Vividh Program Research

Stateless Ethereum

Source: <https://www.zeroknowledge.fm/156>

Type: Podcast

[Vector commitments](https://eprint.iacr.org/2011/495.pdf) is a type of authenticated Data Structure. It represents a vector as a succinct commitment. Vector commitments can be used for stateless validation.

In Ethereum, state is a vector that maps a position i to the public key of user[i] and balance of user[i]. In current Ethereum,all miners store this vector which is various GBs long. Miners can easily verify the transactions using this vector and update state easily in this vector.

In Stateless Ethereum, the miners don't need to store this long vector but store the succinct commitment of vectors. The transactions can be validated against this vector given that a user provides a witness with the transaction. For example, if user[i] sends ether to user[j], user[i] shall send proof of balance along with the transaction. The miner can verify the transaction against the vector commitment present with him and validate the transaction.

Further Reading:

1. Vector Commitments
2. Accumulators
3. Authenticated Dictionary
4. Authenticated Data Structures.
5. Reading and understanding state in current Ethereum and how it is represented.
6. Understand Cryptography in depth.\*\*

Source : <https://ethresear.ch/t/the-stateless-client-concept/172>

Type: Blog at ETH R&D

STF(S, B) -> S’

S -> State root of S (i.e. 32-bit hash of Merkle Patricia tree containing S).

B -> (B, W) where W is “witness” B is Block or Transaction.

STF -> STF’ which takes as input a state root and a block-plus-witness, uses the witness as a “database” any time the execution of the block needs to read any accounts, storage keys or other state data and output new state root.

That is, full nodes would only store state roots, and it would be miners’ responsibility to package Merkle branches (“witnesses'') along with the blocks, and full nodes would download and verify these expanded blocks. It’s entirely possible for stateless full nodes and regular full nodes to exist alongside each other in a network; you could have translator nodes that take a block B, attach the required witness, and broadcast (B, W) on a different network protocol that stateless nodes live on; if a miner mines a block on this stateless network, then the witness can simply be stripped off, and the block re-broadcasted on the regular network.

The simplest way to conceive the witness in a real protocol is to view it as an RLP-encoded list of objects, which could then be parsed by the client into a {sha3(x): x} key-value map; this map can then simply be plugged into an existing ethereum implementation as a “database”.

Inference :

1. The miners would still have to store the complete state of Blockchain.
2. This method can provide fast sync.

Source: [Modified Merkle Patricia Trie — How Ethereum saves a state](https://medium.com/codechain/modified-merkle-patricia-trie-how-ethereum-saves-a-state-e6d7555078dd)

Type: Medium Blog

1. Patricia Tree:

Patricia trie is a data structure which is also called Prefix tree, radix tree or trie. Trie uses a key as a path so the nodes that share the same prefix can also share the same path. This structure is fastest at finding common prefixes, simple to implement, and requires small memory. Thereby, it is commonly used for implementing routing tables, systems that are used in low specification machines like the router.

1. Merkle Tree:

Merkle tree is a tree of hashes. Leaf nodes store data. Parent nodes contain their children’s hash as well as the hashed value of the sum of their children’s hashes. Since all the nodes except for leaf nodes contain a hash, the Merkle tree is also known as a hash tree.

Source: Cryptography Course on Coursera.

Type: Course.

Timeline:

Enrolled: 05-12-2020

Week 1 : 08-12-2020

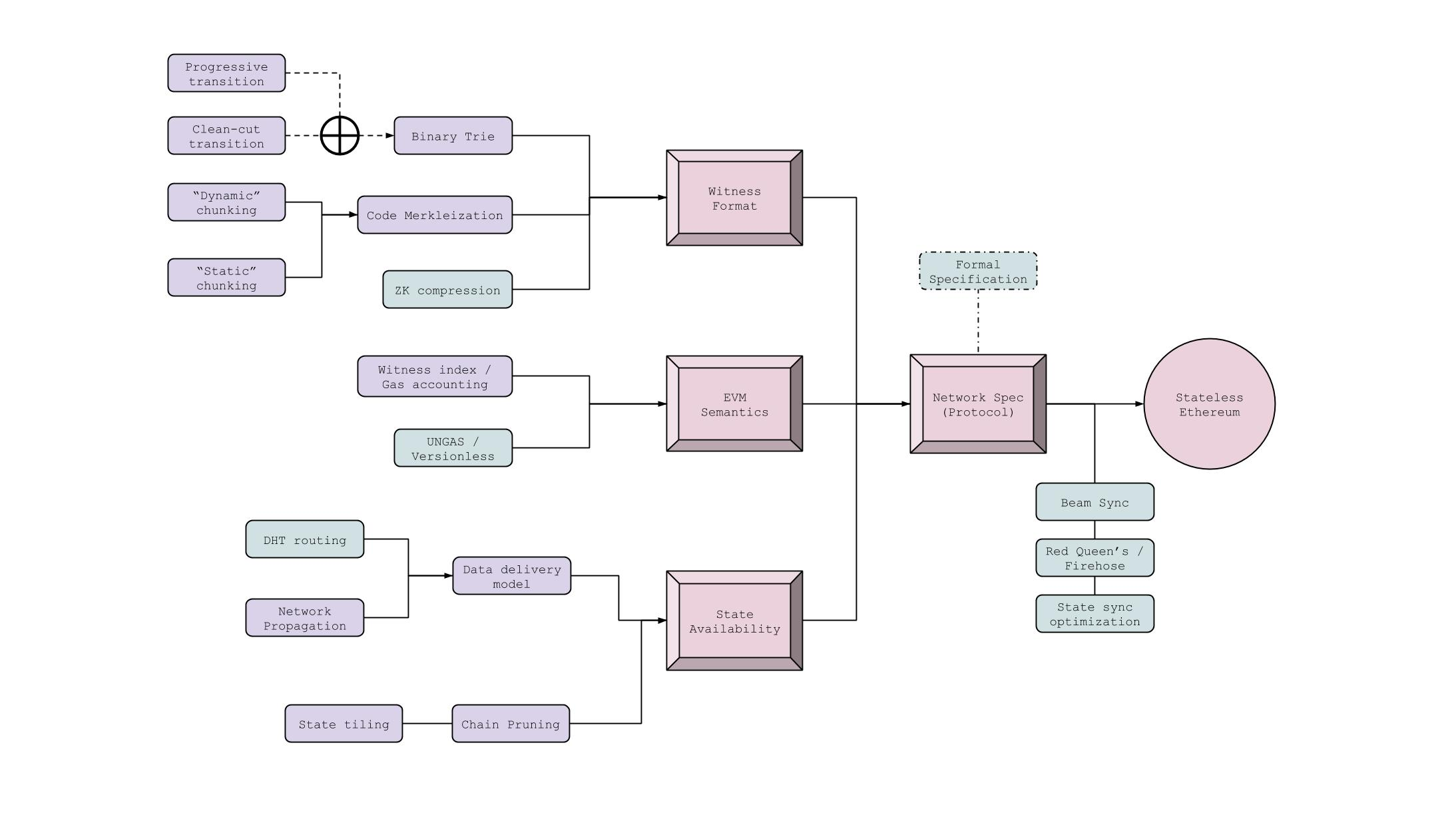
Week 2: 13-12-2020

Week 3: 21-12-2020

Week 4: 24-12-2020

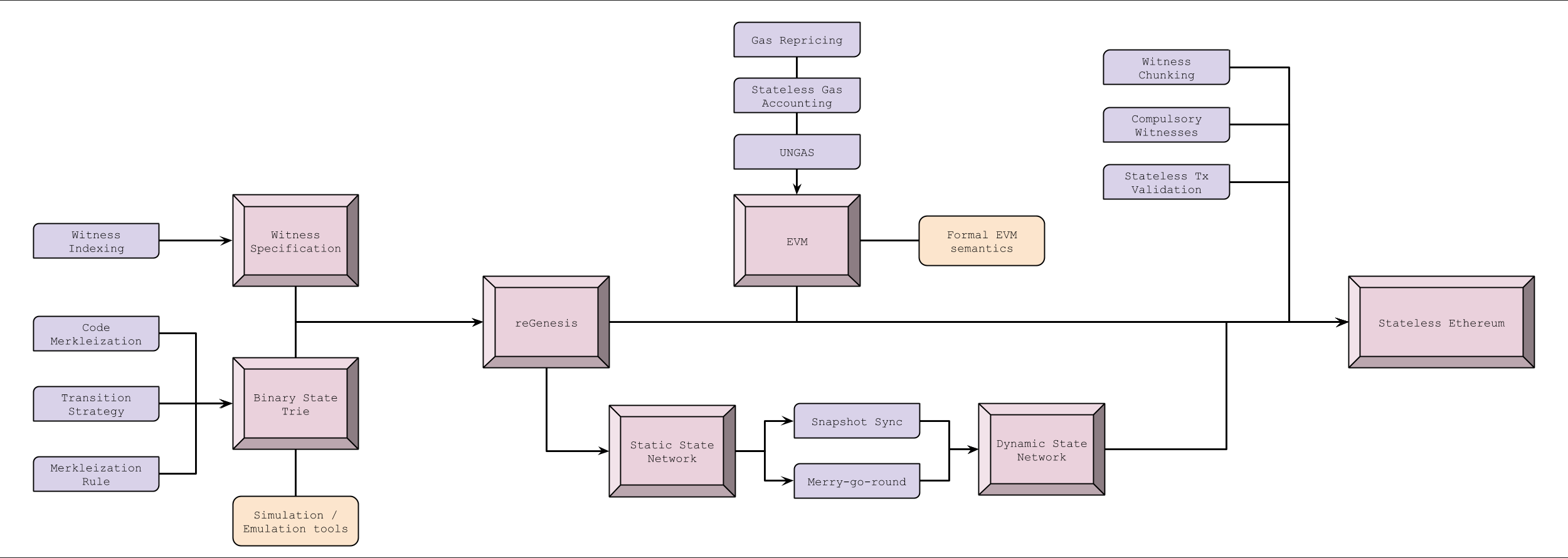
Source: [The 1.x Files: The Stateless Ethereum Tech Tree](https://blog.ethereum.org/2020/01/28/eth1x-files-the-stateless-ethereum-tech-tree/)

