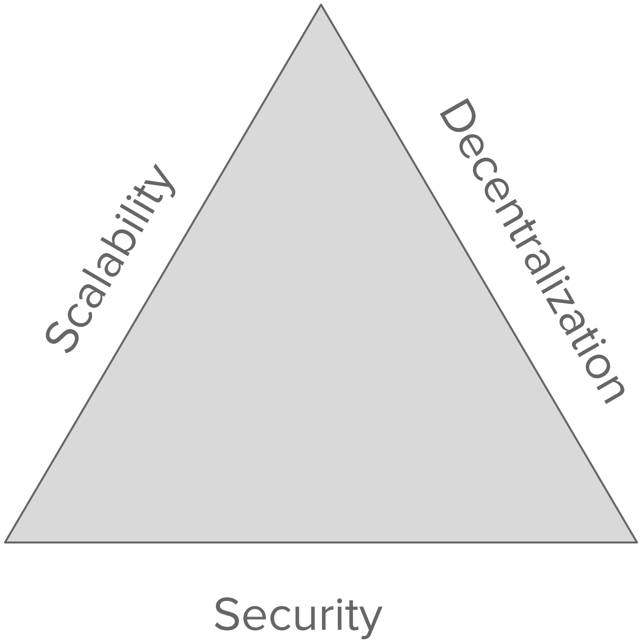
Sharding

Making blockchains scalable, decentralized and secure.

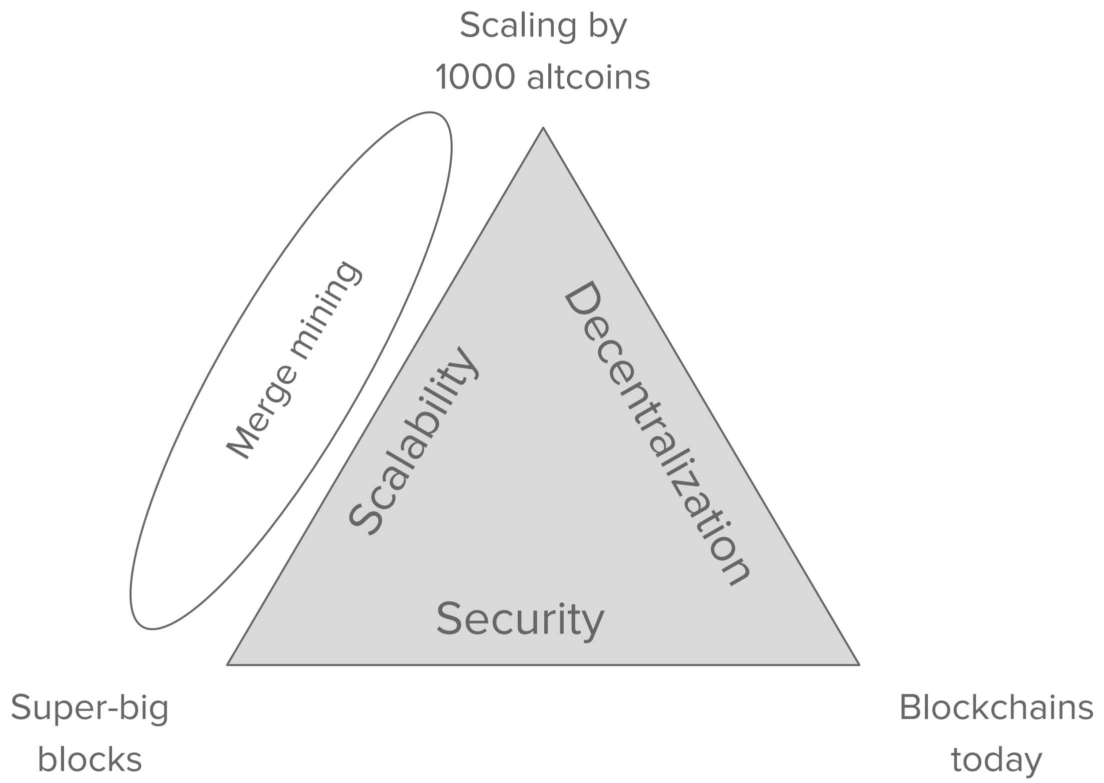
The Scalability Triangle



Semi-formally defining these properties

- Assume the total computational/bandwidth capacity of a regular computer is **O(c)**, and the total load of a blockchain is **O(n)**
- Decentralization: the system can run in an environment where all nodes have **O(c)** resources
    - Possible weakening: can have supernodes, but require only 1 of N supernodes to be honest
- Security: the system can survive attacks up to some specific percentage of all miners/validators (eg. 33%)
- Scalability: the system can handle a load of **O(n)** > **O(c)**
    - Computation
    - State storage
    - Bandwidth

The Scalability Triangle



1% Attacks

- Suppose N = 100 * C. Then, each node can only verify 1% of all data. Therefore, any given piece of data is being verified by 1% of the nodes. What if the attacker corrupts that specific 1%?

Claim: we can reach the middle of the triangle, though we do need to use some more complex tools to get there.

