

# Lecture 07: State Storage & Mining






---

Akash Khosla  
Nick Zoghb





# LECTURE OUTLINE

- 1  ETHEREUM RECAP
- 2  CONSENSUS
- 3  BLOCK-BY-BLOCK
- 4  HASH STRUCTURES
- 5  DAGGER HASHIMOTO / ETHASH

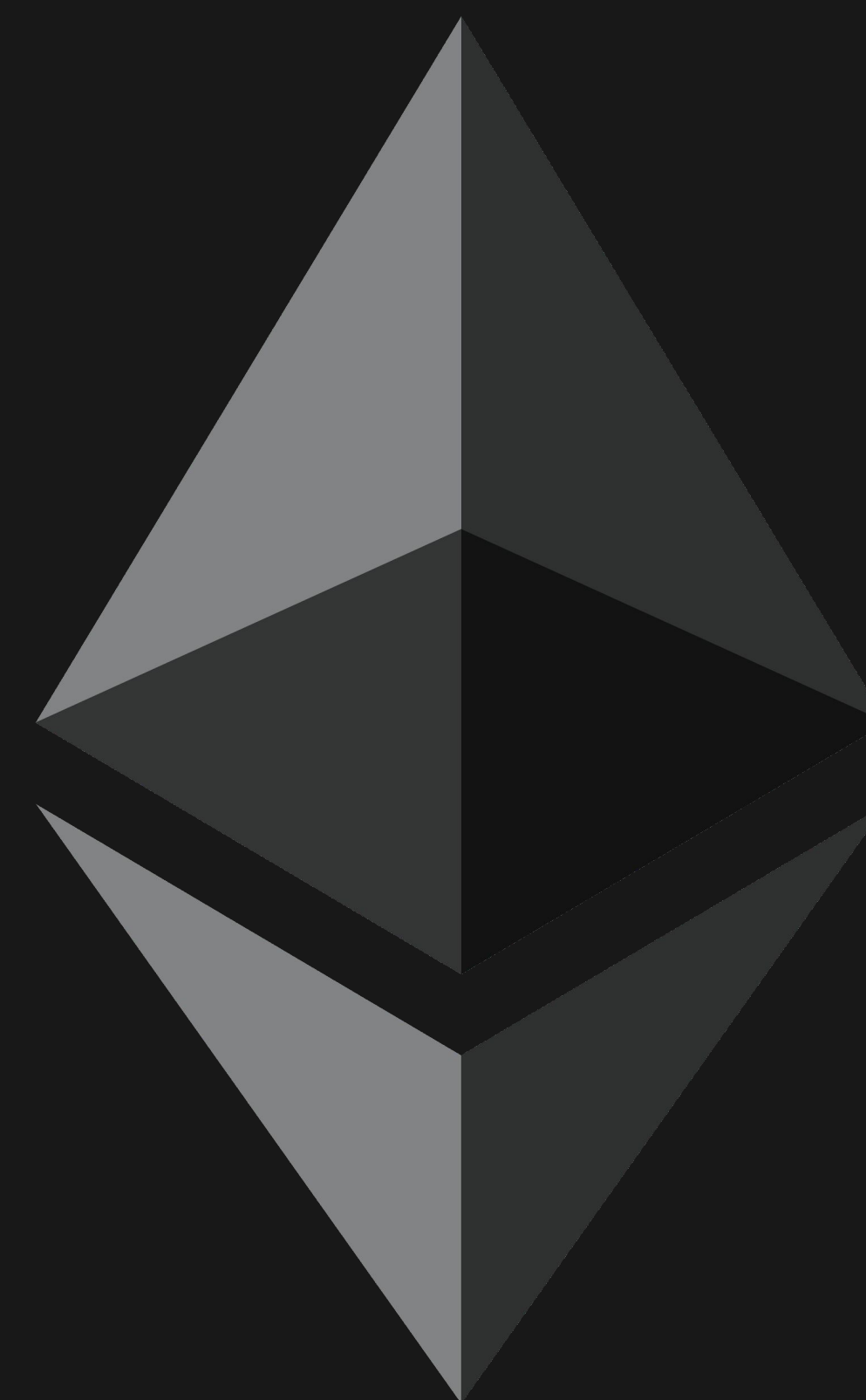
# 1 ETHEREUM RECAP



# WHAT IS ETHEREUM?

## VITALIK'S CHILD

- Ethereum is a **decentralized** platform designed to run **smart contracts**.
  - Resistant to censorship
  - **Distributed server**
- Ethereum maintains a native asset called **ether** that is the basis of value in the Ethereum ecosystem, allows for aligning incentives
- **Account based**, not **UTXO based**
  - Remember: a UTXO is not a transaction, but simple a mechanism to manipulate and maintain state for transactions
  - **Ethereum has transactions which are used to manipulate the global account state**

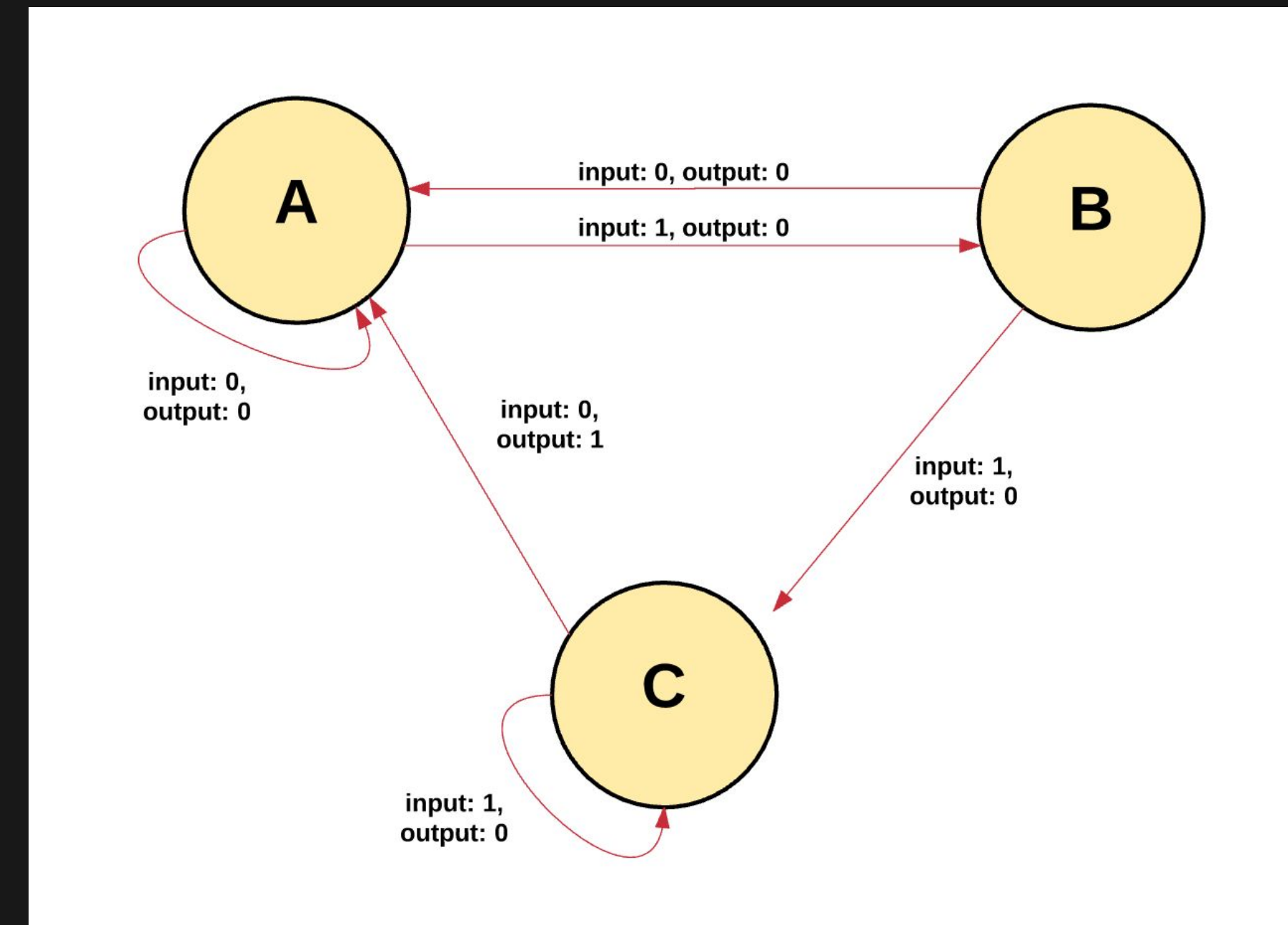




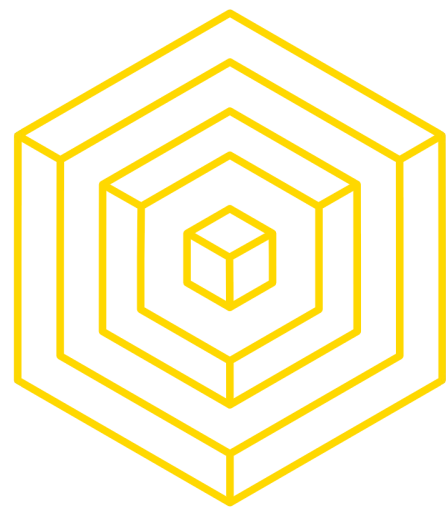
# WHAT IS THE BLOCKCHAIN FOR ETHEREUM

## PUTIN'S FAVORITE CRYPTOCURRENCY

- For Ethereum - a blockchain is a “cryptographically secure transactional singleton machine with shared-state.”
  - **Cryptographically secure** - Can't create fake transactions, erase transactions because of complex mathematical algorithms.
  - **Transaction singleton machine** - single instance of the machine for all the transactions being created in the system (“global truth”)
  - **Shared-state** - state stored on this machine is shared and open to everyone



[Source](#)



# UTXO VS. ACCOUNTS

## THE ULTIMATE COMPARISON

- A Bitcoin user's available **balance** is the **sum of unspent transaction outputs** for which they own the private keys to the output addresses
- Instead Ethereum uses a different concept, called **Accounts**, which already keeps track of **balance**

Bitcoin:

Bob owns private keys to set of UTXOs

5 BTC  $\Rightarrow$  Bob

3 BTC  $\Rightarrow$  Bob

2 BTC  $\Rightarrow$  Bob

Ethereum:

Evan owns private keys to an account

address: "0xfa38b..."

balance: 10 ETH

code:  $c := a + b$



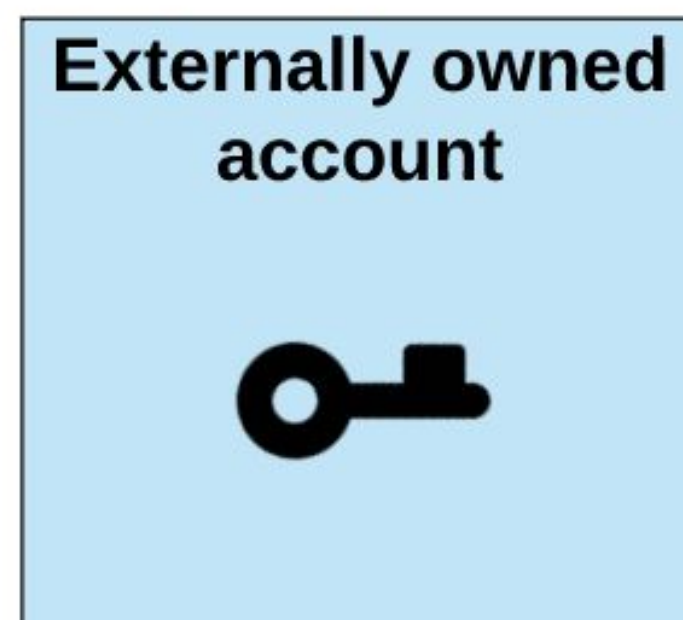


# ACCOUNTS

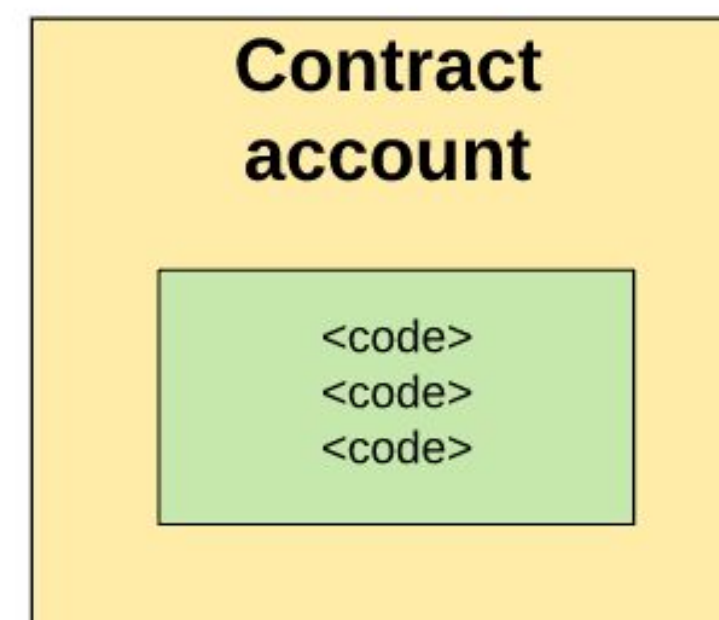
## HOW ETHEREUM MANAGES IDENTITY

7

- The global shared state of ethereum is comprised of many small objects (**Accounts**) that are able to interact with one another through a message-passing framework
- An **Account** has state and 20 byte/160 bit byte identifier, i.e:
  - `0x91fff4cbd6159a527ca4dcce2e3937431086c662`
- Two Types of Accounts:
  - **Externally owned**, which are controlled private keys and have no code associated with them.
  - **Contract accounts**, which are controlled by their code and have code associated with them.



[Source](#)

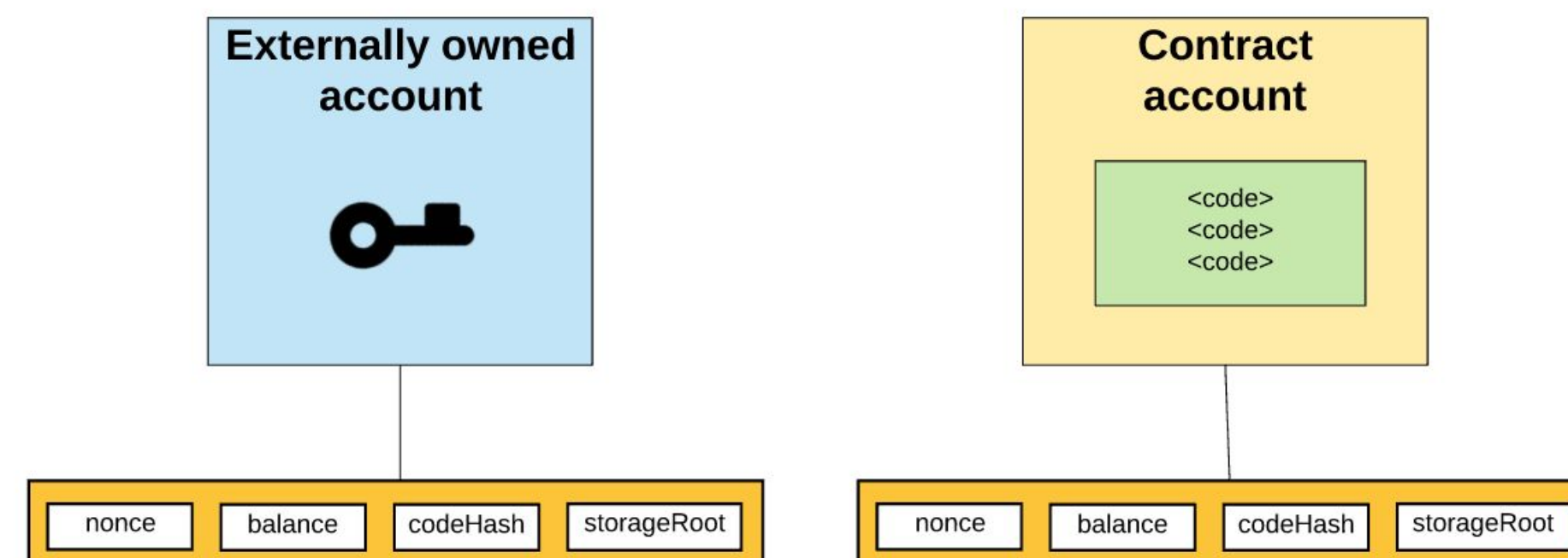




# WHAT IS STORED IN ACCOUNT STATE

## FOUR COMPONENTS

- **nonce**: number of transactions sent from external account, number of contracts created if contract account
- **balance**: The number of Wei owned by this address. 1e18 Wei per 1 Ether.
- **storageRoot**: A hash of the root node of a Merkle Patricia tree. This tree encodes the hash of the storage contents of this account, and is empty by default.
- **codeHash**: The hash of the EVM (Ethereum Virtual Machine—more on this later) code of contract account. Hash( " " ) for external accounts.