Type: Blog



Source: [The Updated Stateless Ethereum Tech Tree](https://blog.ethereum.org/2020/04/02/eth1x-stateless-tech-tree/)

Type: Blog



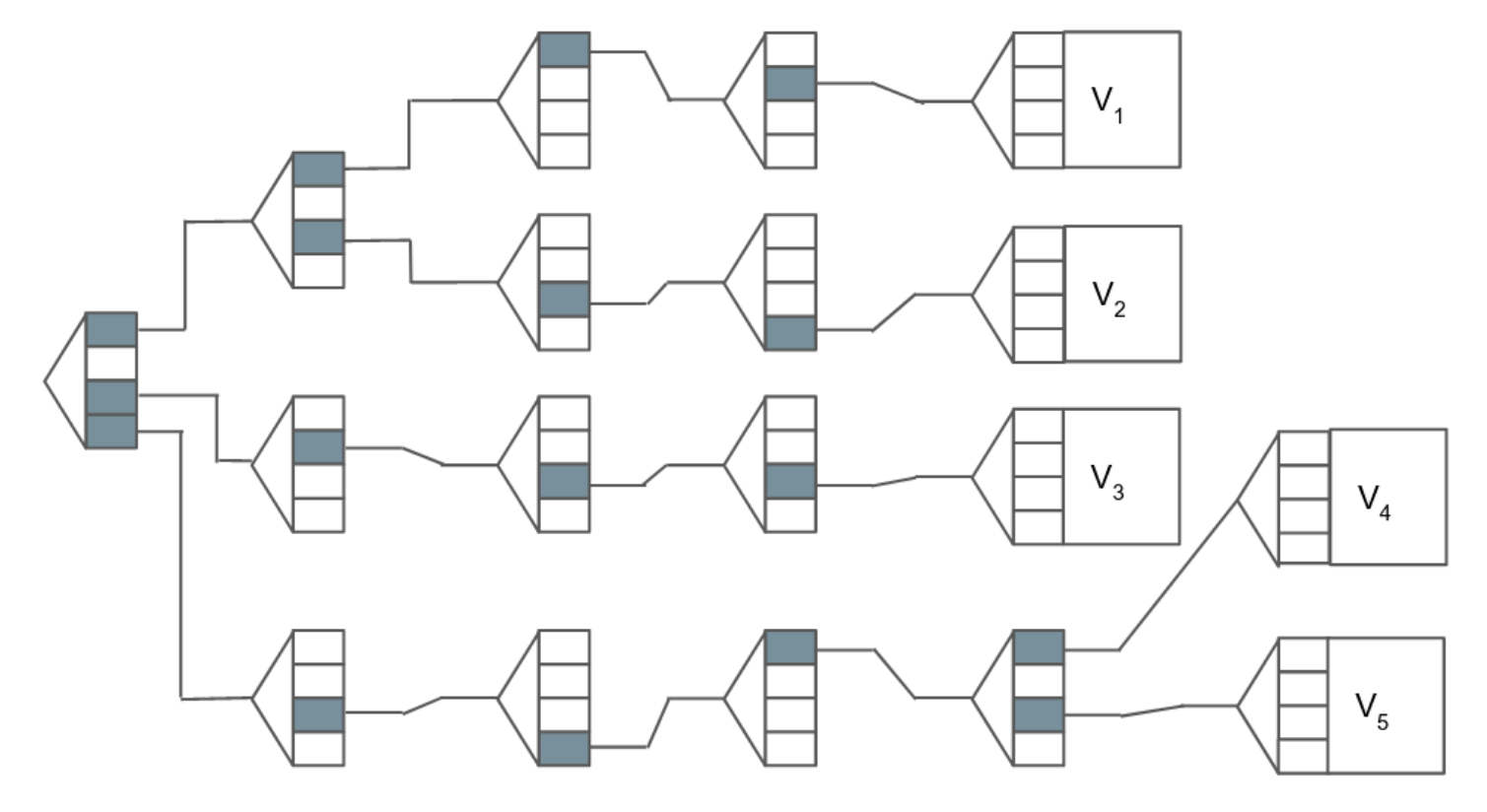
Source: [The 1.x Files: The State of Stateless Ethereum](https://blog.ethereum.org/2019/12/30/eth1x-files-state-of-stateless-ethereum/)

Type: Blog

The complete ‘state’ of Ethereum describes current balances of all accounts and smart contracts deployed and running on EVM. A specific block always has the same state agreed upon by participants of the network. The state of Ethereum is updated when a new block is added to the Blockchain.

Merkle Patricia trie is used in Ethereum to represent state.

