
        - Removing heavy use of non-zk-friendly hashes (i.e. Keccak)
    - Reduce hardware requirements
    - Simplify the implementation of a new (stateless) EL client
    - Potentially allowing increasing the gas limit
    - Might trigger the specialization of protocol roles, eg. different nodes have different roles in the protocol

Design & Architecture

Tldr of the Verkle

- **Execution witness**: Key concept introduced into the protocol, which
    - Contains all the state needed to execute a block
    - Contains a very small cryptographic proof to validate the correctness of the witness
    - Contains all the code needed to excute the contracts

- The execution of Verkle transition is complicated
    - Need to introduce a new cryptography stack
    - Need to change the state tree(s) data structure
    - Need to change gas accounting
    - Need to migrate data from MPT to VKT

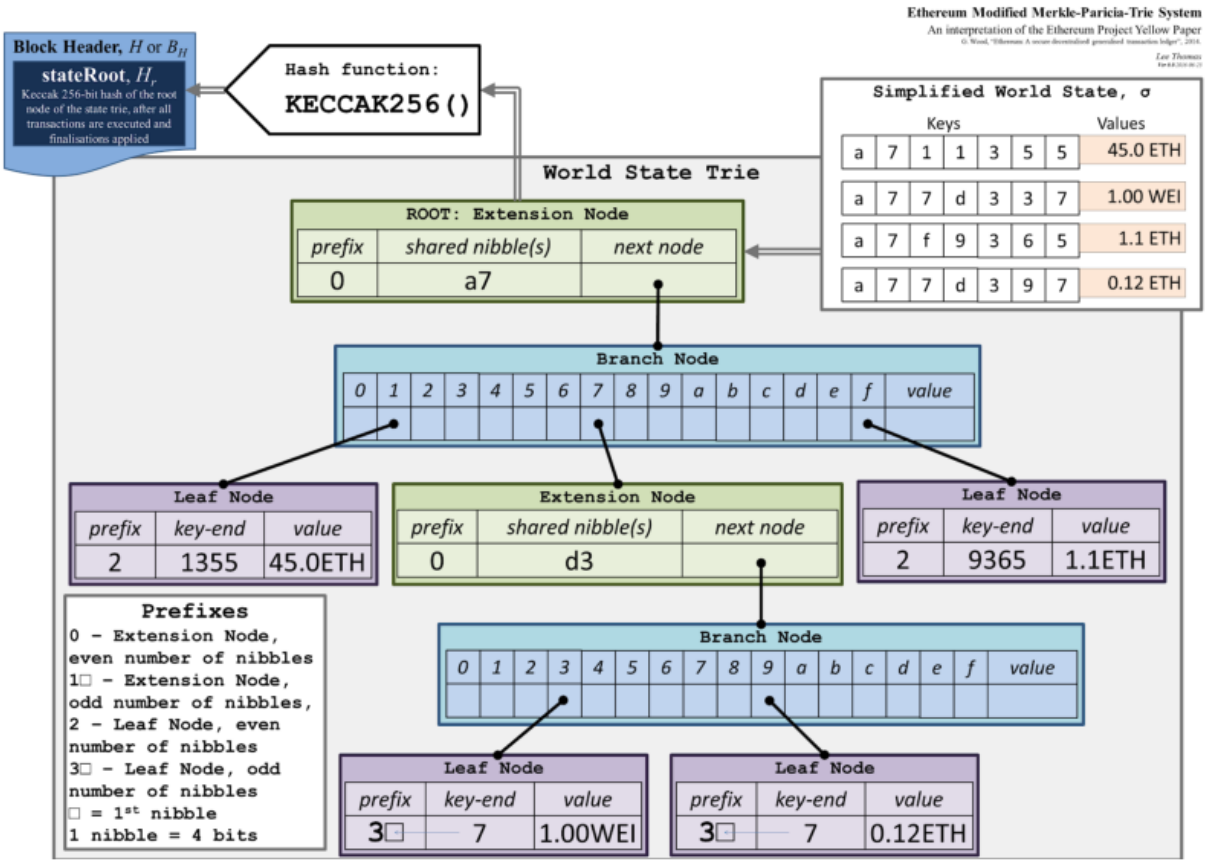
Cryptography

- As the execution witness proof has a small size, it allows

- The witness to be transmitted with each block i.e. Each stateless client will need it
- Keeping the protocol trustless i.e. Each stateless client can verify the block without trusting any 3rd party

