

Ergo platform

Dmitry Meshkov

Prehistory

Motivation

Theory

- Provably secure
- New features
- Impractical

Practice

- 1000 currencies
- Ad-hoc solutions
- Security issues

Motivation

Theory

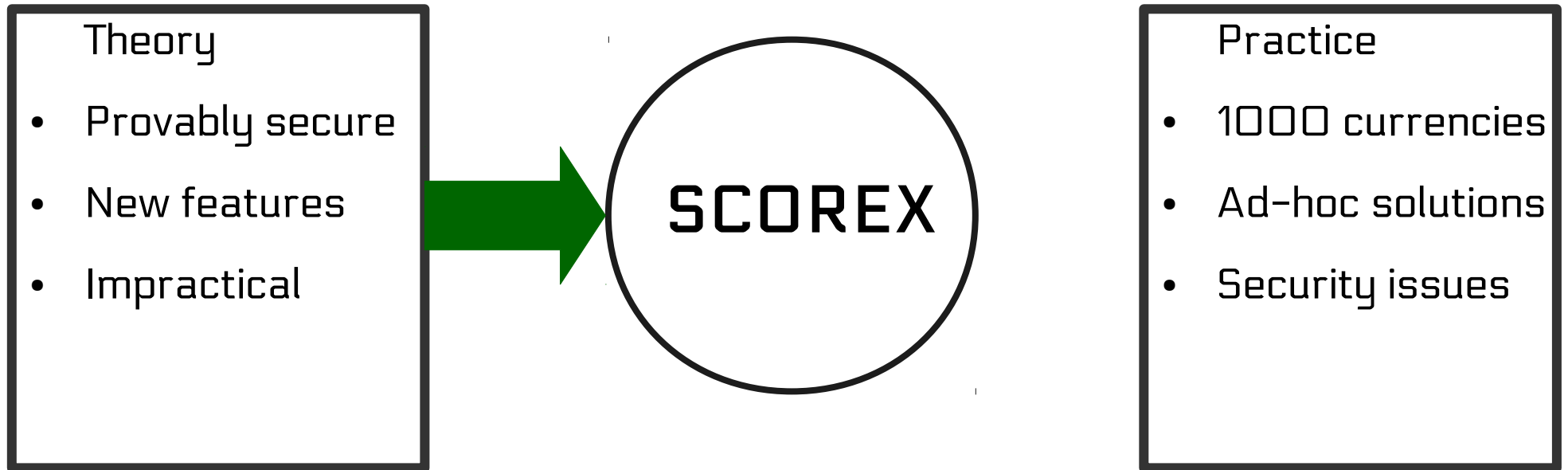
- Provably secure
- New features
- Impractical



Practice

- 1000 currencies
- Ad-hoc solutions
- Security issues

Motivation: Scorex



Motivation: Ergo

Theory

- Provably secure
- New features
- Impractical

ERGO

Practice

- 1000 currencies
- Ad-hoc solutions
- Security issues

General ideas

Blockchain generation

- 1.0 – Bitcoin
- 2.0 – ???

Blockchain generation

- 1.0 – Bitcoin
- 2.0 – smart contracts
- 3.0 – ???

Blockchain generation

- 1.0 – Bitcoin
- 2.0 – smart contracts
- 3.0 – blockchain in a cloud
- 4.0 – ???

Blockchain generation

- 1.0 – Bitcoin
- 1.1 – Ergo
- 2.0 – smart contracts
- 3.0 – blockchain in a cloud
- 4.0 – ???

General ideas

Conservatism:

- 1) Security
- 2) Decentralization
- 3) Survivability
- 4) Other features if they don't reduce 1-3

Consensus

Consensus

Consensus protocol should:

- Consume maximum adversary capabilities (e.g. $< 50\%$ of computational power)
- Have security proofs
- Allow new members to join the network
- Allow to use the blockchain on a commodity hardware without third parties

Consensus

Examples:

- PBFT-based consensus is only secure when $>2/3$ of parties are honest (e.g. Casper)
- Sometimes assume predefined parties (e.g Hashgraph)
- Ad-hoc protocols have no security proofs (e.g. PoS, Iota)
- High-throughput blockchains require high-end hardware (e.g. Bitshares)
- PoS require do download the whole chain

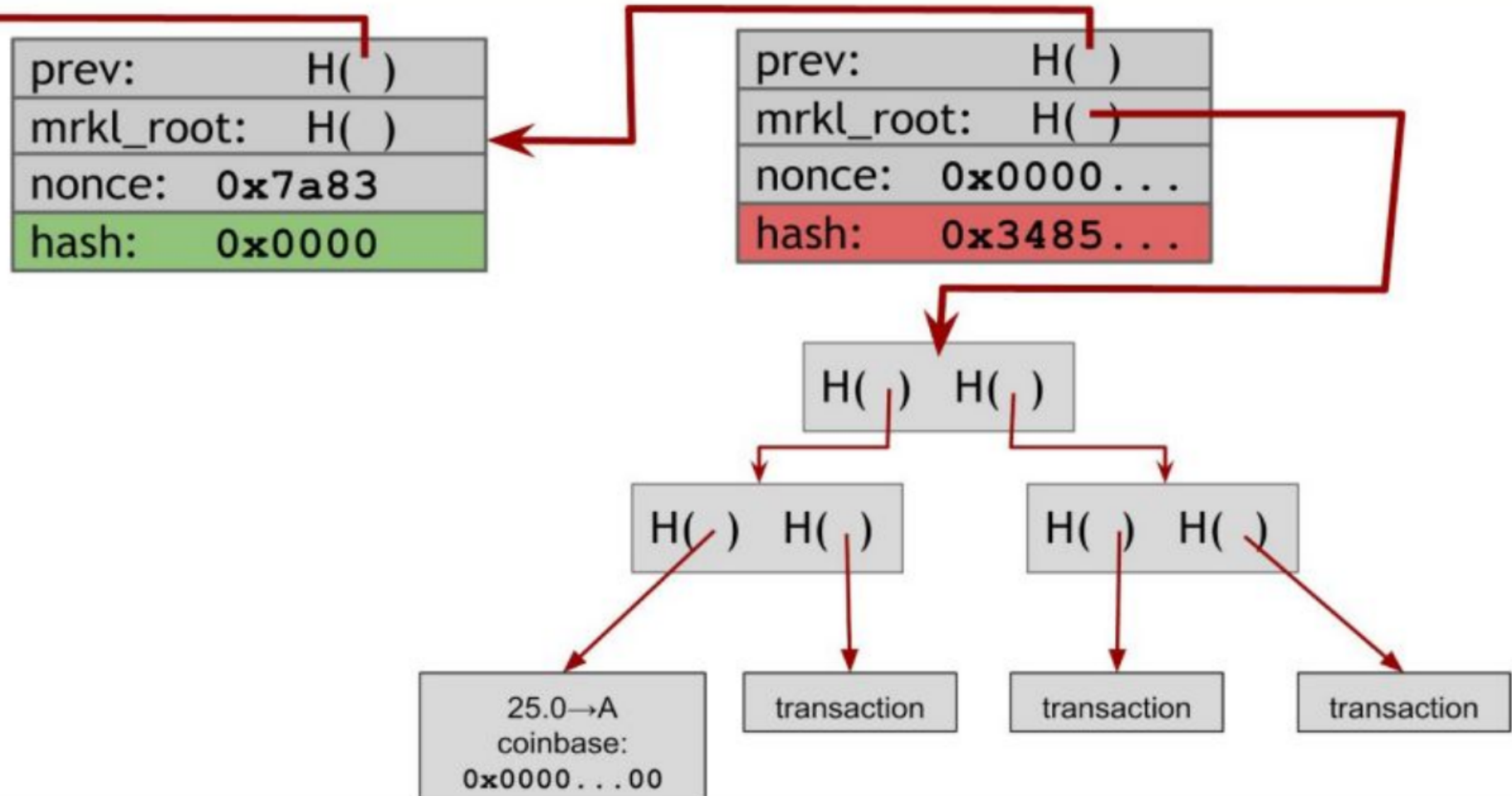
Consensus

PoW:

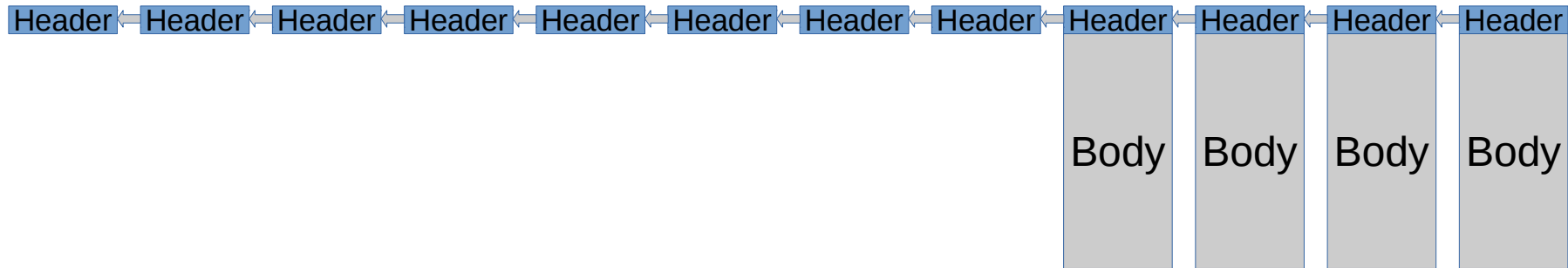
- Have security proofs for 50% honest parties
- Allows anyone to join
- Verifiable on low-end hardware
- Require only headers to check consensus
- Allows PoPoW

