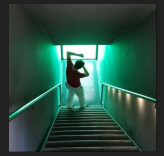
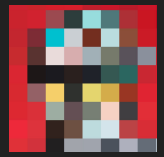
Verkle Trees 101

EF Stateless consensus team



Guillaume Ballet

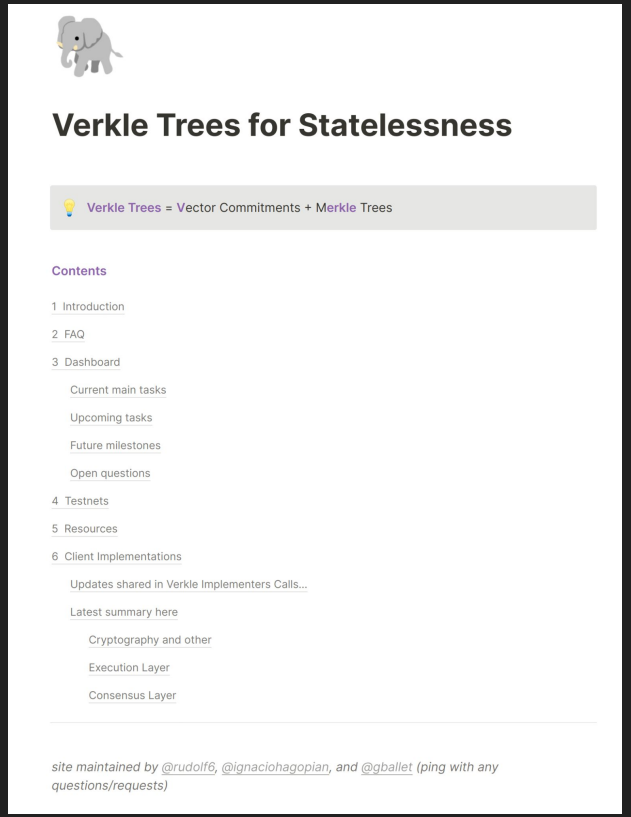
Ignacio Hagopian



Josh Rudolf

Kevaundray Wedderburn

verkle.info



Verkle Trees for Statelessness

Verkle Trees = Vector Commitments + Merkle Trees

Contents

1 Introduction
2 FAQ
3 Dashboard
- Current main tasks
- Upcoming tasks
- Future milestones
- Open questions

4 Testnets
5 Resources
6 Client Implementations
- Updates shared in Verkle Implementers Calls...
- Latest summary here
  - Cryptography and other
  - Execution Layer
  - Consensus Layer

site maintained by @rudolf6, @ignaciohagopian, and @gballet (ping with any questions/requests)

Motivation

Motivation

- Stateful applications are complex!
- State in systems create many challenges
- State (usually) only grows with time
- For Ethereum it puts some pressure on core values

Motivation

- Stateful applications are complex!
- State in systems create many challenges
- State (usually) only grows with time
- For Ethereum it puts some pressure on core values
- Before being able to validate blocks:
    - Download all the state, which takes time
    - Save it somewhere, a full node requires ~(1TiB+300GiB) disk
- Handling state isn’t zk-friendly

Motivation

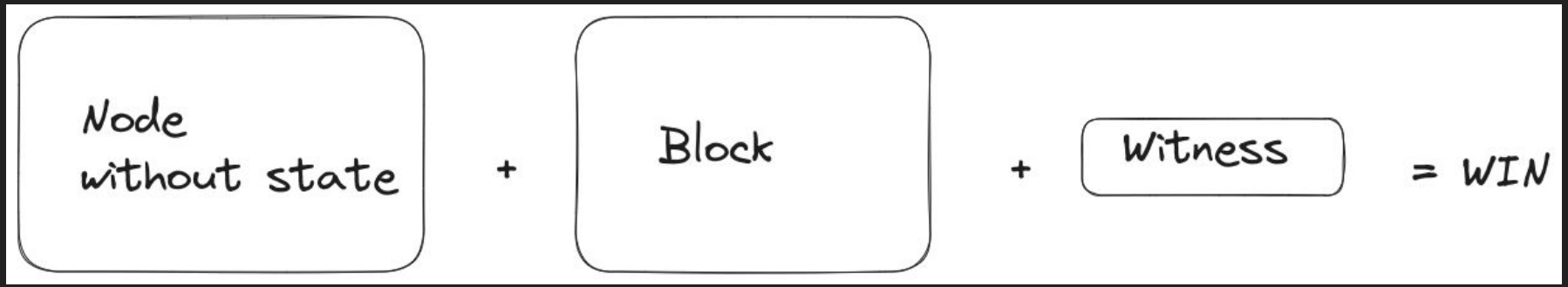
- Let's build a better (stateless) world…

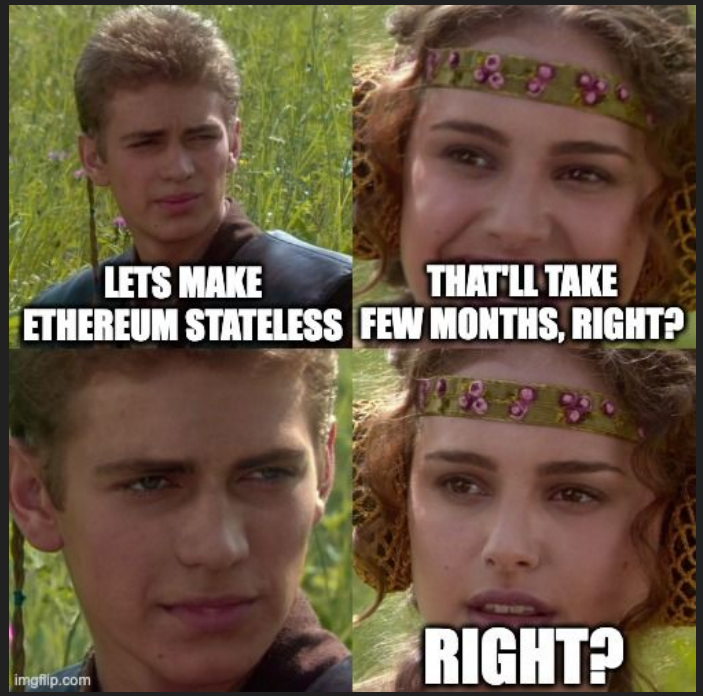
Motivation

- Let's build a better (stateless) world…
- A new node joining the network:
  - Doesn’t have to sync all the state.
  - Doesn’t require disk storage for the EL client
- zk-friendly L1:
  - We remove complex data-structure (MPT)
  - We remove heavy use of non-zk-friendly hashes (i.e: K
- Reduce hardware requirements
- Easier to implement a new (stateless) EL client
- Potentially allow increasing the gas limit
- Might trigger the specialization of protocol roles

TL;DR?

