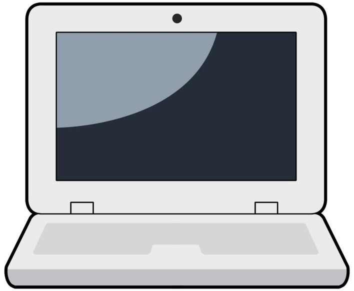
Blockchains & Distributed Ledgers

Lecture 02

Dimitris Karakostas

Slide credits: DK, Aggelos Kiayias, Dionysis Zindros, Christos Nasikas

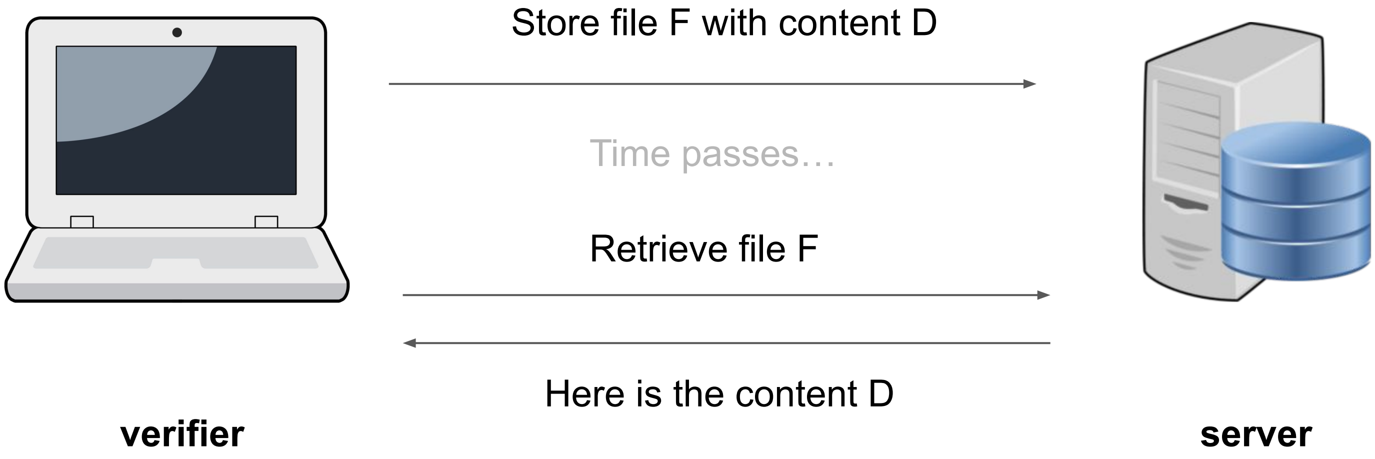
The authenticated file storage problem



Store file F with content D

verifier **server**

The authenticated file storage problem



The authenticated file storage problem

The problem

- Client wants to store a file, with identifier F and content D, on a server
- Clients wants to retrieve D later in time

Usecases

- Save storage space (e.g., cloud)
- Redundancy (e.g., backup)

File storage: Basic protocol

- Client sends file F with content D to server
- Server stores (F, D)
- Client deletes D
- Client requests F from server
- Server returns D
- Client has recovered D

File storage: Basic protocol

- Client sends file F with content D to server
- Server stores (F, D)
- Client deletes D
- Client requests F from server
- Server returns D
- Client has recovered D

What if ***server is corrupted*** *and returns D' != D?*

File storage: Protocol against adversaries

Trivial solution:

- Client does not delete D
- When server returns D’, client compares D and D’

What if client can’t store D for a long time?

Authenticated Data Structures

- Like regular data structures, but cryptographically authenticated
- A **verifier** can store/retrieve/operate on data held by an **<u>untrusted</u> prover**
  - Client wants to store a file, with identifier F and content D, on a server
  - Client wants to delete D
  - Clients wants to retrieve D later in time
  - Prover is *not trusted* - it has to *prove* that the returned data is the correct/original D
- How can this problem be solved using:
  a. A hash function H
  b. A signature scheme Σ = <KeyGen, Sign, Verify>

File storage: Authenticated protocols

Hash-based

- Client sends file F with data D to server
- Server stores (F, D)
- Client computes and stores $H(D)$, deletes D

Time passes…

- Client requests F from server
- Server returns D'
- Client compares $H(D') = H(D)$

File storage: Authenticated protocols

Digital signature-based

- Client creates and stores key pair *(sk, vk)*
- Client computes $\sigma = Sign(sk, \langle F, D \rangle)$
- Client sends (F, D, σ) to server, deletes D, σ
- Server stores (F, D, σ)

Time passes…

- Client requests F from server
- Server returns (D', σ')
- Client checks if $Verify(vk, \langle F, D' \rangle, \sigma') = True$

File storage: Authenticated protocols

Hash-based

- Client sends file F with data D to server
- Server stores (F, D)
- Client computes and stores $H(D)$, deletes D

Time passes…

- Client requests F from server
- Server returns D'
- Client compares $H(D') = H(D)$

Digital signature-based

- Client creates and stores key pair (sk, vk)
- Client computes $\sigma = \mathit{Sign}(sk, <F, D>)$
- Client sends (F, D, σ) to server, deletes D, σ
- Server stores (F, D, σ)

Time passes…

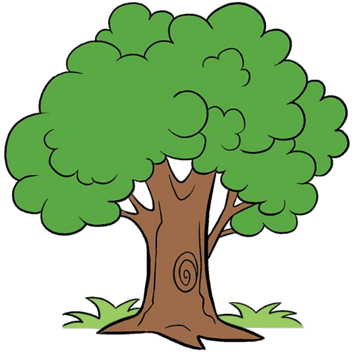
- Client requests F from server
- Server returns (D', σ')
- Client checks if $\mathit{Verify}(vk, <F, D'>, \sigma') = \mathit{True}$

What if client needs only one byte of the file?

Merkle Trees

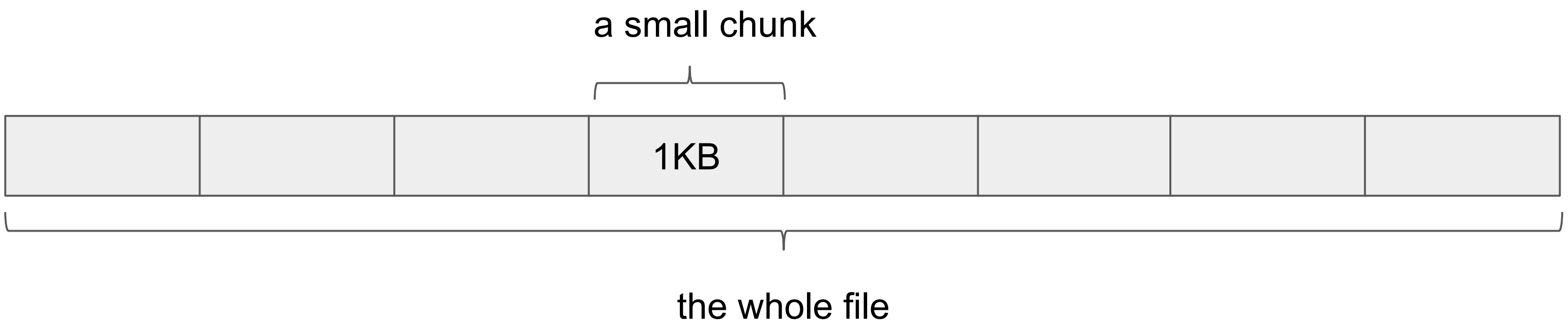
Trees definitions

- **Binary**: every node has at most 2 children
- Binary **full**: every node has either 0 or 2 children
- Binary **complete**: every node in every level, except possibly the second-to-last, has exactly 2 children, and all nodes in the last level are as far left as possible
- **Merkle tree**: an *authenticated* binary tree



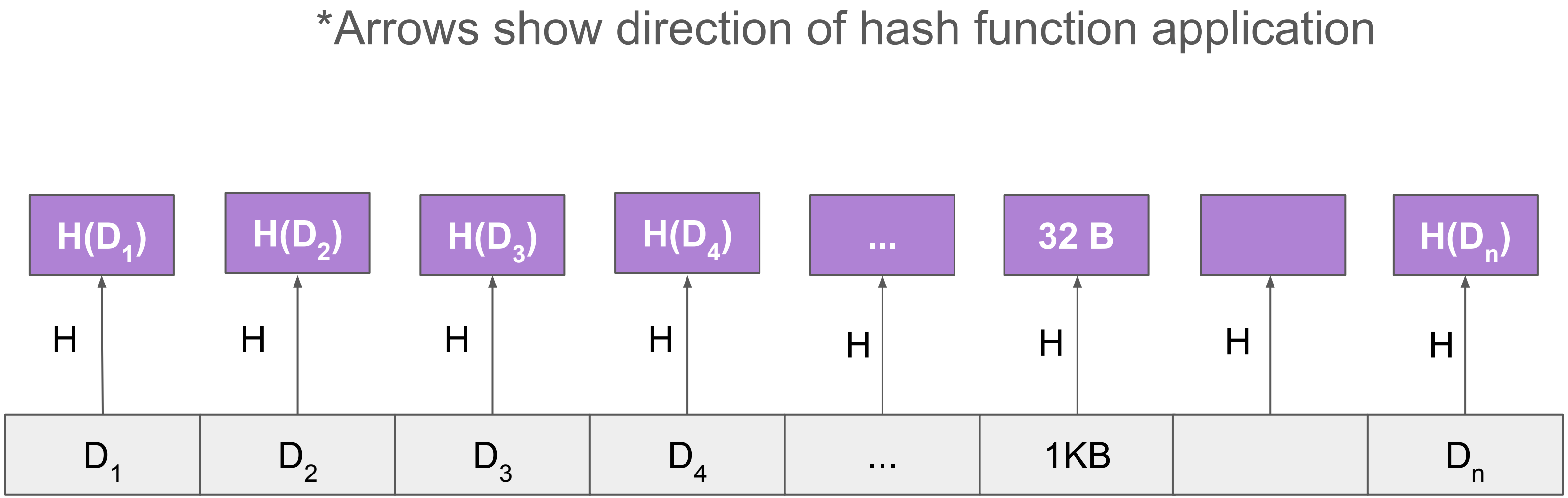
Merkle Tree

- Split file into *small* **chunks** (e.g., 1KB)



