Authenticated Data Structures

University of Athens
Aggelos Kiayias, Christos Nasikas, Dionysis Zindros

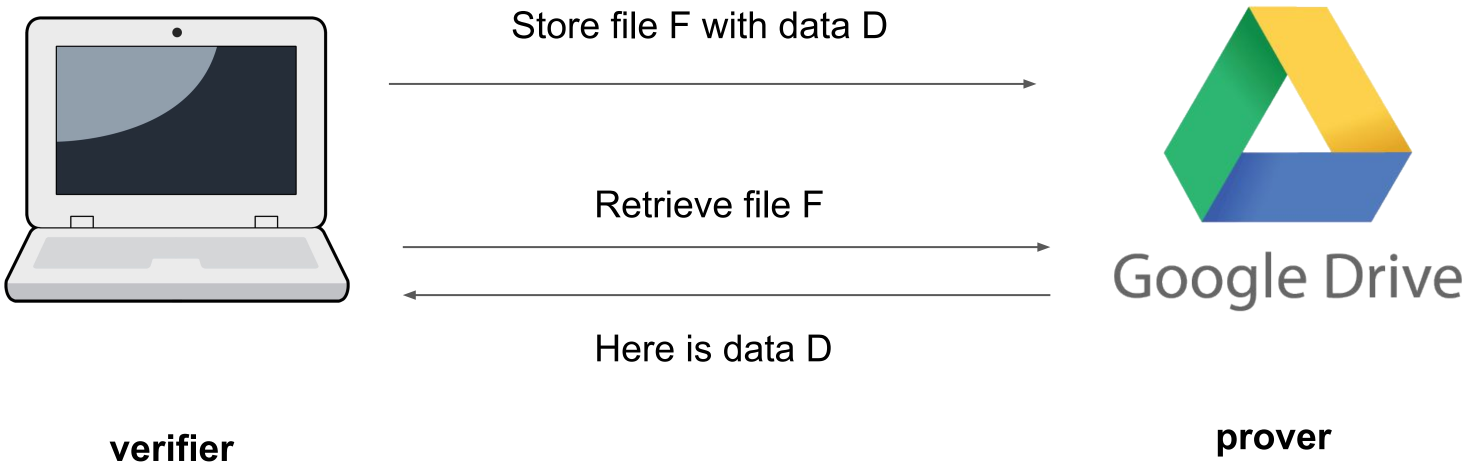
Overview

- 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 file storage problem



The file storage problem

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

File storage: Basic protocol

- Client sends file F with data 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

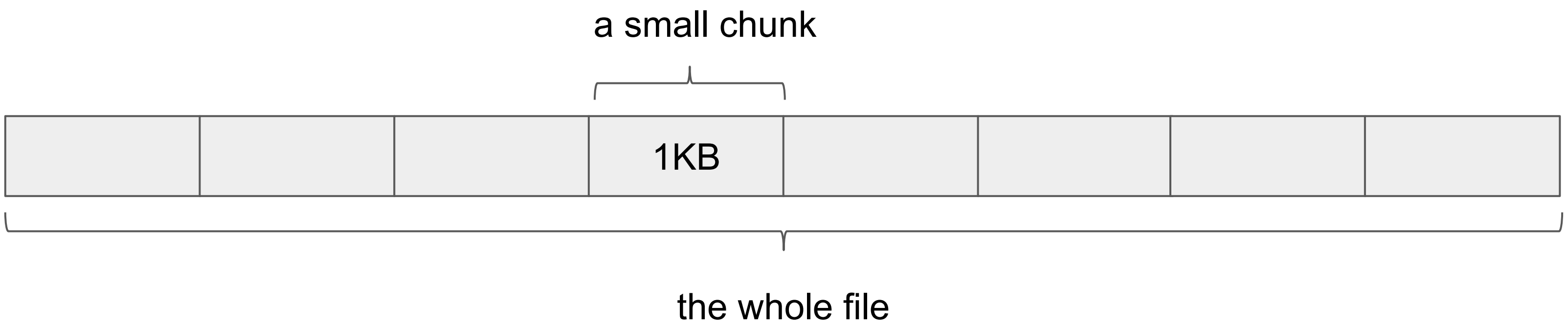
- Client sends file F with data D to server
- Server stores (F, D)
- Client stores H(D), deletes D
- Client requests F from server
- Server returns D’
- Client compares H(D’) = H(D)

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!

