Mining in Logarithmic Space

An exponential improvement on blockchain storage

Aggelos Kiayias, Nikos Leonardos, Dionysis Zindros
ATHECRYPT 2020, NTUA

Notational conventions

Concat: B_0 B_1 B_2 … B_n

i-th item from the beginning or the end: C[i], C[-i]

Range: C[i:j], C[i:], C[:j]

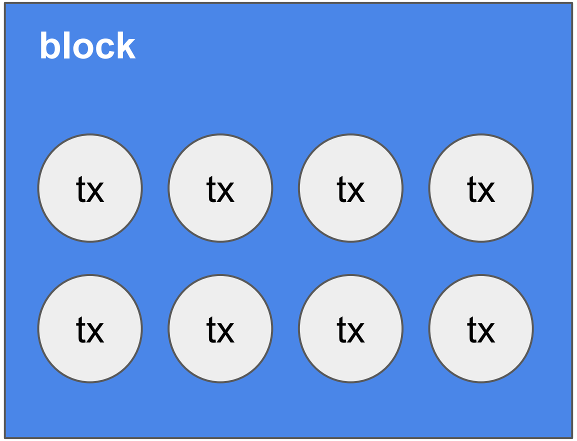
Range with items: C{A:Z}, C{A:}, C{:Z}

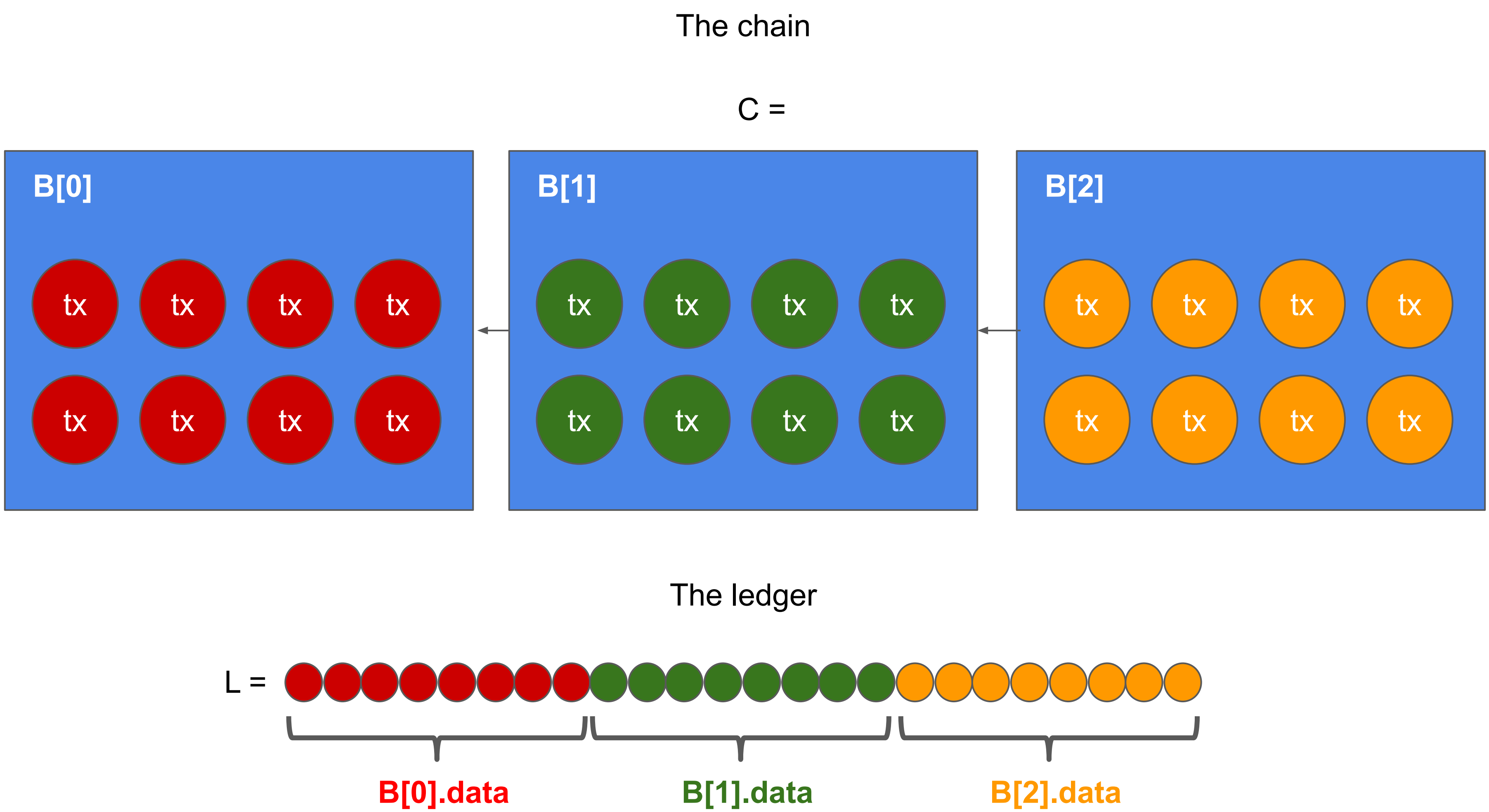
Keep only *μ*-superblocks: $C{\uparrow}^{\mu}$