# ALL ACCOUNTS == NETWORK STATE

GLOBAL “TRUTH”

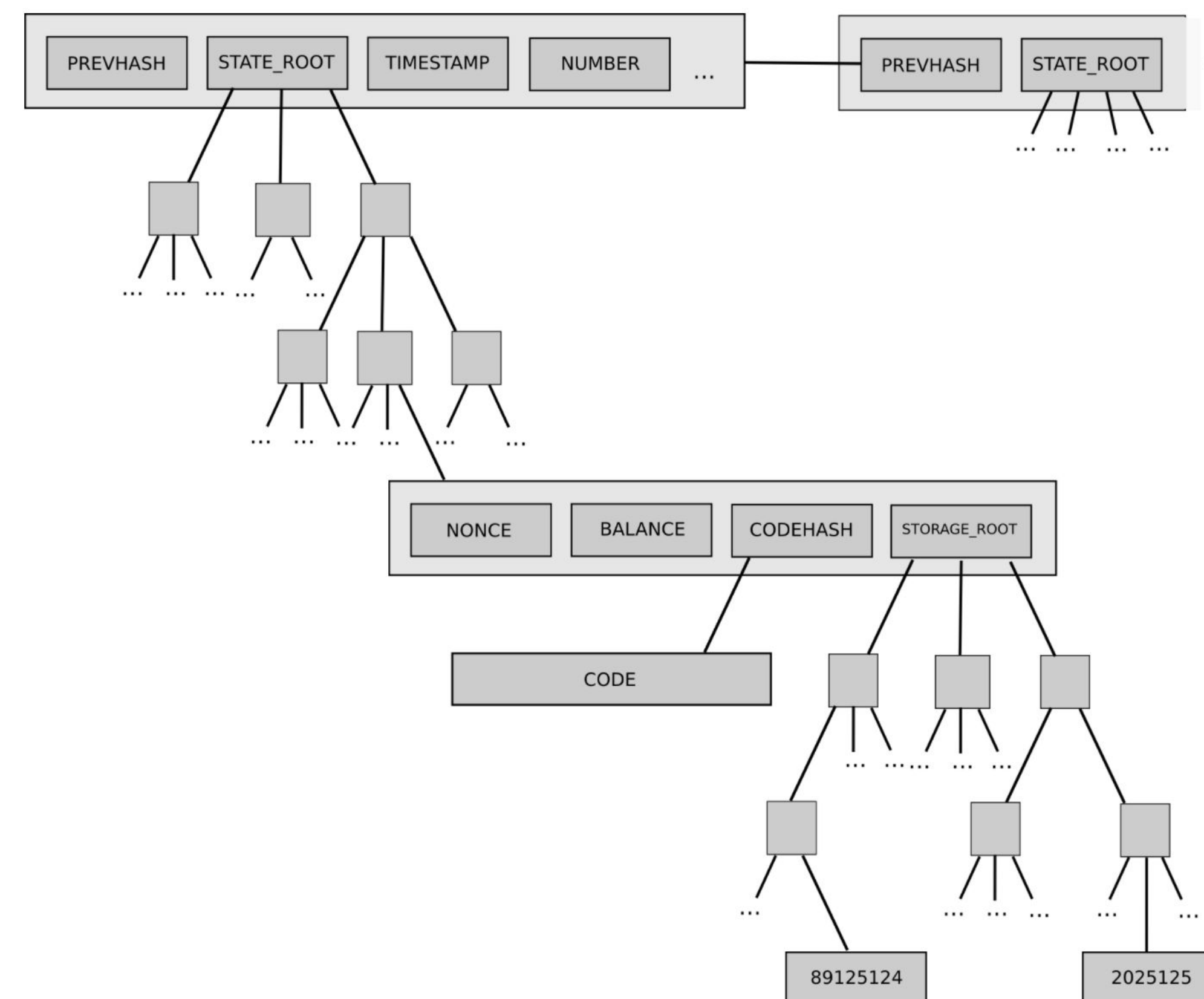
- **State of all accounts is the state of the Ethereum network**
  - Entire Ethereum network agrees on the current balance, storage state, contract code, etc. of every single account
- **Ethereum network state is updated with every block**
  - A block takes the previous state and produces a new network state
    - every node has to agree upon new network state
- **Accounts interact** with network, other accounts, other contracts, and contract state **through transactions**



# ACCOUNTS RATIONALE

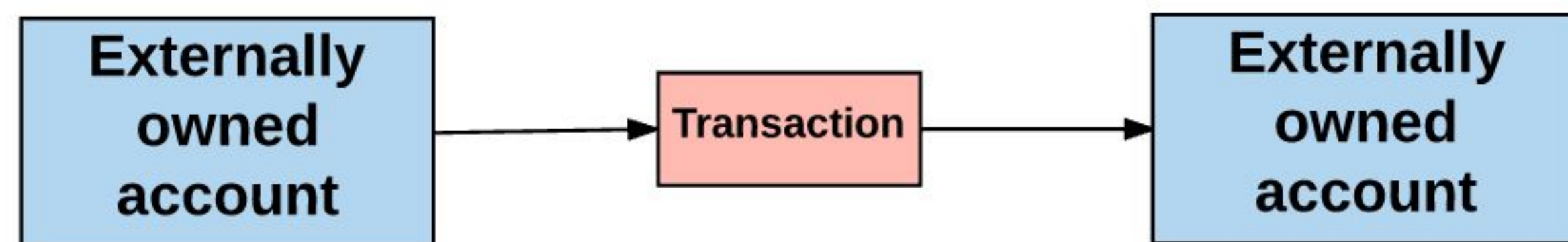
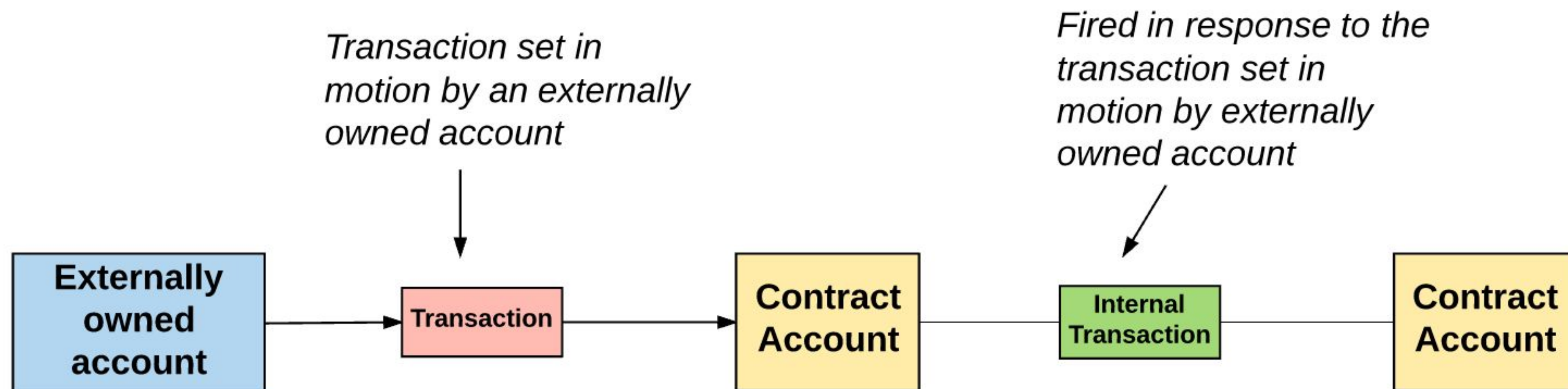
## WHY USE ACCOUNTS

- **Space Savings**
  - Nodes only need to update each account's balance instead of storing every UTXO
- **More Intuitive**
  - Smart contracts are more easier to program when transferring between accounts with a balance vs. constantly updating a UTXO set to compute user's available balance.
- **Comparable efficiency due to “Merklization”**
  - Use of merkle patricia tries allows for SPV light clients to work





# EXTERNALLY OWNED VS CONTRACT ACCOUNTS



Any action that occurs on the Ethereum blockchain is always set in motion by transactions fired from externally controlled accounts.

[Source](#)

AUTHOR: AKASH KHOSLA

BLOCKCHAIN FOR DEVELOPERS



# GAS AND PAYMENT

## FEES

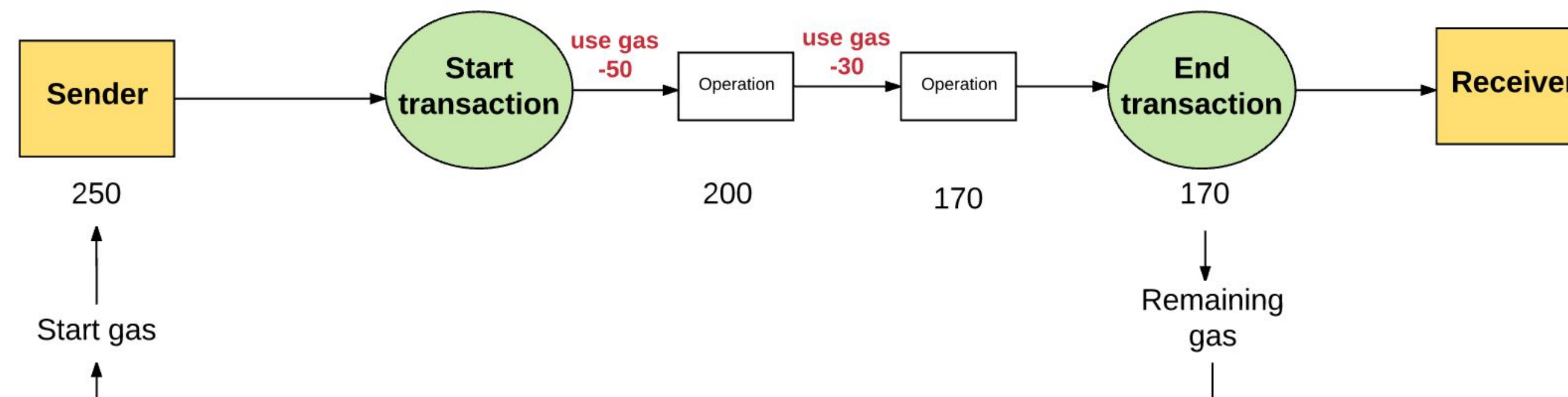
- Every computation that occurs as a result of a transaction on the Ethereum network incurs a fee called **gas**
- **Gas** is the unit used to measure the fees for a particular computation
- **Gas price** is the amount of Ether you are willing to spend on every unit of gas
  - Measured in “gwei” - 1E18 wei = 1 ETH, 1 gwei = 1,000,000,000 wei
- With every transaction, a sender sets a **gas limit** and a **gas price**
  - **gas price** \* **gas limit** = max **amount** of wei **sender is willing to pay** for transaction





# GAS AND PAYMENT FEES

- Example: **Gas Limit:** 50,000, **Gas Price:** 20 gwei
  - Max transaction fee:  $50,000 * 20 \text{ gwei} = 1,000,000,000,000,000,000 = 0.001 \text{ Ether}$
- Since the **gas limit** is the maximum gas the sender is willing to spend money on, if they have enough Ether in their account balance to cover this maximum and the **gas limit** is enough to execute the transaction, transaction will execute
  - `startGas == gasLimit == gasUsed`
- Unused gas is refunded to the sender



Source

AUTHOR: AKASH KHOSLA

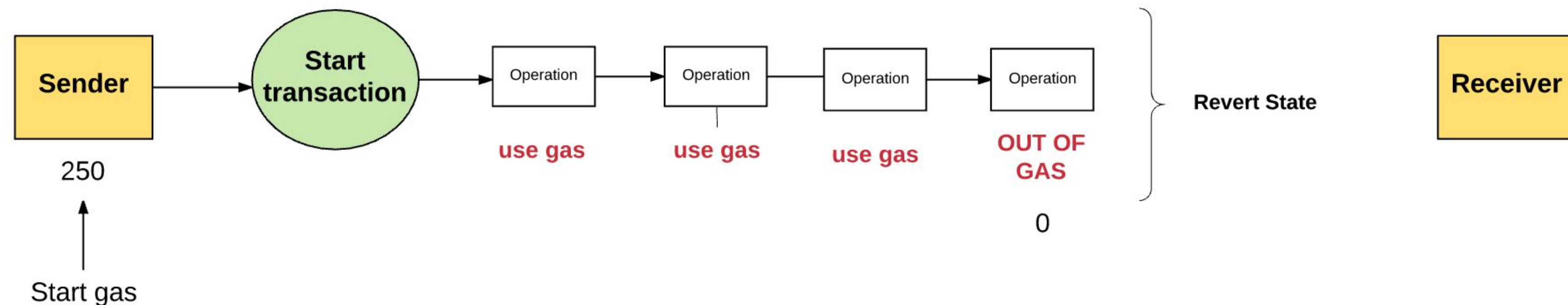




# GAS AND PAYMENT

## WHAT IF NOT ENOUGH IS SENT?

- Not enough gas to execute the transaction?
  - Transaction runs out of gas and therefore is considered invalid
  - State changes are reversed, failing transaction recorded
  - Since computation was already expended by the network, **none of the gas is refunded**



[Source](#)

AUTHOR: AKASH KHOSLA

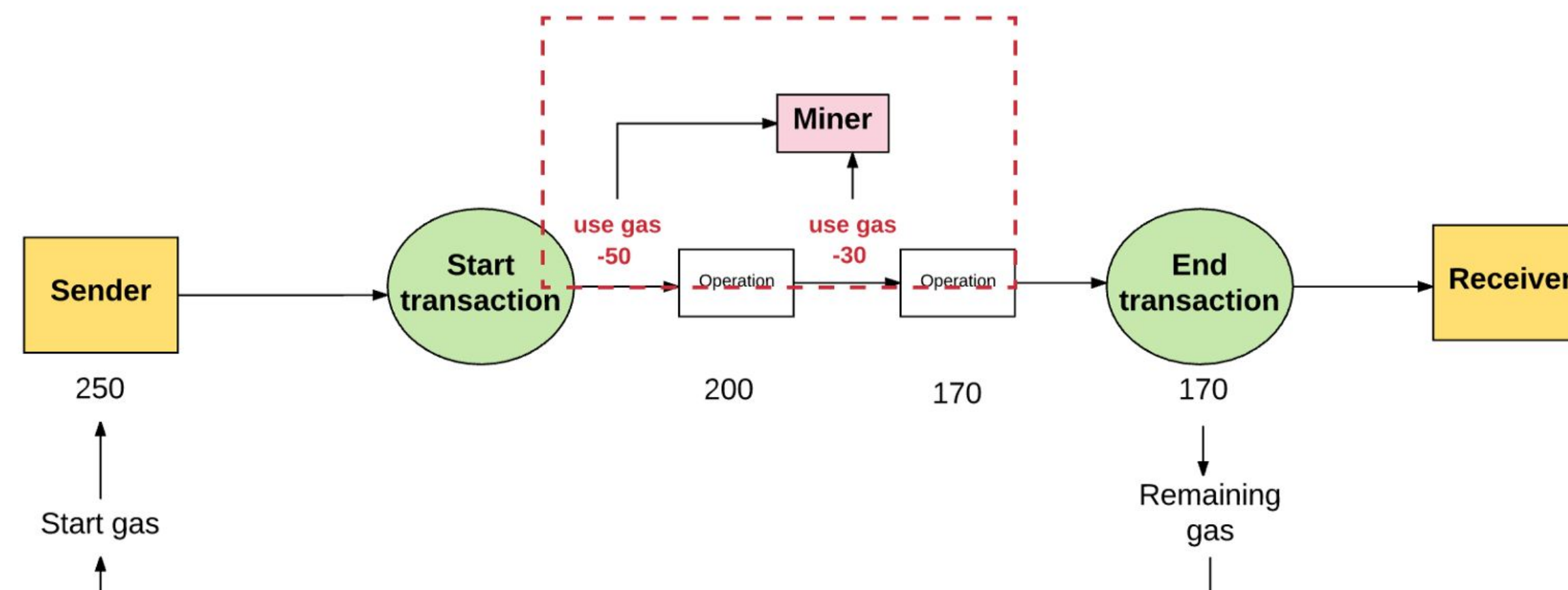
BLOCKCHAIN FOR DEVELOPERS



# GAS AND PAYMENT

## MINER INCENTIVES

- Where does the fee go?
  - **The miner's address**, since they are expanding the effort to run computations and validate transactions - it's the **reward**!
- The higher the gas price the sender is willing to pay, the greater the value the miner derives
  - Therefore more likely to select in block, as miners can choose which transactions to validate and ignore (gas price advertising also possible)



Source

AUTHOR: AKASH KHOSLA



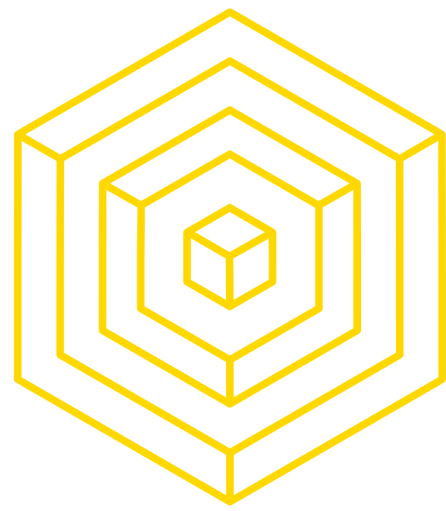
# STORAGE FEES

## NOT JUST FOR COMPUTATION

- **Gas** is also used to pay for **storage**, proportion to the smallest multiple of **32 bytes** used
- Increased storage however increases the size of the Ethereum state on all nodes
  - **Therefore incentives to keep data small**
- So if a transaction has a step that clears an entry in storage, the fee is waved and a **refund** is given for free storage space



BLOCKCHAIN FOR DEVELOPERS



# WHY DO WE NEED FEES?

## PARALLEL, DISTRIBUTED COMPUTATION

- Every single operation executed by the network is simultaneously affected by every full node
  - Computation steps on the EVM are therefore very expensive
- **Smart contracts are therefore best used for simple tasks**
  - Business logic or verifying signatures rather than machine learning or file storage
  - Redundantly parallel, no asynchronous or performant parallel execution
- **Turing completeness** allows for loops and susceptibility to the **halting problem**
  - **Halting problem:** inability to determine whether or not a program will run indefinitely
  - No fees means DDOS through infinite loops





# WHAT IS A TRANSACTION

## STATE CHANGERS

- Transactions move the state of an account within the global state - one state to the next
- **Formal Definition:** A transaction is a cryptographically signed piece of instruction that is generated by an externally owned account, serialized, and then submitted to a blockchain
- Two types: **Message calls and contract creations**