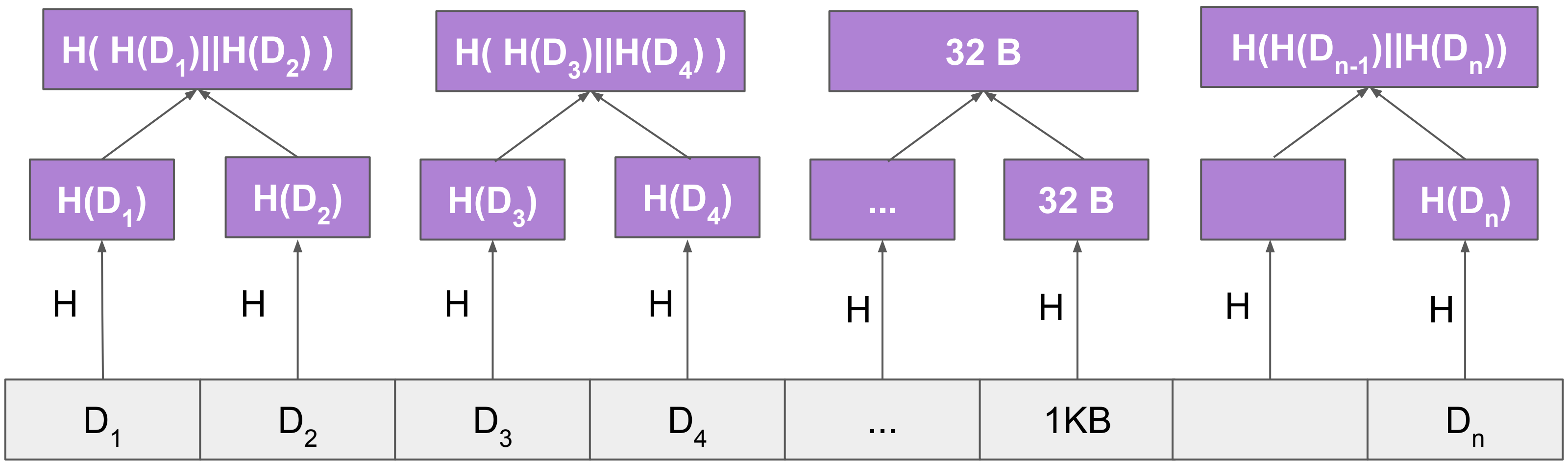
Merkle Tree

- **Hash** each chunk using a cryptographic hash function (e.g., SHA256)

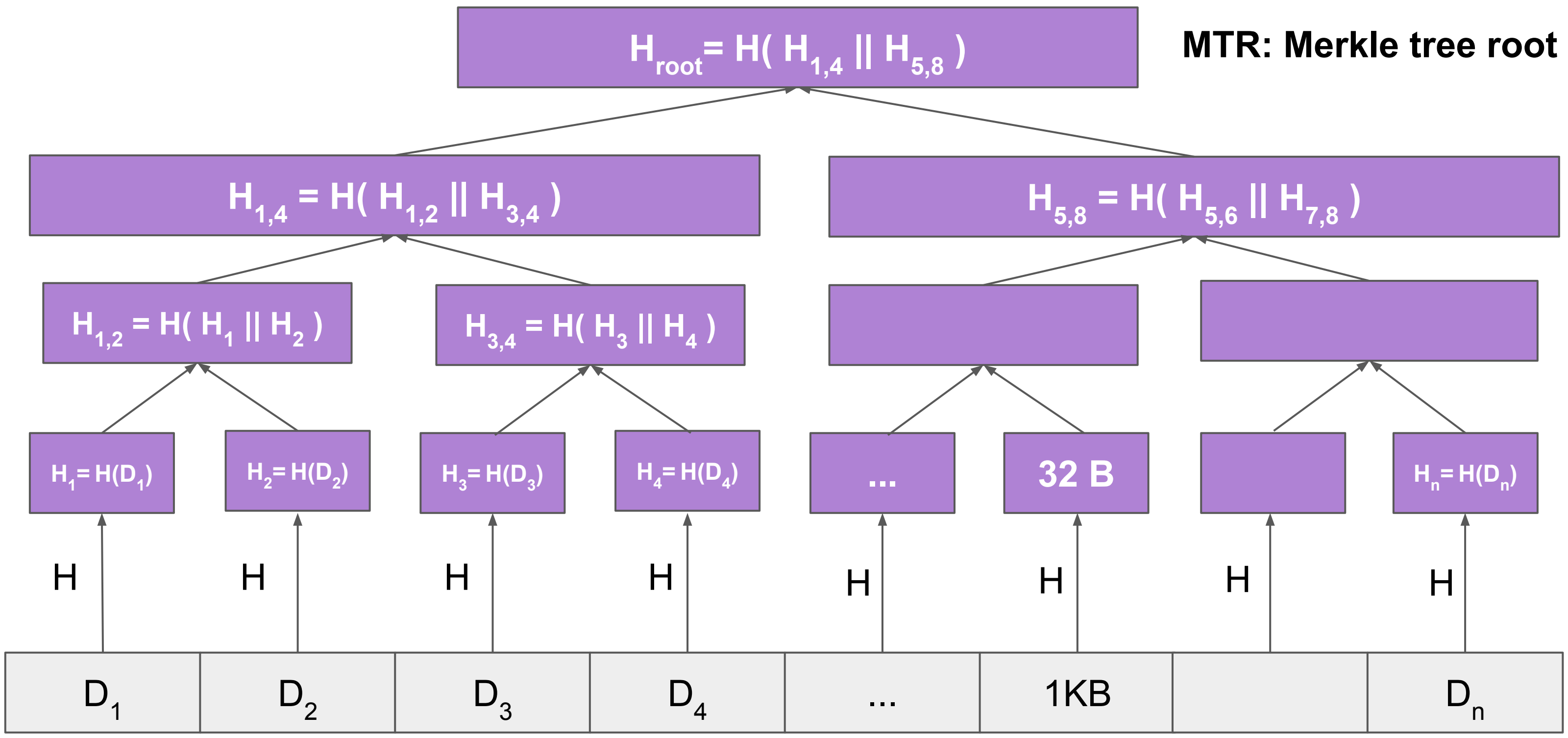


Merkle Tree

- **Combine** them by two to create a binary tree
- Each node stores the **hash** of the **concat** of its children



Merkle Tree



$H_{root} = H(H_{1,4} || H_{5,8})$
MTR: Merkle tree root
$H_{1,4} = H(H_{1,2} || H_{3,4})$
$H_{5,8} = H(H_{5,6} || H_{7,8})$
$H_{1,2} = H(H_1 || H_2)$
$H_{3,4} = H(H_3 || H_4)$
$H_1 = H(D_1)$
$H_2 = H(D_2)$
$H_3 = H(D_3)$
$H_4 = H(D_4)$
...
32 B
$H_n = H(D_n)$
H
H
H
H
H
H
H
H
D_1
D_2
D_3
D_4
...
1KB
D_n

File storage: Merkle tree-based protocol

- Client sends file data D to server
- Client creates Merkle Tree root **MTR** from initial file data D
- Client deletes data D, but stores MTR (32 bytes)

File storage: Merkle tree-based protocol

- Client sends file data D to server
- Client creates Merkle Tree root **MTR** from initial file data D
- Client deletes data D, but stores MTR (32 bytes)

Time passes…

- Client requests chunk x from server
- Server returns chunk x and *short* proof-of-inclusion π
- Client checks whether proof π of chunk x is correct w.r.t. stored MTR

Merkle tree: proof of inclusion

Verifier: $MTR_{abcdefgh}$

Prover: a, b, c, d, e, f, g, h

Merkle tree: proof of inclusion

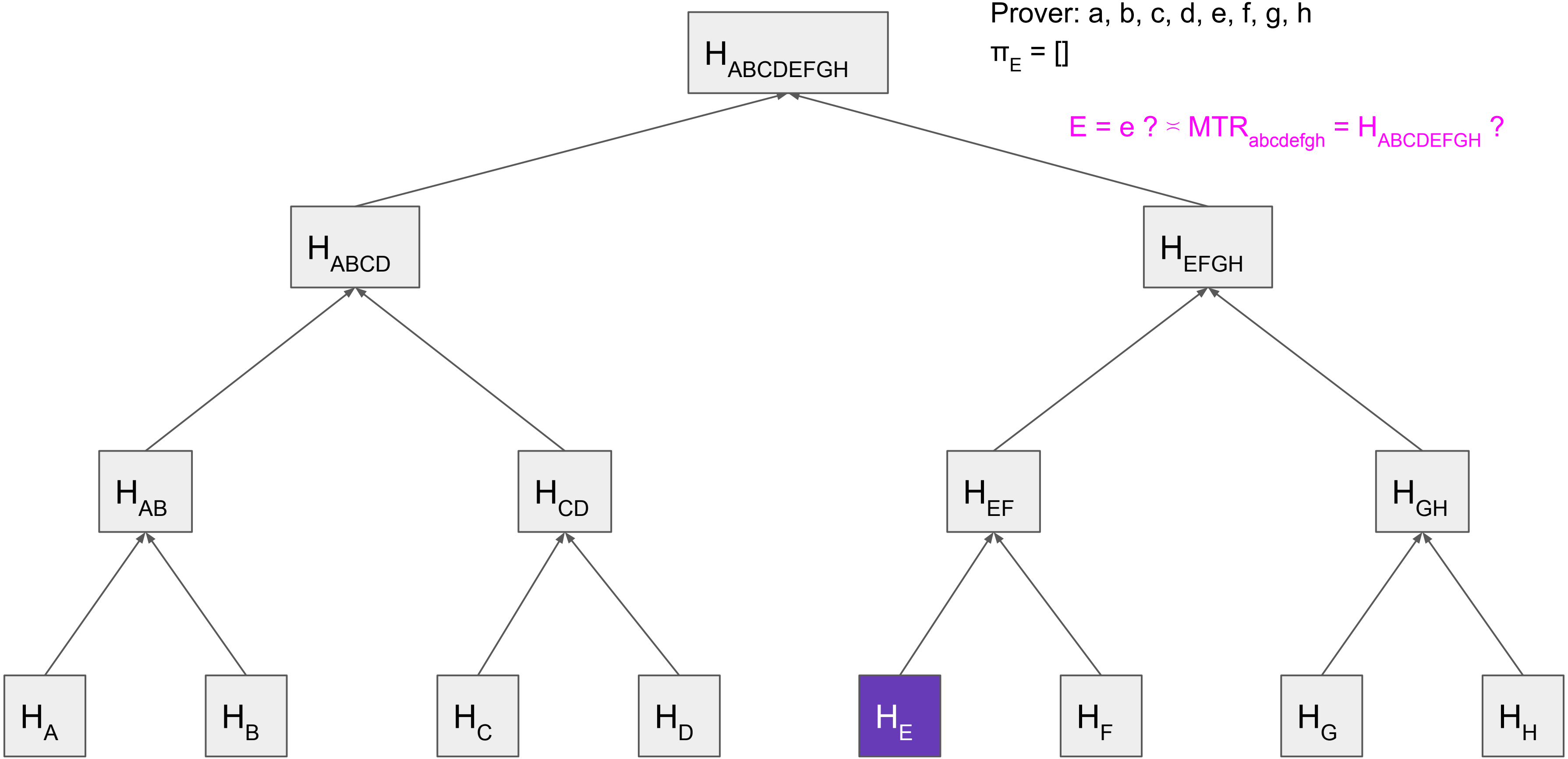
Verifier: $MTR_{abcdefgh}$, E, π_E
Prover: a, b, c, d, e, f, g, h

E = e ?

