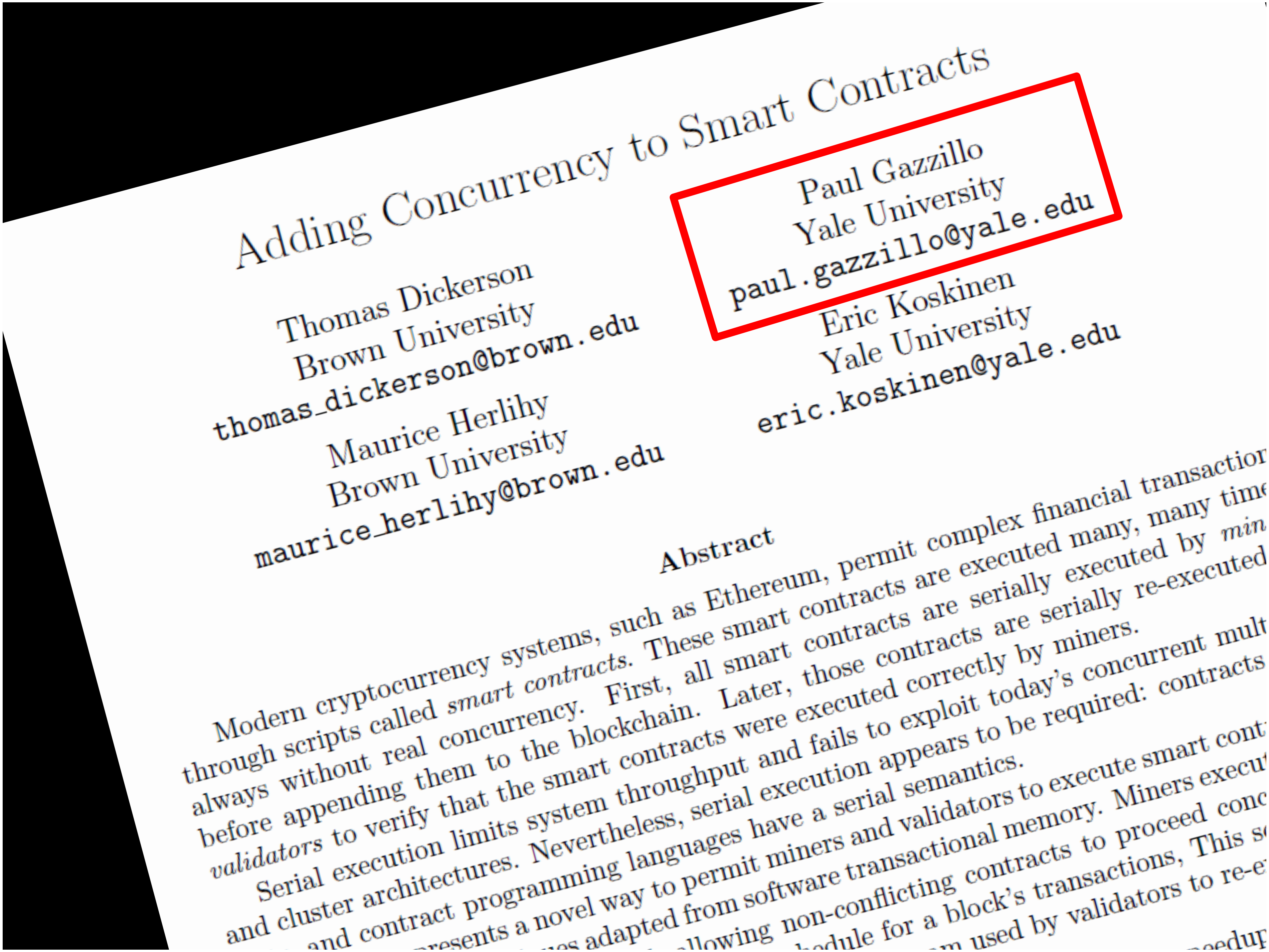


Adding Concurrency to Smart Contracts

Thomas Dickerson
Brown University
thomas_dickerson@brown.edu

Paul Gazzillo
Yale University
paul.gazzillo@yale.edu

Maurice Herlihy
Brown University
maurice_herlihy@brown.edu

Eric Koskinen
Yale University
eric.koskinen@yale.edu

Abstract

Modern cryptocurrency systems, such as Ethereum, permit complex financial transactior
through scripts called *smart contracts*. These smart contracts are executed many, many time
always without real concurrency. First, all smart contracts are serially executed by *min*
before appending them to the blockchain. Later, those contracts are serially re-executed
validators to verify that the smart contracts were executed correctly by miners.

Serial execution limits system throughput and fails to exploit today's concurrent mult
and cluster architectures. Nevertheless, serial execution appears to be required: contracts
and contract programming languages have a serial semantics.
resents a novel way to permit miners and validators to execute smart cont
ues adapted from software transactional memory. Miners execu
llowing non-conflicting contracts to proceed conc
hedule for a block's transactions, This s
m used by validators to re-e
speedup

Bitcoin

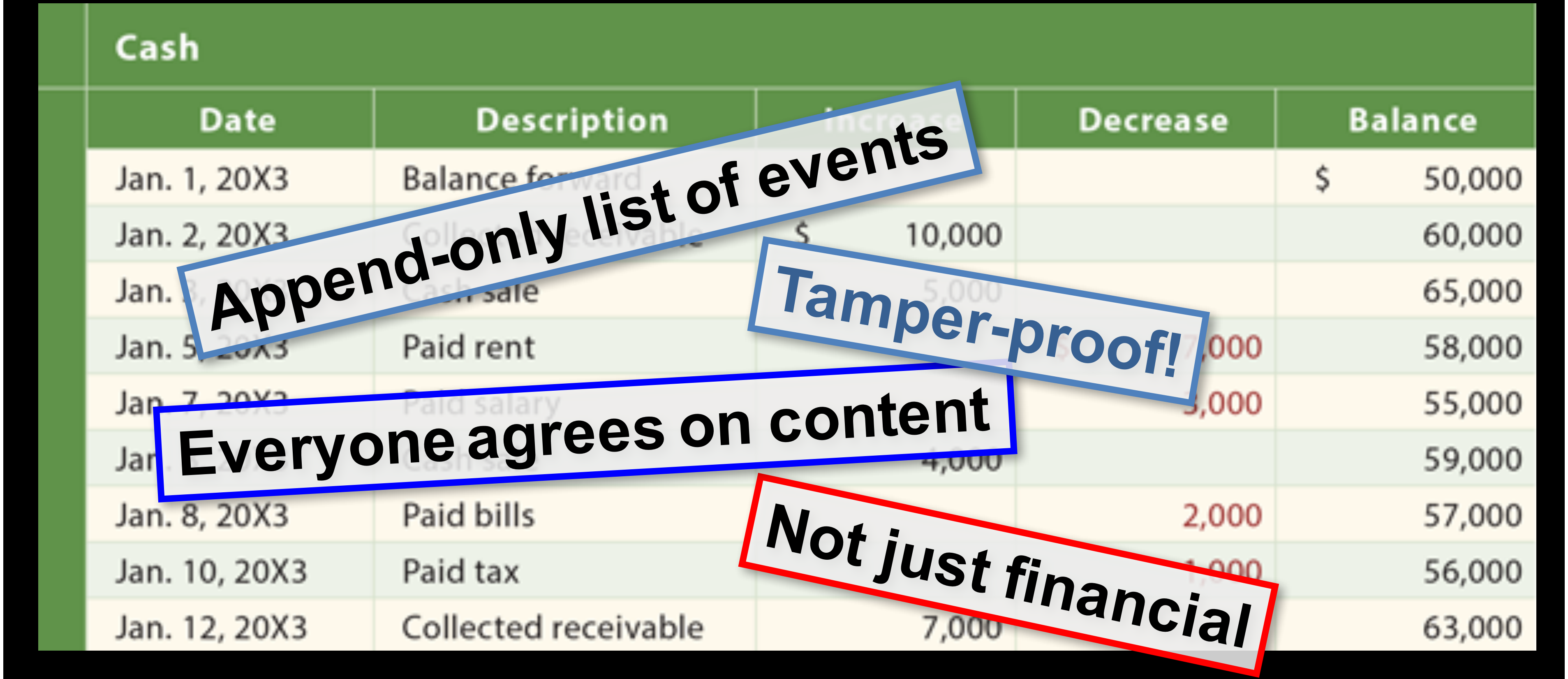
Satoshi Nakamoto 2008

Cryptocurrency

No central authority

Anyone can participate

Abstraction:
Distributed Ledger



Cash

Date	Description	Increase	Decrease	Balance
Jan. 1, 20X3	Balance forward			$ 50,000
Jan. 2, 20X3	Collected receivable	$ 10,000		60,000
Jan. 3, 20X3	Cash sale	5,000		65,000
Jan. 5, 20X3	Paid rent		7,000	58,000
Jan. 7, 20X3	Paid salary		3,000	55,000
Jan. [illegible]	Cash sale	4,000		59,000
Jan. 8, 20X3	Paid bills		2,000	57,000
Jan. 10, 20X3	Paid tax		1,000	56,000
Jan. 12, 20X3	Collected receivable	7,000		63,000

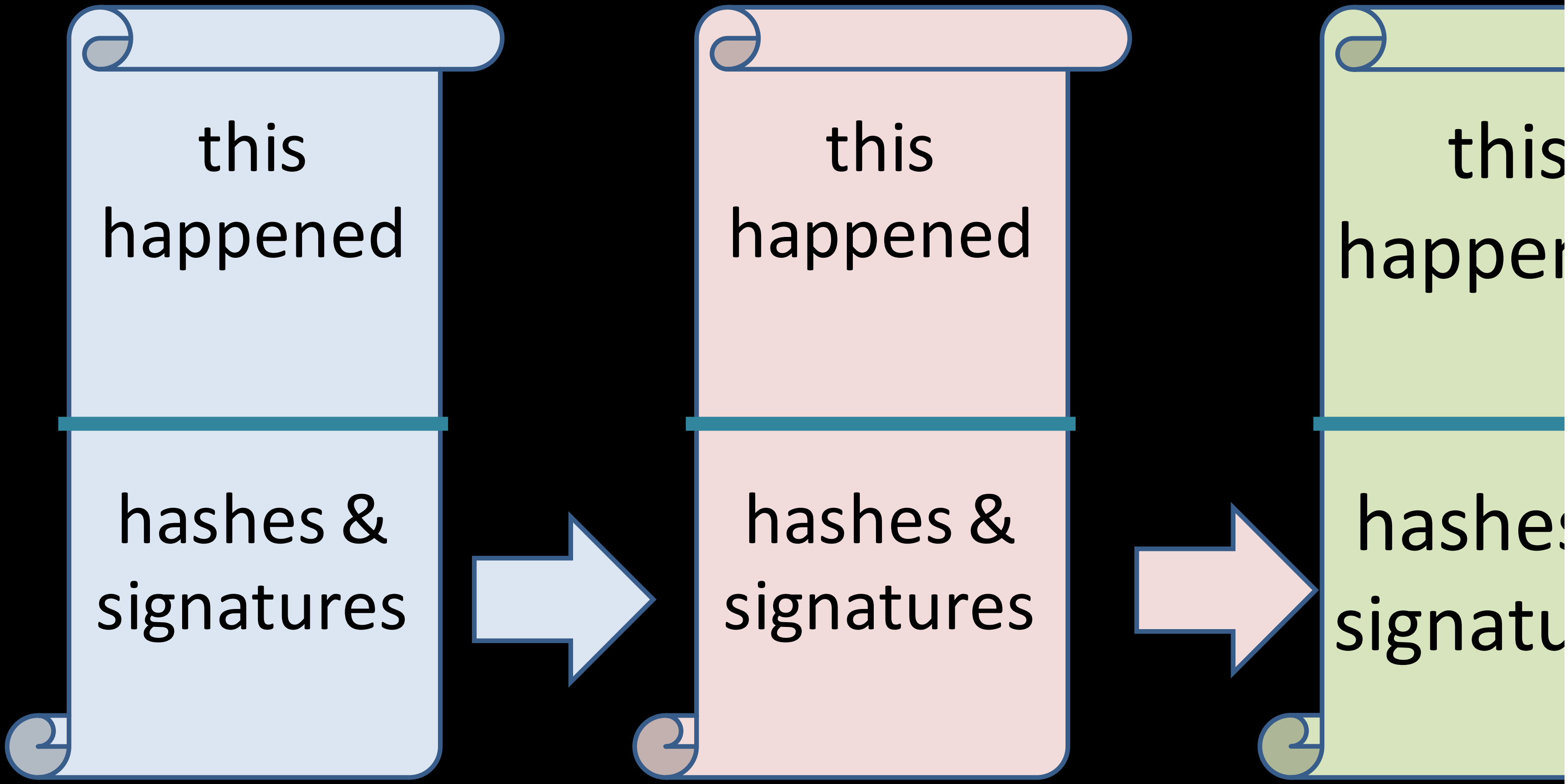
Append-only list of events

Tamper-proof!

