Blockchains & Distributed Ledgers

Lecture 02

Aggelos Kiayias

Slide credits: AK, Dionysis Zindros, Christos Nasikas

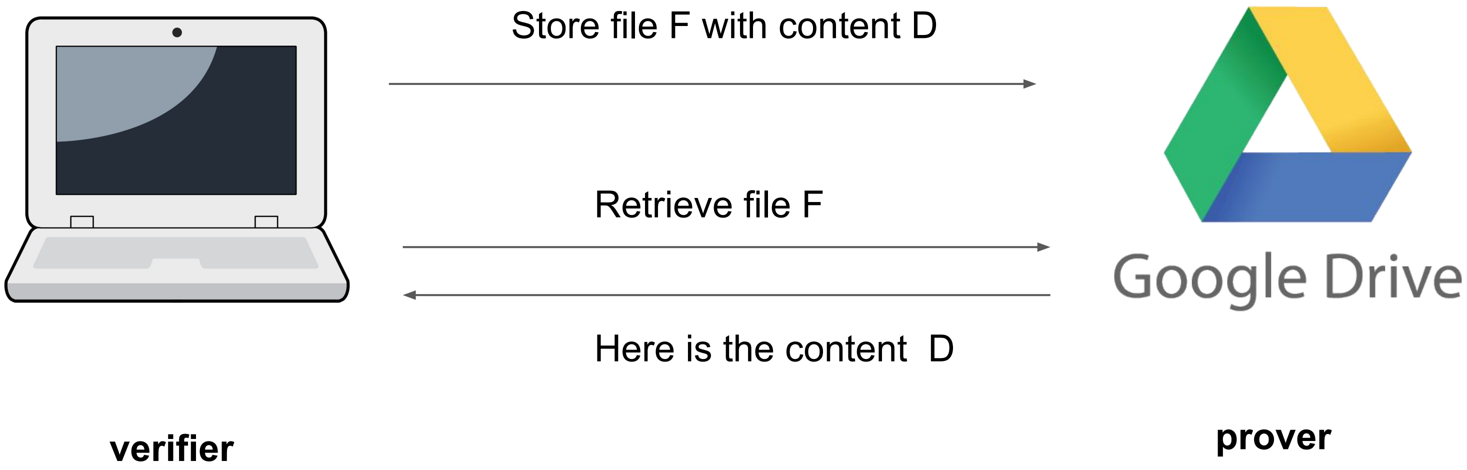
Overview

- Motivation: Server file storage
- Merkle trees to store lists
- Proofs-of-inclusion
- Merkle trees to store sets
- Proofs-of-non-inclusion
- Merkle–Patricia tries to store key:value pairs
- Blocks and blockchains

Authenticated Data Structures

- Like regular data structures, but cryptographically authenticated
- Allows a **verifier** to store, retrieve and operate on data with an untrusted **prover**

The authenticated file storage problem



The file storage problem

- Client wants to store a file on a server
- File has a name F and content D
- Clients wants to retrieve file F later

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: Protocol against adversaries

- What if **server is adversarial** 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 doesn’t have enough memory to store D for a long time?

File storage: Hash-based protocol

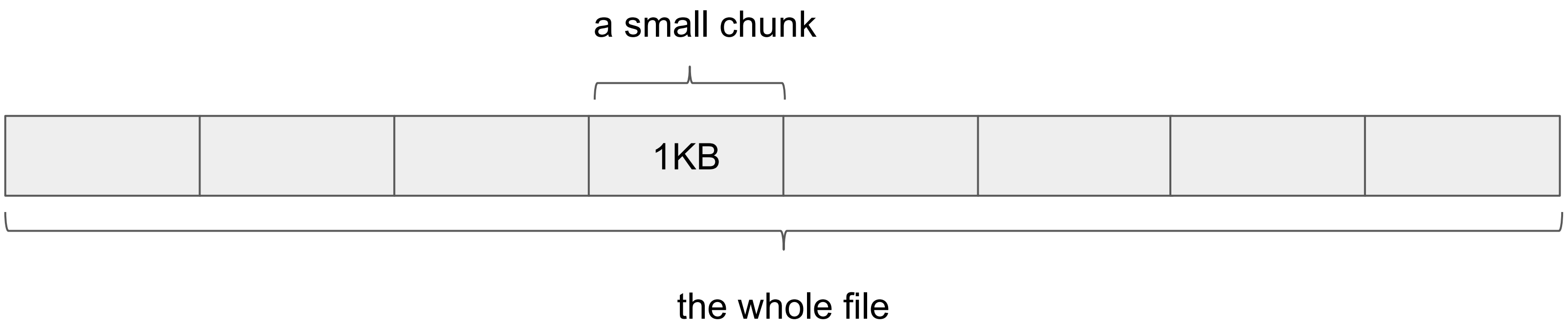
- Client sends file F with data D to server
- Server stores (F, D)
- Client stores H(D), deletes D
- Client requests F from server
- Server returns D’
- Client compares H(D’) = H(D)

File storage: File chunks

- What if client wants to retrieve the 200,019th byte of the file?
- Must download the whole file…
- Merkle trees to the rescue!

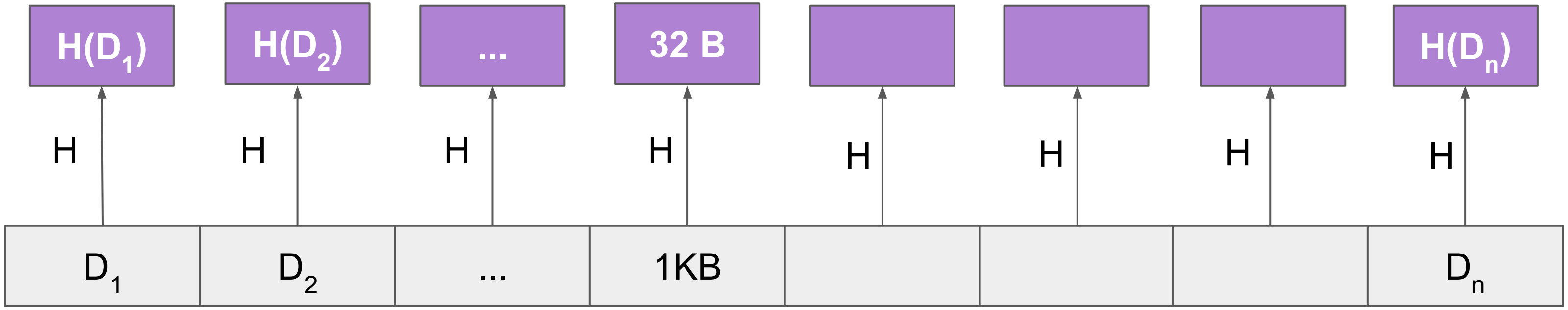
Merkle Tree

- An **authenticated** binary tree
- Split file into **chunks** of, say, 1KB



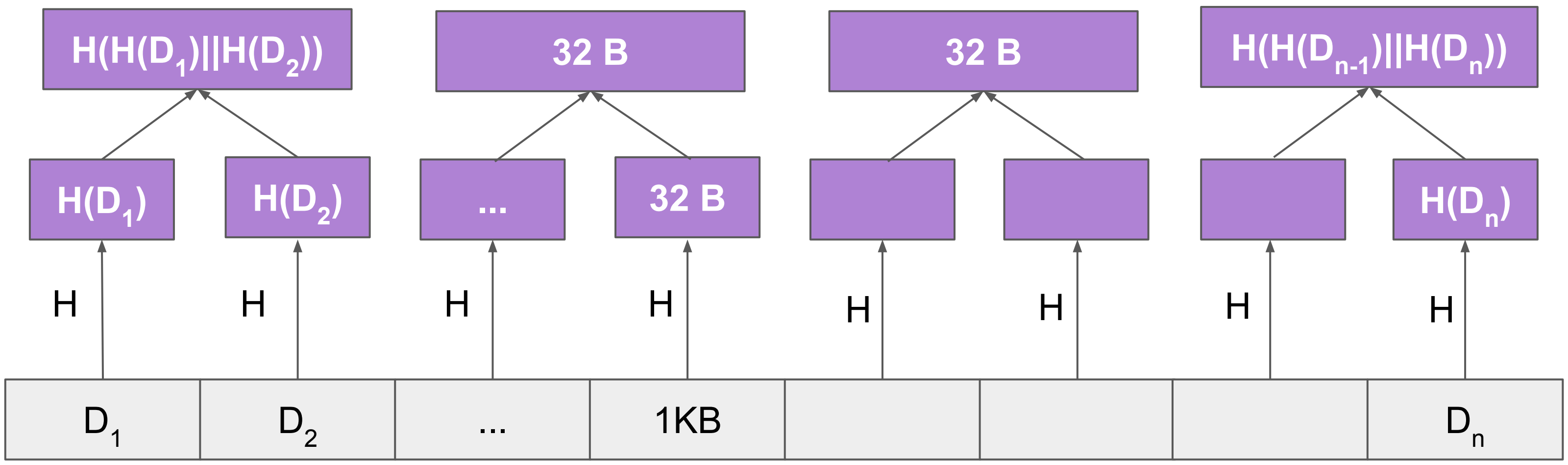
Merkle Tree

- **Hash** each chunk using a cryptographic hash function (SHA256)
- Convention: Arrows show direction of hash function application