The philosophy: proxy validation

- Because **O(n)** > **O(c)**, a node cannot verify the entire blockchain directly. But we can try to verify blocks *indirectly*.
- Ways to verify indirectly:
    - **Committee voting**
    - **Cryptoeconomic**
    - **Cryptographic**
    - **Probabilistic**
- Goal: black-box indirect validation, make analysis of the blockchain maximally the same as the non-scalable case.
- Method: organize data into **O(c)**-sized “shard blocks”. From the PoV of the main chain, the headers of shard blocks are (sort of) like **O(1)**-sized transactions.

Semi-formal security model

- Think of full validation of all blocks as an "ideal" procedure that nodes could run
- Prove that the protocol would work if full validation were used (using same techniques as for non-scalable chains)
- Use indirect validation as a substitute for full validation; prove that the two are equivalent given some security assumptions
  - Honest majority
  - Honest minority (1 of N, >=k of N; of validators or of users)
  - Network latency
  - Cryptographic

Committee voting

- Idea: randomly select 100-1000 validators from a large pool
- Some percentage (eg. ⅔) of them need to vote approving a given shard block for that shard block to be eligible for main chain inclusion
    - These sigs can be aggregated via BLS aggregation, STARK aggregation, etc etc
- Security model: **honest majority**
    - Can be secure against a semi-adaptive or adaptive adversary

Minimum safe committee size

- Possible goal: 2^{-40} chance of safety failure (a ⅓ attacker getting a ⅔ committee)
- Then, minimum committee size is 111 nodes
- If an attacker can **manipulate RNG**, they can give themselves many chances
- Eg. if the attacker has 40 bits of manipulation, need to target 2^{-80} chance of safety failure. Minimum committee size increases to 231 nodes
- If we want a more stringent goal (eg. 2^{-40} chance of ⅖ simulating ⅗), then minimum committee size also increases (to 315 nodes)
- **Sidenote**: private committee selection roughly doubles required safe committee size

Screenshot this to play with binomial distributions in 4 lines of code!

```
def fac(n):
    return n * fac(n-1) if n else 1
def choose(n, k):
    return fac(n) / fac(k) / fac(n-k)
def prob(n, k, p):
    return p**k * (1-p)**(n-k) * choose(n,k)
def probge(n, k, p):
    return sum([prob(n,i,p) for i in range(k,n+1)])
```

Fault proofs: outsourced computation protocol

- Suppose we can represent a computation $y = f(x)$ as $y=f_n(f_{n-1}(\ldots(f_1(x))\ldots))$
- Submitter sends intermediate states of computation:
  - $S_1 = f_1(x)$
  - $S_2 = f_2(S_1)$
  - …
- Each f_i can be computed within a transaction
- Submitter also submits a deposit

Fault proofs: outsourced computation protocol

- Within some challenge period, anyone can submit a “challenge index” i
- If S_{i+1} != $f_{i+1}(S_i)$, then the challenger gets the submitter’s deposit
- If no challenges are made within the challenge period, submitter gets their deposit back plus a reward