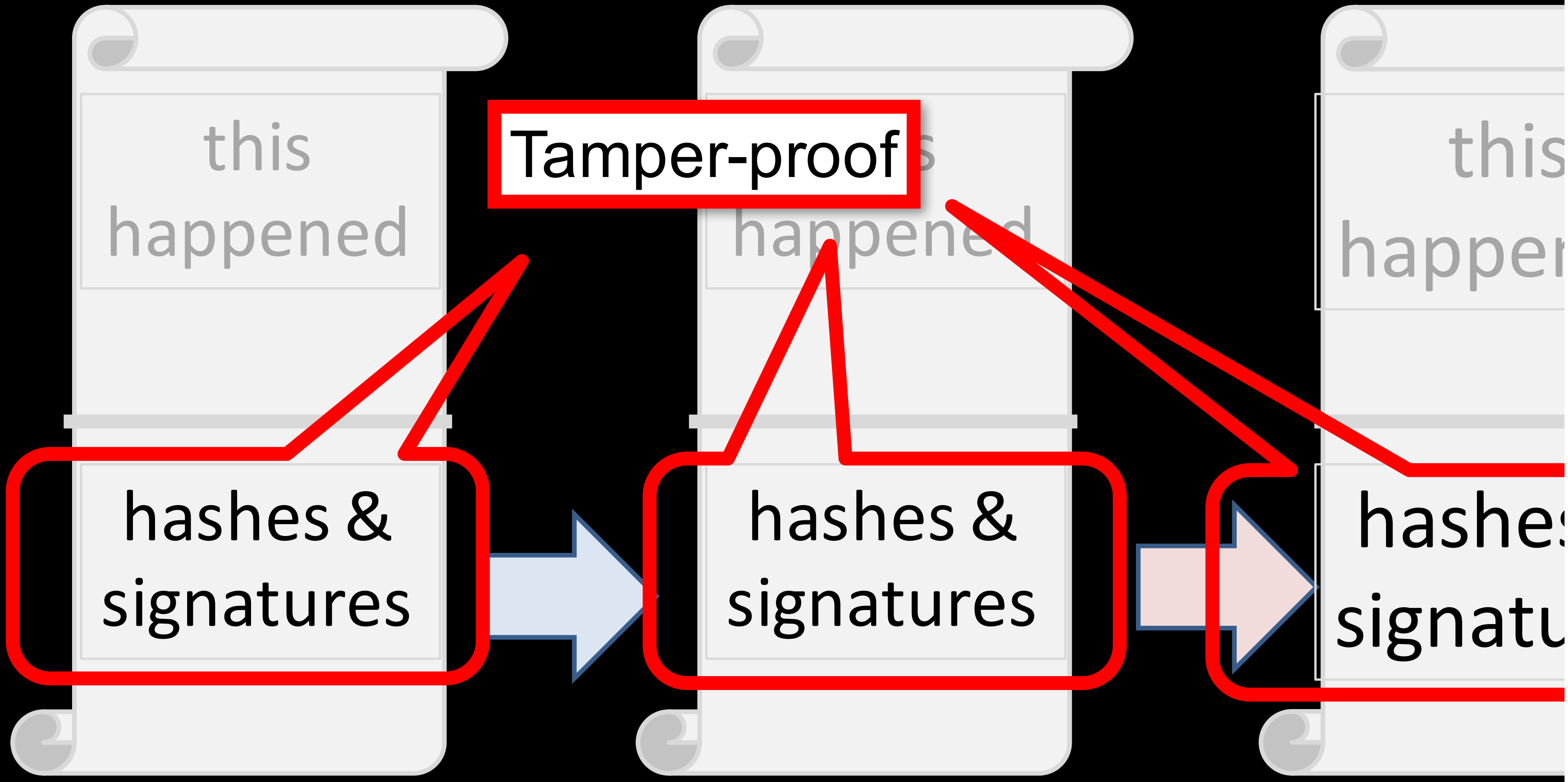
Everyone agrees on content

Not just financial

Implementation: Blockchain



Implementation: Blockchain



this happened

hashes & signatures

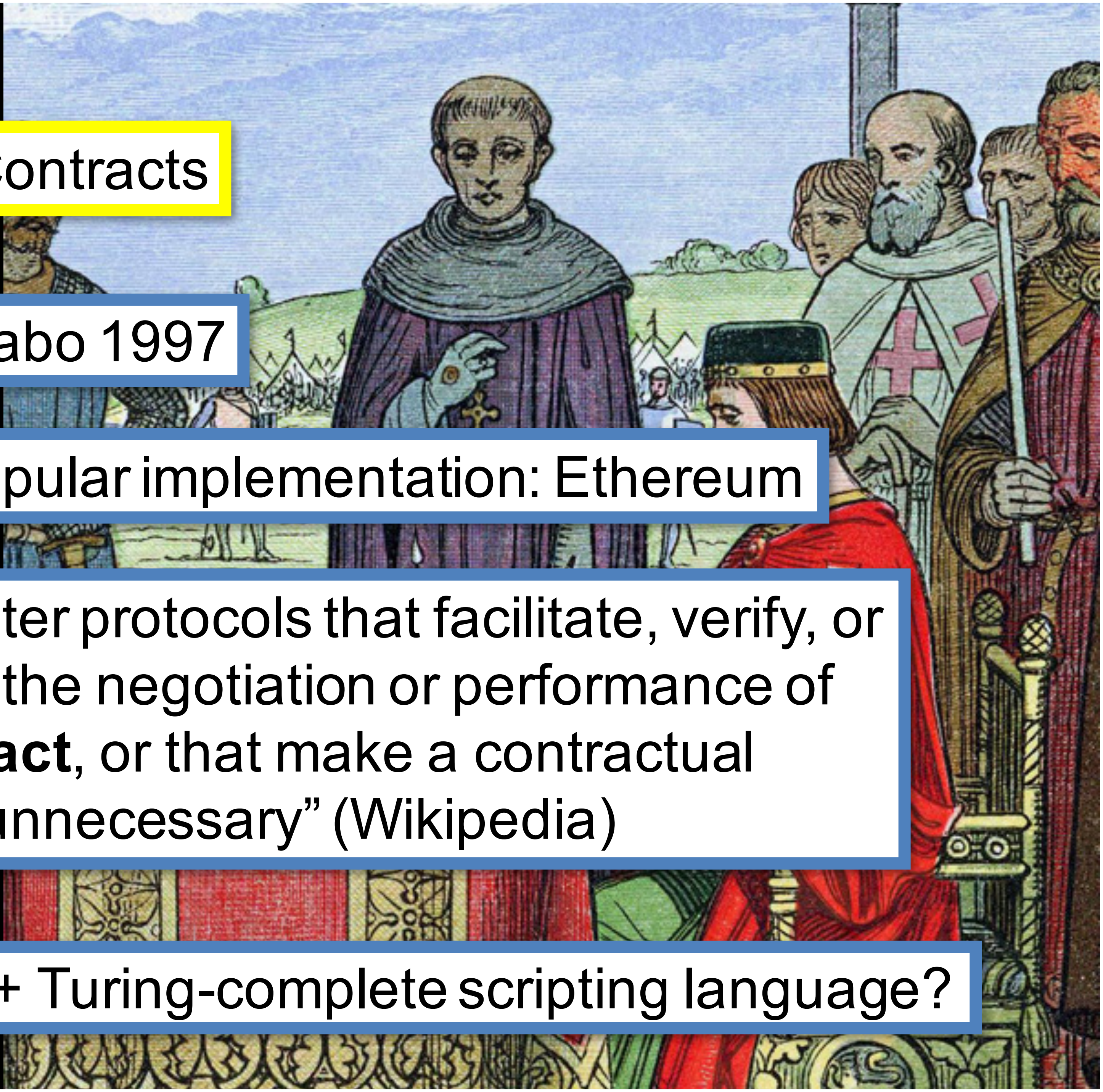
Tamper-proof

this happened

hashes & signatures

this happen

hashes signatu



Smart Contracts

Nick Szabo 1997

Most popular implementation: Ethereum

“Computer protocols that facilitate, verify, or enforce the negotiation or performance of a **contract**, or that make a contractual clause unnecessary” (Wikipedia)

Ledger + Turing-complete scripting language?

```
contract Ballot {
  mapping(address => Voter)
     public voters;
     … // more state decls
  function vote(uint proposal)
    Voter sender = voters[msg.sender];
    if (sender.voted)
      throw;
    sender.voted = true;
    sender.vote = proposal;
    proposals[proposal].voteCount
      += sender.weight;
  }
  …
}
```

Looks like an object in a language

```
contract Ballot {
  mapping(address => Voter)
     public voters;
     … // more state decls
  function vote(uint proposal)
    Voter sender = voters[msg.sender];
    if (sender.voted)

    sender.voted = true;

    proposals[proposal].voteCount

  }
  …
}
```

Long-lived state

Built-in data types: maps, arrays, scalars.

Tracks who can vote, who voted, choices.

```
contract Ballot {
  mapping(address => Voter)
     public voters;
     … // more state decls
  function vote(uint proposal)
    Voter sender = voters[msg.sender];
    if (sender.voted)

    sender.voted = true;
                                l;
    proposals[proposal].voteCount
      += sender.weight;
  }
  …
}
```

Functions to manipulate state

Vote for a particular proposal

```
contract Ballot {
  mapping(address => Voter)
     public voters;
     … // more state decls
  function vote(uint proposal)
    Voter sender = voters[msg.sender];
    if (sender.voted)
      throw;
    sender.voted = true;
    sender.vote = proposal;
    proposals[proposal].voteCount
      += sender.weight;
  }
  …
}
```

No voting twice

```
contract Ballot {
  mapping(address => Voter)
     public voters;
     … // more state decls
  function vote(uint p
    Voter sender = voters[msg.sender];
    if (sender.voted)
      throw;
    sender.voted = true;
    sender.vote = proposal;
    proposals[proposal].voteCount
      += sender.weight;
  }
  …
}
```

Record vote

```
contract Ballot {
  mapping(address => Voter)
     public voters;
     … // more state decls
  function vote(uint proposal)
    Voter sender = voters[msg.sender];
    if (sender.voted)

    sender.voted = true;
    sender.vote = proposal;
    proposals[proposal].voteCount
       += sender.weight;
  }
  …
}
```

On a blockchain this is a shared object!

All contract code executed *sequentially*

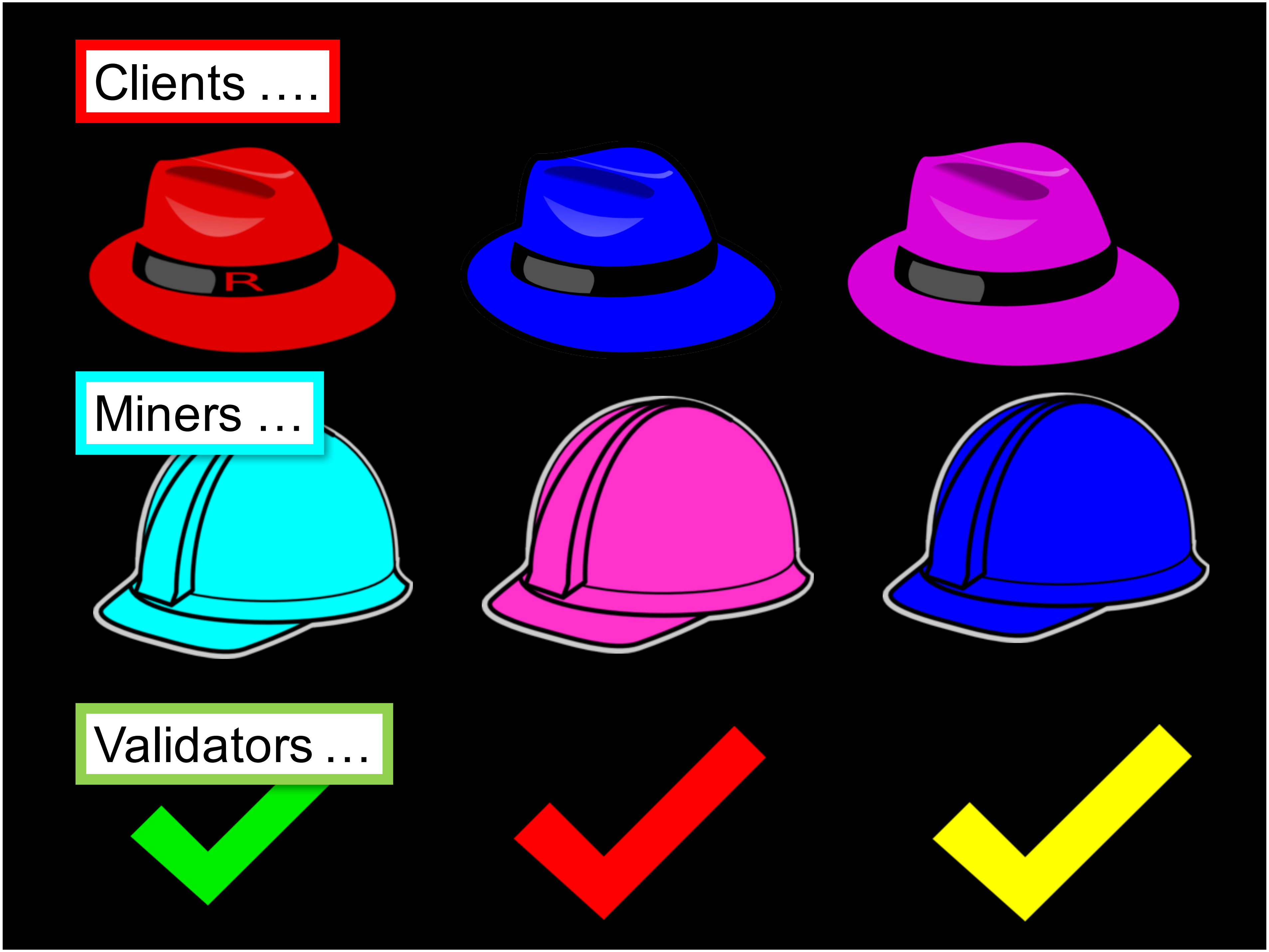
Every transaction executed sequentially by *everyone*

No concurrency control built in to contract language

Big idea #1: permit parallel execution, adapting STM techniques, i.e., speculative execution with rollback

Big idea #2: publish concurrent schedules to the blockchain for everyone to exploit parallelism

Smart Contracts on the Blockchain

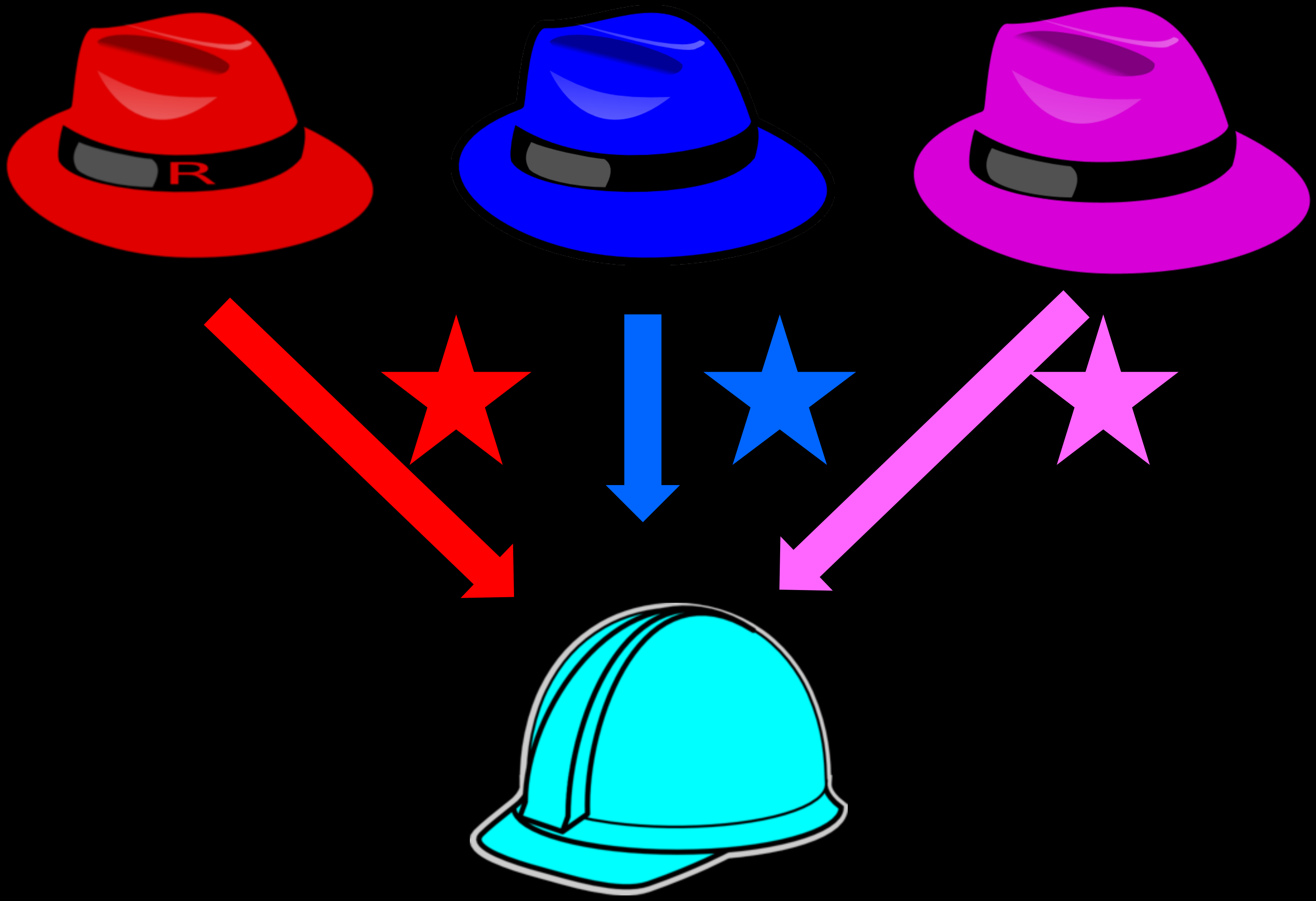


Clients ….

