Problems in Cryptoeconomic Research

First, a few basic principles...

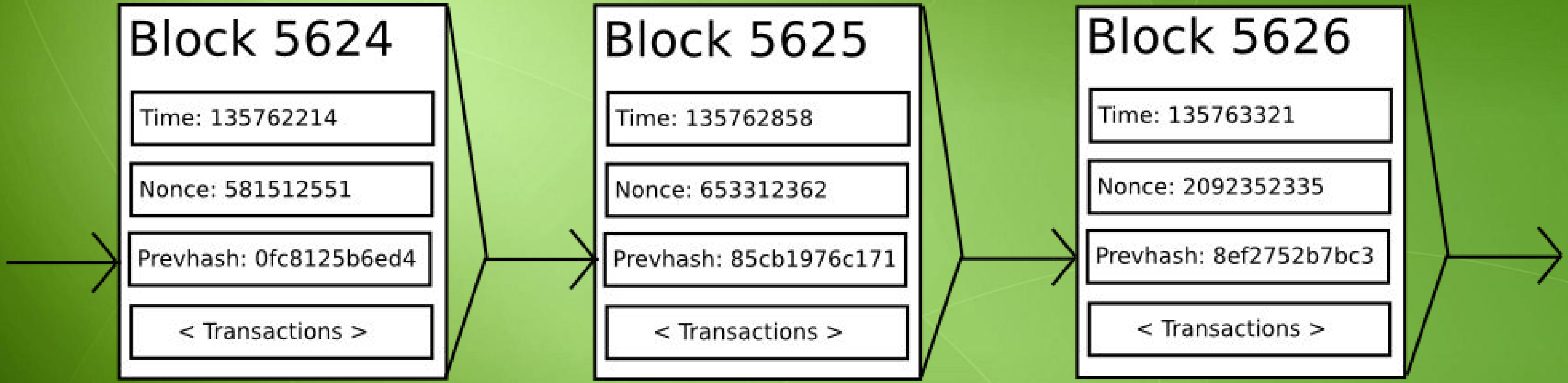
Cryptoeconomics is about...

- Using **cryptography** and **economic incentives** to achieve information security goals
  - Cryptography can prove properties about messages that happened *in the past*
  - Economic incentives defined inside a system can encourage desired properties to hold *into the future*

Simple blockchain: Desired properties

- Create a chain of blocks
- Include transactions in each block
- Maintain a "state"
    - Transactions affect state: `s' = STF(s, tx)`
- Maintain a clock

Simple blockchain: Desired properties



Simple blockchain: Desired properties

- **Convergence**: new blocks can be added to the chain but blocks cannot be replaced or removed
- **Validity**:
  - Only valid transactions should be included in a block
  - Clock should be roughly increasing
- **Data availability:** it should be possible to download full data associated with a block
- **Non-censorship**: transactions should be able to get quickly included if they pay a reasonably high fee

Tools in the toolbox

- **Cryptography**
  - Signatures
  - Hashes (incl. PoW)
  - Spooky advanced stuff (ZKPs, timelock, etc)
- **Economic incentives**
  - Rewards/penalties
  - Privileges

What can we use cryptoeconomics to build?

- Consensus algorithm
- Outsourced computation and storage
- Provably fair random number generation
- Providing true info about the real world (“oracles”)
- Governance (DAOs)
- Stable-value cryptocurrencies (“stablecoins”)
- Bounties for solutions to math or CS problems
- Telling the time

Security models

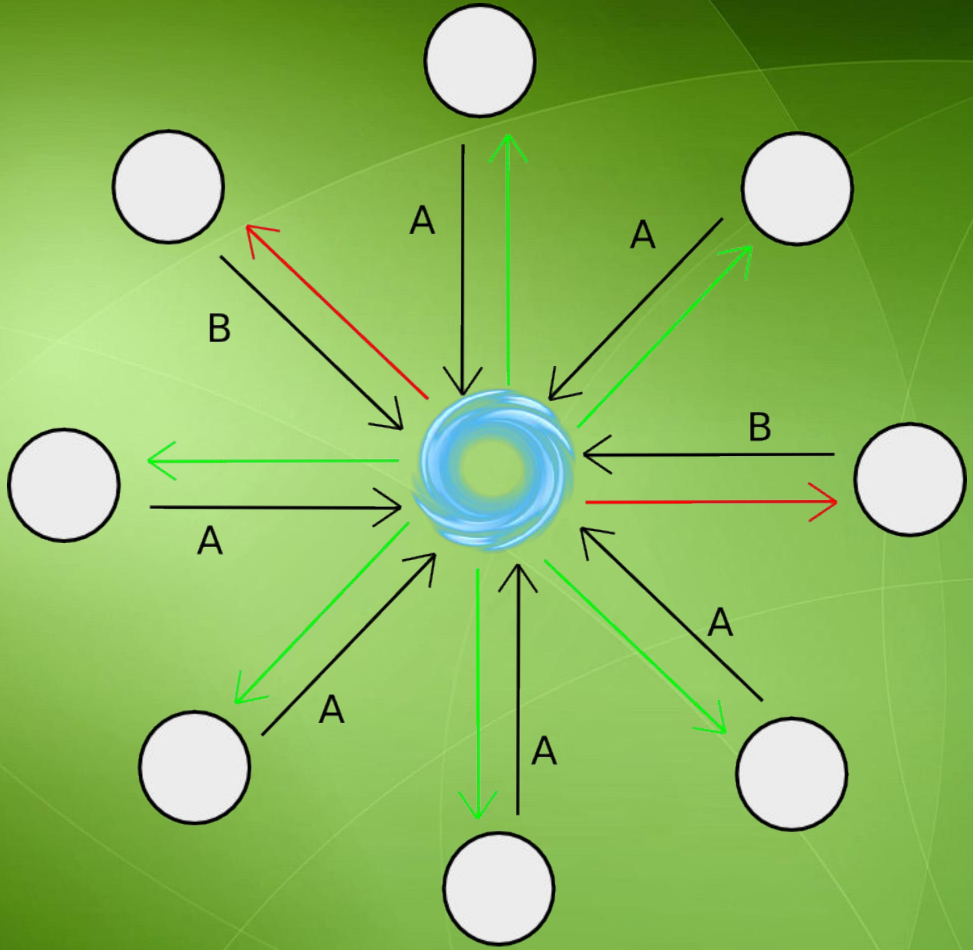
- In traditional fault-tolerance research, we make an **honest majority assumption**, and use this to prove claims about correctness of algorithms
- In cryptoeconomic research, we make assumptions about:
  - Level of **coordination** between participants
  - **Budget** of the attacker
  - **Cost** of the attacker

Fault tolerance of Bitcoin

Model	Fault tolerance / security margin
Honest majority[1]	~½ (as latency approaches zero)
Uncoordinated majority[2]	~0.2321
Coordinated majority	0
Bribing attacker	~13.2 * k **budget**, 0 **cost**

1. http://bravenewcoin.com/assets/Whitepapers/Anonymous-Byzantine-Consensus-from-Moderately-Hard-Puzzles-A-Model-for-Bitcoin.pdf
2. http://fc16.ifca.ai/preproceedings/30_Sapirshtein.pdf

Example: Schellingcoin


A
A
B
B
A
A
A
A
A

Example: Schellingcoin

- Uncoordinated choice: you have the incentive to vote the truth, because everyone else will vote the truth and you only get a reward of P if you agree with them
- Why will everyone else vote the truth? Because they are reasoning in the same way that you are!