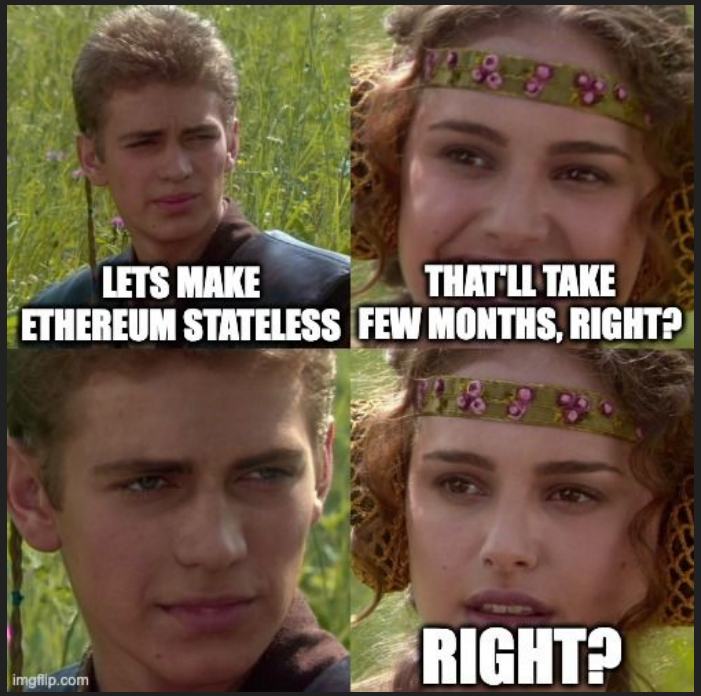
- New kid in town: *Execution witness*
- Contains all the state needed to execute a block
- Contains a (small!) cryptographic proof it’s correct
- “All state needed” includes contract code!




LETS MAKE ETHEREUM STATELESS
THAT'LL TAKE FEW MONTHS, RIGHT?
RIGHT?
imgflip.com

Life 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



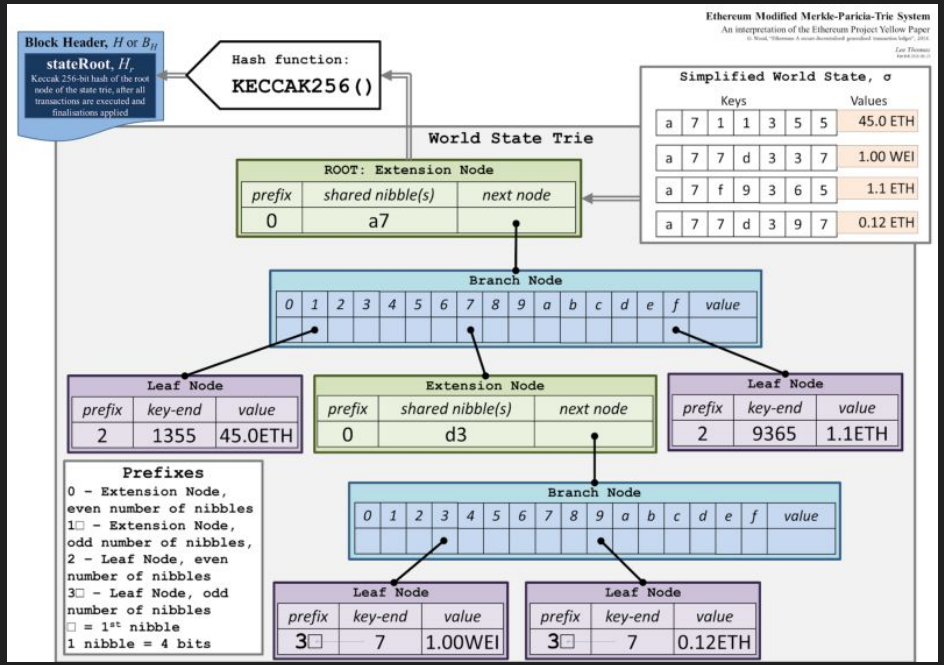
Cryptography

Where the magic happens…

- What makes the *execution witness* proof have a small size:
  - Allows the witness to be transmitted with each block (i.e: each stateless client will need it)
  - Allows keeping the protocol trustless
- Ingredients:
  - *Vector Commitments*
  - *Inner Product Argument*
  - *Multiproofs*

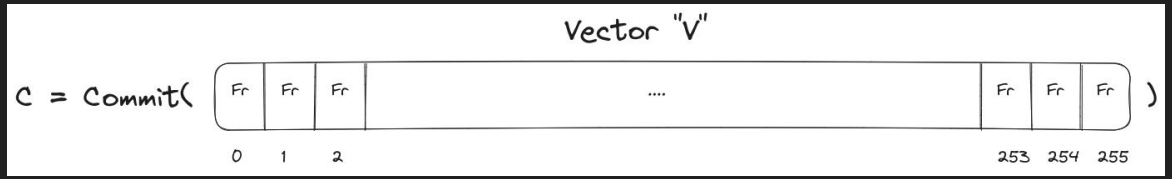
Cryptography used today for the state tree

- Cryptographic hash function: Keccak
- Merkle Patricia Tree:



Cryptography used in Verkle Trees

- Vector commitment:



- Opening:

Prove(V, idx) = π (i.e: prove V[idx]=res)