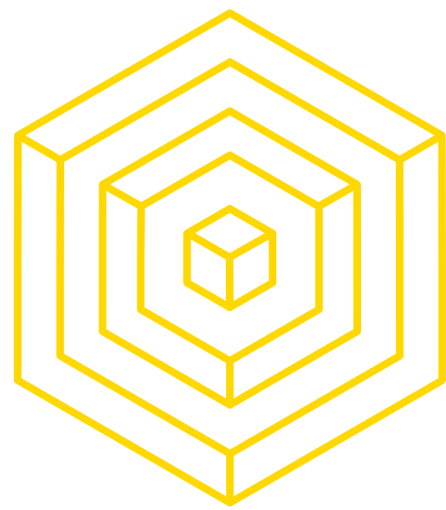




# TRANSACTION COMPONENTS

## STATE CHANGERS

- **nonce**: number of transactions sent by address of txn sender
  - **gasPrice**: amount of Wei sender is willing to pay per unit of gas required to execute the transaction
  - **gasLimit**: max amount of gas the sender is willing to pay for executing this transaction, set before any computation is done
  - **to**: address of the recipient
  - **value**: the amount of Wei to be transferred from the sender to the recipient
  - **v, r, s**: used to generate the signature that identifies the sender of the transaction.
- init** (only exists for contract-creating transactions): An EVM code fragment that is used to initialize the new contract account
- **data** (optional field that only exists for message calls): the input data (i.e. parameters) of the message call



# MESSAGE CALLS

## TRANSACTIONS

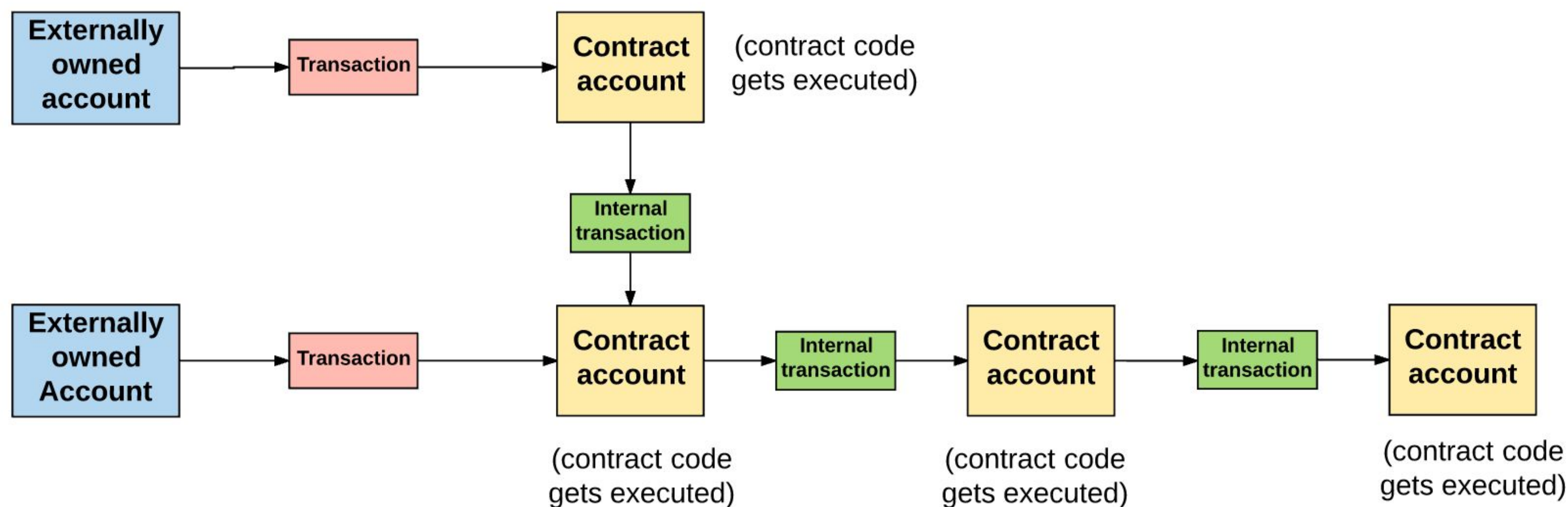
- Both **message calls** and **contract creating** transactions are always initiated by **externally owned accounts**
  - Transactions bridge the external world to the internal state of Ethereum
- Contracts that exist within the global scope of Ethereum can talk to other contracts using **messages** (internal transactions) to other contracts
- We can think of messages as being similar to transactions, except they are not generated by externally owned accounts, only by contracts
  - Virtual objects within the Ethereum execution environment



# NESTED MESSAGE CALLS

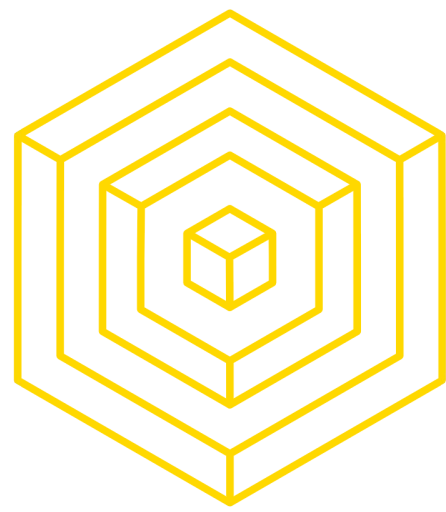
## TRANSACTIONS

When one contract sends an internal transaction to another contract, the associated code that exists on the recipient contract account is executed.



[Source](#)

AUTHOR: AKASH KHOSLA



# NESTED MESSAGES

## INTERNAL TRANSACTIONS

- Messages do not contain a **gasLimit**
- **gasLimit** determined by the external creator of the original transaction
  - Therefore the **gasLimit** that the externally owned account sets must be high enough to carry out the transaction and any other sub executions that occur as a result of that transaction
  - i.e. Contract-to-Contract messages
- **What if we run out of gas within a parent execution?**
  - Current and subsequent message executions will revert, however the parent execution need not revert

2

# CONSENSUS

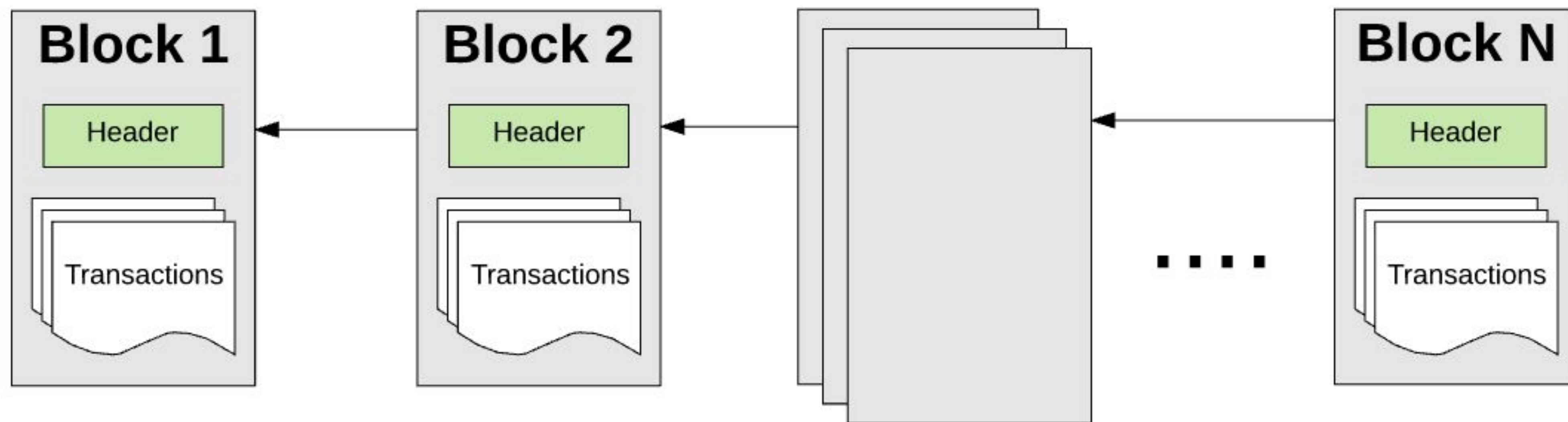




# WHAT DOES CONSENSUS LOOK LIKE

## INTERNAL TRANSACTIONS

In a blockchain, there's this idea of global truth, and everyone accepts this in order to make progress





# WHAT DOES CONSENSUS LOOK LIKE

## INTERNAL TRANSACTIONS

This truth is agreed upon via mining and peer validation

$H(\text{block\_header} \mid \text{nonce}) \leq \text{target (difficulty)}$

`0x0000041261ef648cfac61309685bdcd`

`<= 0x0000ffffffffffffffffffffffffffffffff`

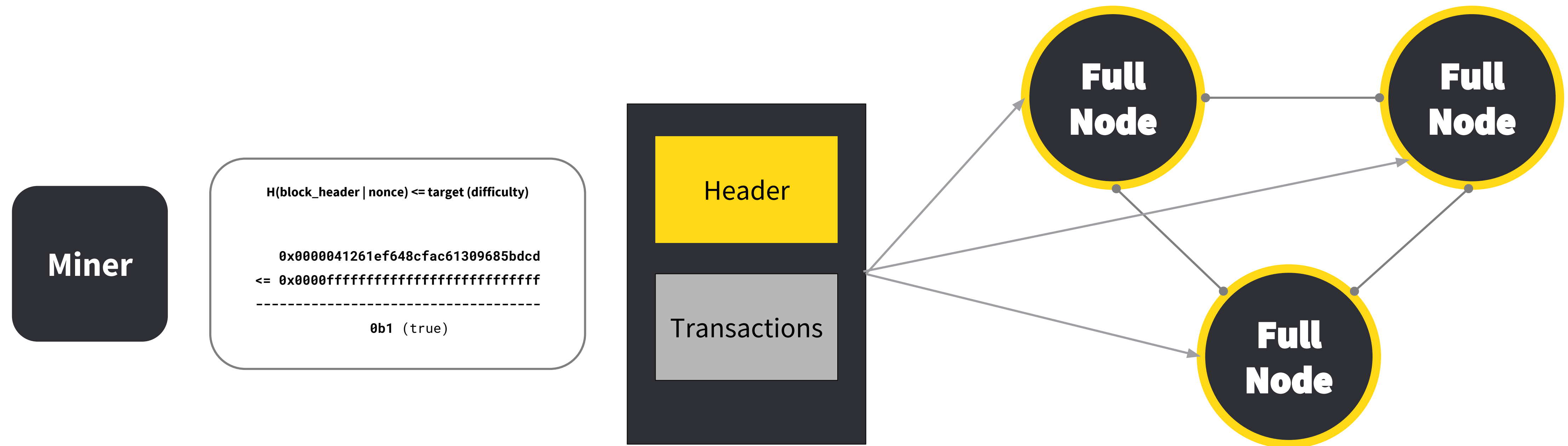
-----

`0b1 (true)`



# WHAT DOES CONSENSUS LOOK LIKE

## INTERNAL TRANSACTIONS





# HOW BITCOIN WORKS

## STATE TRANSITION PERSPECTIVE

- Bitcoin operates on running client software that agrees on state transitions
- **State:** the ownership status of all existing bitcoins
- **State Transition Function:** takes a state and transaction and outputs a new result
- $\text{APPLY}(S, TX) \rightarrow S'$

`/* Bitcoin blockchain update algorithm */`

**def APPLY(state, TX):**

For each input in TX: `# Input includes UTX0 and signature`

`if input.UTX0 not in state: return ERROR # referenced UTX0 not in state`

`# provided signature doesn't match the owner of the UTX0`

`if input.signature != lookup_UTX0(input).signature: return ERROR`

`if sum(input UTX0s) < sum(output UTX0s): return ERROR`

`S.update # Remove input UTX0s and add all output UTX0s`



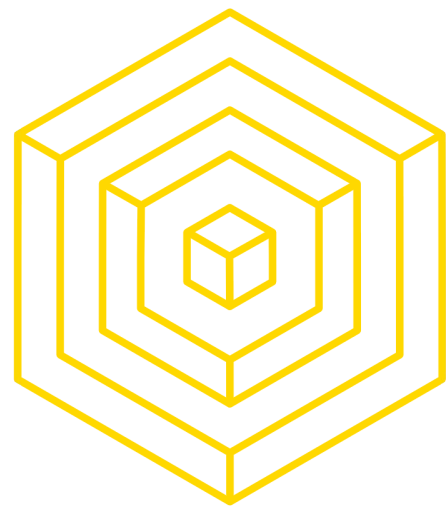
# HOW BTC BLOCK VALIDATION WORKS

## STATE TRANSITION PERSPECTIVE

- The previous algorithm works perfect for a centralized currency system
- To make a decentralized currency system, need **consensus** and **state transition** systems
- Produce a package of transactions every 10 minutes called **blocks**.

```
def VALIDATE_BLOCK(block):
    is_valid(block.prev)
    assert(block.timestamp > block.prev.timestamp)
    # less than 2 hours (120 min) into the future
    assert(block.timestamp < block.prev.timestamp + 120)
    check_pow(block) # valid Proof of Work, used for consensus
    TX_list = block.prev.transactions
    for i in range(len(TX_list)):
        # S[0] is the state, block.prev.S[0] end of the prev block state
        block.S[i] = APPLY(block.prev.S[i], TX_list[i]) # return False on error
    return True
```





# HOW ETHEREUM WORKS

## STATE TRANSITION PERSPECTIVE

`/* Ethereum blockchain update algorithm */`

**def APPLY(state, TX):**

`if not TX.is_valid(): return ERROR # check for well formed txn`

`if not TX.check_sig(): return ERROR # check for valid signature`

`if TX.sender.nonce != TX.nonce: return ERROR # acct nonce == txn nonce`

`fee = txn.gas * txn.gasprice`

`decrement(tx.sender.balance, fee) # Returns error if not enough balance`

`tx.sender.nonce += 1`

`gas = txn.gas - (tx.bytes * miner.gas_per_byte)`

`transfer(txn.value, txn.sender, txn.receiver)`

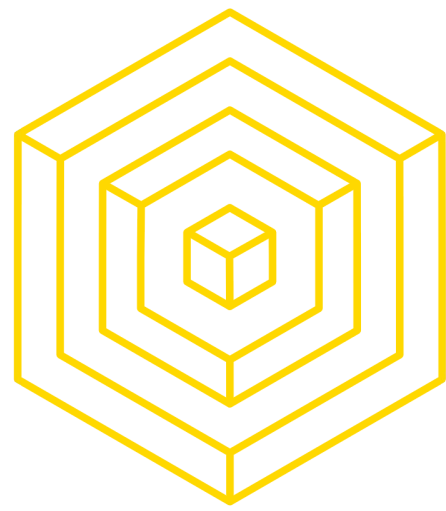
`"""`

`transfer: create receiver if non existant. if receiver is contract, run the code to completion or until gas is out.`

`if failed because not enough money or ran out of gas, revert all state except fees and add the fees to miner`

`account. If success, refund fees for all remaining gas to the sender and send fees paid for gas consumed to miner.`

`"""`



# ETHEREUM BLOCK VALIDATION

## BLOCK VALIDATION FOR FINALITY

```
def VALIDATE_BLOCK(block):
    is_valid(block.prev)
    assert(block.timestamp > block.prev.timestamp)
    # less than 15 min into the future
    assert(block.timestamp < block.prev.timestamp + 15)
    is_valid(block.blk_num, block.difficulty, block.txn_root, block.uncle_root, block.gas)
    check_pow(block) # valid Proof of Work, used for consensus
    TX_list = block.prev.transactions
    for i in range(len(TX_list)):
        # S[0] is the state, block.prev.S[0] end of the prev block state
        block.S[i] = APPLY(block.prev.S[i], TX_list[i]) # return False on error

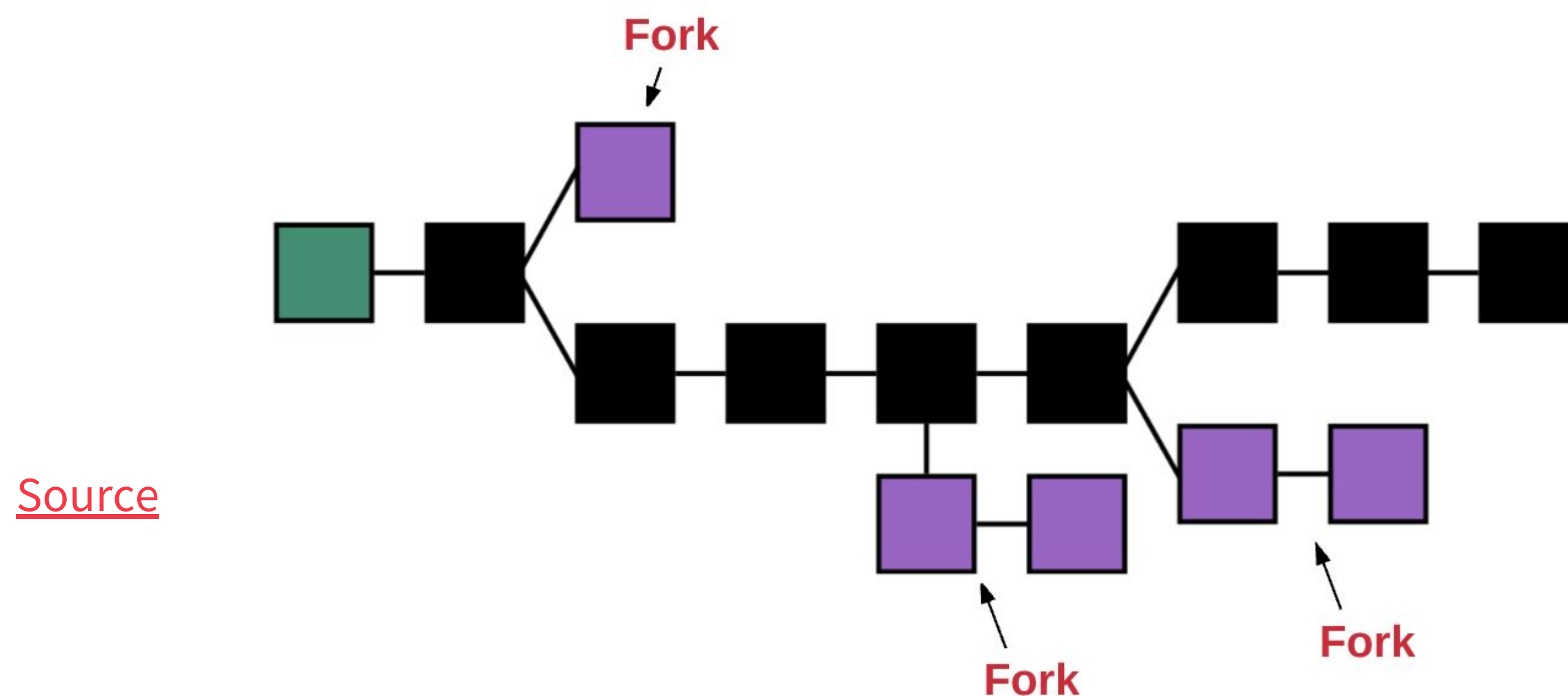
    if S[len(TX_list) - 1].merkle_root == block.txn_root:
        return True
    else: return False
```