Miners …

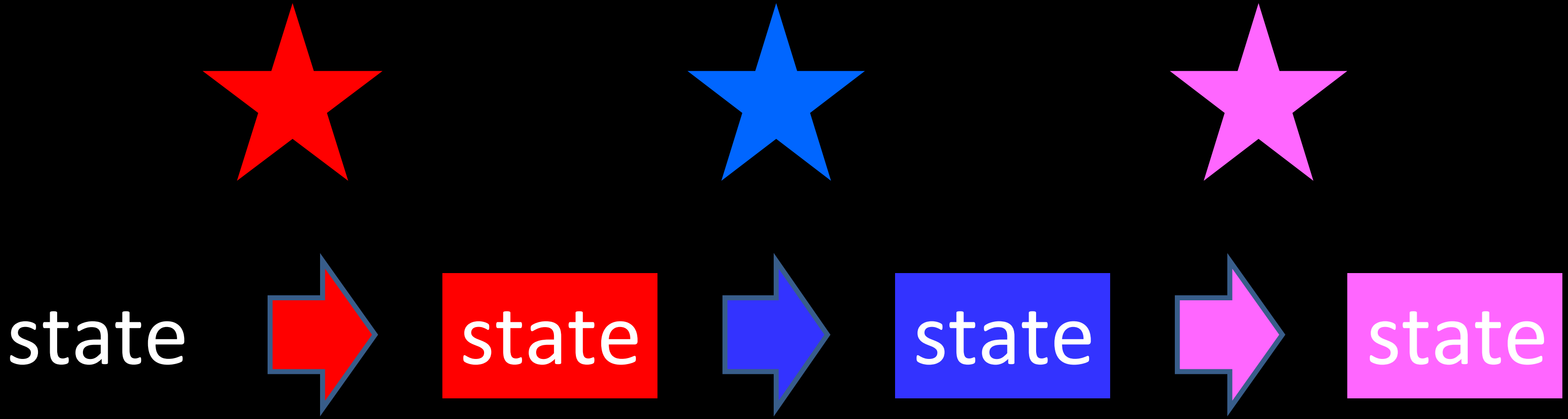
Validators …

Clients send transactions & contracts to miners



Miners collect transactions…

Apply them one-at-a-time to compute new state

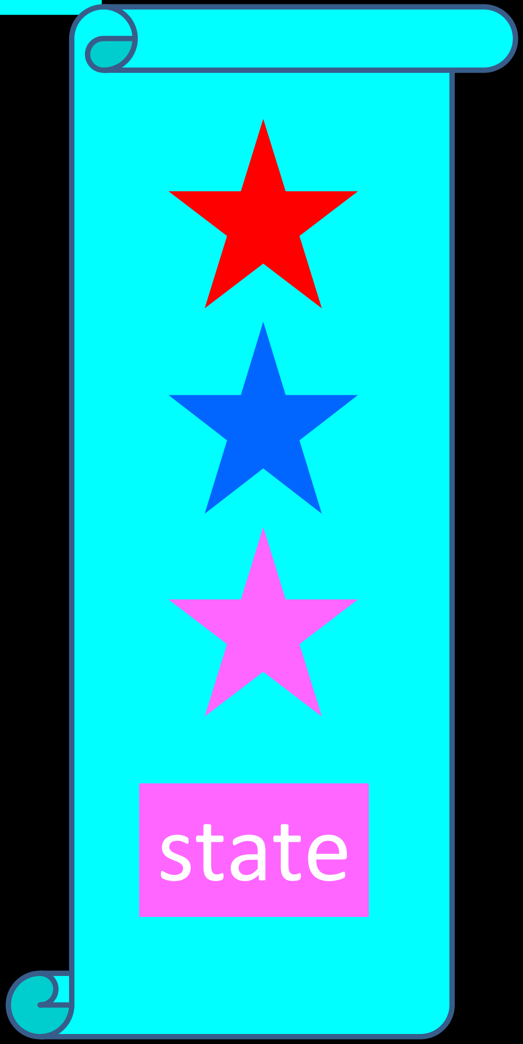

state

state

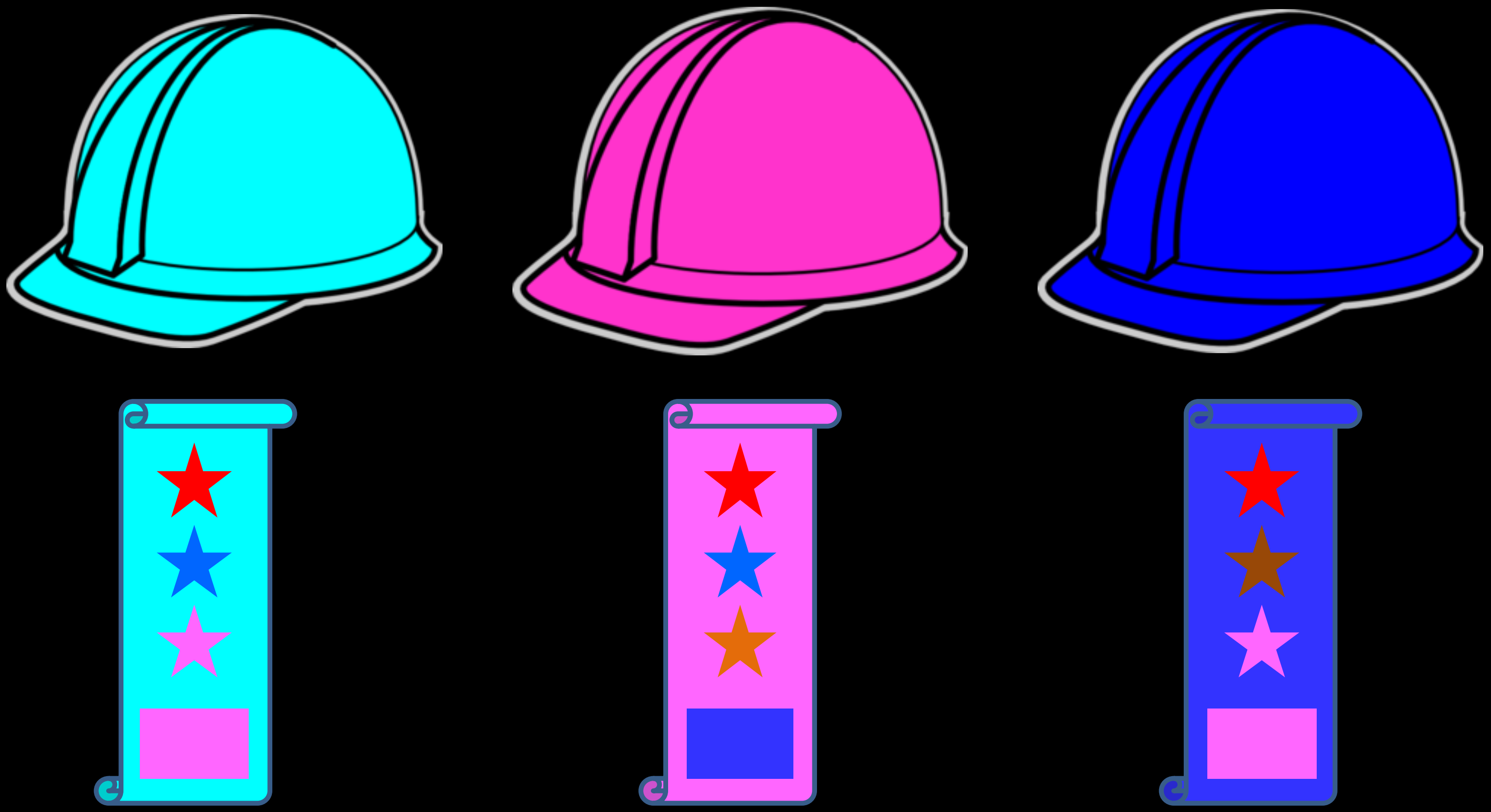
state

state

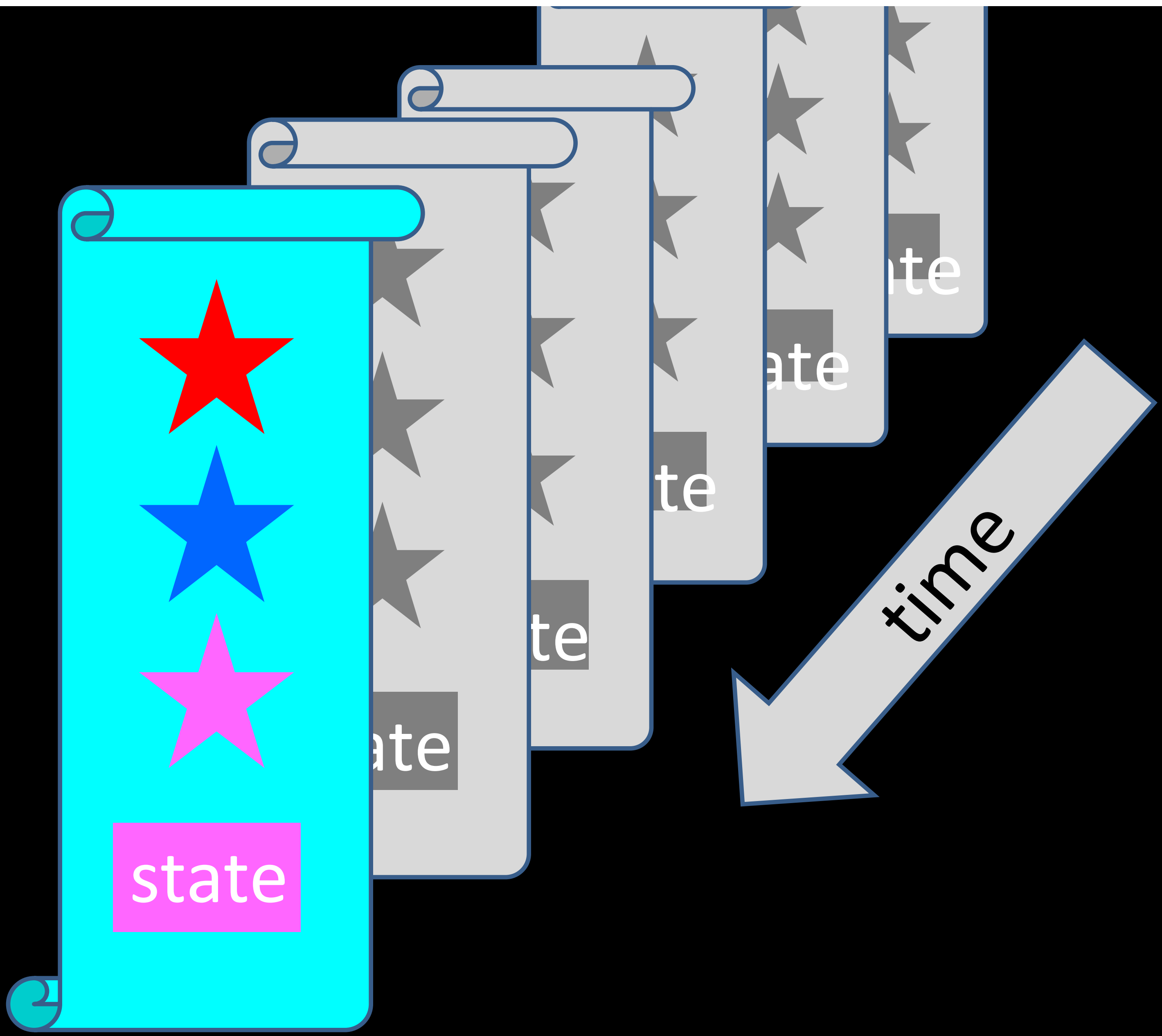
Block has contracts & new state



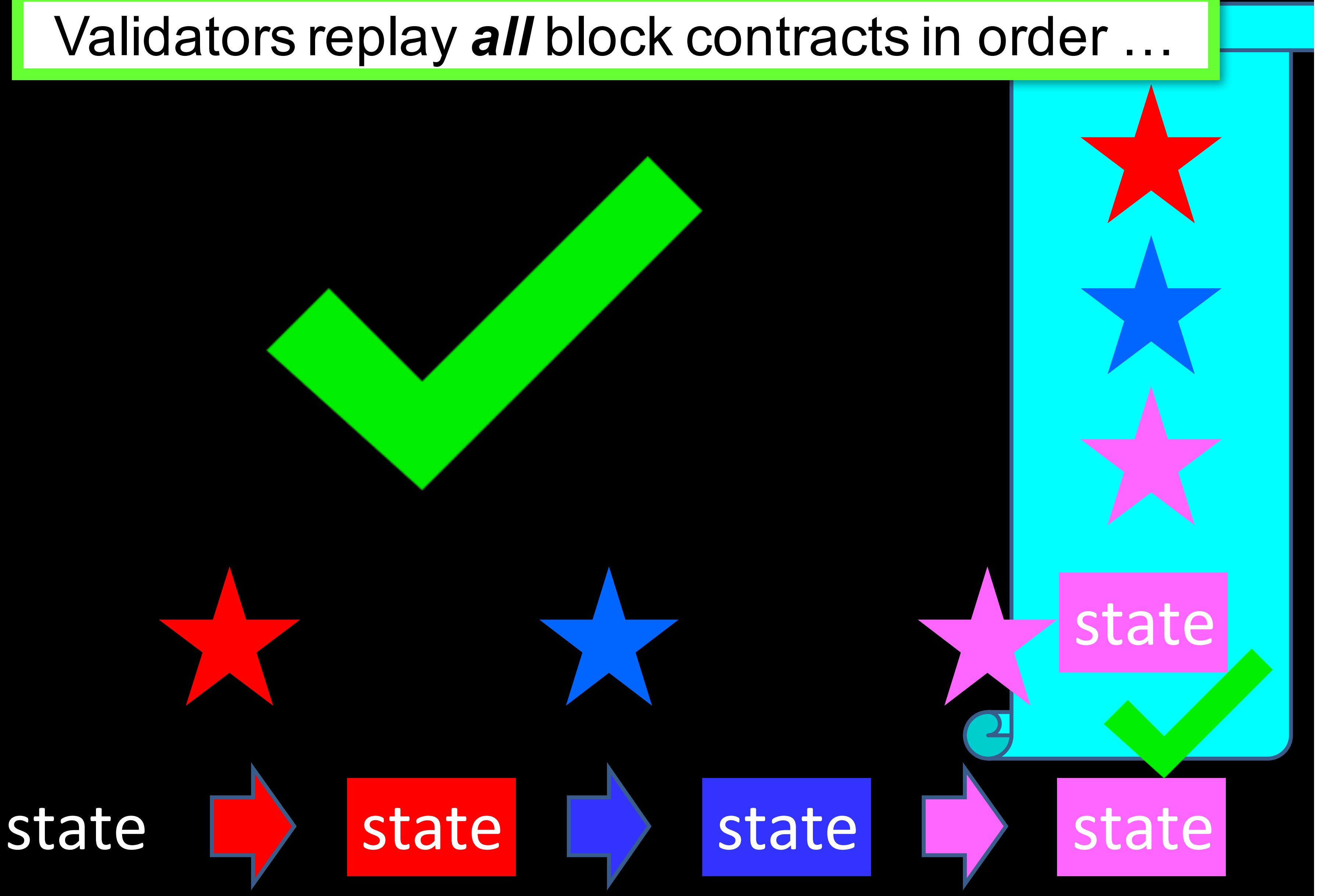
There can only be one…



Miners compete to append *their* new block to the chain


state
ate
te
te
ate
ate
time

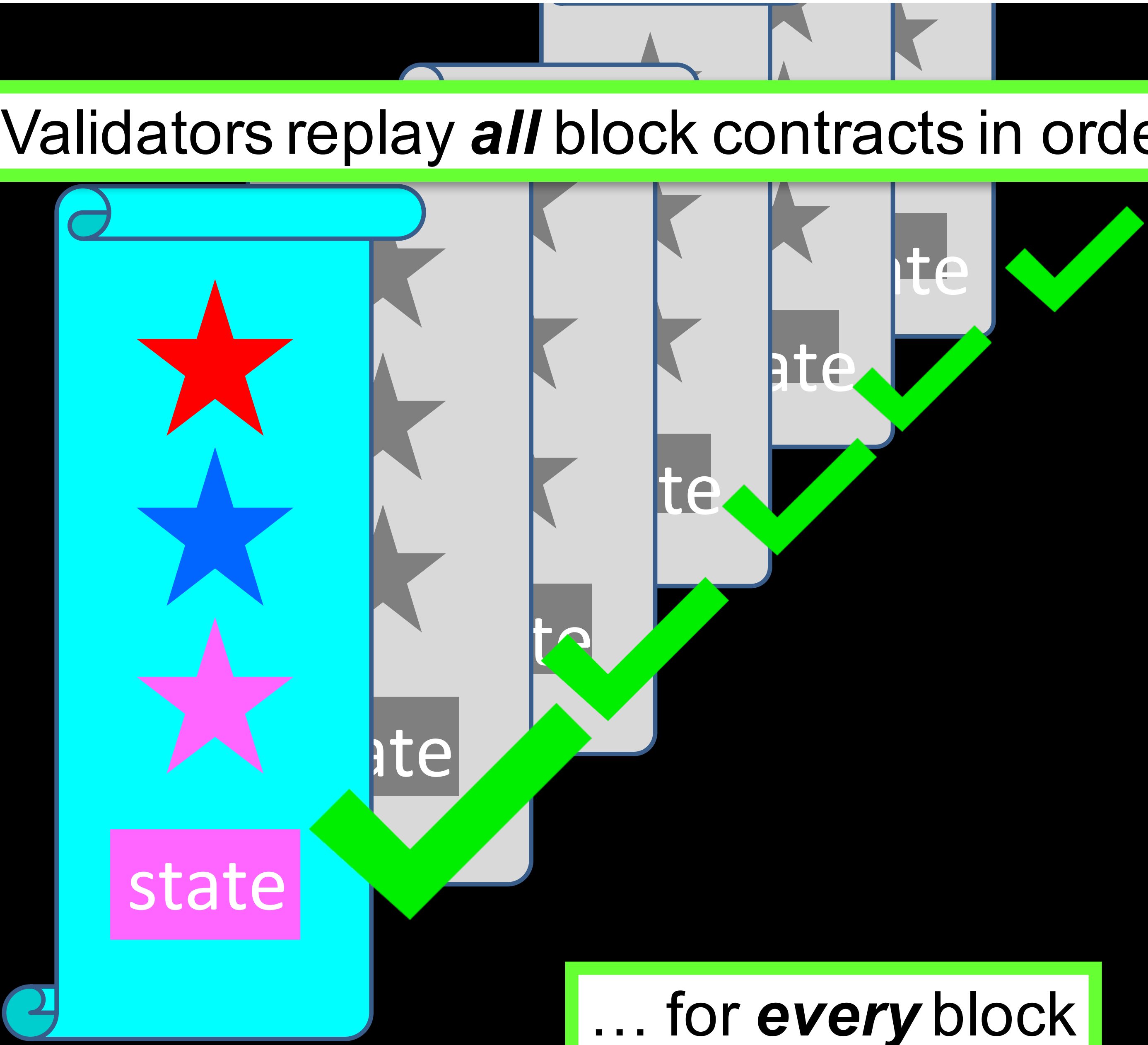
Validators replay ***all*** block contracts in order …