Verify(C, idx, res, π) = true/false

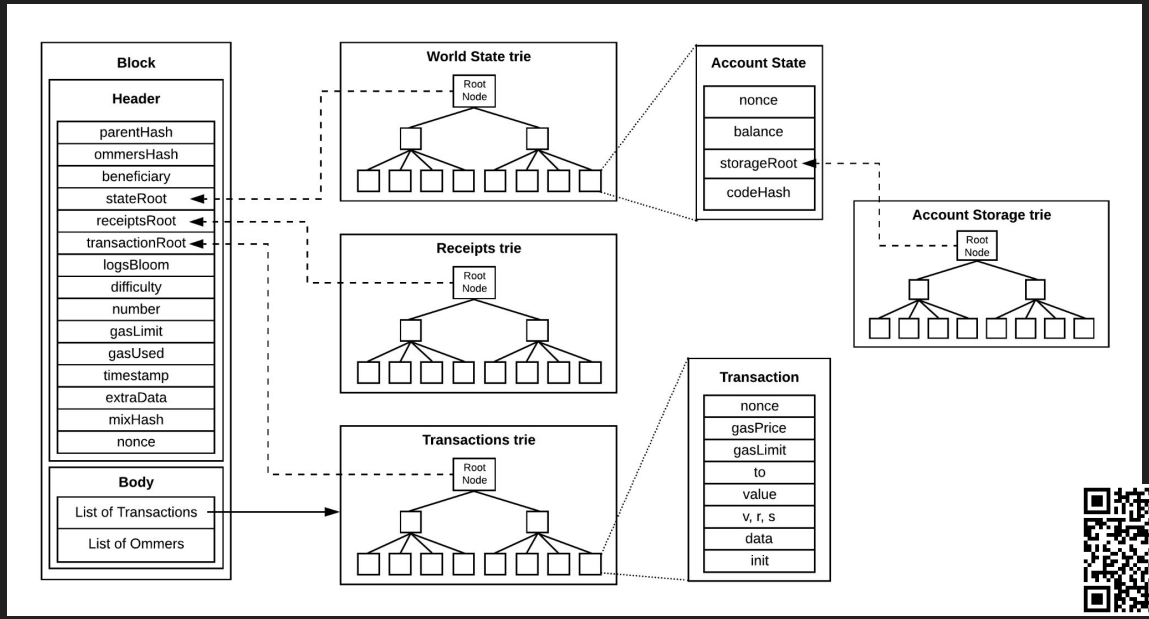
Cryptography used in Verkle Trees

- Group:
  - EC Bandersnatch (Banderwagon, remove cofactor)
  - Scalar field (Fr) = 253 bits
  - Base field (Fp) = 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!)
- Multiproof: aggregate multiple vector openings in a single vector opening!

Data structure

Data structure

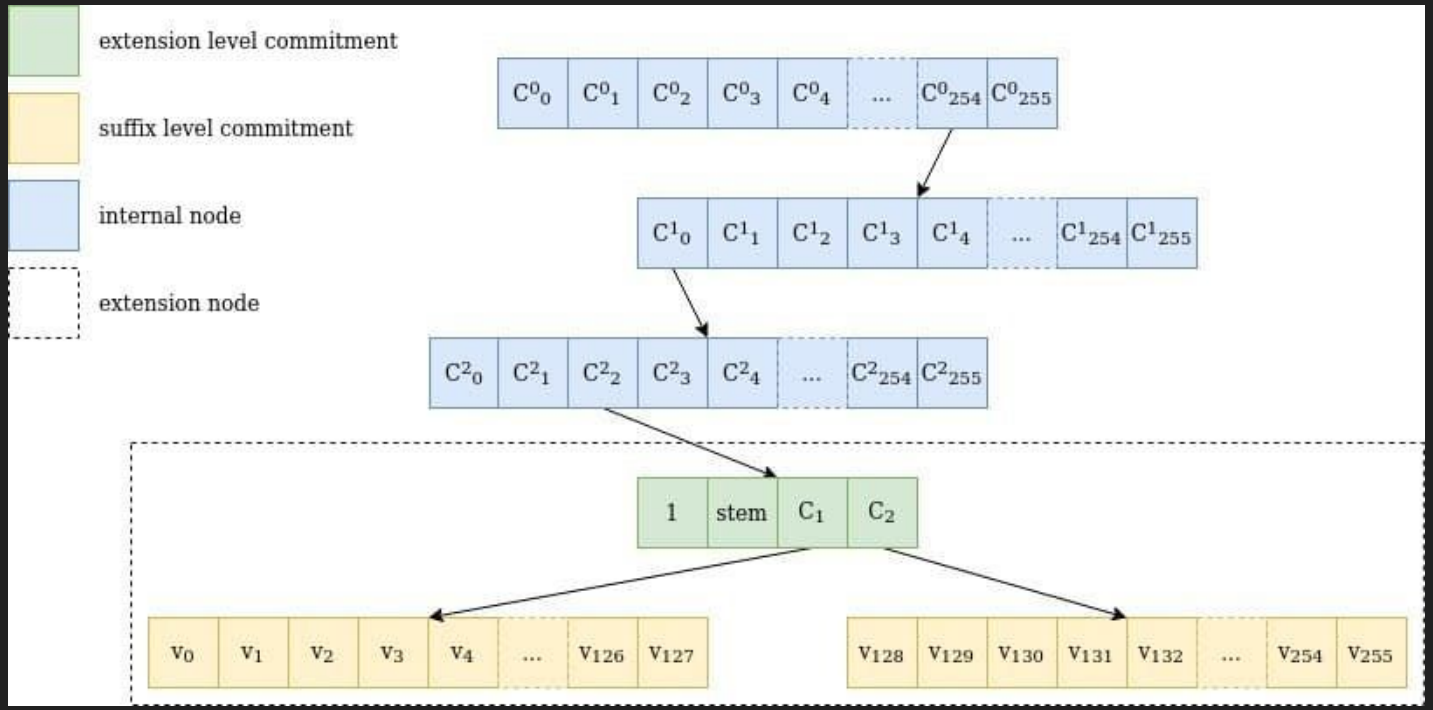
- Merkle Patricia Tree -> Verkle Tree
- Verkle Tree = Vector Commitment + Merkle Tree
- Today:



Data structure (EIP-6800)

blog.ethereum.org/2021/12/02/verkle-tree-structure

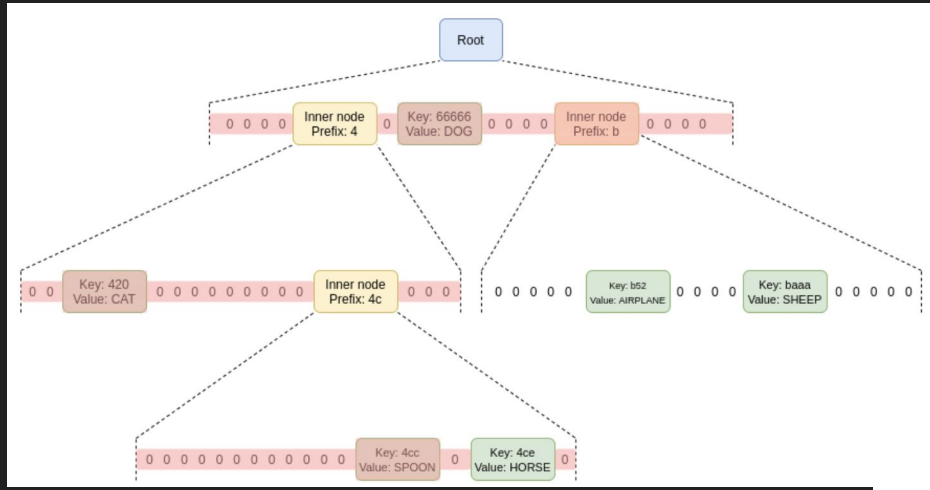
- Future:

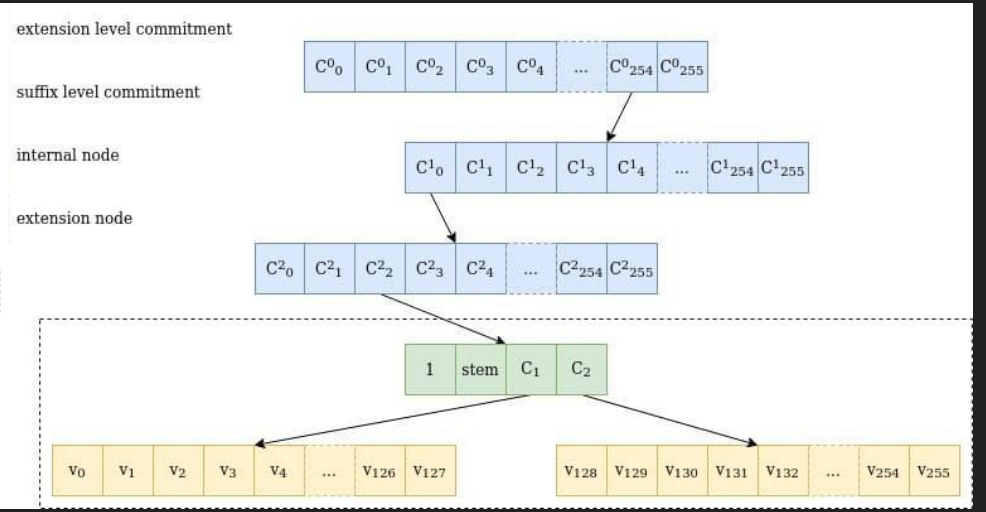


$$C_1 = \text{Commit}(v_0^{(\text{lower,modified})}, v_0^{(\text{upper})}, v_1^{(\text{lower,modified})}, v_1^{(\text{upper})}, \ldots, v_{127}^{(\text{lower,modified})}, v_{127}^{(\text{upper})})$$

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

Proving: MPT vs VKT

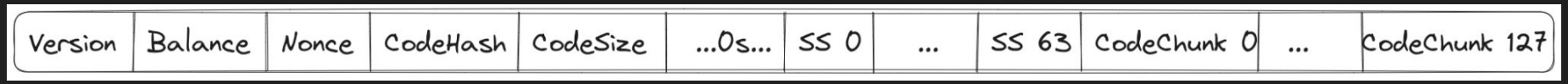




https://vitalik.eth.limo/general/2021/06/18/verkle.html

How to store information in leaves? (EIP-6800)

- Leaf nodes: store 256 (scalar field) values
- Account header:



- Further storage slots:

- Let’s explain better what *CodeChunk X* means…

How to store information in leaves? (EIP-6800)

- Contracts code is also in the tree!
- Code is “chunked”:
  - A code-chunk is a 32-byte value
  - Chunk[0] = How many bytes are a continuation of a PUSHX instruction of prev chunk
  - Chunk[1:32] = Bytecode[0:31]
- Remember leaves are 256 values of 32 bytes:

How tree keys are calculated? (EIP-6800)

- Stop using Keccak! (not zk-friendly)
- Use a 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 tree keys are calculated? (EIP-6800)

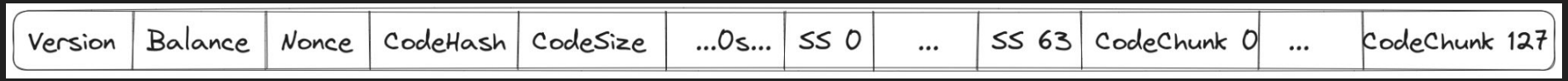
- Stop using Keccak! (not zk-friendly)
- Use a EC based hashing function (i.e: Pedersen hash):

TreeKey(address, treeIndex, subIndex) = Commit(2+256*64, address[0:16], address[16:32], treeIndex[0:16], treeIndex[16:32])[0:31] ++ subIndex

- Account info: TreeKey(address, 0, [Version | Balance | Nonce | …])
  - 0 = Version
  - 1 = Balance
  - 2 = Nonce
  - 3 = CodeHash
  - 4 = CodeSize

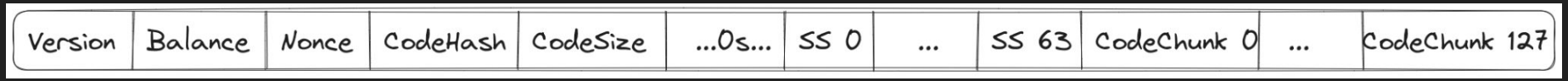
How tree keys are calculated? (EIP-6800)

- Remember the account header had some storage slots and code chunks:



How tree keys are calculated? (EIP-6800)

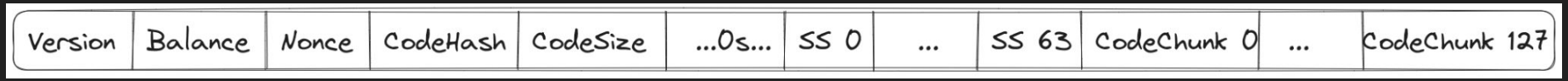
- Remember the account header had some storage slots and code chunks:



- The rest of storage slots:
  - pos = MAIN_STORAGE_OFFSET + storage_key
  - TreeKey(address, pos / VERKLE_NODE_WIDTH, pos % VERKLE_NODE_WIDTH)

How tree keys are calculated? (EIP-6800)

- Remember the account header had some storage slots and code chunks:



- The rest of storage slots:
  - pos = MAIN_STORAGE_OFFSET + storage_key
  - TreeKey(address, pos / VERKLE_NODE_WIDTH, pos % VERKLE_NODE_WIDTH)
- The rest of code chunks:
  - chunk_id = CODE_OFFSET + chunk_id
  - TreeKey(address, chunk_id / VERKLE_NODE_WIDTH, chunk_id % VERKLE_NODE_WIDTH)

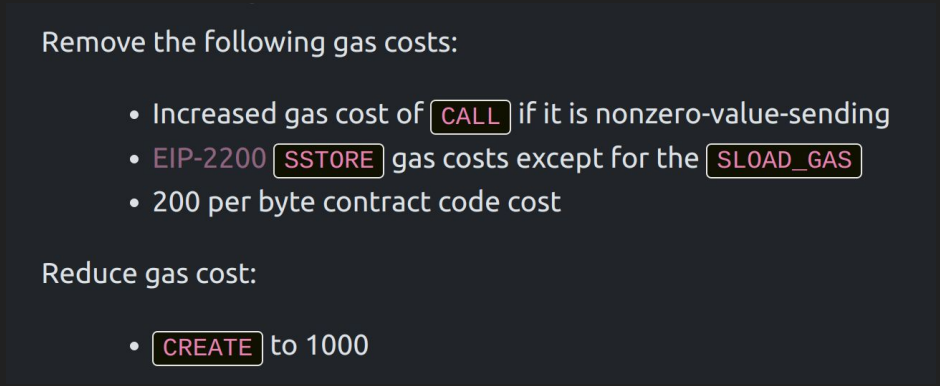
Gas accounting

Gas accounting (EIP-4762)

- In stateless, IO 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!