P + epsilon attack

A bribing attacker can corrupt the Schellingcoin game with a **budget** of P + ε and zero **cost**!

Base game:

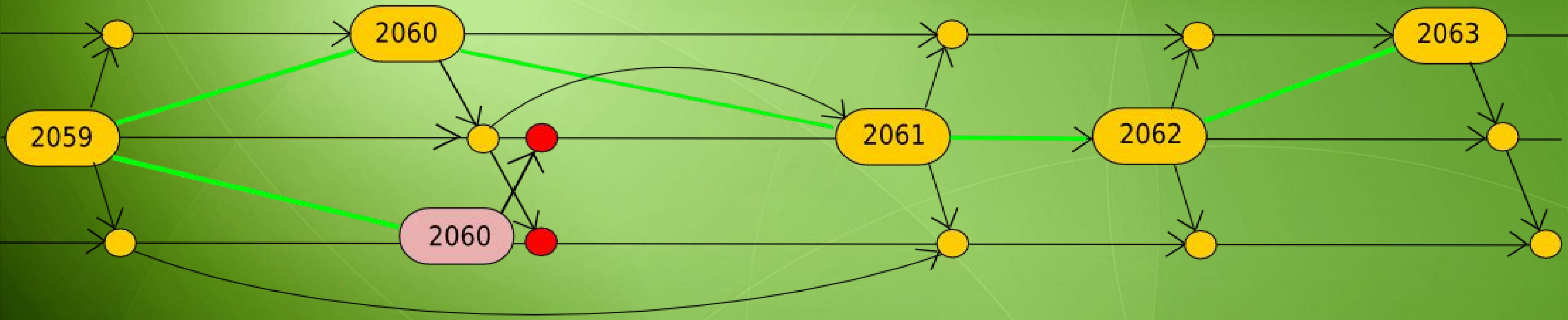
	You vote 0	You vote 1
Others vote 0	P	0
Others vote 1	0	P

With bribe:

	You vote 0	You vote 1
Others vote 0	P	P + ε
Others vote 1	0	P

Example: proof of work

- You get in the main chain, 12.5 BTC reward
- You do not get in the main chain, no reward.



- Same strategy as P+epsilon attack can be used in the bribing attacker model!

Are coordinated choice models realistic?

Yes.

Are bribing attacker models realistic?

- Subsidized mining pools (eg. to influence segwit vs BU voting)
- Subsidized stake pools in PoS

Casper

Casper overview

- **Deposit** + **penalty**-based proof of stake
- Anyone can join as a validator by submitting a deposit (in ETH), and they get rewarded for participating
- The protocol defines a set of **slashing conditions**, which roughly represent cases where a fault can be uniquely attributed to a given validator
- If you send messages that trigger a slashing condition, your deposit is destroyed

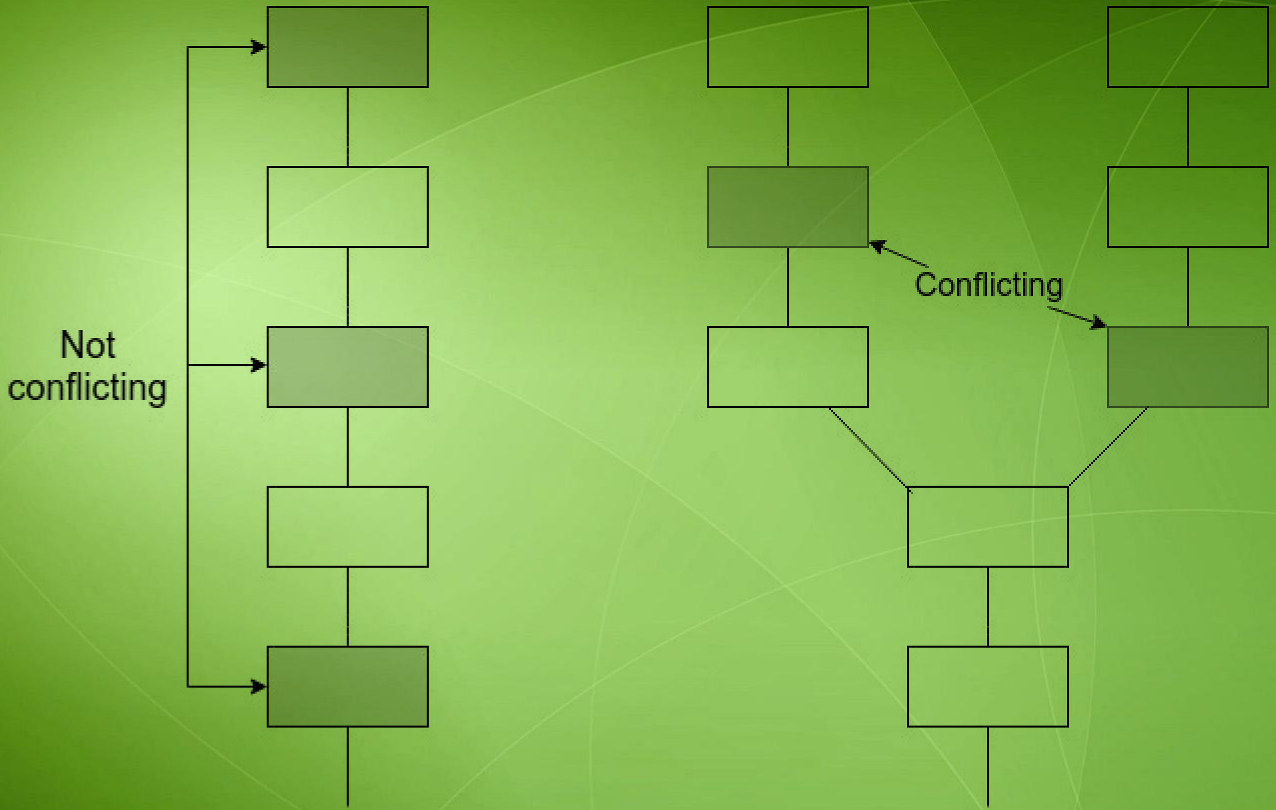
A slashing condition might look like this:

> *If a validator sends a signed message of the form*
>
> `["PREPARE", epoch, HASH1, epoch_source1]`
>
> *and a signed message of the form*
>
> `["PREPARE", epoch, HASH2, epoch_source2]`
>
> *where* `HASH1 != HASH2` *or* `epoch_source1 != epoch_source2`*, but the* `epoch` *value is the same in both messages, then that validator's deposit is* ***slashed*** *(ie. deleted)*

```
# Cannot make two prepares in the same epoch
def double_prepare_slash(validator_index: num, prepare1: bytes <= 1000, prepare2: bytes <= 1000):
    # Get hash for signature, and implicitly assert that it is an RLP list
    # consisting solely of RLP elements
    sighash1 = extract32(raw_call(self.sighasher, prepare1, gas=200000, outsize=32), 0)
    sighash2 = extract32(raw_call(self.sighasher, prepare2, gas=200000, outsize=32), 0)
    # Extract parameters
    values1 = RLPList(prepare1, [num, bytes32, bytes32, num, bytes32, bytes])
    values2 = RLPList(prepare2, [num, bytes32, bytes32, num, bytes32, bytes])
    epoch1 = values1[0]
    sig1 = values1[5]
    epoch2 = values2[0]
    sig2 = values2[5]
    # Check the signatures
    assert ecrecover(sighash1,
                     as_num256(extract32(sig1, 0)),
                     as_num256(extract32(sig1, 32)),
                     as_num256(extract32(sig1, 64))) == self.validators[validator_index].addr
    assert ecrecover(sighash2,
                     as_num256(extract32(sig2, 0)),
                     as_num256(extract32(sig2, 32)),
                     as_num256(extract32(sig2, 64))) == self.validators[validator_index].addr
```