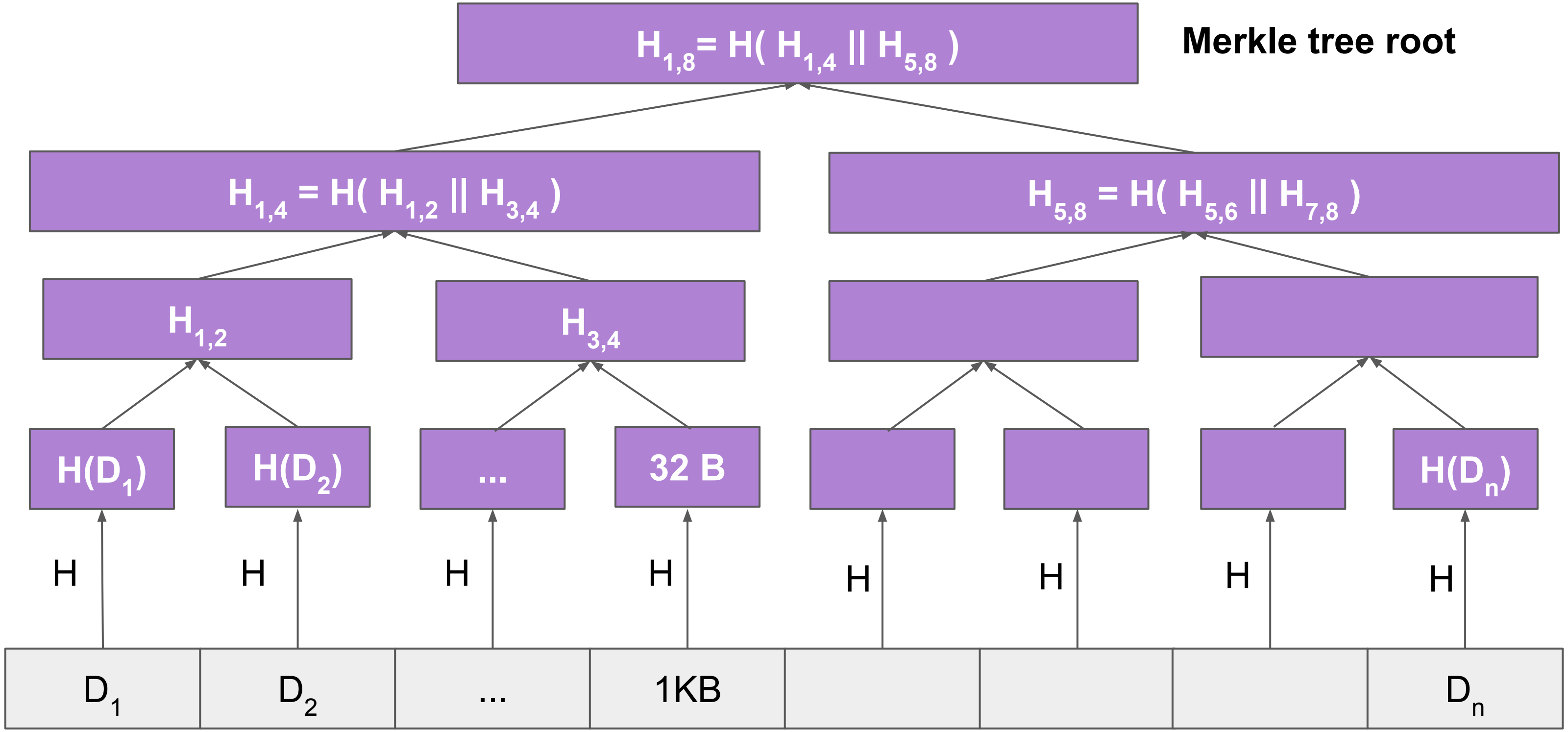


Merkle Tree

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



Merkle Tree

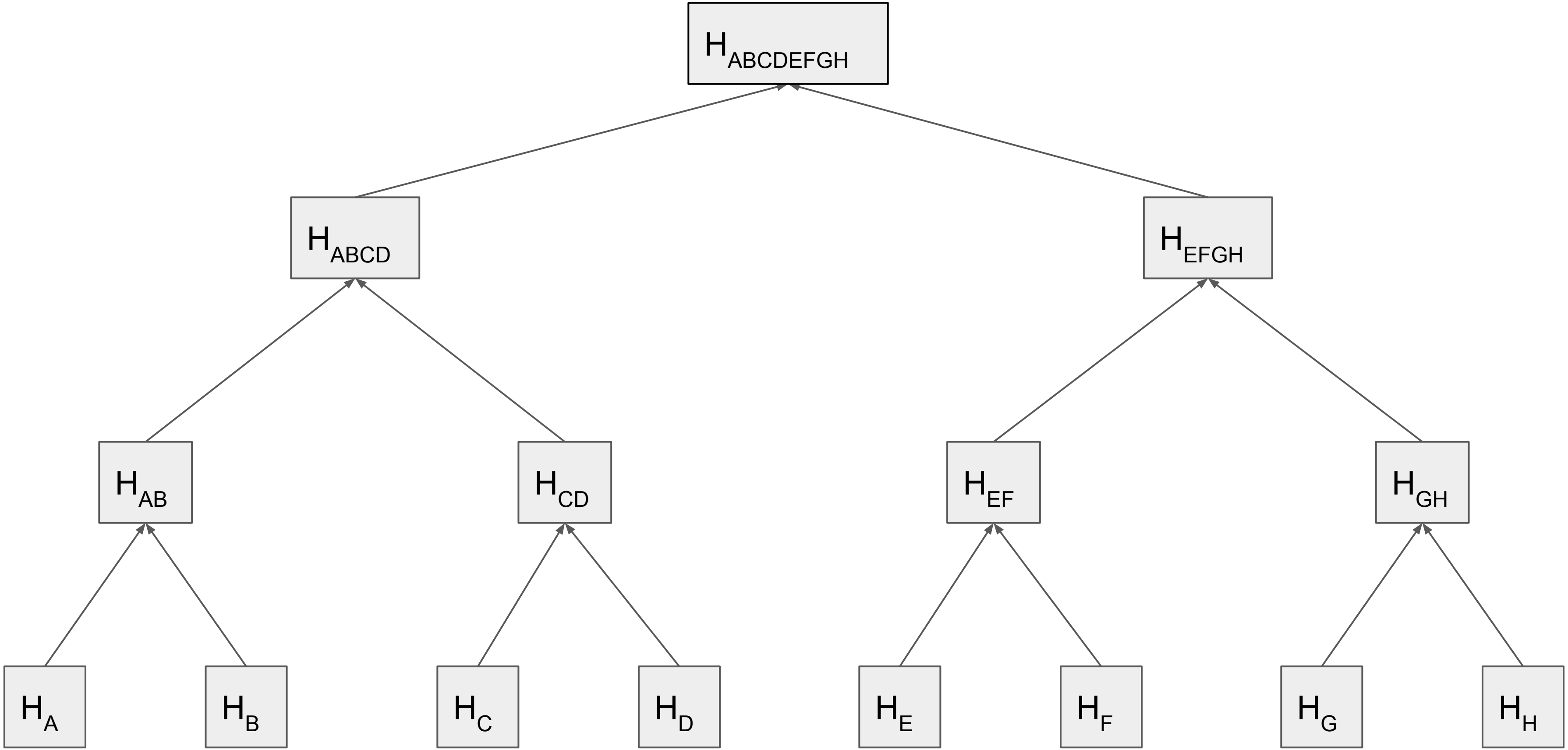


Merkle tree root
$H_{1,8} = H(H_{1,4} \| H_{5,8})$
$H_{1,4} = H(H_{1,2} \| H_{3,4})$
$H_{5,8} = H(H_{5,6} \| H_{7,8})$
$H_{1,2}$
$H_{3,4}$
$H(D_1)$
$H(D_2)$
...
32 B
$H(D_n)$
H
H
H
H
H
H
H
H
D_1
D_2
...
1KB
D_n