Gas accounting (EIP-4762)

- 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)



Gas accounting (EIP-4762)

- 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 (!!)

Gas accounting (EIP-4762)

- 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)
- Notes:
  - These charges only happen once per key per transaction.
  - Any further storage slot access charges “warm access” cost (100)

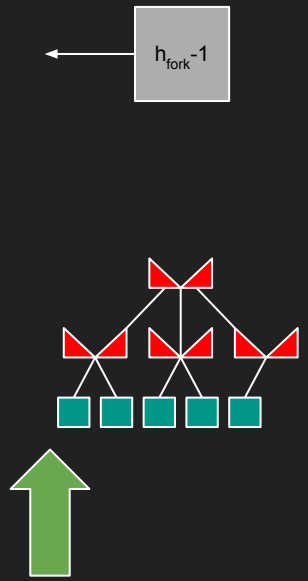

NEW CRYPTO, NEW TREE, NEW
GAS ACCOUNTING... WE'RE DONE, RIGHT?
imgflip.com

State conversion

State conversion

- Need to migrate all the state from MPT -> VKT
- Multiple strategies have been considered
- Currently the “Overlay tree” is the proposed one

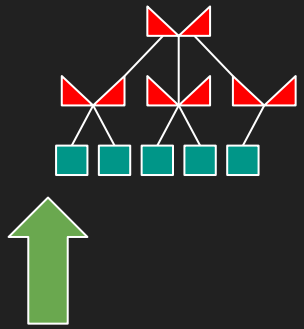
Freeze the tree at block h_{fork}-1



Pre-fork values

Post-fork values

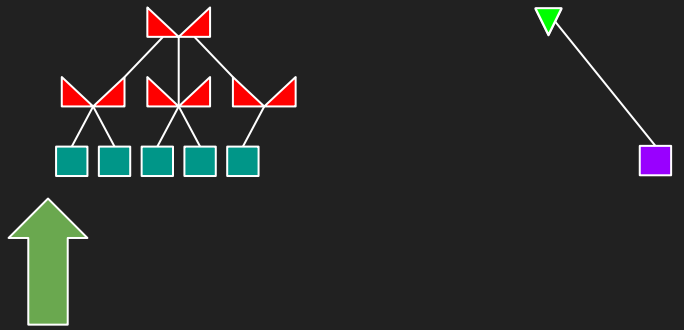
Start with a fresh root



Pre-fork values

Post-fork values

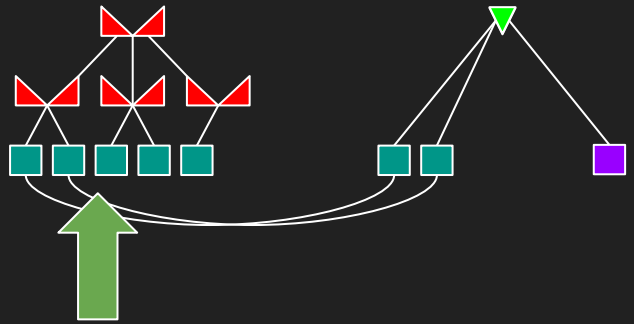
New writes go into the verkle tree



Pre-fork values

Post-fork values

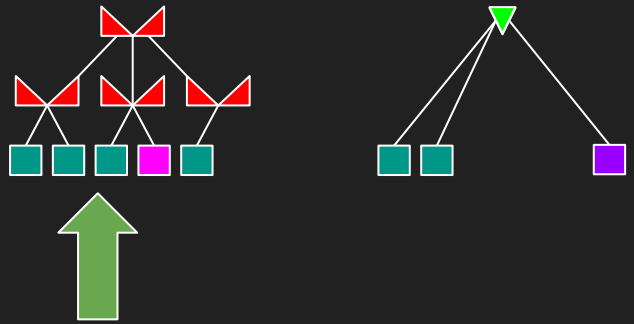
And N leaves are also converted into the verkle tree



Pre-fork values

Post-fork values

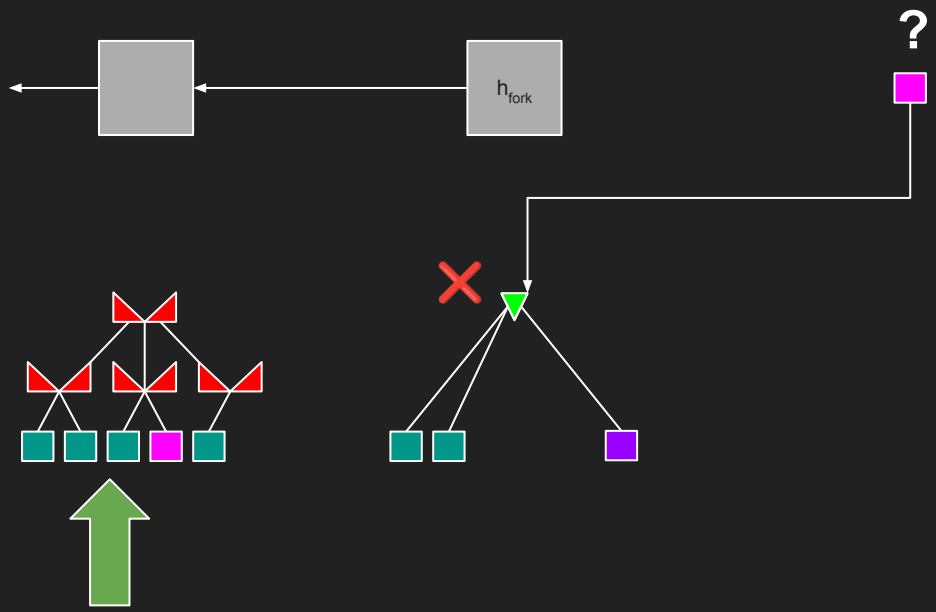
Accessing data



Pre-fork values

Post-fork values

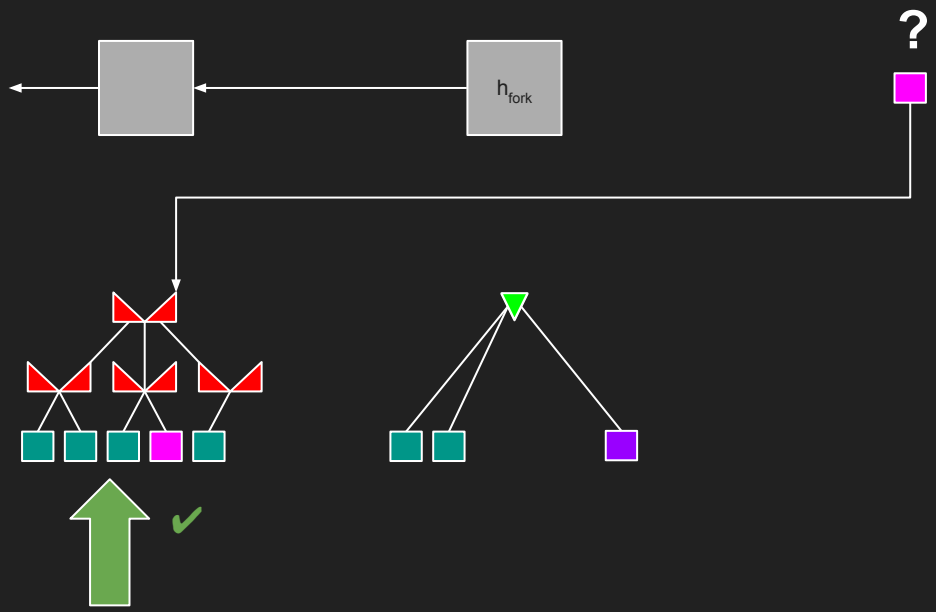
Accessing data : first, attempt reading from the verkle tree