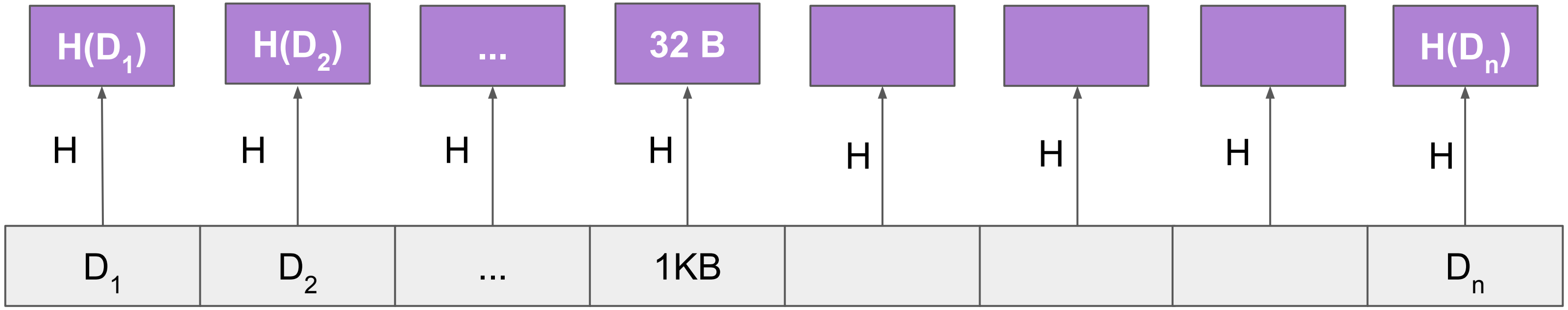
Merkle Tree

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



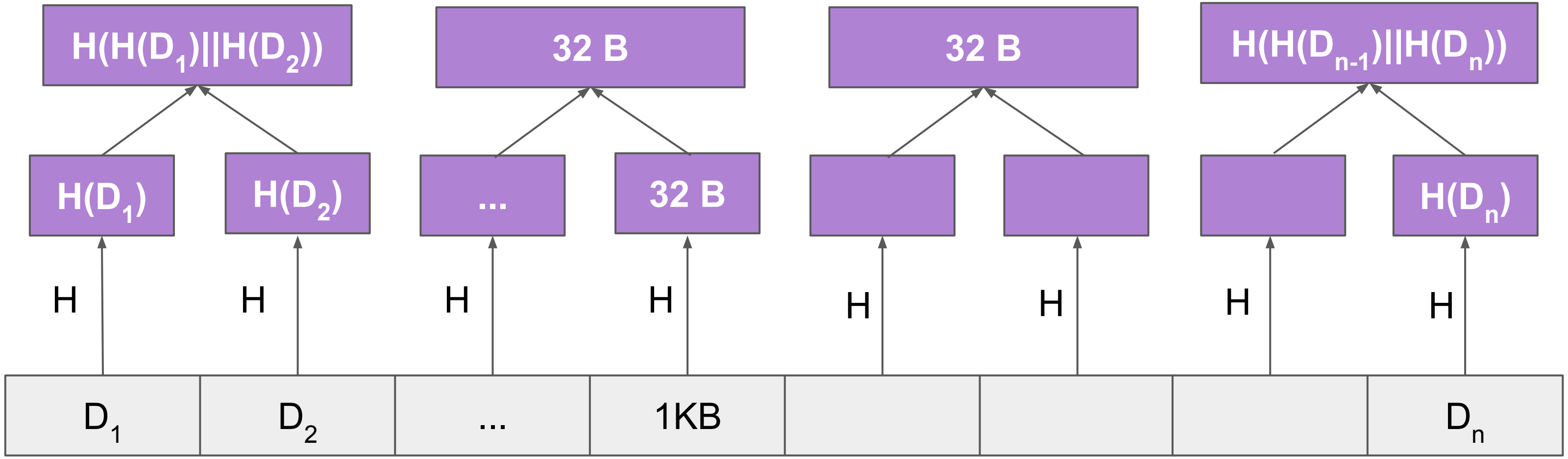
Merkle Tree

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