Merkle tree: proof of inclusion

Verifier: $MTR_{abcdefgh}$, E, π_E
Prover: a, b, c, d, e, f, g, h
π_E = []

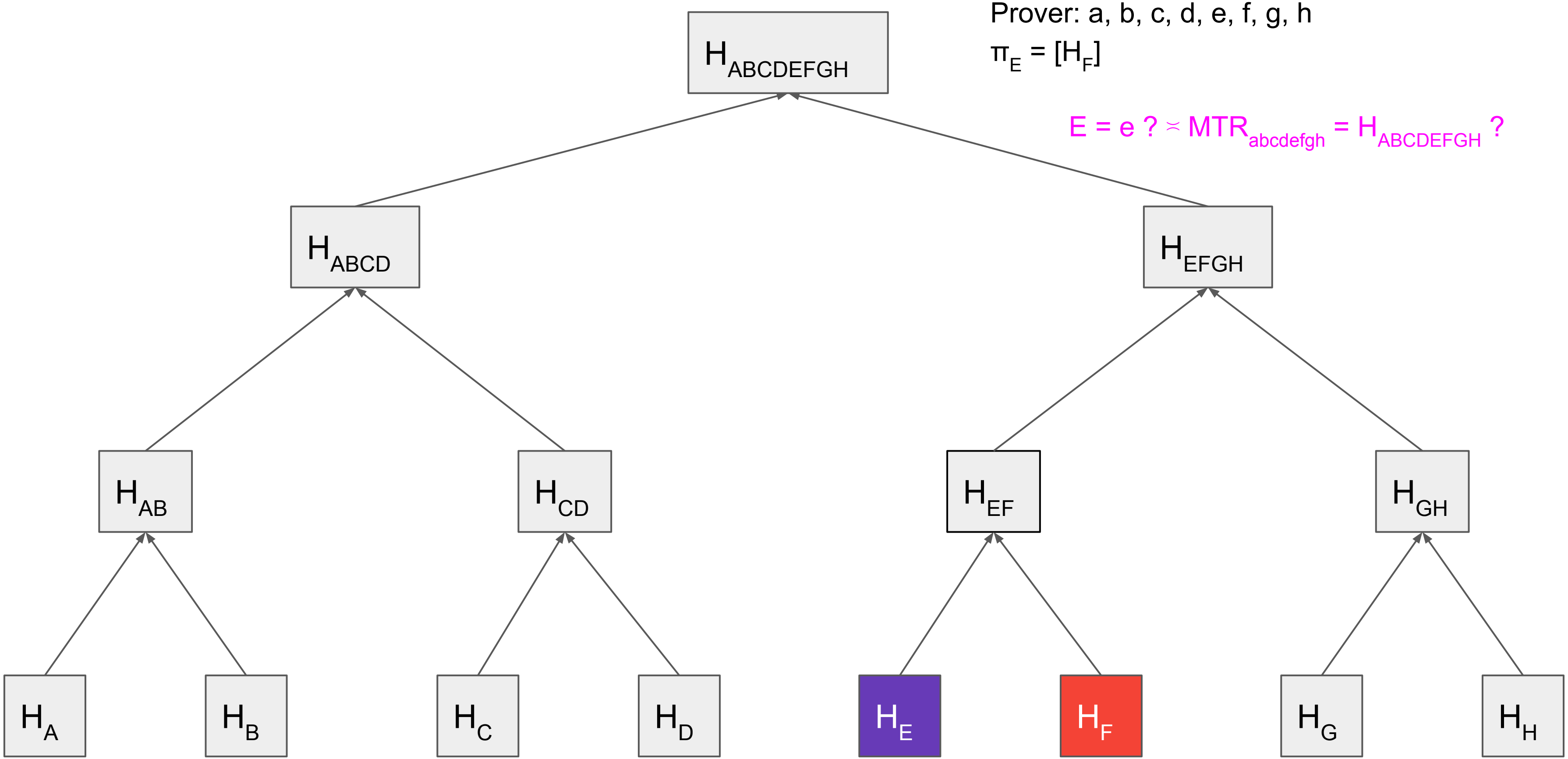
$E = e\ ?\ \asymp\ MTR_{abcdefgh} = H_{ABCDEFGH}\ ?$



Merkle tree: proof of inclusion

Verifier: $MTR_{abcdefgh}$, E, π_E
Prover: a, b, c, d, e, f, g, h
$\pi_E = [H_F]$

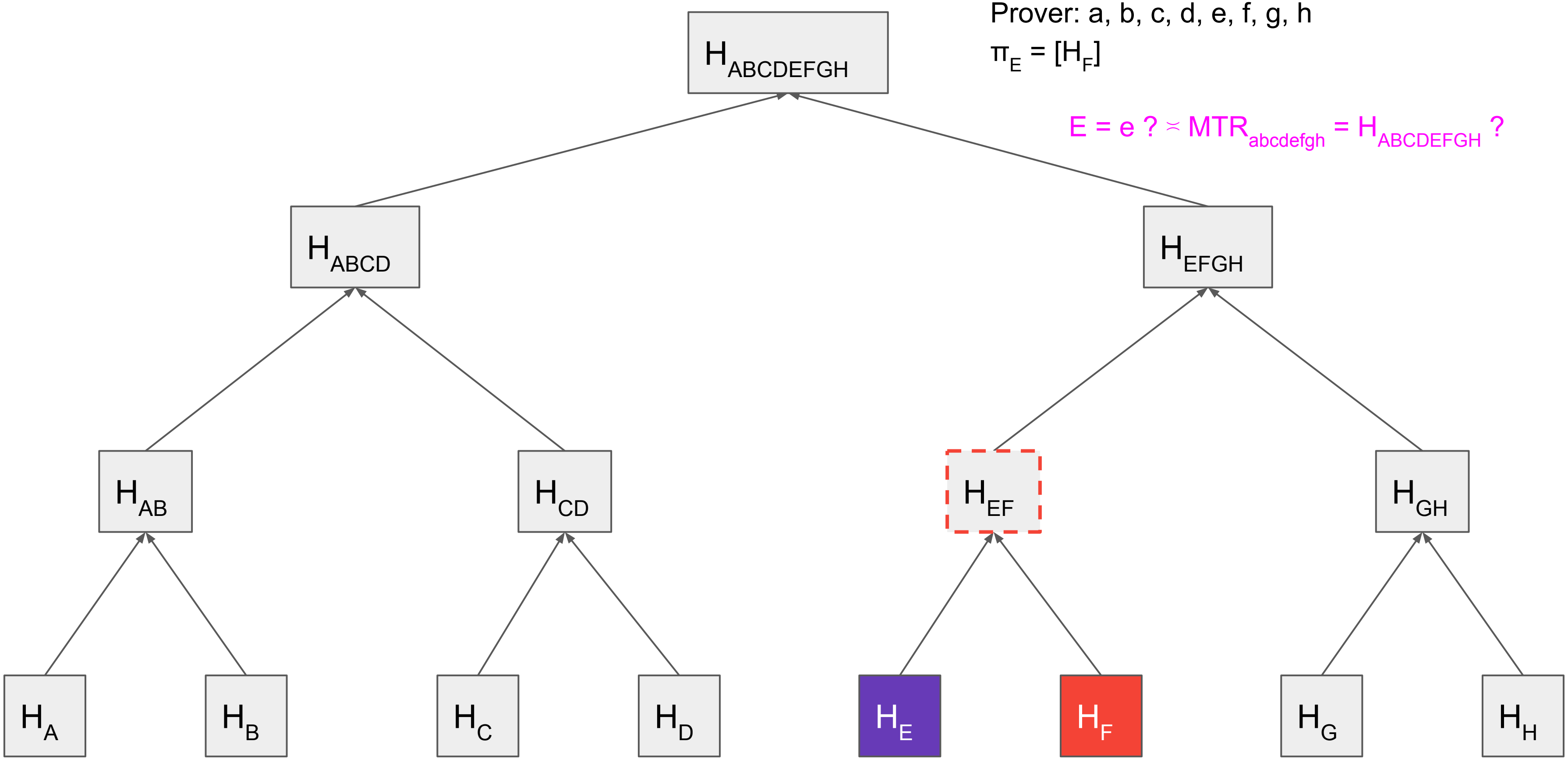
E = e ? ≍ $MTR_{abcdefgh} = H_{ABCDEFGH}$?



Merkle tree: proof of inclusion

Verifier: $MTR_{abcdefgh}$, E, π_E
Prover: a, b, c, d, e, f, g, h
$\pi_E = [H_F]$

