Non-Interactive Proofs of Proof-of-Work

and sidechain applications

Dionysis Zindros

in collaboration with

Peter Gaži, Kostis Karantias, Aggelos Kiayias, Andrew Miller

National Technical University of Athens, CoreLab Study Group, November 2018

the problem

The proof-of-work equation

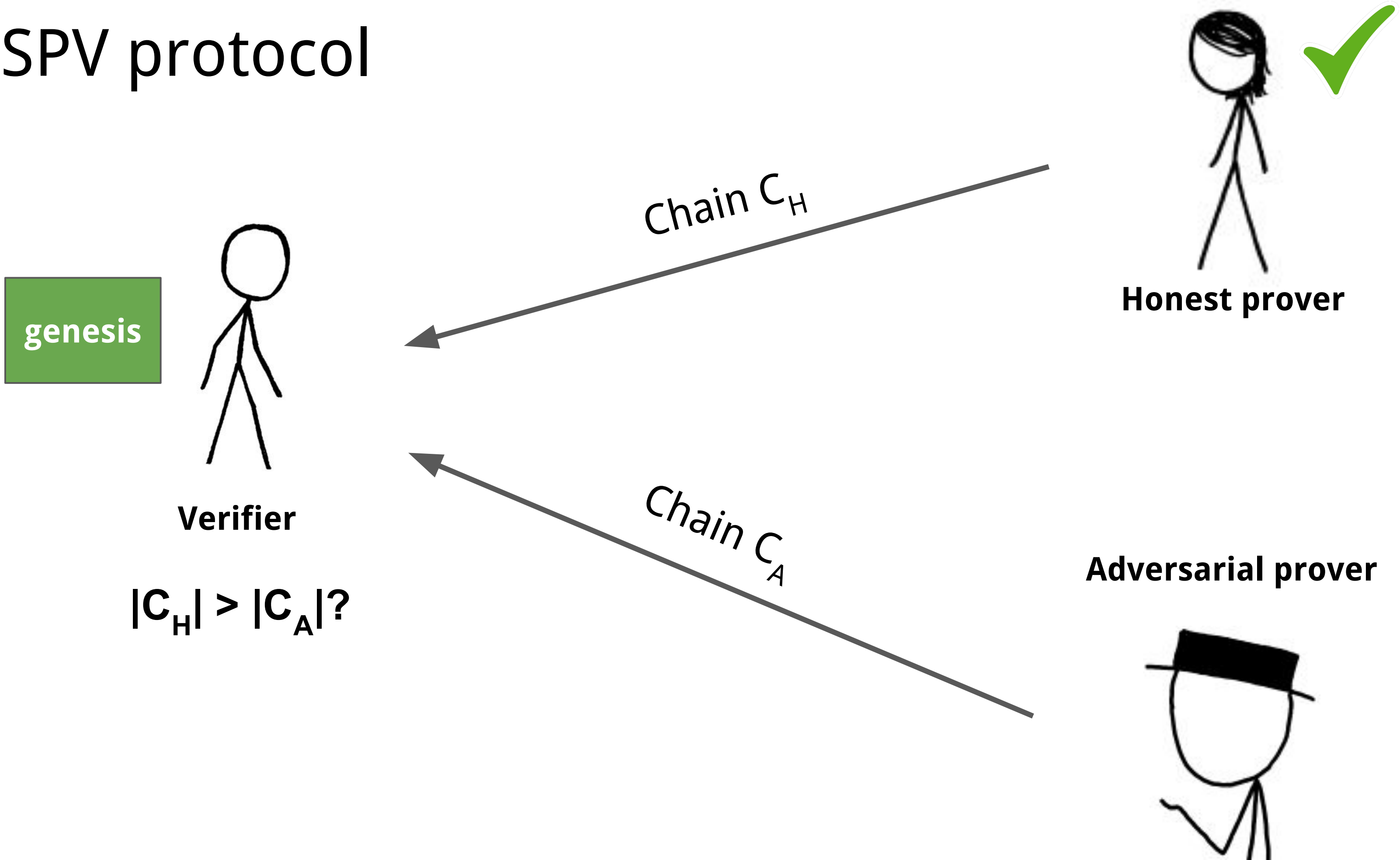
$$H(\text{block}) \le T$$

hash function
(random oracle)

block meta data,
transactions, nonce

proof-of-work
“target”

SPV protocol



The SPV problem

- SPV clients download the **whole** blockchain headers
  - Bitcoin: header = 80 bytes $|C| = 503{,}222$
  - Ethereum: header > 512 bytes $|C| = 4{,}876{,}127$
- Communication: $\Theta(|C|)$
- Can verifier **find latest k = 6 blocks** in $o(|C|)$?
  - **Yes!**

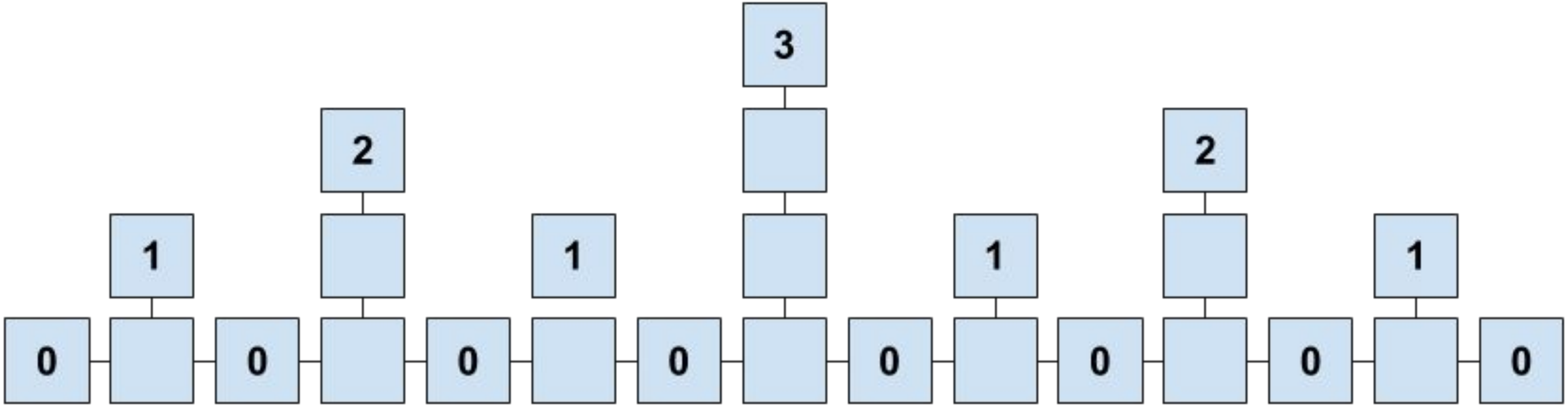
Superblocks

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

$$\Pr[\underbrace{H(\text{block}) \leq \overbrace{T / 2^{\mu}}^{\mu\text{-supertarget}}}_{\text{The } \mu\text{-superblock condition}} \mid H(\text{block}) \leq T] = 2^{-\mu}$$

- 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

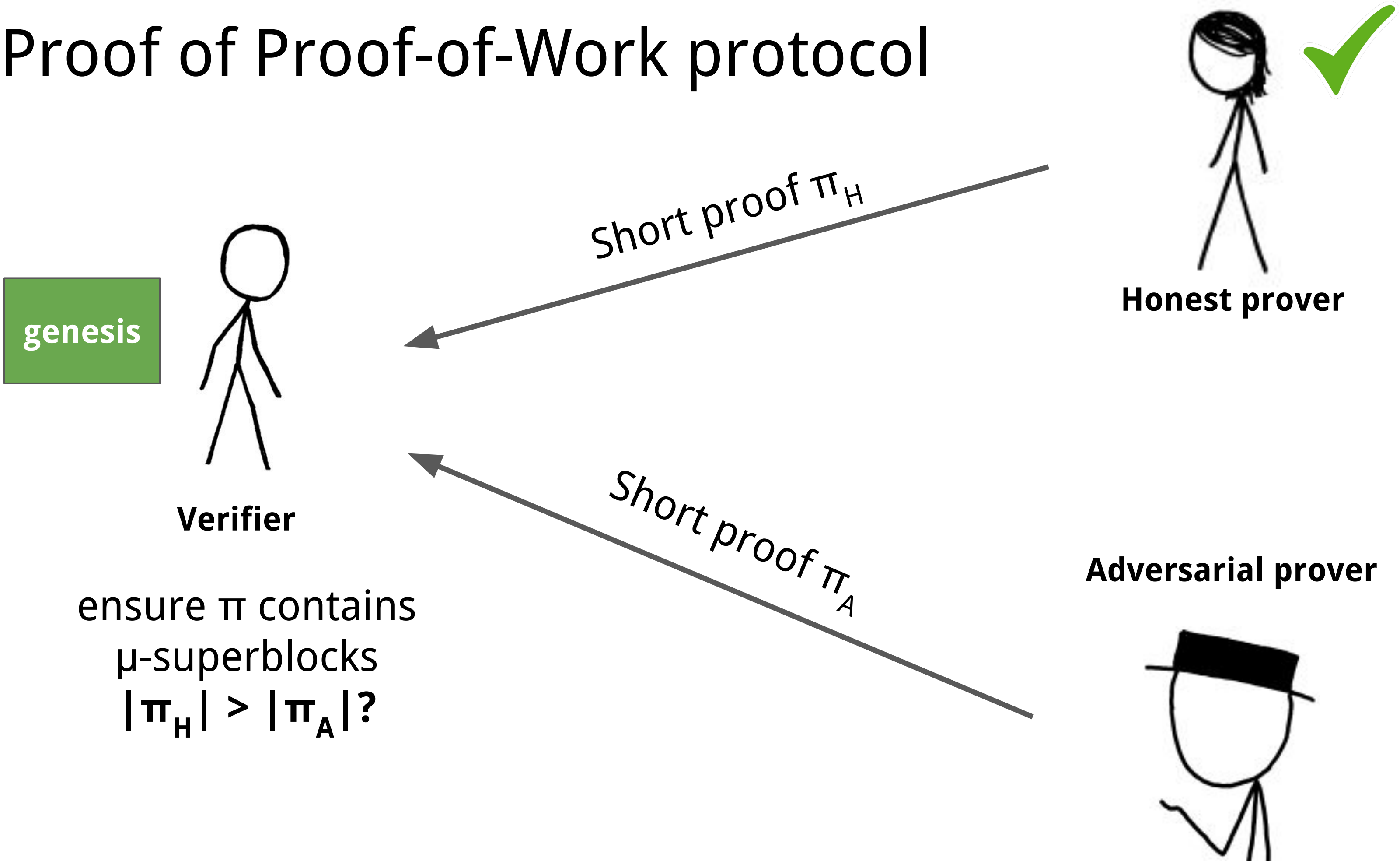
Idea: Proofs of proof-of-work

- Instead of sending the whole block headers...
- ...send a **representative sample π** of at least **m = 128** blocks from **prefix**
- ...plus, **the suffix χ** of the last k = 6 blocks

Number of superblocks in |C| = 4,194,304 (Ethereum):

Number of 0-superblocks: 4,194,304
Number of 1-superblocks: ~ 2,097,152
Number of **μ**-superblocks: ~ 4,194,304 $2^{-\mu}$
…
Number of **15-superblocks: ~ 128**
Number of $\log_2(|C|)$-superblocks: ~1

Proof of Proof-of-Work protocol



ensure π contains
μ-superblocks

$|\pi_H| > |\pi_A|$?

The Prover/Verifier model

- Similar but different to zero-knowledge proofs
- 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
- Verifier wakes up stateless (has genesis block only)
- Each prover sends a *proof* to the verifier
- Verifier chooses one of the proofs as legitimate
- Verifier decides about a value of a predicate *p* of the honest chain

Provable predicates

- Must be **monotonic**: $Q(C) \rightarrow Q(CB)$
- Must be **stable**: $Q(CC_1) = Q(CC_2)$ for $|C_1| = |C_2| \leq k$

Example predicates:

- Block B has been observed by all honest full nodes
- Block B is reported as stable by all honest full nodes
- Blocks B_1 and B_2 have at least 1024 blocks between them
- Transaction tx sending 5 coins from account α to account β is reported stable
- Account α has had exactly 9 coins at block 4,102,997
- Address β is instantiated and contains the CryptoKitties smart contract
- Coin Q was mined and then transferred to address α_1, then α_2, then α_3

Succinctness

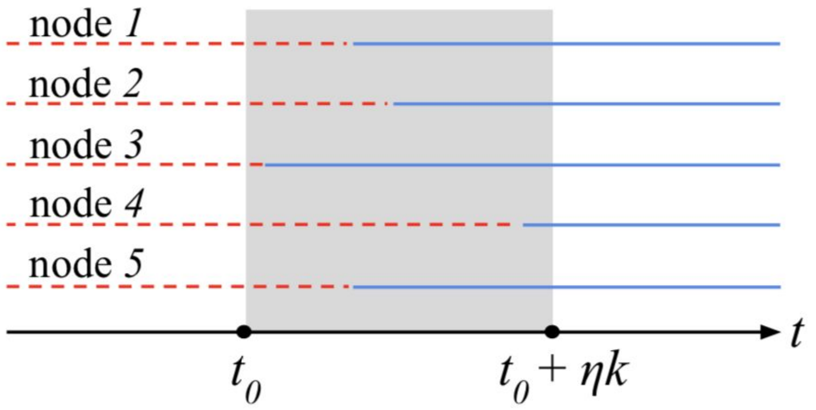
Honest optimistic
All honest provers generate proofs π s.t. $|\pi| \in O(\text{polylog}(|C|))$
(in an optimistic execution)

Adversarial optimistic
The honest verifier reads only $O(\text{polylog}(|C|))$ from each proof
(in an optimistic execution)

The chain-based and round-based formulations are equivalent:
At round r, $|C| \in \Theta(r)$

Definition 3. *(Security) A* blockchain proof protocol (P, V) *about a predicate* Q *is* secure *if for all environments and for all PPT adversaries* $\mathcal{A}$ *and for all rounds* $r \geq \eta k$*, if* V *receives a set of proofs* $\mathcal{P}$ *at the beginning of round* r*, at least one of which has been generated by the honest prover* P*, then the output of* V *at the end of round* r *has the following constraints:*

- *If the output of* V *is* false, *then the evaluation of* $Q(\mathcal{C})$ *for* all *honest parties must be* false *at the end of round* $r - \eta k$.
- *If the output of* V *is* true, *then the evaluation of* $Q(\mathcal{C})$ *for* all *honest parties must be* true *at the end of round* $r + \eta k$.