PoPoW

Block structure



SPV

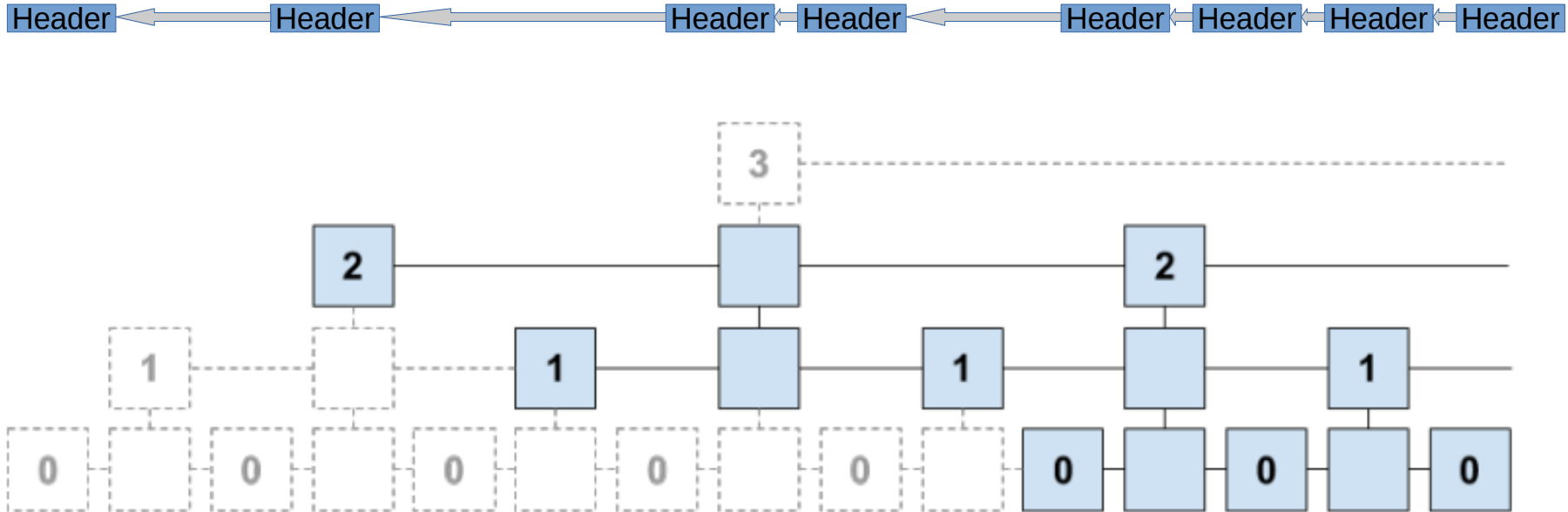


To start using network:

- (Trusted party is not an option)
- **Download headers chain** (~50 Mb in Bitcoin, ~250 Mb in Ethereum)
- Download blocks since wallet creation or current state snapshot or ...

PoPoW

- PoPoW – download extraordinary headers to estimate work done
- Fixed size (~100 Kb)
- Cheat is as difficult as full chain creation
- Requires special header structure



Authenticated state

Motivation: validating transactions

$$PK_A \xrightarrow{14 \text{ } \text{₿}} PK_D$$

Stateless validation: Check syntax, signature of PK_A , etc.

All required info is in the transaction itself

Stateful validation: Check that PK_A has 14 ₿ (+ fee)

Requires knowing how much PK_A has based on prior transactions

$PK_A \rightarrow 36$	$PK_E \rightarrow 347$	$PK_I \rightarrow 12$	$PK_M \rightarrow 12$
$PK_B \rightarrow 684$	$PK_F \rightarrow 98$	$PK_J \rightarrow 285$	$PK_N \rightarrow 76$
$PK_C \rightarrow 2$	$PK_G \rightarrow 50$	$PK_K \rightarrow 463$	$PK_O \rightarrow 3$
$PK_D \rightarrow 13$	$PK_H \rightarrow 54$	$PK_L \rightarrow 3$	$PK_P \rightarrow 88$
...			

Large (and growing!) dictionary data structure.

Currently in Bitcoin: 1.5GB (serialized).

Even worse in multiasset blockchains: one per asset.

Motivation: big state problems

Miners should validate transactions efficiently. They can:

1. Keep State in RAM => Mining centralization
2. Do not keep State => SPV mining



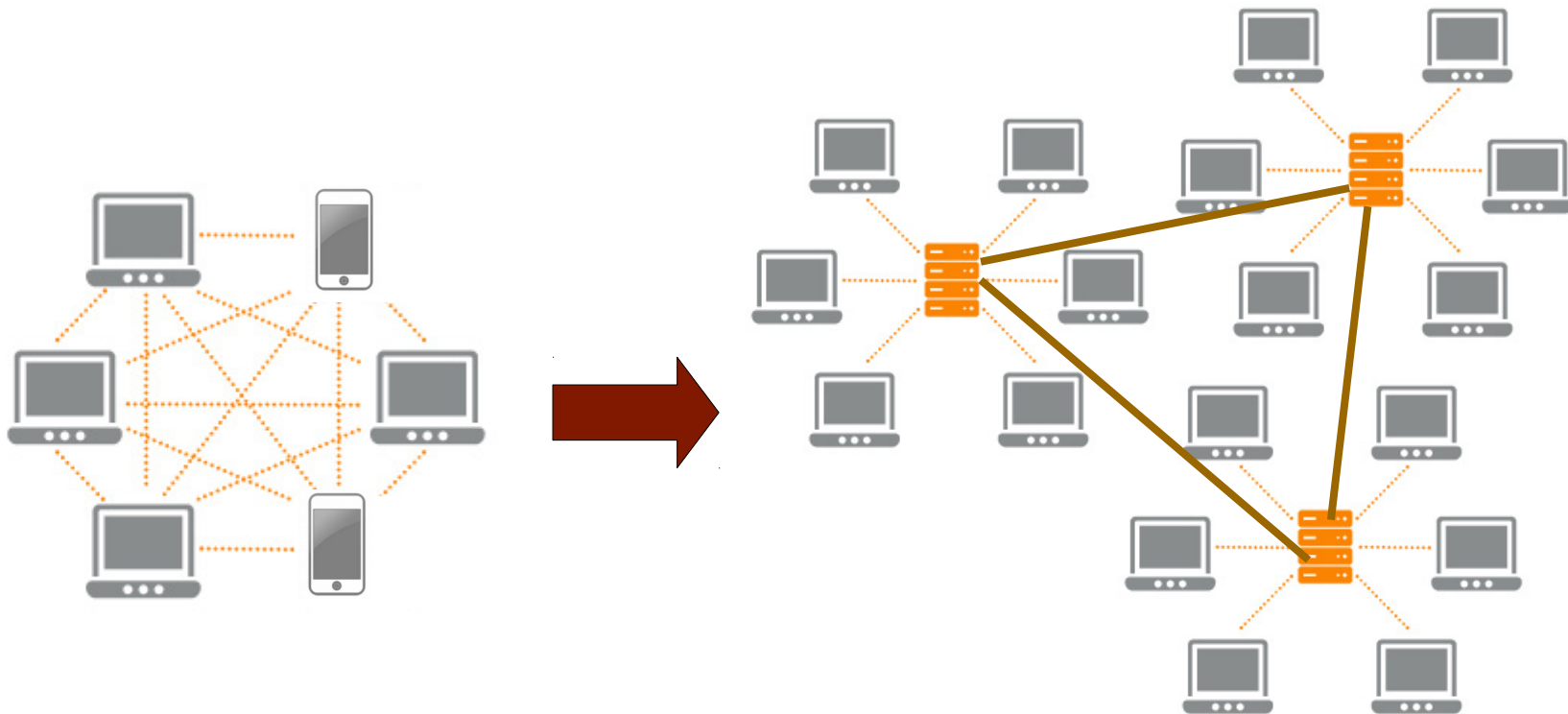
Motivation: big state problems

Problems for users:

- Can't validate blocks on low-end hardware
- Long validation on commodity hardware