state

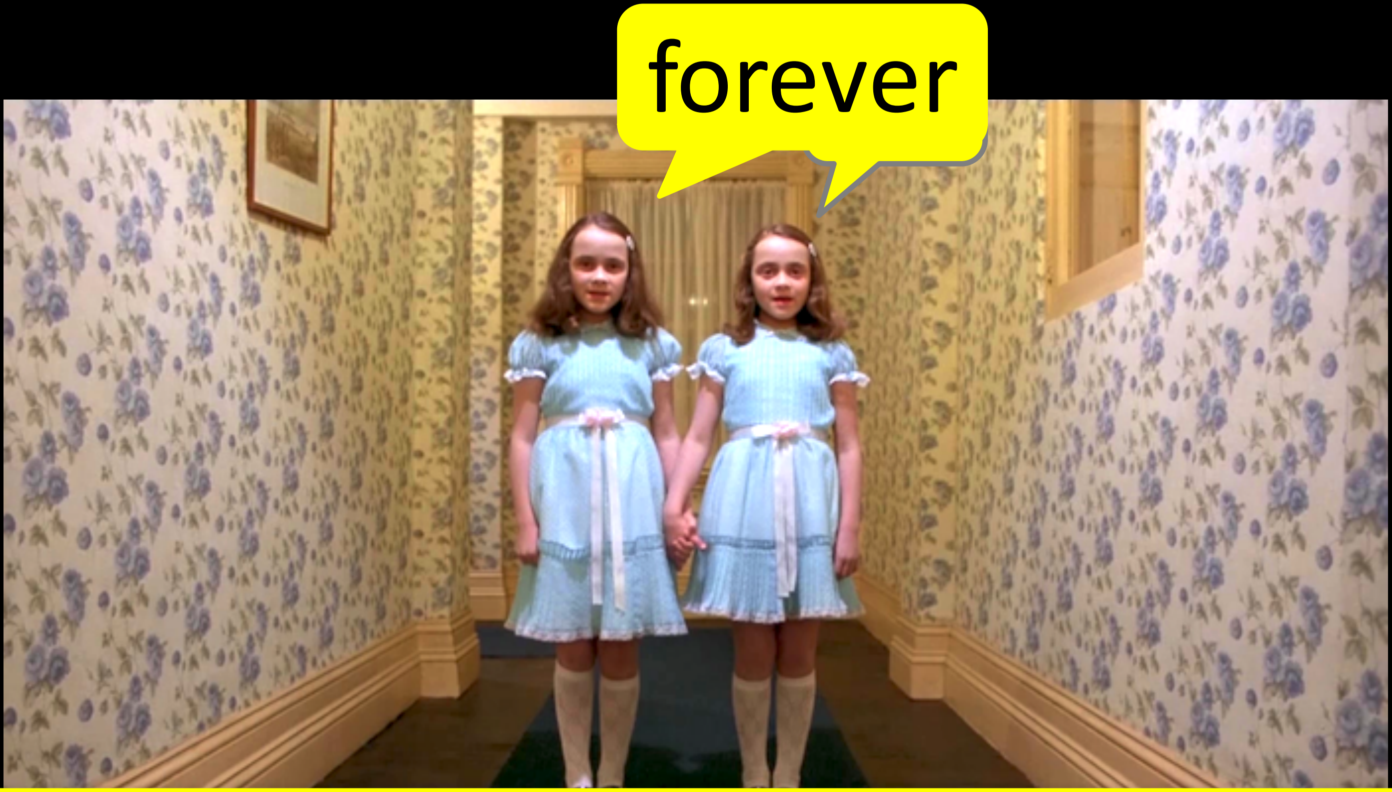
state state state state

Validators replay ***all*** block contracts in order …



… for ***every*** block

Contracts are re-executed…

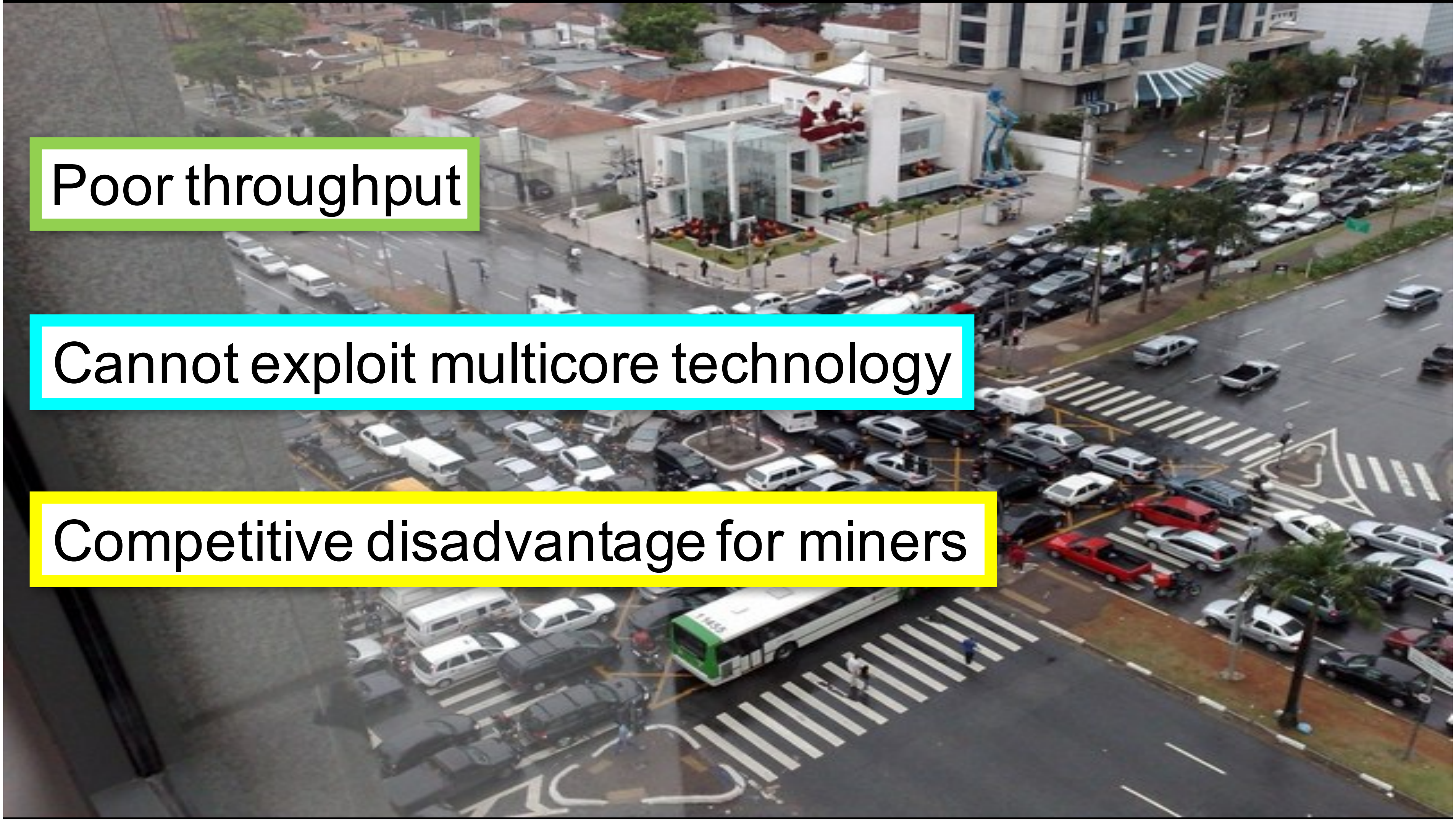


Every validator eventually executes *every* contract

Contracts are re-executed…

sequentially

Why is sequential execution so wrong?



Poor throughput

Cannot exploit multicore technology

Competitive disadvantage for miners

Adding Concurrency

Naïve Concurrency?

Nope

Inconsistent shared state

Voters could vote twice

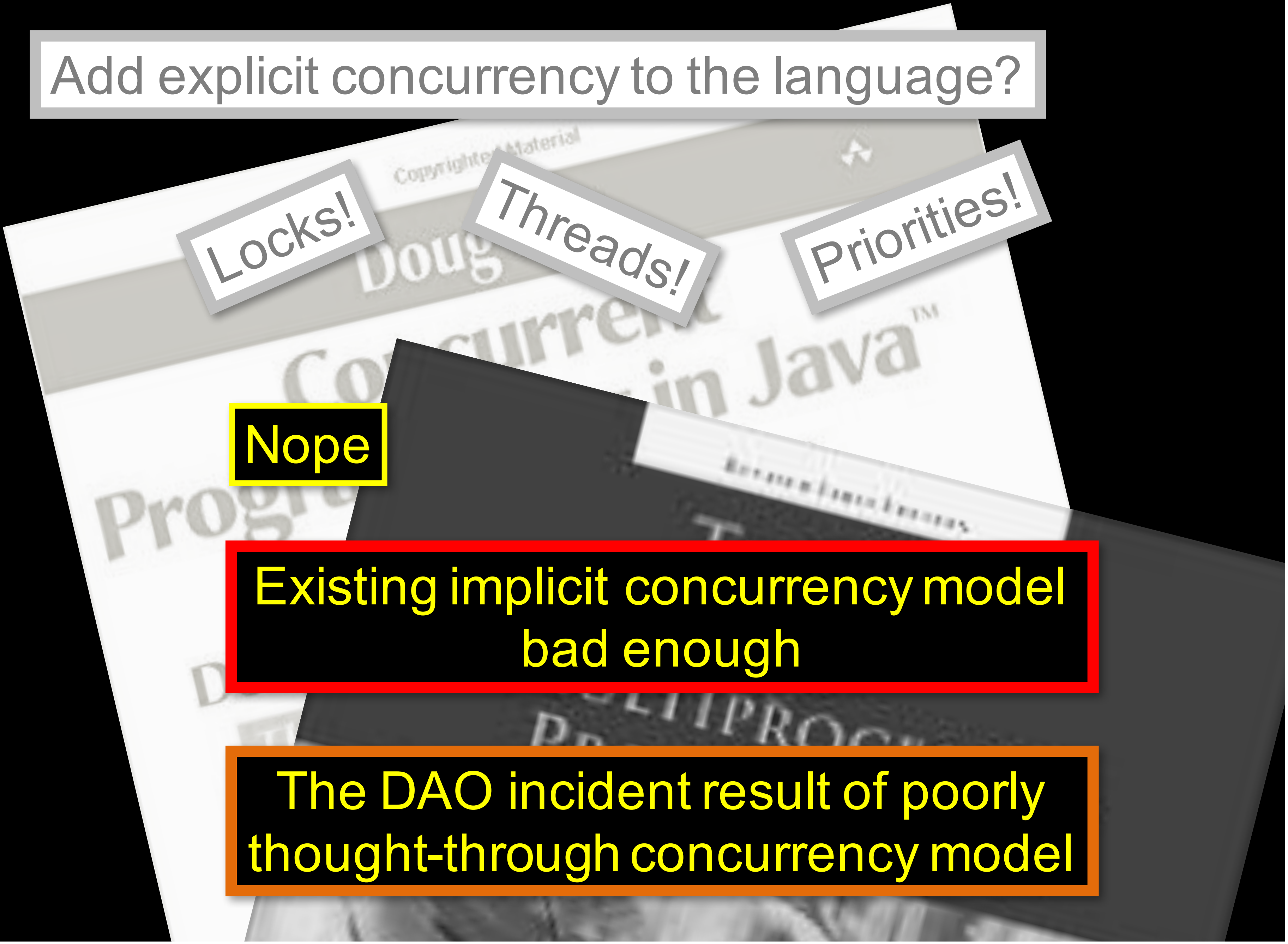
Add explicit concurrency to the language?

Locks!

Threads!

Priorities!

Add explicit concurrency to the language?



Locks!

Threads!

Priorities!

Nope

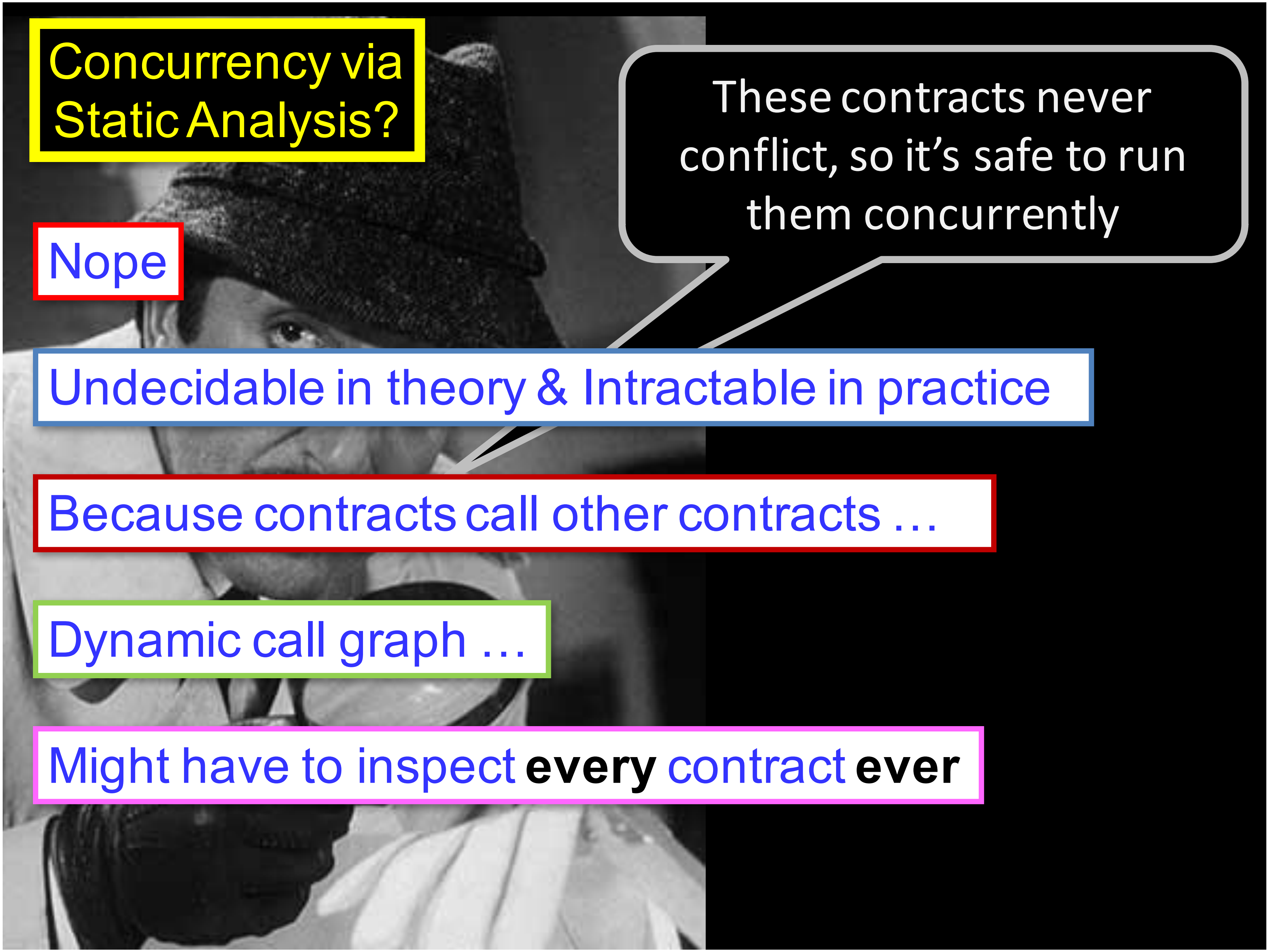
Existing implicit concurrency model bad enough

The DAO incident result of poorly thought-through concurrency model

Concurrency via Static Analysis?

These contracts never conflict, so it's safe to run them concurrently

Concurrency via Static Analysis?


These contracts never conflict, so it's safe to run them concurrently

Nope

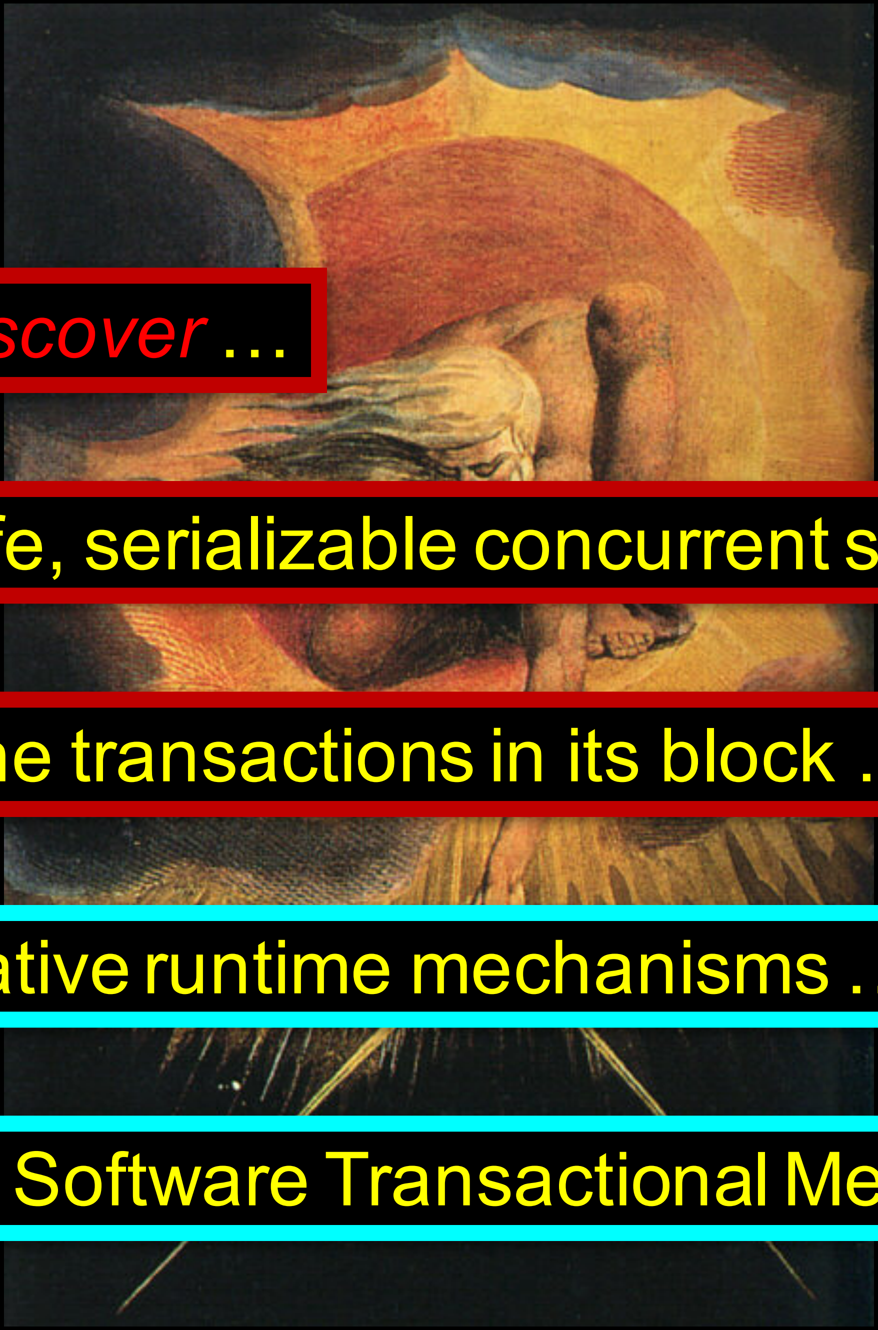
Undecidable in theory & Intractable in practice

Because contracts call other contracts …

Dynamic call graph …

Might have to inspect **every** contract **ever**

Big Idea #1



Let miners *discover* …

a safe, serializable concurrent schedule …

for the transactions in its block …

using speculative runtime mechanisms …

adapted from Software Transactional Memory.