Cryptography used today for the state tree

- Keccak hash function
- Merkle Patricia Tree

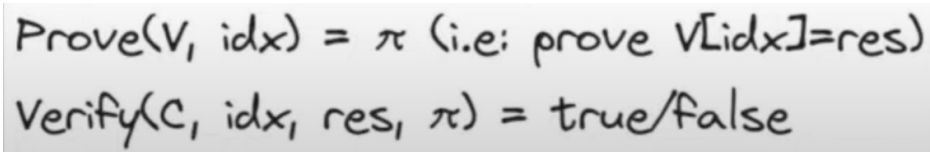


Cryptography used in Verkle trees

- **Vector commitement**

- Vectors:
  - Vectors of 256 elements
  - Each of the vectors is a scalar field element
- Commit function:
  - Allow you to provide these vectors and recieve a commitment function

        - The commitment is binding i.e. when the vector changes, the commitment will also change
    - Opening



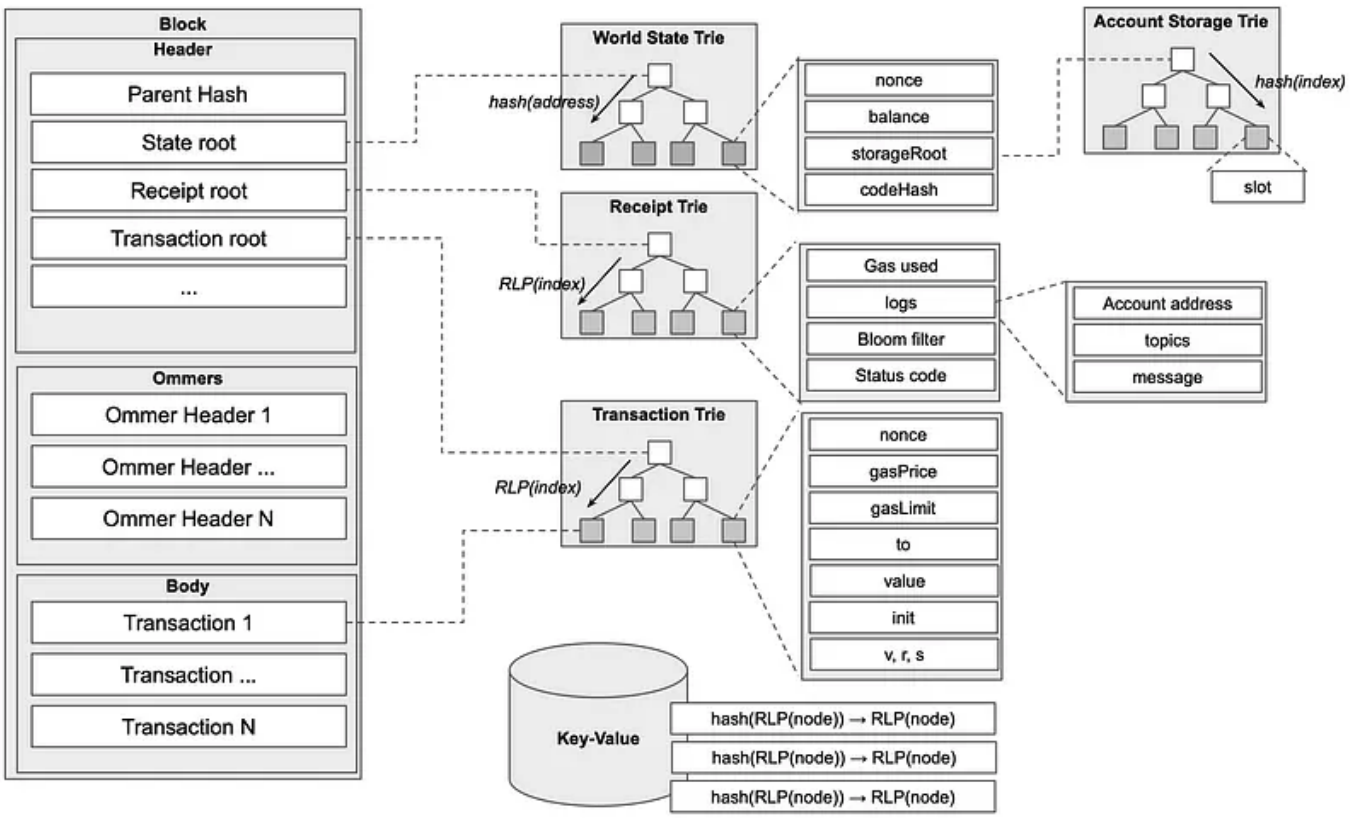
        - Prove function:
            - Input: a vector and an index
            - Output: π, which represents the proof of the element's value corresponding to the index
        - Verify function:
            - Input: commitment of the vector, the index, the claimed value, and the proof
            - Output: true of false, depending on the correctness of the proof
            - Key part: **the verify function only needs the commitment, rather than the vector itself**
    - Group
        - The factor commitment and proving scheme is used under a setup of a new elliptic curve called **Bandersnatch**
            - **Banderwagon** is also used as abstraction on top to remove cofactor and make the group safe
            - Notes: Kevaundray's blog on Banderwagon (high level), and from Bandersnatch to Banderwagon
        - Scalar field (Fr) of the curve = 253 bits
        - Base field (Fp) of the curve = 255 bits
        - No pairings (i.e. Smaller fields -> more efficient)
        - Zk-friendly
            - Base field (Fp) is the Scalar field (Fr) of BLS12-381
            - Doing EC operations in a circuit are native field operations (i.e. Not emulating fields)
- **Inner product argument**
    - Single vector opening (no trusted setup!)
- **Multiproofs**