$E = e \ ? \asymp MTR_{abcdefgh} = H_{ABCDEFGH} \ ?$



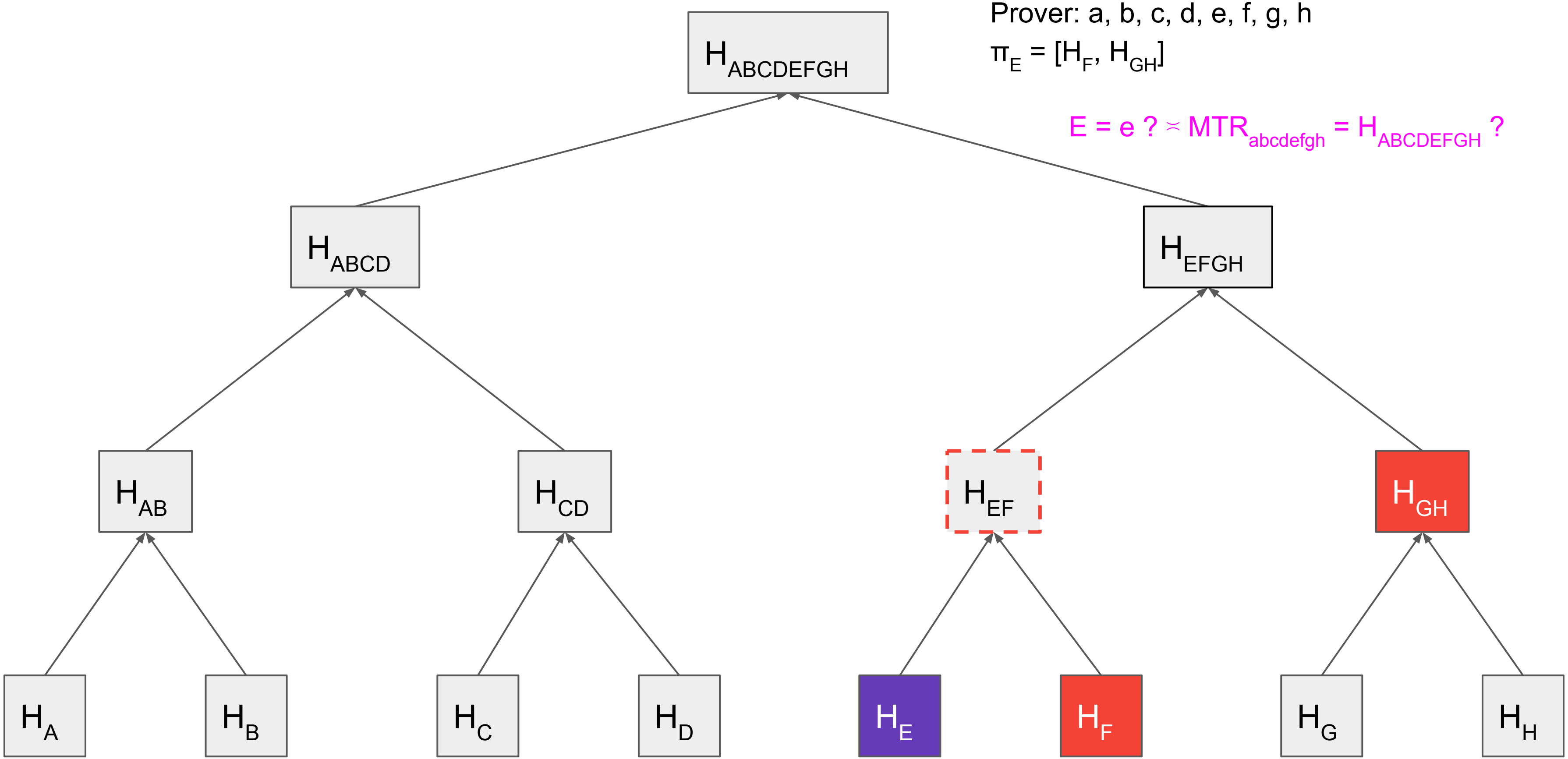
Merkle tree: proof of inclusion

Verifier: $MTR_{abcdefgh}$, E, π_E

Prover: a, b, c, d, e, f, g, h

$\pi_E = [H_F, H_{GH}]$

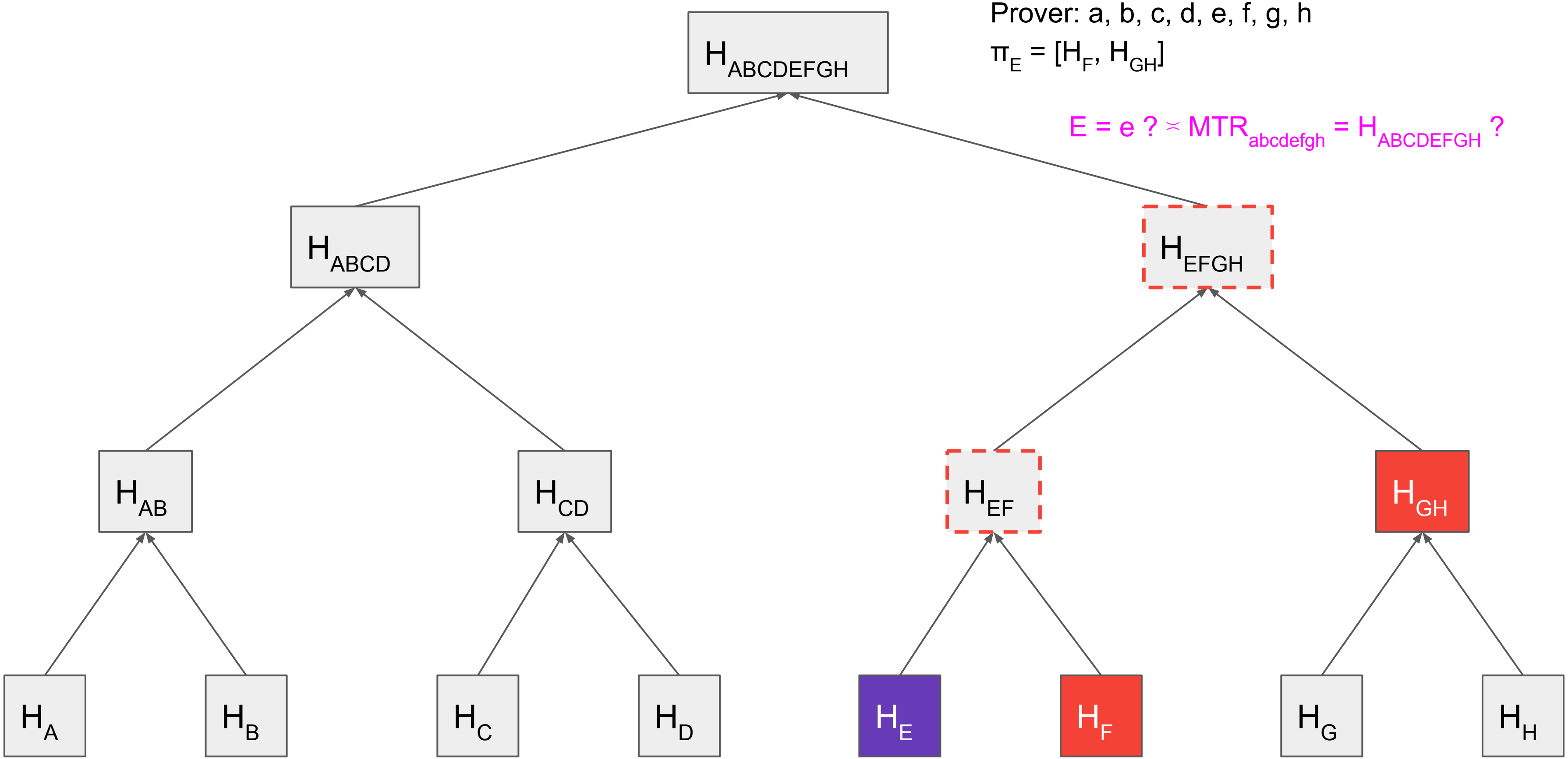
$E = e\ ? \asymp MTR_{abcdefgh} = H_{ABCDEFGH}\ ?$



Merkle tree: proof of inclusion

Verifier: $MTR_{abcdefgh}$, E, π_E
Prover: a, b, c, d, e, f, g, h
$\pi_E = [H_F, H_{GH}]$