Instrument shared objects & variables

E.g., locks on methods and accessors

Function are atomic sections

Conflict detected?

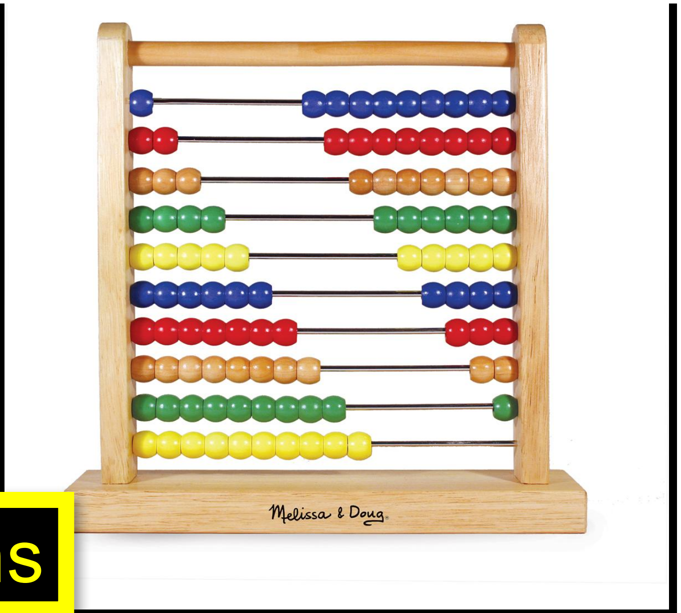
Delay or restart one thread

Keep track of “happens before”

Result is safe concurrent schedule + description

Positive

Usually, conflict is rare



Easy concurrent executions

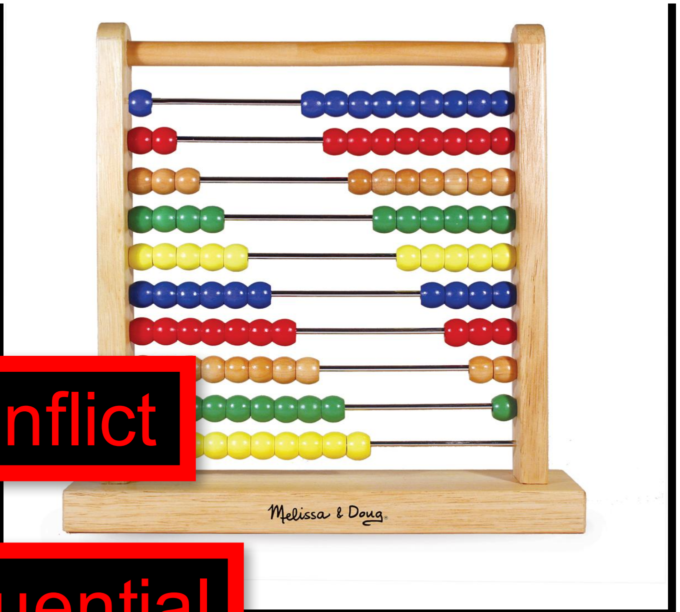
Less delay is competitive advantage

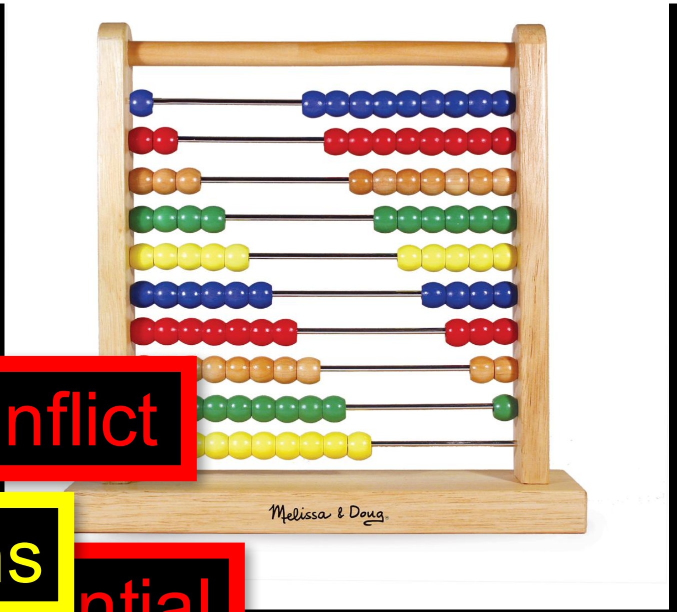
Better HW usage, less energy, etc.

Negative

Sometimes transactions do conflict

- Executions must be sequential
- Synchronization overhead means delay
- But here are many tricks …





Positive

Negative

Usually, conflict is rare

Sometimes transactions do conflict

Easy concurrent executions

Executions must be sequential

Less delay is competitive advantage

Synchronization overhead means delay

Better HW usage, less energy, etc.

But here are many tricks …

Take your choice

What about validators

✓ ✓ ✓

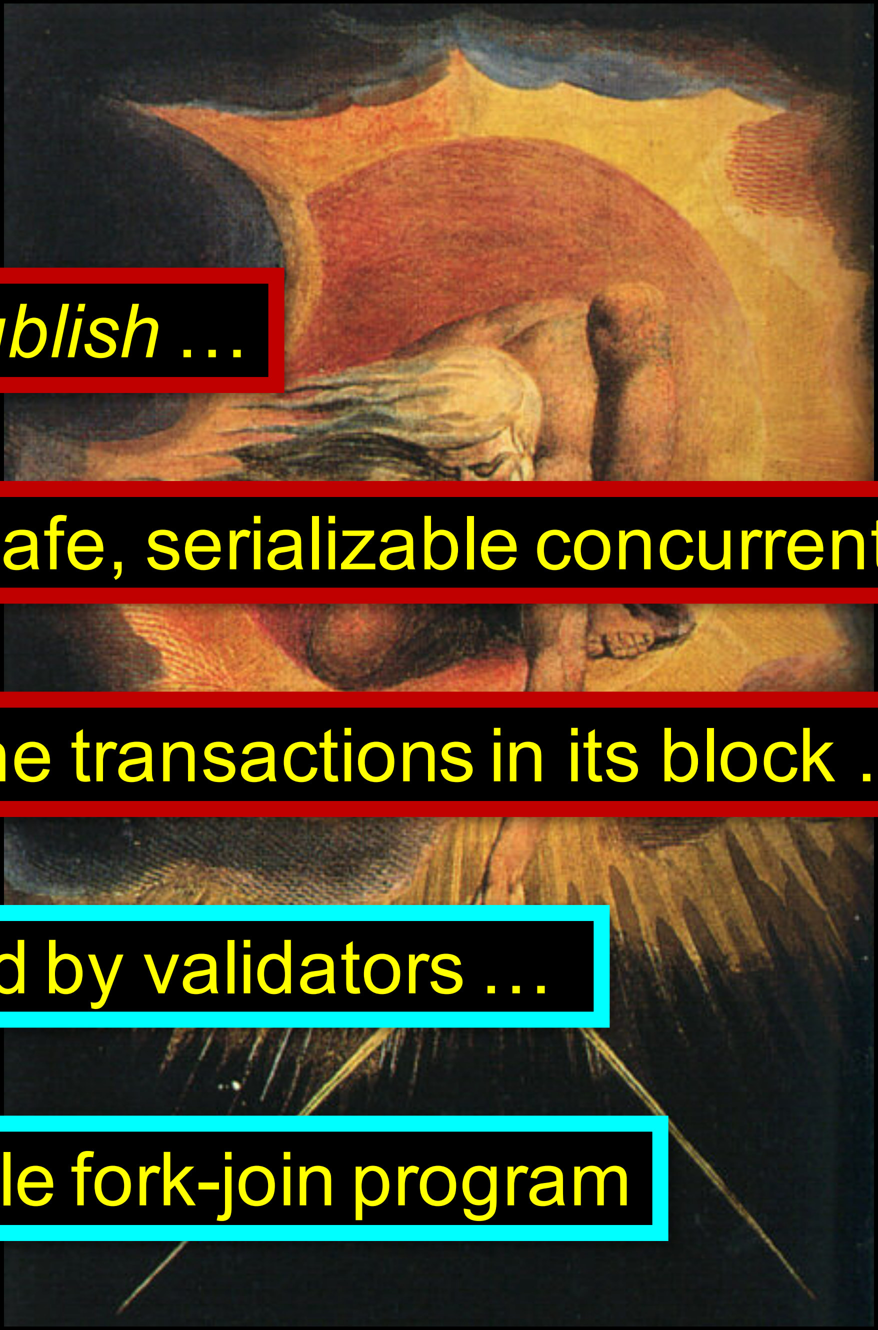
Cannot mimic miners by discovering schedules

Parallel executions non-deterministic

Might find a different safe concurrent schedule

Or resort to sequential execution

Big Idea #2



Let miners *publish* …

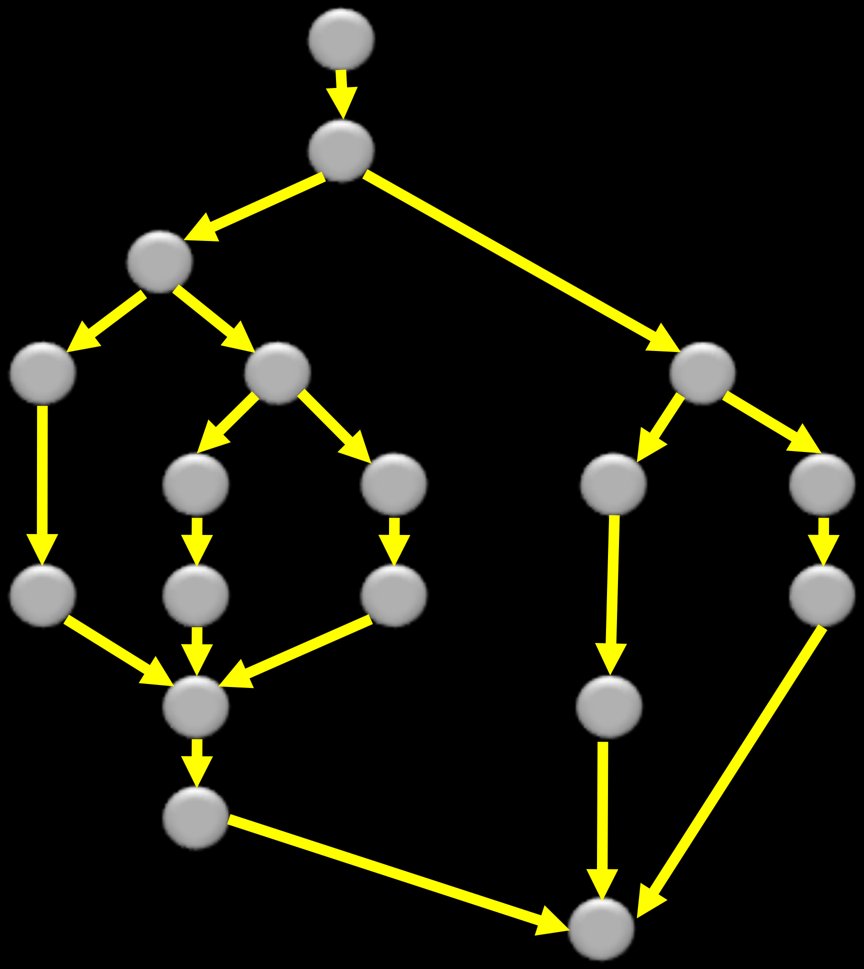
the safe, serializable concurrent schedule …

for the transactions in its block …

to be replayed by validators …

as a checkable fork-join program

Generate a Fork-Join Program



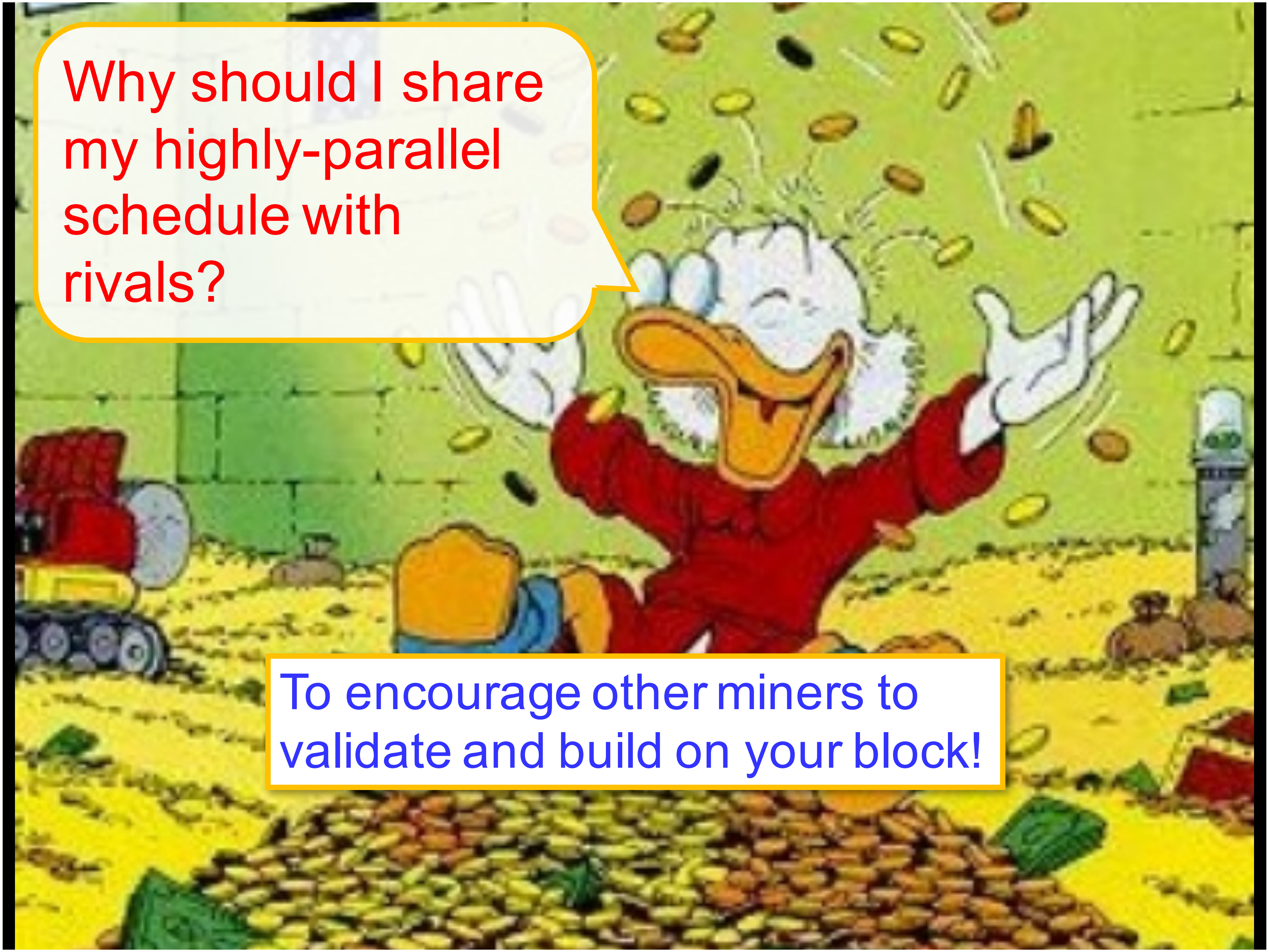
Similar to CILK model

Efficient work-stealing scheduler

Can check validity

No locks, rollbacks

deterministic



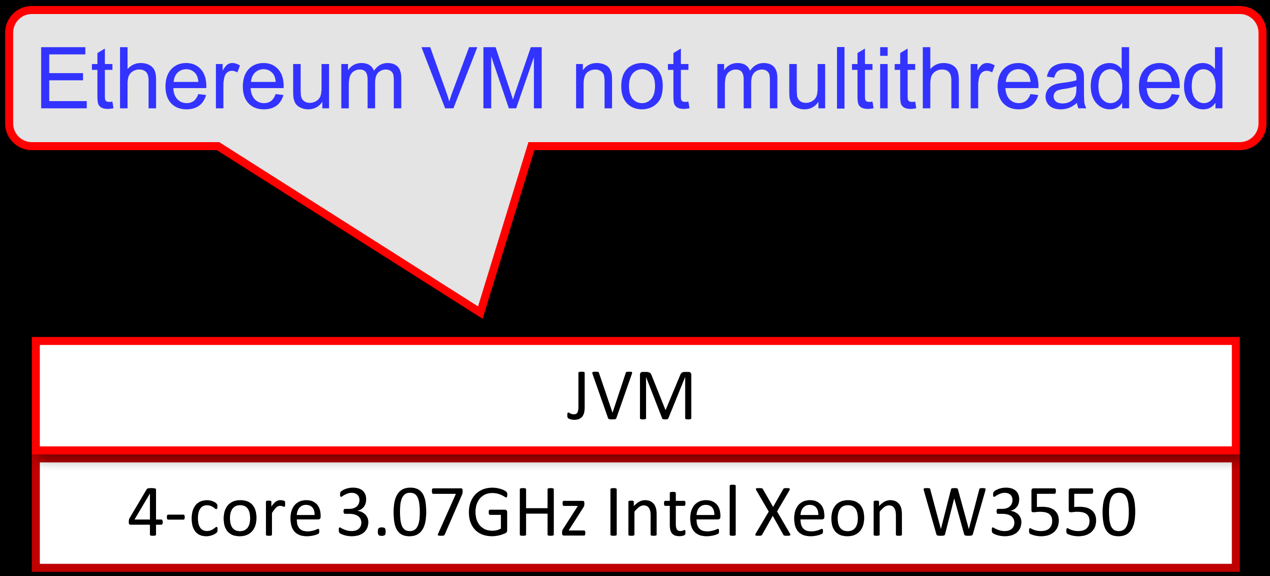
Why should I share my highly-parallel schedule with rivals?