- Aggregate multiple vector openings in a single vector opening, which can make the proof even shorter

Data structure

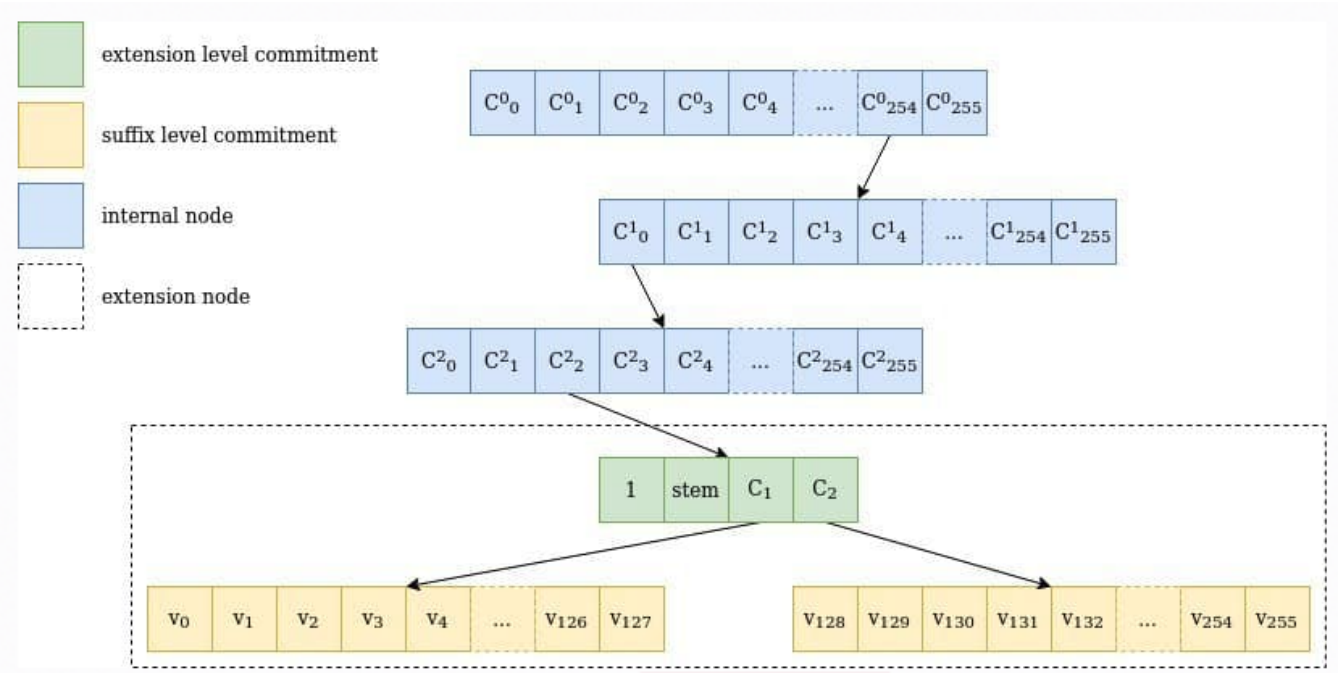
Merkle Patricia Tree (MPT) -> Verkle Tree (VKT)