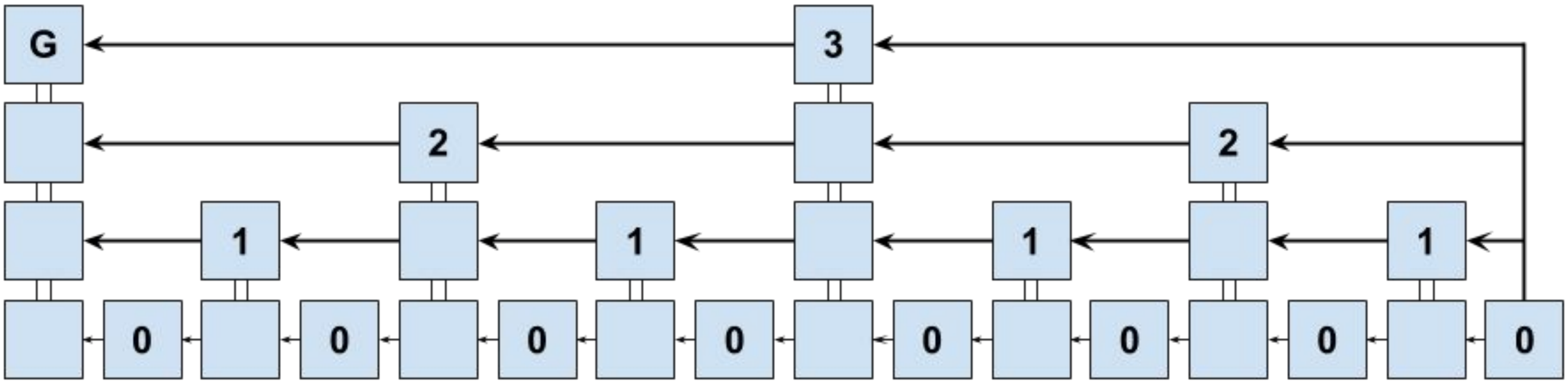


suffix proofs

Interlinking the chain

- Blocks still need to be in order
- Must **link** each μ-superblock to previous μ-superblock
- Link pointer must be included in PoW
- We can’t know if block will become μ-superblock when mining!
- When mining new block, include:
  for all μ: a pointer to **most recent μ-superblock**

The interlinked chain



$$\mathrm{level}(B) = \lfloor \log(T\ ) - \log(\mathrm{id}(B)) \rfloor$$

(block level)

Algorithm 1 updateInterlink

function updateInterlink(B')
    interlink $\leftarrow$ B'.interlink
    for $\mu = 0$ to *level*(B') **do**
        interlink[μ] $\leftarrow$ id(B')
    end for
    return interlink
end function

Notational conventions: Blockchain addressing

- |C| the number of *blocks* in C (not transactions)
- C[i], the i^{th} element of C
- C[0], the first block (if *genesis*, the chain is *anchored*)
- C[-i], the i^{th} element from the end
- C[-1], the tip
- C[-k] is still unstable
- C[i:j] subchain from i^{th} block (inclusive) to j^{th} block (exclusive)
    - i, j can be negative
- C[i:] subchain from i^{th} block to end
- C[:j] subchain from beginning to j^{th} block
- C{a:z} subchain from block a to block z (one of them can be omitted)
- C[:-k] stable chain

A *sequence* of *blocks C* is a *chain* if for all *i*

C[i] includes a pointer to C[i - 1]

(the chain property)

$$C_1[:|C_2[:-k]|] = C_2[:-k]$$

(the Common Prefix property)

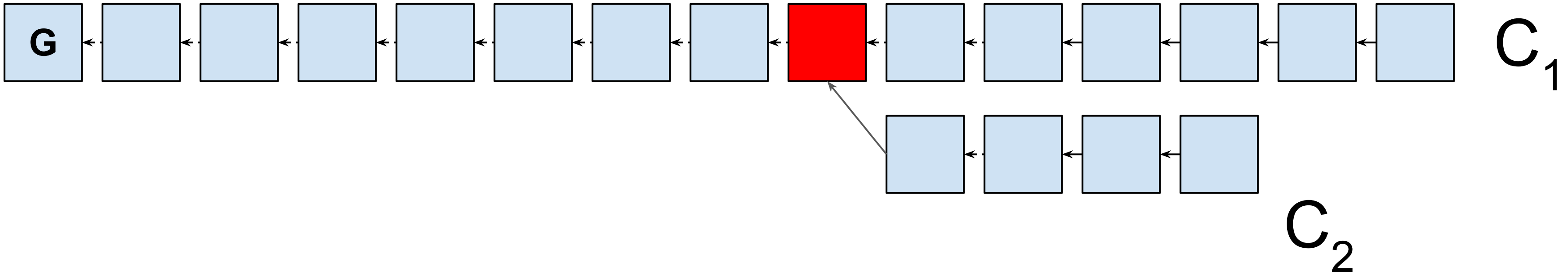
Notational conventions: Blockchain traversing

Given chains C_1, C_2:

- $C_1 C_2$: the concat chain
- $\{B \in C: p(B)\}$: chain with all blocks satisfying predicate *p*
  - $C_1 \subseteq C_2$ means there exists some predicate p such that $C_1 = \{ B \in C_2: p(B) \}$

- $C_1 \cup C_2$: topologically order blocks in C_1 and C_2 to create a chain
- $C_1 \cap C_2$: $\{B \in C_1: B \in C_2\}$
- $C\uparrow^{\mu} = \{B \in C: \text{level}(C) \geq \mu \}$
- $C\uparrow^{\mu}\downarrow = C$
- $B \in C$ means block B is in C ($\exists\ C_1 C_2 : C = C_1 B C_2$)
- These are not sets but sequences: Order is important (and unique -- why?)
- In all cases, the chain property must be maintained! (Check is needed)

$(C_1 \cap C_2)[-1]$



The generic verifier

- Receive several *candidate* proofs $\mathcal{P}$
- Each proof $\pi\chi$ in $\mathcal{P}$ contains a prefix proof π and a suffix χ
- $\pi\chi$ must be a chain
- $|\chi| = k$
- π contains a sample of blocks (but forms a chain)
- For honest prover with C chain adopted: $\pi\chi \subseteq C$
- Verifier chooses the “best” proof $\pi\chi$
- Outputs the claimed value of predicate Q

Suffix proof protocols

We start by proving predicates that are *suffix sensitive*, i.e., only depend on the last k + 1 of the chain. $\tilde{Q}(\chi) = Q(C)$ for $C = \pi\chi$ where $\tilde{Q}$ is efficiently computable

suffix sensitivity + *stability*: Predicate depends exactly on C[-k].

Example:

- Ethereum account α contains 5 ETH at block height 4,779,196
- Possible because ETH commits to state Merkle Tree in every block

We will generalize these later

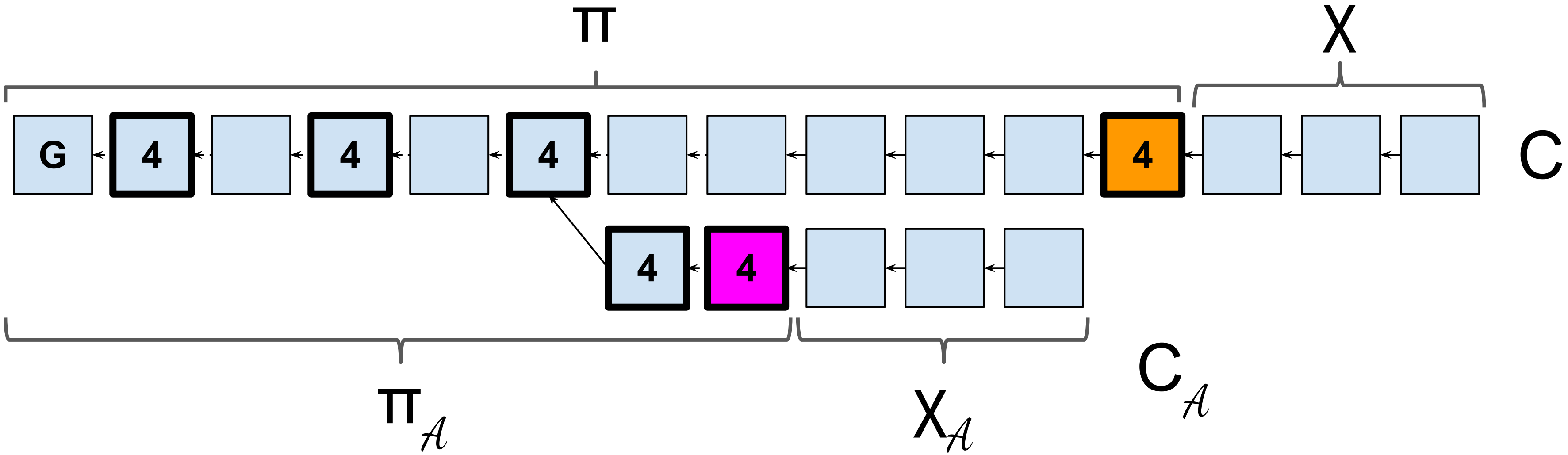
Algorithm 2 The Verify algorithm for the NIPoPoW protocol

function $\mathsf{Verify}^{Q}_{m,k}(\mathcal{P})$
    $\tilde{\pi} \leftarrow (\text{Gen})$ ▷ Trivial anchored blockchain
    for $(\pi, \chi) \in \mathcal{P}$ **do** ▷ Examine each proof (π, χ) in $\mathcal{P}$
        if $\mathsf{validChain}(\pi\chi) \wedge |\chi| = k \wedge \pi \geq_m \tilde{\pi}$ **then**
            $\tilde{\pi} \leftarrow \pi$
            $\tilde{\chi} \leftarrow \chi$ ▷ Update current best
        end if
    end for
    return $\tilde{Q}(\tilde{\chi})$
end function

First idea: Just include one level

- Define a security parameter m pertaining to the prefix π
- Find highest level containing at least m blocks
- That's π
- Succinctness is easy: $|\pi| = m \in \Theta(1)$
- Suppose both provers prove at the same level
- Verifier compares count
- Security?

One-level proof attack for m = 4, k = 3



k-Common-Prefix not violated, but bad proof wins

...we need something more clever

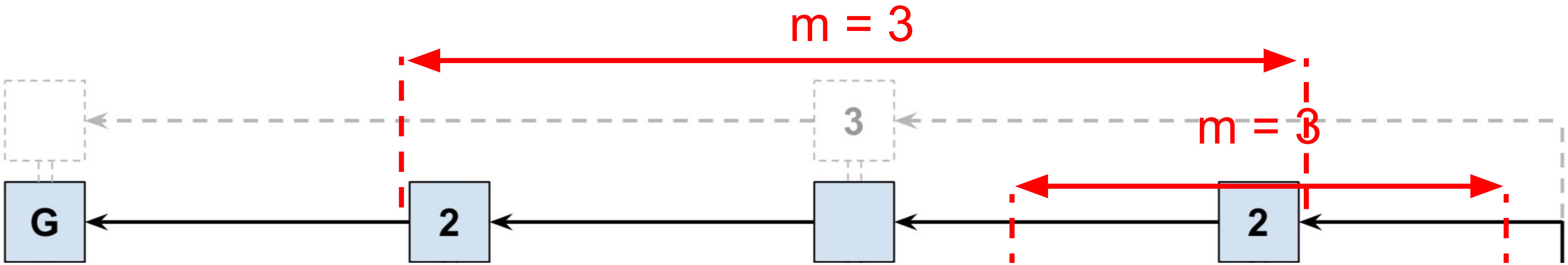
Non-interactive Proofs of Proof-of-Work

We want honest prover to just **publish a proof** and **go offline**

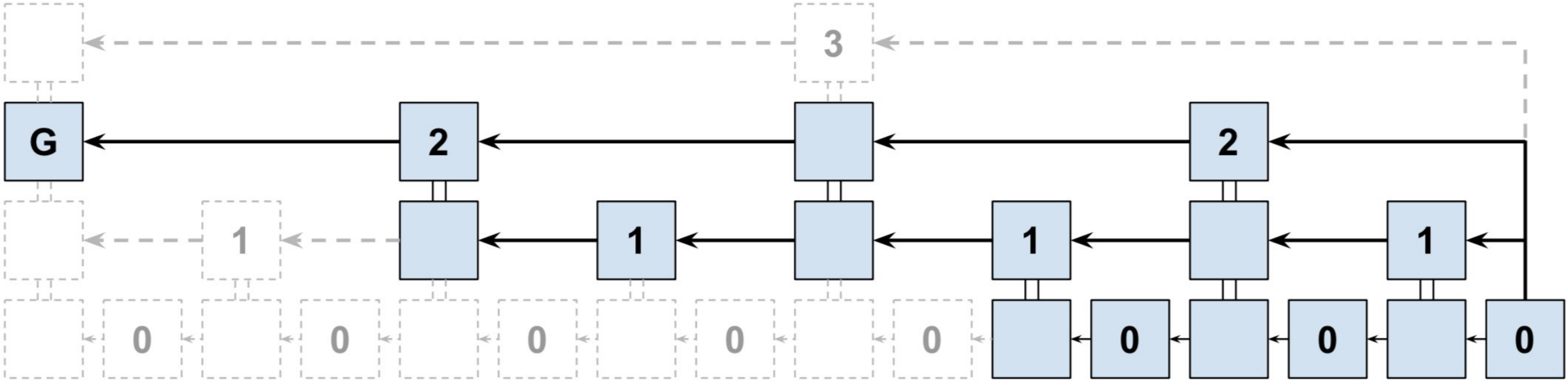
Prover construction:

- Split blockchain into *stable* C[:-k] and *unstable* C[-k:] portion
- Include the whole *unstable* portion χ = C[-k:] (with $|\chi| = k$)
- For the stable portion, create a *prefix proof* π
- Find the top-most level μ with at least m blocks
- For every next level:
- Include enough blocks from level i - 1 to “cover” the **m**-suffix of level i
- Final proof is $\pi\chi$

A NIPoPoW prefix π for m = 3



A NIPoPoW prefix π for m = 3



Algorithm 3 The Prove algorithm for the NIPoPoW protocol

function $\mathsf{Prove}_{m,k,\delta}(\mathcal{C})$
    $B \leftarrow \mathcal{C}[0]$ $\triangleright$ Genesis
    for $\mu = |\mathcal{C}[-k-1].\mathsf{interlink}|$ down to 0 **do**
        $\alpha \leftarrow \mathcal{C}[:-k]\{B:\}\uparrow^{\mu}$
        $\pi \leftarrow \pi \cup \alpha$
        if $m < |\alpha|$ **then**
            $B \leftarrow \alpha[-m]$
        end if
    end for
    $\chi \leftarrow \mathcal{C}[-k:]$
    return $\pi\chi$
end function

Succinctness of NIPoPoWs

- Number of μ-levels is log(|C|)
- Top level has approx between m and 2m blocks
  - By definition, must have at least m
  - If it had much more than 2m, there would be one more level on top with more than m
- Number of μ - 1 blocks needed to cover **m** μ-superblocks: ~2m
  - There's 2 blocks of level μ - 1 for every block of level μ
- Total proof size: Θ(log(|C|)m + k) blocks

How to compare NIPoPoWs?