$E = e\ ?\ \asymp MTR_{abcdefgh} = H_{ABCDEFGH}\ ?$



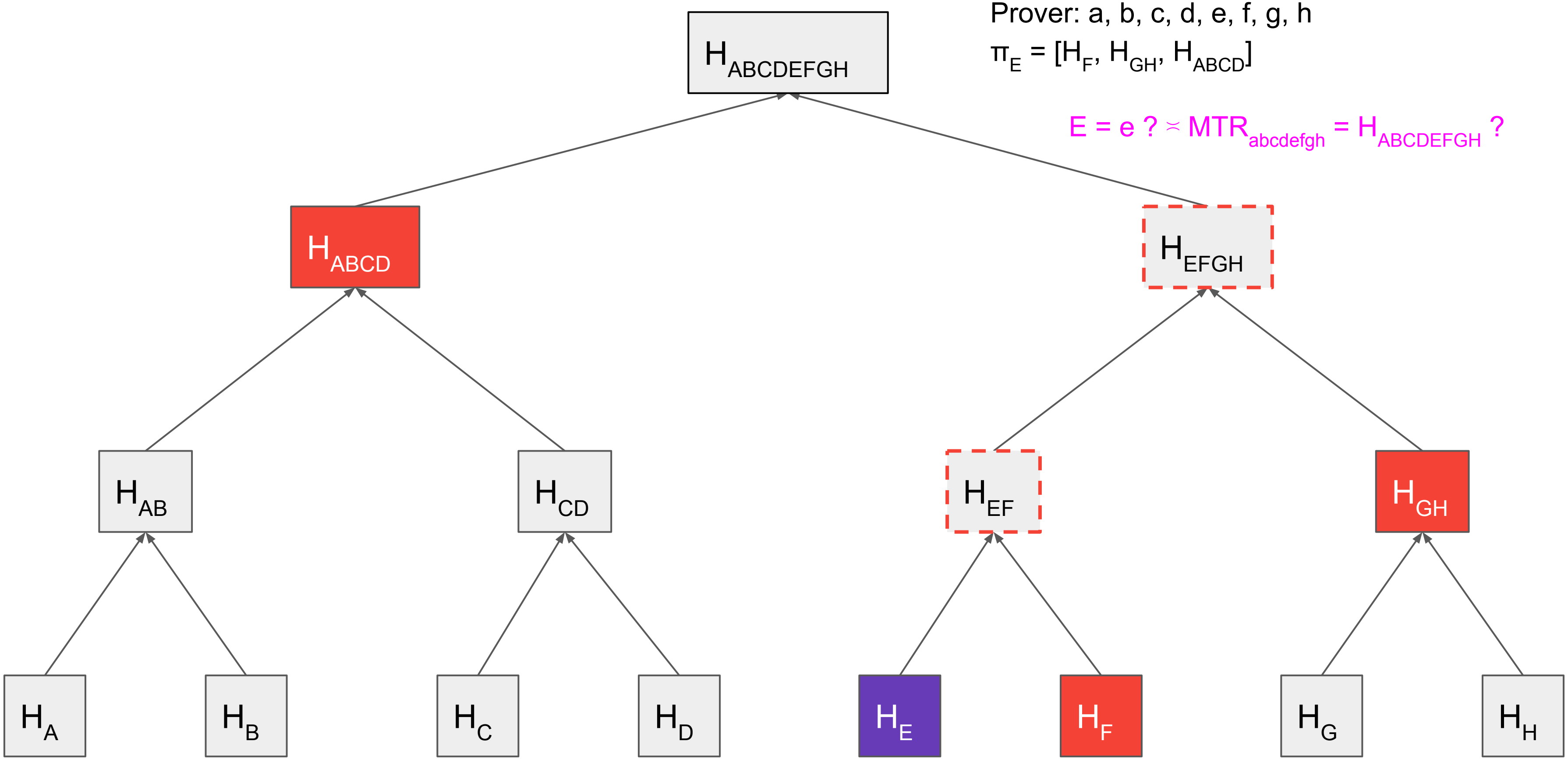
Merkle tree: proof of inclusion

Verifier: $MTR_{abcdefgh}$, E, π_E

Prover: a, b, c, d, e, f, g, h

$\pi_E = [H_F, H_{GH}, H_{ABCD}]$

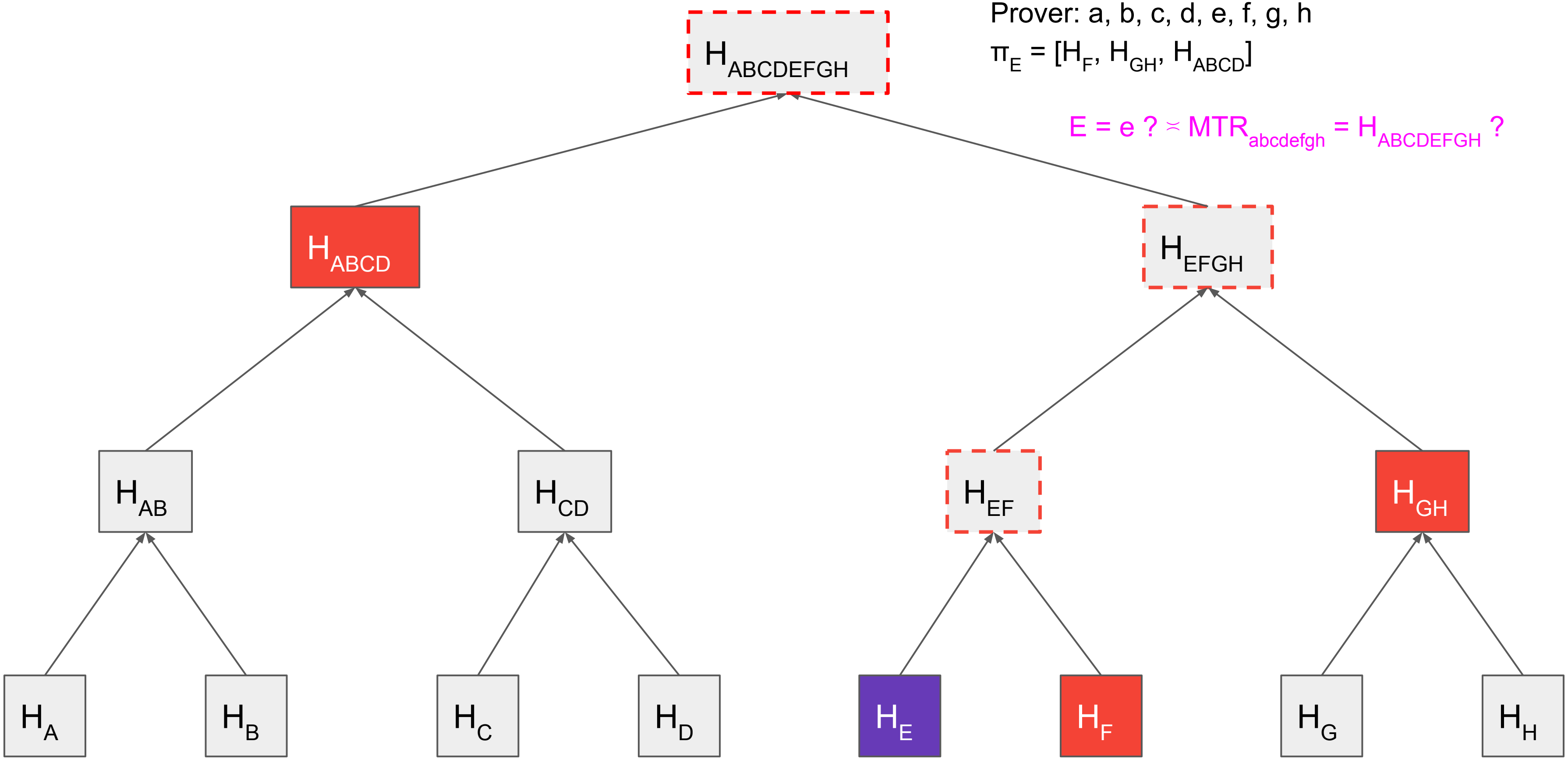
$E = e \ ? \asymp MTR_{abcdefgh} = H_{ABCDEFGH} \ ?$



Merkle tree: proof of inclusion

Verifier: $MTR_{abcdefgh}$, E, π_E
Prover: a, b, c, d, e, f, g, h
$\pi_E = [H_F, H_{GH}, H_{ABCD}]$

$E = e \ ? \asymp MTR_{abcdefgh} = H_{ABCDEFGH} \ ?$



Merkle Tree proof-of-inclusion

- Prover sends chunk
- Prover sends **siblings** along path connecting leaf to MTR
- Verifier computes hashes along the path connecting leaf to MTR
- Verifier checks that computed root is equal to MTR
- How big is proof-of-inclusion?

Merkle Tree proof-of-inclusion

- Prover sends chunk
- Prover sends **siblings** along path connecting leaf to MTR
- Verifier computes hashes along the path connecting leaf to MTR
- Verifier checks that computed root is equal to MTR
- How big is proof-of-inclusion?

$$|\pi| \in \Theta(\log_2|D|)$$

Merkle tree applications

- BitTorrent uses Merkle trees to verify exchanged files
- Bitcoin uses Merkle trees to store transactions
- Ethereum uses Merkle-Patricia tries for storage and transactions

Storing *sets* instead of lists

- Merkle trees can be used to store *sets* of keys instead of lists
- Verifier asks prover to store a set of keys
- Verifier deletes set
- Verifier later asks prover if key belongs to set
- Prover provides proof-of-inclusion or proof-of-non-inclusion
- Prover can be adversarial

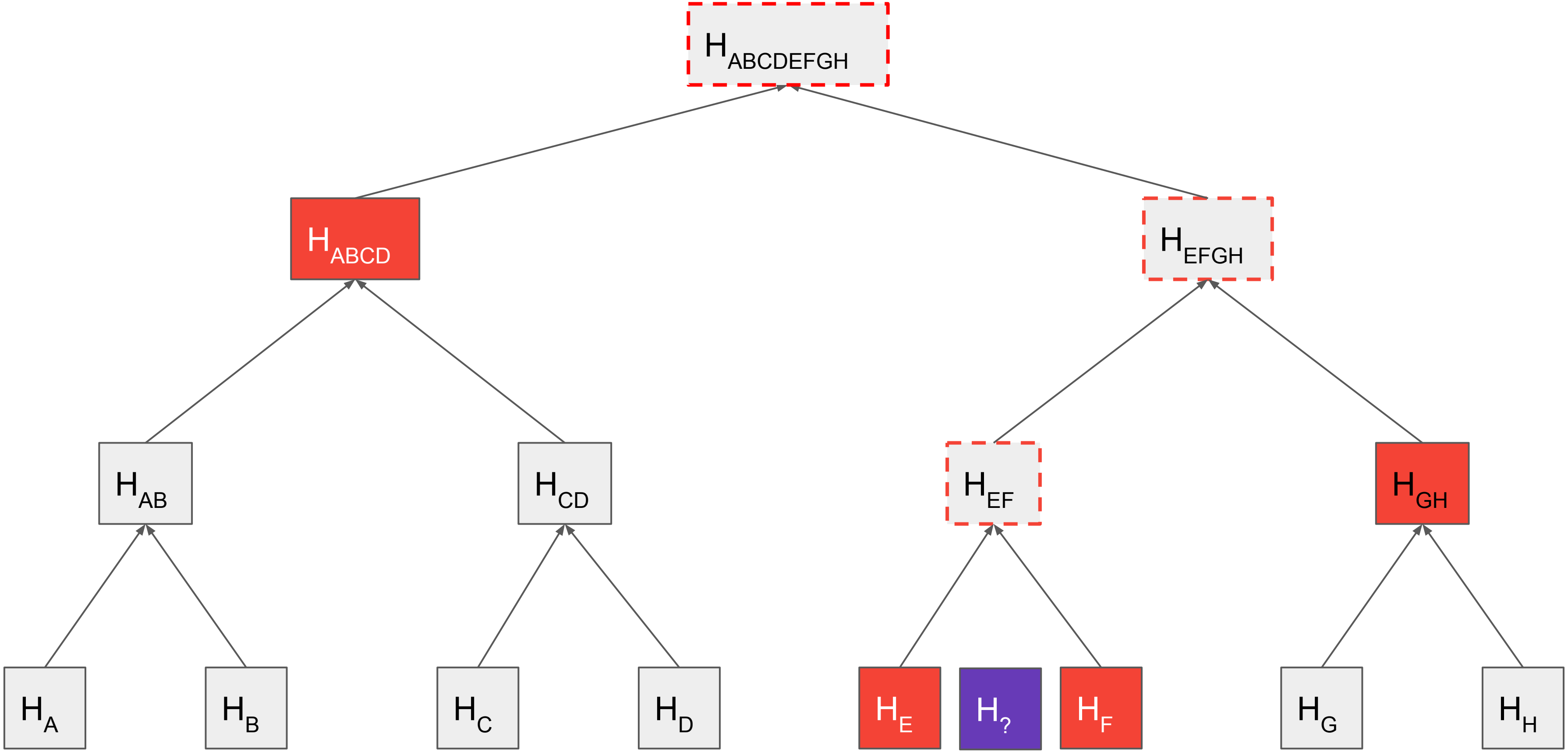
Merkle trees for set storage

- Verifier sorts set elements
- Creates MTR on sorted set
- Proof-of-inclusion as before

Merkle trees for set storage

- Verifier sorts set elements
- Creates MTR on sorted set
- Proof-of-inclusion as before
- Proof-of-non-inclusion for x
  - Show proof-of-inclusion for previous $H_<$ and next $H_>$ element in set
  - Verifier checks that $H_<$, $H_>$ proofs-of-inclusion are correct
  - Verifier checks that $H_<$, $H_>$ are adjacent in tree
  - Verifier checks that $H_< < x$ and $H_> > x$
  - Question: How to compress the two proofs-of-inclusion into one?

Merkle tree: proof of inclusion / non-inclusion



Tries

Tries

- Also called radix or prefix tree
- Search tree: ordered data structure
- Used to store a set or an associative array (key/value store)
- Keys are usually strings

Tries

- **Initialize**: Start with empty root
- Supports two operations: **add** and **query**
- **add** adds a string to the set
- **query** checks if a string is in the set (true/false)

Tries / Patricia tries as key/value store

- Marking can contain arbitrary value
- This allows to map keys to values
- **add(key, value)**
- **query(key) → value**

Tries: add(string)

- Start at root
- Split string into characters
- For every character, follow an edge labelled by that character
- If edge does not exist, create it
- Mark the node you arrive at

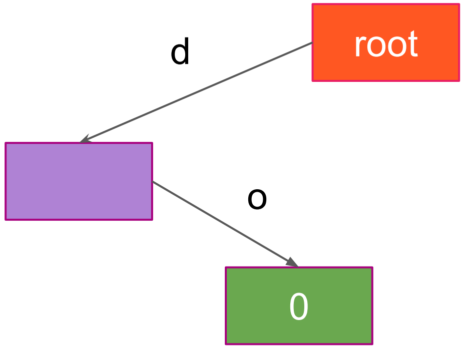
Tries: query(string)

- Start at root
- Split string into characters
- For every character, follow an edge labelled by that character
- If edge does not exist, return false
- When you arrive at a node and your string is consumed, check if node is marked
    - If it is marked, return **yes** (and marked value)
    - Otherwise, return **no**