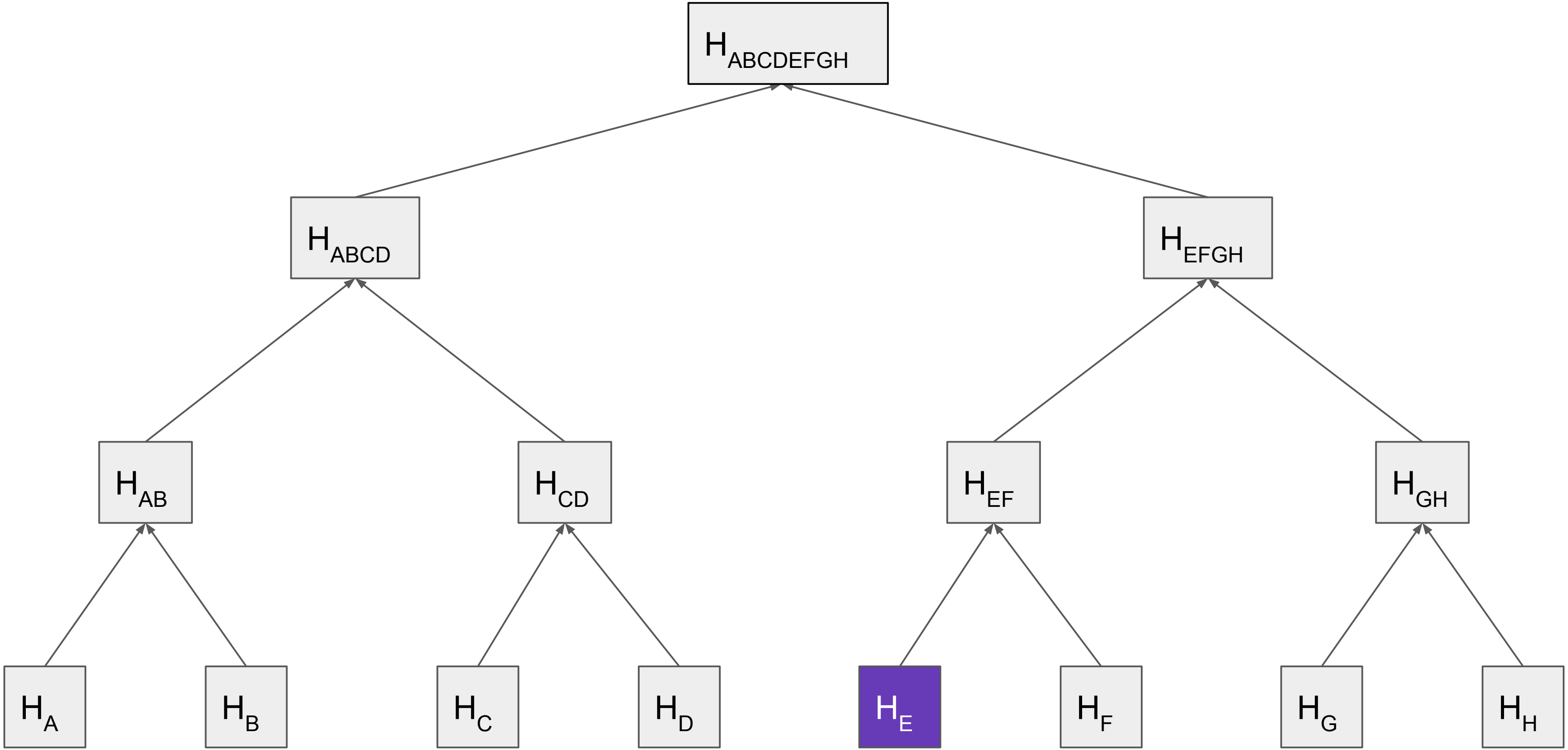
Proofs-of-inclusion

- Client creates Merkle Tree root **MTR** from initial file data D
- Client sends file data D to server
- Client deletes data D, but stores MTR (32 bytes)
- Client requests chunk x from server
- Server returns chunk x and short proof-of-inclusion π
- Client checks that chunk x is included in MTR using proof π