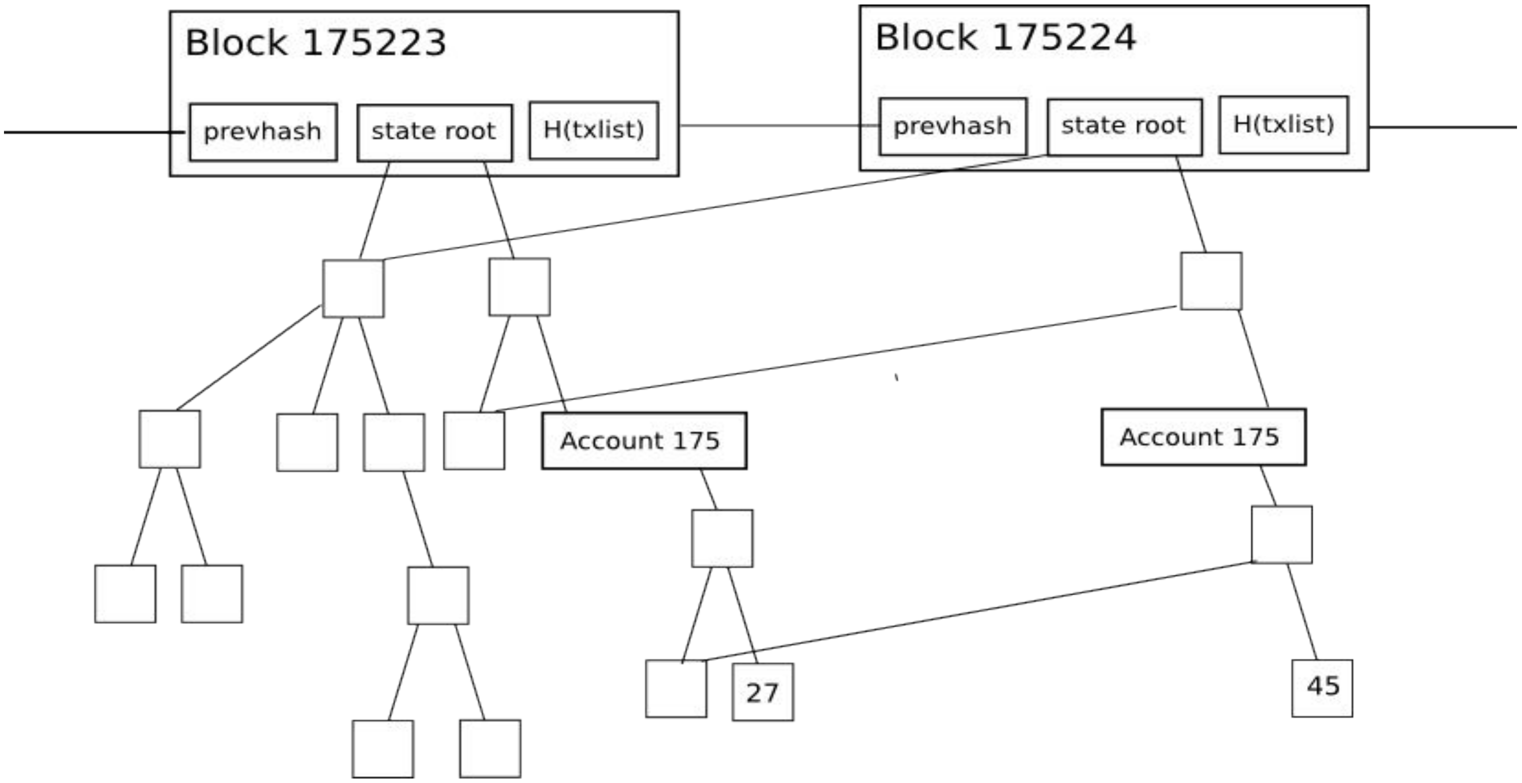
Fault proofs in block validation

- No need for specified “challenge period”, clients can execute this protocol subjectively
- A client can accept a computation after it has (i) seen and rebroadcasted this computation, and (ii) it has not seen a valid challenge for a privately chosen period δ after that

Fault proofs in block validation

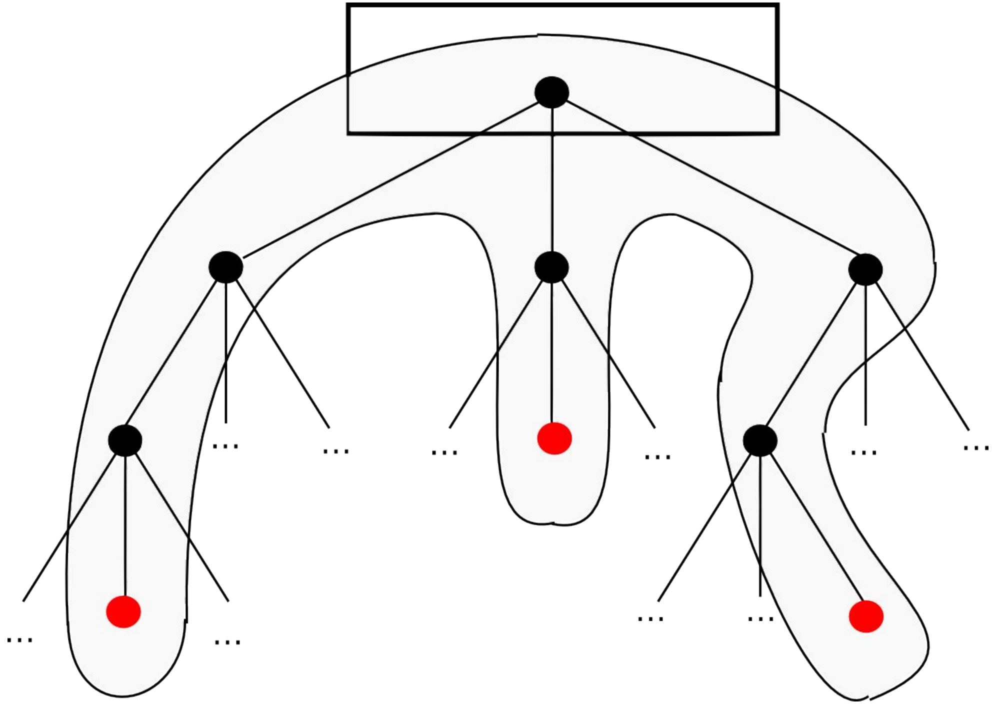
- Problem: in practice, if blockchain load is > O(c) sized, the state is also > O(c) sized. How to compute in f_i isolation?
- Solution: Merkle state trees + witnesses