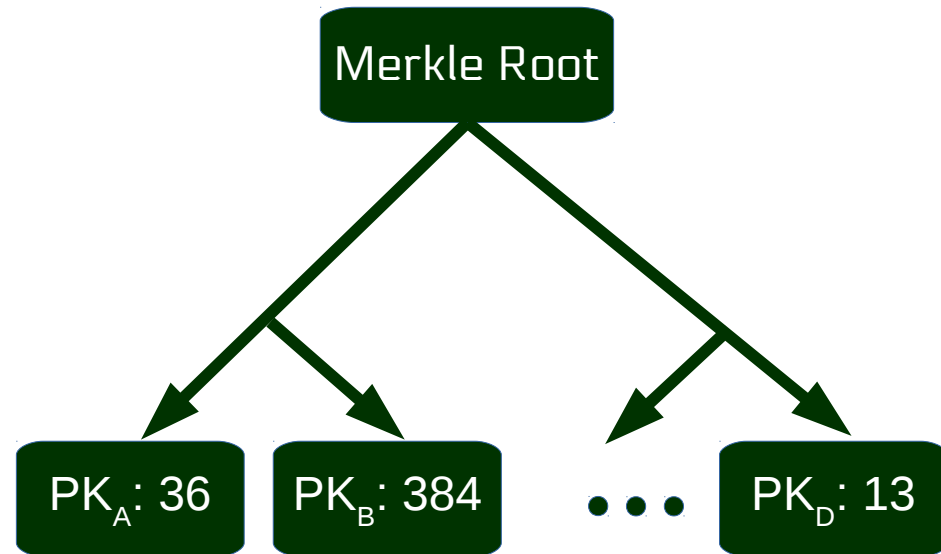
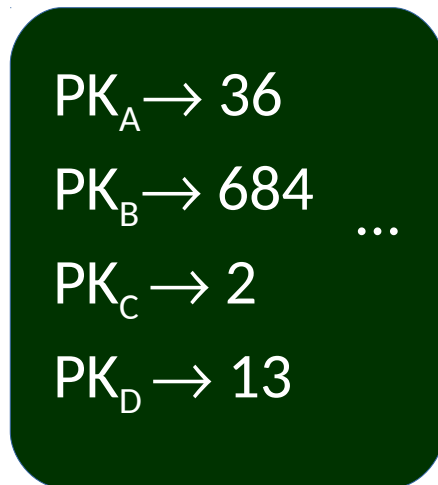
=>

- Users move to centralized services



Proofs: authenticated state

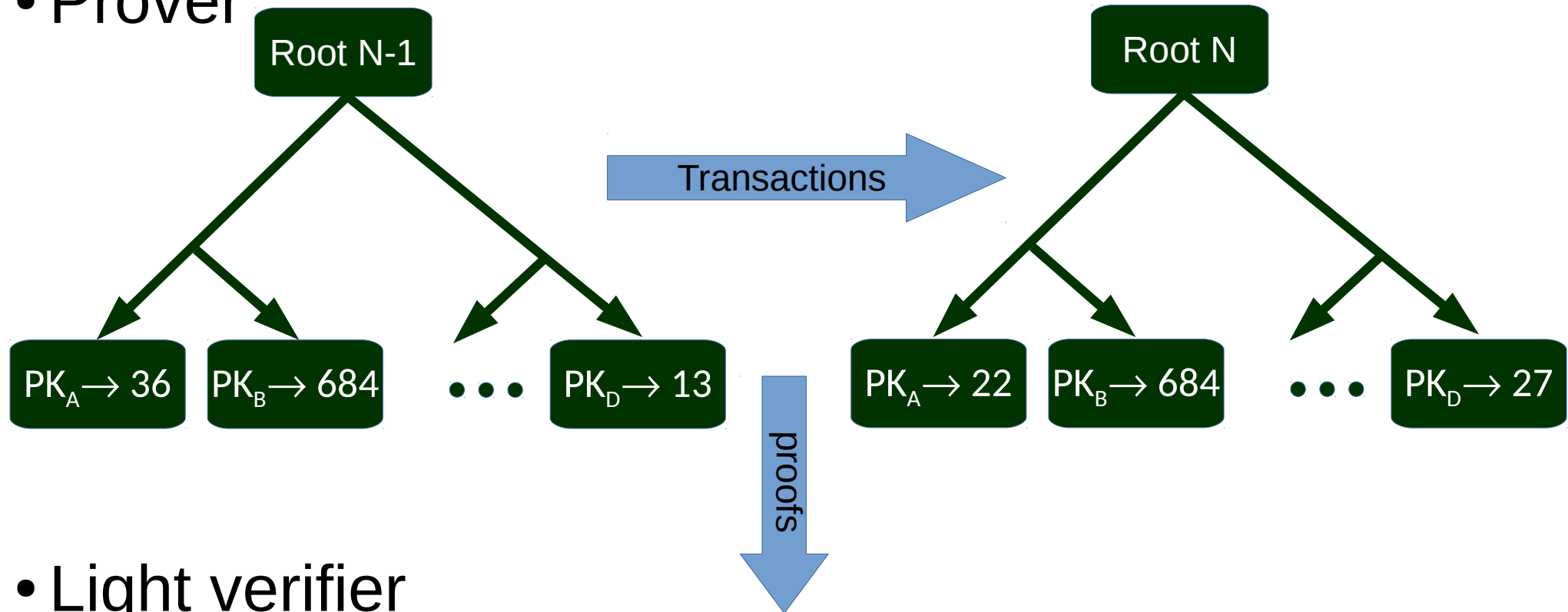
- Make state authenticated



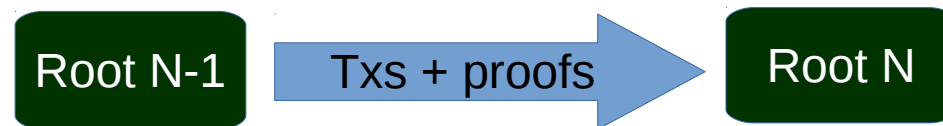
- Easy: proof of a sender's balance (standard Merkle tree proof with respect to the root).
- More complicated: ensuring the prover changed the balances correctly.
- Important: we do not wish to trust the prover!

Blockchain parties

- Prover

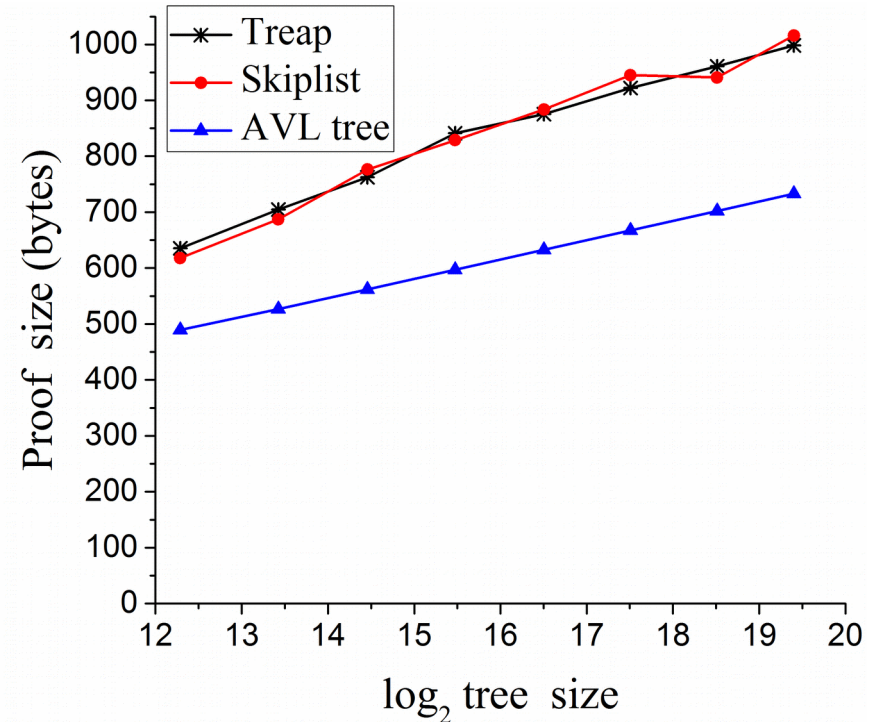
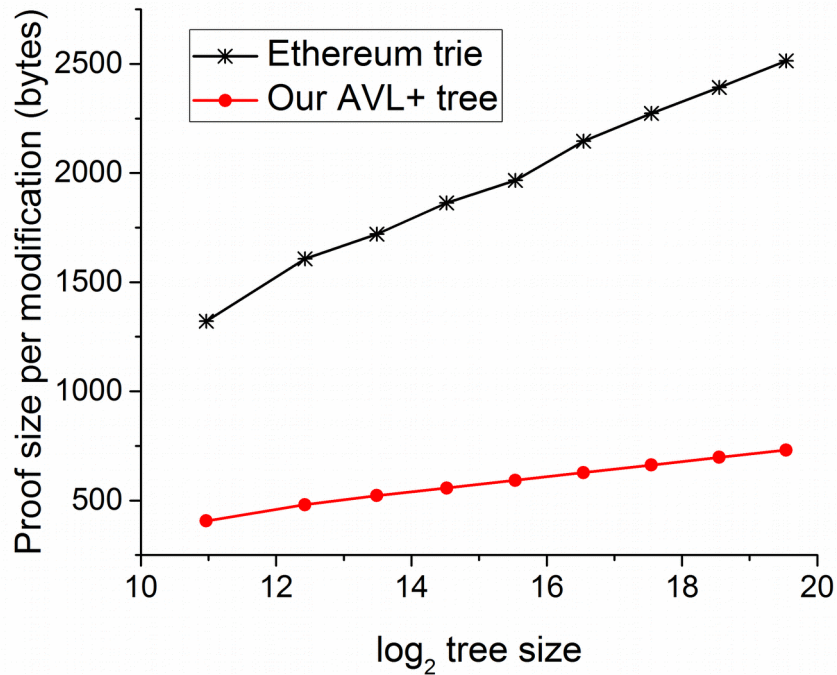