- To compare two proofs π_1 π_2 compute $b = (\pi_1 \cap \pi_2)[-1]$
- For each prefix π of a suffix proof $\pi\chi$, compute a *score*
- For each μ, obtain *argument* $\pi\{b:\}\uparrow^\mu$ as long as $|\pi\{b:\}\uparrow^\mu| \geq m$
- Compute a *score* for the argument: $2^\mu|\pi\{b:\}\uparrow^\mu|$
- Best score among all arguments is score of proof
- To compare two proofs, compare their scores

function $\textsf{best-arg}_m(\pi, b)$

  $M \leftarrow \{\mu : |\pi{\uparrow}^{\mu}\{b:\}| \geq m\} \cup \{0\}$

  return $\max_{\mu \in M}\{2^{\mu} \cdot |\pi{\uparrow}^{\mu}\{b:\}|\}$

end function

operator $\pi_A \geq_m \pi_B$

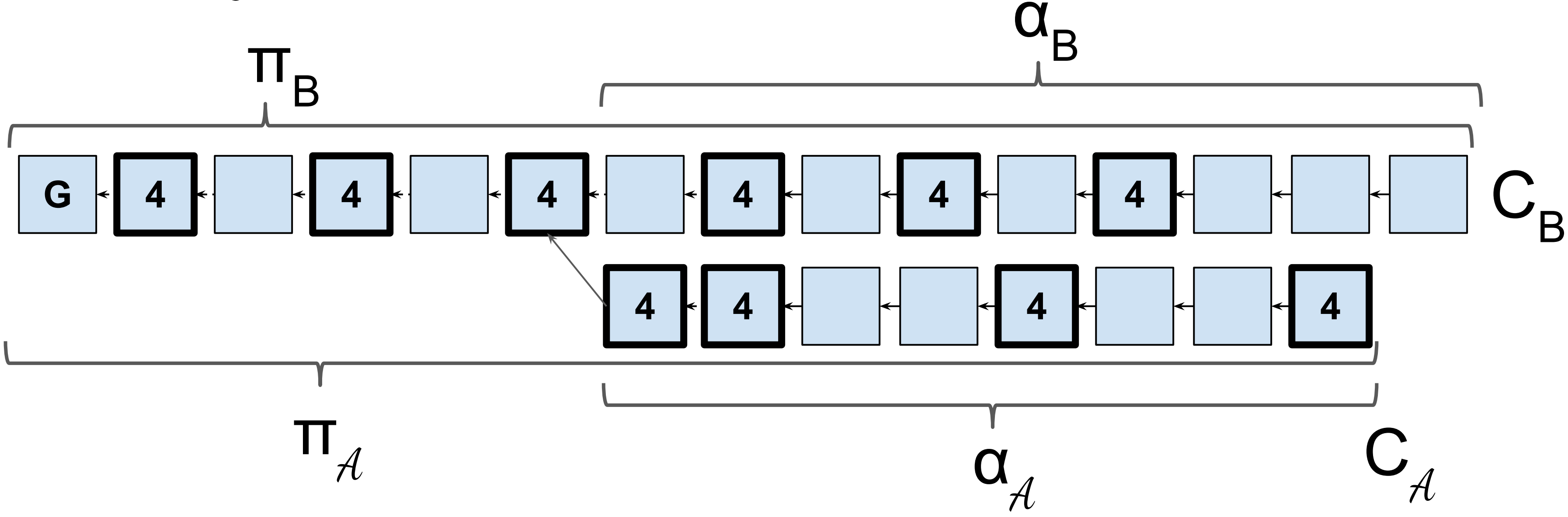
  $b \leftarrow (\pi_{\mathcal{A}} \cap \pi_B)[-1]$

  return $\textsf{best-arg}_m(\pi_A, b) \geq \textsf{best-arg}_m(\pi_B, b)$

end operator

security

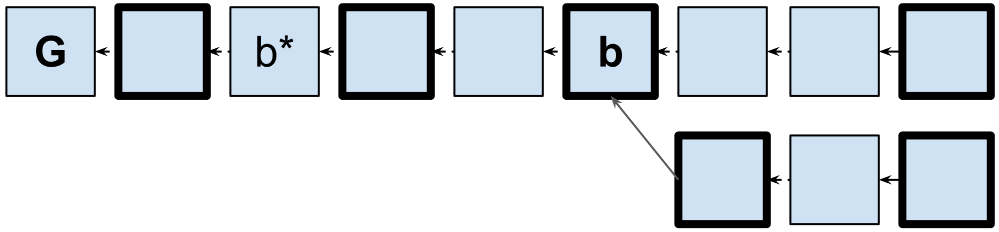
Security intuition, m = 3



as $|\alpha_B| \geq m$, we have $E[|\alpha_{\mathcal{A}}|] < E[|\alpha_B|]$ and so $\Pr[|\alpha_{\mathcal{A}}| \geq |\alpha_B|] = \text{negl}(m)$

Security proof

- By contradiction, suppose adversary is winning
- Let honest proof π_B be generated at round r_3
- Let b be LCA
- Let b* be most recent honestly generated block before b (b* will exist)
- Let $m = k_1 + k_2 + k_3$



Claim: Highest argument contains all blocks

For LCA b and honest comparison level μ_B, we have $\alpha_B = C_B\{b:\}\uparrow^{\mu B}$

- By induction on μ from top to bottom: B always precedes b
- Highest μ contains genesis
- Level μ - 1 spans all m blocks of level μ
- But condition for α_B is $|\alpha_B| \geq m$
- If there were a block in $C_B\{b:\}\uparrow^{\mu B}$ not in α_B, the higher level would have won

Claim: Goodness

- Consider a chain C
- Consider $C_1 = C\uparrow^{\mu1}$, $C_2 = C\uparrow^{\mu2}$ with $\mu_1 < \mu_2$
- If $|C_1| > k_1$, then $|C_2| > (1 - \delta_1)(1 - \delta_2)2^{\mu1 - \mu2}|C_1|$
- By a negative binomial Chernoff bound on C_1
  - Number of rounds $|S|$ needed to generate C_1: $|S| > (1 - \delta_1)|C_1|2^{\mu1}/(pqn)$
- And a binomial Chernoff bound on C_2
  - Number of blocks in C_2: $|C_2| > (1 - \delta_2)2^{-\mu2}|S|pqn = (1 - \delta_2)(1 - \delta_1)2^{\mu1 - \mu2}|C_1|$

Algorithm 3 The Prove algorithm for the NIPoPoW protocol

function $\mathsf{Prove}_{m,k,\delta}(\mathcal{C})$
    $B \leftarrow \mathcal{C}[0]$ ▷ Genesis
    for $\mu = |\mathcal{C}[-k-1].\mathsf{interlink}|$ down to 0 **do**
        $\alpha \leftarrow \mathcal{C}[:-k]\{B:\}{\uparrow^{\mu}}$
        $\pi \leftarrow \pi \cup \alpha$
        if $m < |\alpha|$ **then**
            $B \leftarrow \alpha[-m]$
        end if
    end for
    $\chi \leftarrow \mathcal{C}[-k:]$
    return $\pi\chi$
end function

Claim: $\alpha_{\mathcal{A}}[1:]$ and C{b:}[1:] are mostly disjoint

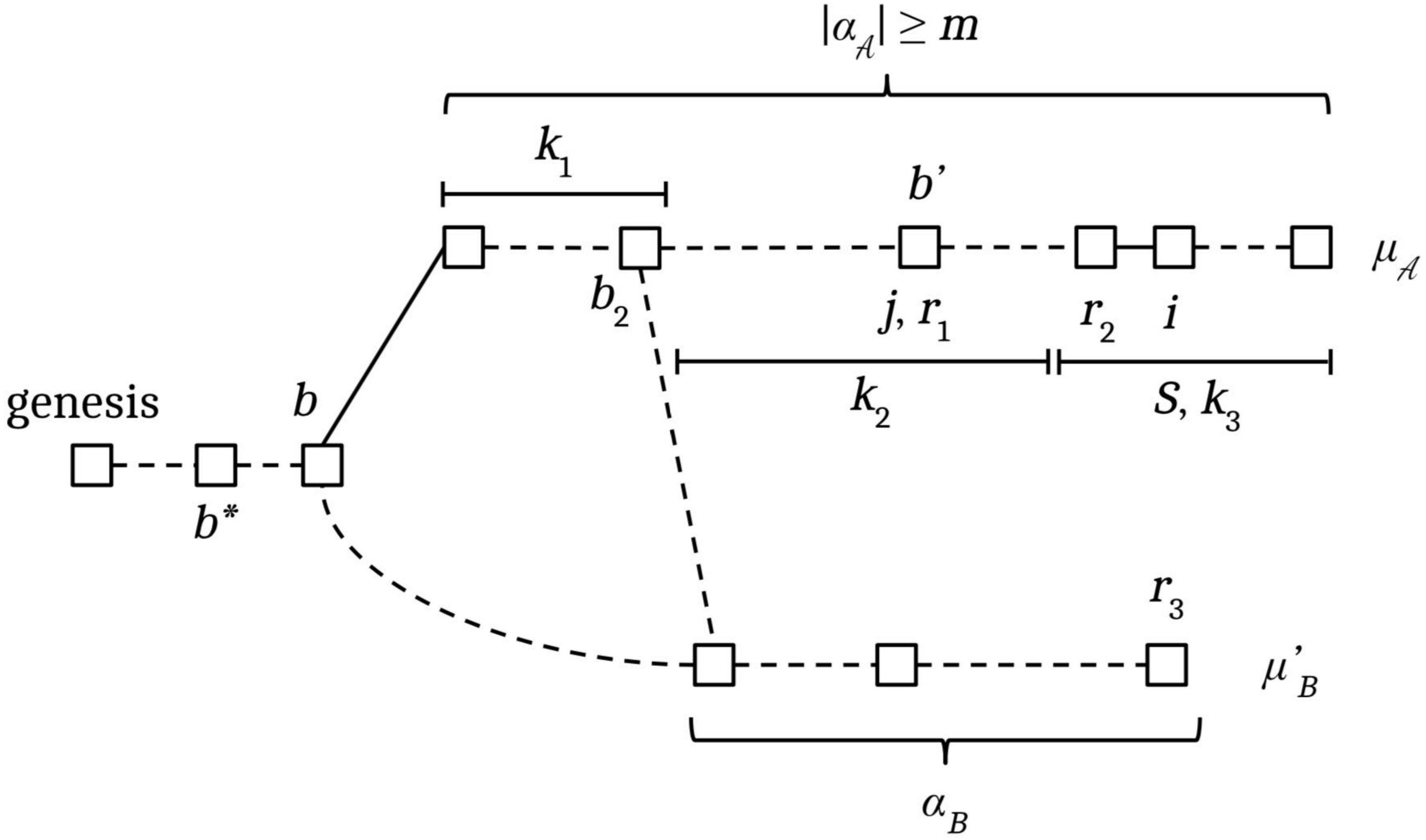
Case $\mu_B \leq \mu_{\mathcal{A}}$:

- $\alpha_{\mathcal{A}}[1:]$ and C{b:}[1:] are completely disjoint

Case $\mu_B >= \mu_{\mathcal{A}}$:

- $|\alpha_{\mathcal{A}}[1:] \cap C\{b:\}[1:]| > k_1$ is impossible
- Otherwise C{b:} goodness is violated wrt level $\mu_{\mathcal{A}}$

Let $b_2 = (C_B \cap \alpha_{\mathcal{A}})[-1]$


$|\alpha_{\mathcal{A}}| \geq m$
k_1
b'
$\mu_{\mathcal{A}}$
b_2
j, r_1
r_2
i
genesis
b
k_2
S, k_3
b^*
r_3
μ'_B
α_B

Claim: The last k_3 blocks of $\alpha_{\mathcal{A}}$ are adversarial

By contradiction

- Suppose $D = \alpha_{\mathcal{A}}[k_1 + k_2 + i]$ was honestly generated at round r_2
- This means that at r_2 , D belongs to some honestly adopted chain $C_{\mathcal{A}}$
- But at round r_3 an honest party has adopted C_B
- $C_{\mathcal{A}}$ has $k_2 + k_3$ blocks not in C_B
- The violates Common Prefix with parameter k_2

It took a long time to mine k_3 adversarial blocks

- Let b' be the most recent honestly generated block of $\alpha_{\mathcal{A}}$ (or b' = b* if no such block)
- Let b' be generated at round r_2
- There are at least k_3 blocks after b*
- Consider the rounds $S = r_2 \ldots r_3$
- Apply a negative binomial Chernoff bound to $|S|$
- $|S| \geq (1 - \varepsilon')2^{\mu\mathcal{A}}|\alpha_{\mathcal{A}}|/(pqt)$

Honest superchain grew a lot during S

- Applying Chain Growth with chain velocity f
- $|C_B\{b^*:\}| \geq (1 - \delta)f|S|$
- By goodness of $C_B\{b^*:\}$ we have $|C_B\{b^*:\}\uparrow^{\mu B}| \geq (1 - \delta)(1 - \varepsilon)2^{-\mu B}f|S|$

Claim: $\alpha_{\mathcal{A}}$ cannot exist

With overwhelming probability in m...

$$|C_B\{b^*:\}{\uparrow}^{\mu B}| \geq (1 - \delta)(1 - \varepsilon)(1 - \varepsilon') \, f \, 2^{\mu\mathcal{A} - \mu B} |\alpha_{\mathcal{A}}|/(pqt)$$

Hence, the adversarial prover could not have won, which is a contradiction.

infix proofs

How to prove predicates deep within the chain?

- So far we can prove *suffix* predicates
- What about more generic predicates?
- Our proofs must be $\Theta(\text{polylog}(|C|))$
- Predicates must depend on $\text{polylog}(|C|)$ number of blocks
- We will extend suffix proofs to include more blocks
- The *set* of blocks the predicate depends on are the *witness blocks*

Definition 6 (Infix sensitivity). *A chain predicate* $Q_{d,k}$ *is* infix sensitive *if it can be written in the form*

witness must be stable

witness must efficiently attest

witness blockset (not blockchain)

$$Q_{d,k}(\mathcal{C}) = \begin{cases} \text{true}, & \textit{if } \exists \mathcal{C}' \subseteq \mathcal{C}[:-k] \wedge |\mathcal{C}'| \le d \wedge D(\mathcal{C}') \\ \text{false}, & \textit{otherwise} \end{cases}$$

where D *is an arbitrary efficiently computable predicate.*

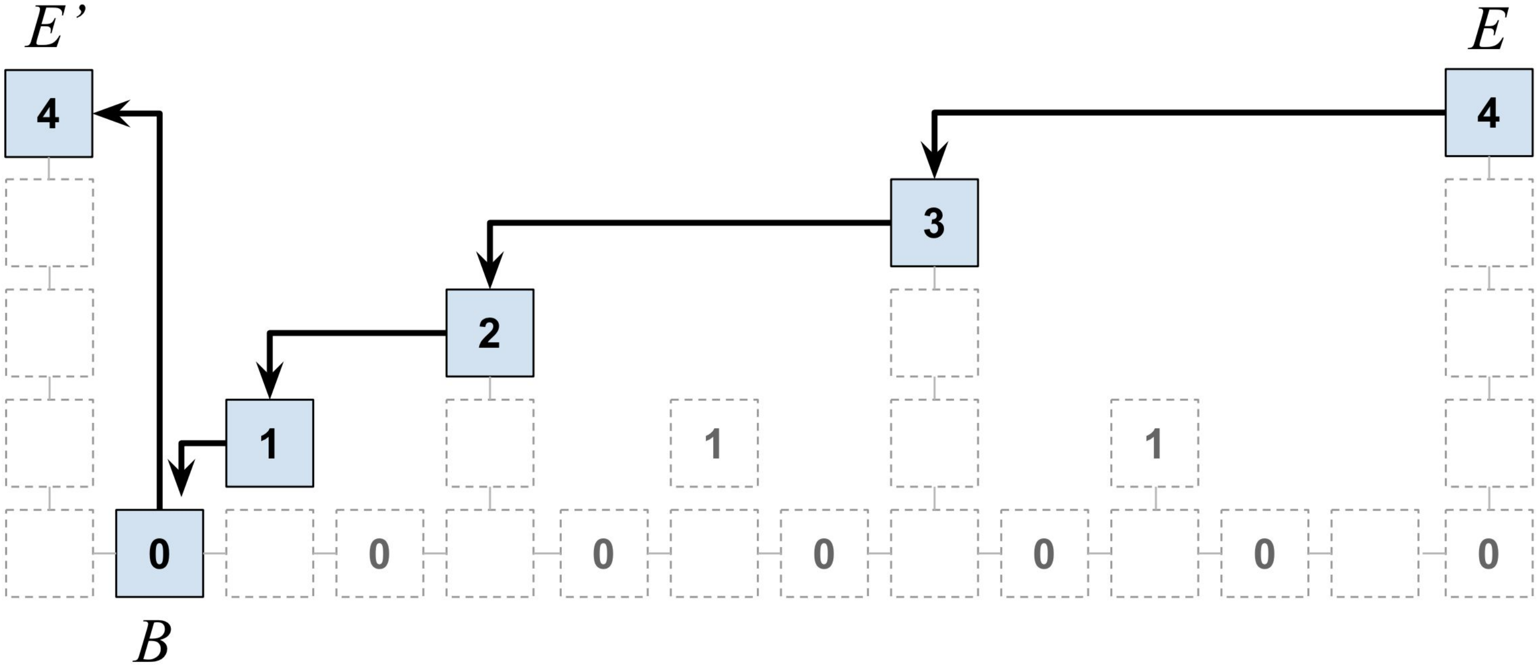
witness must be short
set d = polylog(|C|)