State trees

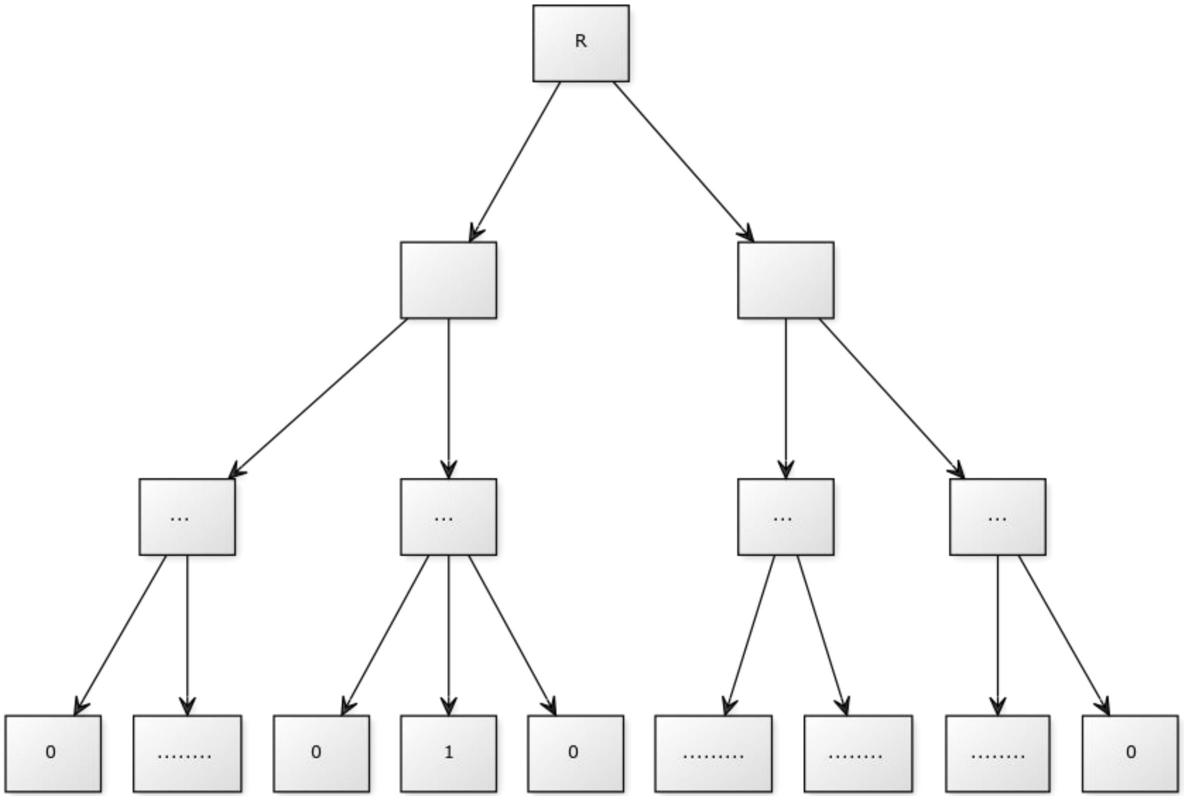


Block 175223
prevhash
state root
H(txlist)
Block 175224
prevhash
state root
H(txlist)
Account 175
Account 175
27
45