Casper overview

- If a sufficient number of messages of a certain type ("commits") reference a block, then that block is **economically finalized**


Not
conflicting
Conflicting

Safety of Casper

- Goal: it should not be possible to finalize two **conflicting** blocks without ~⅓ of validators triggering a slashing condition ("**accountable safety"**) and thus lose their deposit
- **Plausible liveness** (algorithm can't get "stuck")

Yoichi Hirai Follow
a convenience logician
Feb 25 · 7 min read

Formal methods on some PoS stuff

In Paris, I received a description of a distributed consensus mechanism. If the description below looks ambiguous or impenetrable, this post is for you. I banged my head to formal verification tools called Alloy and Isabelle/HOL to clarify my understanding (code).

```
Message types:

* commit(HASH, view), 0 <= view
* prepare(HASH, view, view_source), -1 <= view_source < view

Slashing conditions:

* [i] commit(H, v) REQUIRES 2/3 prepare(H, v, vs) for some
consistent vs
```

Yoichi Hirai Follow

a convenience logician

Mar 18 · 4 min read

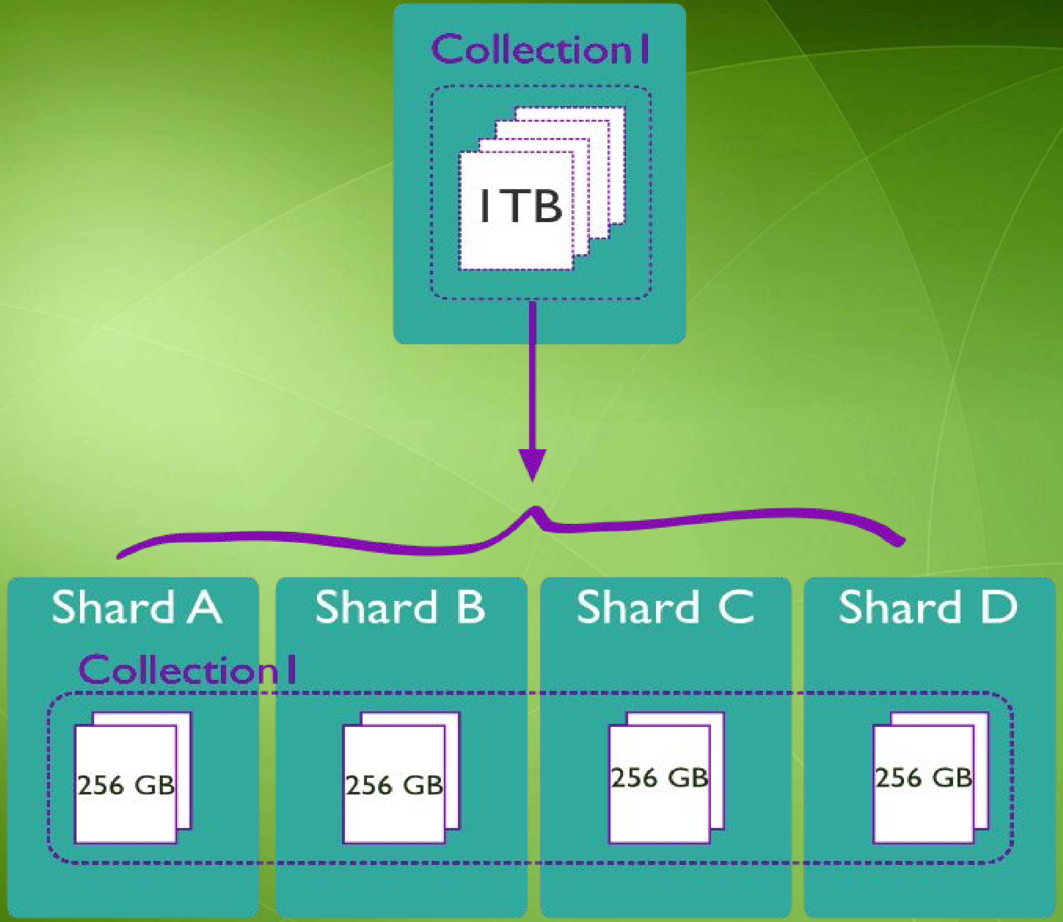
A mechanized safety proof for PoS with dynamic validators

I used a theorem prover Isabelle/HOL to check Vitalik's post about a proof-of-stake protocol that uses dynamic validator sets. (If you haven't seen, I did something similar for the simpler proof-of-stake protocol where the validator set is constant.) The proof script is available online.

Casper as overlay

1. Slashing conditions can be used as an “**overlay**” onto chain-based algorithms, prepares/commits can be used to agree on “checkpoints”
2. We can create a **hybrid fork choice rule**, roughly:
   a. Look for finalized checkpoints
   b. Look for checkpoints that are close to being finalized
   c. Longest chain