Blockchains as transaction serializers

- Blockchains are **chains of blocks**: $C = B_0\ B_1\ B_2 \ldots B_n$
- The first block is **Genesis**: C[0] = G
- A **transaction** is part of a block
  - and belongs to a transaction language L_{tx}
- Each block contains **transaction sequences**: $B.data = tx_0\ tx_1\ tx_2 \ldots tx_m$
- The blockchain serializes transactions into a **ledger**:
  - $L = B_0.data \mid\mid B_1.data \mid\mid \ldots \mid\mid B_n.data$
- A ledger is a **transaction sequence** across the whole chain

B =





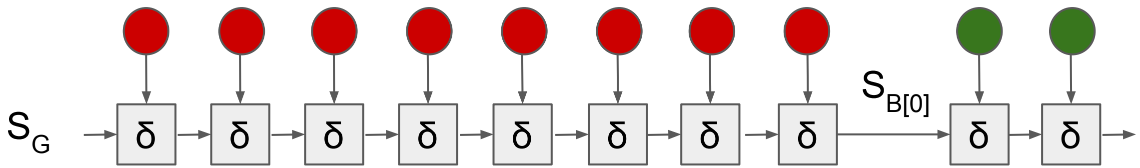
The chain
C =
B[0]
tx tx tx tx
tx tx tx tx
B[1]
tx tx tx tx
tx tx tx tx
B[2]
tx tx tx tx
tx tx tx tx
The ledger
L =
B[0].data
B[1].data
B[2].data

Blockchains as a state machine

- Blockchains have a **current** state
  - Belongs to a state language L_S
- It starts with the Genesis state: S_G
- It evolves by applying a **transition function δ**
  - It takes a **previous state** and a **transaction**
  - It outputs the **next state**, or ⊥ if transaction cannot be applied
- $\delta: L_S \times L_{tx} \rightarrow L_S \cup \{\bot\}$
- $S' = \delta(S, tx)$

Applying the transition function repeatedly

- We can apply δ multiple times:
  $\delta^*(S, \varepsilon) = S$
  $\delta^*(S, tx_0 \,||\, \mathbf{tx}) = \delta^*(\delta(S, tx_0), \mathbf{tx})$, if $\delta(S, tx_0) \neq \bot$
  $\bot$, otherwise
- We can apply δ to blocks:
  $\delta(S, B) = \delta^*(S, tx_0\, tx_1 \ldots tx_n)$, where $B.data = tx_0\, tx_1 \ldots tx_n$
- We can apply δ* to chains:
  $\delta^*(S, \varepsilon) = S$
  $\delta^*(S, BC) = \delta^*(\delta(S, B), C)$, if $\delta(S, B)$ if $\delta(S, B) \neq \bot$
  $\bot$, otherwise



S_G
δ
δ
δ
δ
δ
δ
δ
δ
$S_{B[0]}$
δ
δ

Historical VS current state

- Blockchain systems contain two types of *state*:
  - Historical state
  - Current state
- *Historical state* involves all transactions
- *Current state* involves data needed to verify new blocks

Examples of current state and evolution

BTC current state: S is the UTXO set.
Transaction consumes UTXOs and produces new UTXOs:

$\delta(S, (ins, outs)) = S \setminus ins \cup outs$, if $ins \subseteq S$ and $\Sigma_{p \in ins}\, p.v > \Sigma_{p \in outs}\, p.v$
$\perp$, otherwise

ETH current state: S is account balances.
Transaction consumes balance and creates new balance:

$\delta(S, (from, to, value)) = S \setminus \{(from, a), (to, b)\} \cup \{(from, a - value), (to, b + value)\}$,
if $(from, a) \in S$, $(to, b) \in S$ and $a \geq value$
$\perp$, otherwise

The problem of state storage

Idea! As full nodes, we can **drop history** and keep **current state** only:

- We can know how much balance every user has (in UTXOs or accounts)
- We cannot know what transactions they did in the past
- We cannot know how their balance evolved over time

Historical data is *pruned* for efficiency. Interested parties can still store it.

Dropping both block headers and txs

- We propose a new blockchain protocol in which *no one* stores the chain
- We prune:
  - Historical txs
  - Old blocks
  - Old block *headers*

What to store? What to send?

- Proposal: Full nodes **no longer store a chain**
- Instead, they store a *compressed chain*
- They *mine* on top of a *compressed chain* and extend it
- If they mine successfully, they send the *new compressed chain* to the network
- Upon receiving a *compressed chain* from the network, the nodes *compare* it against their currently adopted *compressed chain* for length
- Nodes adopt the *longest compressed chain*

The chain compression algorithm

- We want an algorithm that, given a chain C, compresses it to compressed chain π = compress(C)
- We want to be able to mine on top of π
- Given π, mine a block B, then calculate the post-mining compressed state
- The *online condition* must hold:
  if compress(C) = π, then compress(π || B) = compress(CB)

Extending the Backbone protocol to compress

Algorithm 2 The function that finds the "best" chain, parameterized by function $\max(\cdot)$. The input is $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$.

function maxvalid($\mathcal{C}_1, \ldots, \mathcal{C}_k$) — maxvalid($\pi_1$, π_2, … π_k)
    $temp \leftarrow \varepsilon$
    for $i = 1$ to k **do**
        if validate($\mathcal{C}_i$) **then**
            $temp \leftarrow \max(\mathcal{C}_i, temp)$ — max predicate must be defined
        end if
    end for
    return $temp$
end function