Stateless validation



Stateless validation



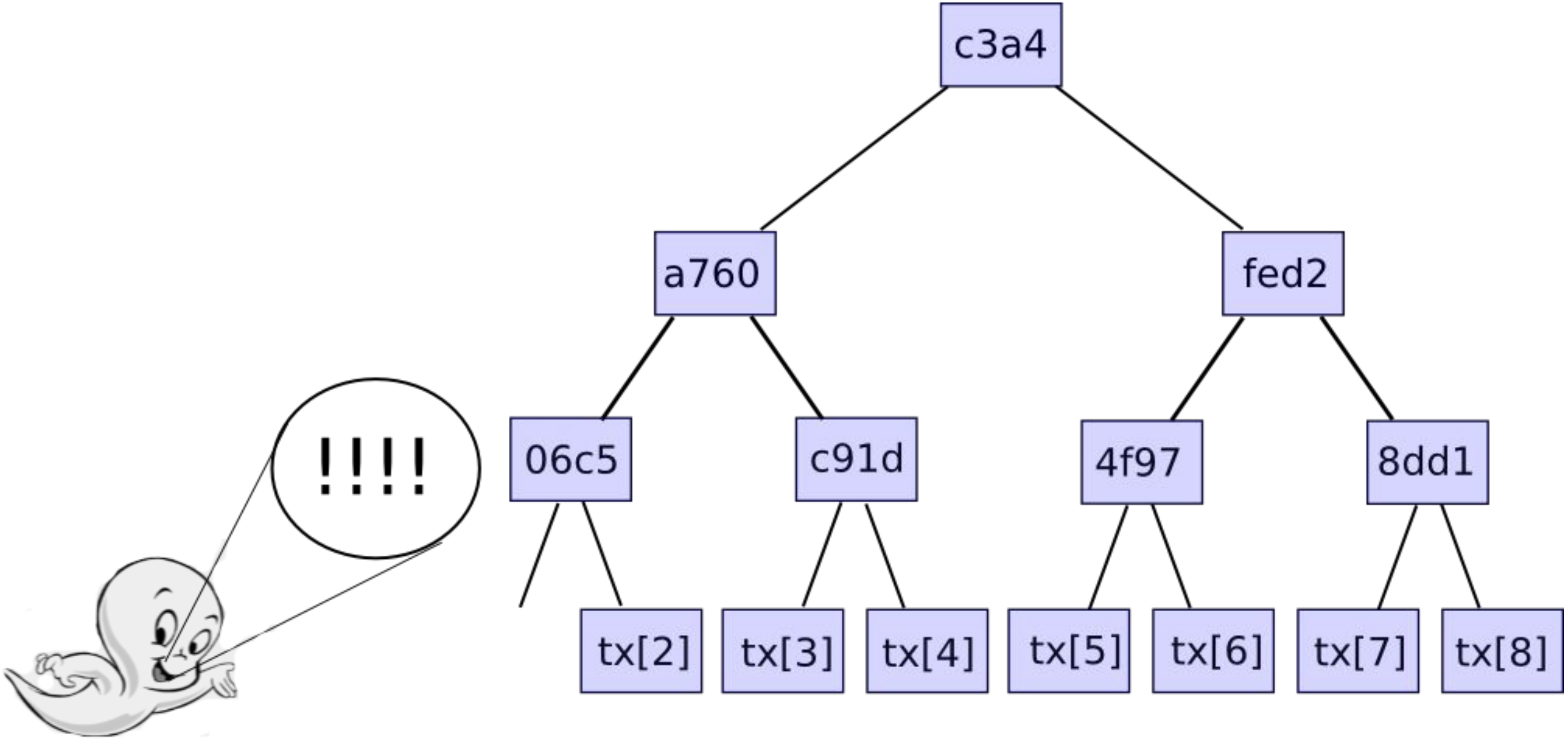
Stateless validation stats

- Optimal tree structure: likely **sparse Merkle tree**
- Ethereum today: ~2^{25} accounts
- Branch length: 32 * 25 bytes per account accessed
- N branches: 32 * (25 - log(N)) with batching
- Example: Ethereum block full of simple transactions
  - 380 txs
  - 2 accounts accessed per tx
  - 760 * 32 * (25 - log(760)) = 24320 * (25 - 9.57) ~= 375kb + some overhead for account state
  - Raw size: ~= 38 kb

Succinct proofs (SNARKs and STARKs)

- Make a proof that **f(x, w) = y** (where w can be large and not published), which anyone can verify much more quickly than computing **f**
- If you don't know how these work, try:
  - https://medium.com/@VitalikButerin/zk-snarks-under-the-hood-b33151a013f6 and dependencies
  - https://vitalik.ca/general/2017/11/09/starks_part_1.html and https://vitalik.ca/general/2017/11/22/starks_part_2.html
- Can replace fault proofs
- Problem: high overhead (~500-50000x)

Data availability problem



What can data unavailability attacks do?

- Prevent fault proofs from working
- Prevent nodes from learning the state
- Prevent nodes from being able to create blocks or transactions because they lack witness data

Theorem (Philippe Camacho, 2009)

- CANNOT make **O(n)**-sized updates to an “accumulator” without broadcasting **O(n)** data
- Information theoretic argument:
    - Miner creates block that sends 1 ETH to k<n of n accounts
    - The network needs to know which accounts have money, so that transactions from the accounts that do have 1 ETH can succeed
    - Hence, n bits of information must have been transmitted