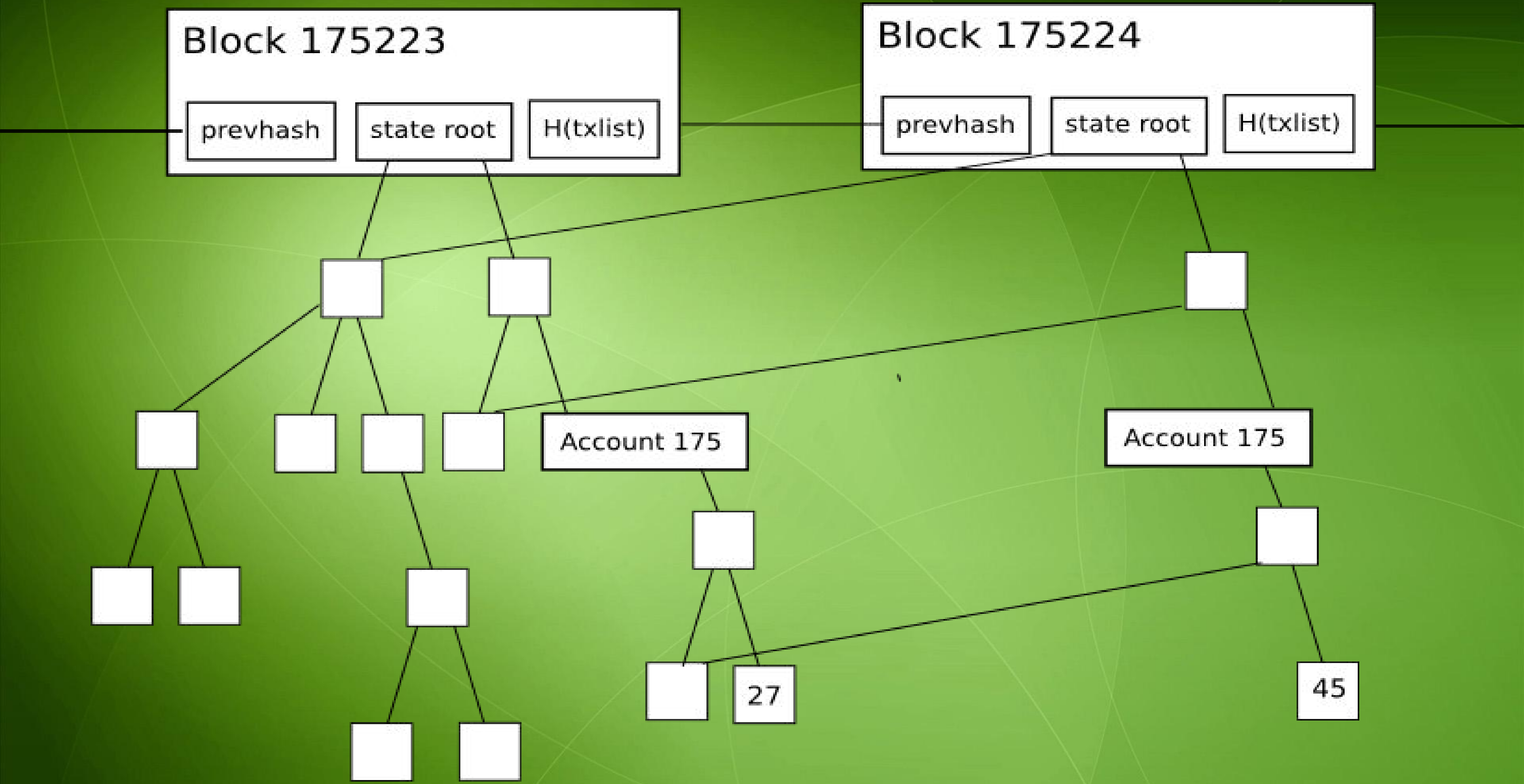
Let’s talk about sharding!


Collection1
1TB
Shard A
Shard B
Shard C
Shard D
Collection1
256 GB
256 GB
256 GB
256 GB

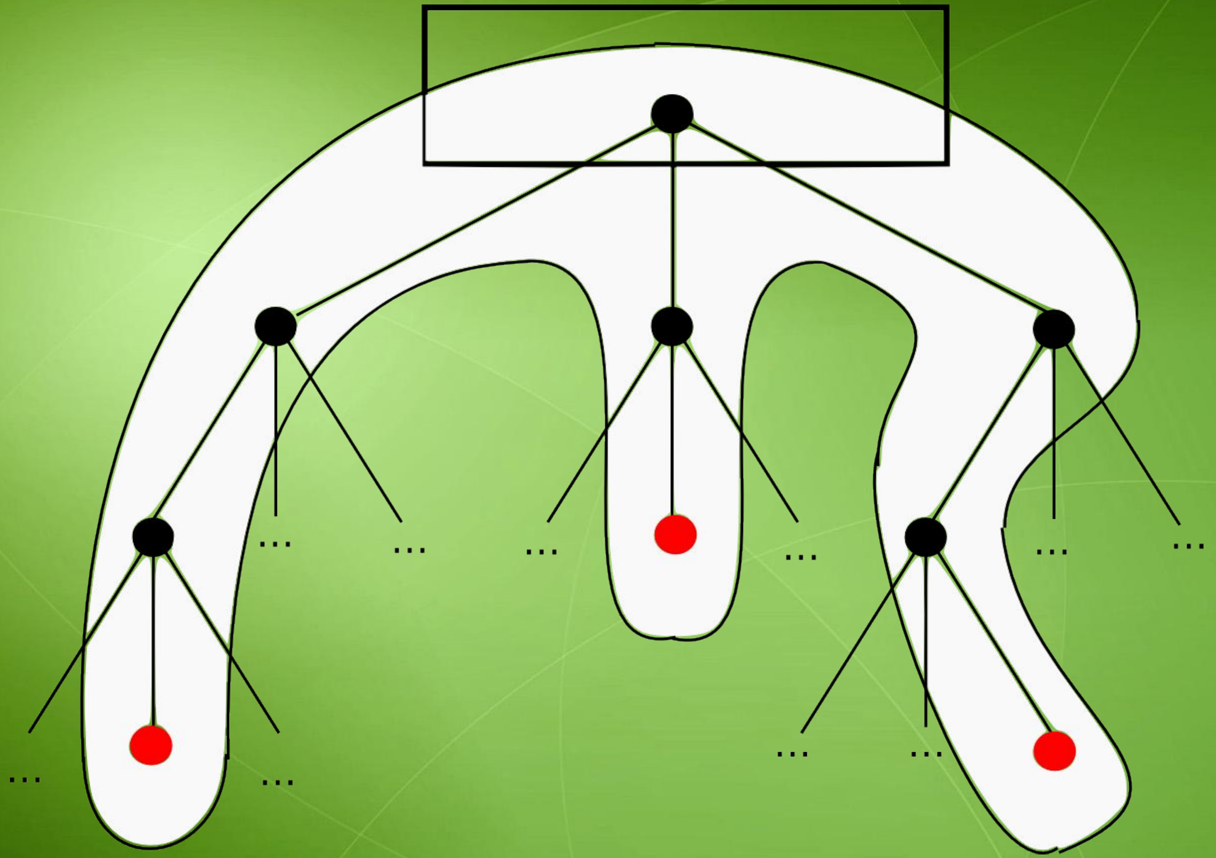
Properties of a sharded chain

- Very large amount of storage and transaction processing
- Every validator/user only interacts with a small portion of the chain
- Execution parallelized and split across all validators/nodes