Algorithm 3 The *proof of work* function, parameterized by q, T and hash functions $H(\cdot), G(\cdot)$. The input is $(x, \mathcal{C})$.

```
function pow(x, C)                                   pow(x, π)
    if C = ε then                                    ▷ Determine proof of work instance
        s ← 0
    else
        ⟨s′, x′, ctr′⟩ ← head(C)
        s ← H(ctr′, G(s′, x′))
    end if
    ctr ← 1
    B ← ε
    h ← G(s, x)
    while (ctr ≤ q) do
        if (H(ctr, h) < T) then                      ▷ This H(·) invocation subject to the q-bound
            B ← ⟨s, x, ctr⟩
            break
        end if
        ctr ← ctr + 1
    end while
    C ← CB              π = compress(πB)             ▷ Extend chain
    return C            return π
end function
```

Committing to current state within a block

- If we know only the tip B of the blockchain C is valid and we receive a new block B’
- How do we validate the application data of CB’?
- How to validate application data of block B’?
- The known valid block B must commit to the *current state:* B.commit = $\delta(S_G, C[:-1])$
- Then it suffices to check that $\delta(B.commit, B'.data) \neq \perp$
- Ethereum already does this: Blocks include the root of the State Merkle–Patricia Trie
- Bitcoin doesn’t do it, but can easily (soft fork) be extended to include a UTXO Merkle Tree root

The problem of block verification: Forking chains

Can we verify a new incoming block for correctness if we don't have history? Yes!

Consider incoming block B extending chain C into CB.
Consider state after block C, $S_C = \delta(S_G, C)$. If $\delta(S_C, B) \neq \bot$, then block B is valid.

Suppose we receive from the network a chain C' longer than our adopted chain C.
We have already validated C. All we need to do is check:
$\delta(S_{C \cap C'}, C'\{(C \cap C')[-1]:\}) \neq \bot$

We have already (C∩C')[-1].state. To validate, we need to know *all* the blocks in C'{(C∩C')[-1]:}.

Extending the Backbone protocol to compress

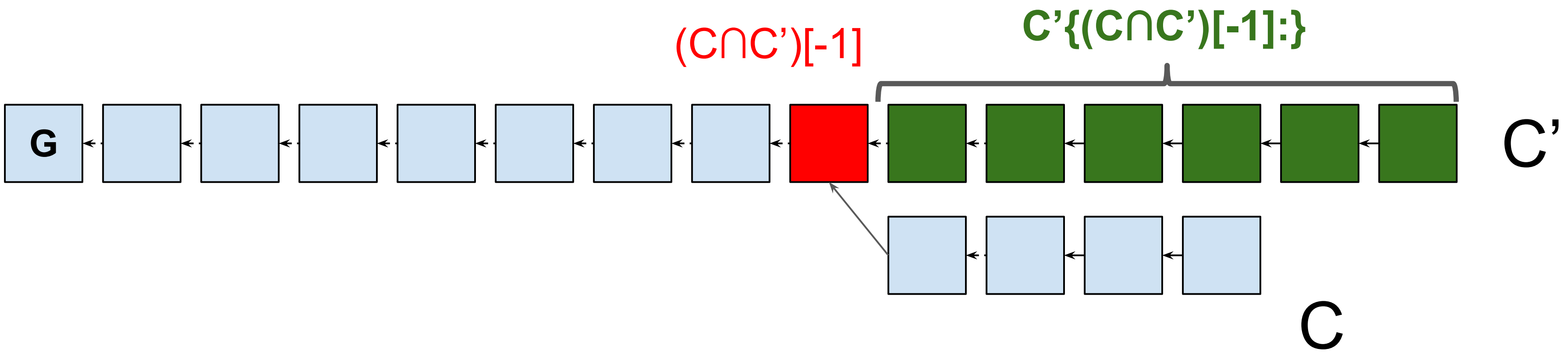
Algorithm 1 The *chain validation predicate*, parameterized by q, T, the hash functions $G(\cdot), H(\cdot)$, and the *content validation predicate* $V(\cdot)$. The input is $\mathcal{C}$.

function validate($\mathcal{C}$)
    $b \leftarrow V(\mathbf{x}_\mathcal{C})$ **sufficient to validate δ in last k blocks**
    if $b \wedge (\mathcal{C} \neq \varepsilon)$ **then** ▷ The chain is non-empty and meaningful w.r.t. $V(\cdot)$
        $\langle s, x, ctr \rangle \leftarrow \text{head}(\mathcal{C})$
        $s' \leftarrow H(ctr, G(s, x))$
        repeat
            $\langle s, x, ctr \rangle \leftarrow \text{head}(\mathcal{C})$
            if $\text{validblock}_q^T(\langle s, x, ctr \rangle) \wedge (H(ctr, G(s, x)) = s')$ **then**
                $s' \leftarrow s$ ▷ Retain hash value
                $\mathcal{C} \leftarrow \mathcal{C}^{\lceil 1}$ ▷ Remove the head from $\mathcal{C}$
            else
                $b \leftarrow \text{False}$
            end if
        until $(\mathcal{C} = \varepsilon) \vee (b = \text{False})$
    end if
    return (b)
end function