- Light verifier



Proofs verify balances and calculate new root hash

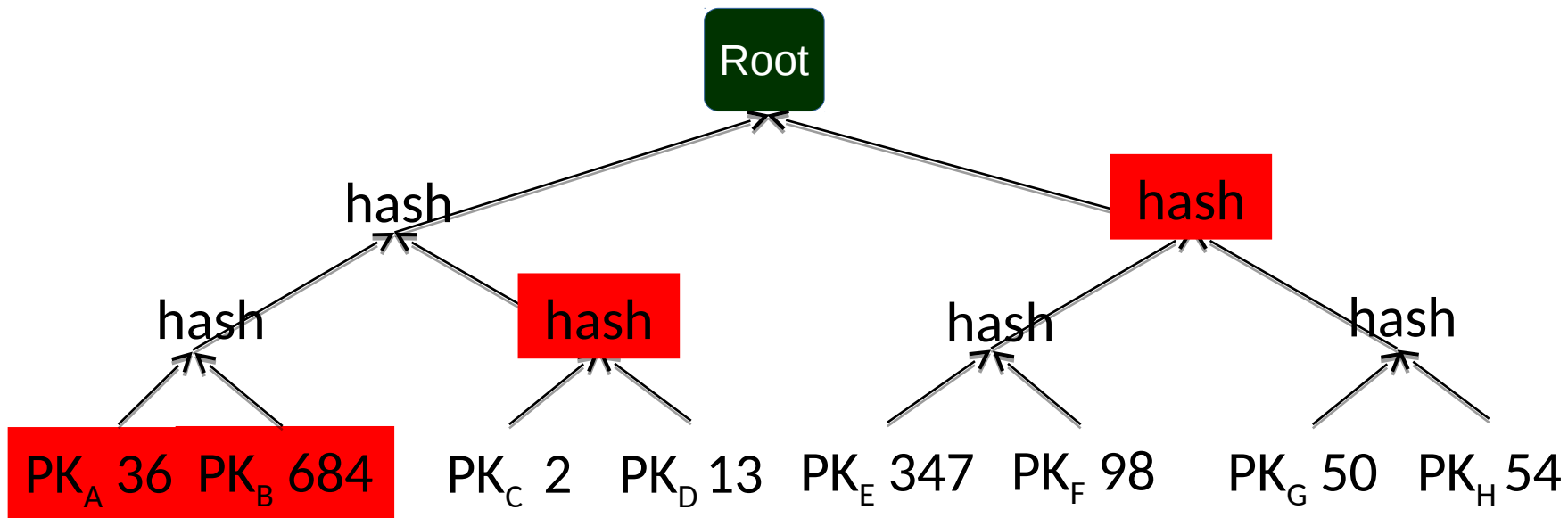
Single operation proof size



- For $N=10^6$, AVL+ proof size \approx 765 bytes
(32-byte hashes, 32-byte keys, 8-byte values)

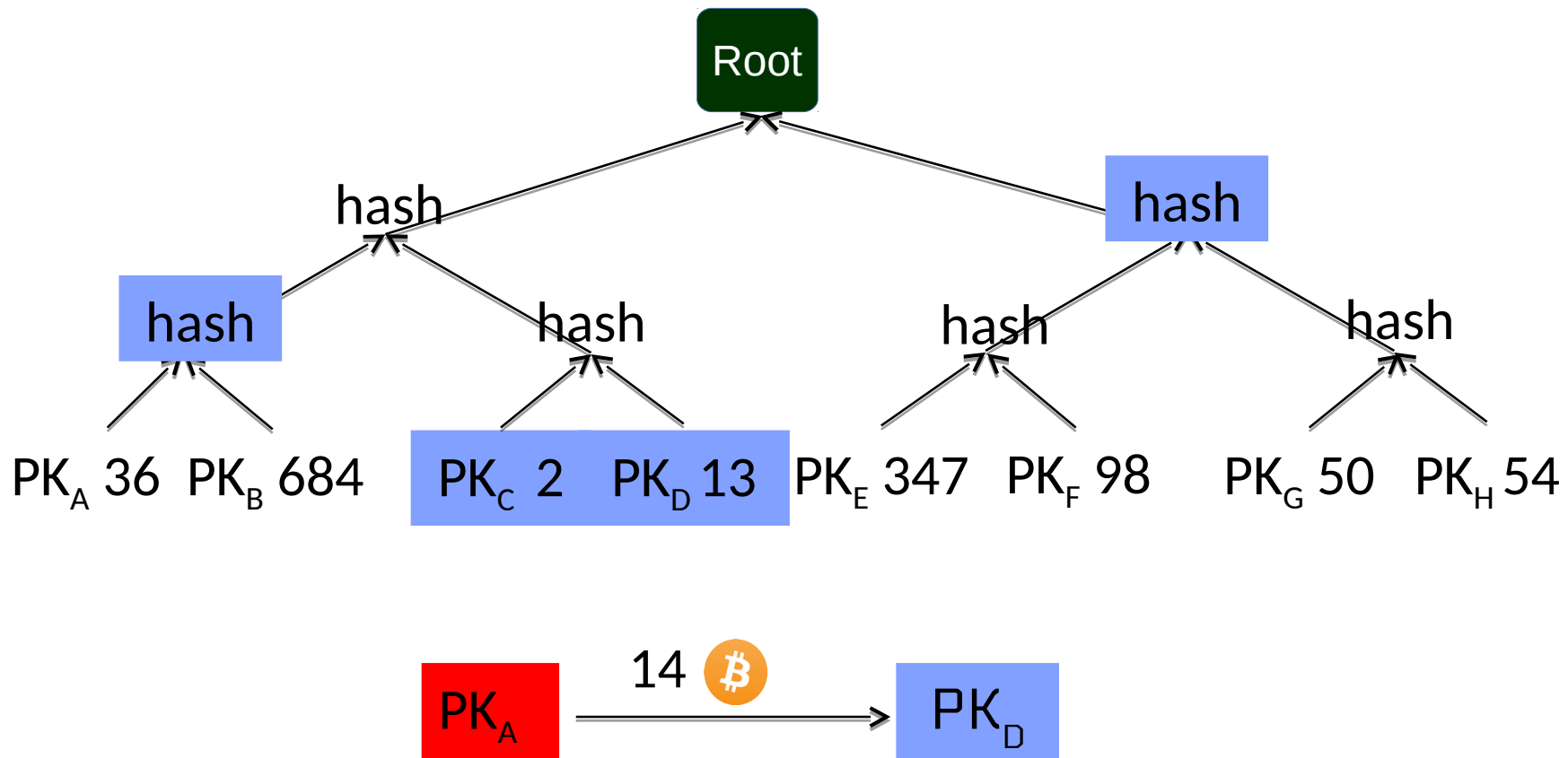
Our improvements: batching

- There are a lot of transactions in block
- Transactions may change same public key
- Multiple proofs can be combined together