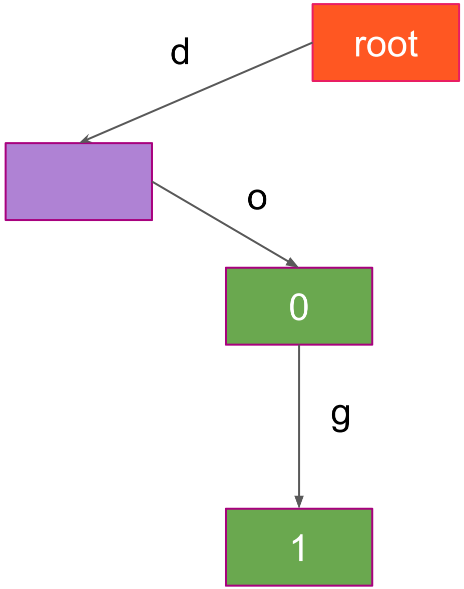
{ }

root

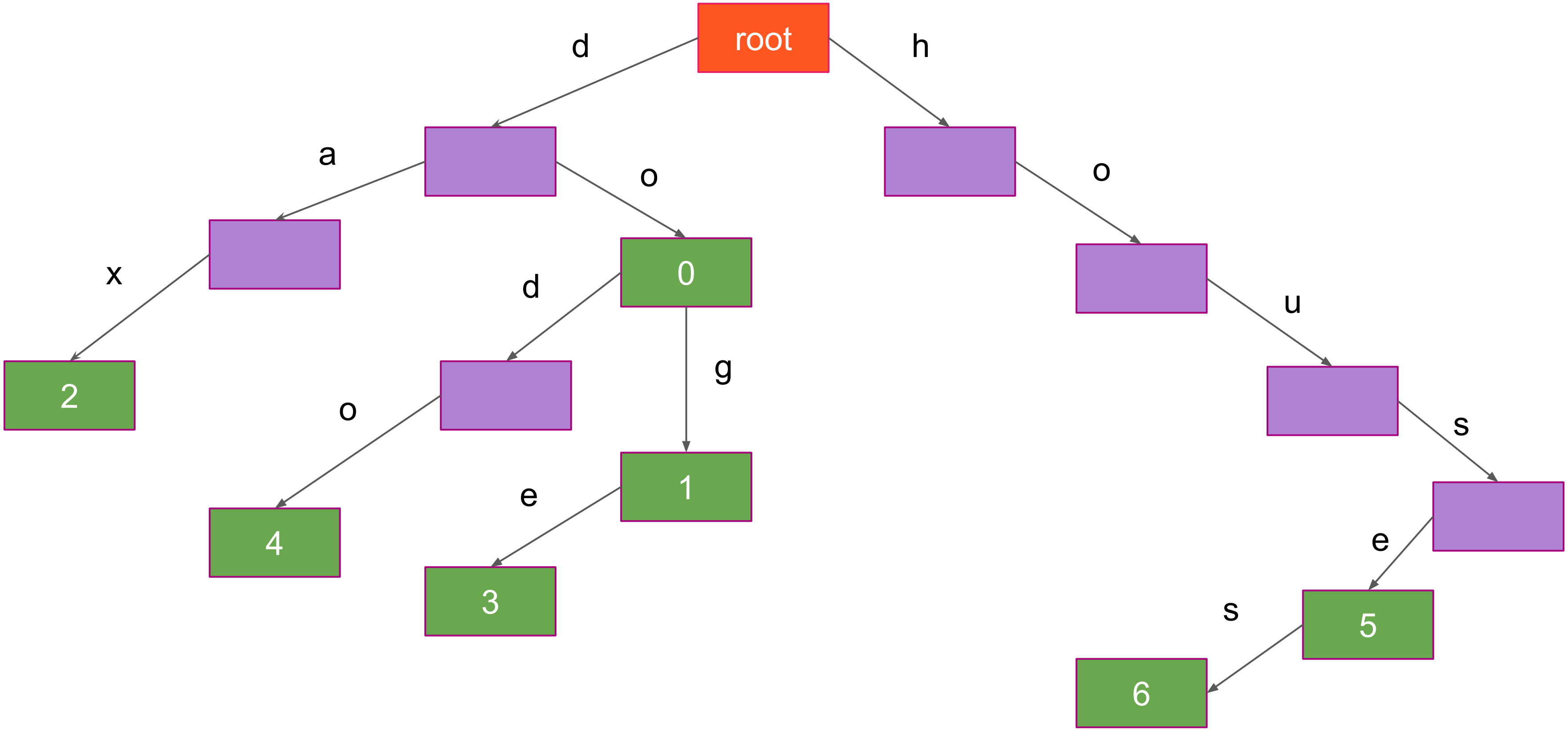
{ **do**: 0 }



{ **do**: 0, **dog**: 1 }



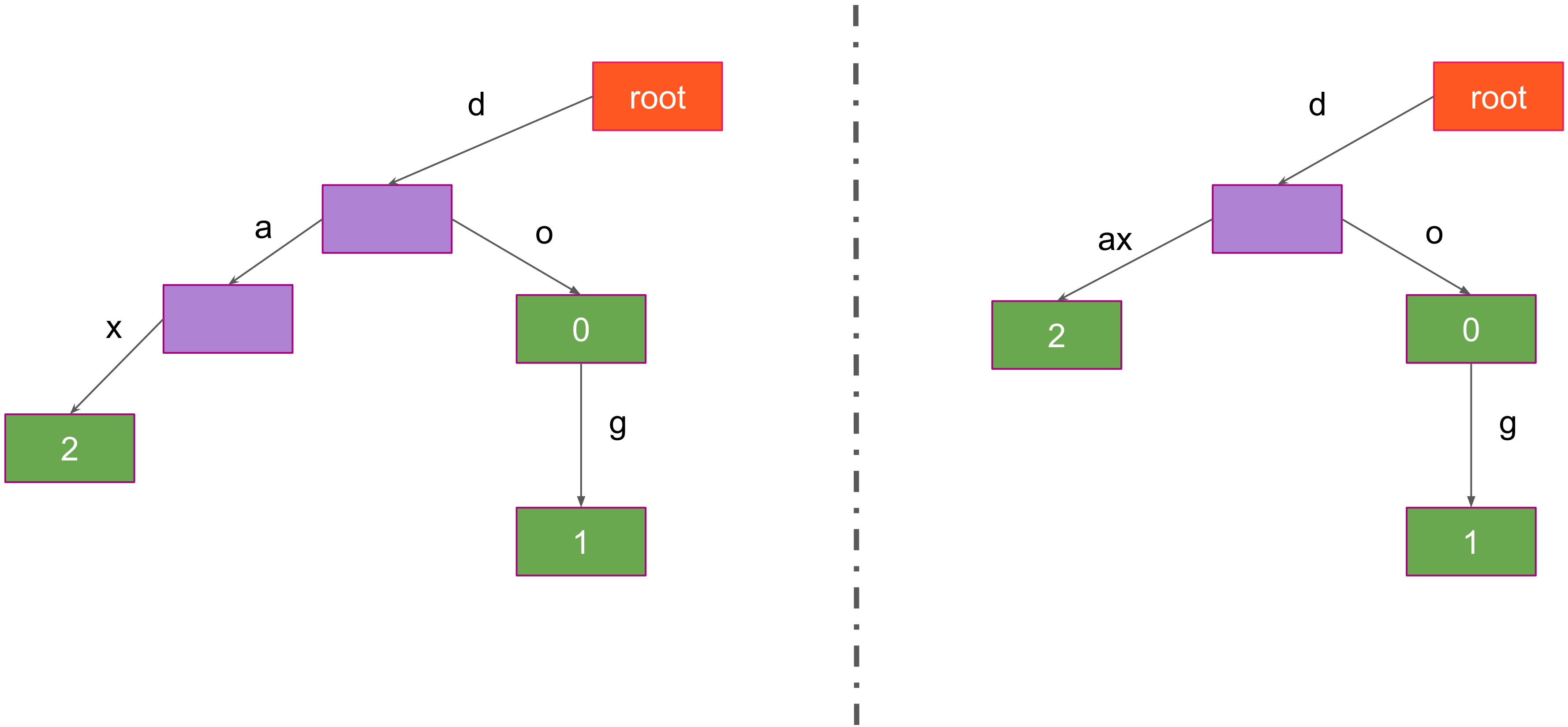
{ **do**: 0, **dog**: 1, **dax**: 2, **doge**: 3, **dodo**: 4, **house**: 5, **houses**: 6 }



Patricia (or radix) tree

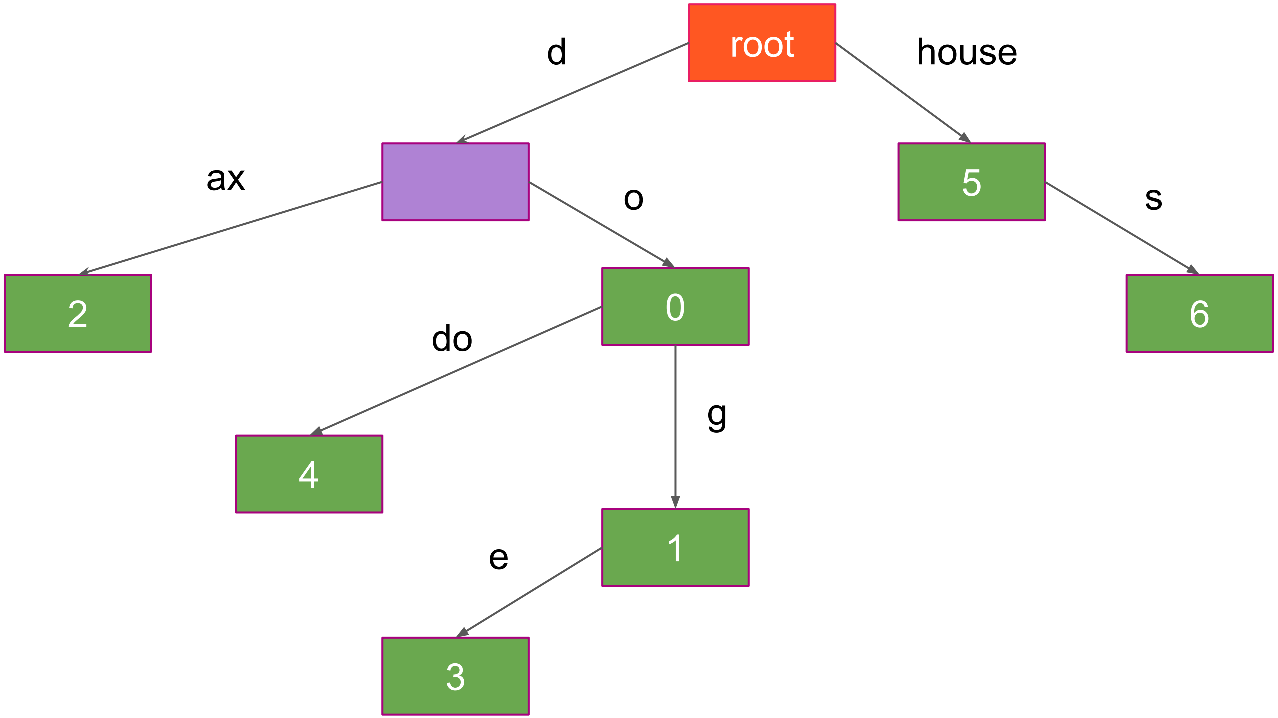
- Space-optimized trie
- An isolated path, with *unmarked* nodes which are *only children*, is merged into single edge
- The label of the merged edge is the concatenation of the labels of merged nodes