- The naming of "Verkle tree" comes from Vector commitment + Merkle tree
- Today:



  - The Ethereum state is stored in 4 different modified merkle patricia tries (MMPTs):
    - World State Trie: linked to State root
      - Leaf nodes:
        - EOA: contain nonce & balance
        - Smart contract accounts: contain storage root & code hash
    - Account State Trie: linked to storage root
    - Receipt Trie: linked to receipt root
    - Transaction Trie: linked to transaction root
  - Reference
    - Data structure: https://epf.wiki/#/wiki/protocol/data-structures

    - MPT: https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/
- Future: EIP-6800
    - Instead of trees of trees under MPT, we will have a single tree under VKT
    - The tree shape will change as well

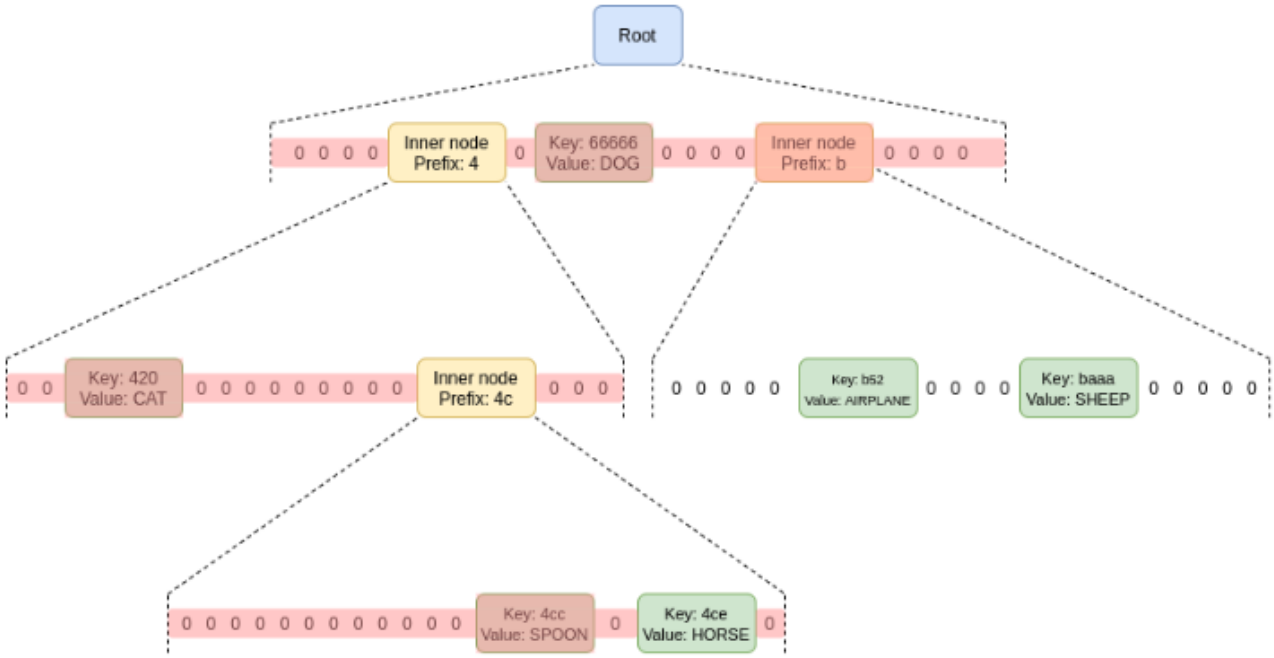


$$C_1 = \text{Commit}(v_0^{(\text{lower,modified})}, v_0^{(\text{upper})}, v_1^{(\text{lower,modified})}, v_1^{(\text{upper})}, \ldots, v_{127}^{(\text{lower,modified})}, v_{127}^{(\text{upper})})$$
$$C_2 = \text{Commit}(v_{128}^{(\text{lower,modified})}, v_{128}^{(\text{upper})}, v_{129}^{(\text{lower,modified})}, v_{129}^{(\text{upper})}, \ldots, v_{255}^{(\text{lower,modified})}, v_{255}^{(\text{upper})})$$