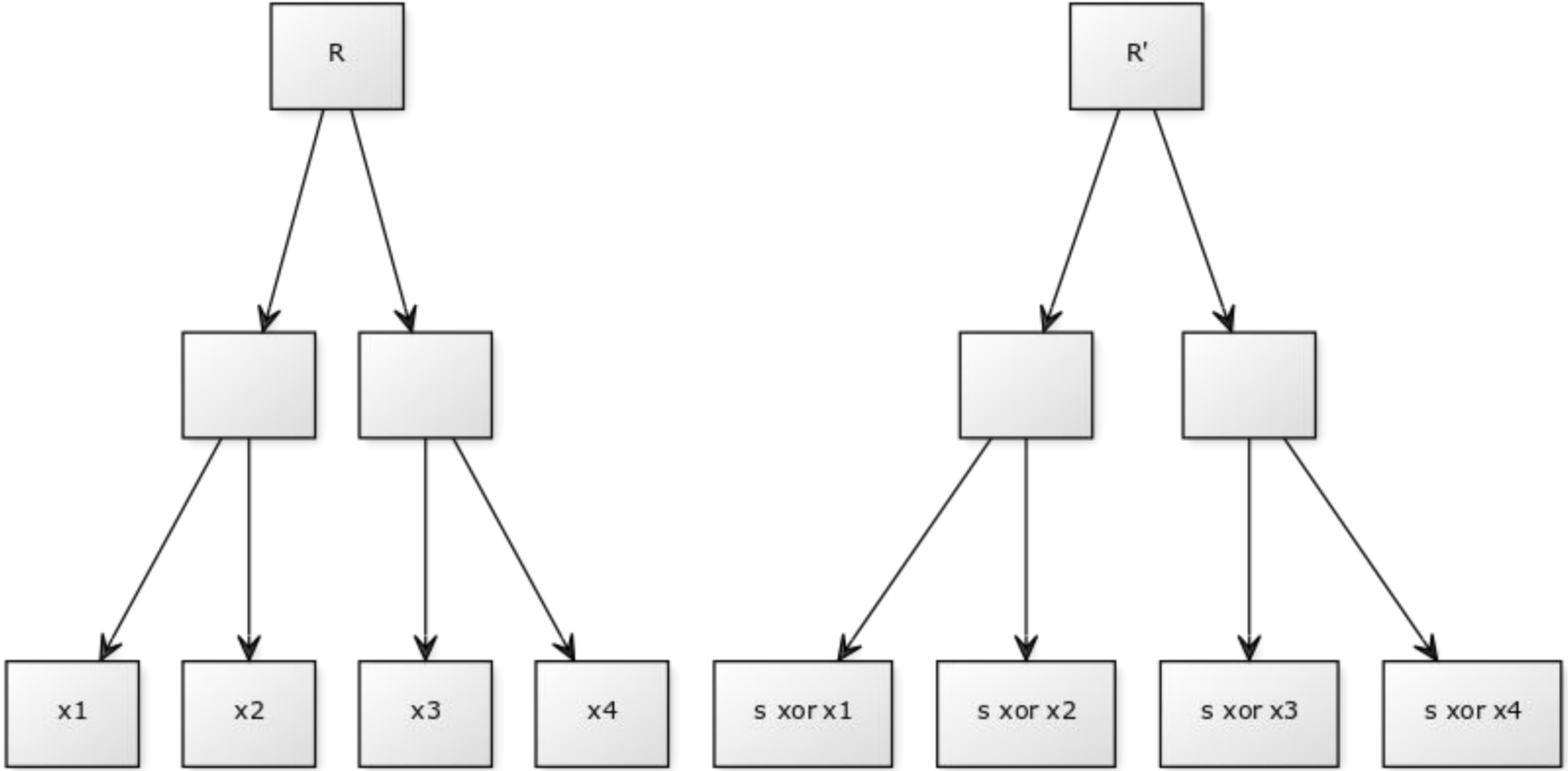
Custody bond

- Each member of a committee puts down a deposit
- They can be challenged with an index within 30 days. They must reply to each challenge with the corresponding Merkle branch of the data

Problems:

- No proof of **independent** storage
- Incentive to not check if everyone else is (“Verifier’s dilemma”)

Proof of custody



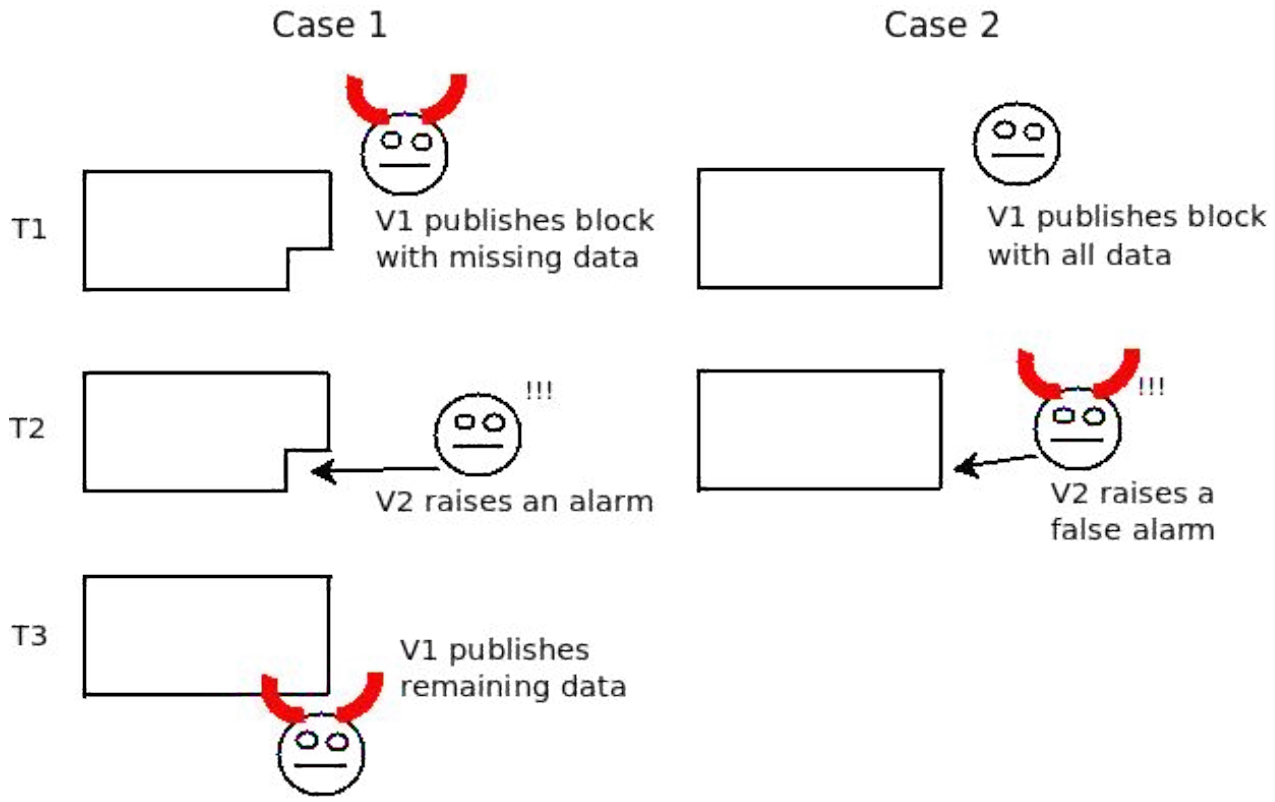
Custody bond

- Each member of a committee puts down a deposit, and precommits to **H(s)**. Some time after publishing blocks, every node must reveal **s** and precommit to a new **H(s')**
- They can be challenged with an index within 30 days. They must reply to each challenge with **s** and the Merkle branch of the proof of custody tree