Trie vs. Patricia trie



Patricia trie

{ **do**: 0, **dog**: 1, **dax**: 2, **doge**: 3, **dodo**: 4, **house**: 5, **houses**: 6 }

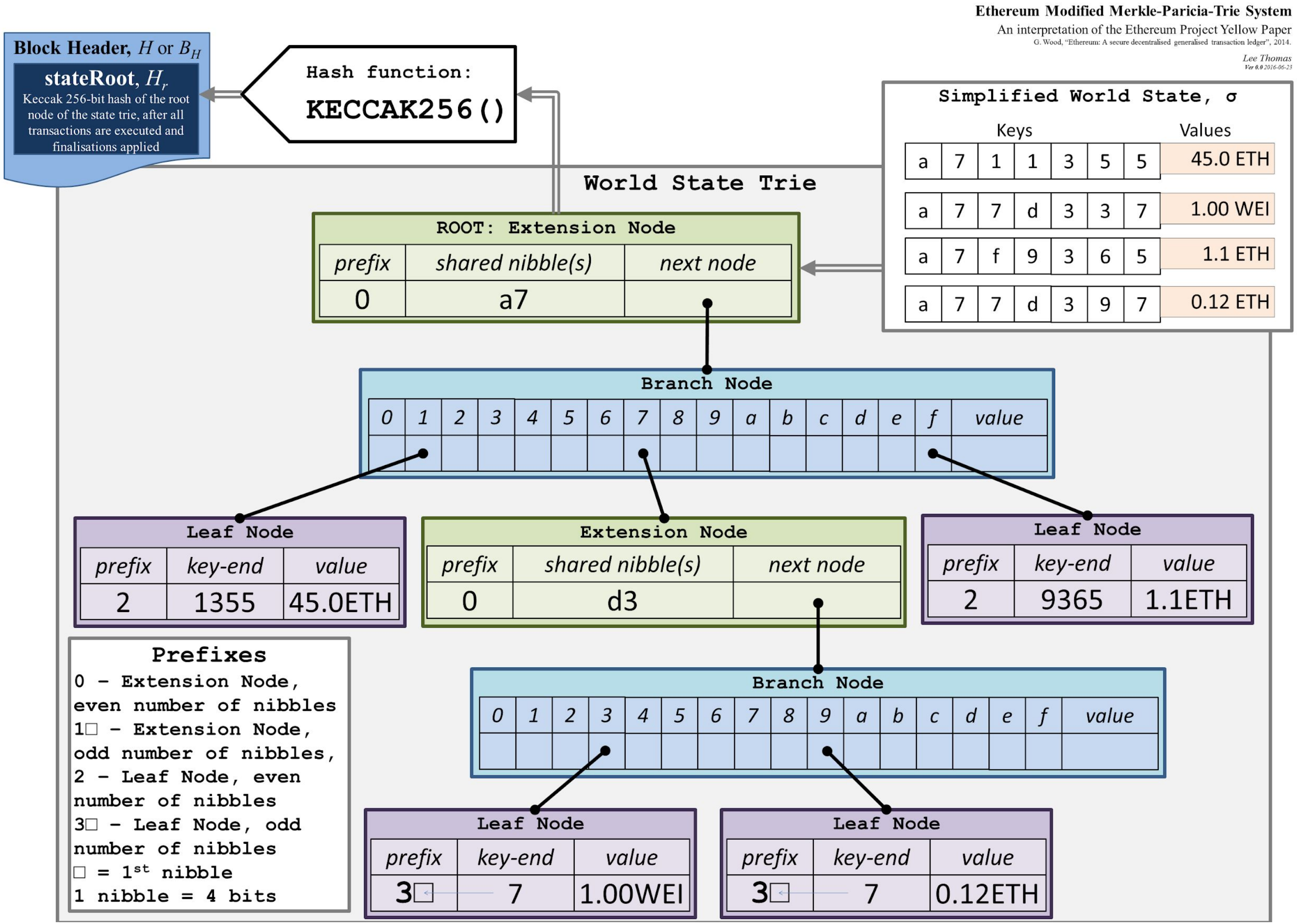


Merkle Patricia trie

- Authenticated Patricia trie
- First implemented in Ethereum
- Allows proof of inclusion (of key, with particular value)
- Allows proof of non-inclusion (by showing key does not exist in trie)

Merkle Patricia trie

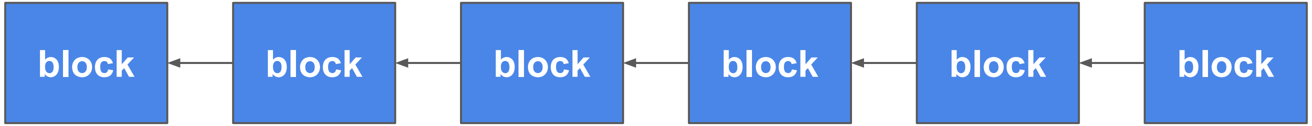
- Split nodes into three types:
  - **Leaf**: Stores edge string leading to it, and **value**
  - **Extension**: Stores **string** of a single edge, **pointer** to next node, and **value** if node marked
  - **Branch**: Stores one pointer to another node per alphabet symbol, and **value** if node marked
- Encode keys as hex, so alphabet size is 16
- Encode all child edges in every node with some encoding (e.g., JSON)
- Pointers are by hash application
- Arguments for correctness and security are same as for Merkle Trees


Ethereum Modified Merkle-Paricia-Trie System
An interpretation of the Ethereum Project Yellow Paper
G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", 2014.
Lee Thomas
Ver 0.0 2016-06-23
Block Header, H or B_H
stateRoot, H_r
Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied
Hash function:
KECCAK256()
Simplified World State, σ
Keys
Values
a 7 1 1 3 5 5 45.0 ETH
a 7 7 d 3 3 7 1.00 WEI
a 7 f 9 3 6 5 1.1 ETH
a 7 7 d 3 9 7 0.12 ETH
World State Trie
ROOT: Extension Node
prefix | shared nibble(s) | next node
0 | a7
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
Leaf Node
prefix | key-end | value
2 | 1355 | 45.0ETH
Extension Node
prefix | shared nibble(s) | next node
0 | d3
Leaf Node
prefix | key-end | value
2 | 9365 | 1.1ETH
Prefixes
0 – Extension Node, even number of nibbles
1□ – Extension Node, odd number of nibbles,
2 – Leaf Node, even number of nibbles
3□ – Leaf Node, odd number of nibbles
□ = 1st nibble
1 nibble = 4 bits
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
Leaf Node
prefix | key-end | value
3□ | 7 | 1.00WEI
Leaf Node
prefix | key-end | value
3□ | 7 | 0.12ETH

Authenticated data in blockchains

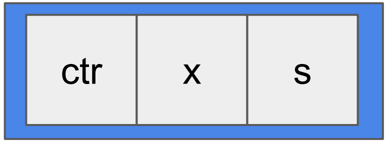
Blockchain

- Each block references a **previous** block
- This reference is by **hash** to its **previous** block
- This linked list is called the **blockchain**



*Convention: Arrows show authenticated inclusion

Blocks



- Data structure with three parts:
  - nonce (ctr), data (**x**), reference (s)
  - Typically called the **block header**
- data (**x**) is application-dependent
  - In Bitcoin it stores financial data ("UTXO"-based)
  - In Ethereum it stores contract data (account-based)
- Block validity:
  - Data must be valid (application-defined validity)
- *s*: pointer to the previous block by hash

Proof-of-work in blocks

- Blocks must satisfy proof-of-work equation

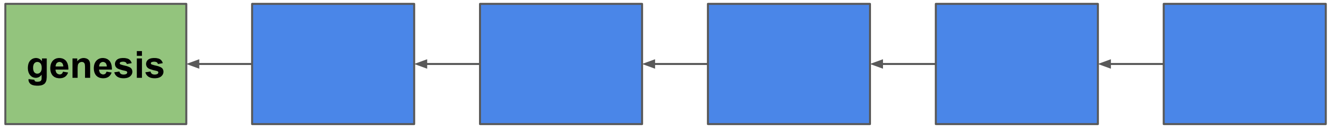
$$H(ctr \,||\, \mathbf{x} \,||\, s) <= T$$

  for some (protocol-specific) T

- ctr is the nonce used to solve Proof-of-work
- The value $H(ctr \,||\, x \,||\, s)$ is known as the **blockid**

Blockchain

- The **first** block of a blockchain is called the Genesis Block

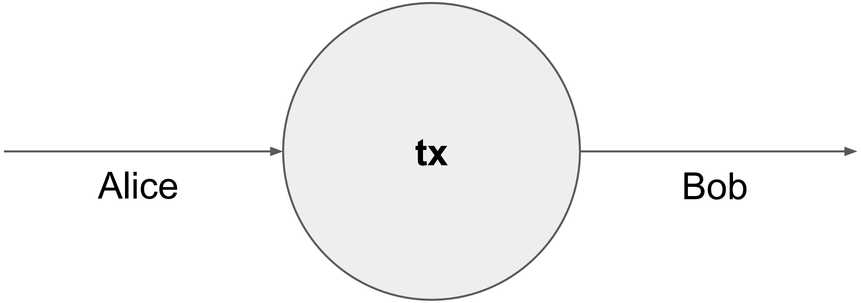


Transactions

A simple transaction for financial data

- Input: contains a proof of spending an existing UTxO*
- Output: contains a verification procedure and a value

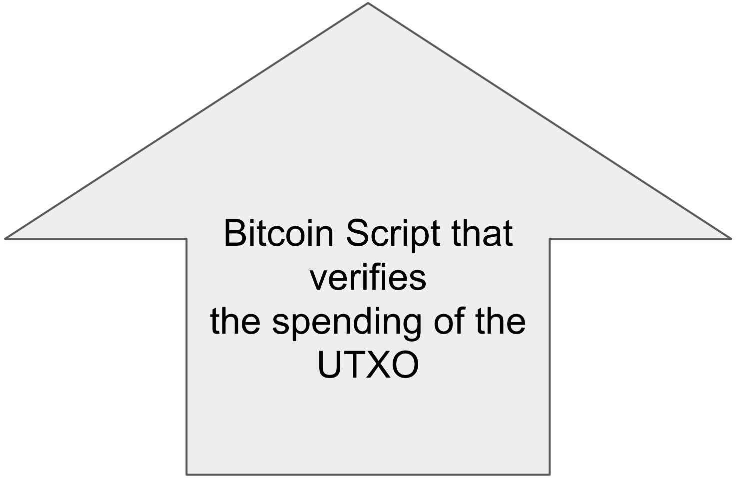
*UTxO = “Unspent Transaction Output”