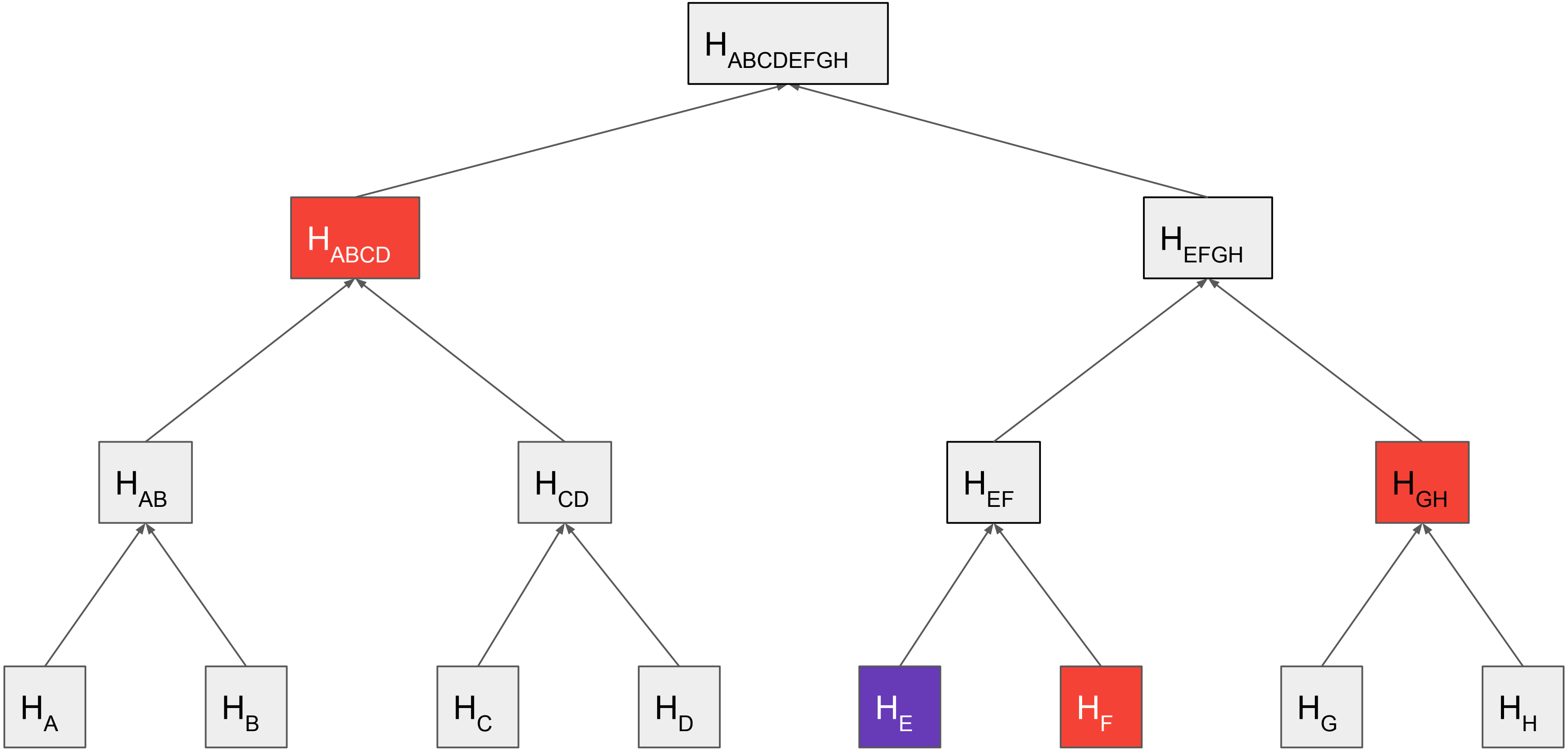
Merkle tree: proof of inclusion



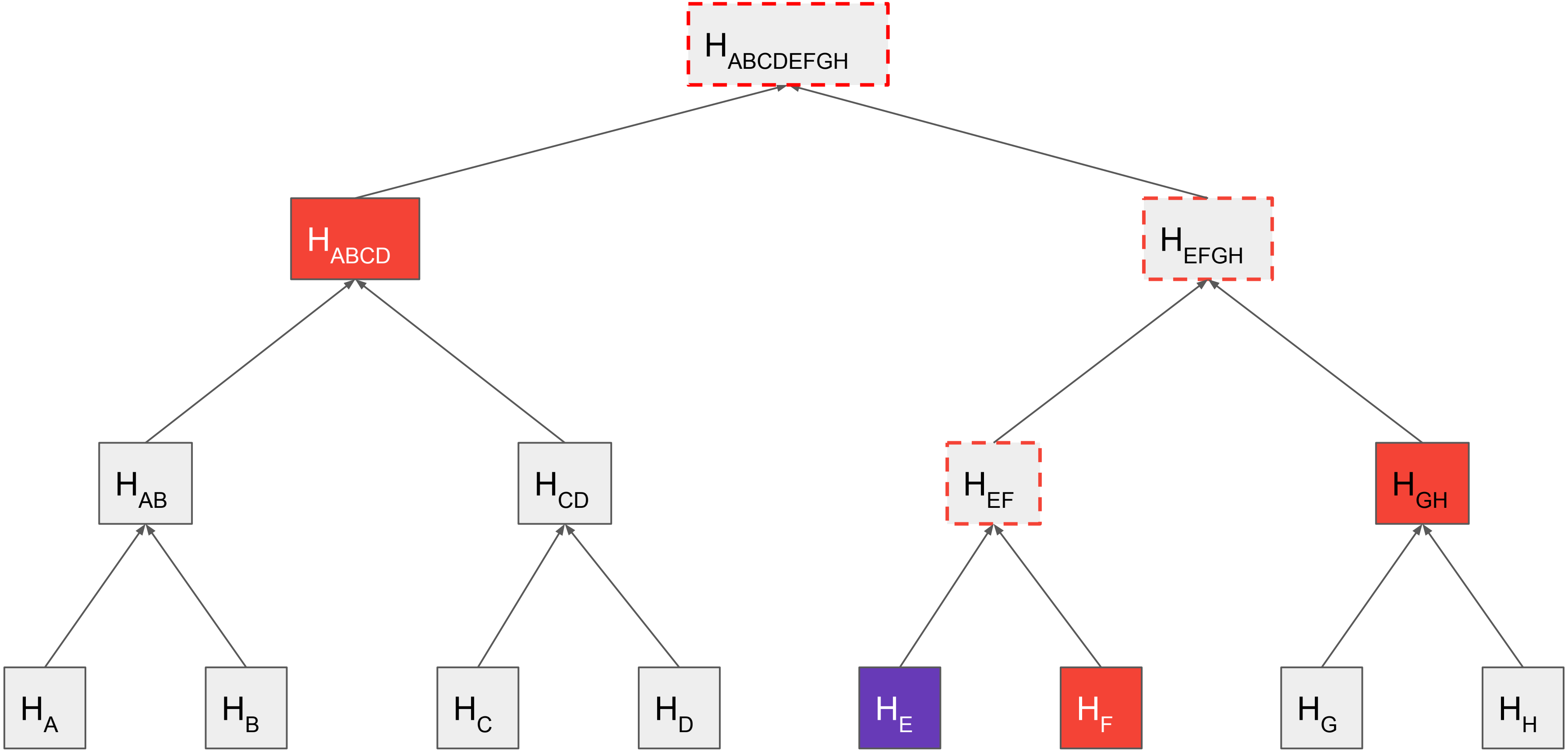
Merkle tree: proof of inclusion



Merkle tree: proof of inclusion



Merkle tree: 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 = MTR
- How big is proof-of-inclusion?

Proof-of-inclusion succinctness

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

Notation :
lg = log base 2

Merkle Tree proof-of-inclusion security

- If adversary can present proof-of-inclusion for incorrect leaf, then we can break the hash function

Merkle tree applications

- Bitcoin uses Merkle trees to store transactions
- BitTorrent uses Merkle Tree to exchange files
- Ethereum uses Merkle–Patricia tries for storage and transactions

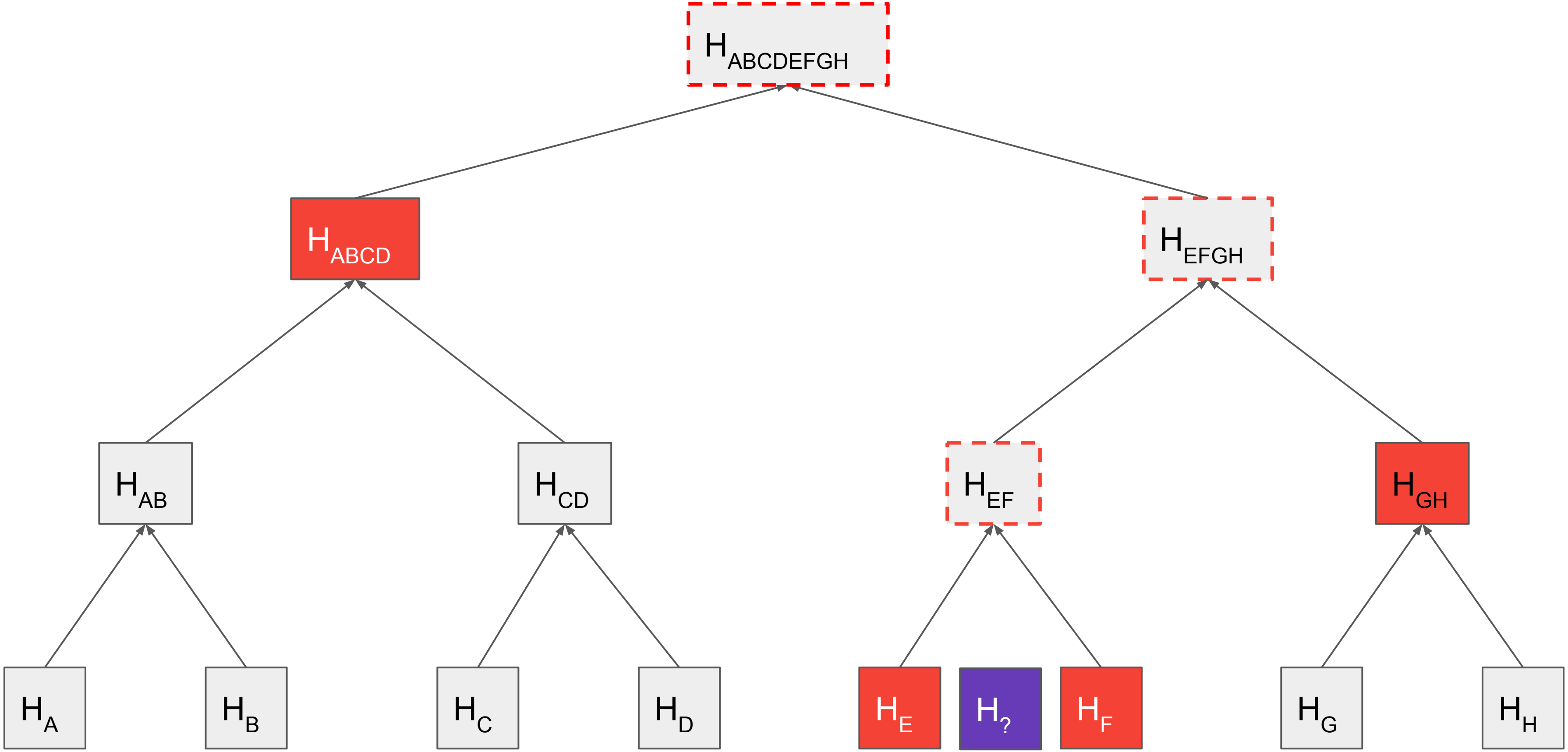
Storing *sets* instead of lists

- 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
- 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 $H_> > x$
- The two proofs-of-inclusion can be compressed into one

Merkle tree: proof of inclusion / non-inclusion

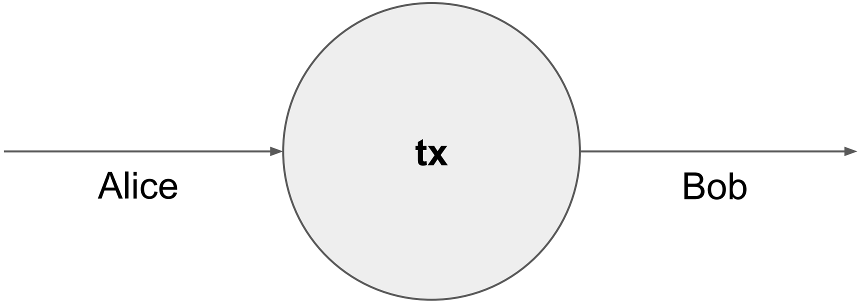


Transactions

A simple transaction for financial data

- Input: contains a signature and a public-key
- Output: contains a verification procedure and a value

UTXO = “unspent transaction output”



Example from Bitcoin

Input:
Previous tx: f5d8ee39a430901c91a5917b9f2dc19d6d1a0e9cea205b009ca73dd04470b9a6
Index: 0
scriptSig: 304502206e21798a42fae0e854281abd38bacd1aeed3ee3738d9e1446618c4571d10
90db022100e2ac980643b0b82c0e88ffdfec6b64e3e6ba35e7ba5fdd7d5d6cc8d25c6b241501

Output:
Value: 5000000000
scriptPubKey: OP_DUP OP_HASH160 404371705fa9bd789a2fcd52d2c580b65d35549d
OP_EQUALVERIFY OP_CHECKSIG

The input imports 50 BTC from output #0 of tx f5d..
and sends them to a Bitcoin address 404…

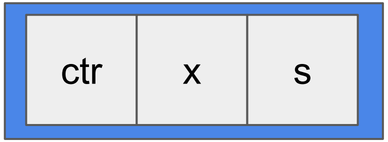
Transaction Verification

scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

scriptSig: <sig> <pubKey>

Bitcoin
Script
That verifies
The
spending of
the UTXO

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)
    - In Namecoin it stores name data
    - We leave this undefined for now -- we will come back to this in future lectures
- Block validity:
    - Data must be valid (application-defined validity)