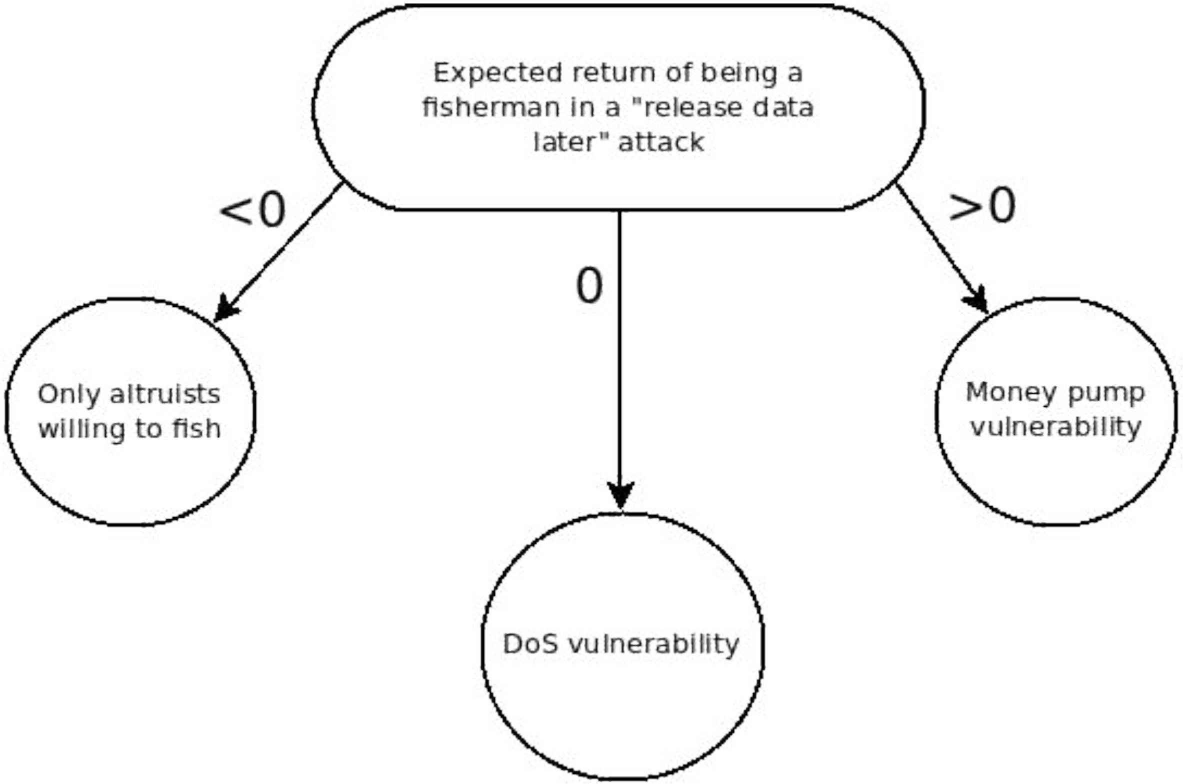
Fisherman's dilemma

- Q: Can you make a challenge-based scheme for data availability as robust as those for fault tolerance?
- A: definitive **NO**.

Fisherman’s dilemma

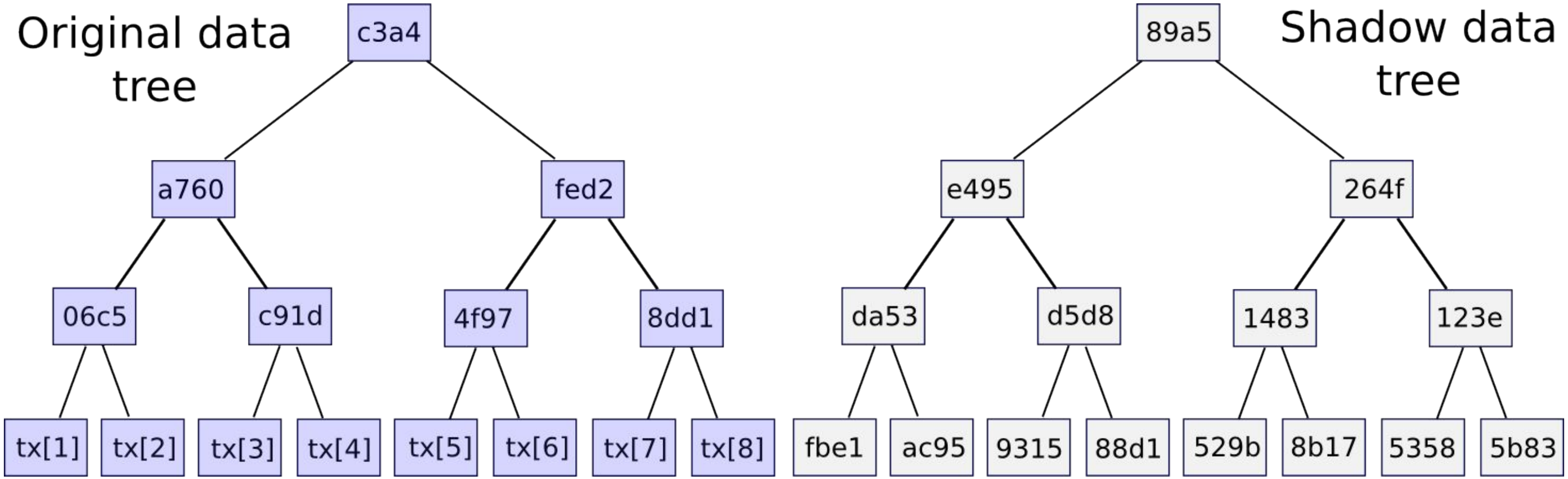


Fisherman's dilemma



Erasure coding as a solution

- Use erasure codes to “extend” length-N data into length-2N data, where any 50% of the extended data can recover the original data
- A client can randomly sample to check the availability of the extended data