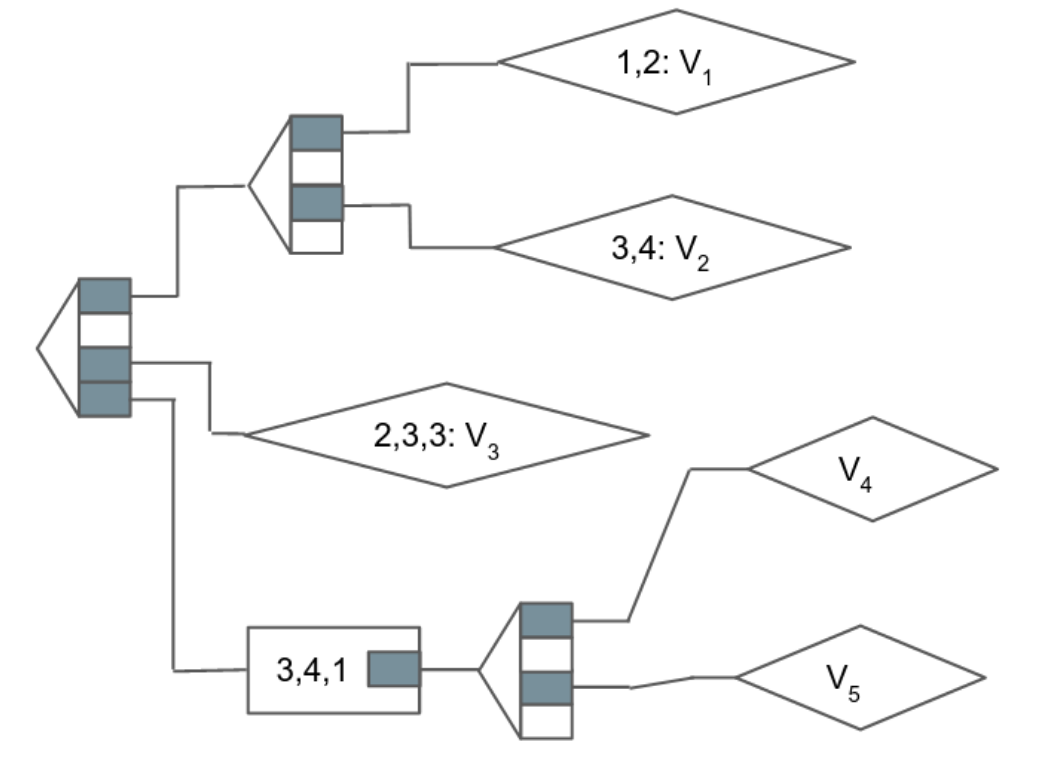
At one end, there are pieces of data (right side) that describe state (value nodes). This could be account balance or variable stored in smart contract. In middle, there are Branch nodes which links values through hashing. A branch node is an array containing the hashes of its child nodes, and each branch node is subsequently hashed and put into the array of its parent node. This successive hashing eventually arrives at a single state root node on the other end of the trie.