Pre-fork values

Post-fork values

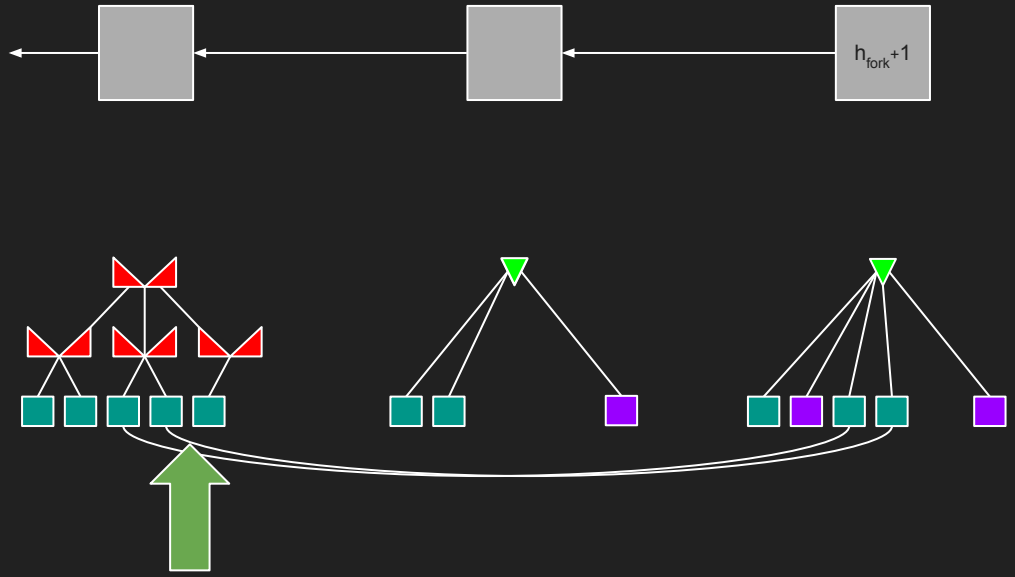
Accessing data : if not present, read from the MPT



Pre-fork values

Post-fork values

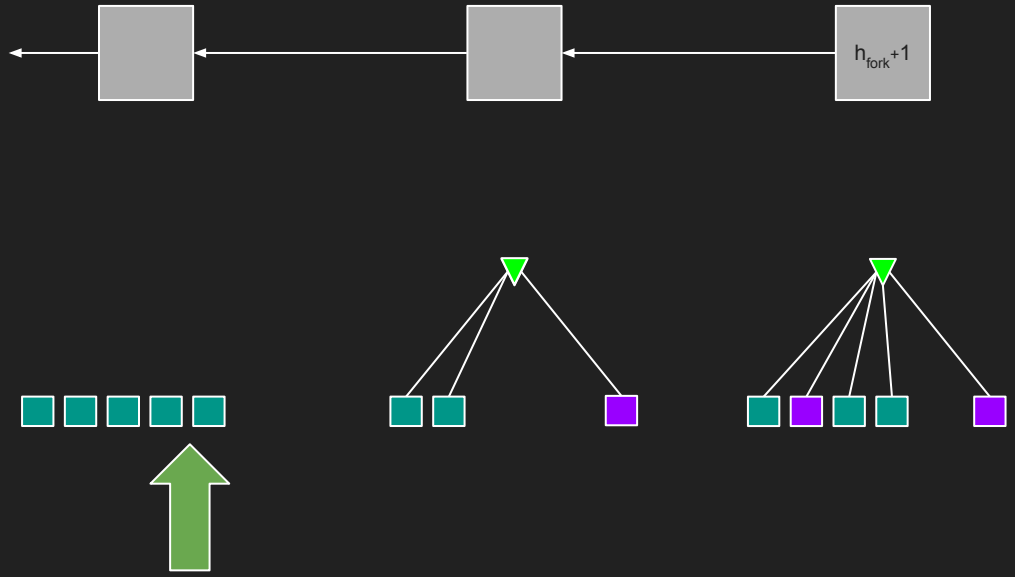
The same process repeats on the next block