Transaction Verification

scriptPubKey (output): OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

scriptSig (input): <sig> <pubKey>



Data and Transactions

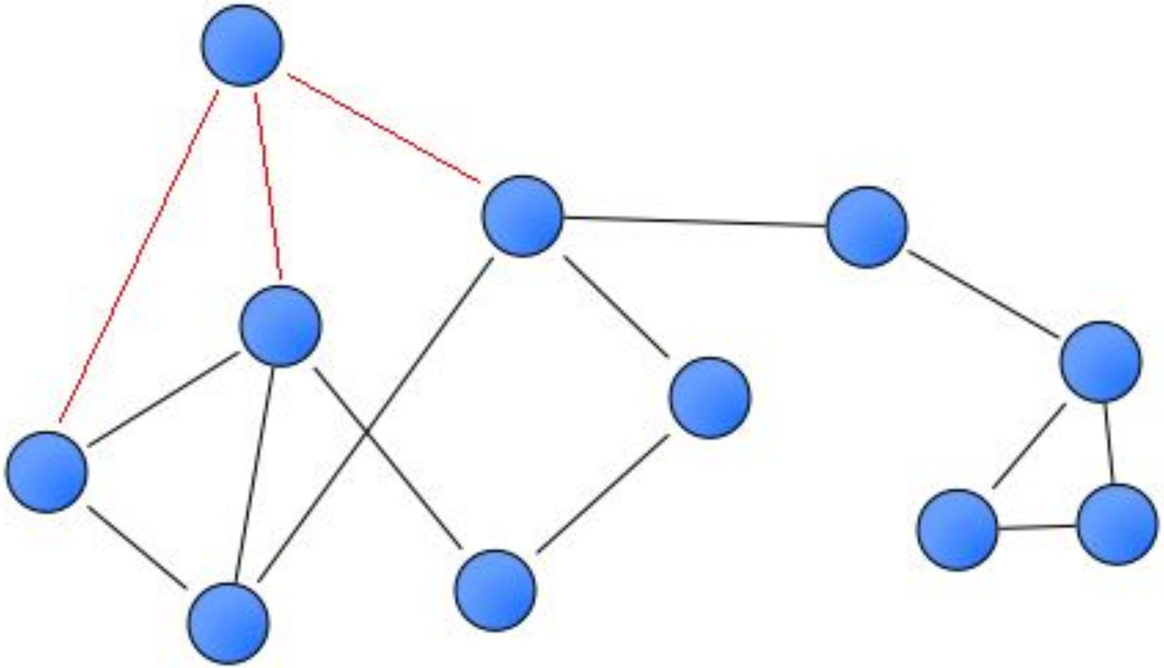
- Financial data is encoded in the form of *transactions*
- Each block organizes transactions in an authenticated data structure
  - Bitcoin: Merkle Tree
  - Ethereum: Merkle Patricia Trie
- Every transaction is sent on the network to everyone via a gossip protocol

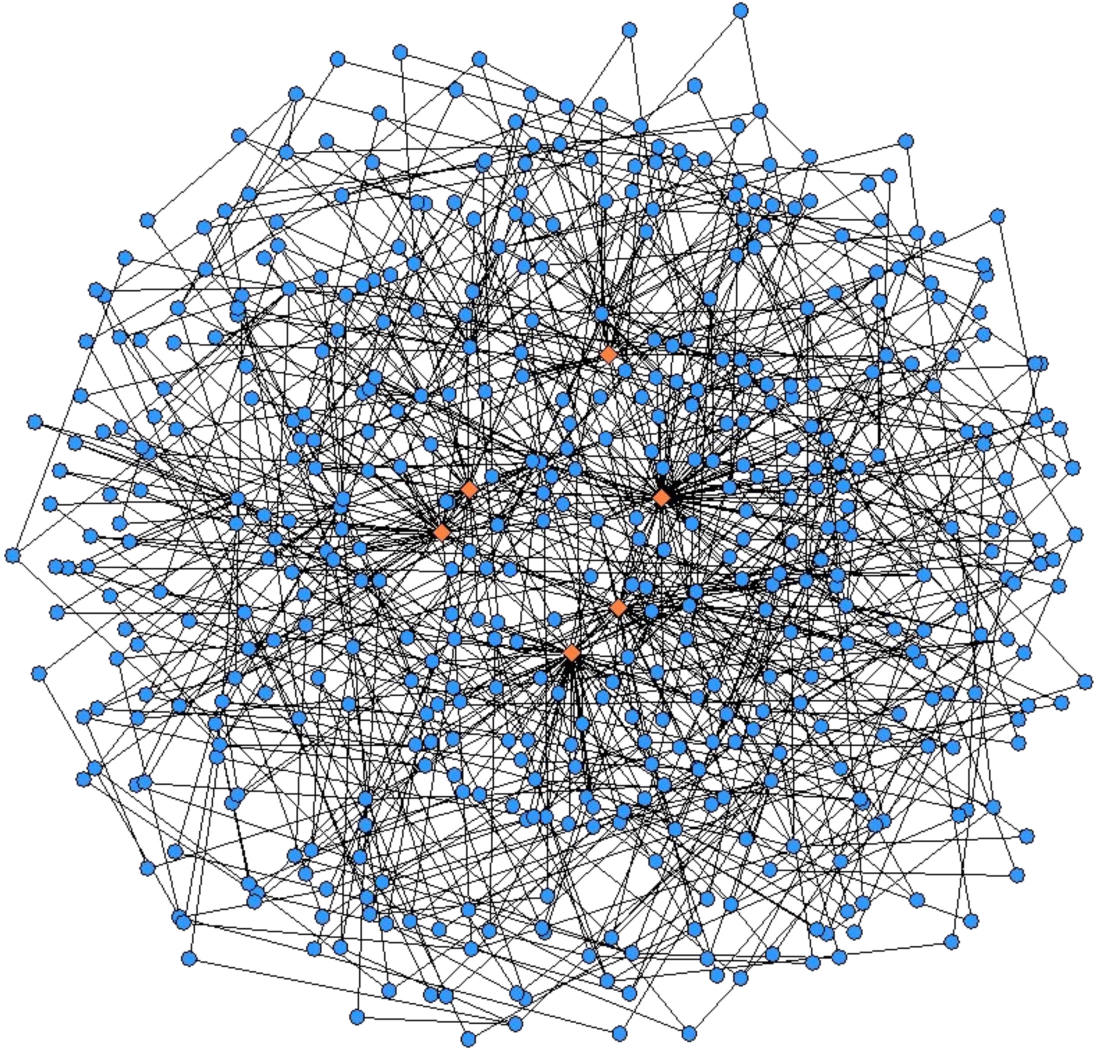
- Question: Is it necessary to download the entire block (header + transactions) to verify whether a transaction is included in it?

The Bitcoin network

The bitcoin network

- All bitcoin nodes connect to a common p2p network
- Each node runs (code that implements) the Bitcoin protocol
- Open source code
- Each node connects to its (network) neighbours
- They continuously exchange data
- Each node can **freely** enter the network – no permission needed!
    - A “permissionless network”
- **The adversarial assumption:**
  There is no trust on the network! Each neighbour can lie.





BITNODES

Bitnodes estimates the relative size of the Bitcoin peer-to-peer network by finding all of its reachable nodes.

REACHABLE BITCOIN NODES

Updated: Wed Sep 28 16:01:53 2022 BST

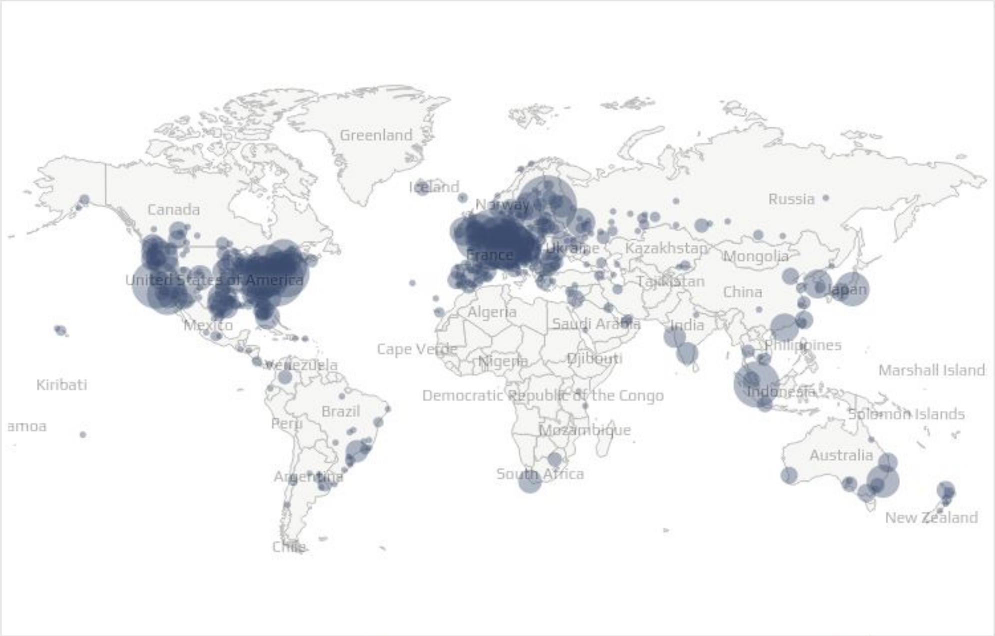
13543 NODES CHARTS

IPv4: -1.5% / IPv6: -1.4% / .onion: -9.1%

Top 10 countries with their respective number of reachable nodes are as follows.

RANK	COUNTRY	NODES
1	n/a	6563 (48.46%)
2	United States	1931 (14.26%)
3	Germany	1399 (10.33%)
4	France	424 (3.13%)
5	Netherlands	381 (2.81%)
6	Canada	313 (2.31%)
7	Finland	243 (1.79%)
8	United Kingdom	229 (1.69%)
9	Russian Federation	196 (1.45%)
10	Singapore	145 (1.07%)

All (95) »



Map shows concentration of reachable Bitcoin nodes found in countries around the world. LIVE MAP

https://bitnodes.io/

Peer discovery

- Each node stores a list of peers (by IP address)
- When Alice connects to Bob, Bob sends Alice his own known peers
- That way, Alice can learn about new peers

Bootstrapping the p2p network

- Peer-to-peer nodes come “pre-installed” with some peers by IP / host
- When running a node, you can specify extra “known peers”

The *gossip* protocol

- **Alice** generates some new data
- Alice **broadcasts** data to its peers
- Each peer broadcasts this data to *its* peers
- If a peer has seen this data before, it ignores it
- If this data is new, it broadcasts it to its peers
- That way, the data spreads like an epidemic, until the whole network learns it
- This process is called **diffuse**

Eclipse attacks

- Isolate some honest nodes in the network, effectively causing a “network split” in two partitions A and B
- If peers in A and peers in B are disjoint and don’t know about each other, the networks will remain isolated
  - Recent attack: Erebus

- The connectivity assumption:
  - There is a path between two nodes on the network
  - **If a node broadcasts a message, every other node *will* learn it**