Proof-of-work in blocks

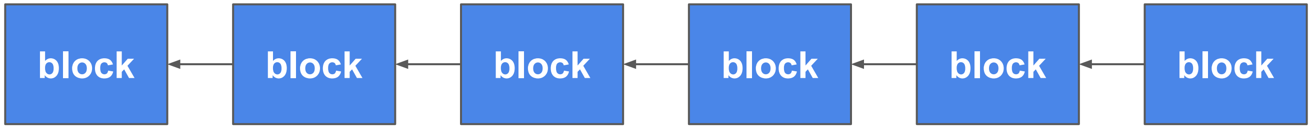
- Blocks must satisfy proof-of-work equation

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

- for some constant T
- ctr is the nonce used to solve proof-of-work
- The value H(ctr || x || s) is known as the **blockid**

Blockchain

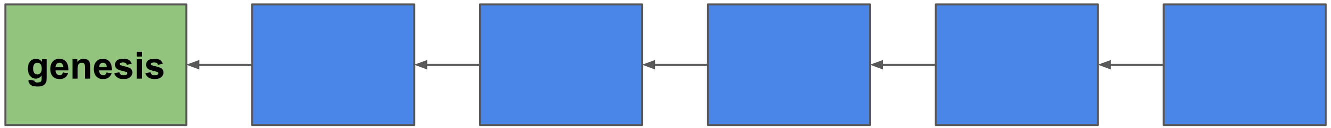
- Each block references a **previous** block
- This reference is by **hash** to its **previous** block, similar to Merkle Trees
- This linked list is called the **blockchain**
- Convention: Arrows show authenticated inclusion



- Blocks use the *s* value to point to the previous block by hash

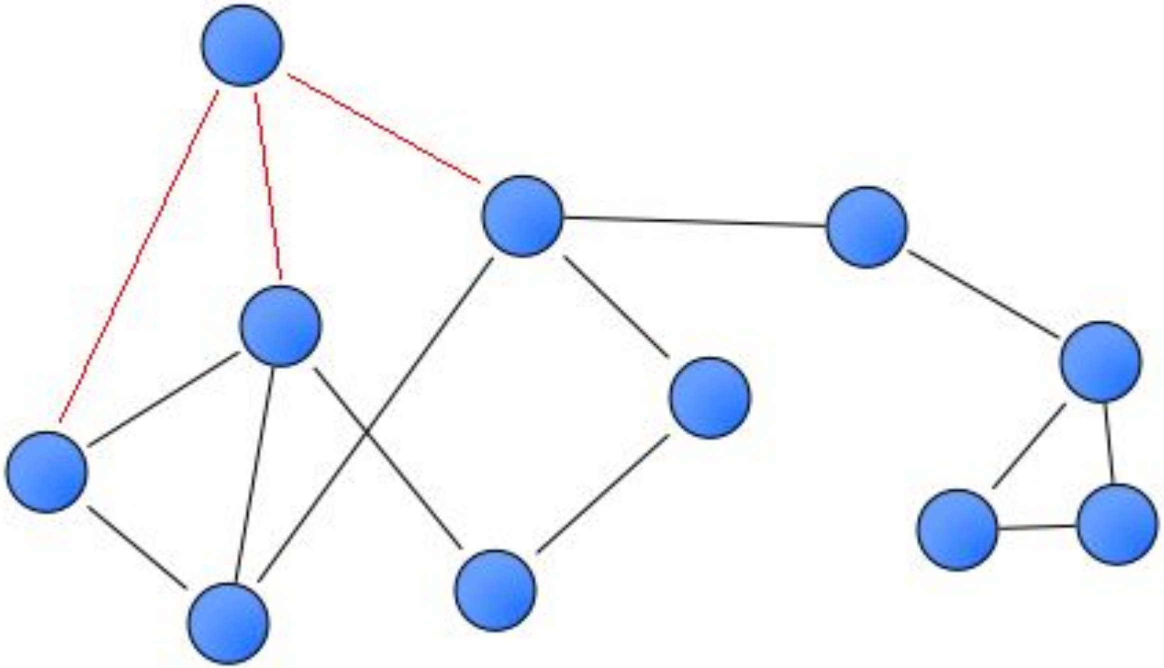
Blockchain

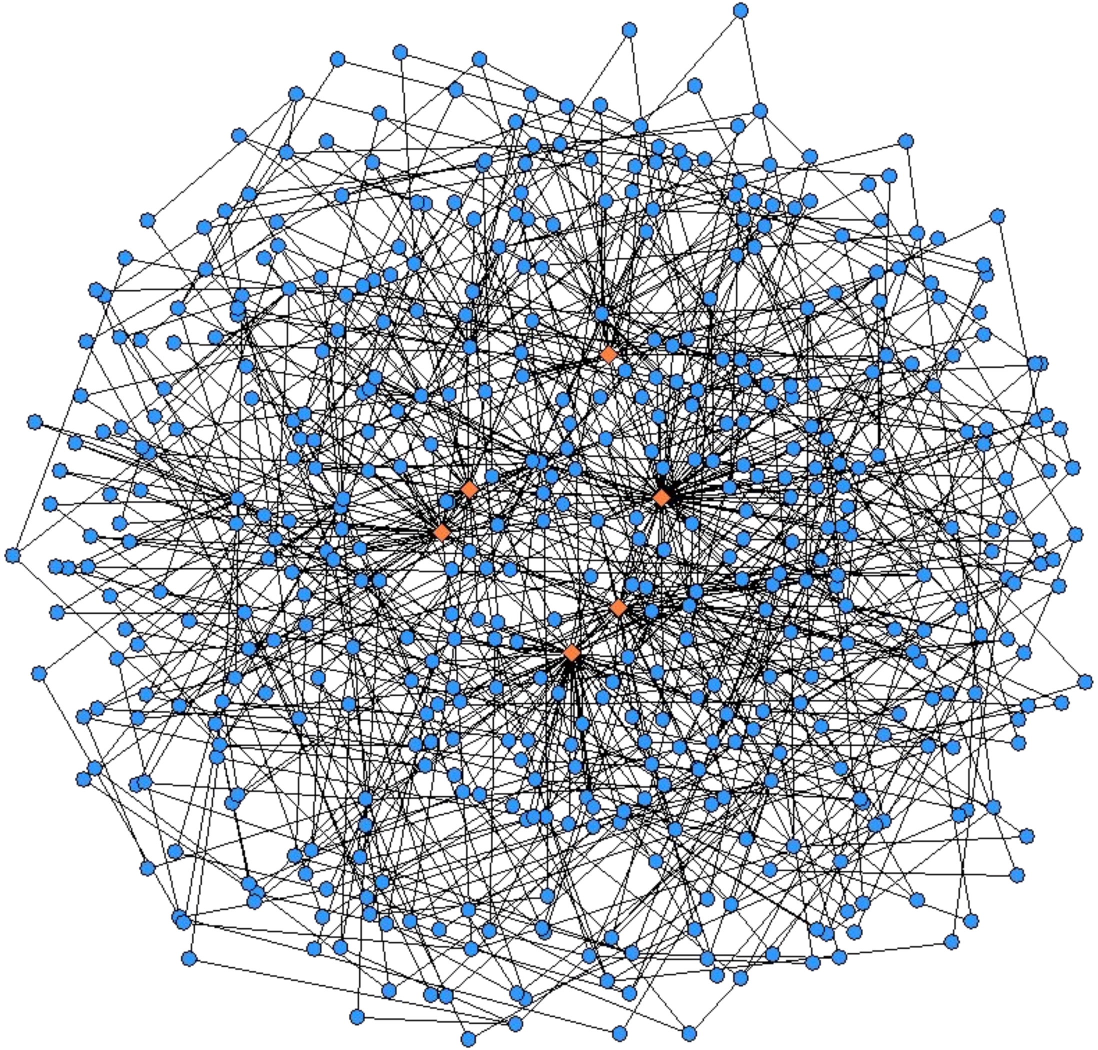
- The **first** block of a blockchain is called the Genesis Block