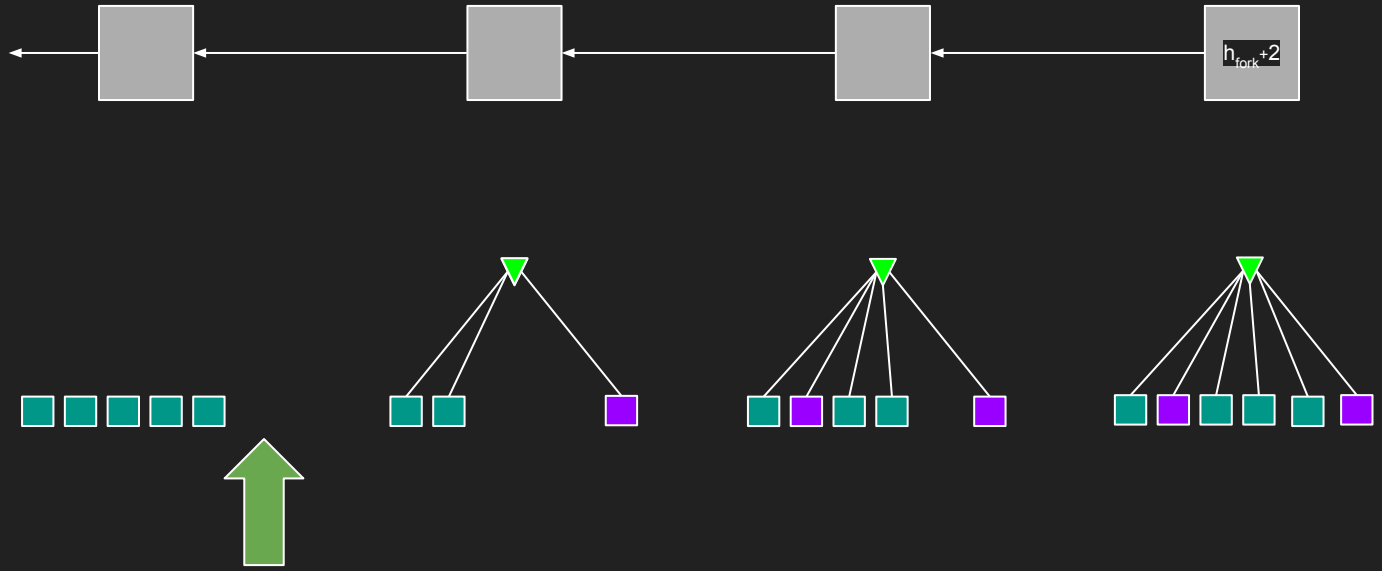
Pre-fork values

Post-fork values

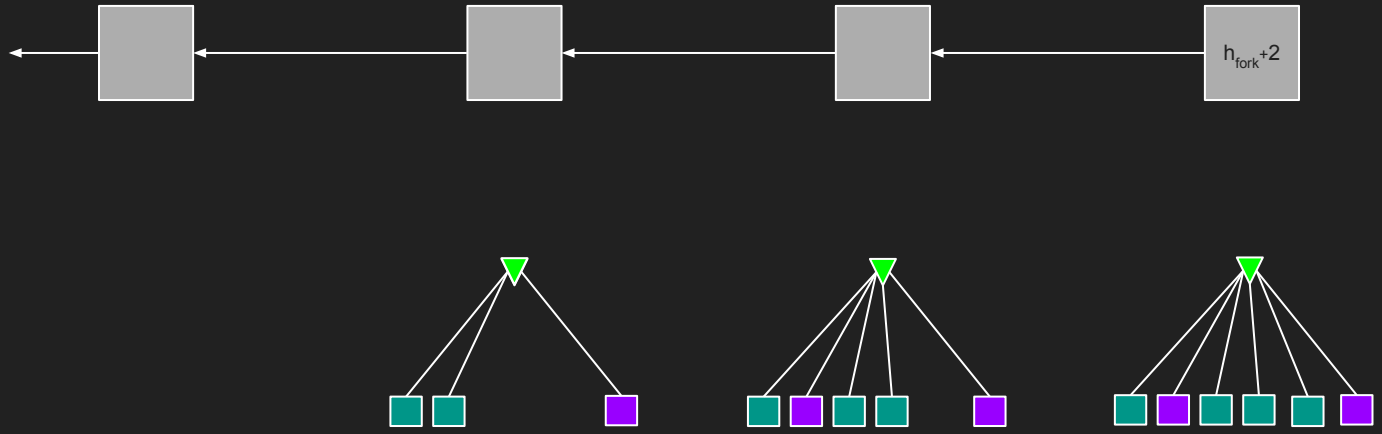
Delete internal MPT nodes when the block is finalized