validation predicate must be redefined

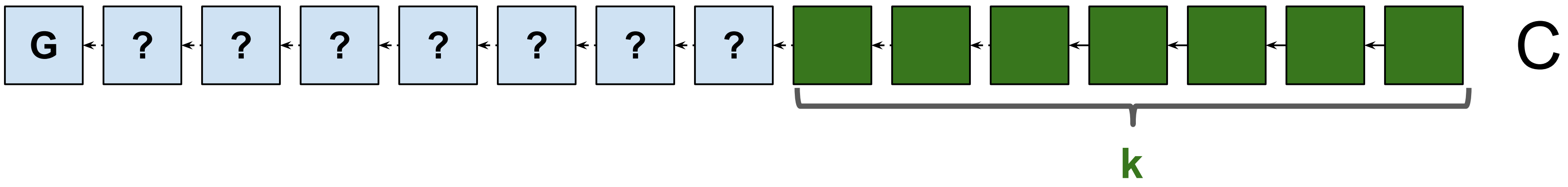
Keeping the last k blocks

- Honest majority assumption gives rise to Common Prefix property
- There will never be accidental forks longer than *k* blocks
  i.e. $|C'\{(C \cap C')[-1]:\}| \leq k$
- Therefore, for validation we just need to keep the last *k* blocks of the chain
- Define compress(C) = C[-k:]
- The online condition holds: If π = compress(C), then
  compress(CB) = C[-k+1:] B = compress(compress(C)B)



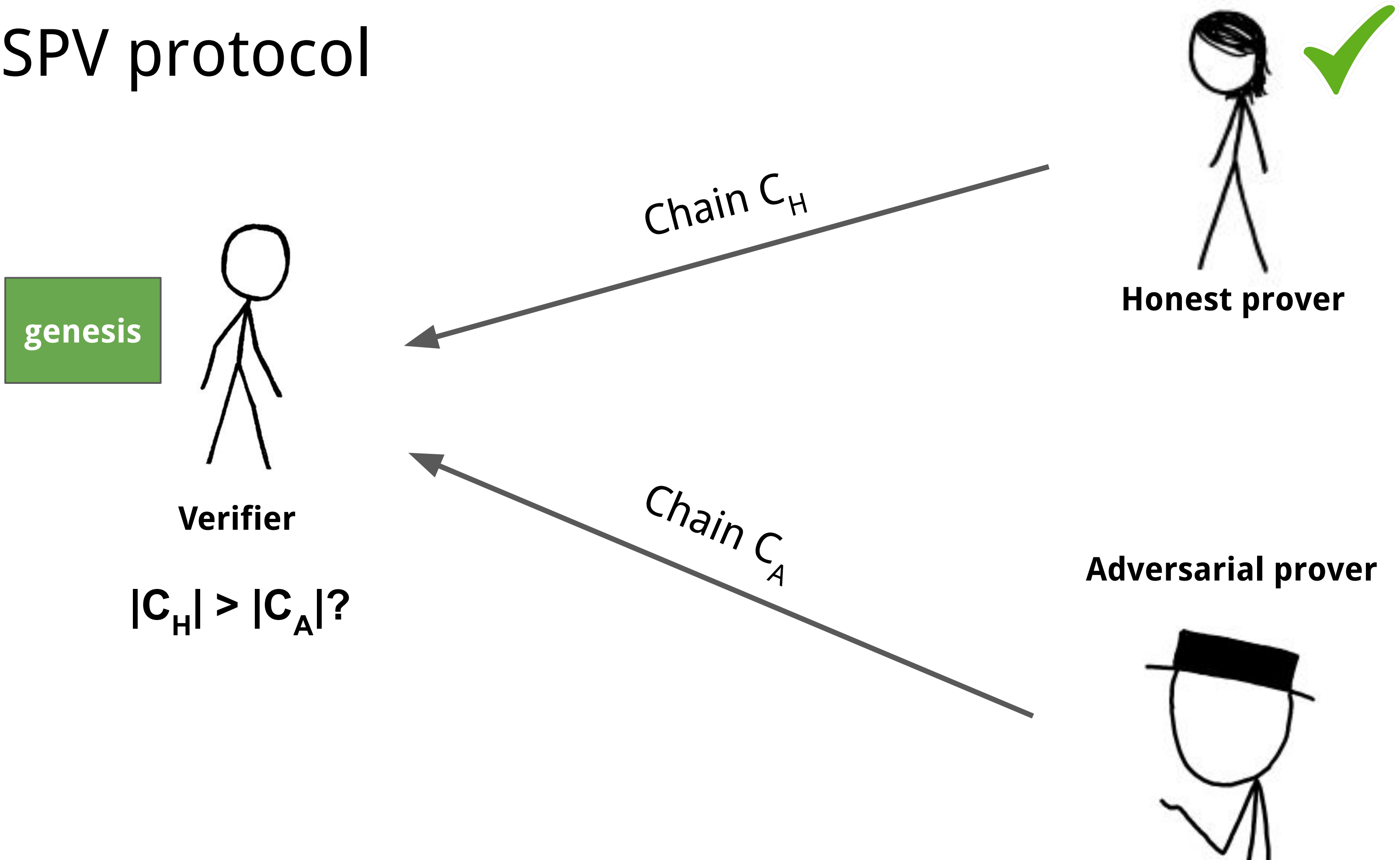
Bootstrapping from genesis

- ...but we can't just keep the last k blocks
- How can a node waking up from genesis sync?
- They need to know what the longest chain is