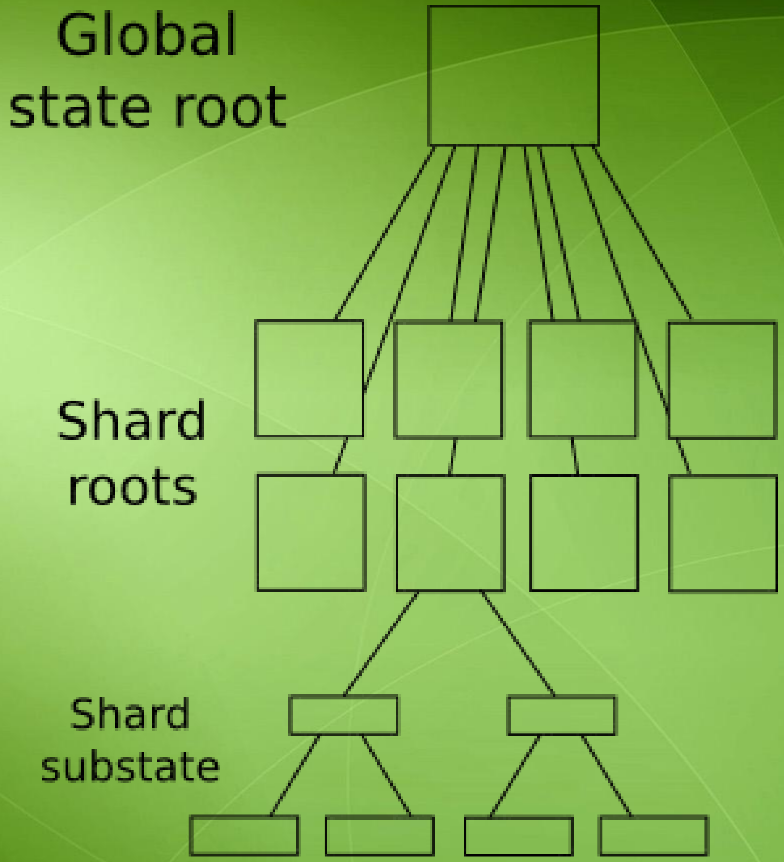
Light client protocols


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

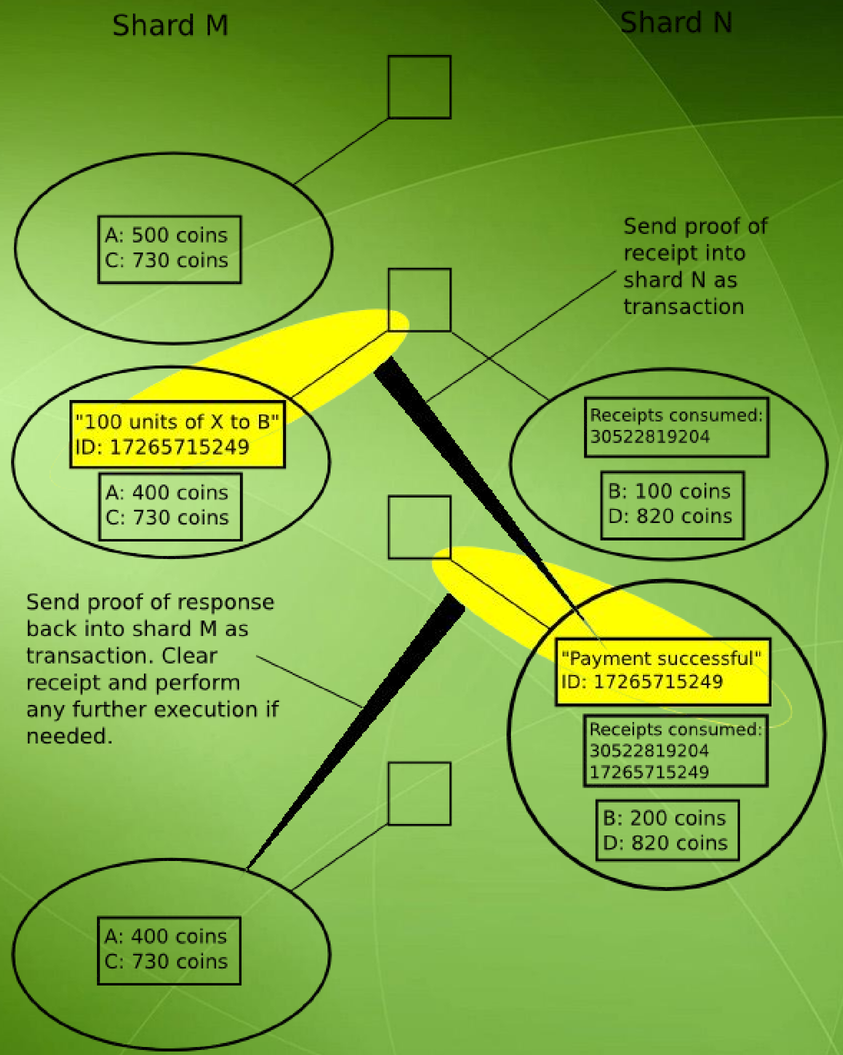
Light client protocols



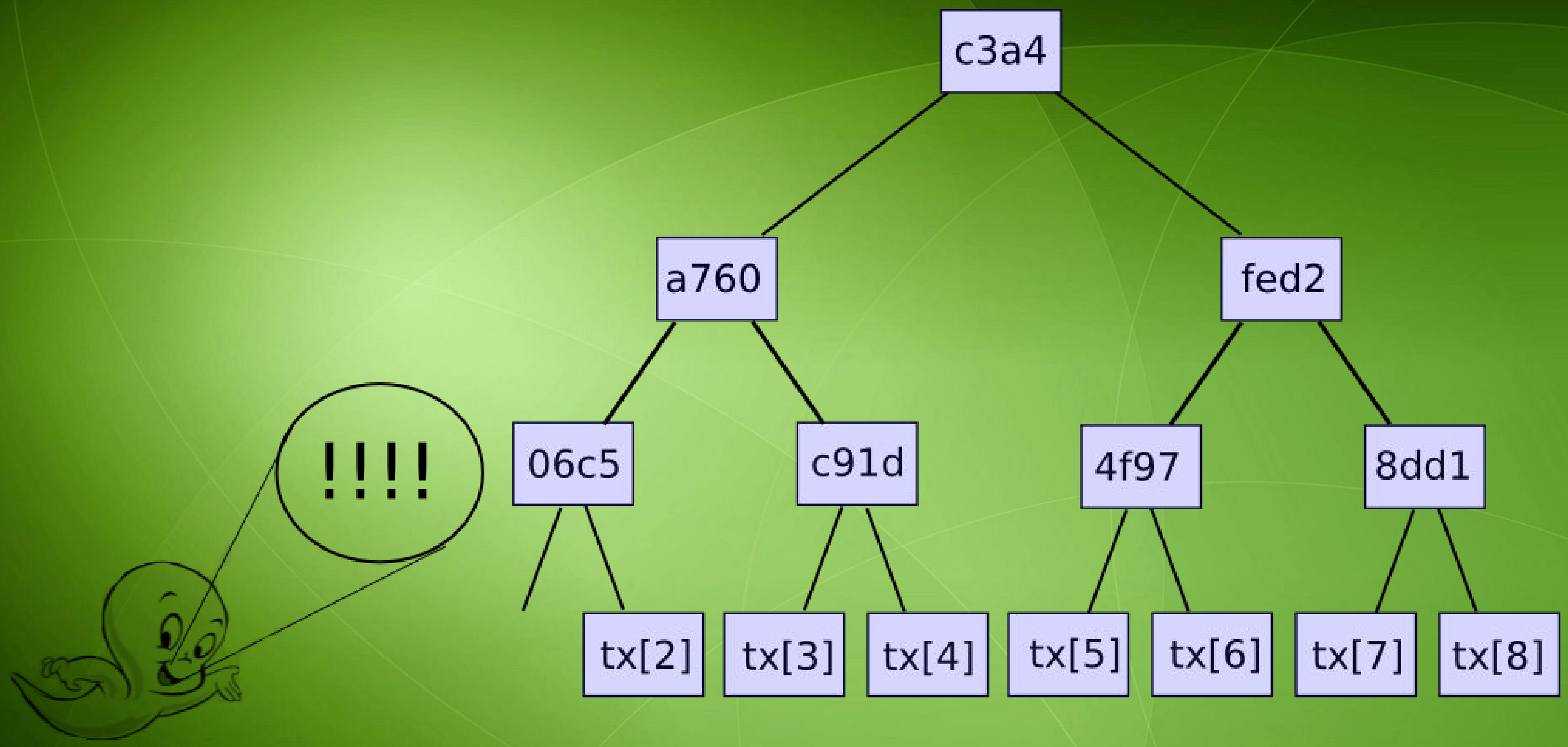
Idea behind sharding: every client, including validators, is a light client



