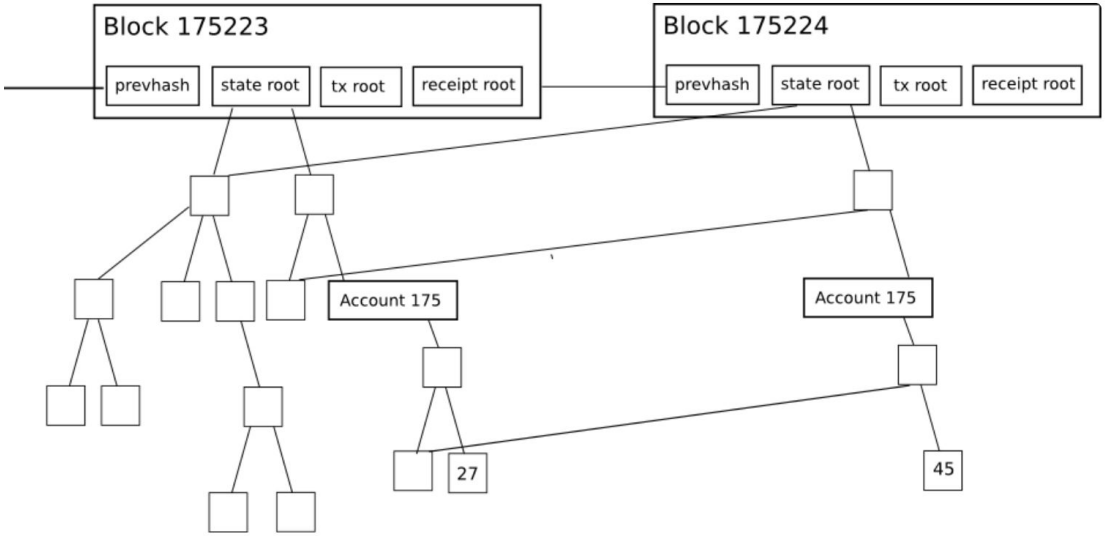
Stateless client witnesses

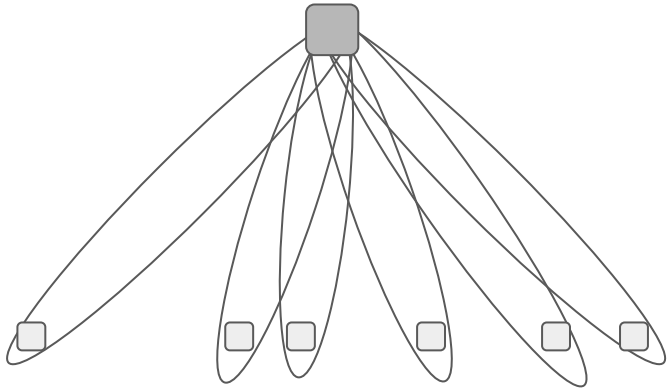
Reminders

- We can model block processing as a **state transition function**:
  `stf(state, block) = state’`
- State is stored in a Merkle tree
- Only a small portion of the state is read/written each block



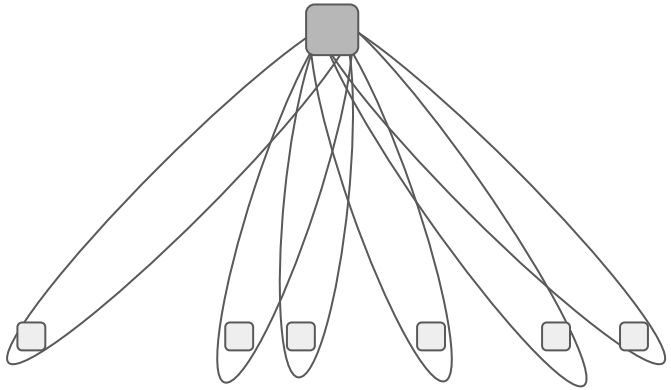
Stateless client basics

- Instead of the client holding the state, the client holds the **state root**
- Original state transition function:
  `stf(state, block) = state’`
- New state transition function:
  `stf’(state_root, block, witness) = state_root’`
- **Witness** = the portions of the state that are read and modified, along with proofs
  - Think Merkle proofs, but we can do other things too



Stateless client benefits

- Near-zero initial sync time
- Clients can validate blocks out-of-order (start with latest then move backwards, or only validate in response to fraud proofs)
- No disk access, so HDD-friendly full nodes
- Fraud proof friendliness
- Mandatory for rapid reshuffling in sharding



Stateless client stats

- Assume total state size N, k objects accessed
    - Typical: $N \sim= 2^{30}$, $k \sim= 2^{10}$
- Witness size $\sim= k * \log(N/k)$ chunks (plus the k objects themselves)
- ~600 bytes per access in practice (so ~600 kB block witness)
- Grinding attacks on the Merkle tree may expand this by ~3x in the worst case

Eth1.0 has mispriced gas schedules for this!