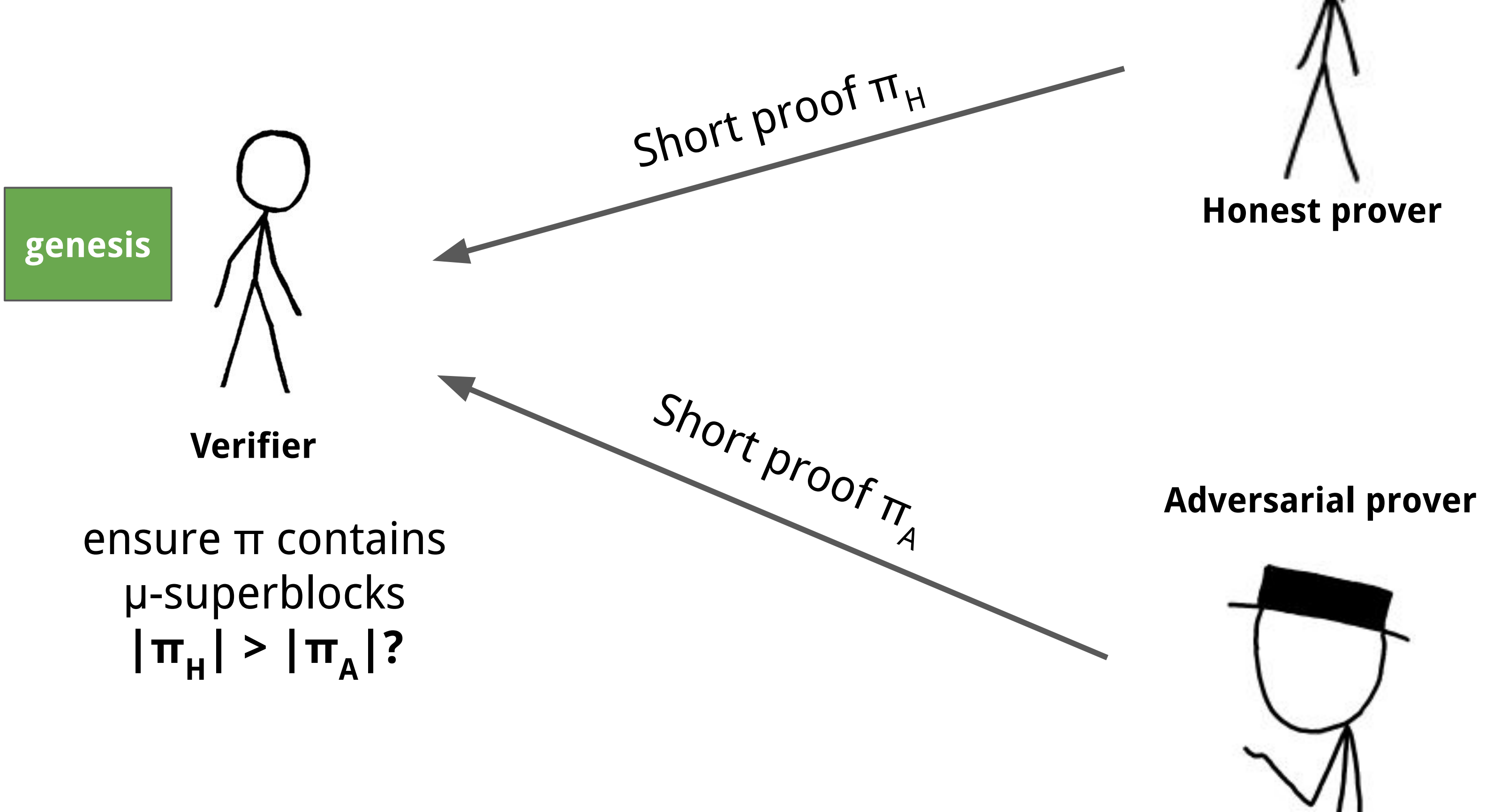


NIPoPoWs to the rescue

SPV protocol



Proof of Proof-of-Work protocol



The Prover/Verifier model

- We don't care about adversarial verifiers!
- Honest verifier connects to *multiple* provers
- At least one prover is honest -- we don't know which
- ~~Prover runs a full node~~
  Prover runs a light node! Everyone runs a light node!
- Verifier wakes up stateless (has genesis block only)
- Each prover sends a *proof* (=compressed chain) to the verifier
- Verifier chooses one of the proofs as legitimate
- Verifier decides about a value of a predicate *p* of the honest chain