Eventually, all the leaves have been converted



So the MPT data can be forgotten

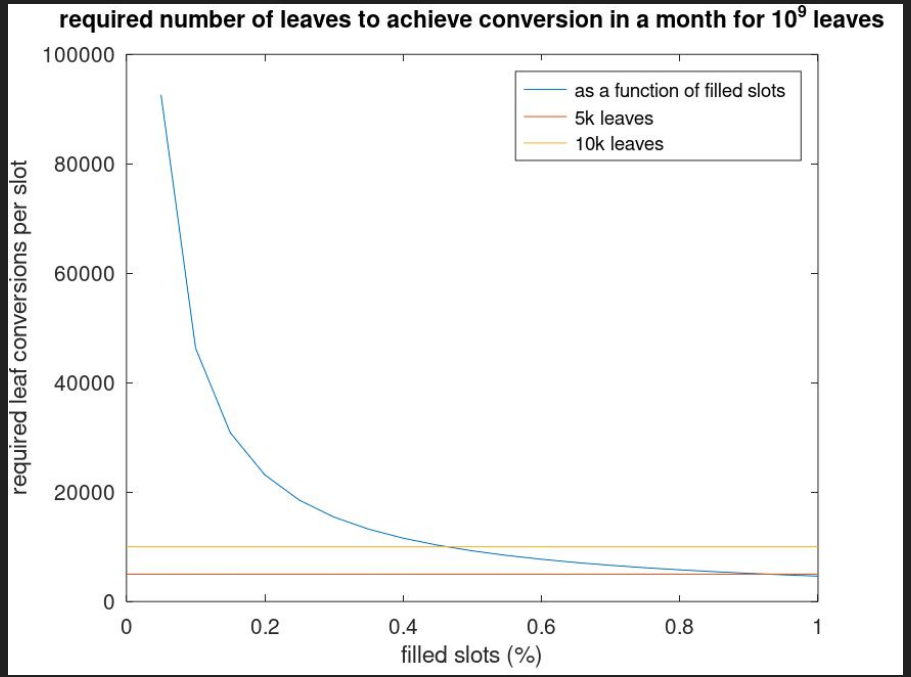


Pre-fork values

Post-fork values

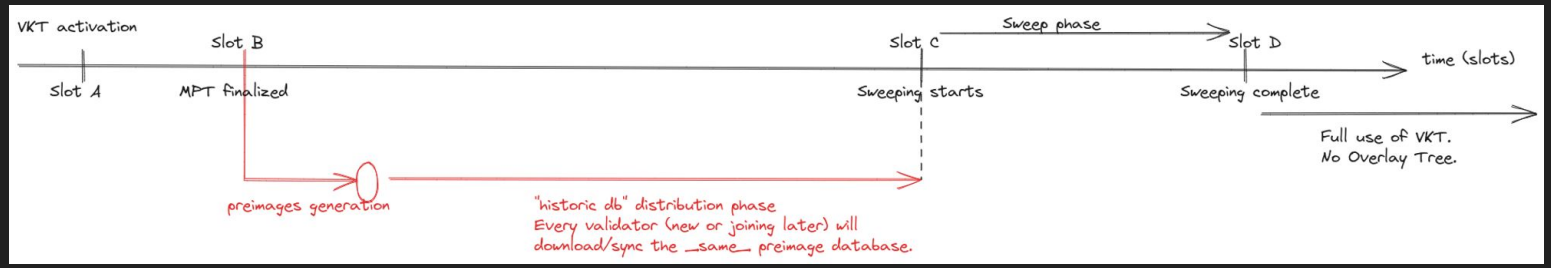
The overlay method : timeline

- N = 1K => 6 months
- N = 5K => 1 month
- N = 10K => 15 days



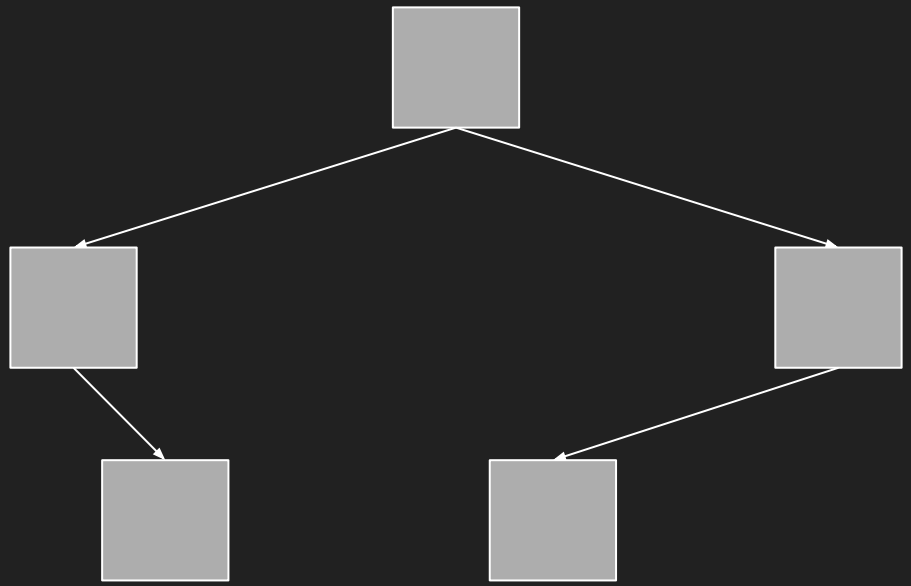
More challenges

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

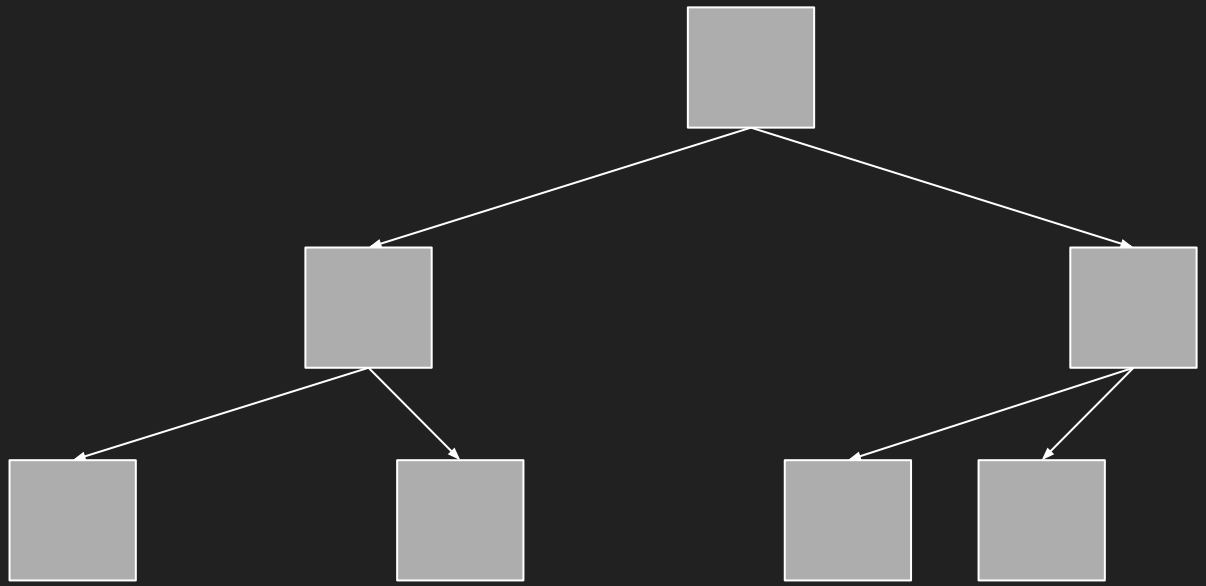


hackmd.io/@jsign/vkt-preimage-generation-and-distribution

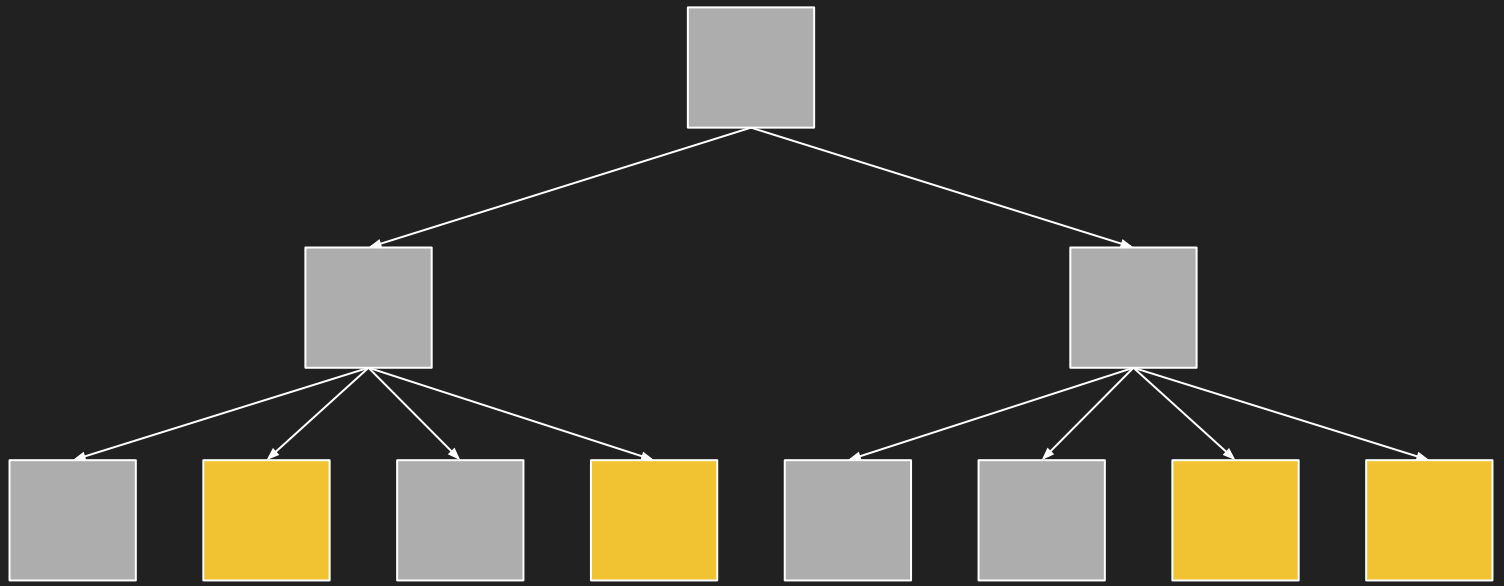
Verkle sync: view at block n



Verkle sync : view at block n+1



Verkle sync: backfill data



Join the efforts!

- Biweekly calls
- #verkle-trie-migration in Ethereum R&D discord

Thanks!