To encourage other miners to validate and build on your block!

Prototype and Evaluation

Available hardware

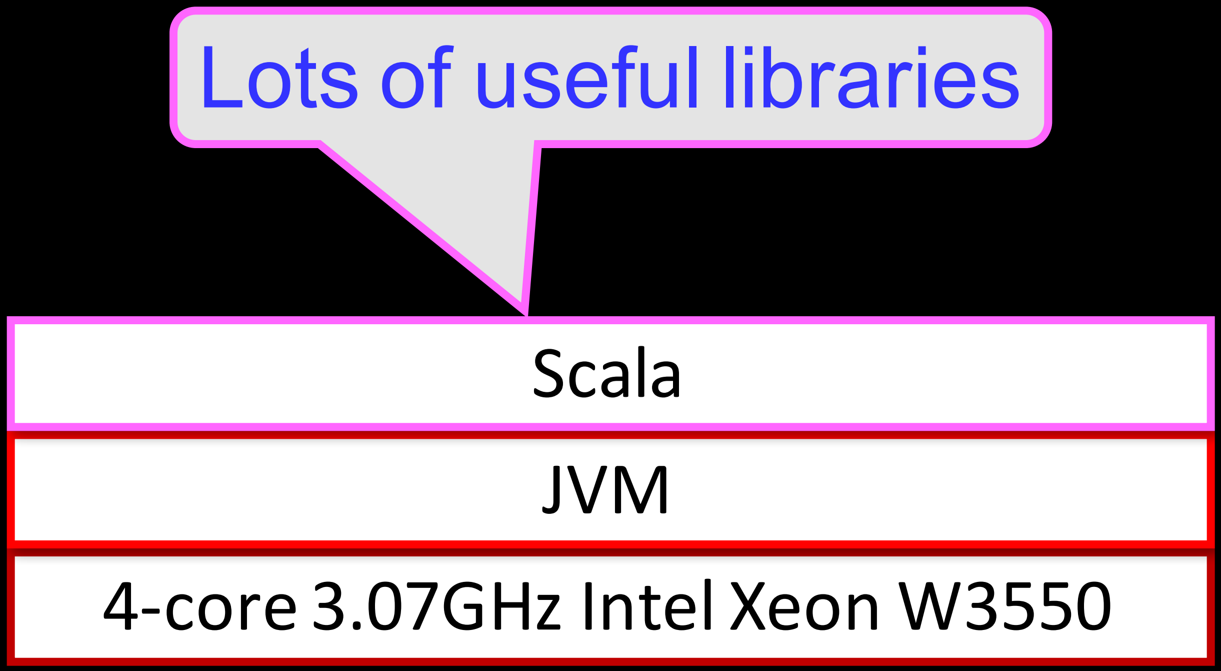
4-core 3.07GHz Intel Xeon W3550



Ethereum VM not multithreaded

JVM

4-core 3.07GHz Intel Xeon W3550

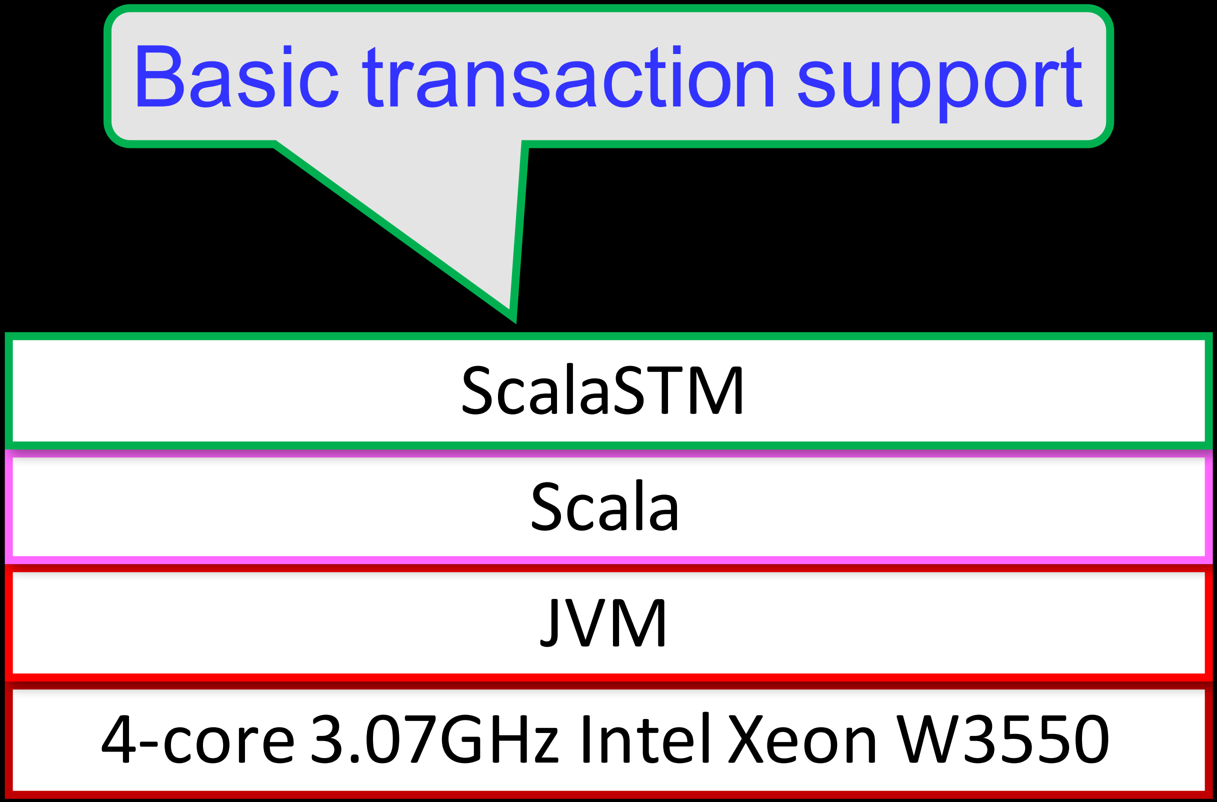


Lots of useful libraries

Scala

JVM

4-core 3.07GHz Intel Xeon W3550



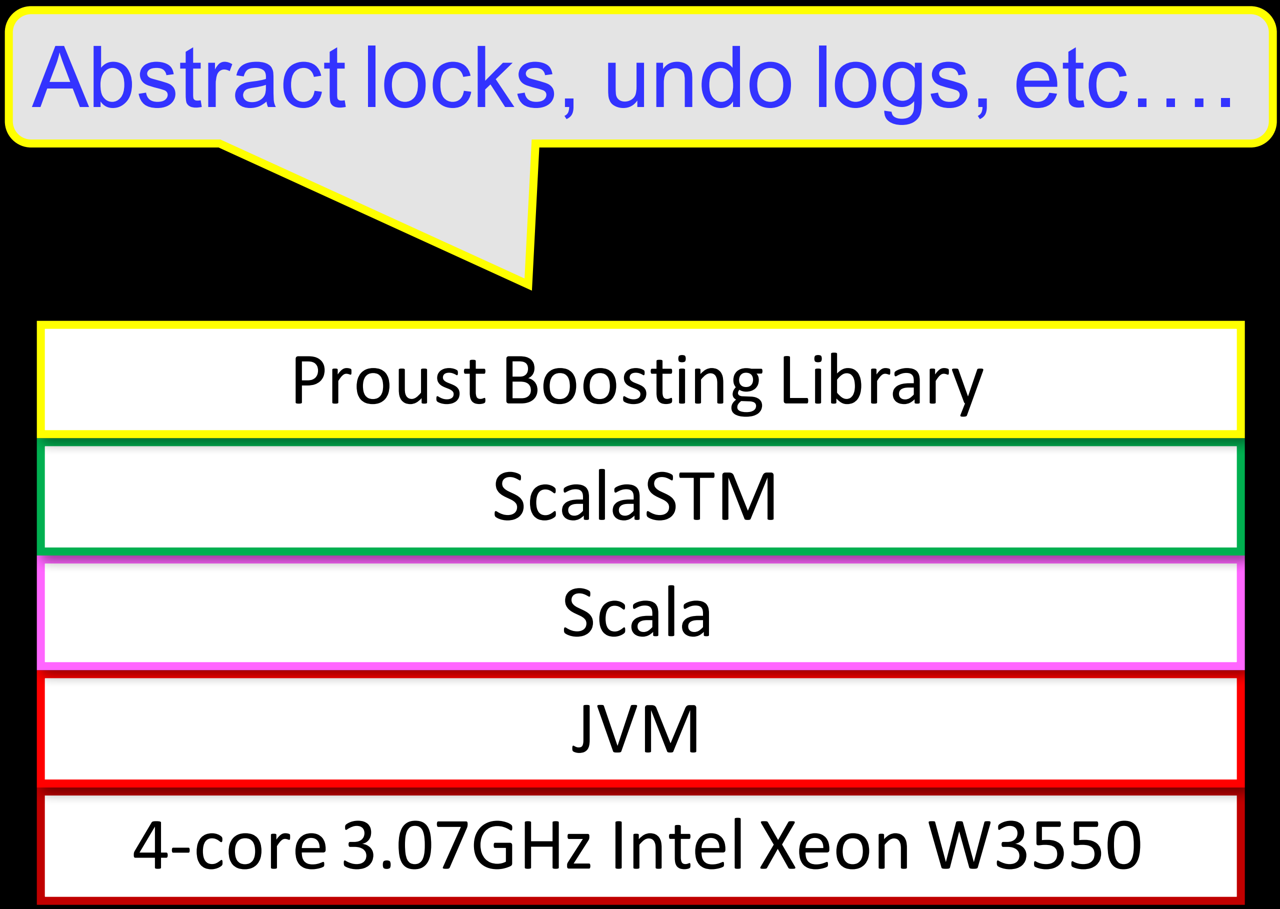
Basic transaction support

ScalaSTM

Scala

JVM

4-core 3.07GHz Intel Xeon W3550



Abstract locks, undo logs, etc.…

Proust Boosting Library

ScalaSTM

Scala

JVM

4-core 3.07GHz Intel Xeon W3550

Benchmarks

JVM with JIT turned off

3 cores (1 more reserved for GC)