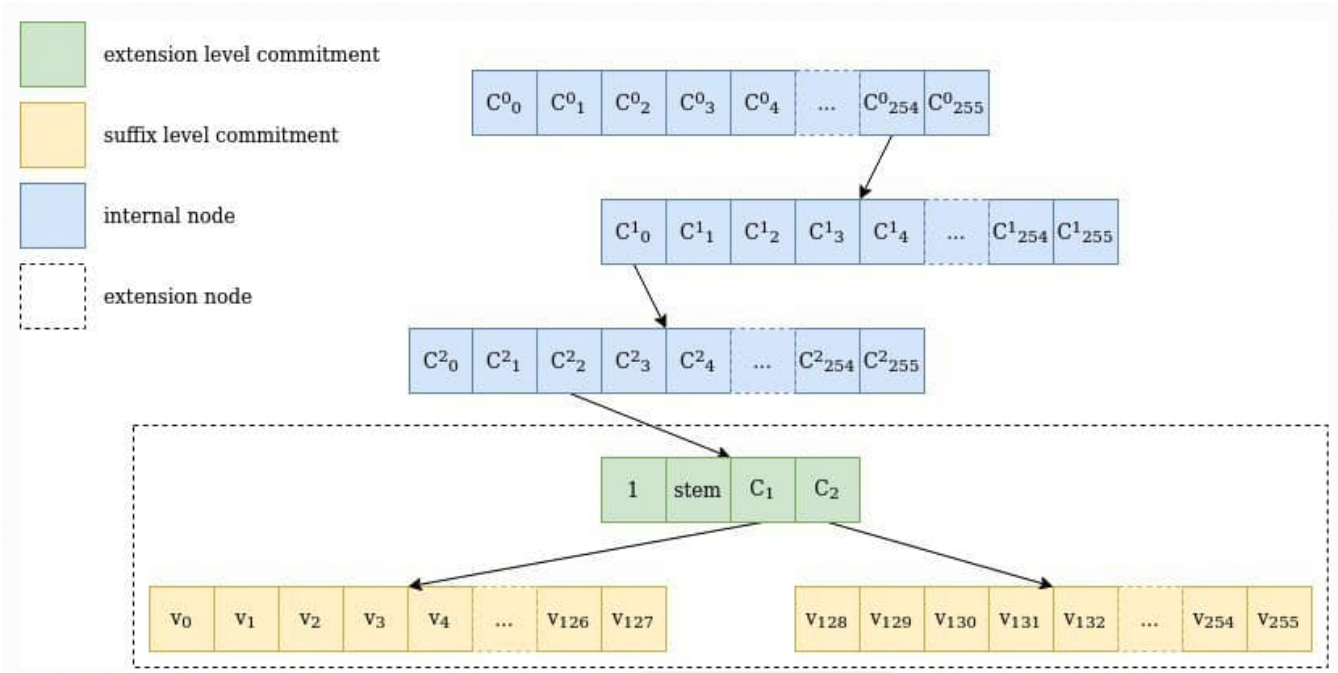
    - Internal node: with a branching factor of 256
    - Extension & suffix level commitment:
        - Tree keys are 32 bytes: The 1st 31 bytes define a stem, which is a path the extension level commitments
        - Each leaf node contains 256 values
    - Reference
        - EIP- 6800 Ethereum state using a unified verkle tree: https://eips.ethereum.org/EIPS/eip-6800
        - Verkle tree structure blog: https://blog.ethereum.org/2021/12/02/verkle-tree-structure

Proof size: MPT vs VKT

- Proof size
    - MPT
        - Still need all the nodes data to create the proof



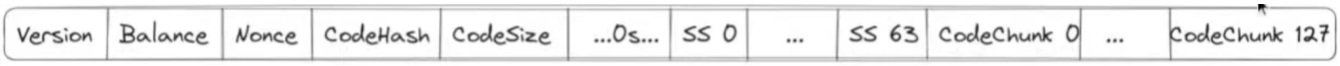
- VKT
  - Since we use the vector commitment proving schemes, vector data itself is not needed



- Reference
  - Vitalik's blog on verkle tree: https://vitalik.eth.limo/general/2021/06/18/verkle.html

How to store info in leaves?