Global
state root
Shard
roots
Shard
substate

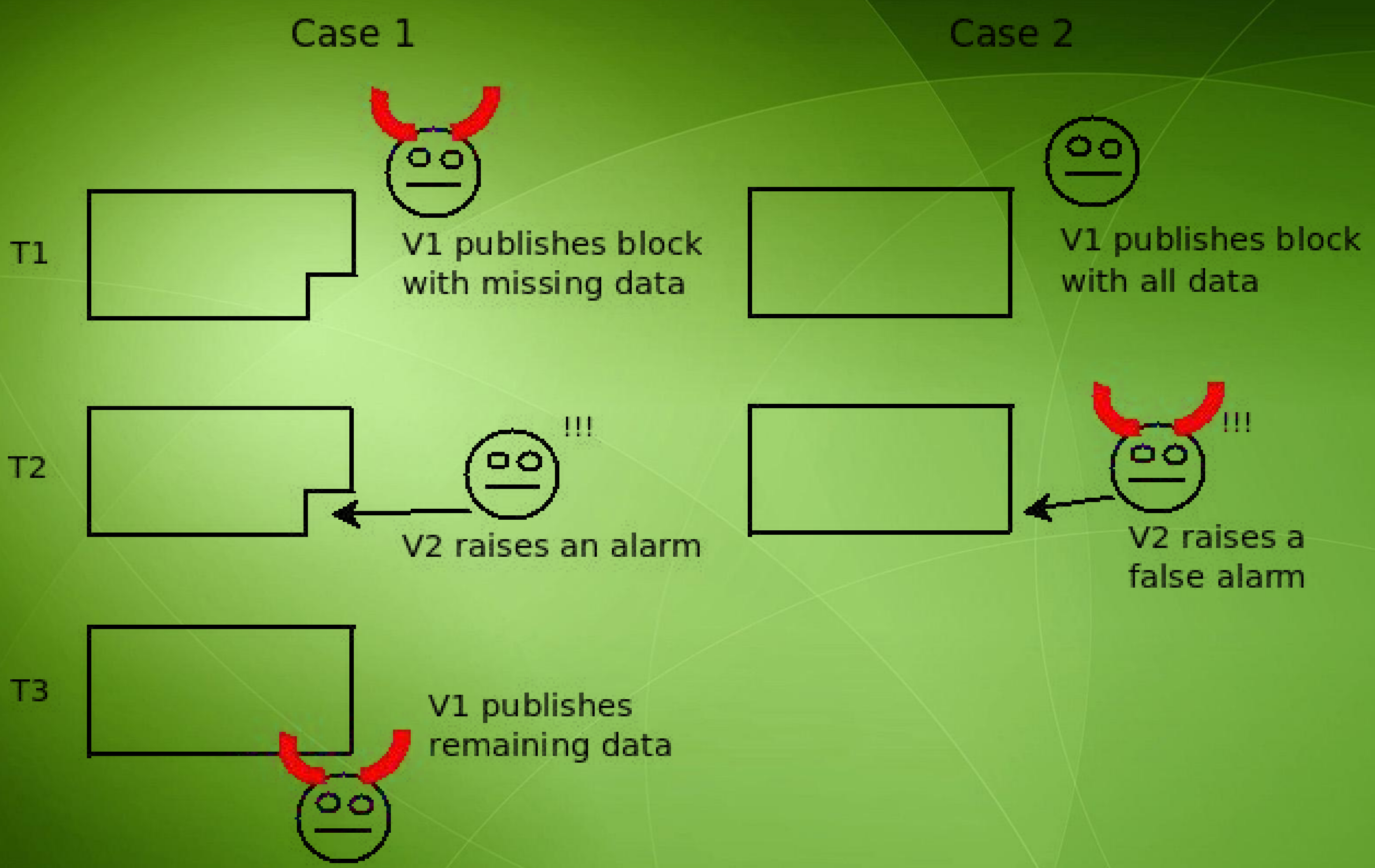

Shard M
Shard N
A: 500 coins
C: 730 coins
Send proof of
receipt into
shard N as
transaction
"100 units of X to B"
ID: 17265715249
A: 400 coins
C: 730 coins
Receipts consumed:
30522819204
B: 100 coins
D: 820 coins
Send proof of response
back into shard M as
transaction. Clear
receipt and perform
any further execution if
needed.
"Payment successful"
ID: 17265715249
Receipts consumed:
30522819204
17265715249
B: 200 coins
D: 820 coins
A: 400 coins
C: 730 coins

The Data Availability Problem

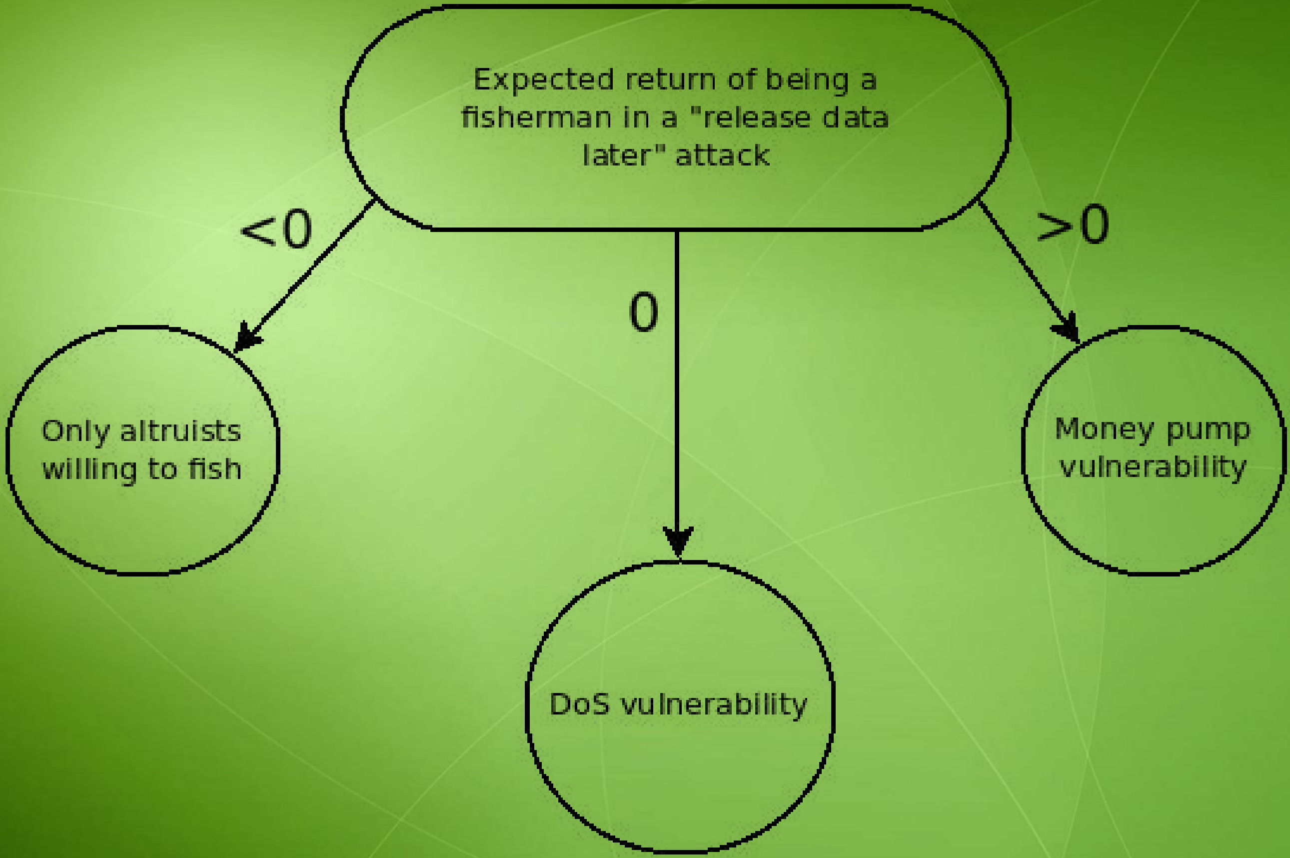

c3a4
a760
fed2
06c5
c91d
4f97
8dd1
tx[2]
tx[3]
tx[4]
tx[5]
tx[6]
tx[7]
tx[8]
!!!!