Single-benchmark blocks

Mixed-benchmark blocks

Tunable Conflict rate

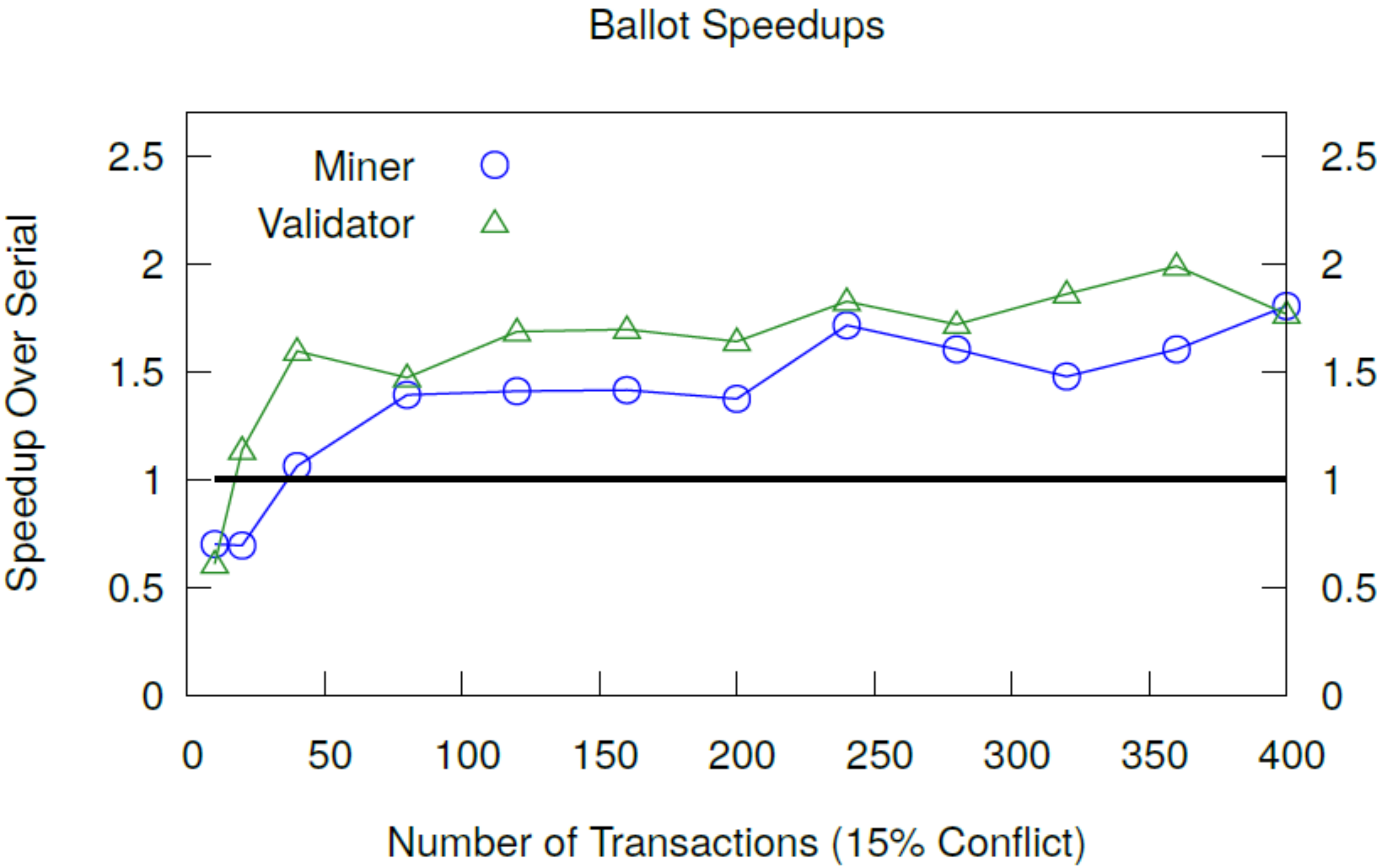
Benchmark #1: Ballot

From Solidity documentation

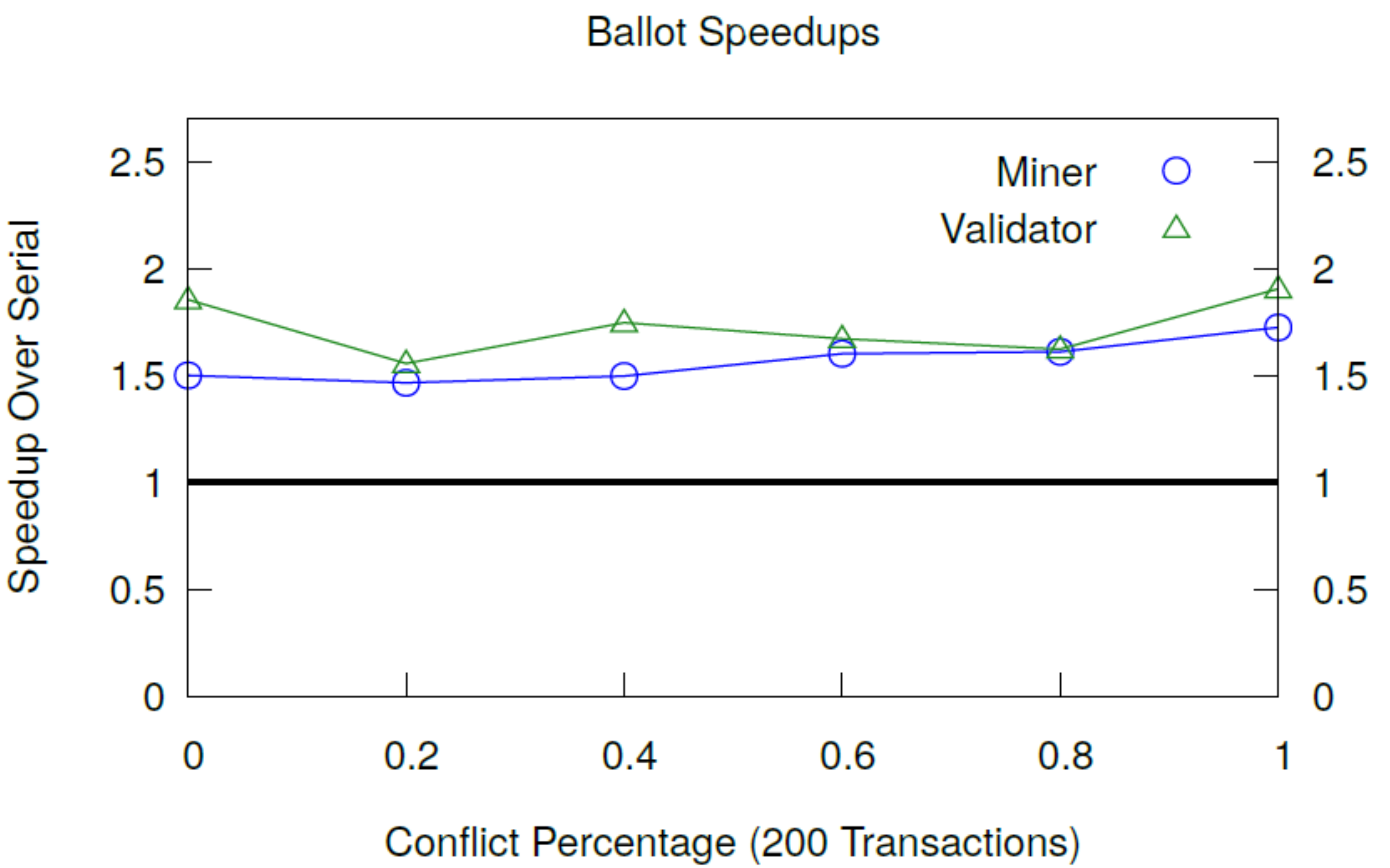
Benchmark: all voters registered, vote only

Shared state: voter mapping

Tunable Conflict = double voting



Varying Transactions per Block



Varying Levels of Conflict

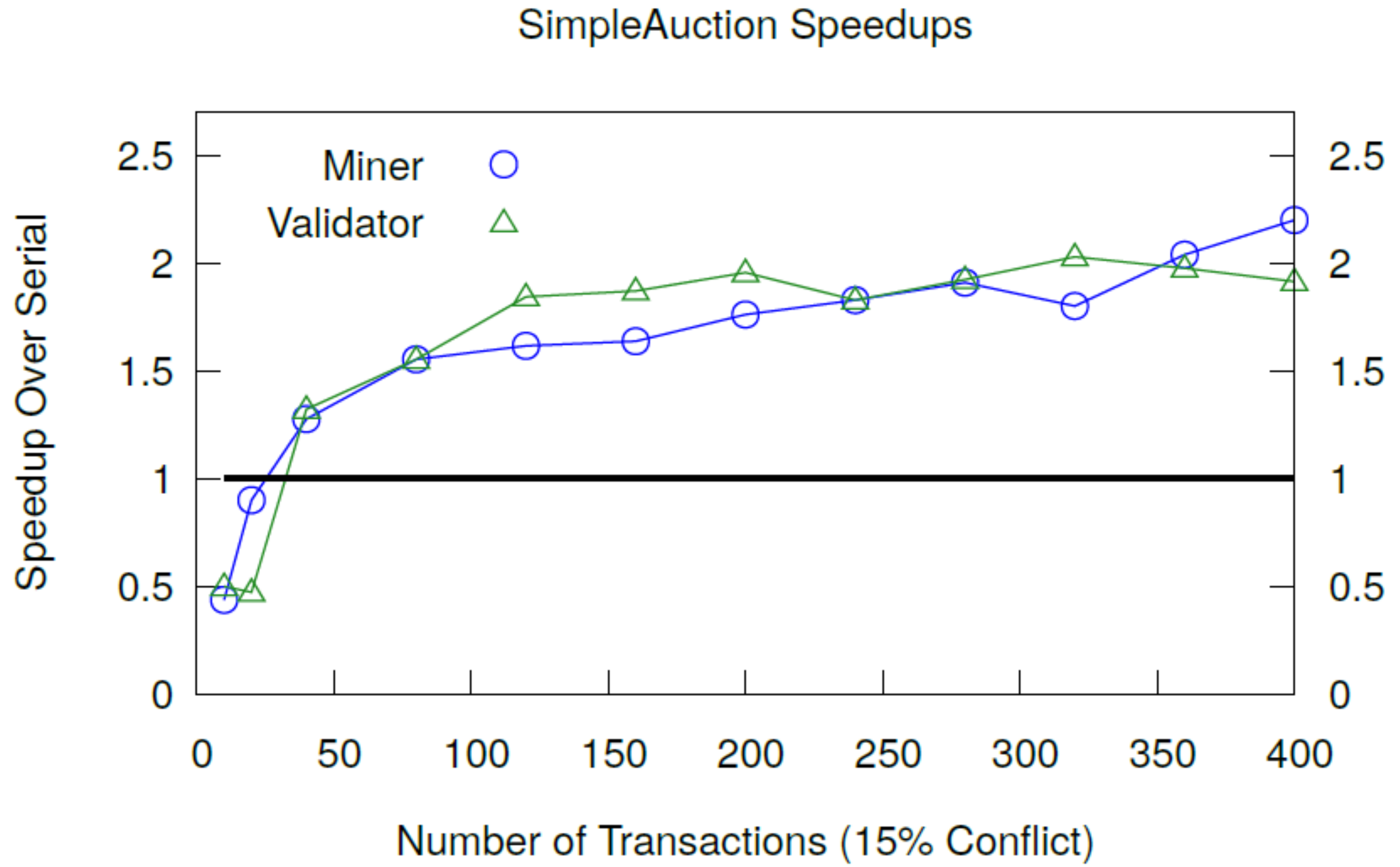
Benchmark #2: SimpleAuction

From Solidity documentation

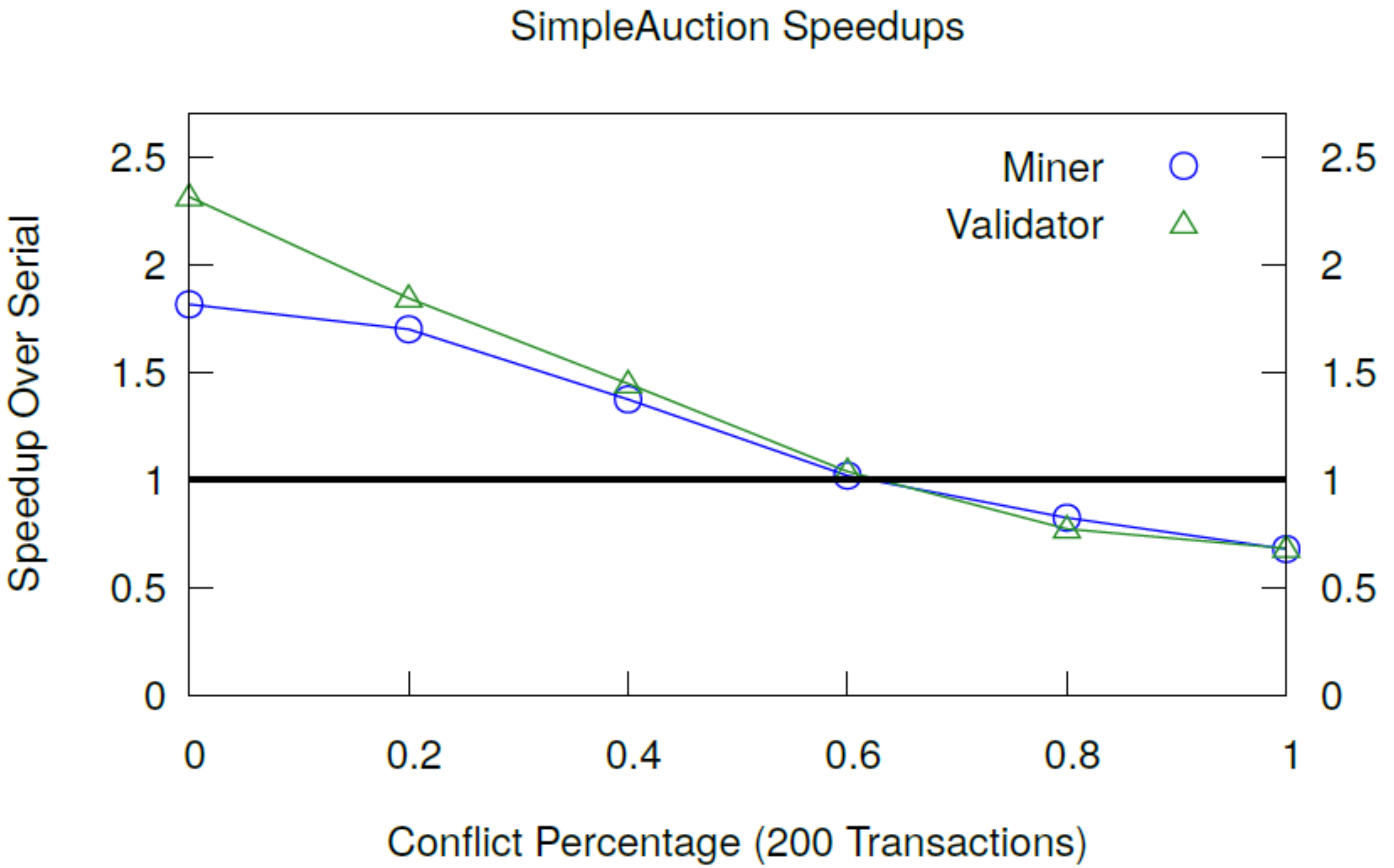
Benchmark: bidders bid, request refunds

Shared state: maxBid

Tunable Conflict = bidPlusOne() vs refund



Varying Transactions per Block



Varying Levels of Conflict

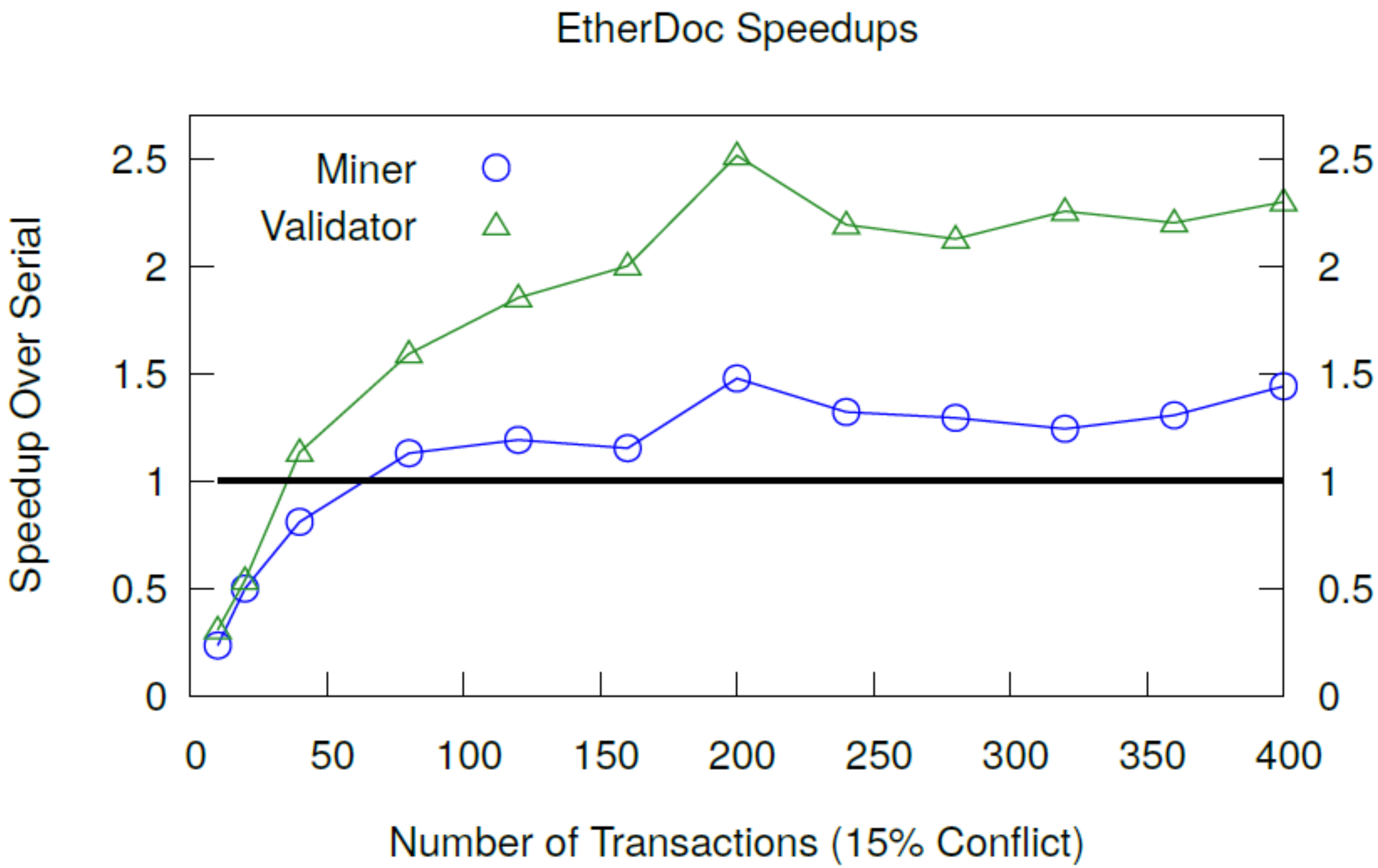
Benchmark #3: EtherDoc

From website

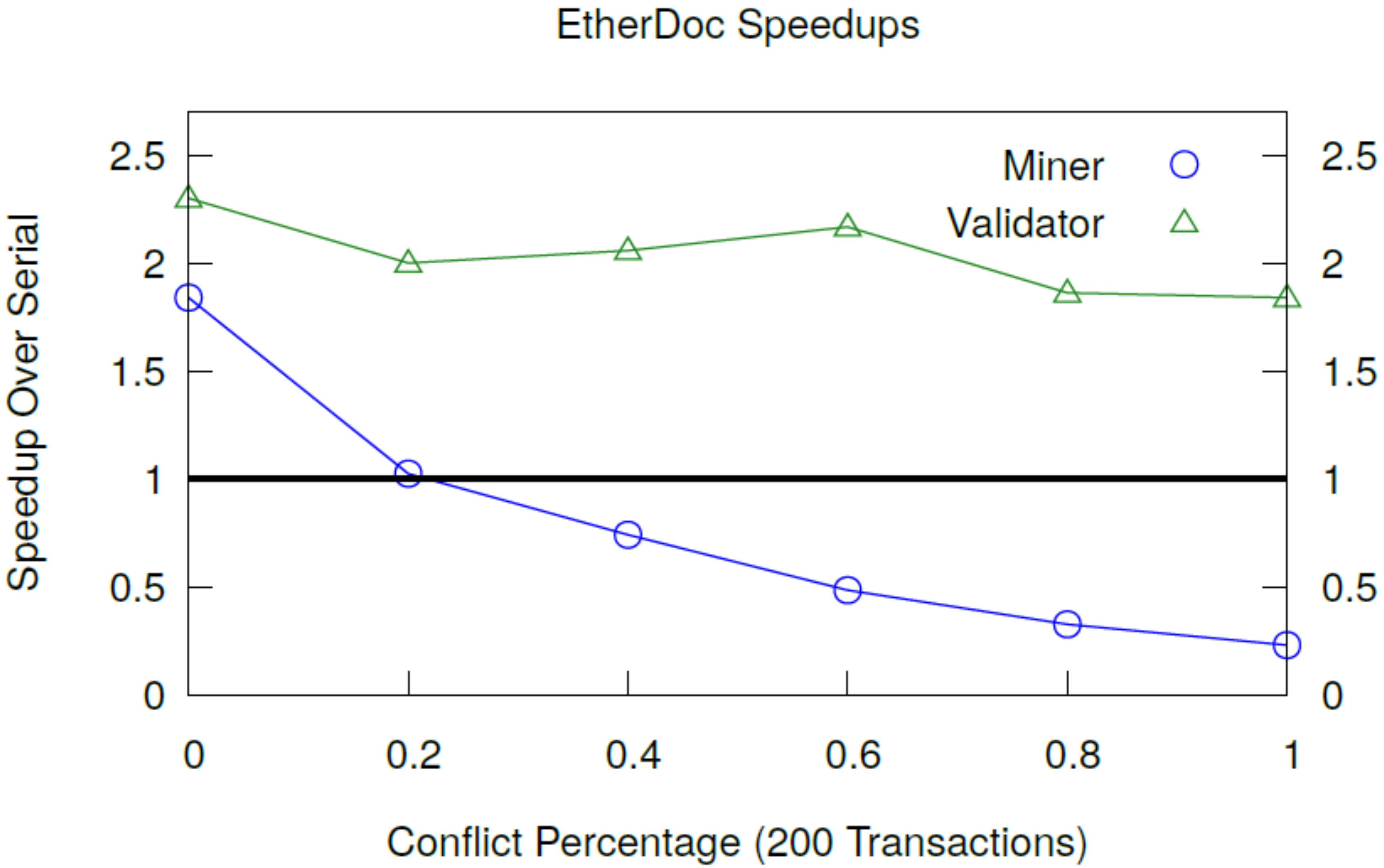
Tracks Document Metadata (including owner)