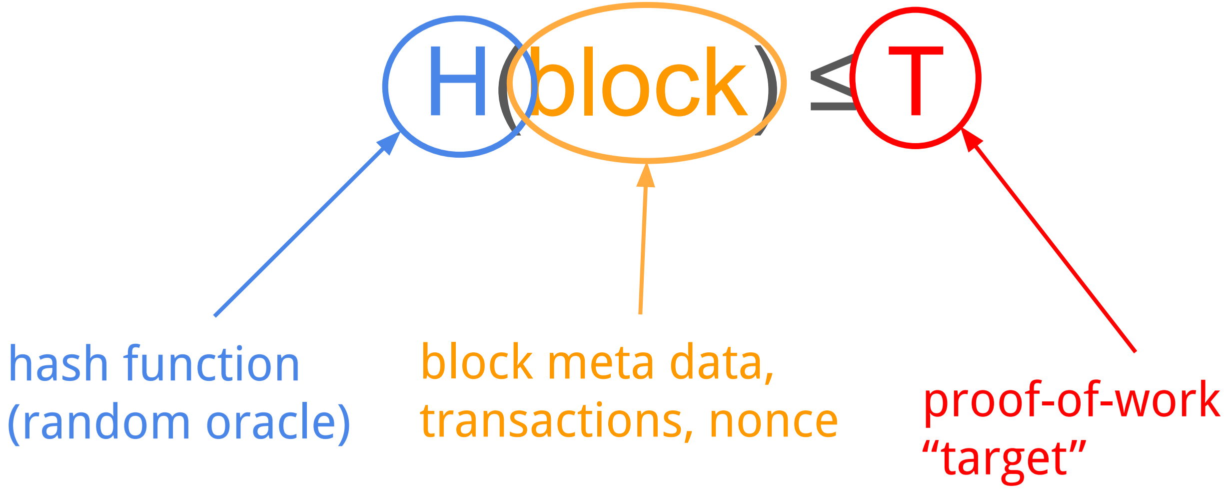
Can we use any NIPoPoW?

There are two NIPoPoW constructions in the literature:

- Superblock NIPoPoWs (Kiayias, Miller, Z)
  - Deterministic
- FlyClient NIPoPoWs (Benedikt Bünz, Kiffer, Luu, Zamani)
  - Probabilistic

Online property **requires determinism**. We will use superblock NIPoPoWs.

The proof-of-work equation



Superblocks

Some blocks achieve a **lower target** than required

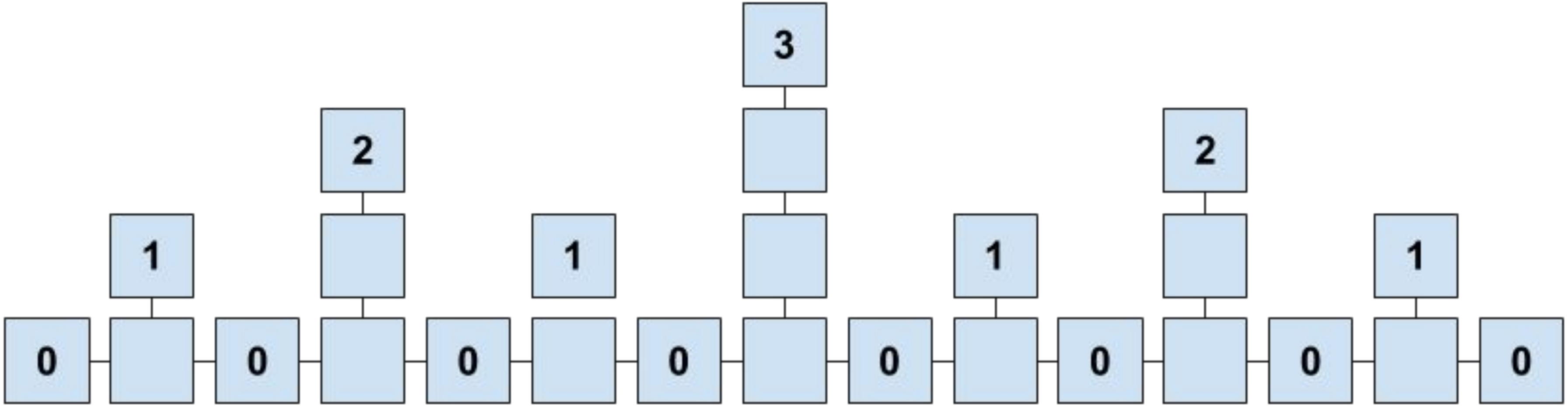
μ-supertarget

$$\Pr[H(\text{block}) \leq T / 2^{\mu} \mid H(\text{block}) \leq T] = 2^{-\mu}$$

The μ-superblock condition

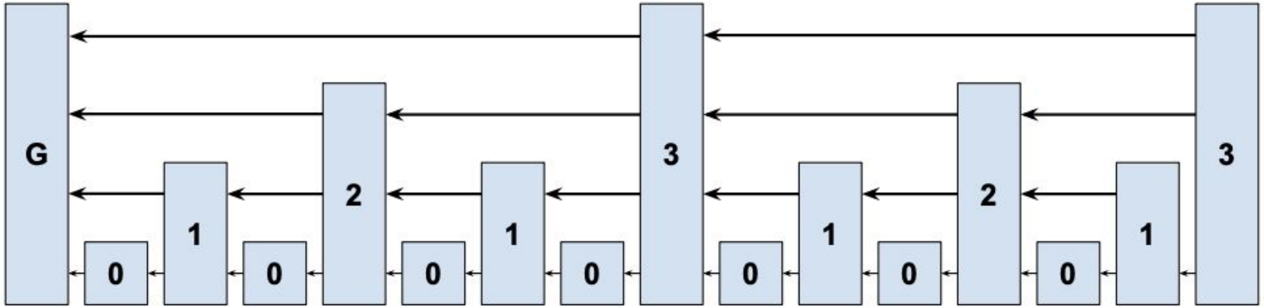
- All blocks are 0-superblocks
- Half the blocks are 1-superblocks
- ¼ of blocks are 2-superblocks
- ⅛ of blocks are 3-superblocks