In simplified diagram above, the path followed to reach

1. V1 => 1, 1, 1, 2
2. V2 => 1, 3, 3, 4
3. V3 => 3, 2, 3, 3
4. V4 => 4, 3, 4, 1, 1
5. V5 => 4, 3, 4, 1, 3

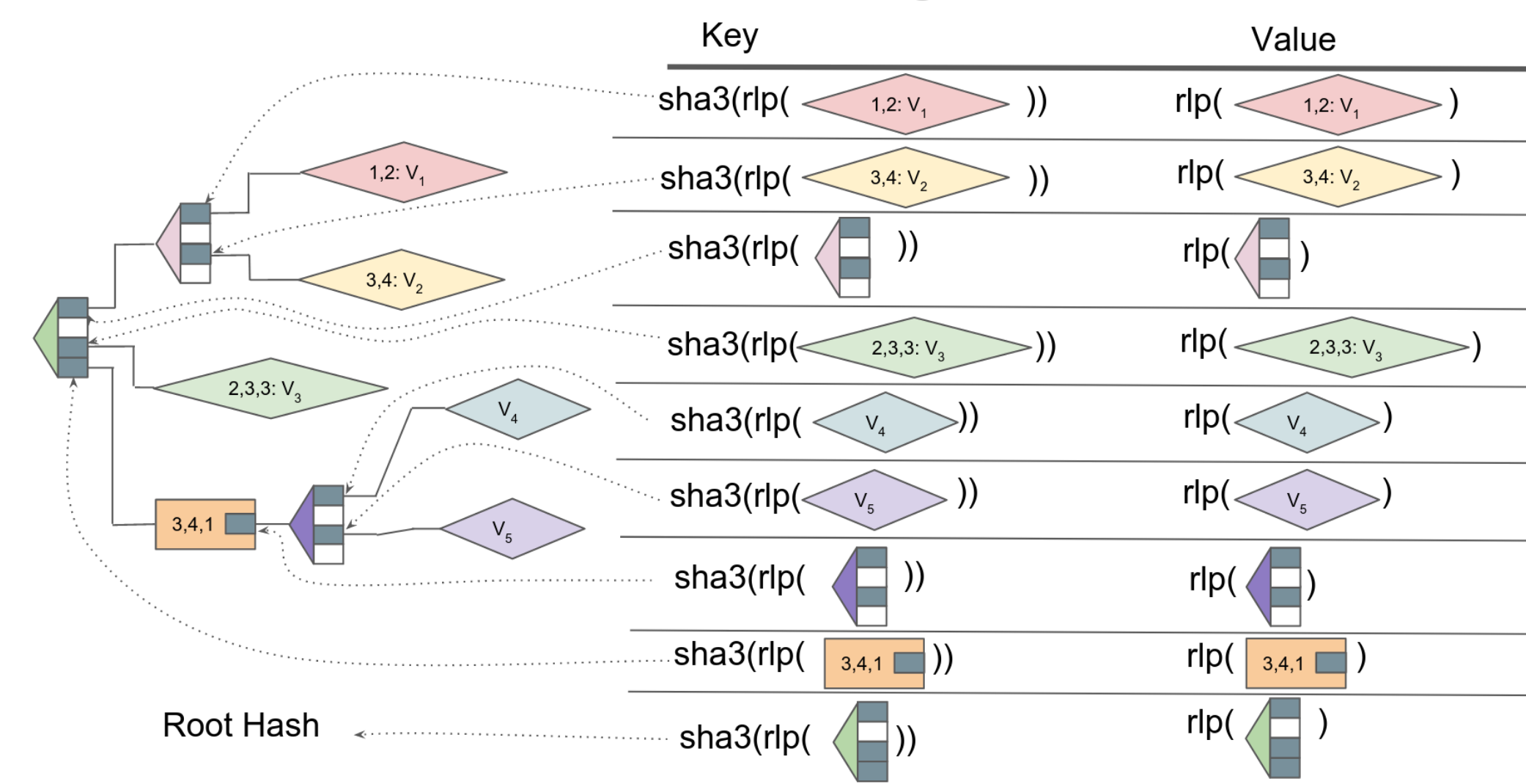
The Structure is deterministic and Cryptographically verifiable. The only way to generate a state root is by computing it from each individual piece of the state, and two states that are identical can be easily proven so by comparing the root hash and the hashes that led to it (a Merkle proof). Conversely, there is no way to create two different states with the same root hash, and any attempt to modify state with different values will result in a different state root hash. Ethereum optimizes the trie structure by introducing a few new node types that improve efficiency: extension nodes and leaf nodes. These encode parts of the path into nodes so that the trie is more compact.



To arrive at a particular part of state (such as an account’s current balance of Ether), one needs to start at the state root and crawl along the trie from node to node until the desired value is reached.

In the ‘real’ version used by Ethereum, paths are the hashes of an address 64 characters (256 bits) in length, and values are RLP-encoded data. Branch nodes are arrays that contain 17 elements (sixteen for each of the possible hexadecimal characters, and one for a value), while leaf nodes and extension nodes contain 2 elements (one partial path and either a value or the hash of the next child node).

At this point we should remind ourselves that the trie structure is just an abstract concept. It’s a way of packing the totality of Ethereum state into one unified structure. That structure, however, then needs to be implemented in the code of the client, and stored on a disk (or a few thousand of them scattered around the globe). This means taking a multi-dimensional trie and stuffing it into an ordinary database, which understands only [key, value] pairs. In most Ethereum clients (all except turbo-geth), the Merkle-Patricia Trie is implemented by creating a distinct [key, value] pair for each node, where the value is the node itself, and the key is the hash of that node.



The process of traversing the trie however remains the same as the theoretical process described.

The process of traversing the trie, then, is more or less the same as the theoretical process described earlier. To look up an account balance, we would start with the root hash, and look up its value in the database to get the first branch node. Using the first character of our hashed address, we find the hash of the first node. We look that hash up in the database, and get our second node. Using the next character of the hashed address, we find the hash of the third node. If we’re lucky, we might find an extension or leaf node along the way, and not need to go through all 64 nibbles – but eventually, we’ll arrive at our desired account, and be able to retrieve its balance from the database.

Computing the hash of each new block is largely the same process, but in reverse: Starting with all the edge nodes (accounts), the trie is built through successive hashings, until finally a new root hash is built and compared with the last agreed-upon block in the chain.

Here’s where that bit about the apparent efficiency of the state trie comes into play: re-building the whole trie is very intensive on disk, and the modified Merkle-Patricia trie structure used by Ethereum is more protocol efficient at the cost of implementation efficiency. Those extra node types, leaf and extension, theoretically save on memory needed to store the trie, but they make the algorithms that modify the state inside the regular database more complex. Of course, a decently powerful computer can perform the process at blazing speed. Sheer processing power, however, only goes so far.

So far we’ve limited our scope to what’s going on in an individual computer running an Ethereum implementation like geth. But Ethereum is a network,and it must have the same unified state consistent across thousands of computers worldwide, and between different implementations of the protocol.