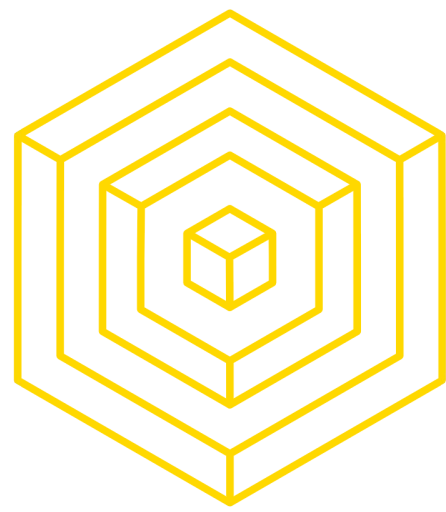


# FORK RESOLUTION

## INTERNAL TRANSACTIONS

- If a blockchain is global truth that every must agree on to make progress, then what happens when we have multiple blocks submitted at the same time, with different presentation
  - (i.e. they are valid blocks but transactions are ordered differently? Or maybe they submit a valid block using a completely different set of transactions)

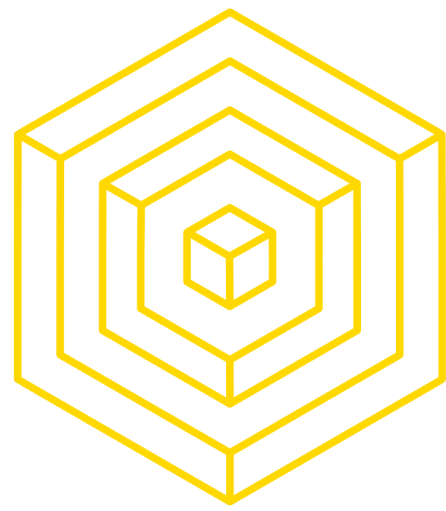




# FORK RESOLUTION

## INTERNAL TRANSACTIONS

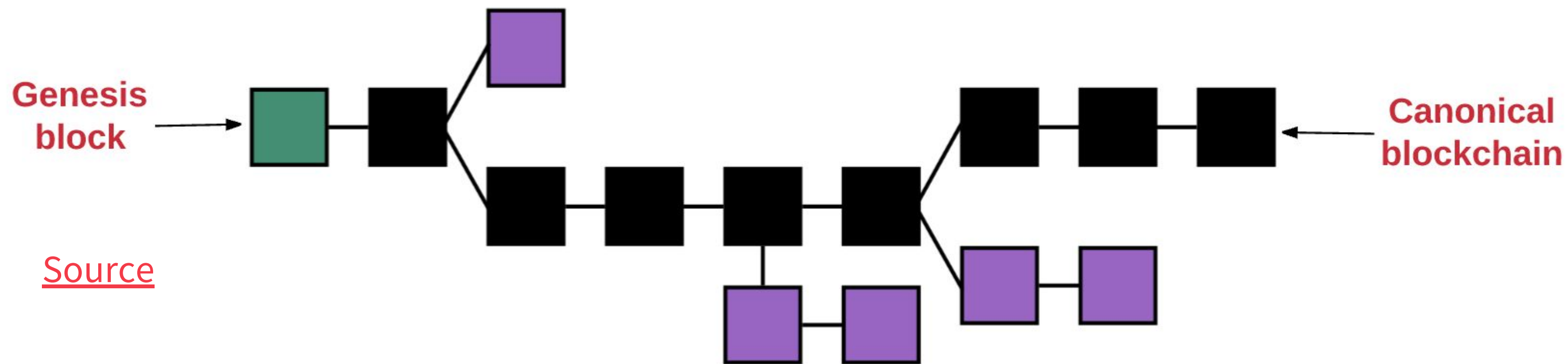
- If **block-time** is fast (12 seconds on Ethereum), it's not that hard to have multiple blocks that are submitted at the same time
- Having multiple states/chains destroys consensus
- A **fork** occurs as a result of multiple paths
- Forces people to choose a chain
- **How do we determine which blockchain is most valid?**



# GHOST PROTOCOL

GREEDY HEAVIEST OBSERVED SUBTREE

- We must be on the blockchain that has had the most computation done on it
- One way to determine that path is to use the **block number** of the most recent block (the “leaf” block) which represents the total number of blocks in the current path, except the genesis block
- The **higher the block number**, the longer the path, and therefore the **greater the mining effort** that must have gone into the block







# WORLD STATE

## ETHEREUM'S WAY OF STATE TRANSITIONS

- **World state** is a mapping of Addresses : Account states
- It is divided by **blocks**
  - **Every single block** is representative of an epoch where transactions that were broadcasted are then included in a block to then alter the world state
  - State is encoded with **RLP** and stored in a **Merkle Patricia Trie**
- The Merkle Patricia Trie is stored in a database that maintains a mapping of bytearrays (organized chunks of binary data) to bytearrays (merkle patricia trie data)
  - Relationships between each node in this trie constructs a mapping of Ethereum's state
- **State Database** is an off-chain data store on the computer running an Ethereum client
- We come to **consensus** on what we all **store**!



# RLP ENCODING

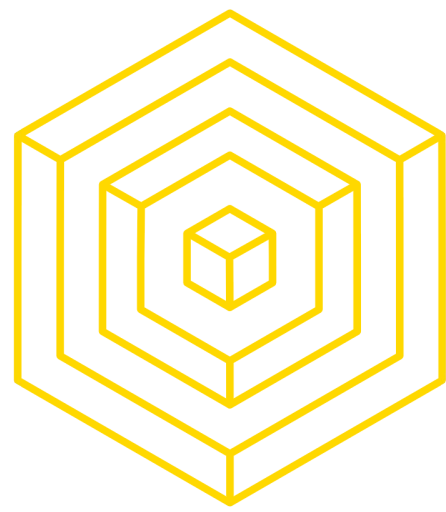
## HOW TO SERIALIZE DATA

- The purpose of **RLP (Recursive Length Prefix)** is to encode arbitrarily nested arrays of binary data, and RLP is the main encoding method used to serialize objects in Ethereum
- The only purpose of RLP is to encode structure; encoding specific data types (eg. strings, floats) is left up to higher-order protocol - there's also a way to decode data to use it
- Starts off with **byte value (0x80) + the length of the string**, and sometimes a **byte value (0xc0) + length of the list**
- Examples:
  - The string "dog" = [ 0x83, 'd', 'o', 'g' ]
  - The list [ "cat", "dog" ] = [ 0xc8, 0x83, 'c', 'a', 't', 0x83, 'd', 'o', 'g' ]

Worth reading the spec: <https://github.com/ethereum/wiki/wiki/RLP>

3

# BLOCK-BY-BLOCK



# BLOCKS

IS THAT THE STUFF YOU CHAIN TOGETHER

- Blocks are the way we keep track of world state
- “Chained” together by means of a **parentHash**, **each block has a reference to its parent**
- Blocks contain:
  - Block Header
  - Set of transactions
  - A set of other block headers for the current block’s **uncles**
- **All of which are determined by the miners submitting the block**

# 3.1

## UNCLES/ OMMERS





# UNCLES = OMMERS

## PART OF GHOST PROTOCOL

- A **block** whose **parent** is equal to the **current block's parent's parent**
  - This is because you can only specify omners after a block is submitted to the main-chain
- **Rationale:**
  - With lower block times, we get faster transaction processing
  - However competing block solutions are found by miners more often
  - These competing blocks that don't make it into the main blockchain, are orphaned blocks
  - Uncles help reward miners for including these orphaned blocks



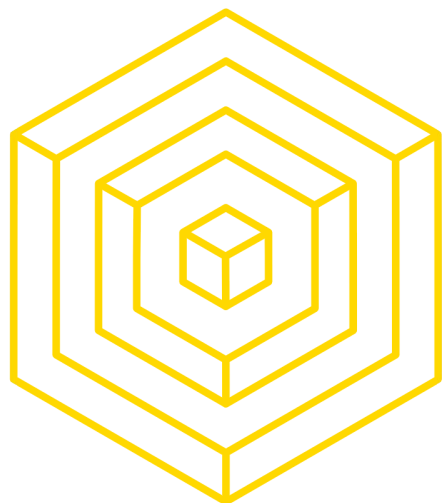
# UNCLES = OMMERS

## PART OF GHOST PROTOCOL

- Valid **uncles** are within the sixth generation or smaller of the present block
- After six children, orphaned blocks are now stale, and can no longer be references
  - They receive a smaller reward than a full block, but there is incentive to include these and reap a reward
- The use of uncles on Ethereum increases:
  - **Decentralization** because it decreases the need for mining pools (where the reward for finding a block is distributed among a network of pooled miners)
  - **Security** by acknowledging the energy spent creating the uncle blocks

# 3.2

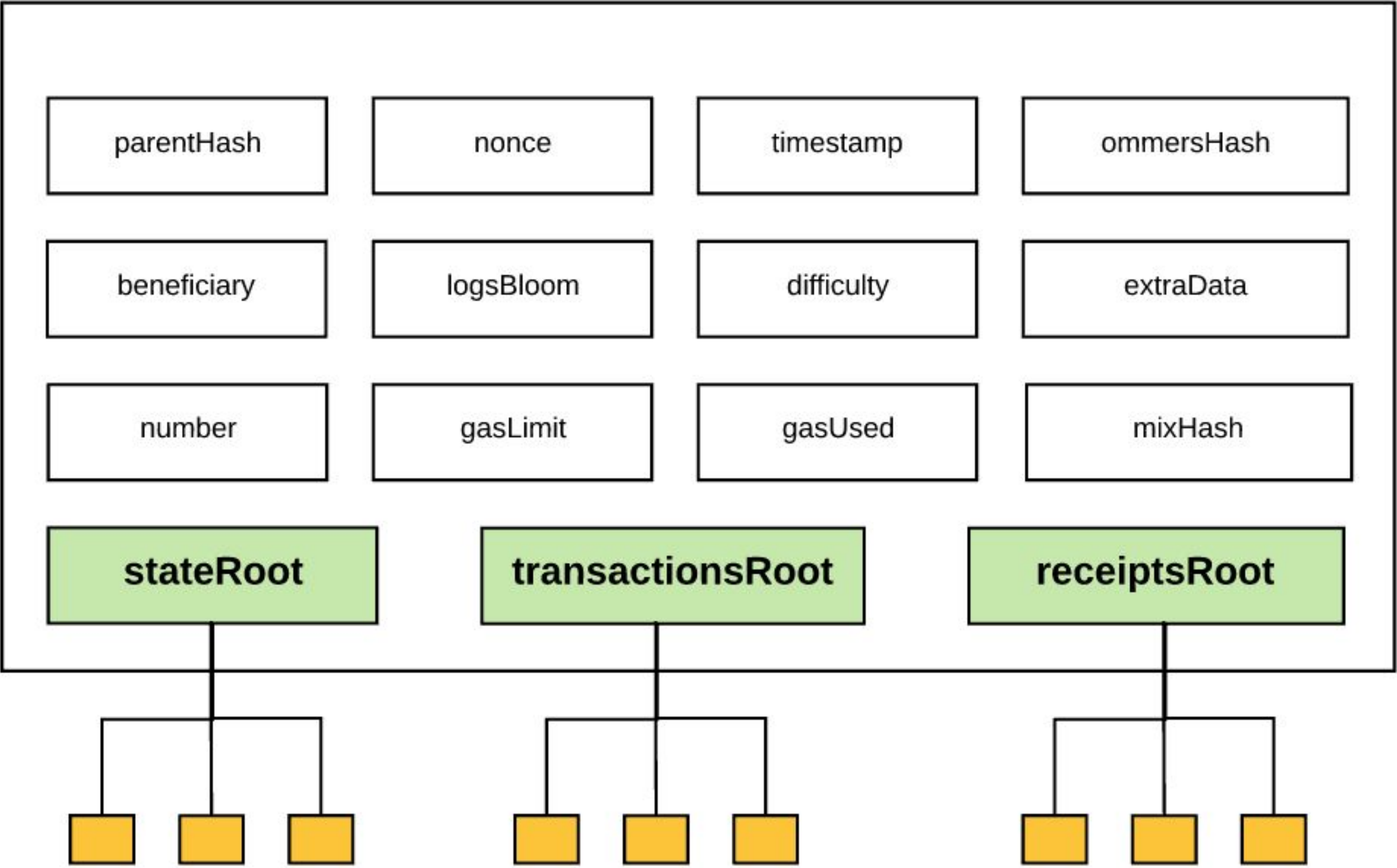
## BLOCK HEADER



# BLOCK HEADER

THE MAGIC BEHIND THE MADNESS

Block header



Source



# BLOCK HEADER

THE MAGIC BEHIND THE MADNESS

- **parentHash**: a hash of the parent's block header
- **ommerHash** (or colloquially uncleHash): a hash of the current block's list of omners
- **Beneficiary**: account address that receives the fees for mining this block
- **stateRoot**: the hash of the root node of state trie
- **transactionsRoot**: the hash of the root node of the trie that contains all transactions listed in this block
- **receiptsRoot**: the hash of the root node of the trie that contains the receipts of all transactions listed in this block
- **logsBloom**: a Bloom filter (probablistic data structure) that consists of log information





# BLOCK HEADER

THE MAGIC BEHIND THE MADNESS

- **Difficulty**: the difficulty level of this block
- **Number**: the count of the current block (genesis is 0, incremented by 1 for each subsequent block)
- **gasLimit**: the current gas limit per block
- **gasUsed**: the sum of the total gas used by transactions in this block
- **timestamp**: the unix timestamp of this block's inception
- **extraData**: extra data related to this block
- **mixHash**: a hash that, when combined with the nonce, proves that this block has carried out enough computation
- **nonce**: a hash that, when combined with the mixHash, proves that this block has carried out enough computation

# 3.3

## ETHEREUM LOGS



# ETHEREUM LOGS

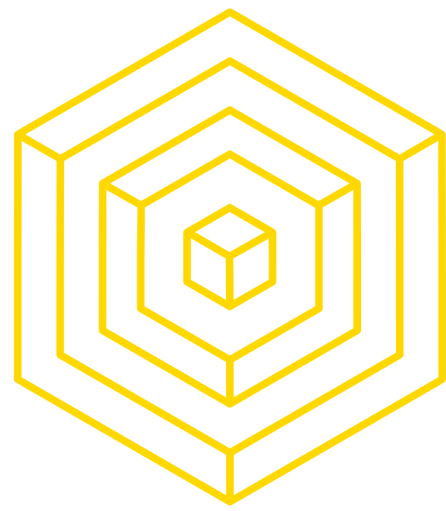
## EVENTS ARE BACK

46

### A few uses:

- Smart contract return values for the user interface
  - Return values of **sendTransaction** methods is the hash of the transaction that's created.
  - Transactions not immediately mined
- Triggers with data
  - Frontend can carry out actions based on event values
- A cheaper form of storage

Logs were designed to be a form of storage that costs significantly less gas than contract storage. Logs basically cost 8 gas per byte, whereas contract storage costs 20,000 gas per 32 bytes. Although logs offer gargantuan gas savings, logs are not accessible from any contracts. Nevertheless, there are use cases for using logs as cheap storage, instead of triggers for the frontend. A suitable example for logs is storing historical data that can be rendered by the frontend.