Data availability problem

- Incorrectness can be proven, even to a light client
  - “Fraud proofs”
- But data unavailability cannot
  - Data unavailability is **not a uniquely attributable fault**


Case 1
Case 2
T1
V1 publishes block with missing data
V1 publishes block with all data
T2
!!!
V2 raises an alarm
!!!
V2 raises a false alarm
T3
V1 publishes remaining data

The problem with “fishermen”



Data availability solution 1: rely on honesty

- Randomly sample validators, have them attest to availability
- Honest minority assumptions: honest X% can prevent unavailable data from being finalized, but malicious X% can permanently stall
- **Most protocols in academia that attempt this only get this far**

Data availability solutions: uncoordinated majority

- Introduce 0.001% chance any challenge will be immediately Schelling-voted on by the entire network
- Being a fisherman is now like playing the lottery, but positive-EV

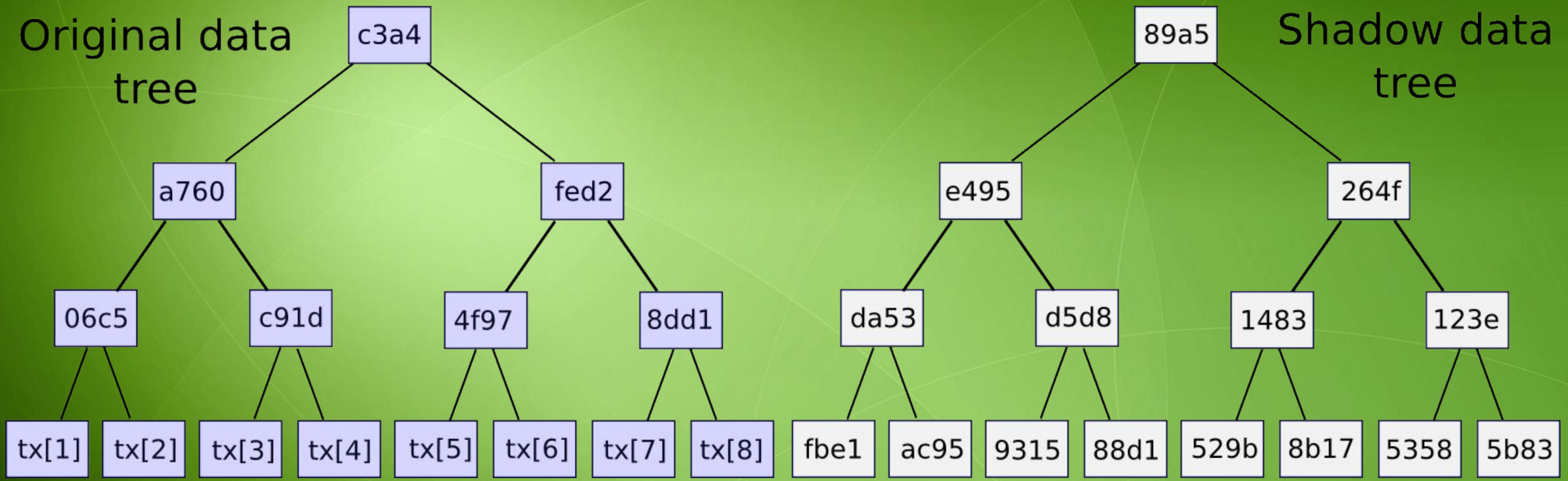
Data availability solutions: nearly trustless

- Client-side random sampling
  - Select 100 random indices, try to download the data at those indices, accept if 100/100 pass
- Problem: works against attackers that withhold 100% or 50%, but what if an attacker withholds 0.01%

Erasure codes

Erasure codes

- Encode a file with M chunks into a larger file of N chunks
- Any M of the N chunks can be used to recover the original
- Goal: convert the "100% data availability" problem into a "50% data availability" problem, that *can* be solved through random sampling