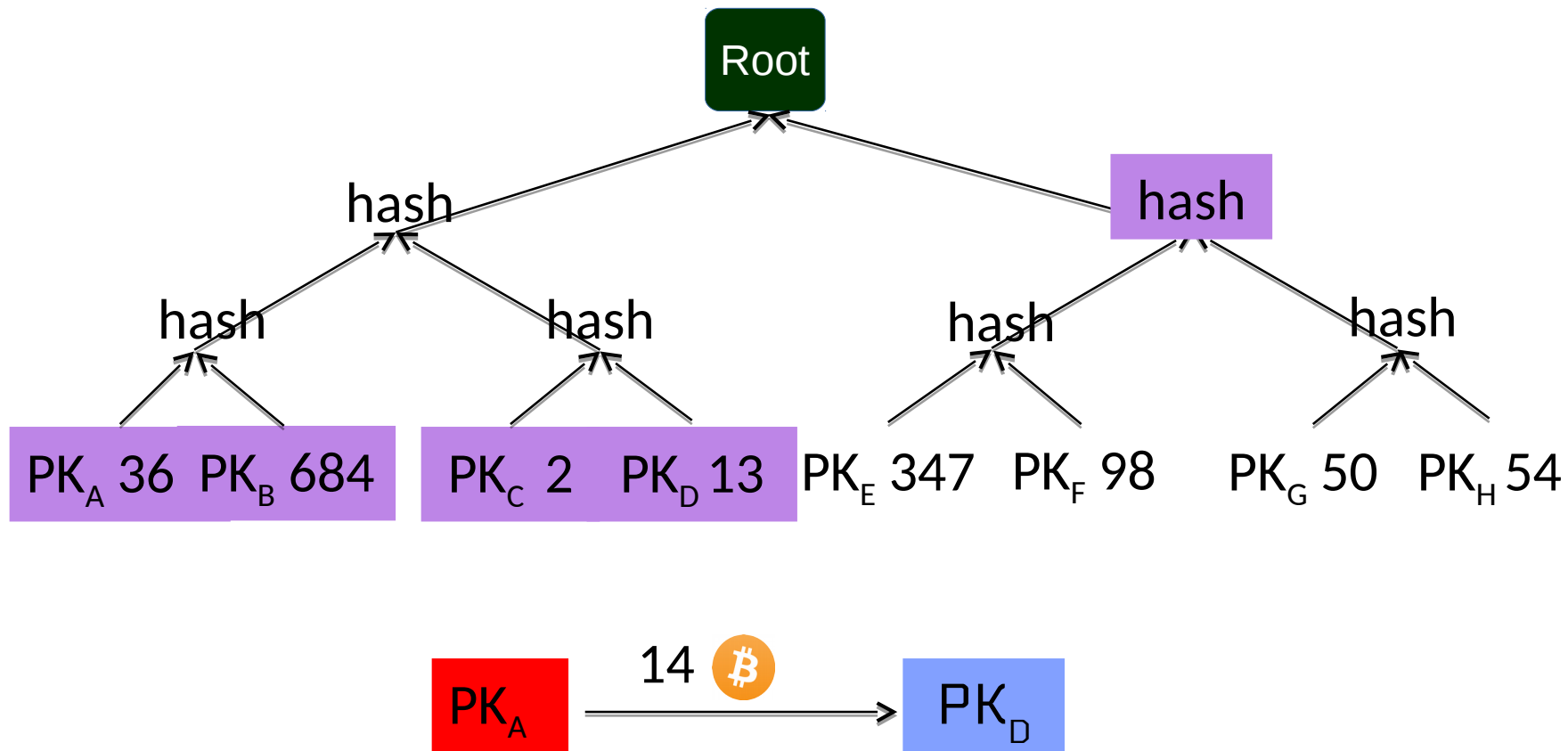
Our improvements: batching

- Proofs can contain same hashes

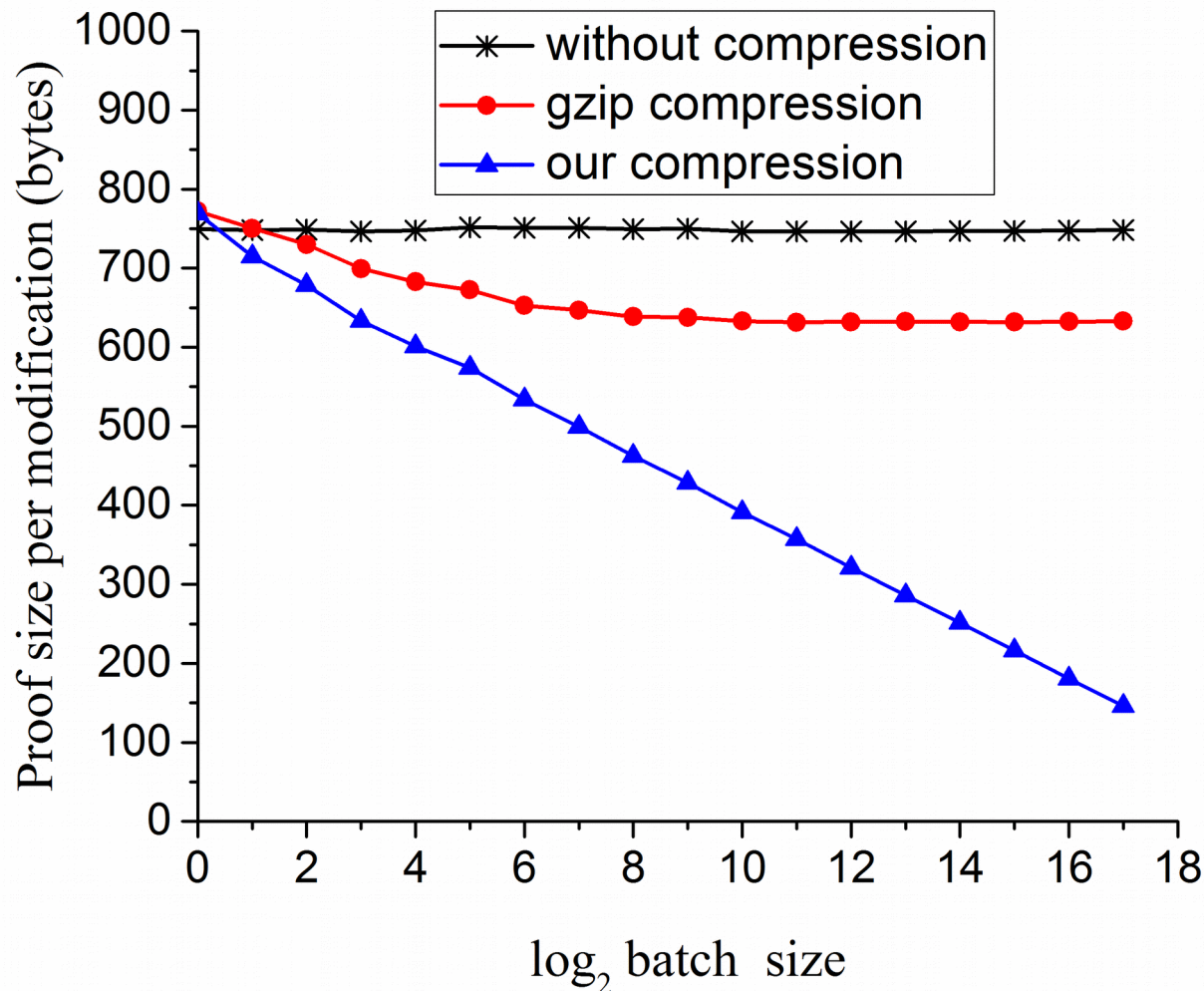


Our improvements: batching

- Some hashes from one proof may be calculated from other proofs

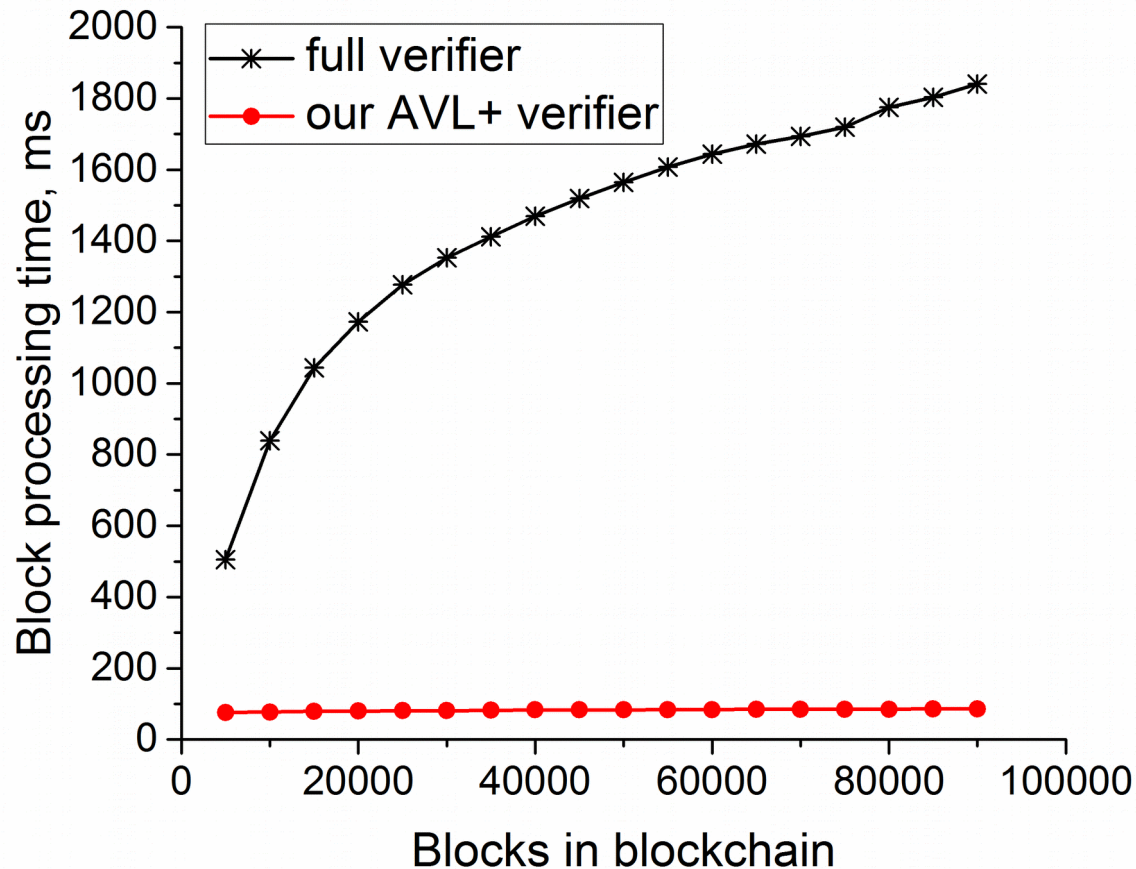


Multiple operations proof size



- For tree $N=10^6$ and batch $B=10^3$, compressed proof size is ≈ 400 bytes, plain ≈ 765 bytes

Validation time



Authenticated data structures enable:

- Verification on low-end hardware
- Mining on commodity hardware

Authenticated state


Allows:

- Download state for random block header and check correctness
- Validate arbitrary transaction
- Validate block transition
- Speed-up validation time
- Keep part of the state and validate proofs for other parts

Light nodes

PoPoW + Authenticated state:

- PoPoW (~100 Kb) + state root hash (32 b)
- Validate both PoW and transactions for arbitrary block