# ETHEREUM LOGS

EVENTS ARE BACK

47

- Logs stored in the **block header** come from the log information contained in the **transaction receipt**

## Contains:

- Account address of the logger
- Topics that represent various events carried about by this transaction
- Any data associated with events









# LOG EXAMPLE

## CREATE AND FIRE EVENT

This will create a low level EVM log entry with topics

- 0x6be15e8568869b1e100750dd5079151b32637268ec08d199b318b793181b8a7d (Keccak-256 hash of `PersonCreated(uint256, uint256)`)
- 0x36383cc9cfbf1dc87c78c2529ae2fcd4e3fc4e575e154b357ae3a8b2739113cf (Keccak-256 hash of `age`), value 26
- 0x048dd4d5794e69cea63353d940276ad61f89c65942226a2bb5bd352536892f82 (Keccak-256 hash of `height`), value 176

```
PersonCreated(uint indexed age,  
              uint indexed height);  
  
function foobar() {  
    PersonCreated(26, 176);  
}
```



# LOG EXAMPLE

## LISTEN FOR EVENT

50

```
var createdEvent = myContract.PersonCreated({age: 26});
createdEvent.watch(function(err, result) {
  if (err) {
    console.log(err)
    return;
  }
  console.log("Found ", result);
})
```

# 3.4

## TRANSACTION RECEIPTS



# TRANSACTIONS

## TRANSACTIONS VS TRANSACTION RECEIPTS

### TRANSACTIONS $\neq$ TRANSACTION RECEIPTS

- **Transactions** record transaction request vectors
- **Transaction Receipts** record the transaction outcome



# TRANSACTIONS

## TRANSACTIONS VS TRANSACTION RECEIPTS

### Transactions

- Nonce
- Gas price
- Gas limit
- Recipient
- Transfer value
- Transaction signature values
- Account initialization (if transaction is of contract creation type), or transaction data (if transaction is a message call)

### Transaction Receipts

- Post-transaction state
- The cumulative gas used
- The set of logs created through execution of the transaction
- The Bloom filter composed from information in those logs



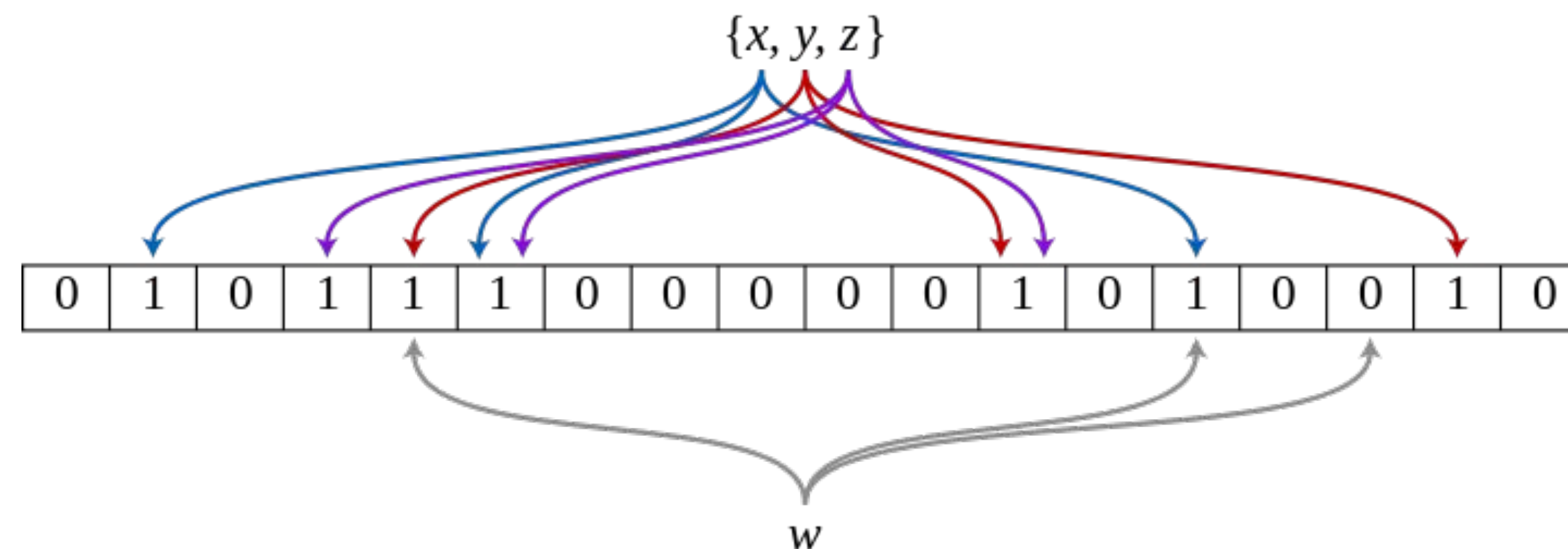
# 3.5

## BLOOM FILTERS



# BLOOM FILTERS

## THE ALGORITHM



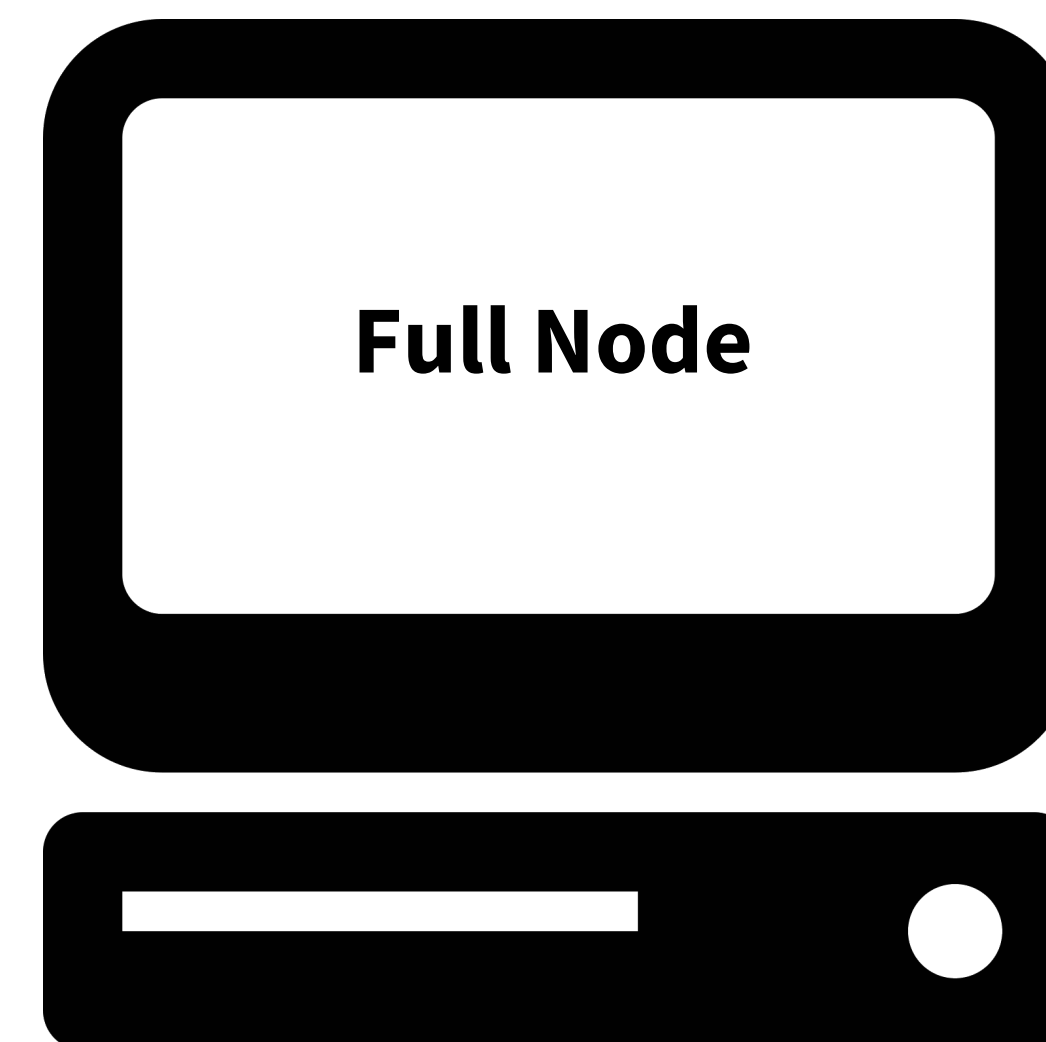
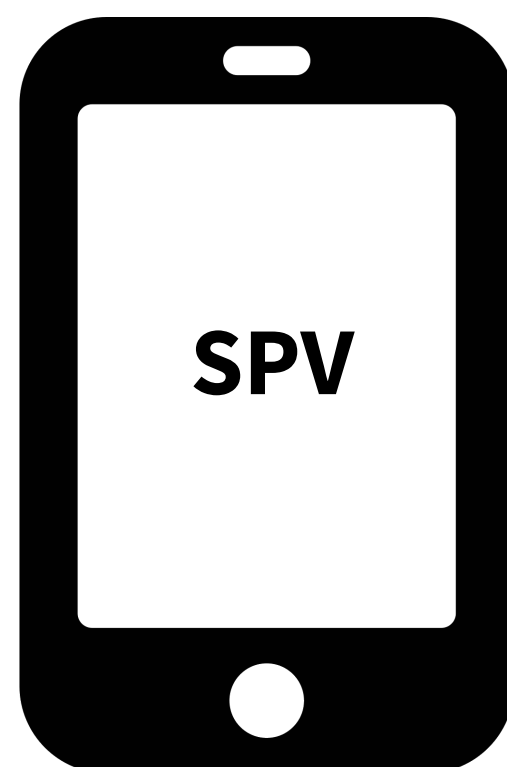
An example of a Bloom filter, representing the set  $\{x, y, z\}$ . The colored arrows show the positions in the bit array that each set element is mapped to. The element  $w$  is not in the set  $\{x, y, z\}$ , because it hashes to one bit-array position containing 0. For this figure,  **$m = 18$**  and  **$k = 3$** .



# BLOOM FILTERS

REDUCING WORK IN BITCOIN

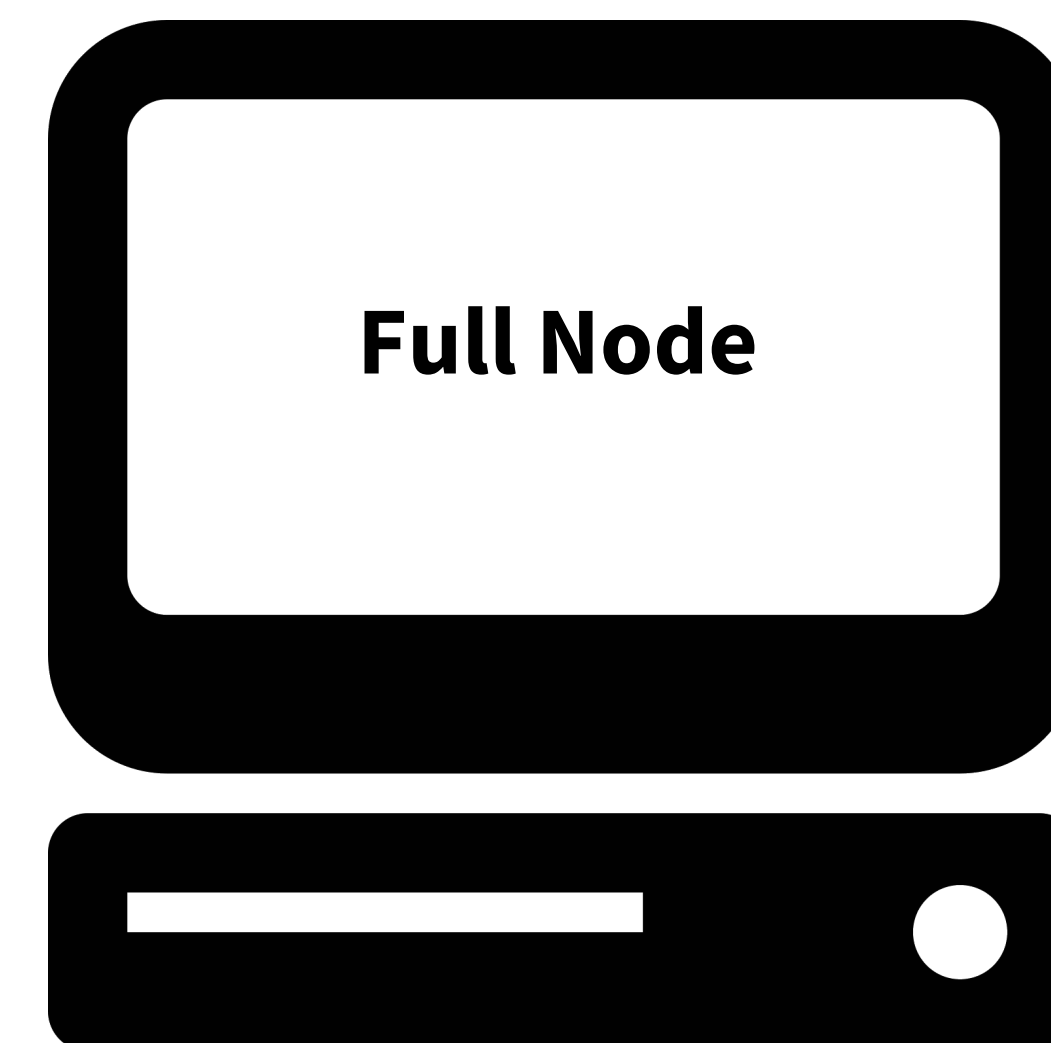
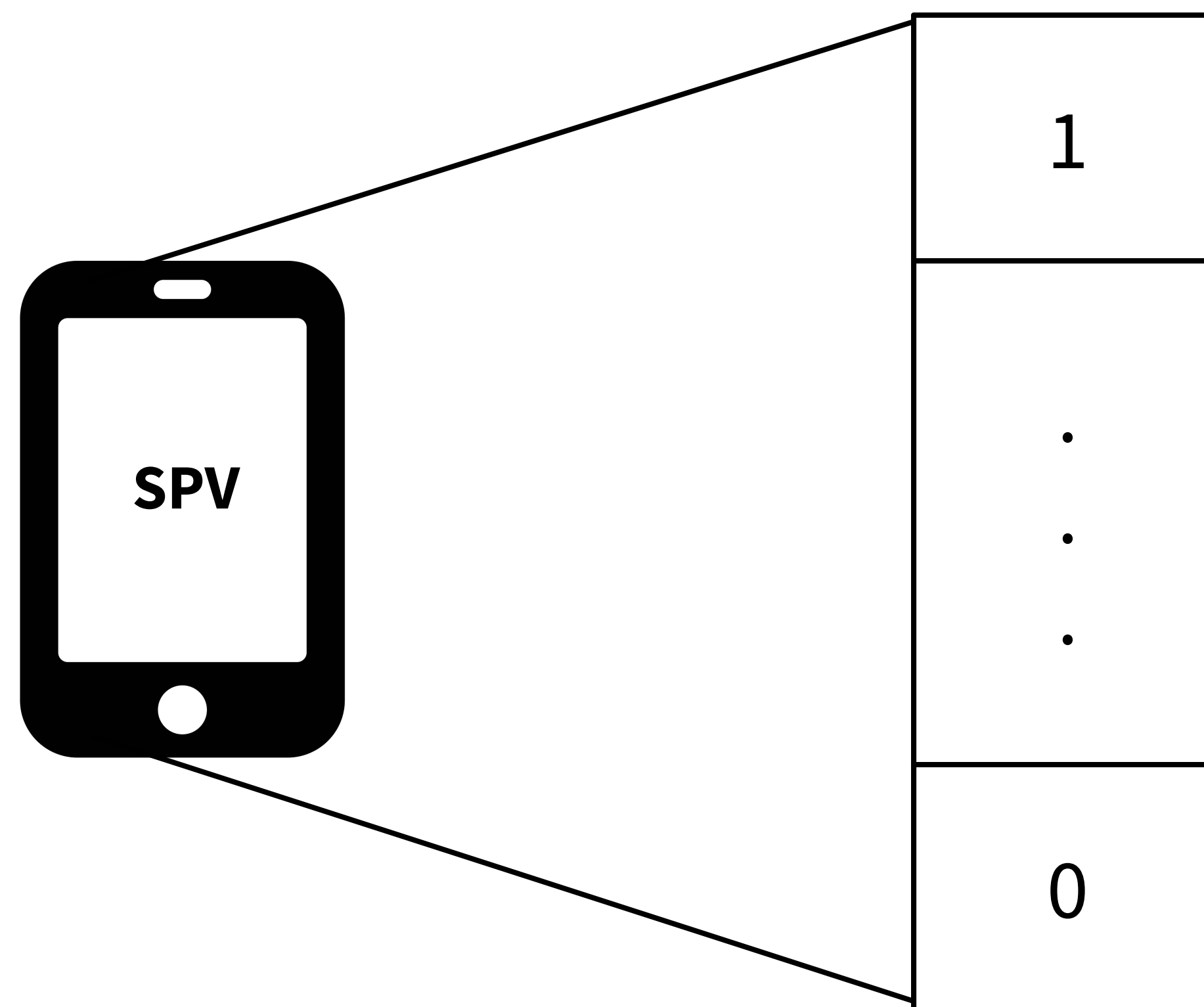
56