Shared state: owner’s list of docs

Tunable Conflict = transfer vs query



Varying Transactions per Block

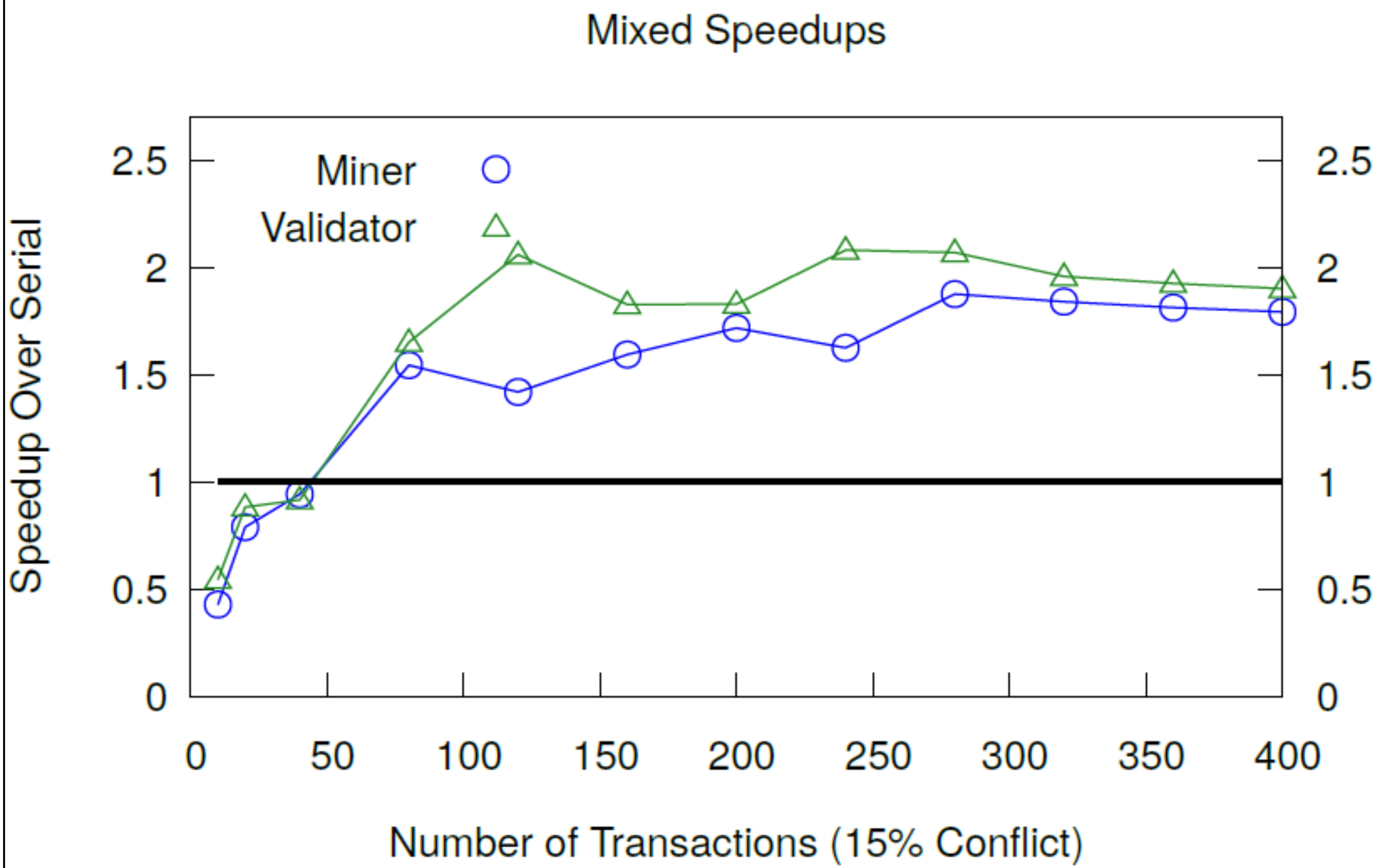


Varying Levels of Conflict

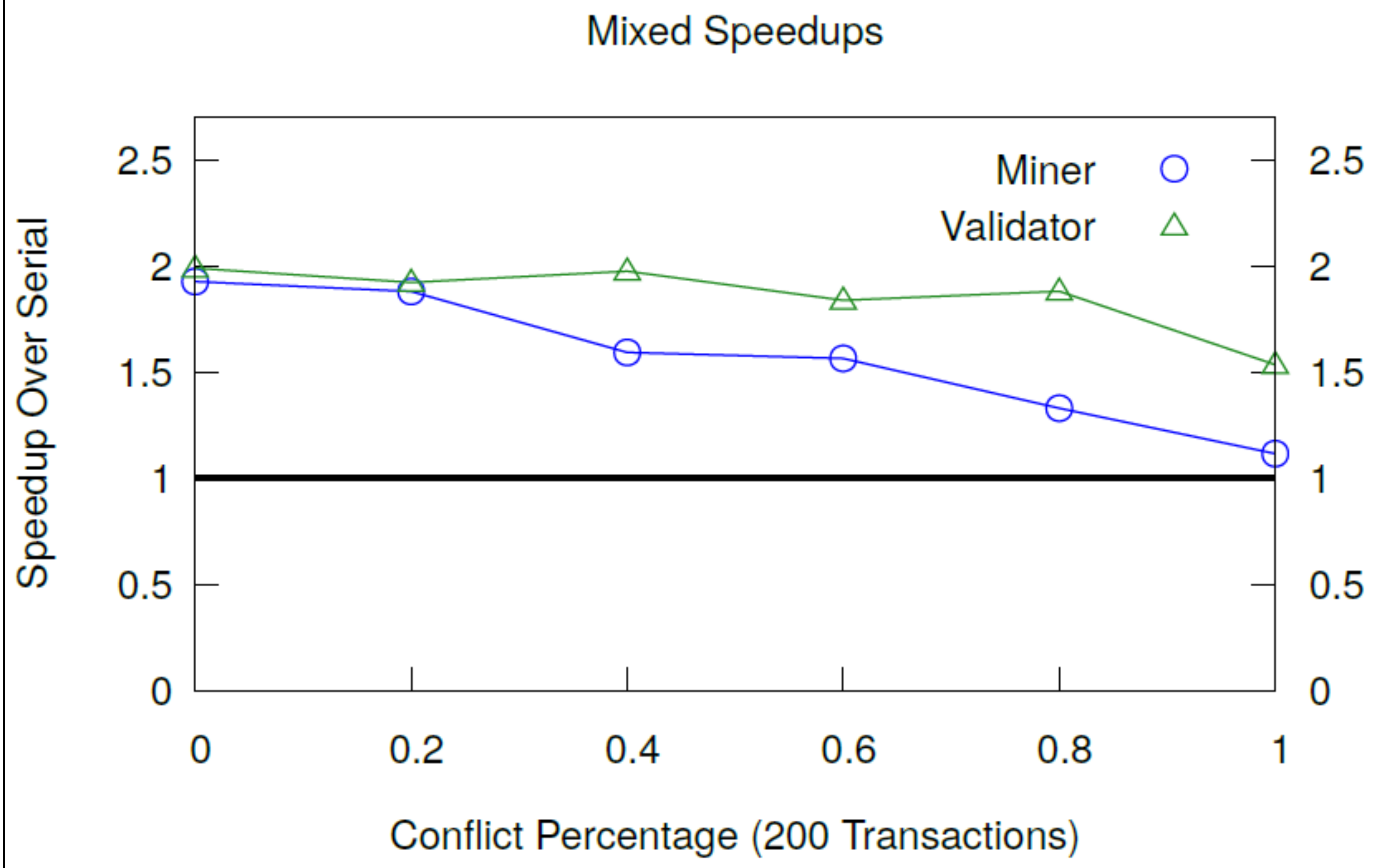
Benchmark #4: Mixed

All of the above

Equal proportions

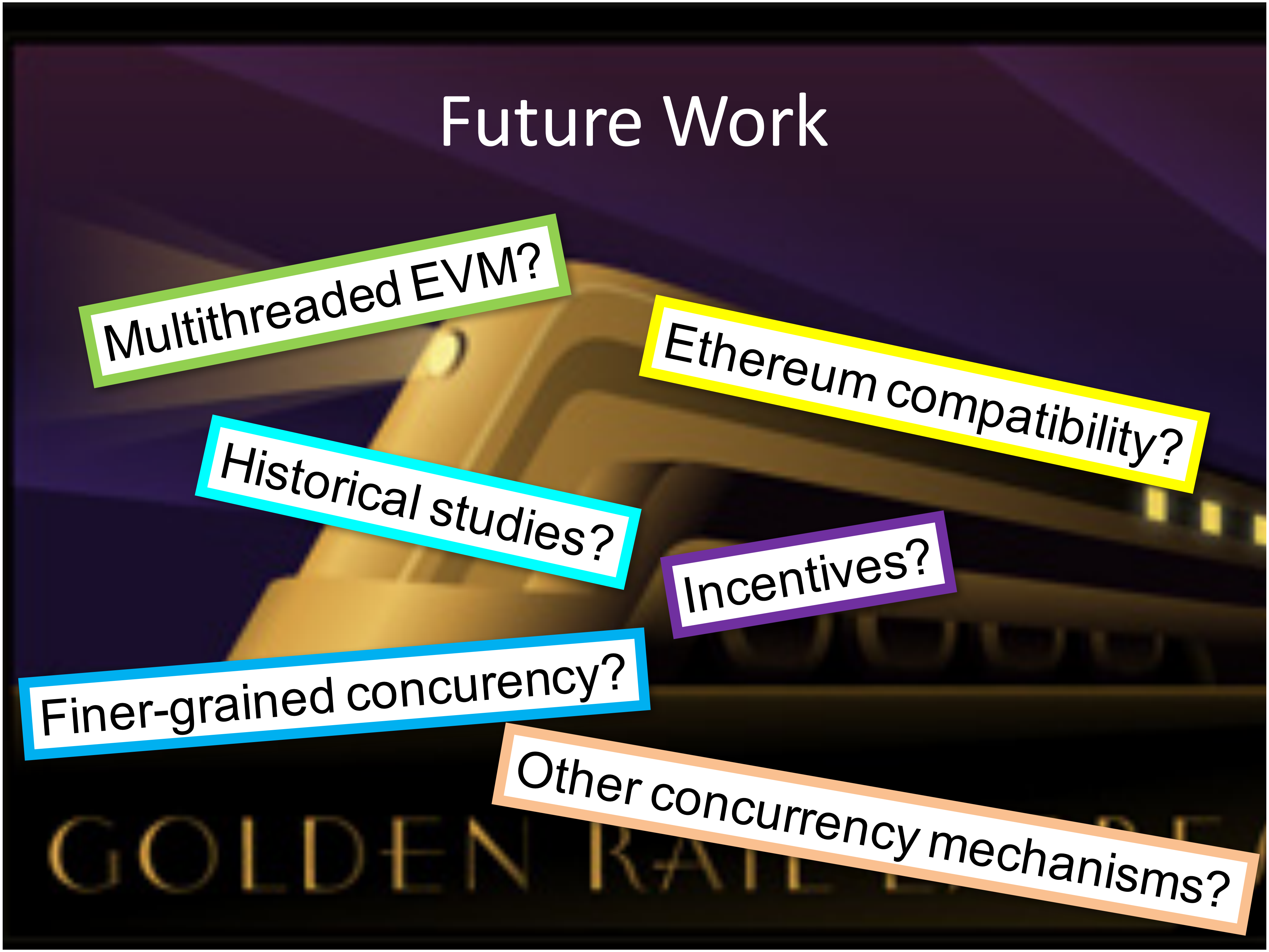


Varying Transactions per Block



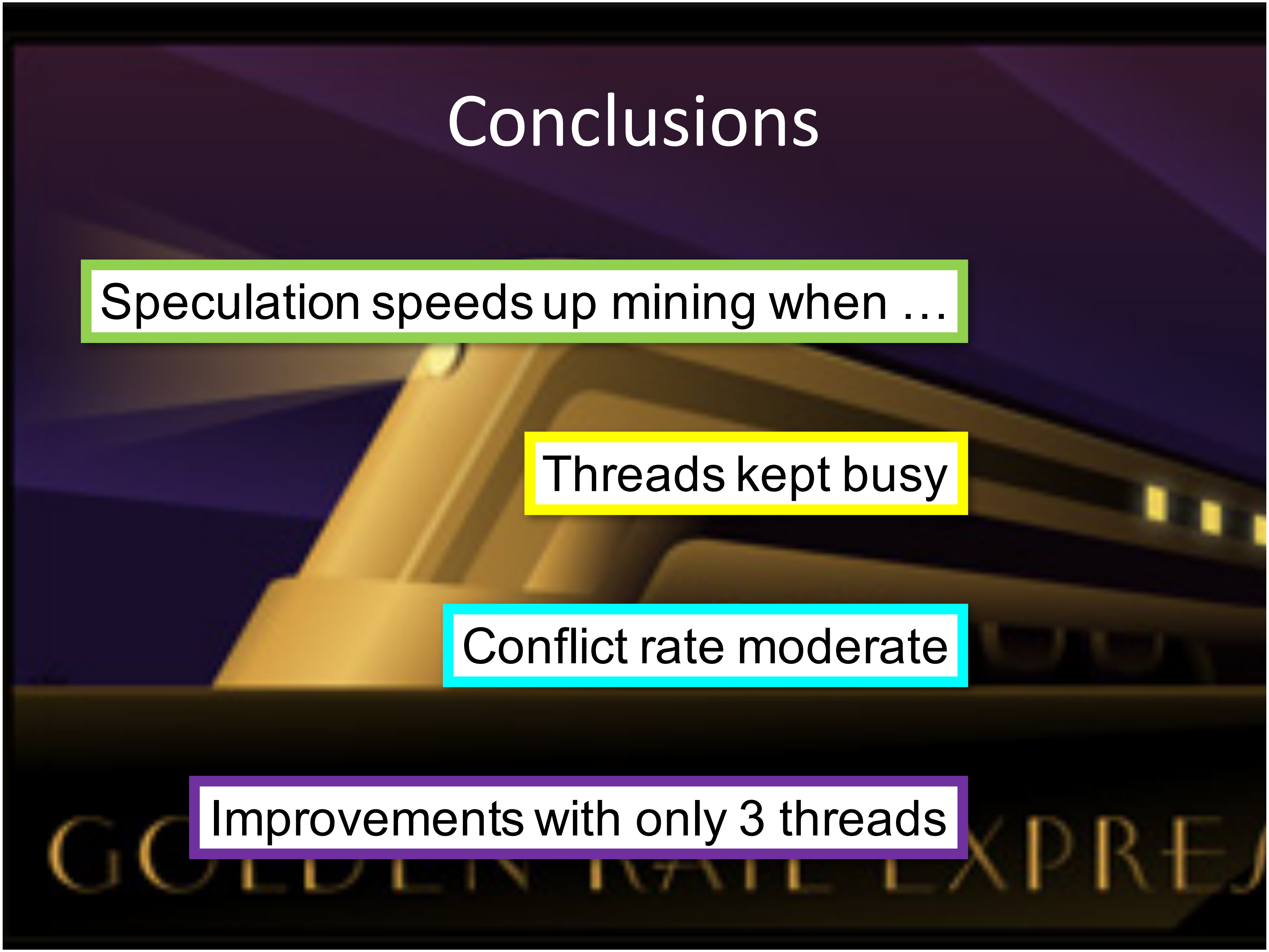
Varying Levels of Conflict

Future Work



- Multithreaded EVM?
- Ethereum compatibility?
- Historical studies?
- Incentives?
- Finer-grained concurrency?
- Other concurrency mechanisms?

Conclusions



Speculation speeds up mining when …

Threads kept busy

Conflict rate moderate

Improvements with only 3 threads

Thank You!
Questions?