- Theoretical max: CALL into many accounts
- 12m gas / 800 gas per call (700 gas cost + overhead) = 15000 calls
- Each call: 3000 byte witness + 24000 byte code
- Total: 15000 * 27000 = 405 MB witness (!!)
- Fixes
  - Switch from hexary to binary tree (witness cost drops from $\sim 512 * \log_{16}(n)$ to $\sim 32 * \log_{2}(n)$, a 4x drop)
  - Gas cost changes
  - Code merklization

Fixes in the pipeline...

Summary of gas cost change recommendations

- Charge 3 gas per byte for accessing contract code. This could be either charging for the full contract, or we could implement a scheme where you are only charged for the chunks you access plus Merkle proofs for those chunks.
- Increase gas cost of EXTCODESIZE, BALANCE, *CALL to 2400
- Increase gas cost of SLOAD to 2000
- In all of the above cases, charge only for first-time accesses in a block.

Summary of effects of gas cost changes plus tree binarification

- ETH transfer block witness size drops from 2.31 MB to 658 KB (~780 KB if we increase gas limit to 12M)
- ERC20 transfer block witness size drops from 1.57 MB to 445 KB (~512 KB if we increase gas limit to 12M)
- Worst-case block witness size drops from 324 MB to 3.3 MB (~4 MB if we increase gas limit to 12M)

Binary trie format

Eth1.x Research

gballet 3 7d

Thanks to @sinamahmoodi for his input, @poemm and @carver for pointing out typos.

A method for the conversion of hexary tries into binary tries has been proposed in [1]. This proposal focuses on how the trie is represented, independently of the conversion method.
It is made up of three parts: the first part is a simpler write up, the second part illustrates the structure with an example, and the last part presents the formalization of the structure, using the notation from the yellow paper[2].

Overview

The proposed structure has only one type of node, which merges the functionalities of branch, extension and leaf nodes.

Some quick numbers on code merkelization

Eth1.x Research ■ stateless

lithp Apr 10

Some key numbers in determining the viability of stateless ethereum are (1) how big the witnesses will be, and (2) what witness size the network can support. (1) must be less than (2). To that end there are a couple different proposals for making witnesses as small as possible, one of them is code merkelization. It would take a while to get perfect numbers on the impact of merkelization so I tried to come up with a quick estimation of the benefits.

I wrote some logic to record which bytes of contract bytecode are accessed during transaction execution, and then ran it over a sampling of recent blocks. I scanned every 50th block, starting with 9375962 and ending with 9775962 (a total of 8001 blocks).

This post is kind of a grab-bag, I don't have any conclusions but wanted to put this data somewhere public so we can refer to it later. Here are some results which seemed useful:

Challenge

- Can we do better than 600 bytes average case, ~2 kB worst case per chunk?
- Two paths
  - SNARK Merkle trees
  - Vector/polynomial commitments

Polynomial commitments

- “Hash” `com(P)` of some polynomial `P(x)` that has extra properties
- Given `P` and `z`, Prover can provide “opening” `Q` that shows `P(z) = a`
- Verifier can take `com(P)`, `z`, `a`, `Q` and verify that `P(z) = a`
- Can be used as alternative to Merkle tree: given data `D[0] ... D[n-1]`, you can create a `deg < n` polynomial `P` where `D[i] = P(i)`. You can use openings instead of Merkle branches
- But why? We’ll see….

Polynomial commitments

- Kate
  - Trusted setup: secret `s`, provide `G`, `G * s`, `G * s^2`, `G * s^3` … `G * s^d`, `G2 * s`
  - Encode `P(x) = c_0 + c_1x + c_2x^2 + …` with `G * c_0 + (G*s) * c_1 + (G*s^2) * c_2 + …`
  - Prove `P(z) = a` by proving `P(x) - a` has a zero at `z`, by providing a commitment to `Q(x) = (P(x) - a) / (X - z)`
  - Verify with pairings: `e(com(Q), (G2*s) - G2 * z) = e(com(P) - G * a, G2)`
- FRI
  - Given `P(x)`, define
    - `EVENS_p(x) = (P(x) + P(-x)) / 2` (only even-degree coefficients)
    - `ODDS_p(x) = (P(x) - P(-x)) / 2` (only odd-degree coefficients)
  - Notice that `EVENS_p(x)` and `ODDS_p(x)/X` are both degree `N/2` polynomials in `x^2`
  - Let `P'(x) = EVENS_p(x) + r * ODDS_p(x)/X` for random `r`.
  - Prover provides Merkle root of many (eg. 8N) evaluations of `P(x)`.
  - Prover provides Merkle root of many evaluations of `P'(x)`
  - Make many random Merkle branch checks (use Merkle root of `P(x)` as seed) to prove `P'` and `P` satisfy the above relation at most points
  - Repeat until you get to degree 1 polynomial (or low enough to check directly)