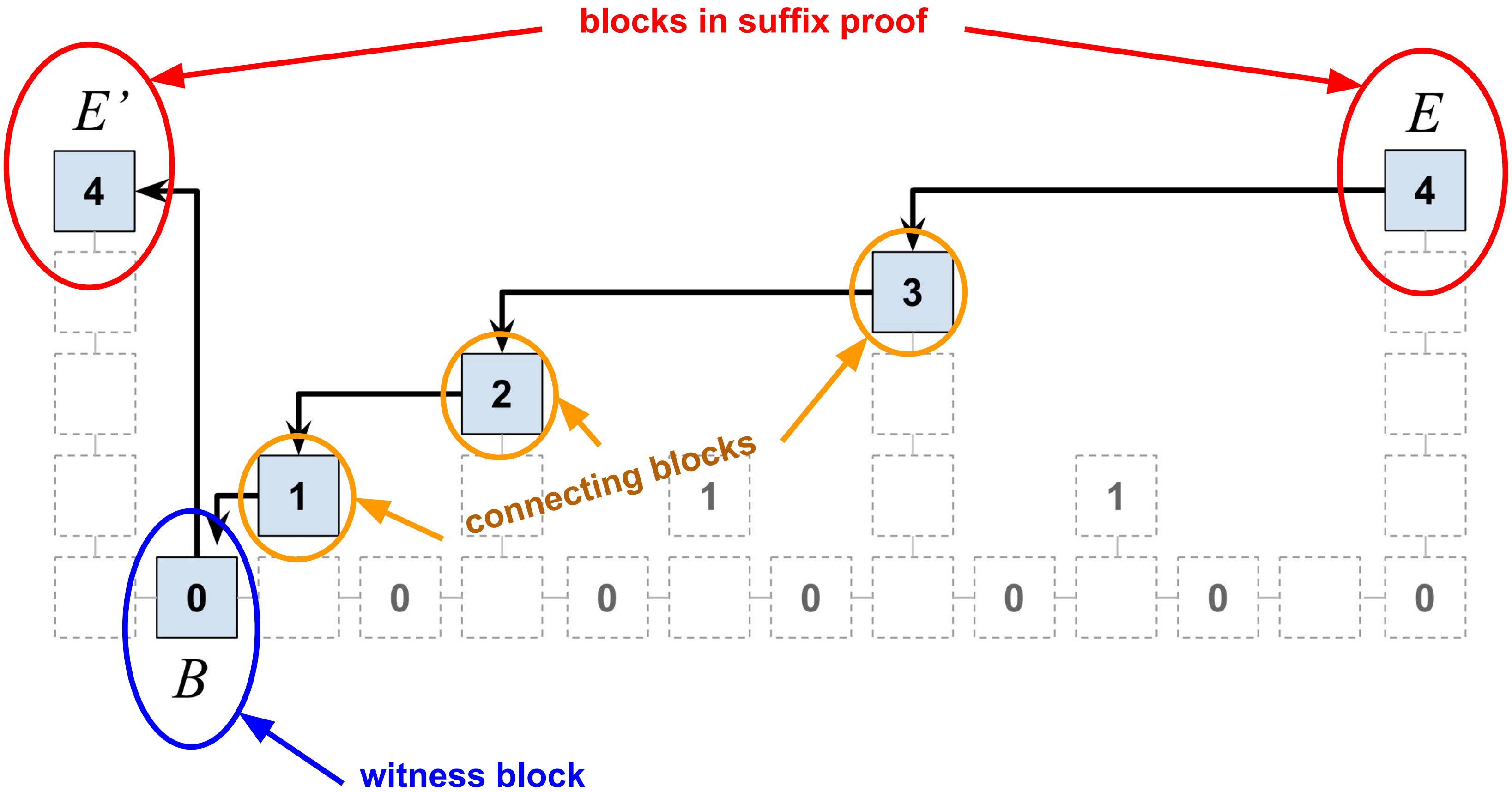
Infix proofs

- NIPoPoW construction allows publishing a **verifiable** string securely showing that: suffix χ is the **k**-long end of the **current** blockchain
- We can also prove that each block **b** in the witness is part of the blockchain

Infix proofs

- Suppose we have proven that blocks at μ = 4 are in chain
- We can prove that a block of μ < 4 is between them



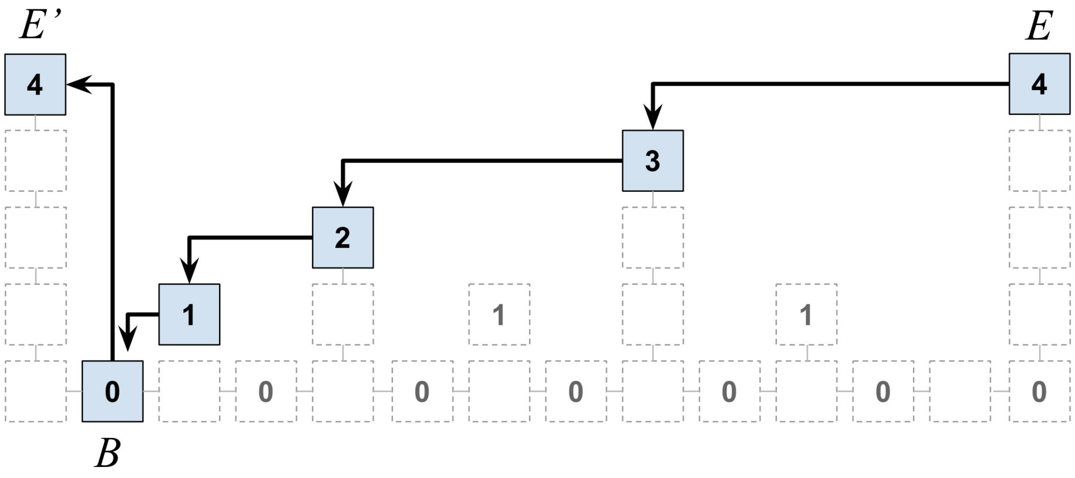

blocks in suffix proof
E'
4
E
4
3
2
1
connecting blocks
1
1
0
0
0
0
0
0
0
0
B
witness block

Algorithm 5 The Prove algorithm for infix proofs

function $\mathsf{ProveInfix}_{m,k}(\mathcal{C}, \mathcal{C}', \mathsf{height})$
    $aux \leftarrow \emptyset$
    $(\pi, \chi) \leftarrow \mathsf{Prove}_{m,k}(\mathcal{C})$ ▷ Start with a suffix proof
    for $B \in \mathcal{C}'$ **do**
        for $E \in \pi$ **do**
            if $\mathsf{height}[E] \geq \mathsf{height}[B]$ **then**
                $aux \leftarrow aux \cup \mathsf{followDown}(E, B, \mathsf{height})$
                break
            end if
        end for
    end for
    return $(aux \cup \pi, \chi)$
end function

Following "down"

- Suppose E, E' are consecutive blocks in π and E'BE ⊆ C
  - lvl(E') ≥ lvl(E)
- C contains only blocks with level < lvl(E') between E' and E
- Follow pointers from highest to lowest until B is reached
  - Sometimes multiple pointers at same level will have to be followed
- B connects back up to E' with *one* pointer
- Collect these blocks into *aux*
- *aux* ∪ *π* forms a chain!

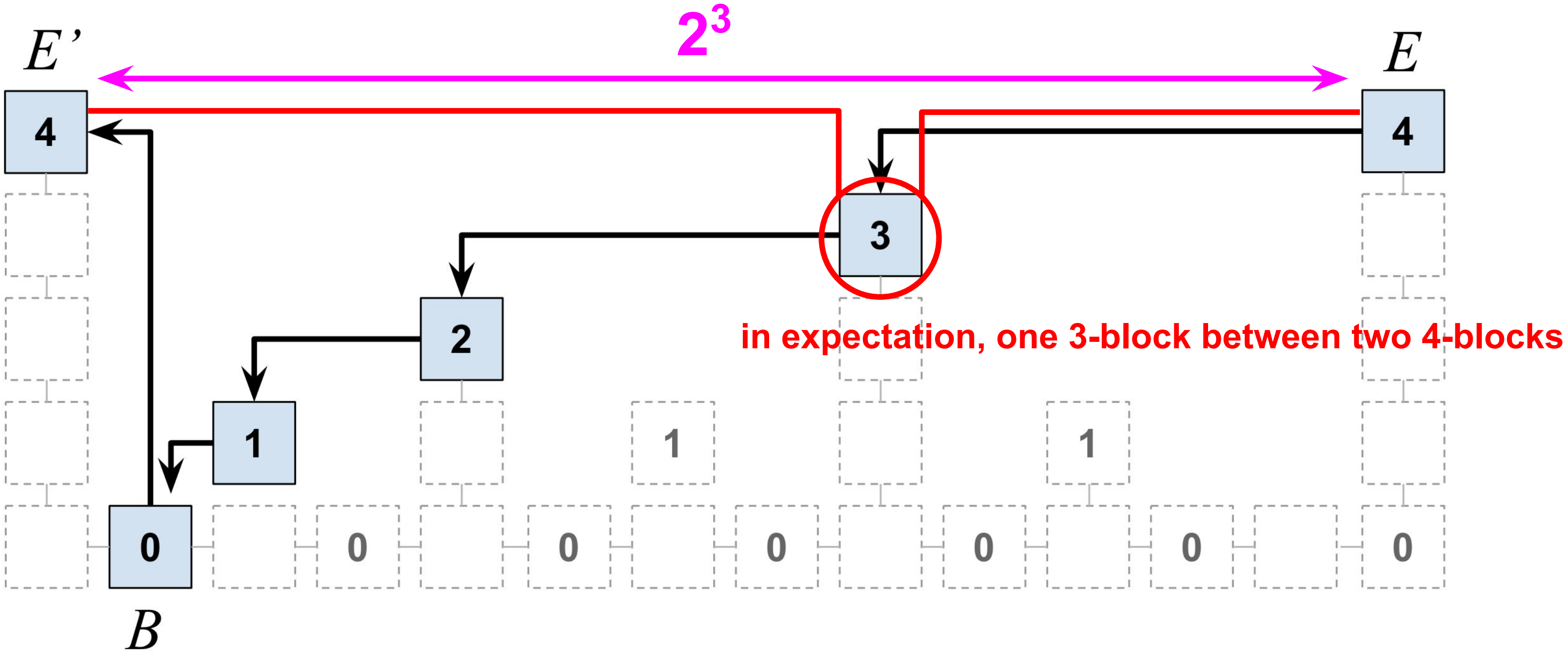


Algorithm 6 The followDown function which produces the necessary blocks to connect a superblock E to a preceeding regular block B.

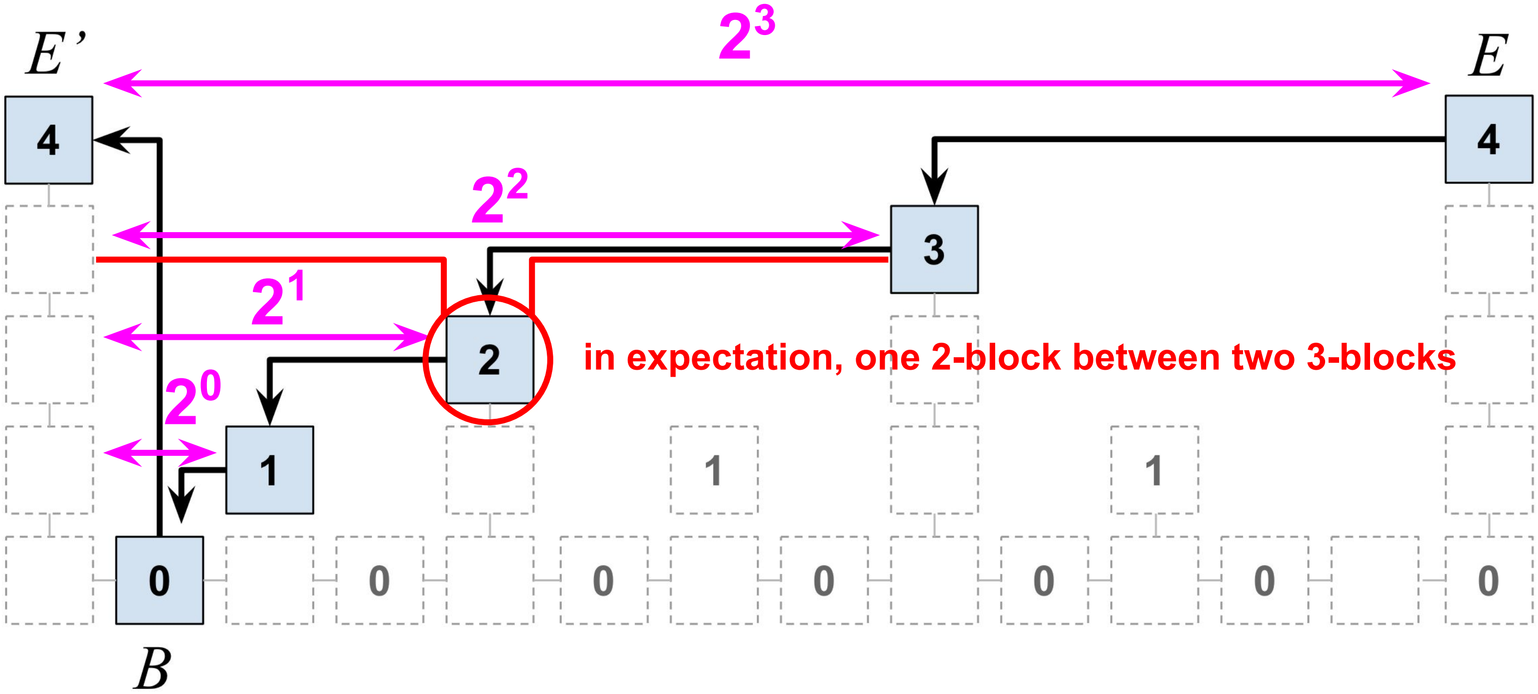
```
function followDown(E, B, height)
    aux ← ∅; μ ← level(E)
    while E ≠ B do
        B′ ← blockById[E.interlink[μ]]
        if height[B′] < height[B] then
            μ ← μ − 1
        else
            aux ← aux ∪ {E}
            E ← B′
        end if
    end while
    return aux
end function
```

Infix succinctness

- Infix size = $|\pi| + |\chi| + |aux|$
- $|\pi| + |\chi| \in \Theta(\mathrm{polylog}(|C|))$
- $|aux| = |C'|\ |\mathrm{followDown}|$
- $|C'| \in \Theta(\mathrm{polylog}(|C|))$
- It remains to show that $|\mathrm{followDown}| \in \Theta(\mathrm{polylog}(|C|))$



2^3
E'
E
4
4
3
in expectation, one 3-block between two 4-blocks
2
1
1
1
0
0
0
0
0
0
0
0
B



E'
E
2^3
4
4
2^2
3
2^1
2
in expectation, one 2-block between two 3-blocks
2^0
1
1
1
0
0
0
0
0
0
0
0
B

Infix succinctness

- followDown produces in expectation lvl(E’) blocks
- $|aux| = |C'|\log(|C|) = \mathrm{polylog}(|C|)\log(|C|) \in \Theta(\mathrm{polylog}(|C|))$