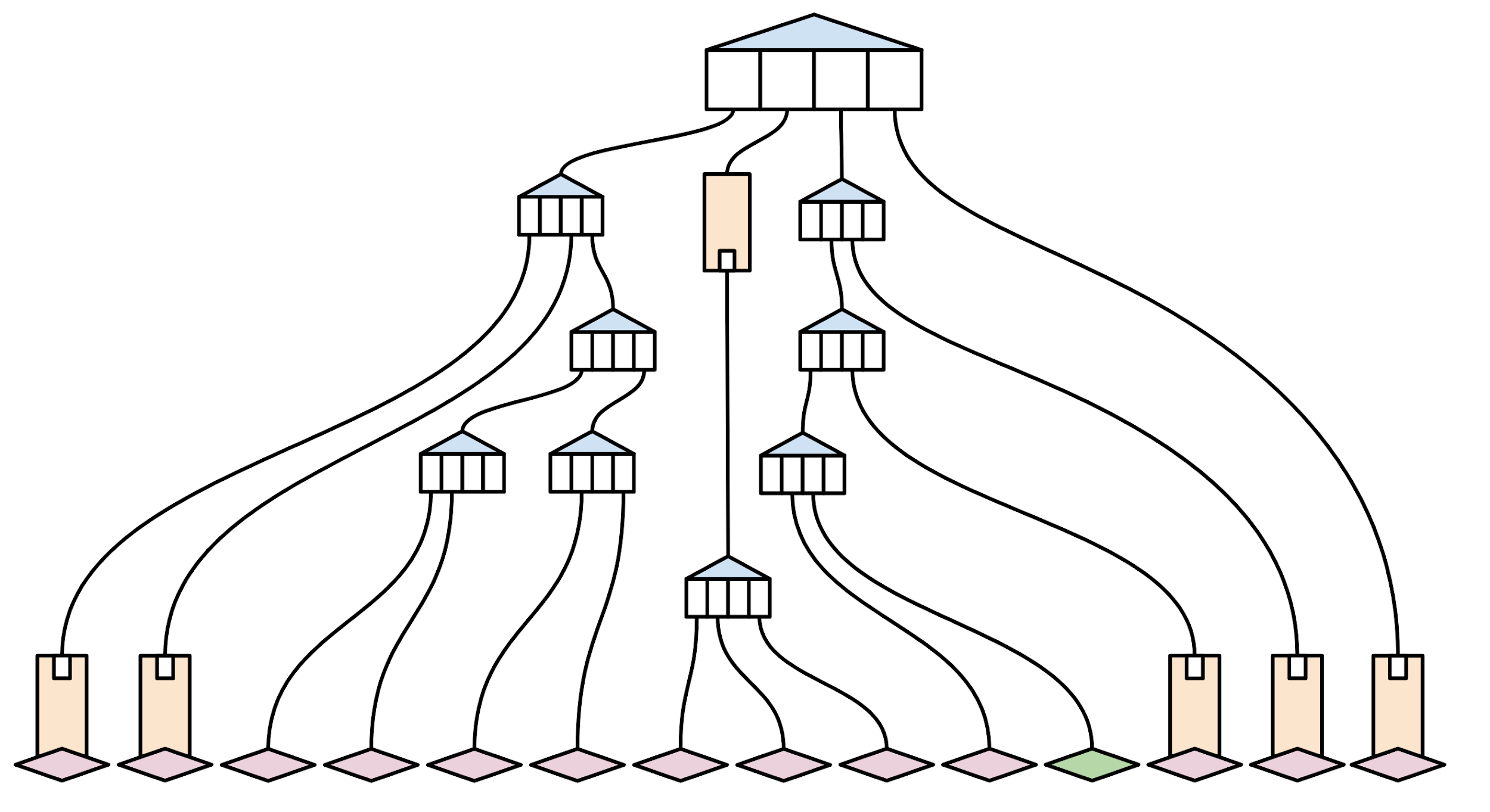
The state of Ethereum changes with addition of each block.

Methods to sync a node:

1. Full-sync :
   1. Starting from genesis block, a list of every transaction in each block is retrieved and a state trie is built.
   2. With each block, the state trie is updated and the entire history of the blockchain is replayed.
   3. It takes about a week to download and execute every state change from the beginning.
2. Fast-sync:
   1. A new client can request transactions from state entries from a recent trusted ‘checkpoint’ block.
   2. It’s far less total information to download, but it is still a lot of information to process– sync is not currently limited by bandwidth, but by disk performance.