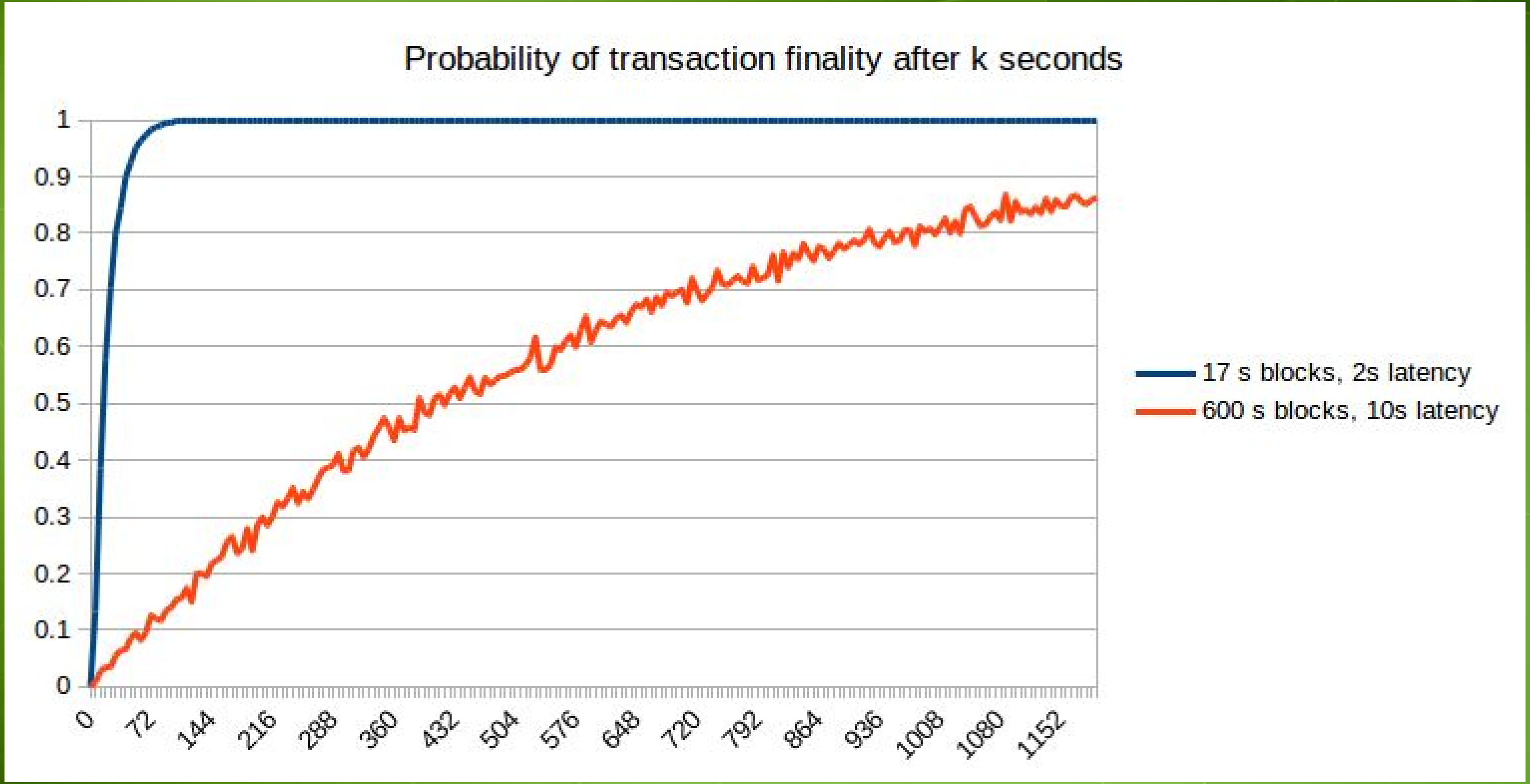

Original data tree
c3a4
a760
fed2
06c5
c91d
4f97
8dd1
tx[1]
tx[2]
tx[3]
tx[4]
tx[5]
tx[6]
tx[7]
tx[8]
Shadow data tree
89a5
e495
264f
da53
d5d8
1483
123e
fbe1
ac95
9315
88d1
529b
8b17
5358
5b83

How would a light client work?

1. Get all block headers
2. Verify consensus, ignore headers that are not backed by consensus
3. Randomly sample to verify data availability
4. Listen for fraud proofs
   a. Invalid data / false state transition execution
   b. Inconsistency in erasure code
5. If all checks pass, accept the block/state as valid

Can we reduce block times to 500ms?

Question 1: do you mean **block time** or **time-to-finality**?



Probability of transaction finality after k seconds
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
72
144
216
288
360
432
504
576
648
720
792
864
936
1008
1080
1152
17 s blocks, 2s latency
600 s blocks, 10s latency

Fast block times and centralization risk

- Challenge: avoid creating large incentives for nodes to gather in the same data center
- PoW: Poisson process, orphan rate
  - ~0.23% revenue gain per 100ms latency drop
  - Down from 0.7% due to uncle mechanism
- PoS: can rely on hard time limits (eg. 4s), can show fairness given strong network+clock synchrony assumptions

Time to finality

(in seconds)

Decentralization

(number of validators)

$$f * o \geq d * 2/3$$

Overhead

(consensus messages per second)

Can we reduce the overhead with clever tricks?

- Signatures get validated for two reasons
  - To determine which blocks have consensus
  - Incentivization
- The first can be dealt with through random sampling
- The second can be dealt with through sharding

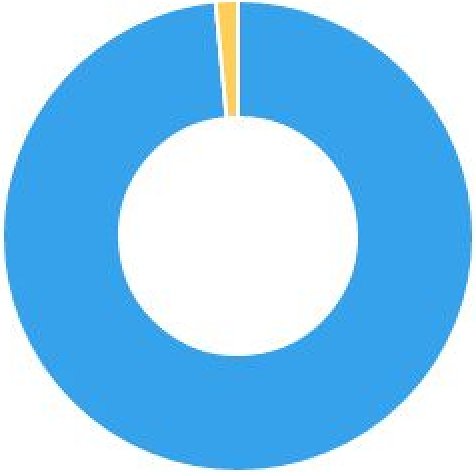
Let’s talk about in-protocol governance!

Last Block: 3428340

NO

1.5188%

12954.2467 ether



YES

98.4812%

839961.3717 ether

DAOs / in-protocol governance

- Area of increasing interest (eg. Tezos, Dash, Bitshares)
- Avoid concerns over centralization in off-chain governance by creating an on-chain DAO to govern a protocol
  - Alternatively, DAOs can govern second-layer protocols, eg. MakerDAO
- Voting between a set of participants usually used as governance mechanism

Problem 1: voter apathy

Vote	Voter participation
Bitshares DPOS delegate voting[1]	5-18%
DAO carbonvote[2]	4.5%
Most active DAO proposal vote[3]	9.62%
Dash masternode votes[4]	0-50%

1. https://bitcointalk.org/index.php?topic=916696.330;imode
2. http://carbonvote.com
3. https://daostats.github.io/proposals.html
4. https://www.dash.org/forum/threads/how-to-strenghten-dashs-voting-system.7309/

Problem 2: economic security

- Low incentive to vote
- Very vulnerable to bribe attacks
  - Bribe attacks in reality: favorable interest rates from stake pools and exchanges

Solutions

- Make voting accountable
  - Coin lock voting
  - Futarchy
- Use social layer to explicitly account for “attacks” against voting
  - Governance can *use* voting, but it must be loosely coupled

Privacy

The verification / privacy tradeoff

- Intuition: if you want N nodes to help you verify that something is being done correctly, you need N nodes to see the data

Circumventing the verification / privacy tradeoff

- Keep data off the chain
  - Proof of existence
  - State channels
- Using fancy cryptography
  - Ring signatures
  - ZK-SNARKs
    - And friends (eg. STARKs)

Zk-SNARKs: Under the Hood

This is the third part of a series of articles explaining how the technology behind zk-SNARKs works; the previous articles on quadratic arithmetic programs and elliptic curve pairings are required reading, and this article will assume knowledge of both concepts. Basic knowledge of what zk-SNARKs are and what they do is also assumed. See also Christian Reitwiessner's article here for another technical introduction.

In the previous articles, we introduced the quadratic arithmetic program, a way of representing any computational problem with a polynomial equation that is much more amenable to various forms of mathematical trickery. We also introduced elliptic curve pairings, which allow a very limited form of one-way homomorphic encryption that lets you do equality checking. Now, we are going to start from where we left off, and use elliptic curve pairings,

Summary: it’s complicated but it works*

* Problem 1: trusted setup requirement
* Problem 2: takes 40s to create a proof
* Problem 3: relatively expensive to verify

ZK-SNARKs: how do you use them?

- The most computationally complex parts of verifying a ZK-SNARK will be available in Metropolis as precompiles
- Simple application: create a privacy-preserving “wrapper” of any token (ETH, REP, MKR, etc) that supports encrypted transfers
- More complex application: verify execution of arbitrary contract code, with encrypted transactions and state

Problems

- ***Someone*** has to have the decryption key to every state object
  - Often not a problem, eg. in a currency, users can have the decryption key to their own balances
  - Sometimes state is user-specific, so there is a problem
- Technological impediments (see above)

Other weird problems

Other weird problems

- Truth oracles
  - Semi-centralized multisig oracles
  - Augur oracle
- Provably fair random number generation
  - My favorite: timelock
- Special case of oracle: price (even if approximate)
  - Use case: stablecoins
  - Use case: gas limit / block size policy

Conclusion: life is hard*

*But we know much more than we did in 2014.