Infix security

- From suffix security, π[-1] will (eventually) be adopted by all honest parties
- Therefore, can we use the same verifier?
  - No -- there is an attack!
- Does the chain ending in π[-1] contain the witness?
- Cannot trust proof provided as-is
  - Witness could have been skipped
- Adversary generates proof identical to honest, but skips followDown blocks
- Adversarial proof wins in suffix verifier, but does not contain witness!
- We must include all witness *ancestors* of π[-1]

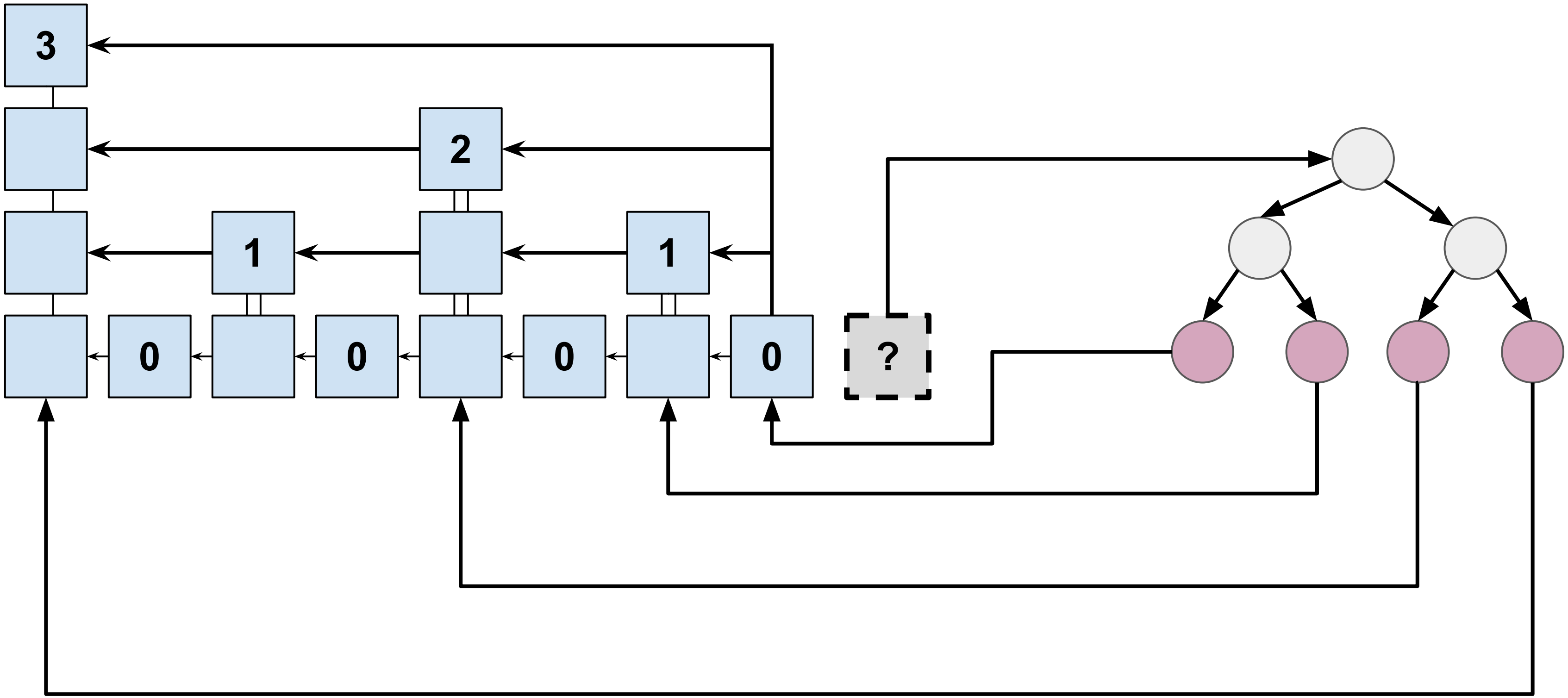
Algorithm 7 The verify algorithm for the NIPoPoW infix protocol

function $\mathsf{ancestors}(B, \mathsf{blockById})$
  if $B = \mathrm{Gen}$ **then**
    return $\{B\}$
  end if
  $\mathcal{C} \leftarrow \emptyset$
  for $\mathsf{id} \in B.\mathsf{interlink}$ **do**
    if $\mathsf{id} \in \mathsf{blockById}$ **then**
      $B' \leftarrow \mathsf{blockById}[\mathsf{id}]$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \mathsf{ancestors}(B', \mathsf{blockById})$ ▷ Collect into DAG
    end if
  end for
  return $\mathcal{C} \cup \{B\}$
end function
function $\mathsf{verify\text{-}infx}^{D}_{\ell,m,k}(\mathcal{P})$
  $\mathsf{blockById} \leftarrow \emptyset$
  for $(\pi, \chi) \in \mathcal{P}$ **do**
    for $B \in \pi$ **do**
      $\mathsf{blockById}[\mathsf{id}(B)] \leftarrow B$
    end for
  end for
  $\tilde{\pi} \leftarrow$ best $\pi \in \mathcal{P}$ according to suffix verifier
  return $D(\mathsf{ancestors}(\tilde{\pi}[-1], \mathsf{blockById}))$
end function

deployment paths

The Interlink Merkle Tree

- Collect all interlinks into a Merkle Tree
- It is sufficient to commit to the interlink merkle tree root
- Proofs for interlink inclusion are then $\Theta(\log(\log(|C|)))$
- Repeated elements in the same Interlink Merkle Tree can be skipped (saves a lot of space after superblocks in practice)

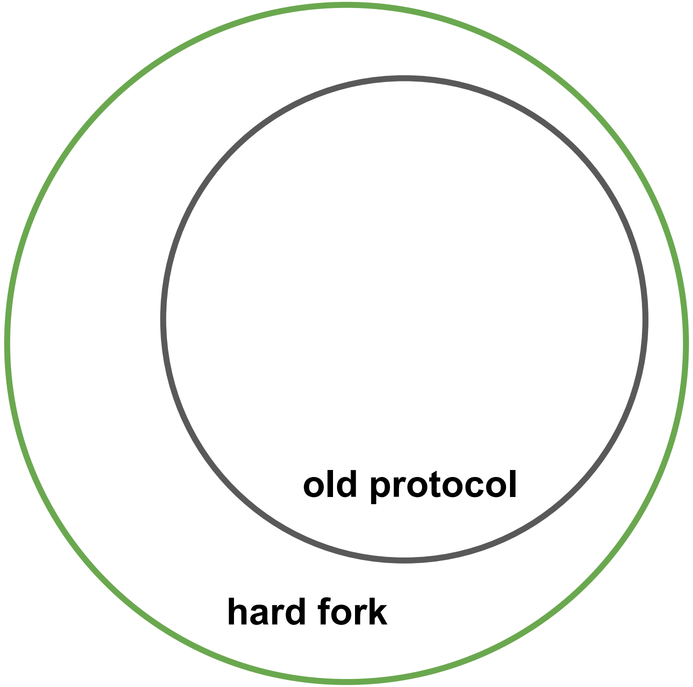

3
2
1
1
0
0
0
0
?

Review: Hard and soft forks

Mechanisms to propose changes in the full node software

Hard fork

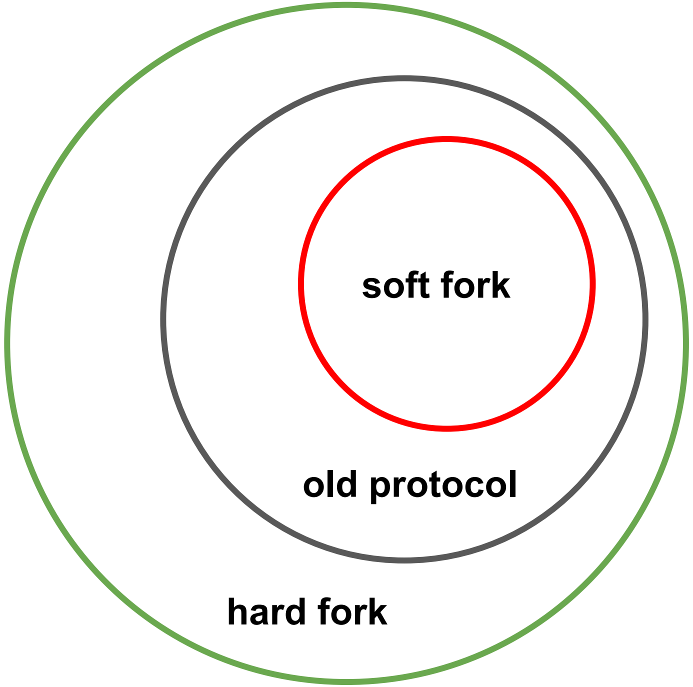
- **Increases** the validity language
- txs / blocks that were invalid are now valid
- All old valid txs / blocks are still valid
- Old miners **reject** some new-style txs / blocks
- New miners **accept** old-style txs / blocks



Review: Hard and soft forks

Soft fork

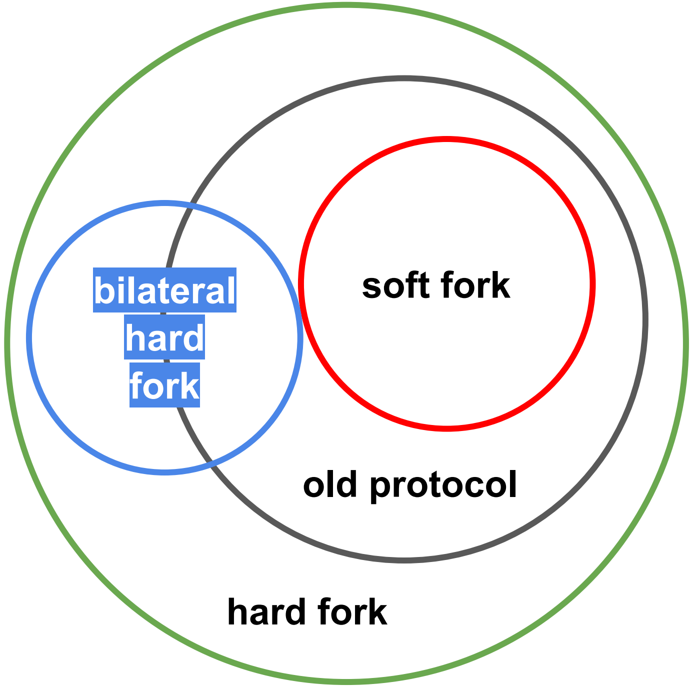
- **Reduces** the validity language
- txs / blocks that were valid are now invalid
- All old invalid txs / blocks are still invalid
- Old miners **accept** new-style txs / blocks
- New miners **reject** some old-style txs / blocks



Review: Hard and soft forks

Bilateral hard fork

- **Modifies** the validity language
- Some txs / blocks that were valid are now invalid
- Some txs / blocks that were invalid are now valid
- New miners reject some old-style txs / blocks
- Old miners reject some new-style txs / blocks



A hard fork

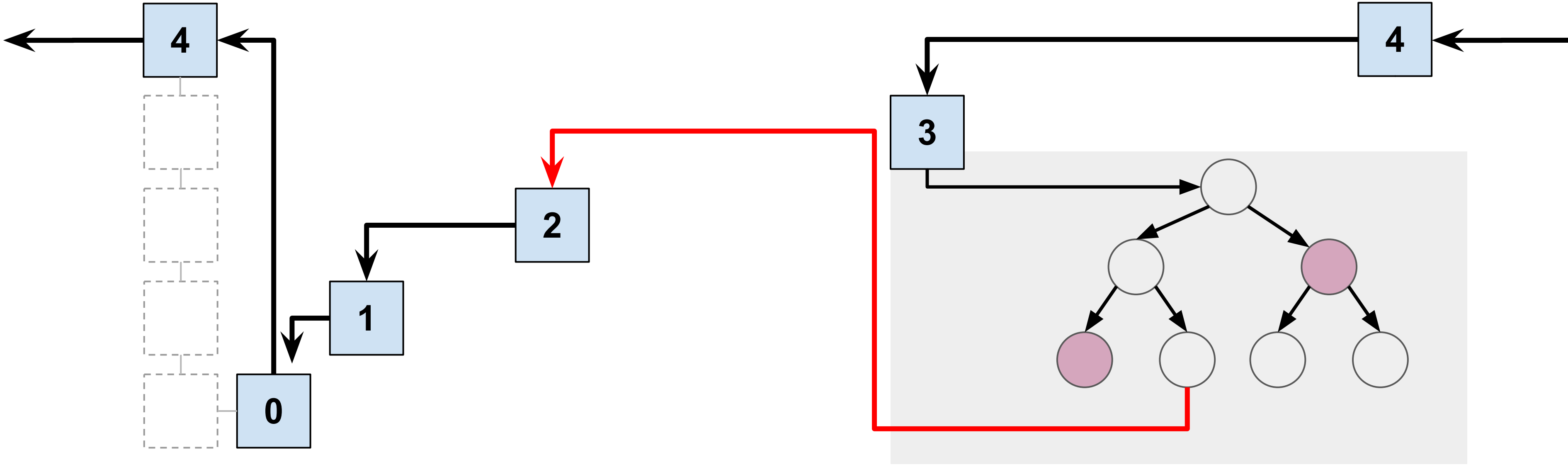
- Change the block format to include interlink
  Old format: B = mtr || nonce || **previd**
  New format: B = mtr || nonce || **interlink-mtr**
- Block data includes all interlinks and is required for full block verification
- Can be adopted for old cryptocurrencies willing to hard fork
- Can be adopted for new cryptocurrencies
- ERGO, nimiq, WebDollar have adopted this NIPoPoW
  - In production, handling > $2,000,000 **currently**

Hard fork prover/verifier

- NIPoPoW prover modified to include interlink proof-of-inclusion for every pointer in proof
- NIPoPoW prover, as full node, can find the MT proof, as it maintains interlink on its own
- NIPoPoW verifier checks interlink proof-of-inclusion for every pointer in proof