# BLOOM FILTERS

REDUCING WORK IN BITCOIN

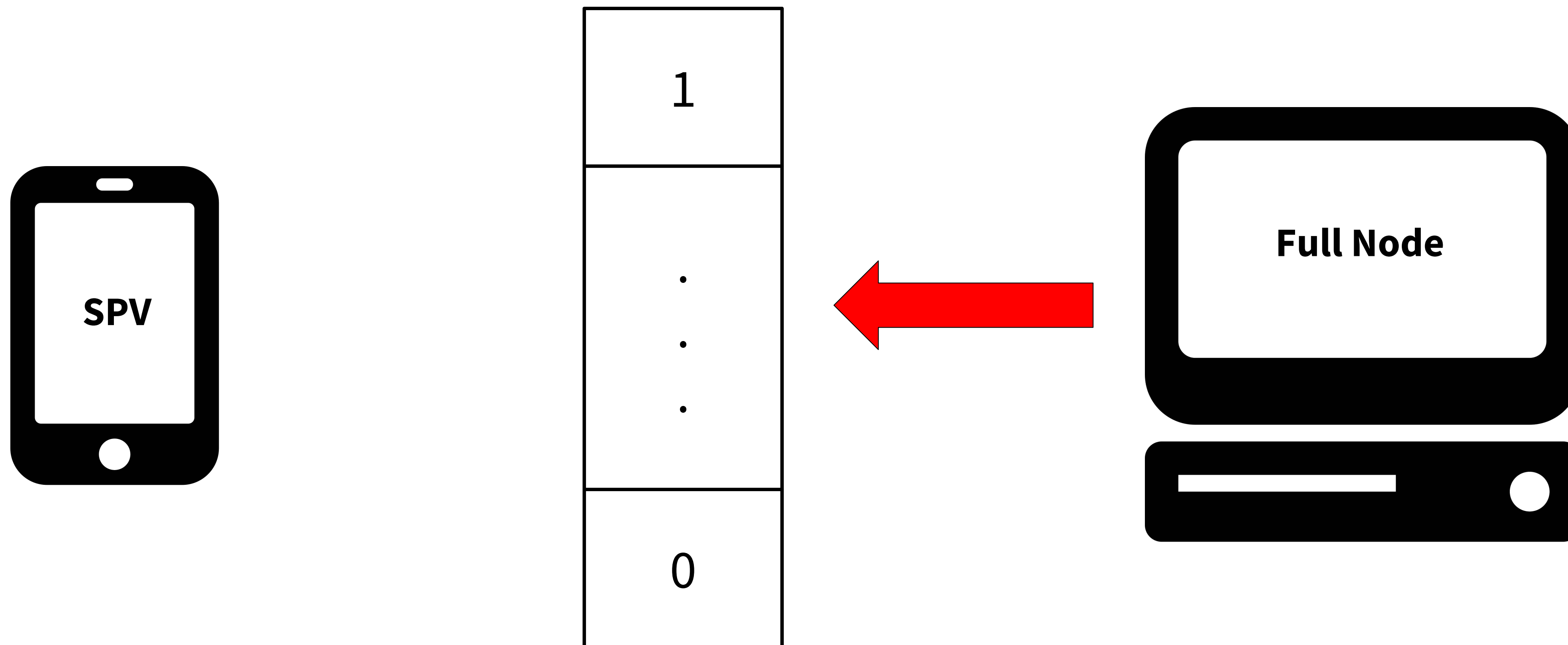




# BLOOM FILTERS

## REDUCING WORK IN BITCOIN

58



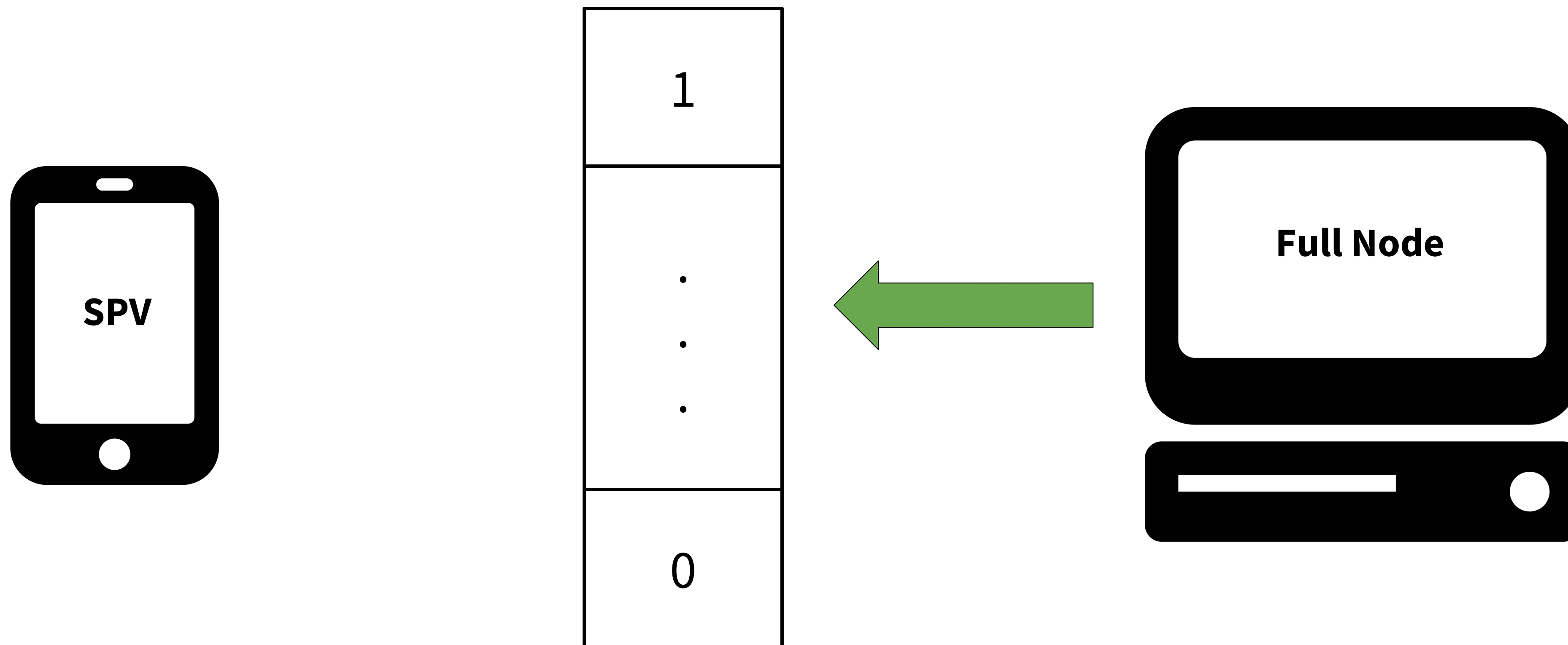




# BLOOM FILTERS

REDUCING WORK IN BITCOIN

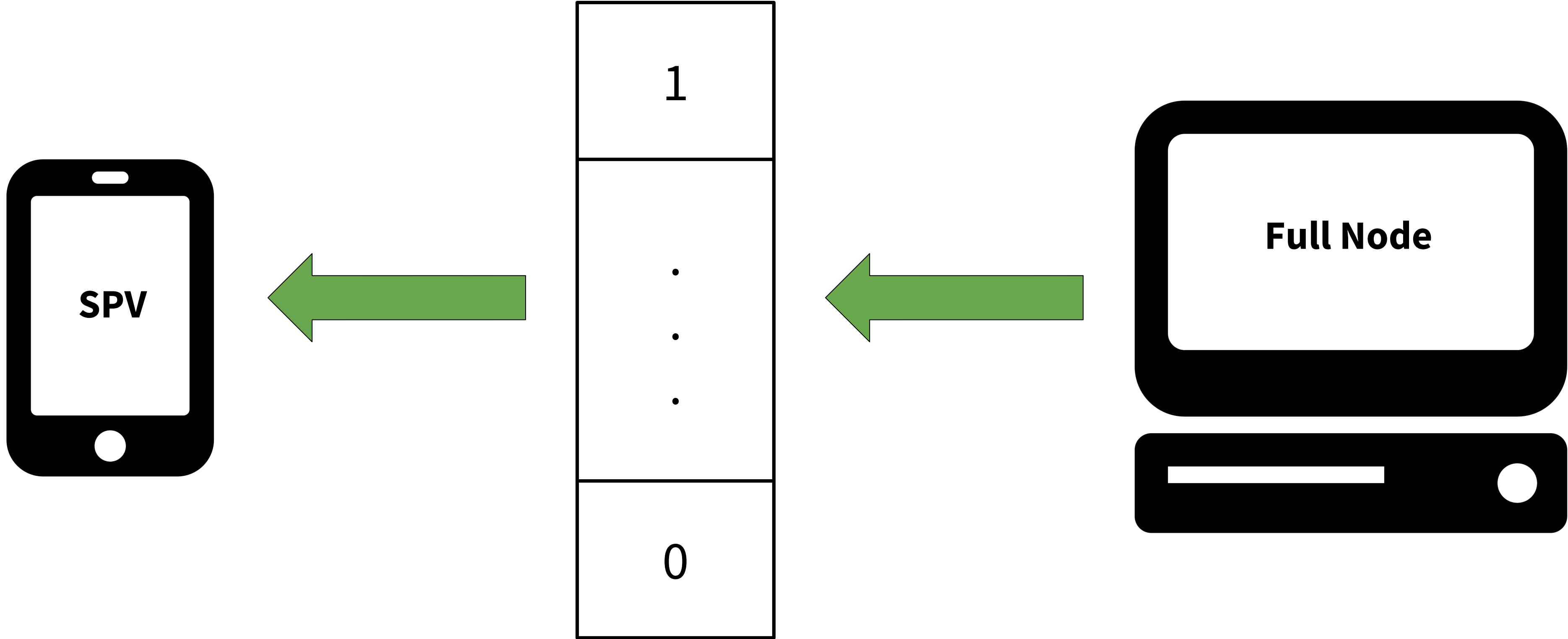
59





# BLOOM FILTERS

REDUCING WORK IN BITCOIN



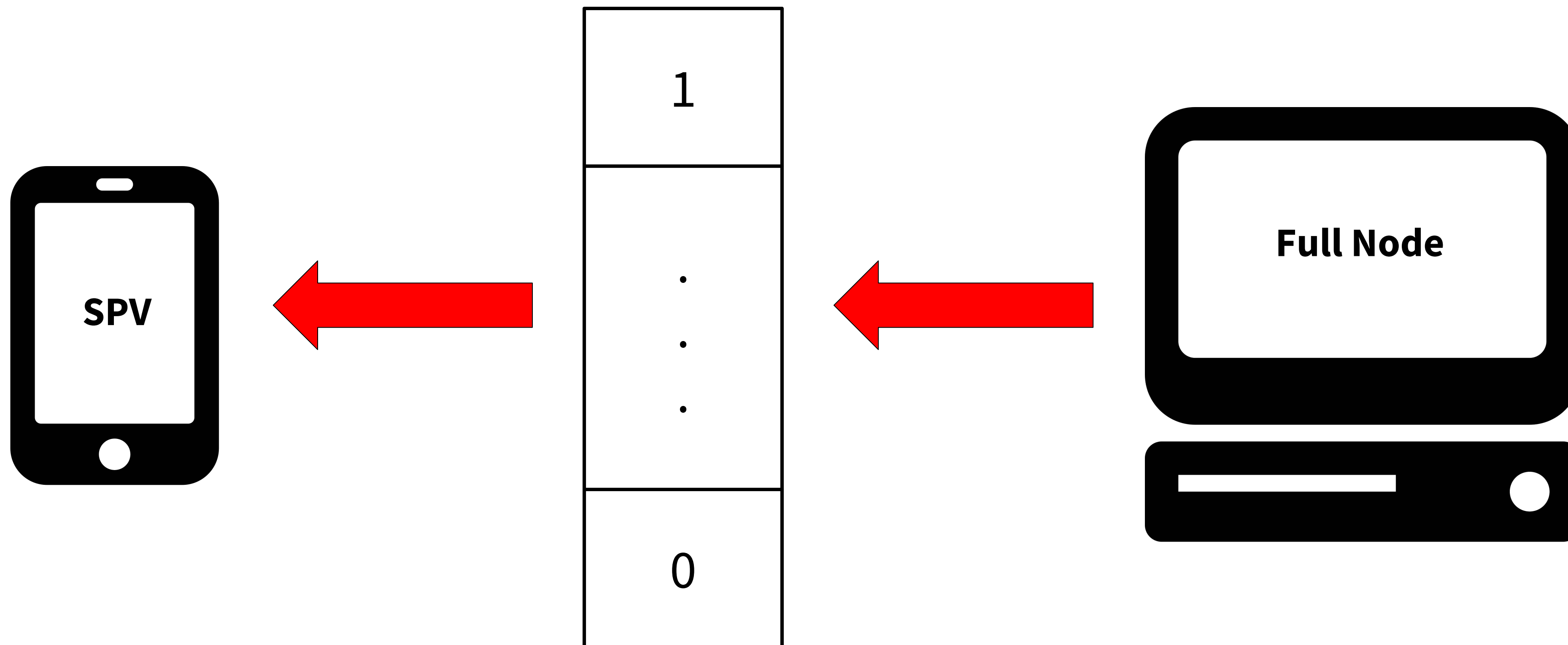


# BLOOM FILTERS

## REDUCING WORK IN BITCOIN

61

“False Positives” happen...



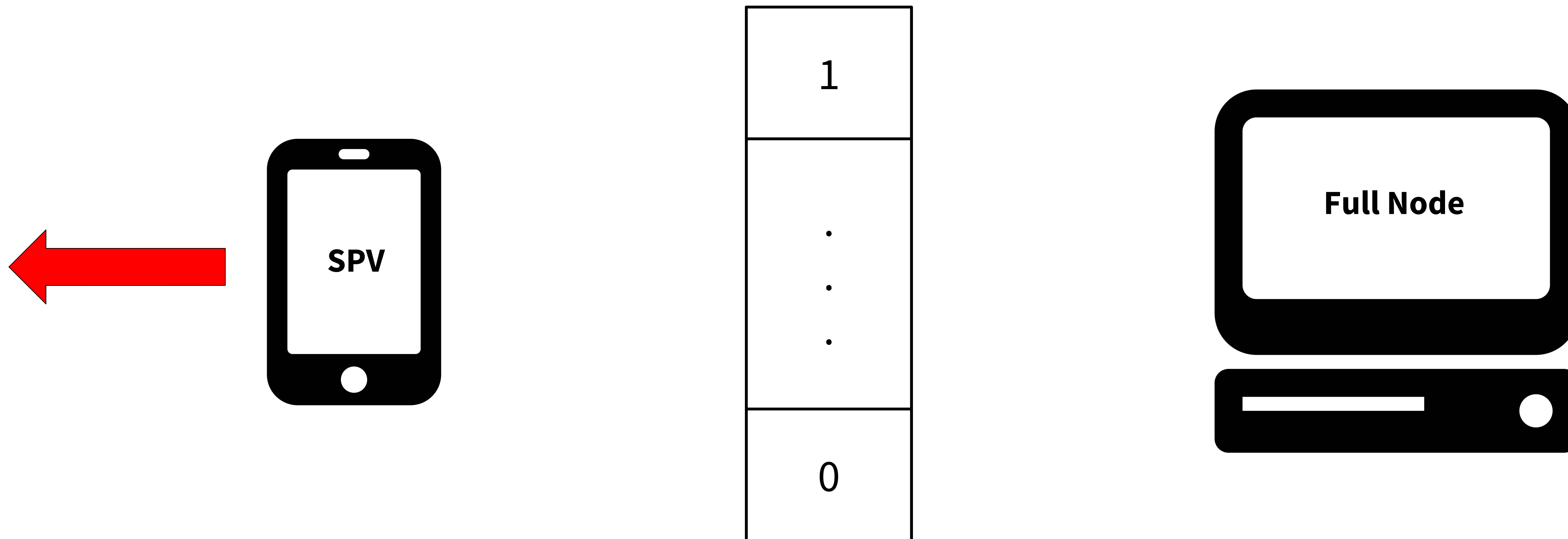


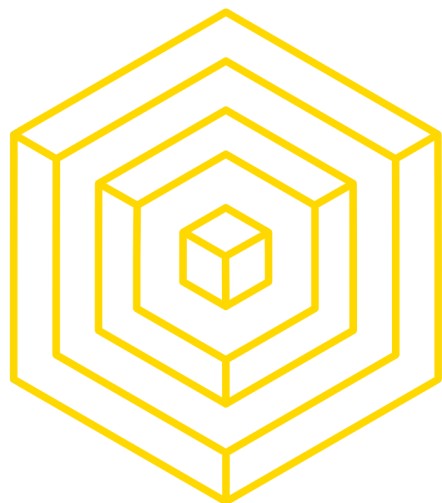
# BLOOM FILTERS

## REDUCING WORK IN BITCOIN

62

“False Positives” happen... and that’s ok!