Erasure coding as a solution

- Challenge: how to prove data is encoded correctly?
- Response 1: fault proofs
- Response 2: low-degree proofs of proximity (see https://vitalik.ca/general/2017/11/22/starks_part_2.html)
- Response 3: STARKs

- Challenge: targeted responses to fool specific clients
- Response 1: **honest client minority** assumption of ~N/size(proof) nodes
- Response 2: request through onion routing

Minimal sharded protocol

- Suppose there are N validators
- Split up state into N partitions ("shards"); transactions specify which shard they are for, and blocks in a shard aggregate transactions for that shard
- Define function `CHOOSE(height, shard_id) -> (proposer, committee)`
- The chain accepts a block at that shard if signed by the proposer and >⅔ of the committee
- Use any consensus algorithm (ideally, dual-use committee signatures as PoS signatures)
- Possible extension: define `CHOOSE` so that shard => proposer mapping is stable, allowing proposers to download entire shard state
    - Committee should be free-floating to maximize defense against adaptive adversaries

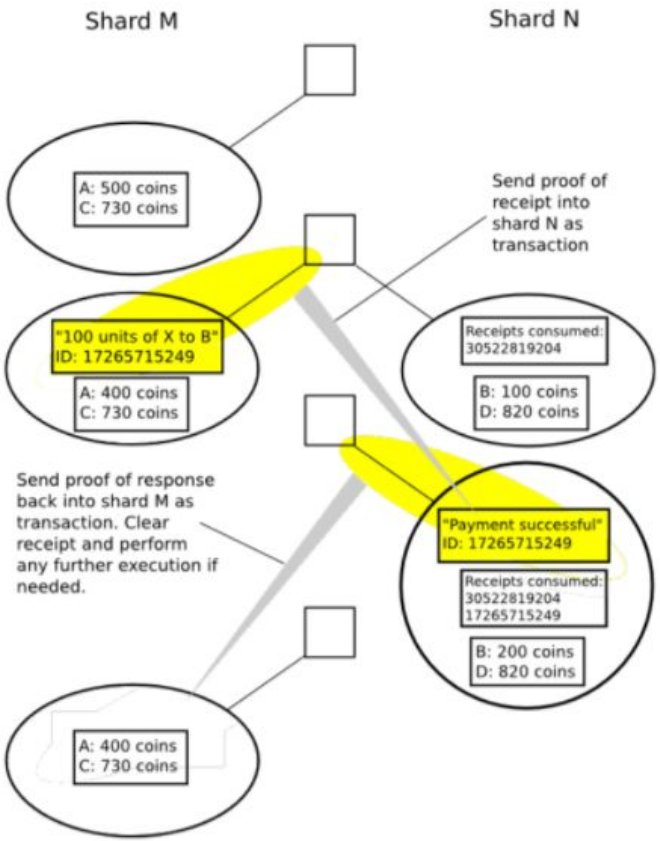
Extended sharded protocol

- The shards have their own blockchains
- An on-main-chain committee is only required to vote on “crosslinks”, that show the main chain the sequence of new block headers agreed on since the previous crosslink for that shard
- Alternative formulation: main chain protocol is the same as it was before, but a shard chain exists as a “coordination gadget” to allow multiple proposers to work together on building the proposal
- Goal: **reduce shard block time without increasing main chain overhead** (at the cost of keeping cross-shard communication overhead high)

Cross-shard communication

- Transactions can only access the state of their own shard
- How to process cross-shard operations (eg. moving ETH from one shard to another)?

Async cross-shard protocol



- Assumption: shards have a (delayed) view of each other's state roots
- log(n) overhead for Merkle branch (but note: intra-shard txs also require Merkle branch overhead for committee members)

Train and hotel problem

- Suppose you want to book a train ticket and a hotel room, but the transaction is worth it only if you book both
- Want to try to book both, but book neither if booking either one fails
- Suppose train and hotel smart contracts live on different shards

Credit to Andrew Miller for original formulation.

Yanking

- A generalization of “locking” schemes
- Contracts can allow themselves to be “yanked” into another shard with an async transaction

- Suppose the train contract is written in such a way that a bookable seat can be extracted and represented as a separate contract
- Step 1: extract bookable seat into separate contract
- Step 2: yank it into shard B
- Step 3: if the hotel is still available, atomically book both. Otherwise, give up
- Step 4: yank seat back into shard A, reinsert it into “main” train contract (if needed)

Synchronous cross-shard calls

- The consensus already gives us a total order on messages
- State execution is delayed until consensus on order settles
- Then, a separate process can compute state roots
- Problem: for a node with the state of only one shard, this should not require too many sequential rounds of network communication to fetch Merkle branches of “foreign” shards

Areas of further research

- Cross-shard calls and gas payment UX
- Synchronous communication schemes
- State calculation schemes
- Use of STARKs to replace Merkle witnesses
- Proofs of custody over state, and not just the most recent block
- Economics
- Faster cross-shard state root awareness