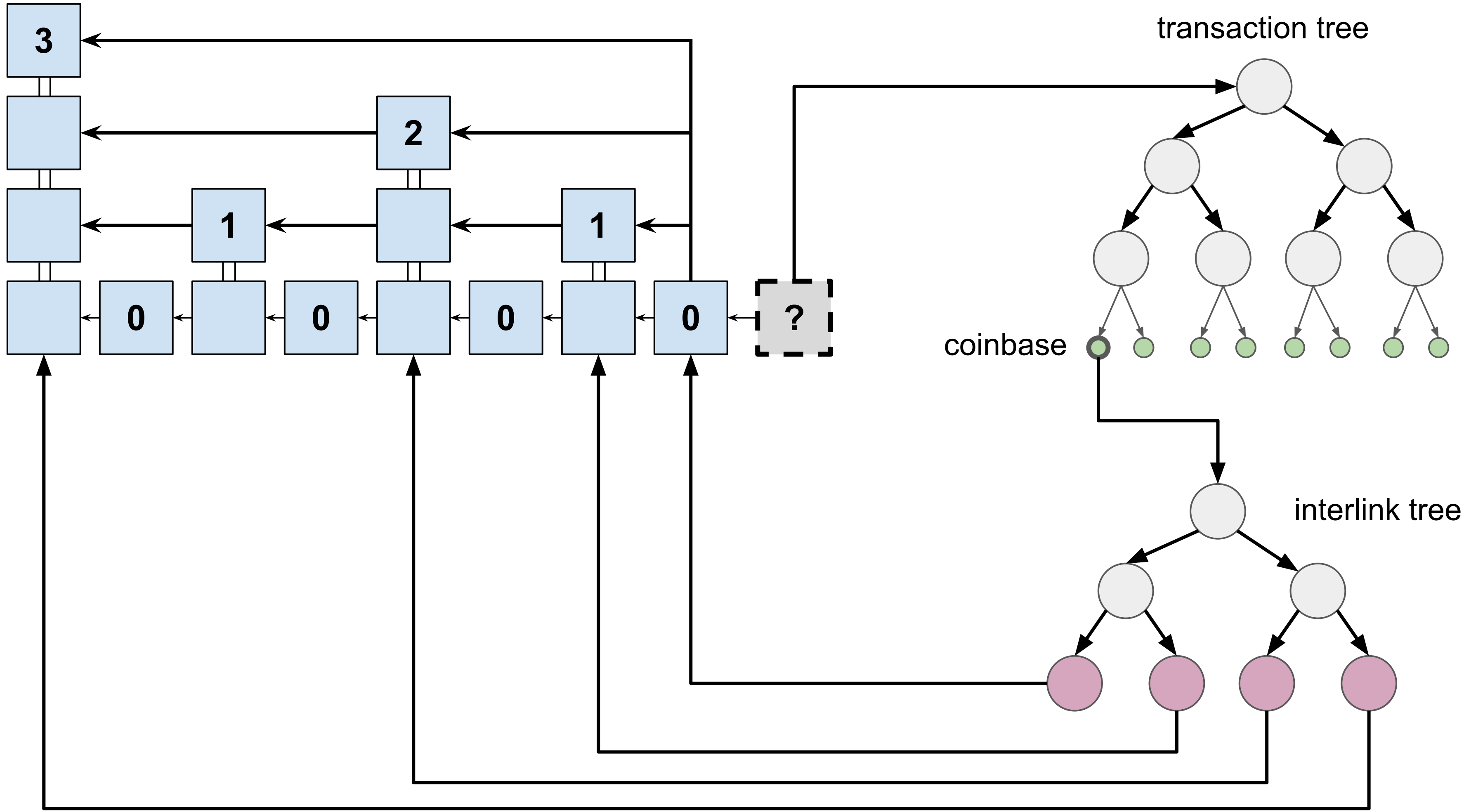
Hard fork NIPoPoW prover

- For every link in the chain, includes an interlink proof-of-inclusion
- Proof size is now (2m log(|C|) + polylog(|C|) log(|C|)) log(log(|C|))



A soft fork

- Keep block format
  B = mtr || nonce || **previd**
- Place interlink-mtr in coinbase transaction
- Change blockchain validity rules:
  - Each block must contain the interlink-mtr in coinbase, otherwise is rejected
- Prover includes for every pointer in chain:
  - Coinbase tx proof-of-inclusion
  - Interlink proof-of-inclusion
- Verifier checks the two proofs-of-inclusion



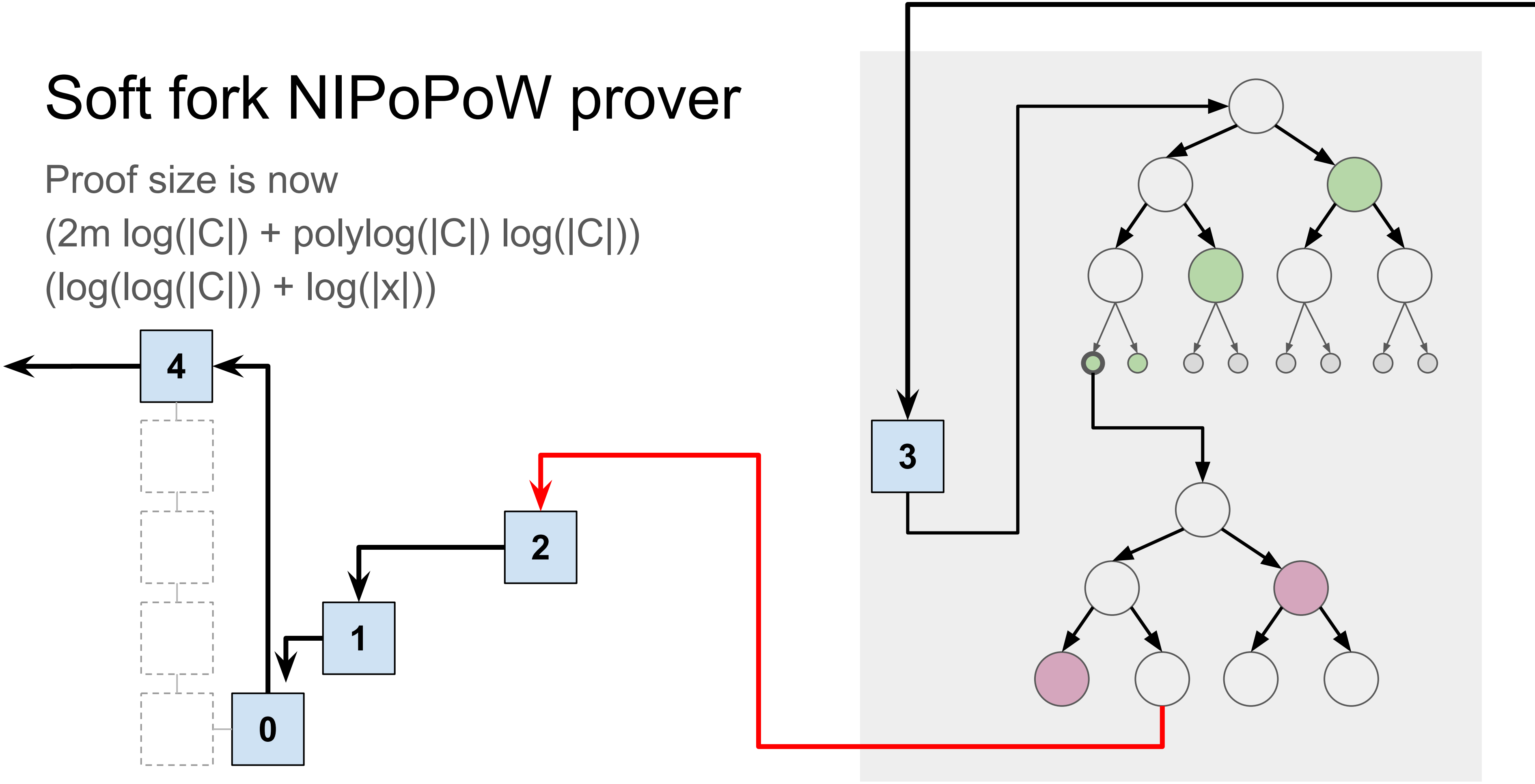
3
2
1
1
0
0
0
0
?
transaction tree
coinbase
interlink tree

Soft fork NIPoPoW prover

Proof size is now

(2m log(|C|) + polylog(|C|) log(|C|))

(log(log(|C|)) + log(|x|))



Velvet forks

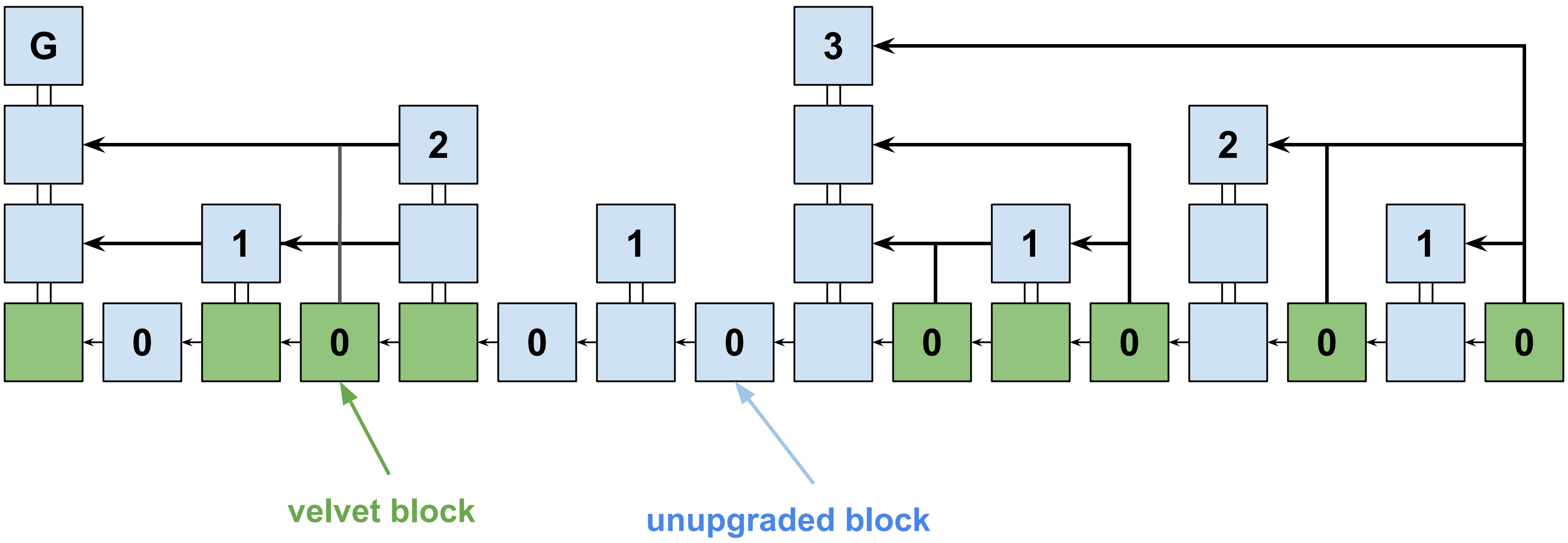
- Validity language does not change
- New miners accept all old-style txs / blocks
- Old miners accept all new-style txs / blocks
- New-style txs / blocks contain additional metadata
- Metadata can be interpreted in a useful way
- If metadata is wrong / incorrect / missing,
  tx / block is still accepted, but data is ignored
- **Metadata must function as verifiable certificate**

old protocol
=
velvet fork protocol

Velvet forked interlinks

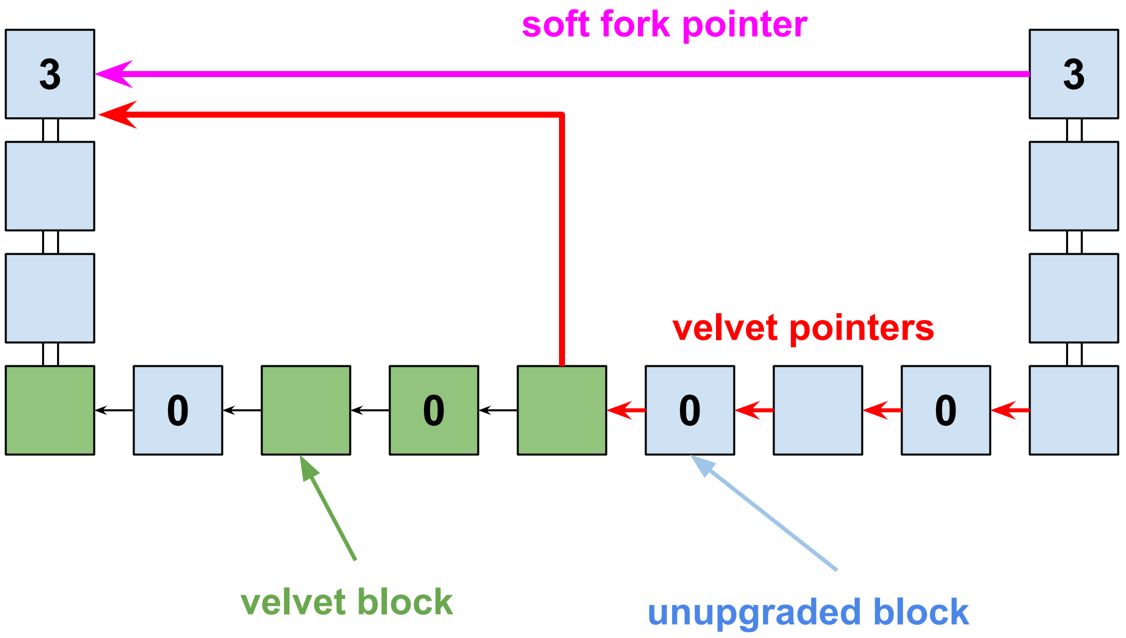
- Can we deploy NIPoPoWs on existing systems without soft/hard forks?
- What if we don't **require** miners to put interlink in coinbase?
- If the miner wants, they can include it
- If not, they don't have to -- they can even include false data
- Upon receiving a block, interlink vector is not validated
- All blocks, regardless of interlink, will be considered valid
- *Some* blocks will contain valid interlinks
- Suppose a percentage g of blocks are upgraded
- We don't need supermajority to make this upgrade!

The velvet forked interlinked chain



Velvet prover

- Prover creates exactly the same infix/suffix proof
- For each block B in proof, we need to join it with its previous block B’
- Prover is a full node
  - Maintains **realLink** for every block -- the interlink that *should have been* included
  - Compares **realLink** to included interlink to determine if interlink vector is valid
- Is the interlink vector of B valid?
  - Yes -- include pointer to B’ as in soft fork
  - No -- include previd pointer to prev block, set B = prev(B)
  - Repeat until we are connected to B’



soft fork pointer
3
3
velvet pointers
0
0
0
0
velvet block
unupgraded block

Algorithm 11 The Prove algorithm for the NIPoPoW protocol, modified for a velvet fork.