# BLOOM FILTERS

REDUCING WORK IN ETHEREUM

tx receipt 1

tx receipt 2

tx receipt 3

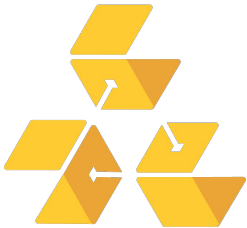
1
.
.
.
0

0
.
.
.
1

1
.
.
.
0

header-level bloom filter

1
.
.
.
1





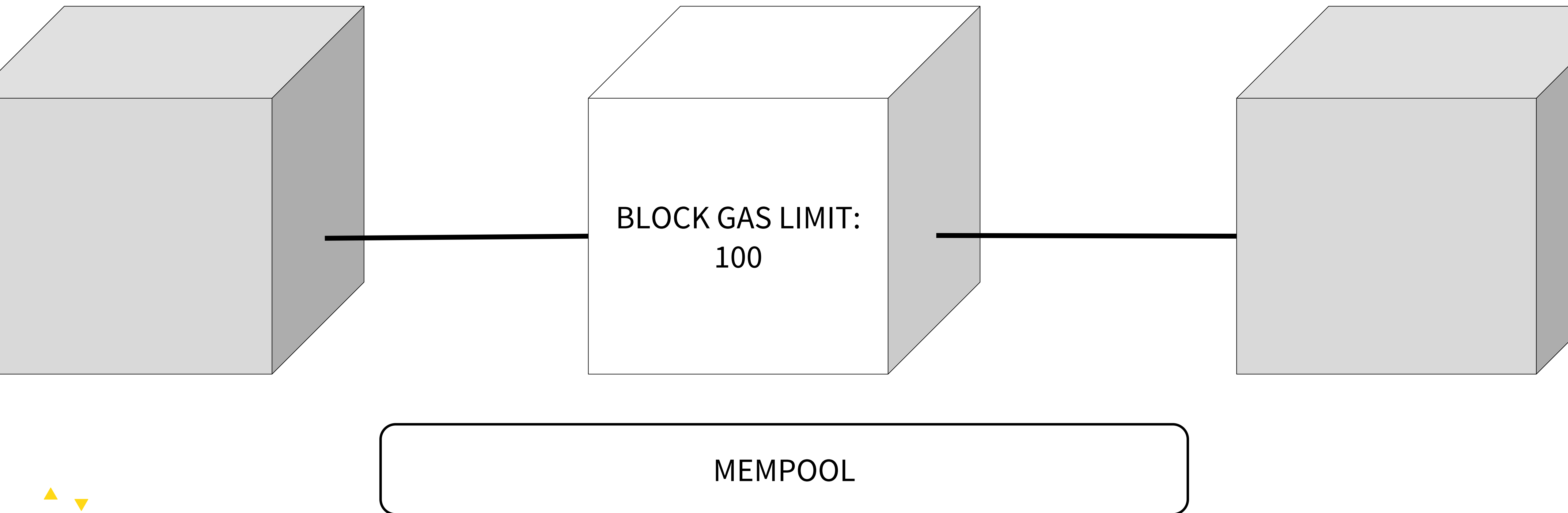
# 3.5

## **BLOCK ~~SIZE~~**



# BLOCK GAS LIMIT

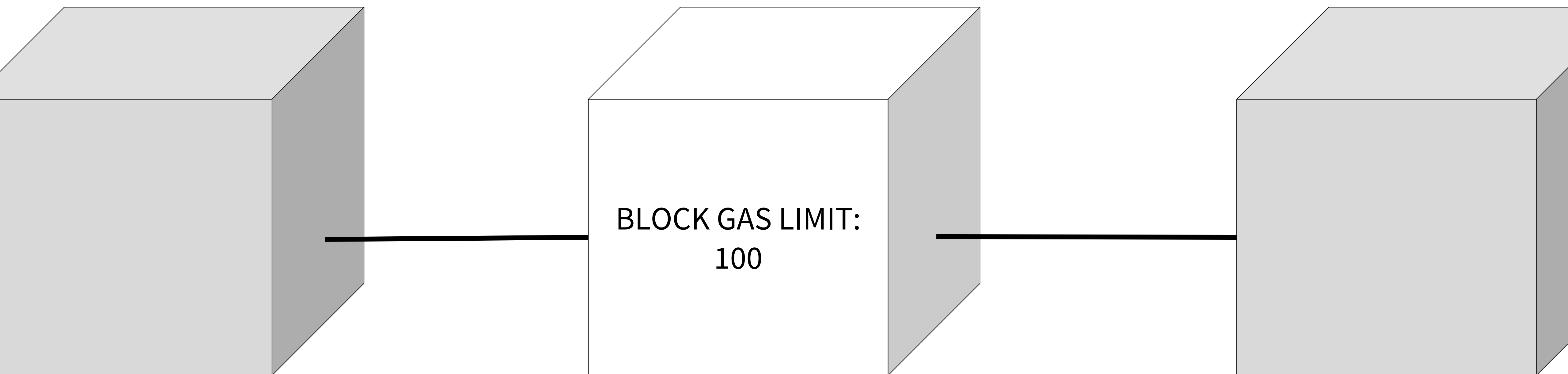
DEFENDING AGAINST DDOS





# BLOCK GAS LIMIT

## CHOOSING YOUR TRANSACTIONS

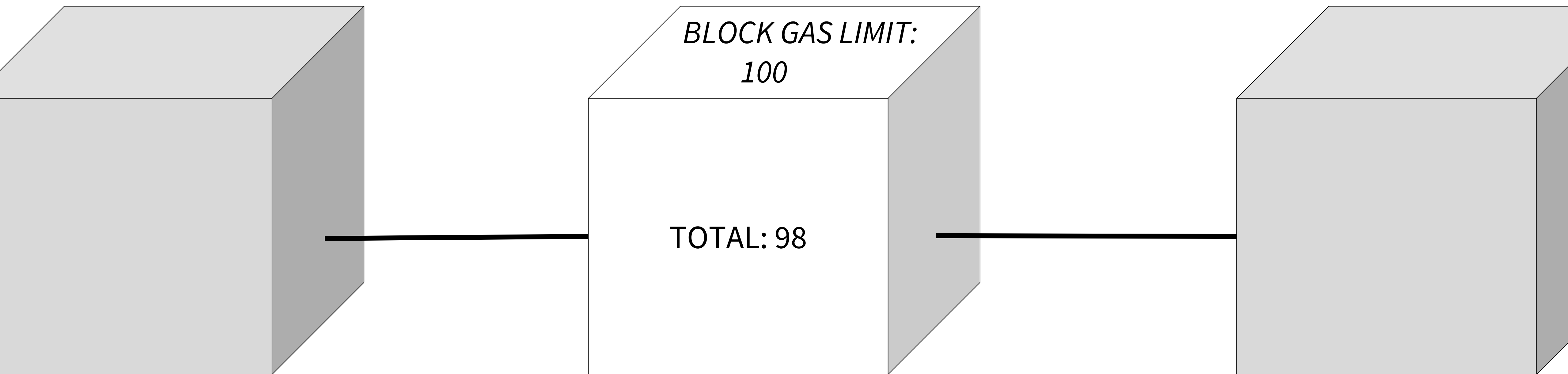


`{tx_id: 0, gas: 33}, {tx_id: 1, gas: 28}, {tx_id: 2, gas: 86},  
{tx_id: 3, gas: 25}, {tx_id: 4, gas: 12}, {tx_id: 5, gas: 40}`



# BLOCK GAS LIMIT

CHOOSING YOUR TRANSACTIONS

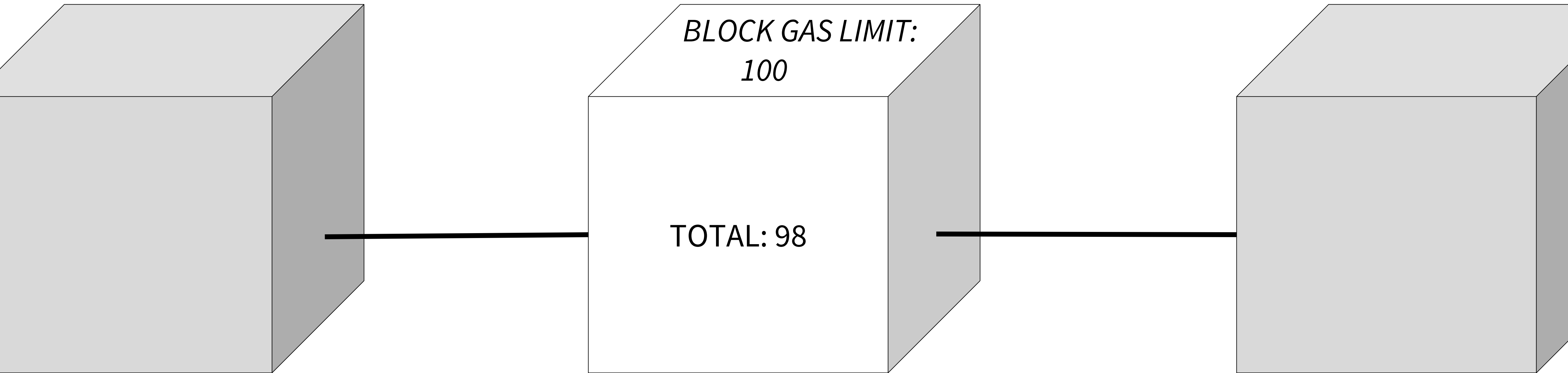


{tx\_id: 0, gas: 33}, {tx\_id: 1, gas: 28}, **{tx\_id: 2, gas: 86}**,  
{tx\_id: 3, gas: 25}, **{tx\_id: 4, gas: 12}**, {tx\_id: 5, gas: 40}



# BLOCK GAS LIMIT

CHOOSING YOUR TRANSACTIONS



**{tx\_id: 0, gas: 33}, {tx\_id: 1, gas: 28}, {tx\_id: 2, gas: 86},  
{tx\_id: 3, gas: 25}, {tx\_id: 4, gas: 12}, {tx\_id: 5, gas: 40}**

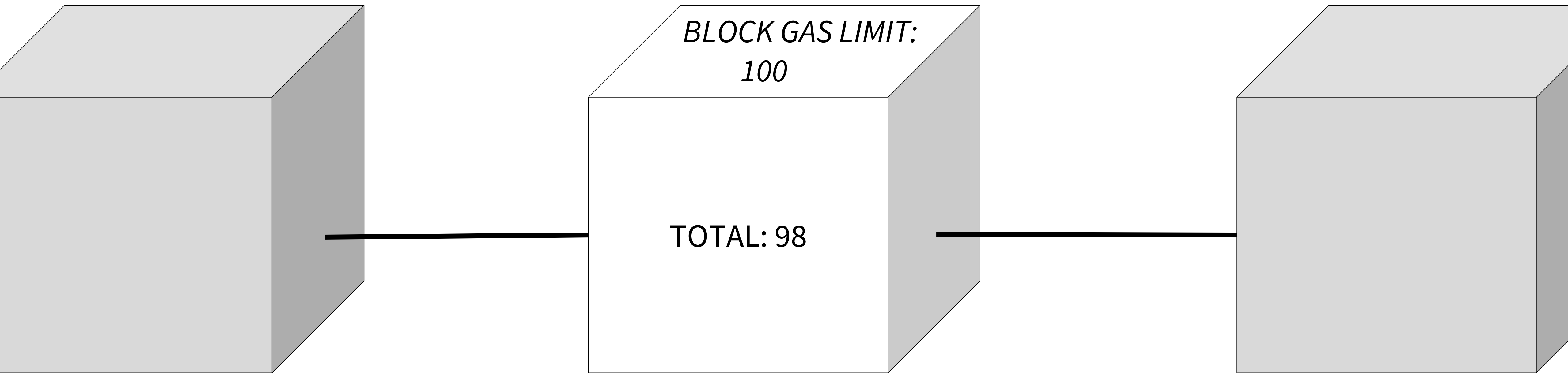




# BLOCK GAS LIMIT

## CHOOSING YOUR TRANSACTIONS

Which one would you pick?



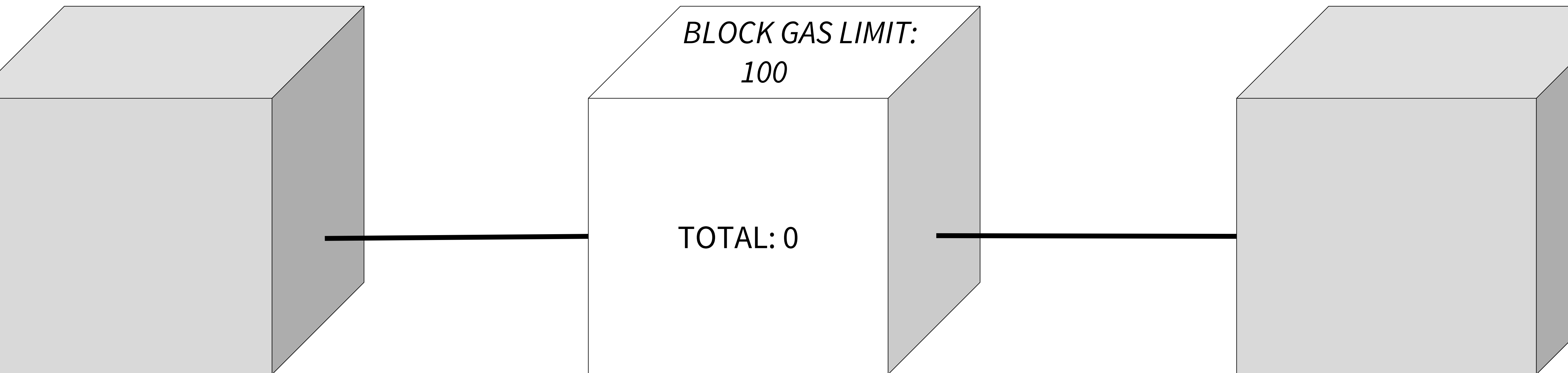
**{tx\_id: 0, gas: 33}, {tx\_id: 1, gas: 28}, {tx\_id: 2, gas: 86},  
{tx\_id: 3, gas: 25}, {tx\_id: 4, gas: 12}, {tx\_id: 5, gas: 40}**



# BLOCK GAS LIMIT

## CHOOSING YOUR TRANSACTIONS

!!! Uh-Oh !!!



{tx\_id: 0, gas: 33}, {tx\_id: 1, gas: 28}, {tx\_id: 2, gas: 86},  
{tx\_id: 3, gas: 25}, {tx\_id: 4, gas: 12}, {tx\_id: 5, gas: 40}



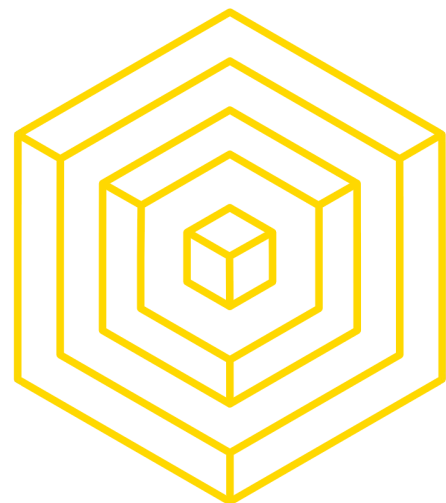
# BLOCK GAS LIMIT

## IT CHANGES

- The miner of a new block can set the block gas limit with  $\pm 0.1\%$  of the parent's block gas limit
- In a sense this works as vote, those wanting to increase the gas limit will submit blocks with an increased gas limit, and those who oppose will submit blocks with decreasing gas limit
  - We can say the group with more blocks resolved will decide the final outcome