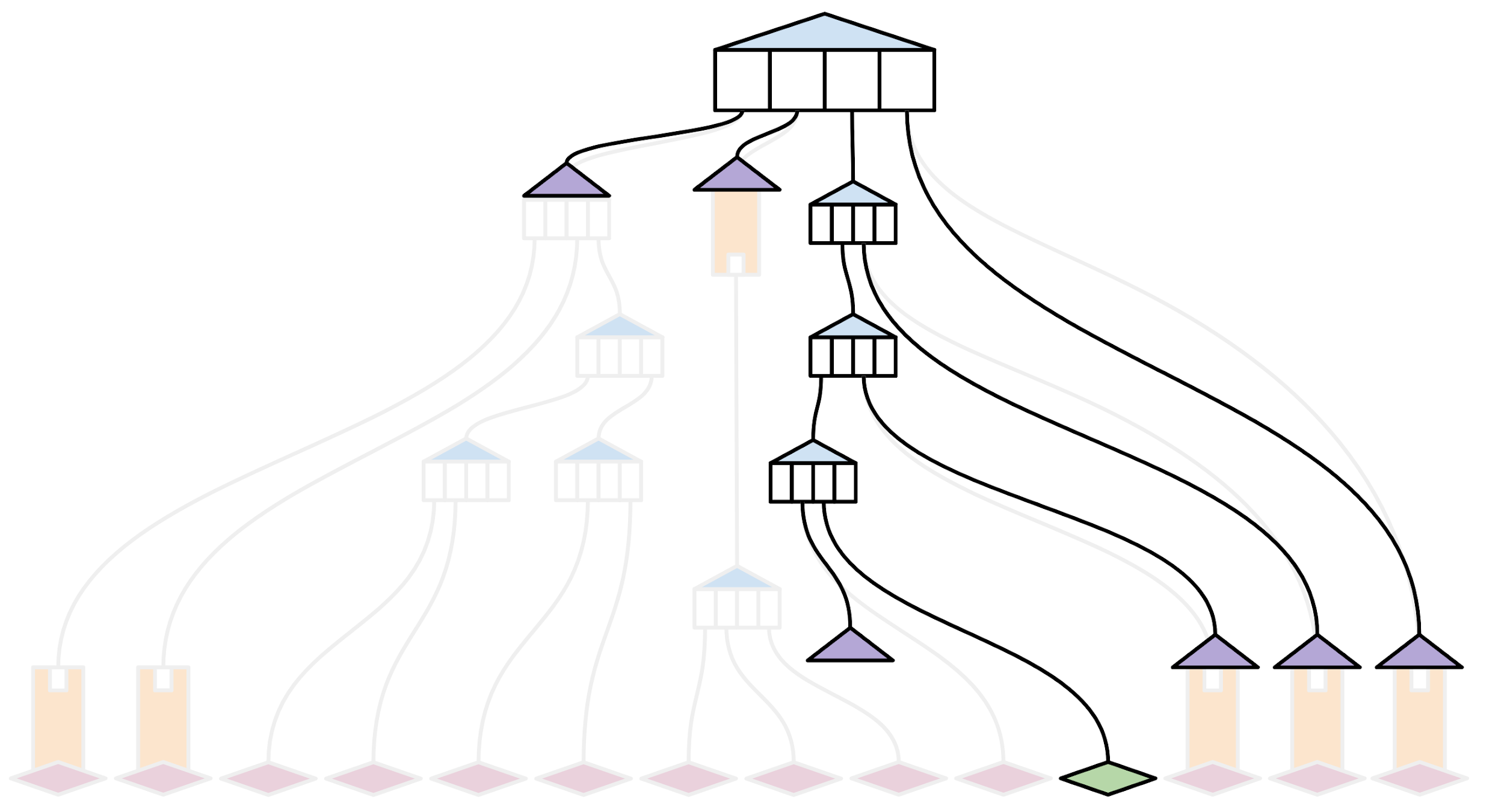
One of the main goals of stateless Ethereum is to make new nodes less painful to spin up. As there is almost 0.1% state change when a block is added, there shall be some means to prevent downloading extra data that is required for full sync switchover. But this is one of the challenges imposed by Ethereum’s cryptographically secure data structure: In a trie, a change to just one value will result in a completely different root hash. That’s a feature, not a bug! It keeps everybody certain that they are on the same page (at the same state) with everyone else on the network.

Suppose that just one value in this trie has changed recently (highlighted in green):



The full node syncing the state would take the pieces of state and hash them together to create a new hash and verify this hash with other nodes to verify that their state is the same.

For someone who has just tuned in, it will need older nodes to provide proof that observed transactions fit in with everything they have seen in state.



In very abstract terms, a block witness proof provides all of the missing hashes in a state trie, combined with some ‘structural’ information about where in the trie those hashes belong. This allows an ‘oblivious’ node to include the new transaction in its state, and to compute the new root hash locally – without requiring them to download an entire copy of the state trie.

In contrast to beam sync, a truly stateless client would never keep a copy of state; it would only grab the latest transactions together with the witness, and have everything it needs to execute the next block.

If the entire network were stateless, this could actually hold up forever, witness for new blocks would be generated from previous blocks.

* **Full-state nodes** would operate as before, but would additionally compute a witness and either attach it to a new block, or propagate it through a secondary network sub-protocol.
* **Partial-state nodes** could keep a full state for just a short number of blocks, or perhaps just ‘watch’ the piece of state that they’re interested in, and get the rest of the data that they need to verify blocks from witnesses. This would help infrastructure-running dapp developers immensely.
* **Zero-state nodes**, who by definition want to keep their clients running as light as possible, could rely entirely on witnesses to verify new blocks.

The elephant in the research room is witness size. Ordinary blocks contain a header, and a list of transactions, and are on the order of 100 kB. This is small enough to make the propagation of blocks quick relative to network latency and the 15 second block time.

Witnesses, however, need to contain the hashes of nodes both at the edges and deep inside the state trie. This means they are much, much bigger: early numbers suggest on the order of 1 MB. Consequently, syncing a witness is much much slower relative to network latency and block time, which could be a problem.