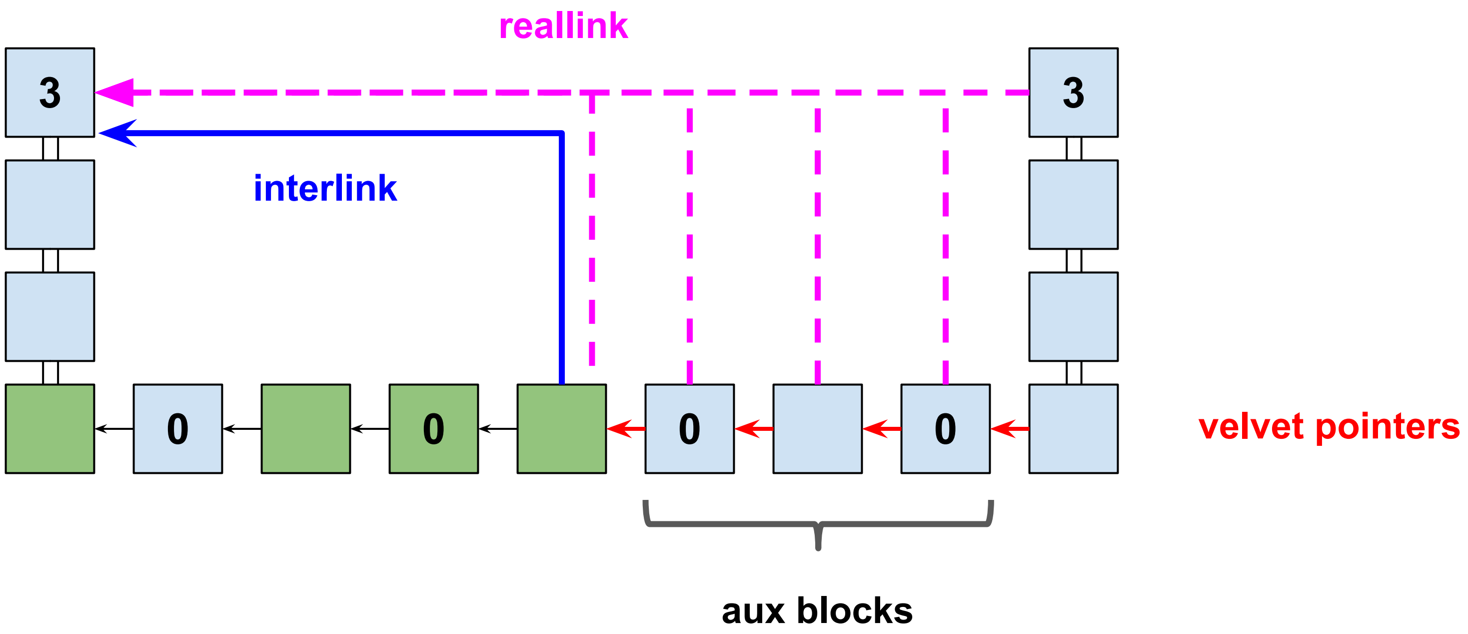
function $\mathsf{Prove'}_{m,k}(\mathcal{C}, \mathsf{realLink}, \mathsf{blockById})$
    $max\mu \leftarrow |\mathsf{realLink}[\mathsf{id}(\mathcal{C}[-k-1])]|$
    $b \leftarrow \mathcal{C}[0]$ ▷ Genesis block
    $\tilde{\Pi} \leftarrow \emptyset$
    for $\mu = max\mu$ down to 0 **do**
        $\pi, aux \leftarrow \mathsf{find}\ \mathcal{C}[:-k]{\uparrow}^{\mu}\ (b, \mathsf{realLink}, \mathsf{blockById})$
        if $|\pi| \geq m$ **then**
            $b \leftarrow \pi[-m]$
        end if
        $\tilde{\Pi} \leftarrow \tilde{\Pi} \cup aux$
    end for
    $\chi \leftarrow \mathcal{C}[-k:]$
    return $\tilde{\Pi}\chi$
end function

Algorithm 9 Supplying the necessary data to calculate a connected $\mathcal{C}{\uparrow}^{\mu}$ during a velvet fork.

function find $\mathcal{C}{\uparrow}^{\mu}(b, \text{realLink}, \text{blockById})$
    $B \leftarrow \mathcal{C}[-1]$
    $\mathsf{aux} \leftarrow \{B\}$
    $\pi \leftarrow [\,]$
    if $\mathit{level}(B) \geq \mu$ **then**
        $\pi \leftarrow \pi B$
    end if
    while $B \neq b$ **do**
        $(B, \mathsf{aux'}) \leftarrow \mathsf{followUp}(B, \mu, \mathsf{realLink}, \mathsf{blockById})$
        $\mathsf{aux} \leftarrow \mathsf{aux} \cup \mathsf{aux'}$
        $\pi \leftarrow \pi B$
    end while
    return π, aux
end function

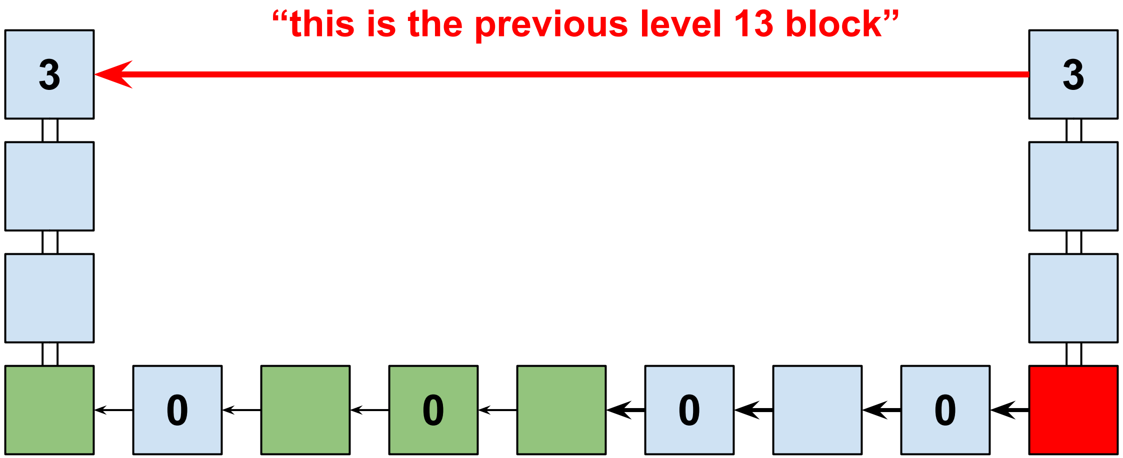
Algorithm 10 followUp produces the blocks to connect two superblocks in velvet forks.

```
function followUp(B, μ, realLink, blockById)
    aux ← {B}
    while B ≠ Gen do
        if B.interlink[μ] = realLink[id(B)][μ] then
            id ← B.interlink[μ]
        else                                        ▷ Invalid interlink
            id ← B.previd
        end if
        B ← blockById[id]
        aux ← aux ∪ {B}
        if level(B) = μ then
            return B, aux
        end if
    end while
    return B, aux
end function
```


reallink
3
3
interlink
0
0
0
0
velvet pointers
aux blocks

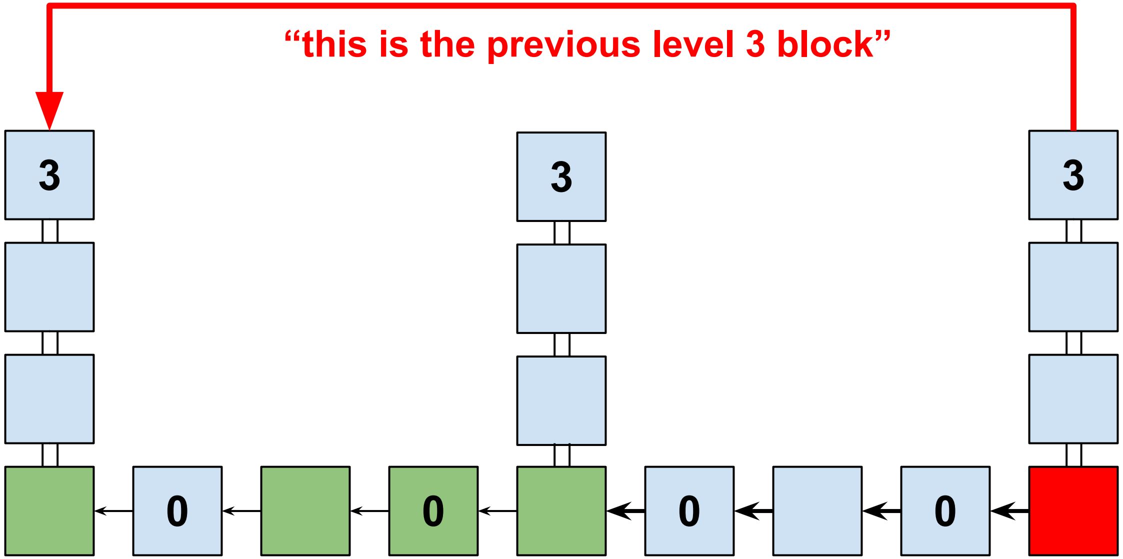
Fake velvet pointers?

- Adversary can lie in interlinks, as they are not verified before inclusion
- If lie includes level, the lie is detected by verifier by hashing referred block
- This is done by regular verifier anyway



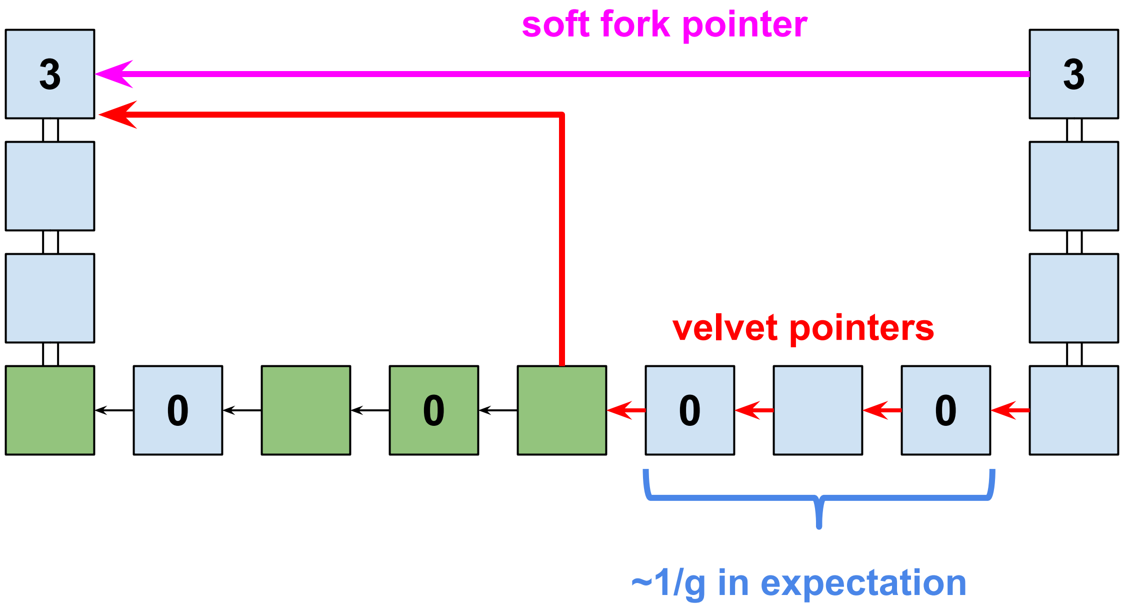
Fake velvet pointers?

- Adversary can include pointer to older block of correct level, not latest
- This cannot harm honest proof -- only adversarial proofs



Velvet security

- Velvet proofs are superchains of non-velvet proofs
- If adversary does not include interlink vector,
  no harm is done in honest proofs
- Adversarial proofs are as before, or worse
- Hence, security argument does not change


soft fork pointer
3
3
velvet pointers
0
0
0
0
~1/g in expectation

Velvet succinctness

Proof size is, in expectation:

$$(1/g)$$
$$(2m \log(|C|) + \mathrm{polylog}(|C|) \log(|C|))$$
$$(\log(\log(|C|)) + \log(|x|))$$

velvet factor

$$(1/g)$$

blocks in
suffix proof

witness size

blocks in
followUp

$$(2m \log(|C|) + \text{polylog}(|C|) \log(|C|))$$

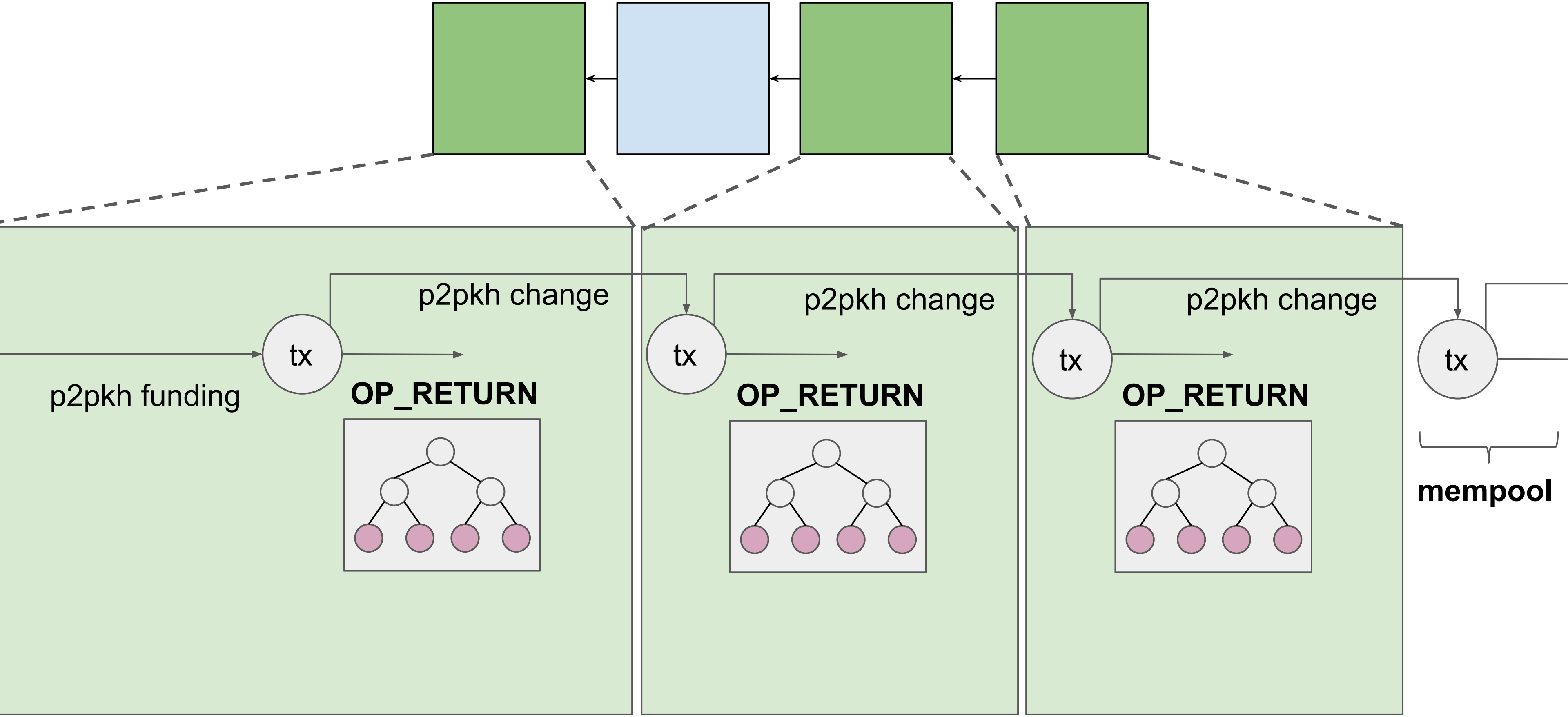
$$(\log(\log(|C|)) + \log(|x|))$$

interlink
proof-of-inclusion

velvet tx
proof-of-inclusion

User-activated velvet fork

- Observe that velvet data does not have to be in coinbase!
- Any tx will do
- What if miners were unaware of our fork?
- Users inject velvet txs with OP_RETURN containing interlink commitment
- Prover knows how to find these txs and include them in their proof



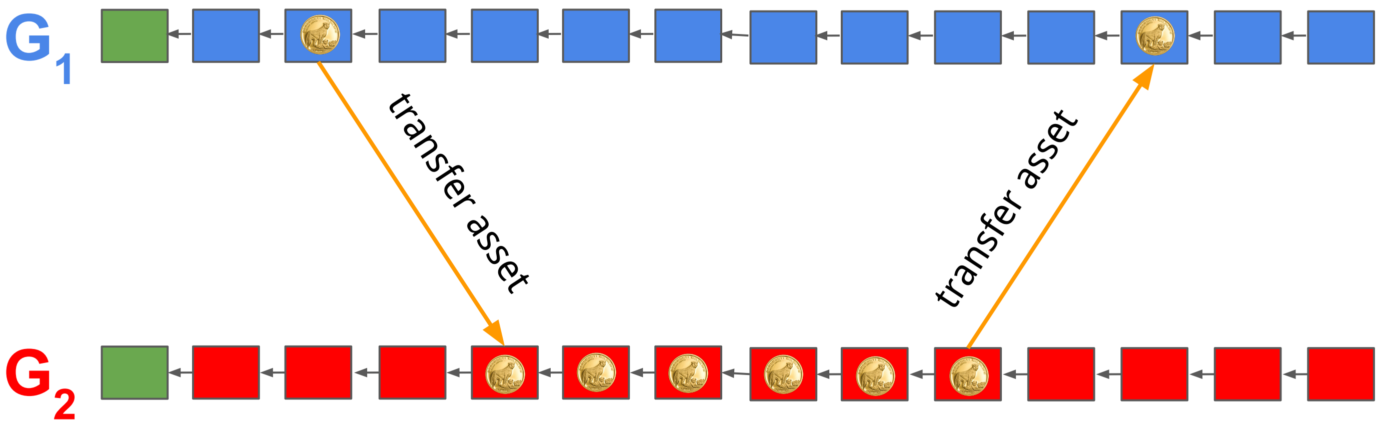
p2pkh change
p2pkh change
p2pkh change
tx
tx
tx
tx
p2pkh funding
OP_RETURN
OP_RETURN
OP_RETURN
mempool

UAVF

- Some tx may not make it in the right block
- That’s okay -- they can be ignored by prover
- (or used partially if *some* of their velvet pointers are correct)

work sidechains

The two-way peg: A cross-chain asset



Two-way pegging

- Given two blockchains G_1, G_2

Assumption: Both chains:

- Are **Turing-complete** (Solidity)
- Have **Proof-of-Work** consensus
- Have (velvet) support for **NIPoPoWs** back to their Geneses
- Are individually secure (2 honest majority assumptions)

Construction must be trustless!