- Leaf nodes: can store 256 (scalar field) values
  - Currently, the first 64 storage slots and the first 128 codechunks are stored in the leaf node of the account
  - Account header



- Future storage slots

- What's CodeChunk?
  - In the execution witness, it contains all the contract code to validate the block
  - In fact, the contract code is chunked to be stored in the tree
    - A code-chunk is a 32-byte value
    - Chunk[0] = how many bytes are a continuation of a PUSHX instruction of previous chunk
    - Chunk[1:32] = Bytecode[0:31]
- Reference
  - EIP- 6800 Ethereum state using a unified verkle tree: https://eips.ethereum.org/EIPS/eip-6800

How tree keys are calculated?

- Keccak will no longer be used
- Instead, Verkle uses an EC based hashing function (i.e. Pedersen hash)
  - TreeKey(address, treeIndex, subIndex) = Commit(2+256*64, address[0:16], address[16:32], treeIndex[0:16], treeIndex[16:32])[0:31] ++ subIndex
  - How it functions
    - It's basically doing a vector commitment of a vector size of 5 as the input
    - The output of the commitment will give a 32-byte value, which we will take the 1st 31, and append it with the subIndex, which is a single byte
- Storage of the rest slots apart from the first 64 slots
  - pos = MAIN_STORAGE_OFFSET + storage_key
  - TreeKey(address, pos / VERKLE_NODE_WIDTH, pos % VERKLE_NODE_WIDTH)
- Storage of the rest codechunks apart from the first 128
  - chunk_id = CODE_OFFSET + chunk_id
  - TreeKey(address, chunk_id / VERKLE_NODE_WIDTH, chunk_id % VERKLE_NODE_WIDTH)
- Reference

- EIP- 6800 Ethereum state using a unified verkle tree: https://eips.ethereum.org/EIPS/eip-6800

Gas accounting

- Gas accounting to reflect the change
  - In stateless world, I/O speed and state size are no longer the primary concern
  - Gas should account for increasing the witness size
- Actions that increase witness size
  - Reading state: state needed to be able to execute the block
  - Writing state: provide how the tree looks like in write positions, to be able to update the tree
  - Executing code: stateless client need the code to execute!
- 5 Gas cost changes
  - Accessing a new tree branch -> WITNESS_BRANCH_COST (1900)
  - Accessing a new value in a leaf -> WITNESS_CHUNK_COST (200)
  - Write triggers updating a branch -> SUBTREE_EDIT_COST (3000)
  - Changed value in leaf -> CHUNK_EDIT_COST (500)
  - Wrote a leaf node which was empty -> CHUNK_FILL_COST (6200)
- The above accesses happen when:
  - Particular opcodes are executed e.g SSTORE, SLOAD, BALANCE, SELFDESTRUCT, etc
  - Indirect tree access in transaction executions e.g: send value FROM to TO (Not charged!)
  - Indirect tree access in block execution e.g: withdrawals, block rewards (Not charged!)
  - Contract code is executed
- Notes
  - These charges only happen once per key per transaction
  - Any further storage slot access charges “warm access” cost (100)
- Reference
  - EIP 4762 Statelessness gas cost changes: https://eips.ethereum.org/EIPS/eip-4762