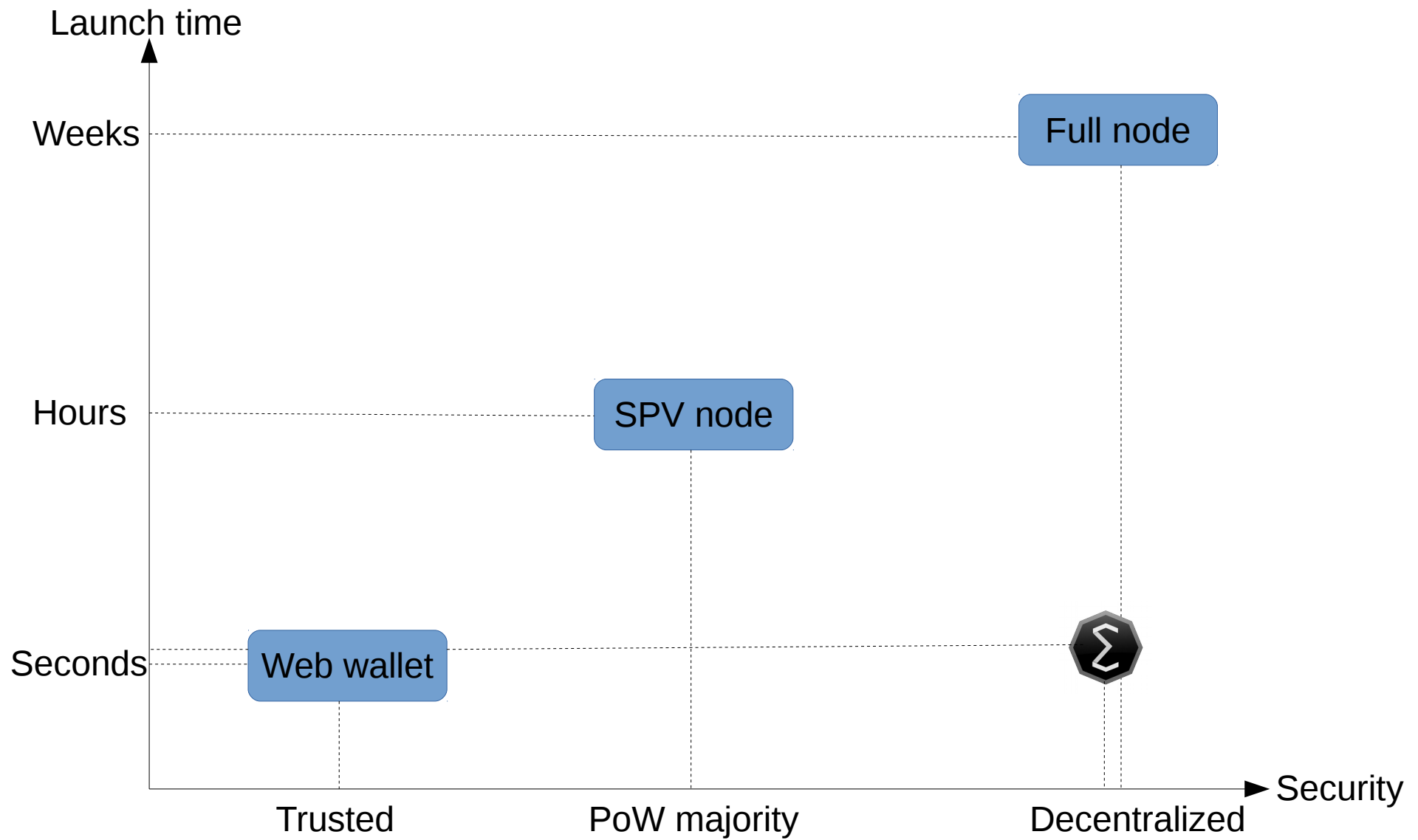
 **hawky** 5:56 PM
uploaded and commented on this image: [IMG_20171214_155505.jpg](#)



1

“ Bitcoin consumes too much energy? Let's see if we can run an Ergo lightnode powered with solar...

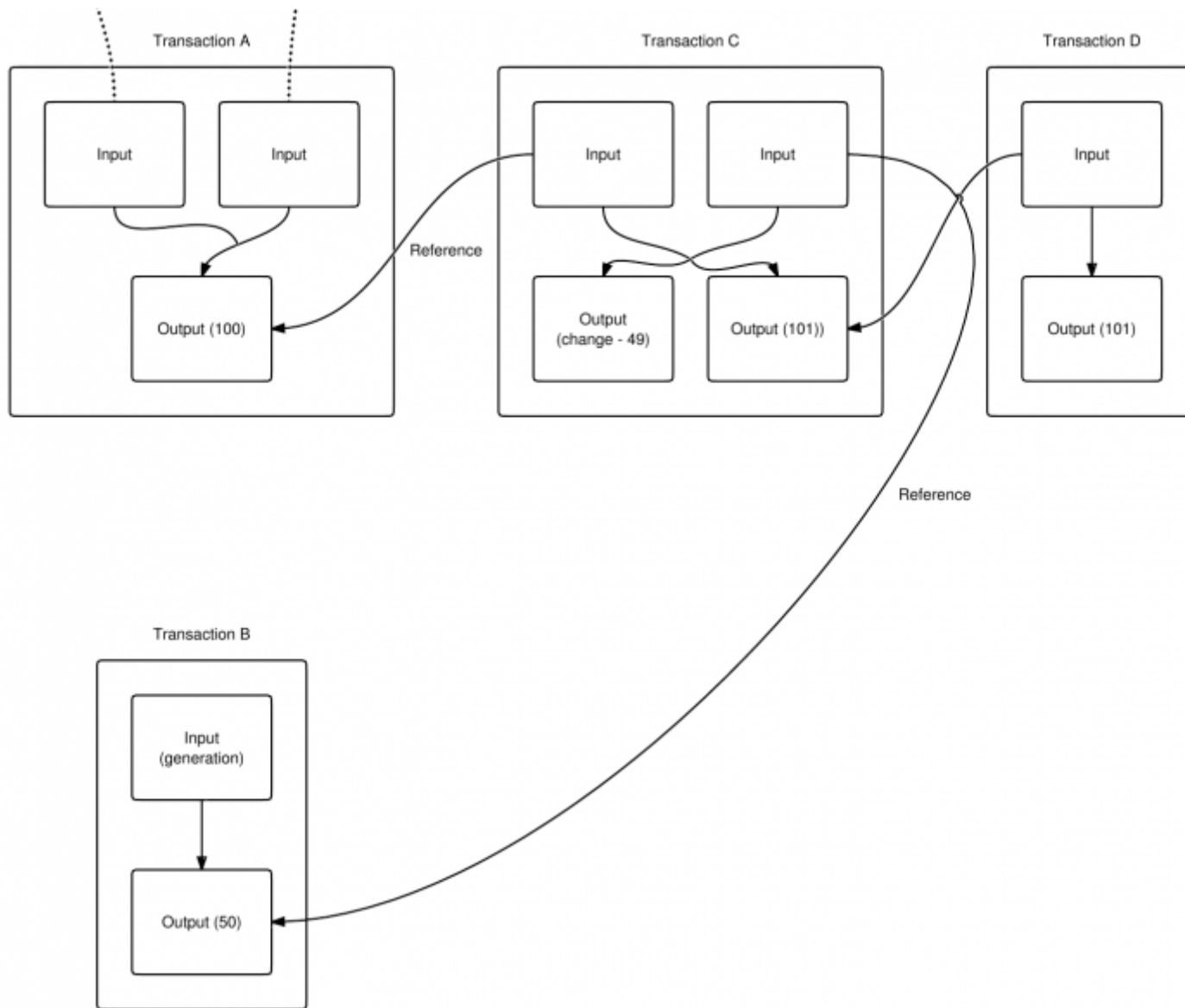
Nodes



Turing completeness of smart contracts

Chepurnoy A., Kharin V., Meshkov D.. Self-Reproducing Coins as Universal Turing Machine

Bitcoin transactions



Bitcoin transactions: Input

Input:

- Previous tx: hash transaction, that created an output we are trying to spend
- Index: output index in that transaction
- ScriptSig: first part of a script. Usually public key and a signature here

Bitcoin transactions: output

Output:

- Value: number of satoshis (1 BTC = 100,000,000 Satoshi), kept in this output
- ScriptPubKey: second part of the script