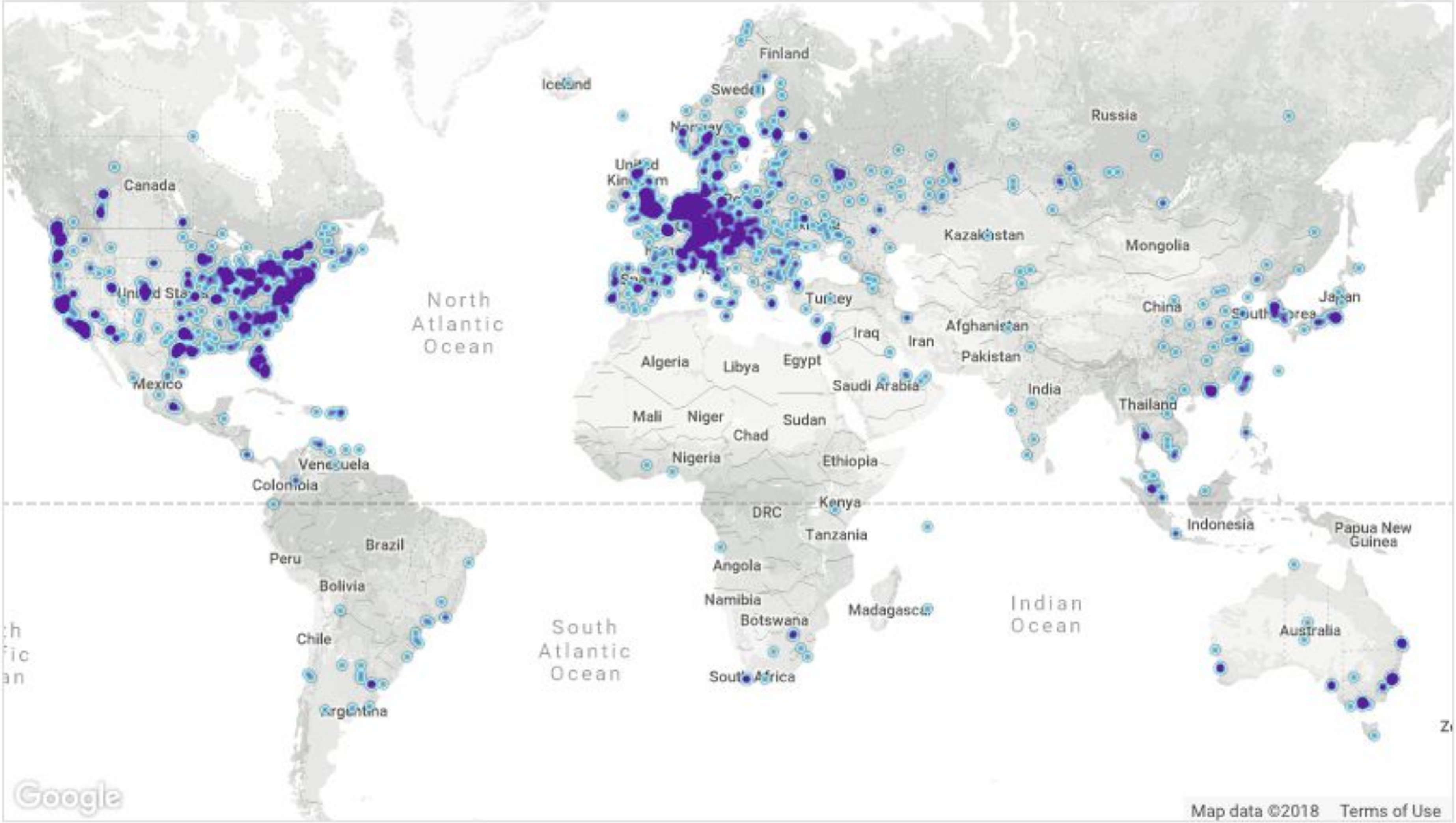


The bitcoin network

- All bitcoin nodes connect to a common p2p network
- Each node runs the code of bitcoin
- A node can run on a phone, computer, etc.
- Open source code
- Each node connects to its neighbours
- They continuously exchange financial 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.






Finland
Russia
Canada
Kazakhstan
Mongolia
China
North Atlantic Ocean
Turkey
Iraq
Iran
Afghanistan
Pakistan
Algeria
Libya
Egypt
Saudi Arabia
India
Mexico
Thailand
Mali
Niger
Sudan
Chad
Nigeria
Ethiopia
Colombia
Kenya
DRC
Tanzania
Indonesia
Papua New Guinea
Brazil
Peru
Angola
Bolivia
Namibia
Botswana
Indian Ocean
South Atlantic Ocean
Chile
Australia
Google
Map data ©2018
Terms of Use

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 “preinstalled” with some peers by IP / host
- When running a node, you can specify extra “known peers”

The *gossip* protocol

- When a node **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**

Financial data and Transactions

- Financial data is encoded in the form of *transactions*
- Every transaction is broadcast on the network to everyone using the gossip protocol

Transactions on blockchain.info

Tries

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

Tries

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

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

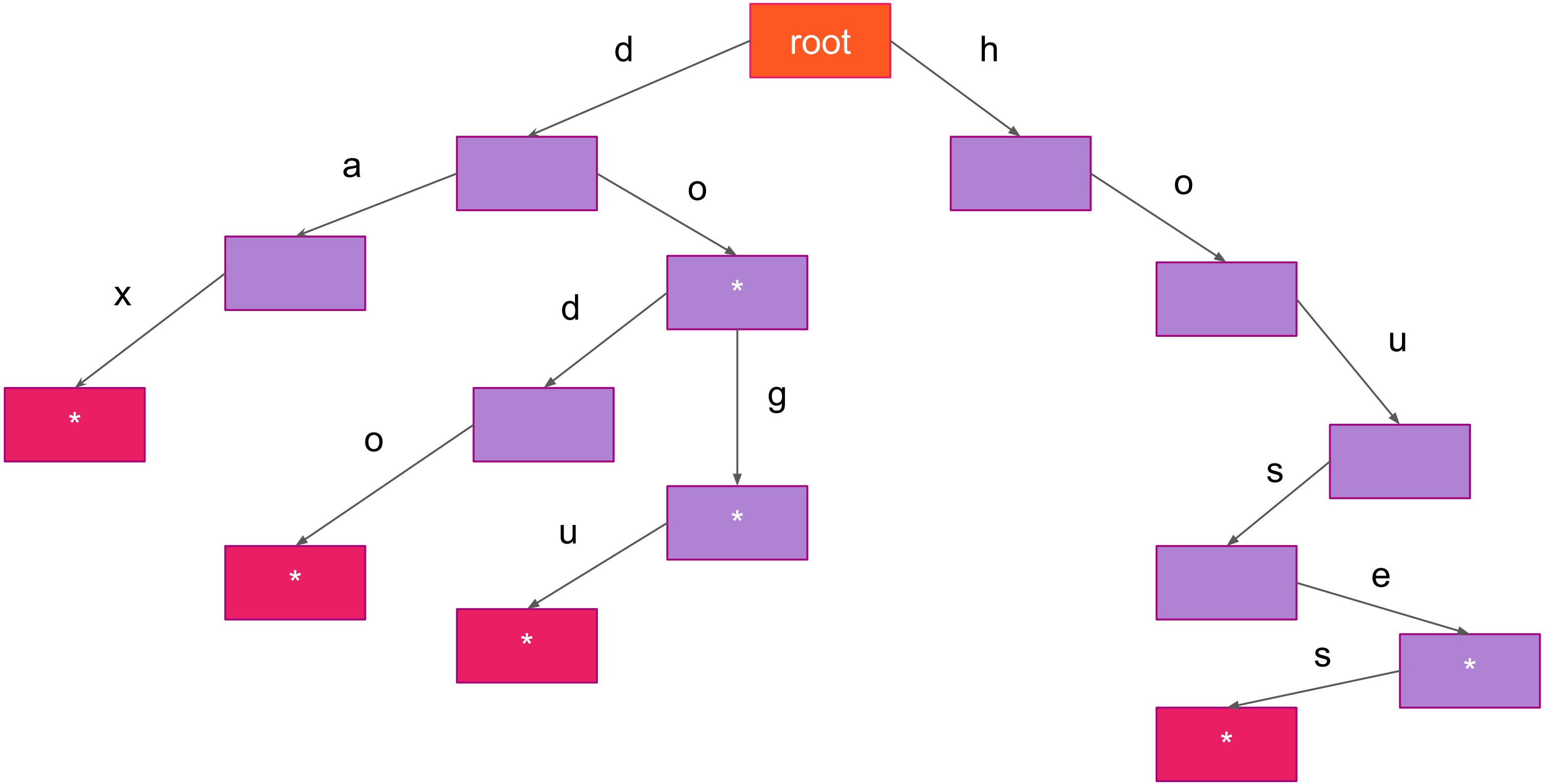
Tries: query(string)

- 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**
- Otherwise, return **no**