One line of inquiry is to think about ways to compress and reduce the size of witnesses by changing the structure of the trie itself (such as a binary trie), to make it more efficient at the implementation level. Another is to prototype the network primitives (bittorrent-style swarming) that allow witnesses to be efficiently passed around between different nodes on the network.

Inferences:

1. How is the state represented in Ethereum?
2. Learn about Merkle Proof.
3. RLP -Encoding
4. Beam sync

Source: [The 1.x Files: A Primer for the Witness Specification](https://blog.ethereum.org/2020/05/04/eth1x-witness-primer/)

Type: Blog

A witness is a bit like a cheat sheet for an oblivious (stateless) student (client). It’s just the minimum amount of information need to pass the exam (submit a valid change of state for inclusion in the next block). Instead of reading the whole textbook (keeping a copy of the current state), the oblivious student (stateless client) asks a friend (full node) for a crib sheet to submit their answers.

Inferences:

1. Backus-Naur form

Source: [The 1.x Files: GHOST in the Stack Machine](https://blog.ethereum.org/2020/07/28/the-1x-files-ghost-in-the-stack-machine/)

Type: Blog

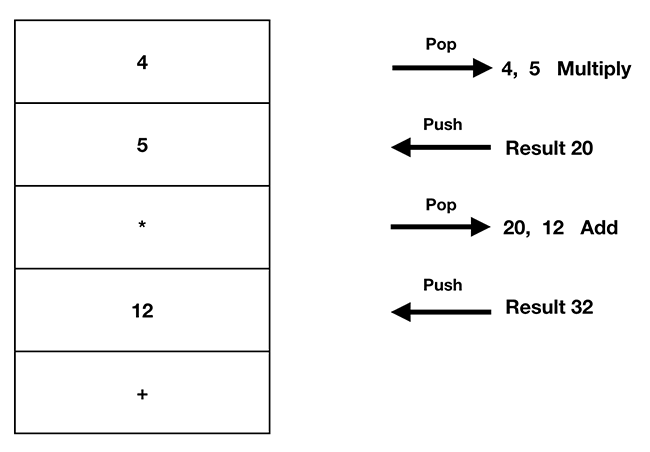
The EVM, which in a way actually is a literal machine chugging along, can be thought about as a function which accepts as inputs some state and outputs a new one based on some arbitrary set of rules. The only valid state transitions are the ones that come from valid transactions (that follow the rules). The abstract machine that will determine a new state (S') given an old valid state (S) and a new set of valid transactions (T) is the Ethereum state transition function: Y(S, T)= S'. EVM runs on thousands of computers connected to Network and running Ethereum Clients. And at any given time, there is one and only one canonical Ethereum state.

Deciding which states are canonical and which states are not is the sole responsibility of miners doing proof-of-work on the chain. Anyone using Ethereum mainnet has, either literally or just figuratively, “bought in” to one particular state history, namely the one with the most computational work put behind it, as determined by Ethereum’s Greedy Heaviest Observed Subtree (GHOST) protocol.

EVM can be compared to Babbage’s Analytical Engine. The steampunk EVM would be a mechanical computer that functions by manipulating physical punch cards. Each card would have 256 places for hole punches, and therefore each card could represent any number between 0 and 2^256. To perform a calculation, one could imagine this computer, through some fancy system of compressed air, putting the cards representing numbers and operations into a stack, and following a simple principle of “first in, last out”, one-by-one it would PUSH new cards to the top of the stack, or POP cards from the top of the stack to read them for next steps. These might be new numbers to calculate with, or arithmetic operations like ADD or MULTIPLY, but they could also be special instructions such as to STORE a card or set of cards for later. Because the cards are simple binary, the operations also have to be ‘encoded’ into a binary number; so we call them operational codes, or just opcodes for short.

If the stack machine were calculating 4 \* 5 + 12, it would go about it like so:

Postorder :- 4\*5+12 => 45\*12+



POP value 4 from the stack, keep it in memory. POP the value 5 off the stack, keep it in memory. POP the value \* from the stack; send everything in memory to the multiplication module; PUSH the returned result (20) the stack. POP the value 20 from the stack; keep it in memory. POP the value 12 from the stack; keep it in memory. POP the value + from the stack; send everything in memory to the addition module; PUSH the returned result (32) the stack. (Source: The EVM Runtime Environment).

The “real” EVM has many different opcodes for doing various things. A certain minimum-viable set of these opcodes are needed to do generalized computation, and the EVM has all of them (along with some special ones for crypto, e.g. the SHA-3 hash function). For better or worse, the idea that the EVM is (or is not) Turing-complete has long been under discussion— it’s this stack-based architecture which has the property of Turing-completeness: The EVM’s rules of execution can in principle, given a long enough time and big enough memory, run any conceivable computer program so long as it’s compiled down to the correct 256-bit words and executed in the stack.

Now we can summarize it a bit more clearly: The EVM is the physical instantiation (read: instance) of the state transition function. A valid state in Ethereum is one that was calculated by the EVM, and the canonical state is the valid state with the most computational work done on it (as determined by the GHOST protocol).