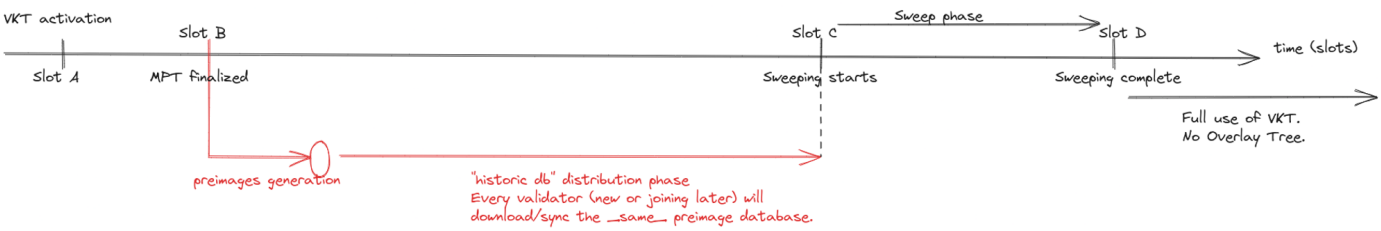
State conversion

- Need to migrate all the state from MPT -> VKT
- Conversion strategies

    - Multiple strategies have been considered
    - Currently, the "Overlay tree" is the proposed one
- The Overlay tree strategy
    - Freeze the current MPT at a fork block
    - Lay another tree, i.e. the VKT, on top
    - Every new writes then goes to the new tree
    - In every block, n values were taken from the MPT and their keys were converted then inserted in the VKT
    - The conversion ends when all leaves from the MPT transferred to the VKT
    - Internal MPT nodes will be deleted when the block is finalized (but can be stored else where)
- Timeline of the overlay method
    - If N = 10k, the conversion will take 15 days; if N = 5k, 1 month; if N = 1k, 6 months
    - Why not N = 100k
        - Consequence: more leaves get translated per block → more nodes dropping from the network
        - Tradeoff: # of machines that can follow <> keep the conversion short enough

More challenges

- Re-hashing tree keys requires the pre-images
- But not all EL clients have a design or default flags to store hash pre-images
- How nodes can have this info before the storage transition starts?



    - Current strategy
        - Erigon & Reth store their data by pre-images, so they have the ability to generate a file that could be distributed
        - But the problem is the pre-image generation takes time, therefore the sweep can get started directly. And pre-image distribution could have issues as well.
        - It's still an open problem, but the current strategy doesn't seem to have big problem.

    - Reference
        - Verkle Trees - Preimages generation & distribution strategy: https://hackmd.io/@jsign/vkt-preimage-generation-and-distribution

Verkle sync

- How Verkle can simplify the sync
    - New node will join the network and begin executing the block stateless
    - When the node keeps rebuilding the VKT, it will accumulate all the leaves
    - In the background, the node will start asking other full nodes in the network for all the leaves that haven't been seen in the stateless view and verify them if they fits the current view
    - It doesn't matter how long it takes to backfill the data, as the nodes can execute blocks stateless at the same time

Join the efforts

- Biweekly calls: Verkle implementers call
- Discussion channel: Ethereum R&D discord **#verkle-trie-migration**

Q&A

- Could you walk through how retrieval changes with Verkle trees? What’ s the benefit on calculating the extra commitments (suffix etc)
    - MPT uses hashes, which can be handled by CPU very fast, but for EC it takes a much longer time to process. Therefore, it's a tradeoff for Verkle to be zk-friendly for SNARK.
- Does the verkle tree implementation of ethereum also have the hiding property?
    - No hiding property, only binding property
- Does Verkle make is easier for future security implementation under quantum computing era?
    - As Verke uses EC, it is not safe for quantum. There is also tradeoff between using quantum safe cryptography and speed & UX impact
    - By converting from MPT the VKT, the team will have the conversion experience ready for future conversion.
- MPTs have a branching factor of 16, while Verkle has it at 256 are there any benchmarks links from the team comparing the tradeoffs for bandwidth and computational effort?

    - Since the proof size and the verification time is linear on the size of the vector, there is tradeoff between the branching factor and the cost of doing openings for verifying openings for verifiers.
    - Increasing the branching factors will make the tree shallower, which results in less elements in the proof.
- What kind of machine (CPU/ ARM) will I need to be able to follow the full transition to verkle trees?
    - Feel comfortable that desktop machine and even Rock 5B can handle 10k leaves per block
- Areas of contribution
    - Current roadmap is focused on sync, conversion, and pre-image distribution
    - Potential contribution areas
        - Implementation on every client eg. Reth, Erigon, Besu
        - Stateless client
        - L2, Dapps compliers' ability to handle of Verkle
        - Tooling support eg. block explorer
        - Zk related (maybe bit too early)
        - State expiry
        - Interaction with account abstraction
- Timeline for Verkle in the hardfork?
    - A year is realistic as most of the work has done with only a few open problems like sync and pre-image distribution