Tries: example

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

Tries



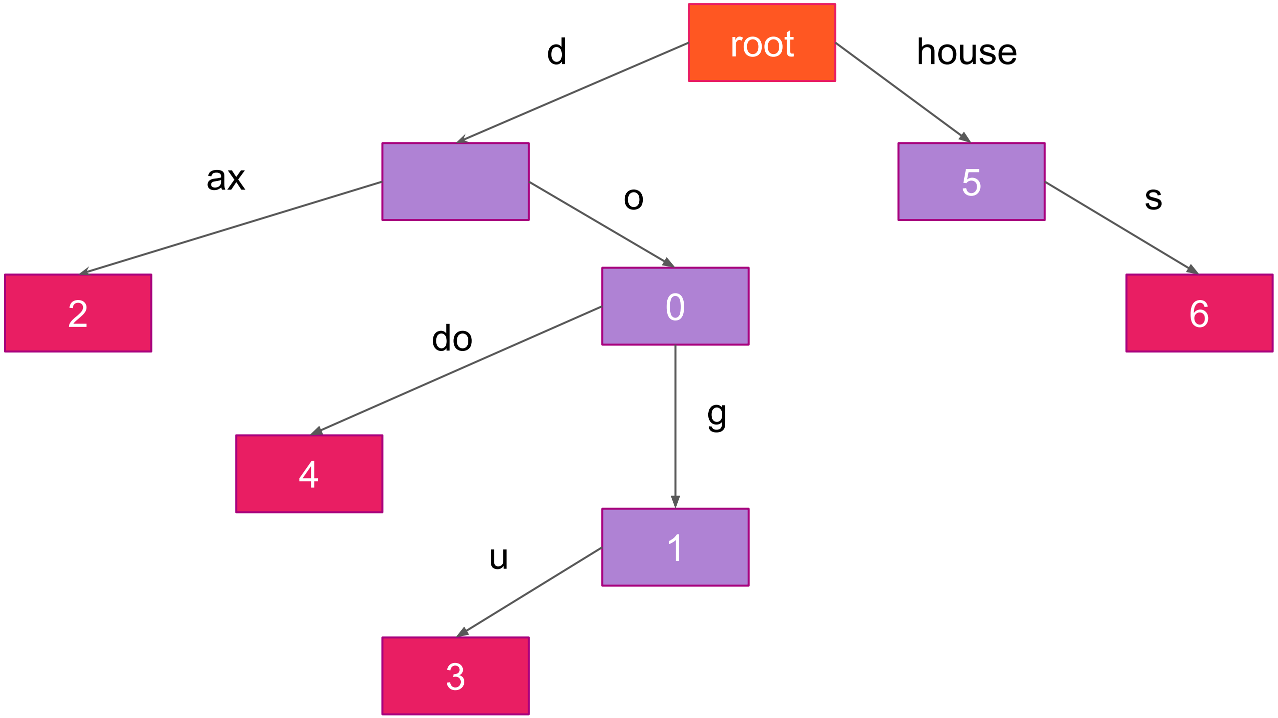
Patricia (or radix) trie

- Space-optimized trie
- An isolated path (with nodes which are only children) with unmarked nodes is *merged* into one edge
- The label of the merged edge is the concatenation of the merged symbols

Tries / Patricia tries as key/value store

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

Patricia trie



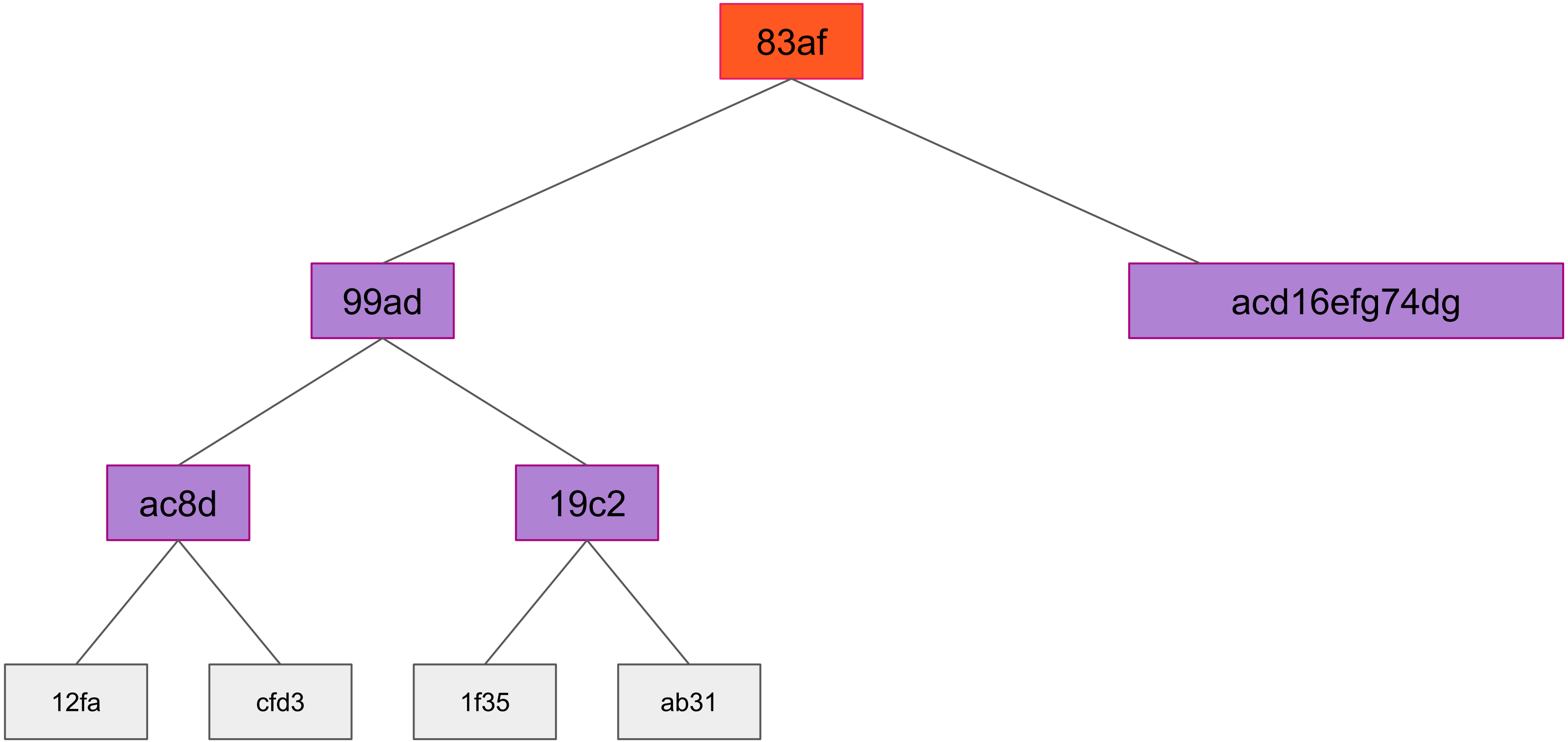
Merkle Patricia trie

- An 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
- We encode keys as hex, so alphabet size is 16
- We 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