Multi-openings

- Suppose given $P(x)$, you want to prove $P(z_i) = a_i$ for many z_i: $z_1 \dots z_k$
- You can do this with one polynomial commitment!
- Let $I(x)$ be the $\deg < k$ polynomial where $I(z_i) = a_i$
- Let $Z(x) = (X - z_1) * \dots * (X - z_k)$
- Strategy: prove $P(z_i) = a_i$ for all i by proving $(P - I)(z_i) = 0$, which we do by proving $P - I$ is a multiple of Z
- We do that by providing $Q(x) = (P(x) - I(x)) / Z(x)$
- Verifier can compute I and Z and check the equation directly (note that in Kate this requires a few extra G2 points in the trusted setup, though there are ways to get around this if needed)
- Result: prove k values with fixed-size witness!

Unsolved problem….

Open problem: ideal vector commitment

Cryptography ■ vector-commitment ■ polynomial-commitment

vbuterin 3 May 14

We're looking for a commitment scheme to commit to a list of N values (think `N ~= 2**28`) which has the following properties:

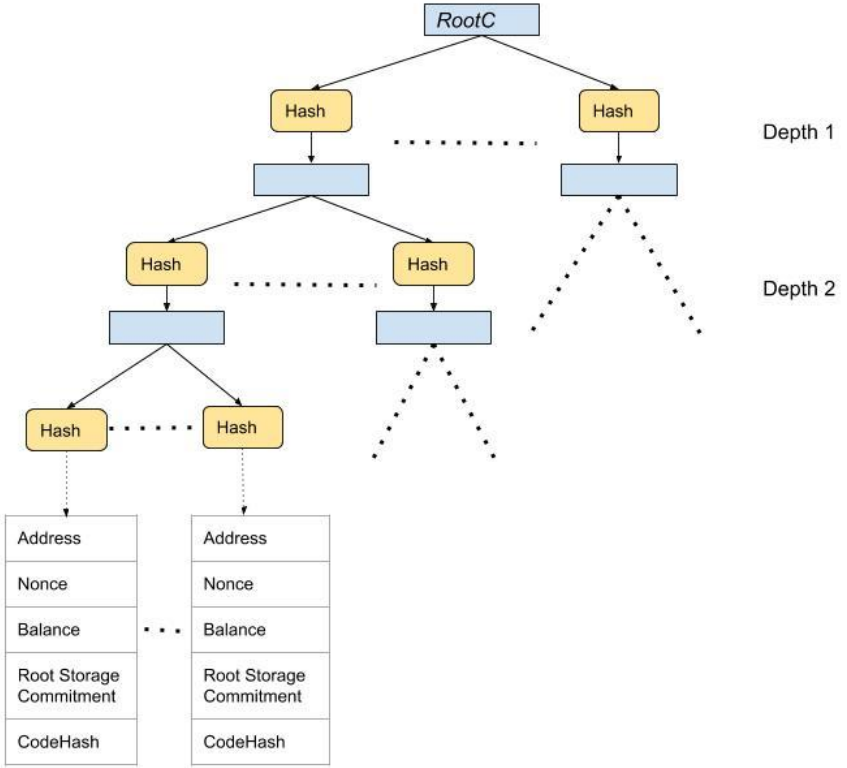
1. The commitment should be small (fixed size or polylog)
2. The commitment should be (computationally) binding, ie. a commitment `c` constructed from one vector `V = [v1 ... vN]` should not match against any other feasibly-discoverable vectors. (We don't care about hiding properties)
3. For any given set of positions `x1 ... xk` where `1 <= xi <= N`, there should be an efficient (ie. quasilinear time to calculate, *sublinear proof size*) way to prove that the values `V[x1]`, `V[x2]` ... `V[xn]` are part of the vector committed to by `c`
4. There should be an efficient (ie. ideally `O(k)` but `O(k * log^c(n)` is okay too) way to compute such a proof for any `x1 ... xk`. Requiring `O(n)` or even slightly larger precomputation before you receive the coordinates is okay.
5. Given a set of updates `(x1, y1) ... (xk, yk)` to a vector there should be:
    - (i) an efficient (ie. ideally `O(k)`, but `O(k * log^c(n))` is okay too) way to update `c`
    - (ii) an efficient (ie. ideally `O(k)`, but `O(k * log^c(n))` is okay too) way to update any precomputed tables required to generate proofs (that's *updating the entire precomputed table needed to generate all witnesses*, not updating a single witness)