The superchain*



* your results may vary – **probabilistic** structure

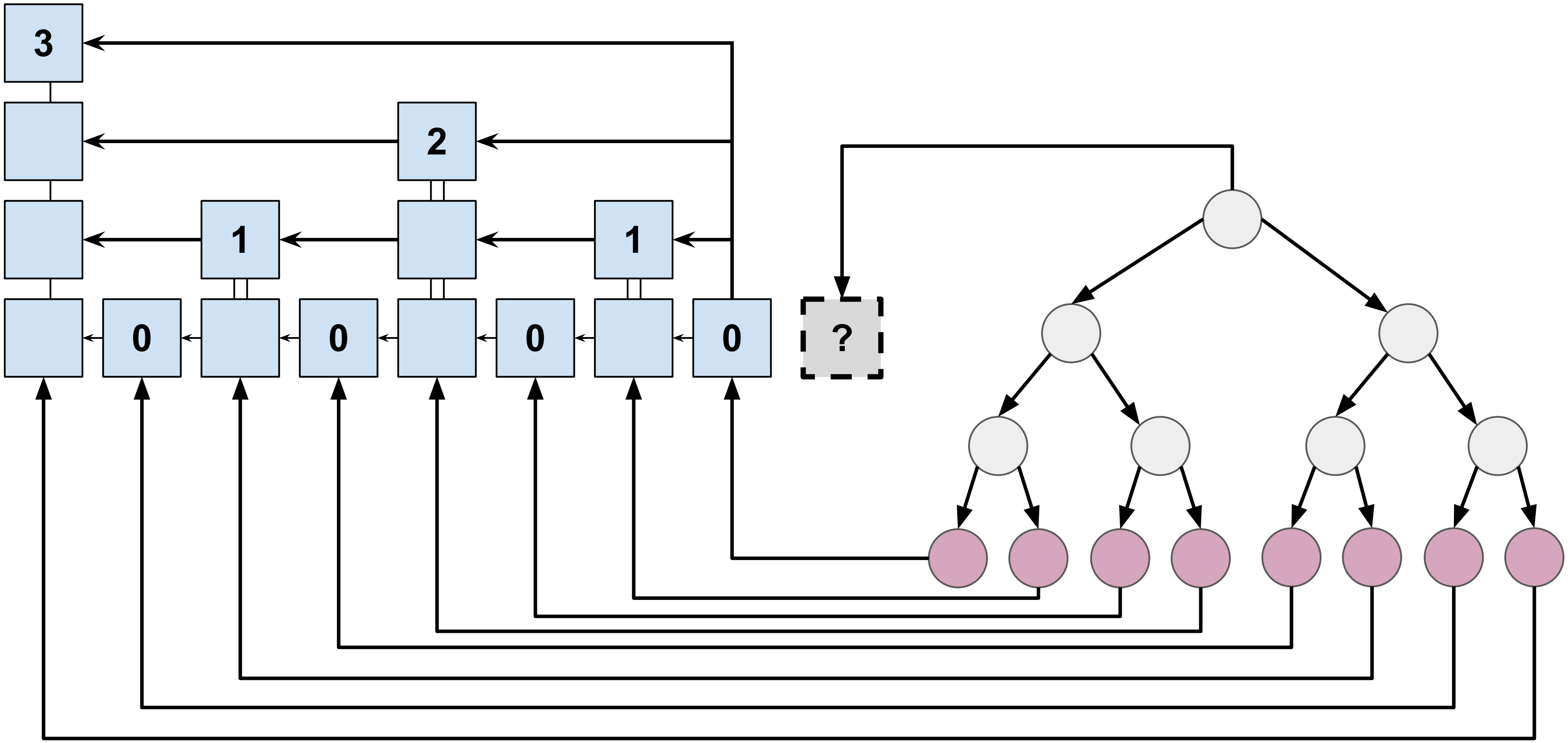
Fig. 1. The probabilistic hierarchical blockchain. Higher levels have achieved a higher difficulty during mining. All blocks are connected to the genesis block G.



Is the chain interlinked in practice?

- Yes! Ethereum has approved EIP-210 for adoption
- Will be implemented soon
- EIP-210 commits to block headers of *all* past blocks in an MMR in every block

"... it allows blocks to directly point to blocks far behind them, which enables extremely efficient and secure light client protocols" –Vitalik Buterin



3
2
1
1
0
0
0
0
?