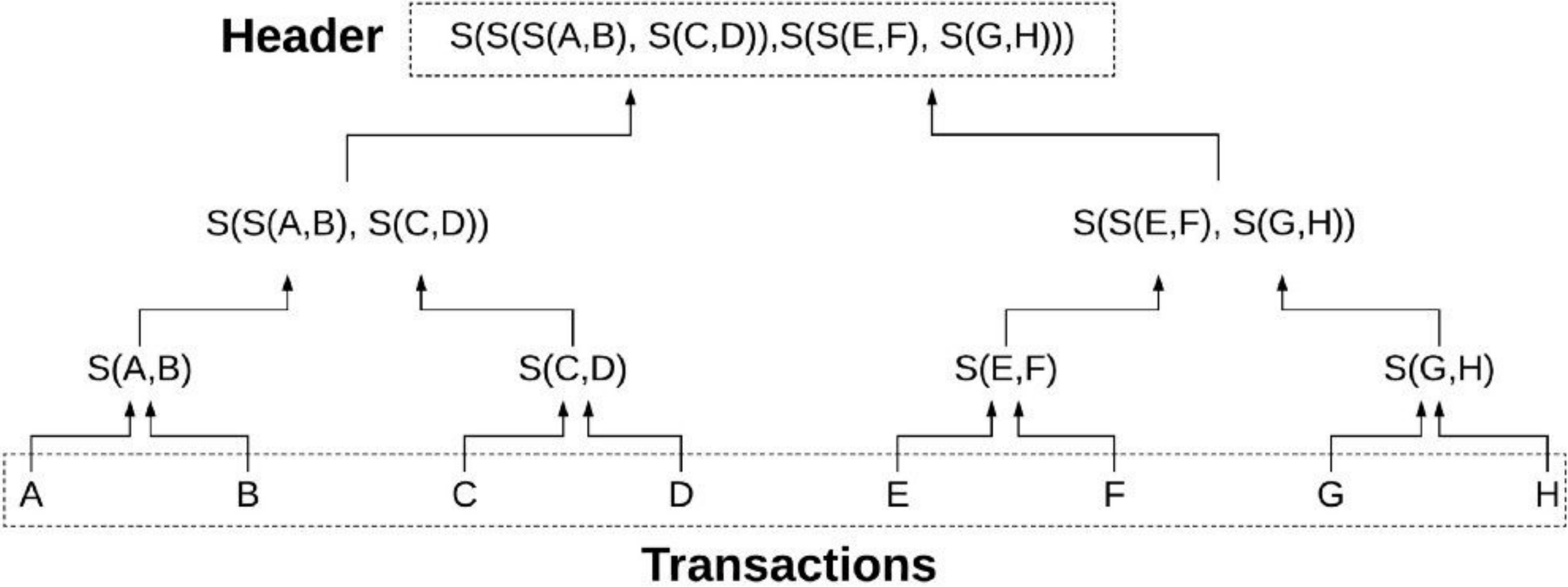
4

# HASH STRUCTURES



# MERKLE TREES

NICE AND SIMPLE

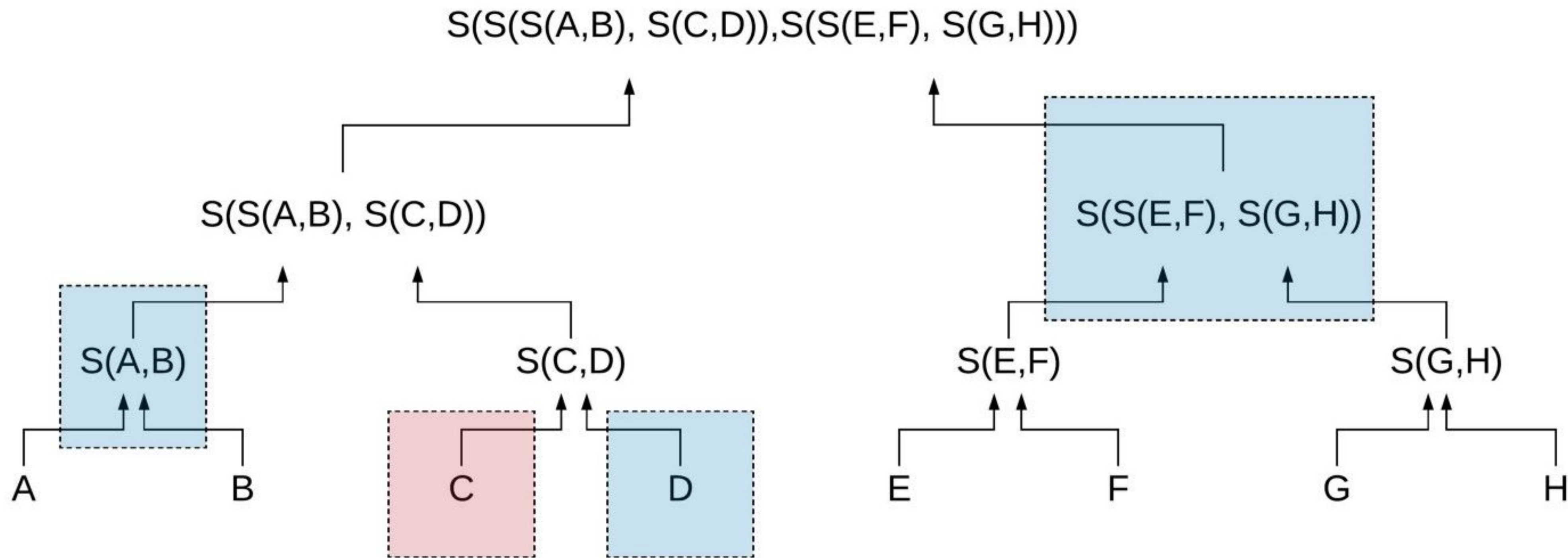






# MERKLE TREE PROOFS

NICE AND SIMPLE





# MERKLE TREES

## BETTER THAN REGULAR HASHING

75

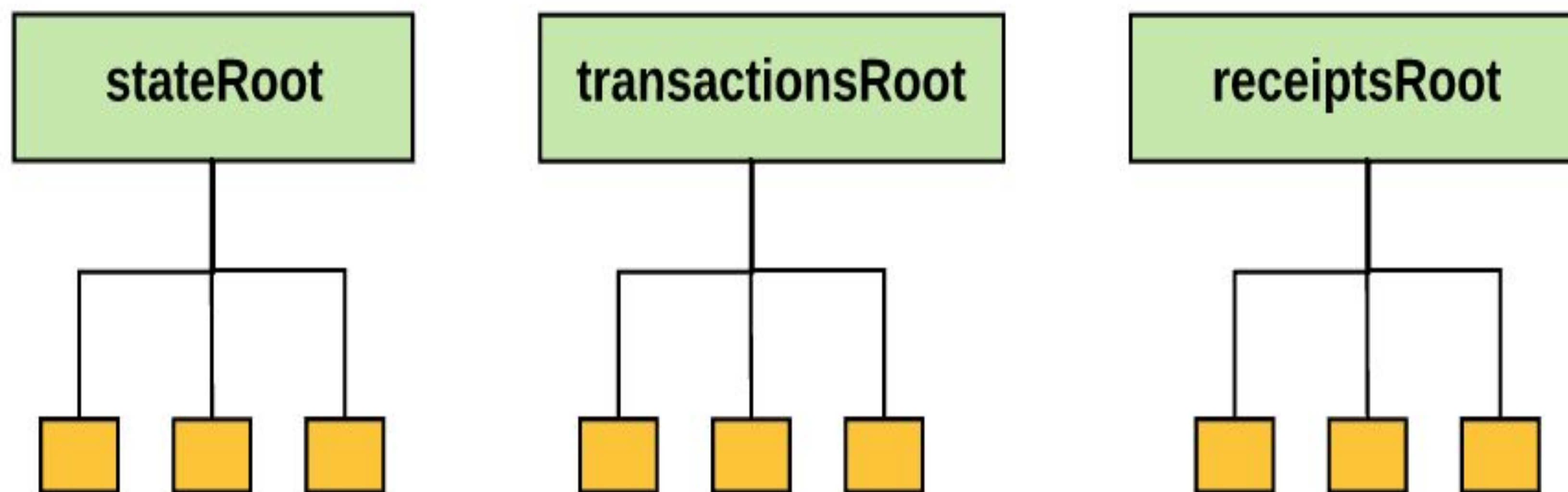
Why not hash all the transactions?

$$S(A, B, C, D, E, F, G, H)$$



# MERKLE PATRICIA TRIES

OH BOY...



## 5

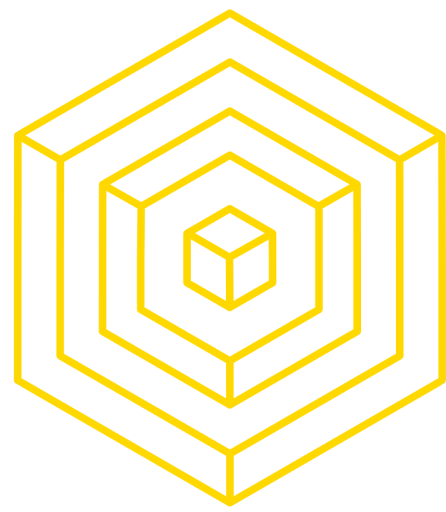
# MINING REVISITED



# BLOCK DIFFICULTY

THE MORE YOU TRY, THE HARD IT GETS

- The difficulty of a block is used to enforce consistency in the time it takes to validate blocks
- Genesis block starts with difficulty of 131,072 and a special formula is used to calculate the block difficulty every block thereafter.
- If a certain block is validated more quickly than the previous block, the Ethereum protocol increases that block's difficulty
- The difficulty of a block affects the nonce, where a **nonce**: value in a block that be adjusted to try and satisfy the proof of work condition



# BLOCK DIFFICULTY

THE MORE YOU TRY, THE HARD IT GETS

- The relationship between the **nonce** and the **block difficulty** is formalized as such:

$$n \leq \frac{2^{256}}{H_d}$$

$H_d$  := difficulty of the block

- So, by adjusting the difficulty of a block, the protocol can adjust how long it takes to validate a block





# BLOCK FINALITY

THE MORE YOU TRY, THE HARD IT GETS

## Validate (or, if mining, determine) ommers

- Each ommer block within the block header must be a valid header and be within the sixth generation of the present block.

## Validate (or, if mining, determine) transactions

- The **gasUsed** number on the block must be equal to the cumulative gas used by the transactions listed in the block. (Recall that when executing a transaction, we keep track of the block gas counter, which keeps track of the total gas used by all transactions in the block).



# BLOCK FINALITY

THE MORE YOU TRY, THE HARD IT GETS

## Apply rewards (only if mining)

- The beneficiary address is awarded 5 Ether for mining the block. (Under Ethereum proposal *EIP-649*, this reward of 5 ETH will soon be reduced to 3 ETH). Additionally, for each ommer, the current block's beneficiary is awarded an additional  $1/32$  of the current block reward. Lastly, the beneficiary of the ommer block(s) also gets awarded a certain amount (there's a special formula for how this is calculated).

## Verify (or, if mining, compute a valid) state and nonce

- Ensure that all transactions and resultant state changes are applied, and then define the new block as the state after the block reward has been applied to the final transaction's resultant state.
- ▲ ▼ Verification occurs by checking this final state against the state trie stored in the header.



# BLOCK REWARDS

## FREE MONEY

82

- **Block reward** - each block mined creates 5 fresh ETH
- **Uncle reward** - if uncles are referenced by later blocks as uncles, they create 4.375 ETH for the miner of the uncle (7/8th of the full 5 ETH reward)
  - A miner who references an uncle also gets about 0.15 ETH (1/32 of the full block reward per uncle (max of 2 uncles can be referenced))
  - Remember the tradeoffs of including more transactions!!!

# 5.1

## ETHASH



# ALGORITHM

## PROOF OF WORK MINING

- The algorithm is formally defined as:

$$(m, n) = PoW(H_{\tilde{n}}, H_n, d)$$

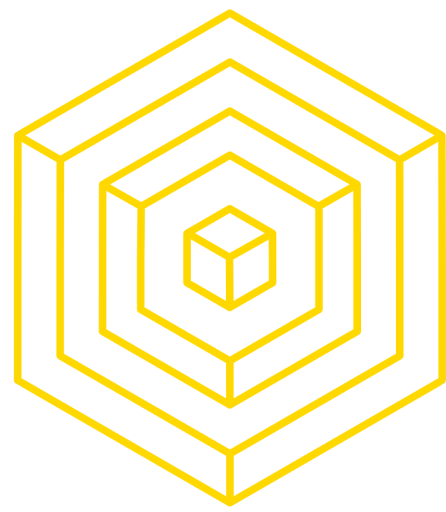
$m$  is the mixHash

$n$  is the nonce

$H_{\tilde{n}}$  is the new block's header, without the nonce and mixHash components

$H_n$  is the nonce of the block header

$d$  is the DAG, which is a large dataset



# ALGORITHM

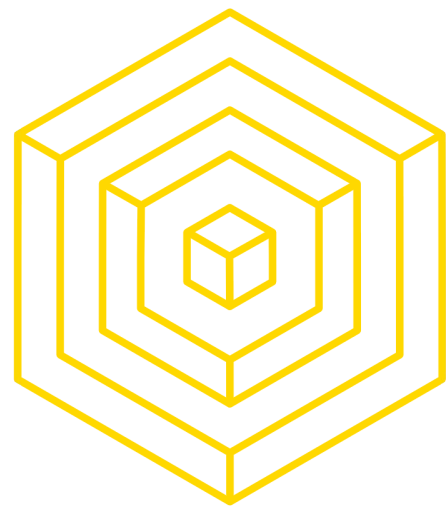
## PROOF OF WORK MINING

85

- **Recall that:**

- **mixHash** is a hash that, when combined with the nonce, proves that this block has carried out enough computation
- **nonce** is a hash that, when combined with the mixHash, proves that this block has carried out enough computation

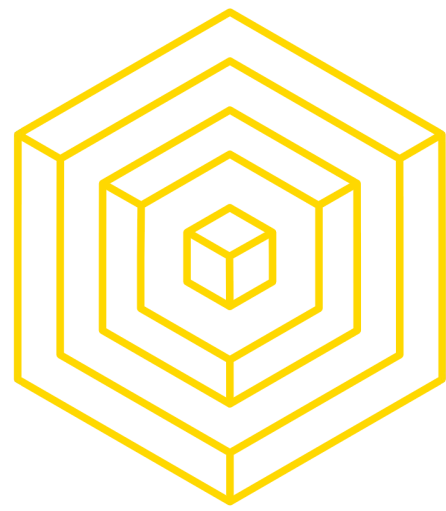




# STEP 1: SEED CALCULATION

## PROOF OF WORK MINING

- A **seed** is calculated for each block by scanning through block headers
  - This seed is different for every “epoch,” where each epoch is 30,000 blocks long
  - For the first epoch, the seed is the hash of a series of 32 bytes of zeros
  - For every subsequent epoch, it is the hash of the previous seed hash
- Using this seed, a node can calculate a pseudo-random “cache.”
  - This cache is incredibly useful because it enables light nodes
  - A light node can verify the validity of a transaction based solely on this cache, because the cache can regenerate the specific block it needs to verify



# STEP 2: DAG GENERATION

## PROOF OF WORK MINING

- Using the cache, a node can generate the **DAG** “dataset,” where each item in the **dataset** depends on a small number of pseudo-randomly-selected items from the cache
- In order to be a miner, you must generate this full dataset; all full clients and miners store this dataset, and the dataset grows linearly with time



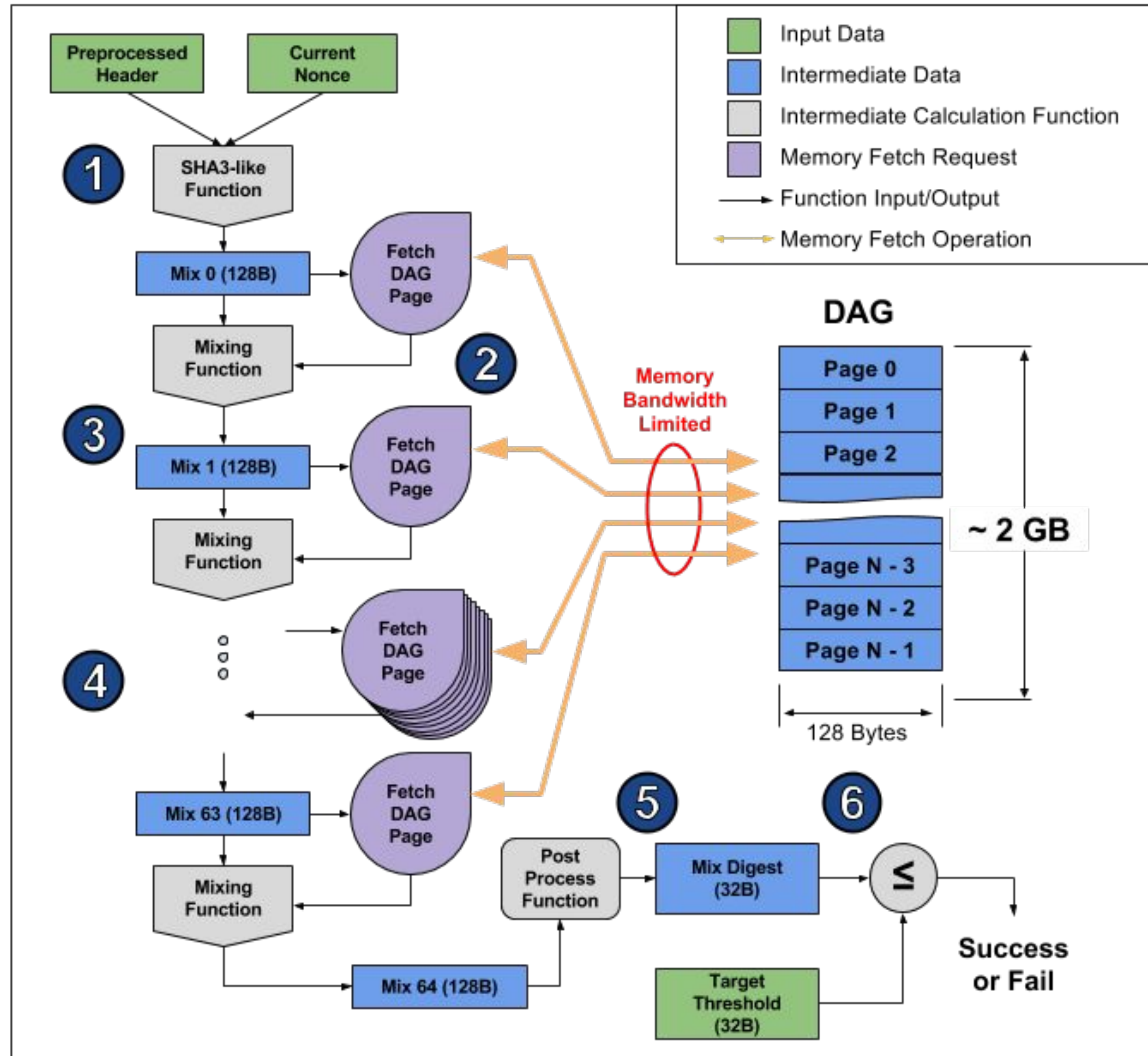
# STEP 3: PARTIAL COLLISION PUZZLE

## PROOF OF WORK MINING

- Miners can then take random slices of the dataset and put them through a mathematical function to hash them together into a **mixHash**  $m$
- A miner will repeatedly generate a **mixHash** until the output is below the desired target nonce
- When the output meets this requirement, this nonce is considered valid and the block can be added to the chain

$$(m, n) = PoW(H_{\tilde{n}}, H_n, d)$$

## Ethash Hashing Algorithm



Find out more about  
[memory hardness](#)





# RATIONALE

OLD NAME: DAGGER HASHIMOTO

- IO saturation
- GPU friendliness
- Light client verifiability
- Light client slowdown
- Light client fast startup

Read more [here](#)!



# RATIONALE

OLD NAME: DAGGER HASHIMOTO

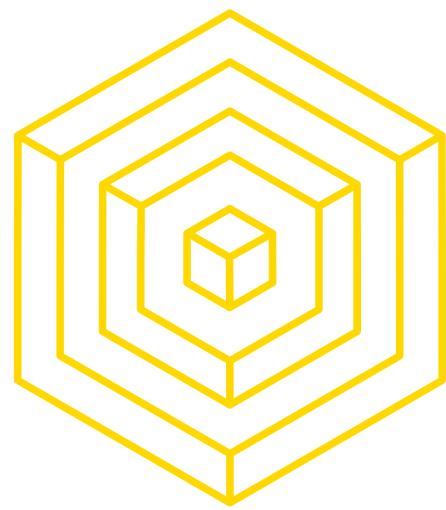
## IO saturation

- The algorithm should consume nearly the entire available memory access bandwidth (this is a strategy toward achieving ASIC resistance, the argument being that commodity RAM, especially in GPUs, is much closer to the theoretical optimum than commodity computing capacity)

## GPU friendliness

- We try to make it as easy as possible to mine with GPUs. Targeting CPUs is almost certainly impossible, as potential specialization gains are too great, and there do exist criticisms of CPU-friendly algorithms that they are vulnerable to botnets, so we target GPUs as a compromise.





# RATIONALE

OLD NAME: DAGGER HASHIMOTO

92

## Light client verifiability

- A light client should be able to verify a round of mining in under 0.01 seconds on a desktop in C, and under 0.1 seconds in Python or Javascript, with at most 1 MB of memory (but exponentially increasing)

## Light client slowdown

- The process of running the algorithm with a light client should be much slower than the process with a full client, to the point that the light client algorithm is not an economically viable route toward making a mining implementation, including via specialized hardware



# RATIONALE

OLD NAME: DAGGER HASHIMOTO

93

## Light client fast startup

- A light client should be able to become fully operational and able to verify blocks within 40 seconds in Javascript

# SEE YOU NEXT TIME

Ethereum Virtual Machine

Byte Code

Opcode

Gas Revisit

Stack-Based VM

Program Execution