Challenge

- When the state gets updated, need to update all the witnesses
- Merkle trees do this easily: updating one value only updates `log(n)` hashes in the hash tree
- Kate commitments are not good at this: updating one value in `P` updates every witness
- Intuition: witness updating requires witnesses to be computable from data in a “tree-like structure”
- Stated formally: create a structure which stores auxiliary data `D`, such that changing `P(x)` in one position (think evaluation point on a cyclic multiplicative group $\{1, \omega, \omega^2 \ldots \omega^{n-1}\}$) makes `polylog(n)` changes to `D`, and such that $P // (X - \omega^k)$ for any `k` requires reading `D` in `polylog(n)` positions
- Challenge 2: state size > trusted setup size

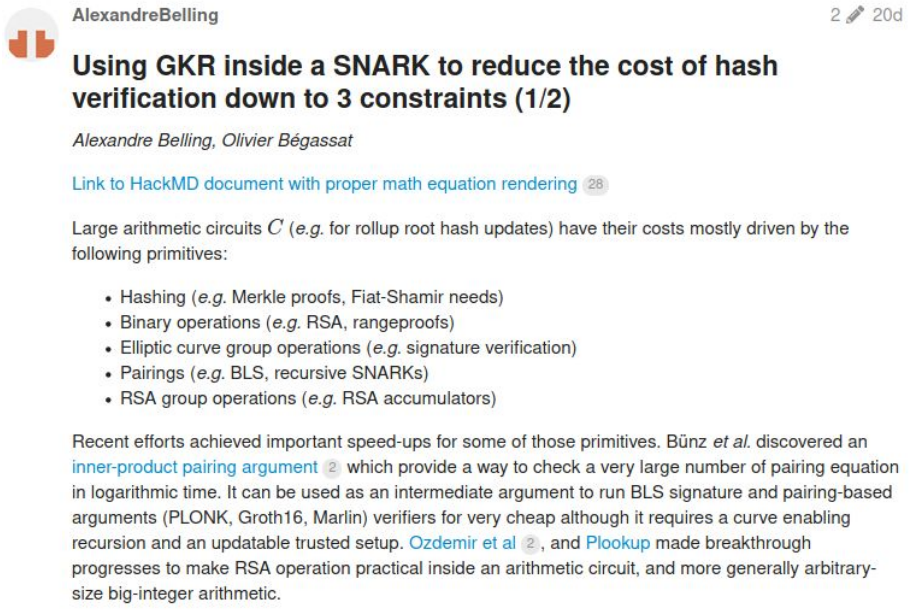
Technique 2: Verkle trees

- Tree of polynomial commitments, eg. each of depth 1024
- Witness: 2 points (96 bytes) per element
- Update cost: minimal
- Proof generation cost: 1024*k field operations + 1024 curve operations
- Verification cost: k pairings (naively), 3 pairings (with some extra work)



Technique 3: SNARKing Merkle trees

- Benefit 1: no sacrifices in witness updating cost
- Benefit 2: post-quantum safe
- Problem: how many hashes/sec can you prove?
- Recent number from Starkware: 10000 hashes/sec
  - ~2s to compress an ethereum block witness!
- Other recent idea: GKR



AlexandreBelling 2 20d

Using GKR inside a SNARK to reduce the cost of hash verification down to 3 constraints (1/2)

Alexandre Belling, Olivier Bégassat

Link to HackMD document with proper math equation rendering 28

Large arithmetic circuits C (*e.g.* for rollup root hash updates) have their costs mostly driven by the following primitives:

- Hashing (*e.g.* Merkle proofs, Fiat-Shamir needs)
- Binary operations (*e.g.* RSA, rangeproofs)
- Elliptic curve group operations (*e.g.* signature verification)
- Pairings (*e.g.* BLS, recursive SNARKs)
- RSA group operations (*e.g.* RSA accumulators)

Recent efforts achieved important speed-ups for some of those primitives. Bünz *et al.* discovered an inner-product pairing argument 2 which provide a way to check a very large number of pairing equation in logarithmic time. It can be used as an intermediate argument to run BLS signature and pairing-based arguments (PLONK, Groth16, Marlin) verifiers for very cheap although it requires a curve enabling recursion and an updatable trusted setup. Ozdemir et al 2, and Plookup made breakthrough progresses to make RSA operation practical inside an arithmetic circuit, and more generally arbitrary-size big-integer arithmetic.

Summary

- Stateless clients are great and can solve lots of problems
- Need to cut witness sizes down
- A bunch of fancy arithmetic techniques can help us do this!
- Still need more research though