Merkle Patricia trie

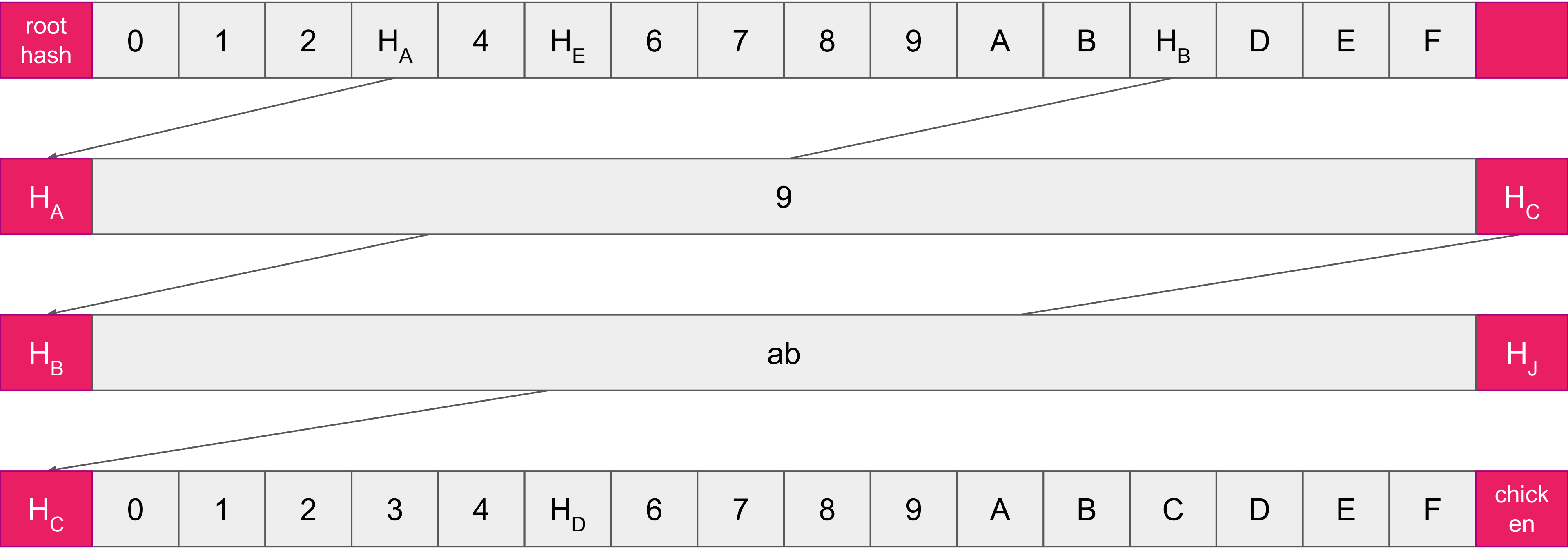


Merkle patricia trie: node

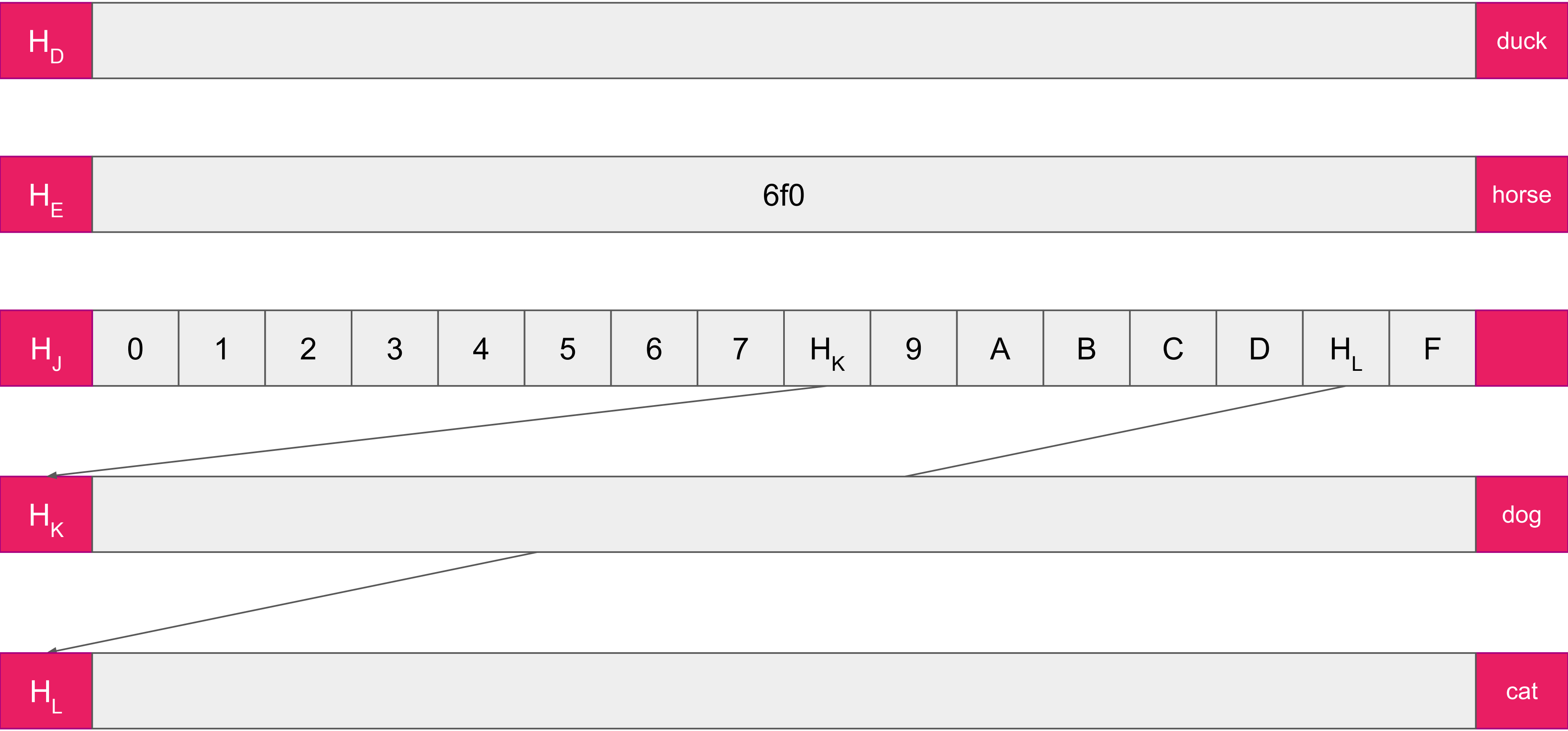
key	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	value

Merkle patricia trie: example

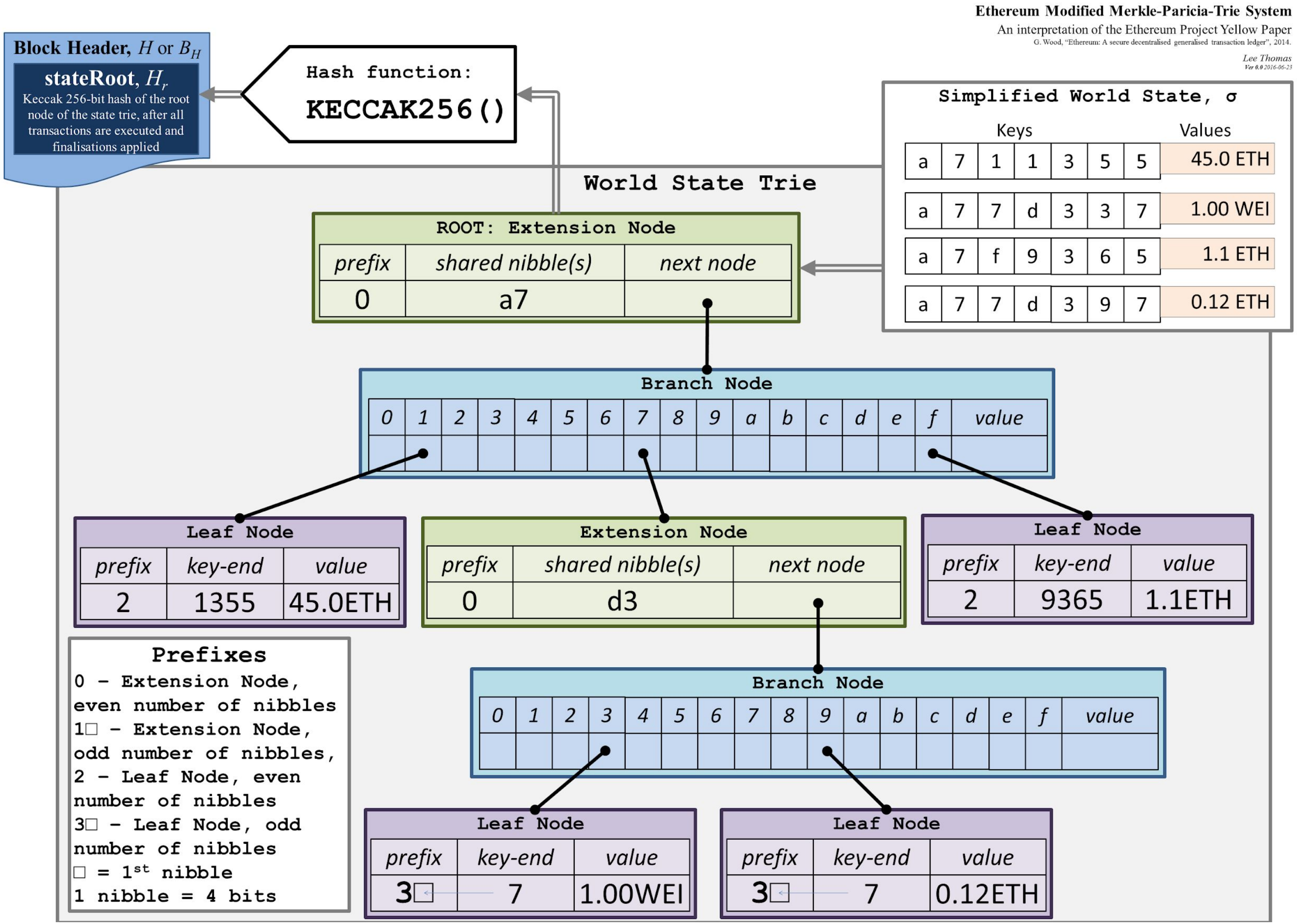
{ **'cab8'**: 'dog', **'cabe'**: 'cat', **'39'**: 'chicken', **'395'**: 'duck', **'56f0'**: 'horse' }



root hash
0
1
2
H_A
4
H_E
6
7
8
9
A
B
H_B
D
E
F
H_A
9
H_C
H_B
ab
H_J
H_C
0
1
2
3
4
H_D
6
7
8
9
A
B
C
D
E
F
chick en



H_D
duck
H_E
6f0
horse
H_J
0
1
2
3
4
5
6
7
H_K
9
A
B
C
D
H_L
F
H_K
dog
H_L
cat



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
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

Eclipse attacks

- Isolate the some honest nodes in the network effectively causing a “net 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
    - More 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