Provable predicates

- We are creating a proof π that *the last k blocks are* $\chi = B_{-k}, B_{-k+1}, \ldots, B_{-1}$

Compressing state

- Proof π is a *bag of blocks*, as subset of blocks from C

  $\pi \subseteq C$

- Take blockchain C and *diffuse* it into levels $\mu = 1 \ldots \log(|C|)$
- For every level μ with more than 2m blocks, create the diffusion $D[\mu]$
- $D[\mu]$ for every level contains **most recent 2m blocks**
- $D[\mu]$ contains blocks to cover **m most recent blocks of $\mu + 1$ level**
- π is the **union** of all $D[\mu]$

Algorithm 4 Chain compression algorithm for transitioning a full miner to a logspace miner. Given a full chain, it compresses it into logspace state.

function $\mathsf{Dissolve}_{m,k}(\mathcal{C})$
    $\mathcal{C}^* \leftarrow \mathcal{C}[: -k]$
    $\mathcal{D} \leftarrow \emptyset$
    if $|\mathcal{C}^*| \geq 2m$ **then**
        $\ell \leftarrow \max\{\mu : |\mathcal{C}^*{\uparrow}^{\mu}| \geq 2m\}$
        $\mathcal{D}[\ell] \leftarrow \mathcal{C}^*{\uparrow}^{\ell}$
        for $\mu \leftarrow \ell - 1$ down to 0 **do**
            $\mathcal{D}[\mu] \leftarrow \mathcal{C}^*{\uparrow}^{\mu}[-2m:] \cup \mathcal{C}^*{\uparrow}^{\mu}\{\mathcal{C}^*{\uparrow}^{\mu+1}[-m]:\}$
        end for
    else
        $\mathcal{D}[0] \leftarrow \mathcal{C}^*$
    end if
    $\chi \leftarrow \mathcal{C}[-k:]$
    return $(\mathcal{D}, \ell, \chi)$
end function
function $\mathsf{Compress}_{m,k}(\mathcal{C})$
    $(\mathcal{D}, \ell, \chi) \leftarrow \mathsf{Dissolve}_{m,k}(\mathcal{C})$
    $\pi \leftarrow \bigcup_{\mu=0}^{\ell} \mathcal{D}[\mu]$
    return $\pi\chi$
end function

Comparing state

- Receive two proofs π_1, π_2
- We want to find which one is the best

Algorithm 5 The state comparison algorithm.

function $\mathsf{maxvalid}_{m,k}(\Pi, \Pi')$
    if Π' is not a chain $\vee\ |\Pi'| = 0 \vee \Pi'[0] \neq \mathcal{G}$ **then**
        return Π
    end if
    $(\chi, \ell, \mathcal{D}) \leftarrow \mathsf{Dissolve}_{m,k}(\Pi)$
    $(\chi', \ell', \mathcal{D}') \leftarrow \mathsf{Dissolve}_{m,k}(\Pi')$
    $M \leftarrow \{\mu \in \mathbb{N} : \mathcal{D}[\mu] \cap \mathcal{D}'[\mu] \neq \emptyset\}$
    if $M = \emptyset$ **then**
        if $\ell' > \ell$ **then**
            return Π'
        end if
        return Π
    end if
    $\mu \leftarrow \min M$
    $b \leftarrow (\mathcal{D}[\mu] \cap \mathcal{D}'[\mu])[-1]$
    if $|\mathcal{D}'[\mu]\{b{:}\}| > |\mathcal{D}[\mu]\{b:\}|$ **then**
        return Π'
    end if
    return Π
end function

Succinctness

How big is $|\pi|$?

- $|D| \in \Theta(polylog(|C|))$
- $\forall \mu: |D[\mu]| \in \Theta(m) = const$

$|\pi| = \Sigma|D[\mu]|$

NIPoPoWs are succinct: $|\pi| \in \Theta(polylog(|C|))$

We have reduced the storage space required by light nodes ($|\pi|$) compared to legacy nodes ($|C|$) exponentially

Shortcomings of the logspace scheme

- Contrary to the full protocol, it cannot withstand temporary dishonest majority
- To understand why, we must redefine persistence as *computational*

Algorithm 1 The challenger for the ledger persistence game.

function $\text{PERSISTENCE-GAME}_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{Z},\mathcal{A}^*,\Pi}(\kappa)$
  $v \leftarrow \text{VIEW}^{t,n}_{\Pi,\mathcal{A}_1,\mathcal{Z}}$
  $r_1, r_2, p_1, p_2 \leftarrow \mathcal{A}_2(v)$
  if $r_2 < r_1 \lor$ (p_1, p_2 are not honest parties in v) **then**
    return false
  end if
  $\overline{tx} \leftarrow \mathcal{S}(v, r_1, r_2, p_1, p_2)$
  $\mathsf{L}_{p_1}, \mathsf{L}_{p_2} \leftarrow$ ledgers of p_1, p_2 in v
  return $(\delta^*(\mathsf{L}_{p_1}[r_1], \overline{\mathsf{tx}}) \neq \mathsf{L}_{p_2}[r_2])$
end function

Definition 1 (Computational persistence). *A protocol Π has* computational persistence *if there is a negligible function $\mathsf{negl}(\kappa)$ such that for all probabilistic polynomial-time adversaries $(\mathcal{A}_1, \mathcal{A}_2)$ and all environments $\mathcal{Z}$ there exists a probabilistic polynomial-time simulator $\mathcal{A}^*$ such that*

$$\Pr[\text{PERSISTENCE-GAME}_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{Z},\mathcal{A}^*,\Pi}(\kappa)] \leq \mathsf{negl}(\kappa)\,.$$

Thanks! Questions?

National and Kapodistrian
UNIVERSITY OF ATHENS