We will work with the **native** asset of G_1 and mirror it as ERC-20 on G_2

Predicate for two-way peg

Predicate Q for $\boldsymbol{G_1}$ (similarly for $\mathbf{G_2}$)

- There was a money-locking transaction *tx*
- *tx* called a particular smart contract function
- *tx* was paid for by *author*
- *tx* has amount *amount*
- *tx* belongs to a block B
- B is k-confirmed in a chain C
- C is an honestly adopted chain
- $C[0] = \boldsymbol{G_1}$

Predicate will be represented by **Solidity event**:
It has a **name** and **parameters**, and the solidity code can **fire** it

Smart contract strategy

- Create a **base** smart contract **crosschain**
- Make two smart contracts, one for each chain

sidechain_1 contract inherits from **crosschain**, runs on **G_1**, and can:

- be **paid** in **G_1** native currency
- **pay back** in **G_1** native currency

fires **Deposited_1** event when paid

sidechain_2 contract inherits from **crosschain**, runs on **G_2**, and can:

- be **paid** in **G_2** ERC-20 currency
- **pay back** in **G_2** ERC-20 currency

fires **Deposited_2** event when paid

contract sidechain$_1$ **extends** crosschain$_{k,m,z}$

payable function deposit(target)

▷ Emit an event to be picked up by the remote contract

ctr $\leftarrow$ ctr $+ 1$

emit Deposited$_1$(target, msg.value, ctr)

payment event emission

end function

end contract

contract sidechain$_2$ **extends** crosschain$_{k,m,z}$; ERC20

mapping(address $\Rightarrow$ int) balances

standard ERC-20 behavior

function deposit(target, amount)

▷ Charge account of sender

if balances[msg.sender] $<$ amount **then**

return $\perp$

end if

balances[msg.sender] $\leftarrow$ balances[msg.sender] $-$ amount

▷ Emit an event to be picked up by the remote contract

ctr $\leftarrow$ ctr $+ 1$

emit Deposited$_2$(target, amount, ctr)

payment event emission

end function

end contract

Initialization timeline

1. **$sidechain_1$** smart contract constructed
2. **$sidechain_1$** instantiated in G_1 address **$sidechain_1^{addr}$**
3. **$sidechain_2$** smart contract constructed
4. **$sidechain_2$** instantiated in G_2 address **$sidechain_2^{addr}$**
5. Owner calls **$sidechain_1$** initializer with parameters: $H(\mathbf{G_2})$, **$sidechain_2^{addr}$**
6. Owner calls **$sidechain_2$** initializer with parameters: $H(\mathbf{G_1})$, **$sidechain_1^{addr}$**
7. Both owners discarded so that the process is now trustless
8. People wishing to use smart contract verify correct initialization

contract **crosschain**$_{k,m,z}$

  internal function initialize($\mathcal{G}$)

    this.$\mathcal{G} \leftarrow \mathcal{G}$ record remote genesis hash

  end function

  ...

end contract

contract **sidechain**$_1$ **extends** **crosschain**$_{k,m,z}$

  initialized $\leftarrow$ *false*

  function initialize($\mathcal{G}_2$, sidechain$_2$)

    if $\neg$initialized **then**

      ctr $\leftarrow 0$

      crosschain.initialize($\mathcal{G}_2$) $\triangleright$ Initialize with the remote chain genesis block

      initialized $\leftarrow$ *true*   make contract owner powerless

      this.sidechain$_2$ $\leftarrow$ sidechain$_2$

    end if

  end function   record remote sibling contract address

  ...

end contract

Recording cross-chain events

- **crosschain** contract allow recording remote events
- **Claim** that event took place is submitted to contract with call to **submit-event-proof** smart contract function
- Event submission takes parameters NIPoPoW $\boldsymbol{\pi}$ and event details **e**
- Event claim is recorded to **events** storage for now

contract crosschain$_{k,m,z}$

NIPoPoW

event name & parameter values

    payable function submit-event-proof(π, e)

        if msg.value $< z$ **then** $\triangleright$ Ensure sufficient collateral

            return $\perp$

event claim for the same event is not pending

        end if

        if events$[e] = \perp \wedge$ verify$^{e,\mathcal{G}}_{k,m}(\{\pi\})$ **then**

            events$[e] \leftarrow \{$expire : block.number $+ k$, proof : π, author : msg.sender$\}$

        end if

    end function

NIPoPoW is stored with event

end contract

event claim is recorded to smart contract storage (with e as dictionary key)

verify NIPoPoW for syntactic validity

Disproving event claims

- An event claim can be **fraudulent**
- Allow an **expiration period** of **k** blocks after claim submission for fraud to be reported
- **k** here corresponds to target blockchain liveness parameter
- Party making claim must **pay collateral *z*** during claim
- If fraud is proved, collateral is **paid to fraud prover**
- If fraud is not proved, collateral is **reclaimed by claimer**
- Fraud proof can be provided by calling **submit-contesting-proof** function
- Parties are “cryptoeconomically” incentivized to submit fraud proofs
- collateral $z \geq$ gas of fraud proof submission + cost of chain monitoring

$\mathbf{contract\ crosschain}_{k,m,z}$

$\quad \mathbf{payable\ function}\ \textsf{submit-event-proof}(\pi, e)$

$\quad\quad \mathbf{if}\ \textsf{msg.value} < \boxed{z}\ \mathbf{then}$ $\qquad\qquad \triangleright$ Ensure sufficient collateral

$\quad\quad\quad \mathbf{return}\ \bot$

$\quad\quad \mathbf{end\ if}$

$\quad\quad \mathbf{if}\ \textsf{events}[e] = \bot \wedge \textsf{verify}^{e,\mathcal{G}}_{k,m}(\{\pi\})\ \mathbf{then}$

$\quad\quad\quad \textsf{events}[e] \leftarrow \{\boxed{\textsf{expire} : \textsf{block.number} + k,}\ \textsf{proof} : \pi, \boxed{\textsf{author} : \textsf{msg.sender}}\}$

$\quad\quad \mathbf{end\ if}$

$\quad \mathbf{end\ function}$

$\mathbf{end\ contract}$

collateral is hard-coded

block after which claim will be finalized and no fraud proofs will be possible

author is recorded with event claim to return collateral if honest

fraud proof

ensure contestation period still active

contract **crosschain**$_{k,m,z}$

  function submit-contesting-proof($\boxed{\pi}$, e)

    if $\text{events}[e] = \perp \vee \boxed{\text{block.number} \geq \text{events}[e].\text{expire}}$ **then**

      return $\perp$

    end if

whole NIPoPoW verifier - finds honest proof

    if $\boxed{\neg\text{verify}^{e,\mathcal{G}}_{k,m}}(\{\boxed{\text{events}[e].\text{proof}}, \boxed{\pi}\})$ **then** ▷ Original proof was fraudulent

      $\boxed{\text{events}[e] \leftarrow \perp}$

      msg.sender.send(z) ▷ Pay collateral to contester

    end if

  end function

end contract

original NIPoPoW - proves truth

contesting NIPoPoW - proves falsity

clear event claim for e

Finalizing an event

- If no fraud claim is made during contestation period, then event proof was honest
- We know event happened
- Record it in **finalized-events** permanently
- Allow child contracts to check if event has been finalized by invoking **event-exists**

```
contract crosschain_{k,m,z}

    function finalize-event(e)
        if events[e] = ⊥ ∨ block.number < events[e].expire then
            return ⊥
        end if
        finalized-events ← finalized-events ∪ {e}
        author ← events[e].author
        events[e] ← ⊥
        author.send(z)                                        ▷ Return collateral
    end function

    function event-exists(e)
        return e ∈ finalized-events
    end function

end contract
```

event no longer a claim, but a fact

Withdrawing money

- Simply check if remote payment event took place
- If so, release funds
- Record event as “used” to avoid double spending (not shown in code)

contract **sidechain**$_1$ **extends** **crosschain**$_{k,m,z}$

    function withdraw(amount, target, ctr)

        $\triangleright$ Validate that event took place on remote chain

        if $\neg$event-exists((sidechain$_2$, Deposited$_2$, (amount, target, ctr))) **then**

            return $\perp$

        end if

        msg.sender.send(amount)

    end function

end contract

contract **sidechain**$_2$ **extends** **crosschain**$_{k,m,z}$; **ERC20**

    function withdraw(amount, target, ctr)

        $\triangleright$ Validate that event took place on remote chain

        if $\neg$event-exists((sidechain$_1$, Deposited$_1$, (amount, target, ctr))) **then**

            return $\perp$

        end if

        $\triangleright$ Credit target account

        balances[target] $\leftarrow$ balances[target] + amount

    end function

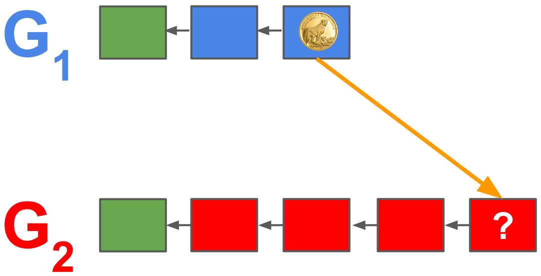
end contract

```
contract crosschain_{k,m,z}
    internal function initialize(𝒢)
        this.𝒢 ← 𝒢
    end function
    payable function submit-event-proof(π, e)
        if msg.value < z then                                   ▷ Ensure sufficient collateral
            return ⊥
        end if
        if events[e] = ⊥ ∧ verify^{e,𝒢}_{k,m}({π}) then
            events[e] ← {expire : block.number + k, proof : π, author : msg.sender}
        end if
    end function
    function finalize-event(e)
        if events[e] = ⊥ ∨ block.number < events[e].expire then
            return ⊥
        end if
        finalized-events ← finalized-events ∪ {e}
        author ← events[e].author
        events[e] ← ⊥
        author.send(z)                                          ▷ Return collateral
    end function
    function submit-contesting-proof(π, e)
        if events[e] = ⊥ ∨ block.number ≥ events[e].expire then
            return ⊥
        end if
        if ¬verify^{e,𝒢}_{k,m}({events[e].proof, π}) then       ▷ Original proof was fraudulent
            events[e] ← ⊥
            msg.sender.send(z)                                  ▷ Pay collateral to contester
        end if
    end function
    function event-exists(e)
        return e ∈ finalized-events
    end function
end contract
```

```
contract sidechain_1 extends crosschain_{k,m,z}
    initialized ← false
    function initialize(G_2, sidechain_2)
        if ¬initialized then
            ctr ← 0
            crosschain.initialize(G_2)    ▷ Initialize with the remote chain genesis block
            initialized ← true
            this.sidechain_2 ← sidechain_2
        end if
    end function
    payable function deposit(target)
        ▷ Emit an event to be picked up by the remote contract
        ctr ← ctr + 1
        emit Deposited_1(target, msg.value, ctr)
    end function
    function withdraw(amount, target, ctr)
        ▷ Validate that event took place on remote chain
        if ¬event-exists((sidechain_2, Deposited_2, (amount, target, ctr))) then
            return ⊥
        end if
        msg.sender.send(amount)
    end function
end contract
```

```
contract sidechain_2 extends crosschain_{k,m,z}; ERC20
    mapping(address ⇒ int) balances
    initialized ← false
    function initialize(G_1, sidechain_1)
        if ¬initialized then
            ctr ← 0
            crosschain.initialize(G_1)    ▷ Initialize with the remote chain genesis block
            initialized ← true
            this.sidechain_1 ← sidechain_1
        end if
    end function
    function deposit(target, amount)
        ▷ Charge account of sender
        if balances[msg.sender] < amount then
            return ⊥
        end if
        balances[msg.sender] ← balances[msg.sender] − amount
        ▷ Emit an event to be picked up by the remote contract
        ctr ← ctr + 1
        emit Deposited_2(target, amount, ctr)
    end function
    function withdraw(amount, target, ctr)
        ▷ Validate that event took place on remote chain
        if ¬event-exists((sidechain_1, Deposited_1, (amount, target, ctr))) then
            return ⊥
        end if
        ▷ Credit target account
        balances[target] ← balances[target] + amount
    end function
end contract
```

Sidechain liveness

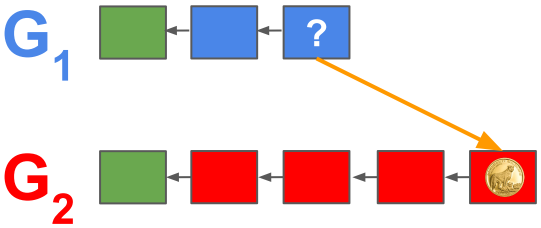


- By NIPoPoW security and backbone liveness

Liveness failure event:
Payment released in destination chain but not authored in source chain

- During destination chain contestation period, **backbone liveness** ensures one block will be honestly generated
- Honestly generated block will contest any adversarial proofs
- NIPoPoW security ensures honest proof will win, event will be rejected

Sidechain security



- By computational reduction to NIPoPoW security and backbone liveness

Security failure event:
Payment released in destination chain but not authored in source chain

- During destination chain contestation period, **backbone liveness** ensures one block will be honestly generated
- Honestly generated block will contest any adversarial proofs
- NIPoPoW security ensures honest proof will win, event will be rejected

References

- Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka
  “**Proofs of Proofs of Work with Sublinear Complexity**”, FC 2016
- Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos
  “**The Bitcoin Backbone Protocol: Analysis and Applications**”, EUROCRYPT 2015
- Aggelos Kiayias, Andrew Miller, Dionysis Zindros
  “**Non-interactive proofs of proof-of-work**”, ePrint 2017-2018
- Peter Gaži, Aggelos Kiayias, Dionysis Zindros
  “**Proof-of-Stake Sidechains**”, to appear in IEEE Security & Privacy 2019
- Aggelos Kiayias, Dionysis Zindros. “**Proof-of-Work Sidechains**”, ePrint 2018
- Kostis Karantias, “**Constructing Interoperable Blockchains Using NIPoPoWs**”,
  Master Thesis manuscript, University of Ioannina 2019
- Aggelos Kiayias, Alexander Russell, Bernardo David, Roman Oliynykov
  “**Ouroboros: A provably secure proof-of-stake blockchain protocol**”, CRYPTO 2017
- A. Zamyatin, N. Stifter, A. Judmayer, P. Schindler, E. Weippl, W. Knottenbelt
  "**A Wild Velvet Fork Appears! Inclusive Blockchain Protocol Changes in Practice**", FC 2018

45DC 00AE FDDF 5D5C B988 EC86 2DA4 50F3 AFB0 46C7

Thanks! Questions?

INPUT | OUTPUT

CARDANO

National and Kapodistrian UNIVERSITY OF ATHENS