Bitcoin transactions

- Usual script – Pay-to-PubkeyHash

Pay-to-PubkeyHash

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG  
scriptSig: <sig> <pubKey>
```

- Validation process

Stack	Script	Description
Empty.	<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey>	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Constants are added to the stack.
<sig> <pubKey> <pubKey>	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is duplicated.
<sig> <pubKey> <pubHashA>	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is hashed.
<sig> <pubKey> <pubHashA> <pubKeyHash>	OP_EQUALVERIFY OP_CHECKSIG	Constant added.
<sig> <pubKey>	OP_CHECKSIG	Equality is checked between the top two stack items.
true	Empty.	Signature is checked for top two stack items.

- OP_DUP – copy top element
- OP_HASH160 - RIPEMD160(SHA256())
- OP_EQUALVERIFY - mark tx invalid if $x1 \neq x2$
- OP_CHECKSIG – check signature of tx hash

Bitcoin contracts

<https://en.bitcoin.it/wiki/Contract>

- Atomic swap
 - Micropayment channels
 - Payment network
 - Pay-for-proof contracts
 - ...
-
- What is possible?
 - Assumed to be not Turing complete (with no proofs though)

Ethereum contracts

<https://github.com/ethereum/wiki/wiki/White-Paper#philosophy>

- **Lack of Turing-completeness** - that is to say, while there is a large subset of computation that the Bitcoin scripting language supports, it does not nearly support everything. The main category that is missing is loops. This is done to avoid infinite loops during transaction verification; theoretically it is a surmountable obstacle for script programmers, since any loop can be simulated by simply repeating the underlying code many times with an if statement, but it does lead to scripts that are very space-inefficient. For example, implementing an alternative elliptic curve signature algorithm would likely require 256 repeated multiplication rounds all individually included in the code.

<https://github.com/ethereum/wiki/wiki/White-Paper#computation-and-turing-completeness>

Computation And Turing-Completeness

An important note is that the Ethereum virtual machine is Turing-complete; this means that EVM code can encode any computation that can be conceivably carried out, including infinite loops. EVM code allows looping in two ways. First, there is a `JUMP` instruction that allows the program to jump back to a previous spot in the code, and a `JUMPI` instruction to do conditional jumping, allowing for statements like `while x < 27: x = x * 2`. Second, contracts can call other contracts, potentially allowing for looping through recursion. This naturally leads to a problem: can malicious users essentially shut miners and full nodes down by forcing them to enter into an infinite loop? The issue arises because of a problem in computer science known as the halting problem: there is no way to tell, in the general case, whether or not a given program will ever halt.

Ethereum contracts

- Ethereum added possibly infinite loops for Turing completeness
- In blockchain computations should be limited, otherwise blockchain won't grow.
- Ethereum introduced gas – during script evaluation, every operation add predefined complexity to consumed gas. When gas limit is reached – computations stops

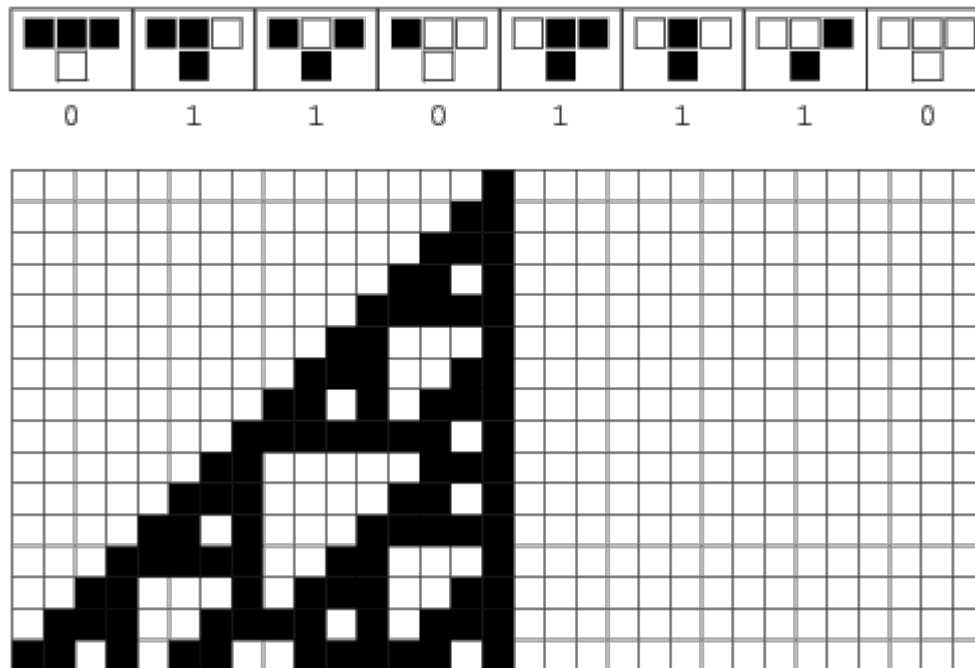
Ethereum contracts

- Runtime cost analysis – DoS attacks
- Block gas limit – still no infinite loops
- The same as Bitcoin from Turing-completeness point of view

Turing-completeness

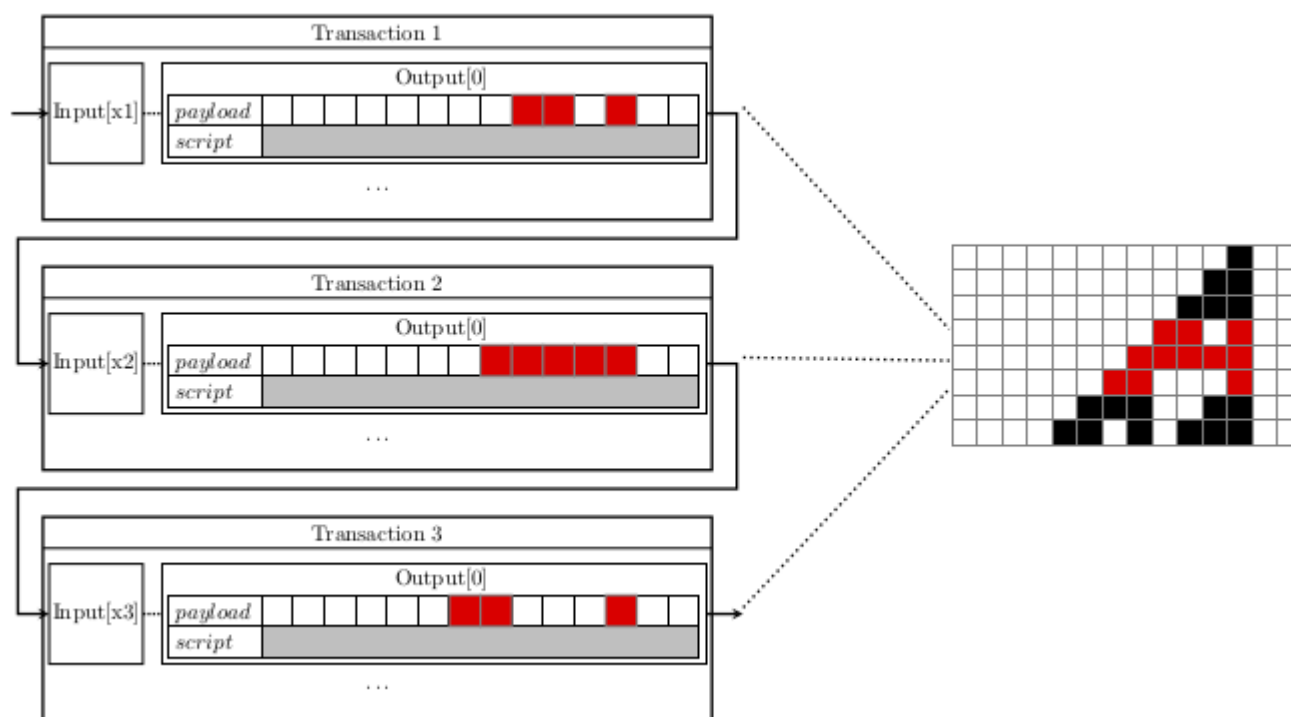
- Turing completeness proof – implementation of known Turing complete system
- Simple case – Rule 110

rule 110



Turing-completeness

- Even if you don't have infinite loop inside a block, you have it between blocks



Turing-completeness

- Rule 110 script pseudocode

Algorithm 1 Transition function of the Rule 110 automaton

```
1: function CALCBIT( $\ell$ ,  $c$ ,  $r$ )  
2:   return  $(\ell \wedge c \wedge r) \oplus (c \wedge r) \oplus c \oplus r$   
3: end function
```

Algorithm 2 Script, that ensures that the transaction performs correct rule 110 transformation keeping the same rules for further iterations

```
1: function VALIDATE(in, out)  
2:   function ISRULE110(inLayer, outLayer)  
3:     function PROCCELL( $i$ )  
4:        $\ell \leftarrow \text{inLayer}[i - 1 \bmod \text{inLayer.size}]$   
5:        $c \leftarrow \text{inLayer}[i]$   
6:        $r \leftarrow \text{inLayer}[i + 1 \bmod \text{inLayer.size}]$   
7:       return CALCBIT( $\ell$ ,  $c$ ,  $r$ )  
8:     end function  
9:     return outLayer == inLayer.indexes.map(PROCCELL)  
10:  end function  
11:  return ISRULE110(self.payload, out[0].payload)  $\wedge$  (self.script == out[0].script)  
12: end function
```

Ergo script example

Rule 110 Ergo-script:

```
let indices: Array[Int] = Array(0, 1, 2, 3, 4, 5)
let inLayer: Array[Byte] = SELF.R3[Array[Byte]].value
fun procCell(i: Int): Byte = {
  let l = inLayer((if (i == 0) 5 else (i - 1)))
  let c = inLayer(i)
  let r = inLayer((i + 1) % 6)
  intToByte((l * c * r + c * r + c + r) % 2)
}
(OUTPUTS(0).R3[Array[Byte]].value == indices.map(procCell)) &&
  (OUTPUTS(0).propositionBytes == SELF.propositionBytes)
```


Turing-completeness

- Number of blocks is infinite → possibly infinite loop
- But to ensure Turing-completeness, you also need infinite input size
- Workaround – process rule110 string by parts

Ergo smart contracts

Chepurnoy A. et al. Σ -State, an Alternative to Bitcoin Script

Ergo script

Ergo output consists of registers:

- R0 – monetary value
- R1 – id of transaction, created this output + this output position
- R2 – protecting script
- R3 – custom tokens value
- R4-R8 – non-mandatory registers

Ergo script

Ergo input consists of:

- boxId – id of output, tx is trying to spend
- SpendingProof:
 - ProofBytes – signature for resulting sigma protocol
 - Extension - map from keys (byte) to values (some constants)

Ergo script

Ergo transaction validation:

- `Inputs.map(_.R1).sum == outputs.map(_.R1).sum`
- For each input:
 - Reduce script of spending output to sigma protocols, using environment and spending proof extension
 - Check that spending proof `ProofBytes` is a correct proof for resulting sigma protocol
- P.s. actually first rule may be enforced by the script

Ergo script

Ergo script:

- Only operations, that allows to estimate script complexity before execution
- Constant-time access to environment
- Not jump operator and infinite loops
- Operations on predefined collections

If you need some computation:

- Estimate work done before execution
- Put to one or multiple transactions

Ergo script

Ergo script have access:

- Transaction inputs/outputs
- Blockchain height (last few headers in future)
- State root hash

Ergo script

Ergo operations:

- Arithmetics: (+, -, *, /, %)
- Boolean operations (||, &&, xor)
- If-then-else
- Arrays: `append`, `slice`, `map`, `fold`,
`exists`, `forall`, etc.
- Lambdas: `OUTPUTS.exists(fun (out: Box) =
out.value >= minToRaise)`
- Deserialize → P2SH, MAST
- Crypto: blake2b, sha256 + ...

Ergo script

Ergo script (crypto):

- Based on Σ protocols
- Composable (OR, AND, k-out-of-n)
- Turns into signatures by using Fiat-Shamir transformation
- Efficient

Ergo script

Ring signature (1 out of 3):

- $\text{pkA} \parallel \text{pkB} \parallel \text{pkC}$

With And ($A+B$ or $C+D$)

- $(\text{pkA} \ \&\& \ \text{pkB}) \parallel (\text{pkC} \ \&\& \ \text{pkD})$

Without revealing who signed

Ergo script example

Emission:

```
let epoch = 1 + ((HEIGHT - fixedRatePeriod) / epochLength)
let out = OUTPUTS(0)
let coinsToIssue = if(HEIGHT < fixedRatePeriod) fixedRate
else fixedRate - (oneEpochReduction * epoch)
let correctCoinsConsumed = coinsToIssue == (SELF.value - out.value)
let sameScriptRule = SELF.propositionBytes == out.propositionBytes
let heightIncreased = HEIGHT > SELF.R3[Long].value
let heightCorrect = out.R3[Long].value == HEIGHT
let lastCoins = SELF.value <= oneEpochReduction
(correctCoinsConsumed && heightCorrect && heightIncreased && sameScriptRule)) ||
(heightIncreased && lastCoins)
```

Ergo script

- MAST

```
AND(DeserializeContext(21.toByte, SBoolean),  
  If(EQ(SizeOf(Inputs), 1),  
    EQ(CalcBlake2b256(TaggedByteArray(21.toByte)), ByteArrayConstant(script1Hash)),  
    EQ(CalcBlake2b256(TaggedByteArray(21.toByte)), ByteArrayConstant(script2Hash))  
  )  
)
```

Economy

-
1. Chepurnoy A., Kharin V., Meshkov D. A Systematic Approach To Cryptocurrency Fees.

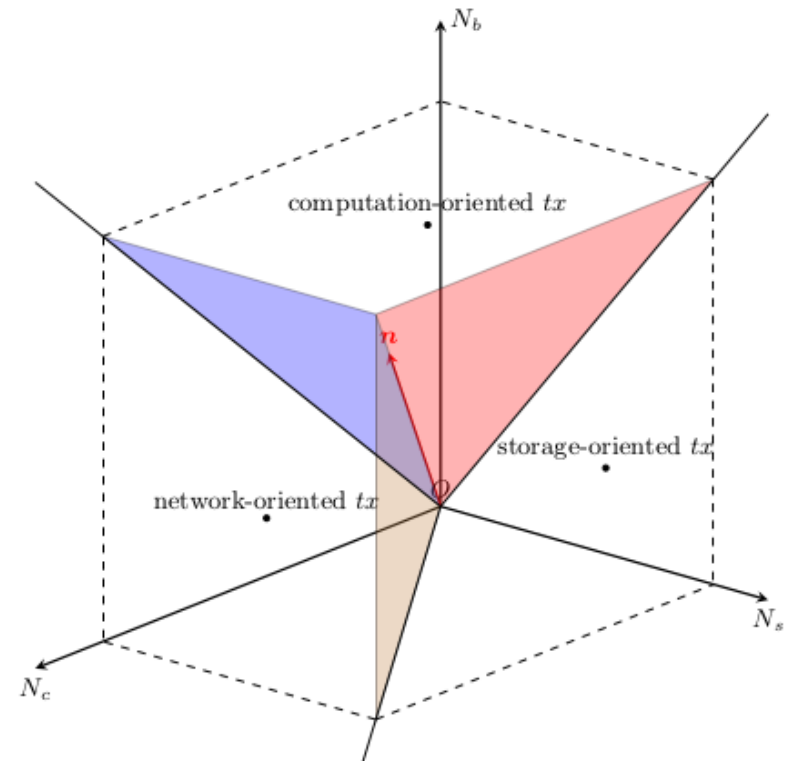
Economy

- Inflation, fees, etc. are hardcoded at the beginning
- Can it survive in the long term?
- Some discussions on forums, few papers, no comprehensive research
- <https://cesc.io> - first attempts to study crypto economics (in 2017)

Fees

Cost	Proportional to	Dominant in
Network	Transaction size	Blockchain for money
Computation	CPU utilization	Blockchain for contracts
Storage	State size changes	Blockchain for database

- Result fee = $F(N_b, N_c, N_s)$
- E.g. $\max(N_b, N_c, N_s)$



Transaction resource consumption

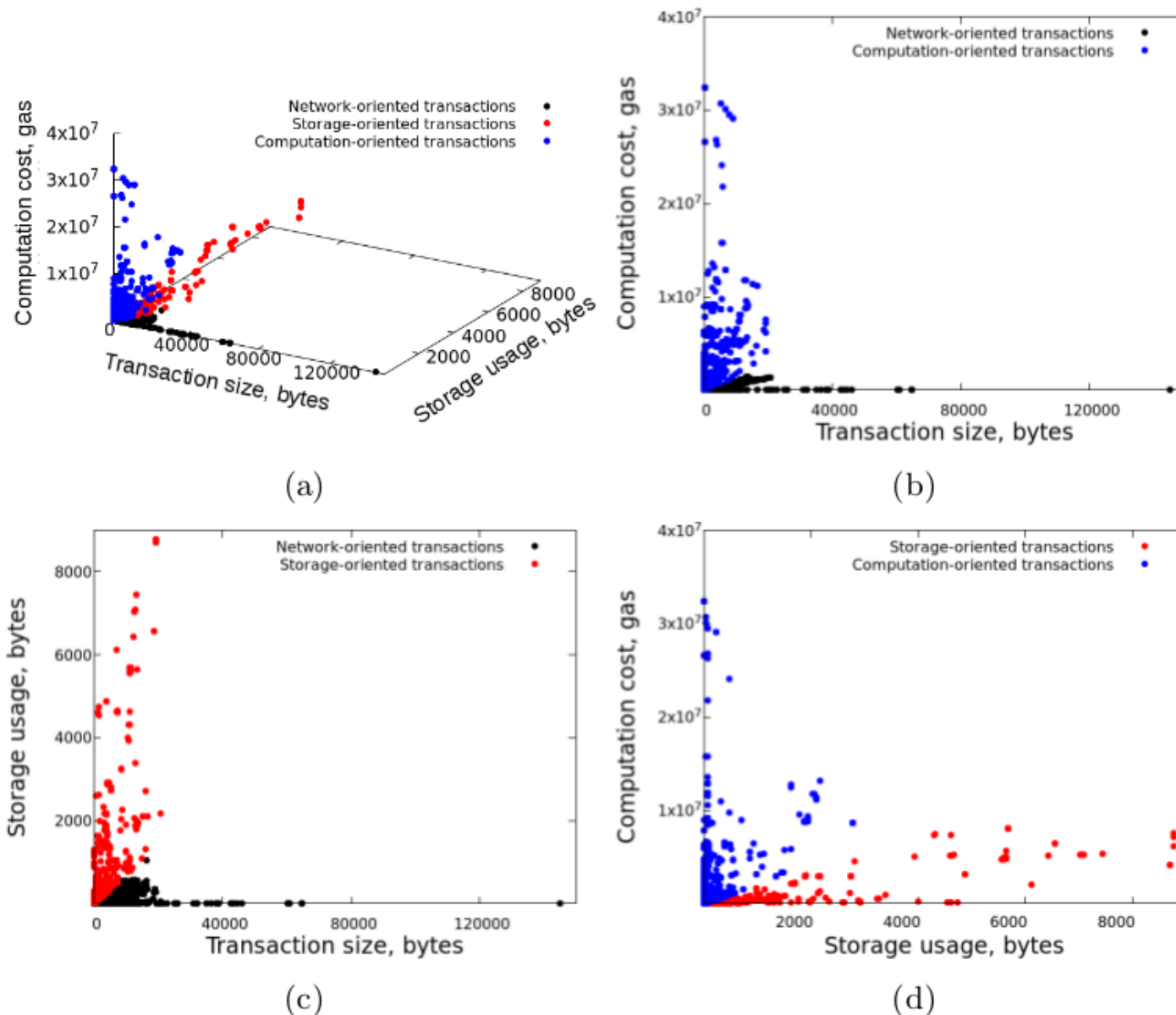


Fig. 4: Ethereum transactions differentiation by resource consumption

Ergo economy

- (not implemented yet)
- Mandatory part of a script, that allows anyone to take part of output for keeping it into State
- Return coins into circulation
- Upper-bound for state size
- Additional miners reward

Contacts

- <https://ergoplatform.org>
- <https://t.me/ergoplatform>
- <https://twitter.com/ergoplatformorg>
- <http://www.slideshare.net/DmitryMeshkov>
 - <https://twitter.com/DmitryMeshkov>
 - Dmitry.meshkov@iohk.io