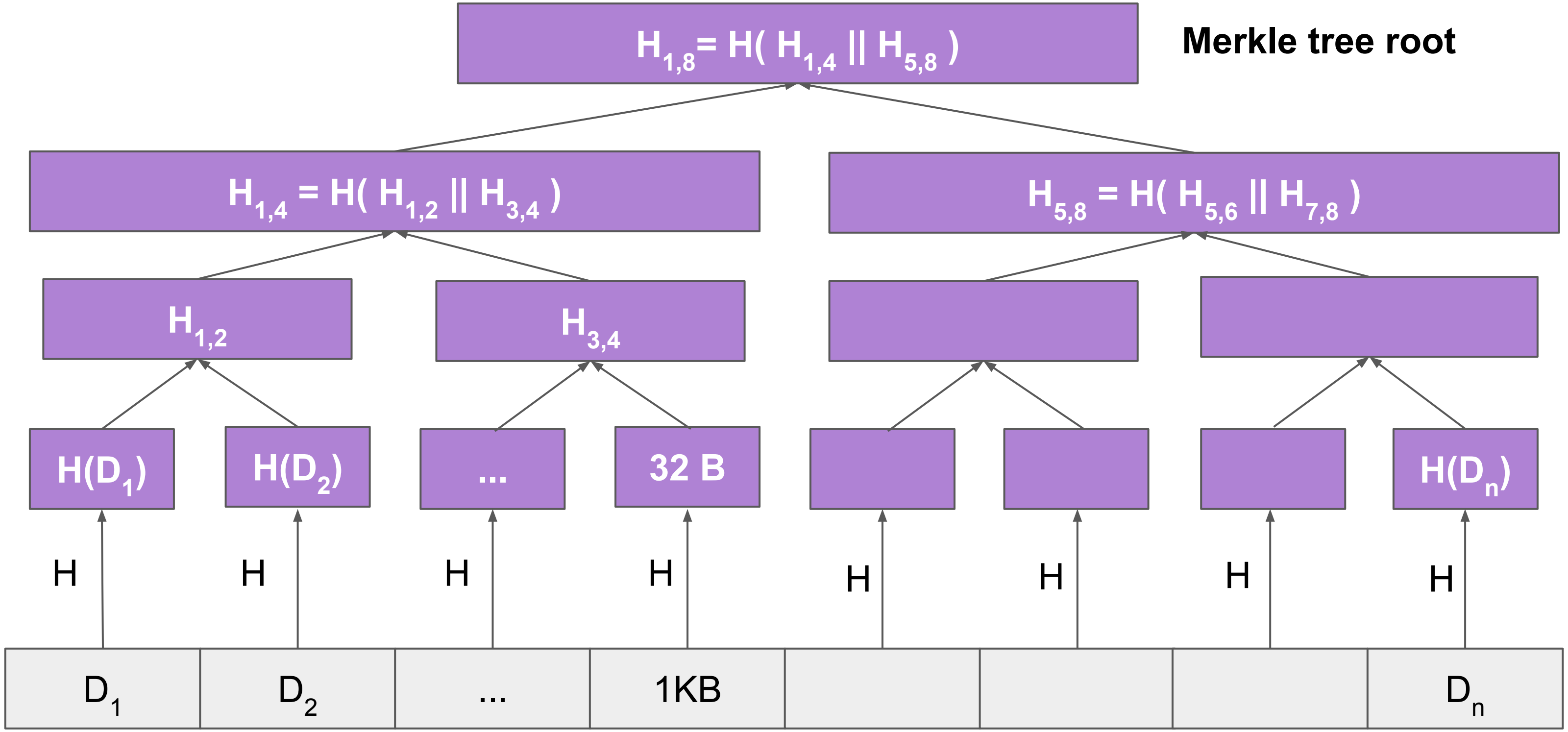


Merkle Tree

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



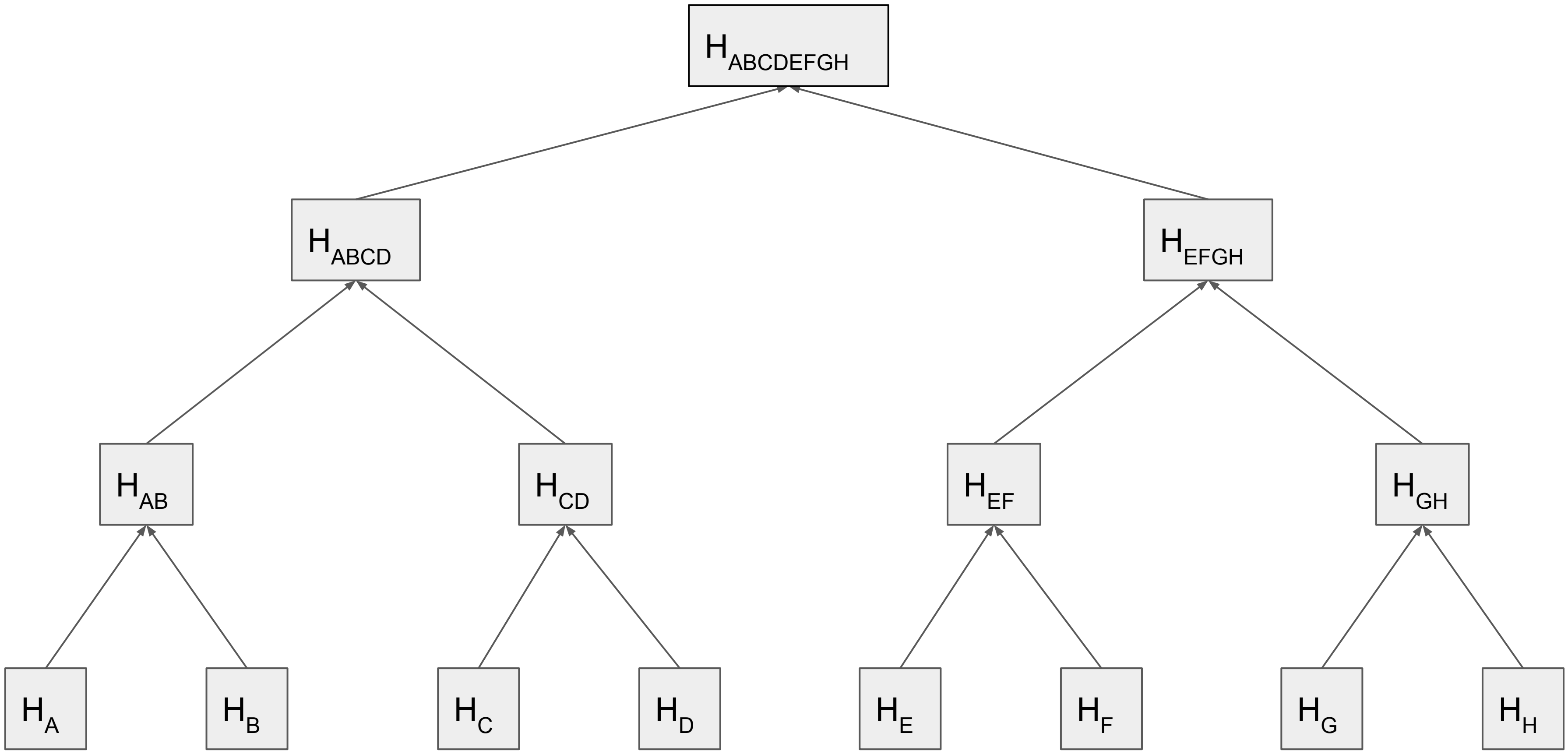
Merkle Tree


$H_{1,8}$= H($H_{1,4}$ || $H_{5,8}$)
Merkle tree root
$H_{1,4}$ = H($H_{1,2}$ || $H_{3,4}$)
$H_{5,8}$ = H($H_{5,6}$ || $H_{7,8}$)
$H_{1,2}$
$H_{3,4}$
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

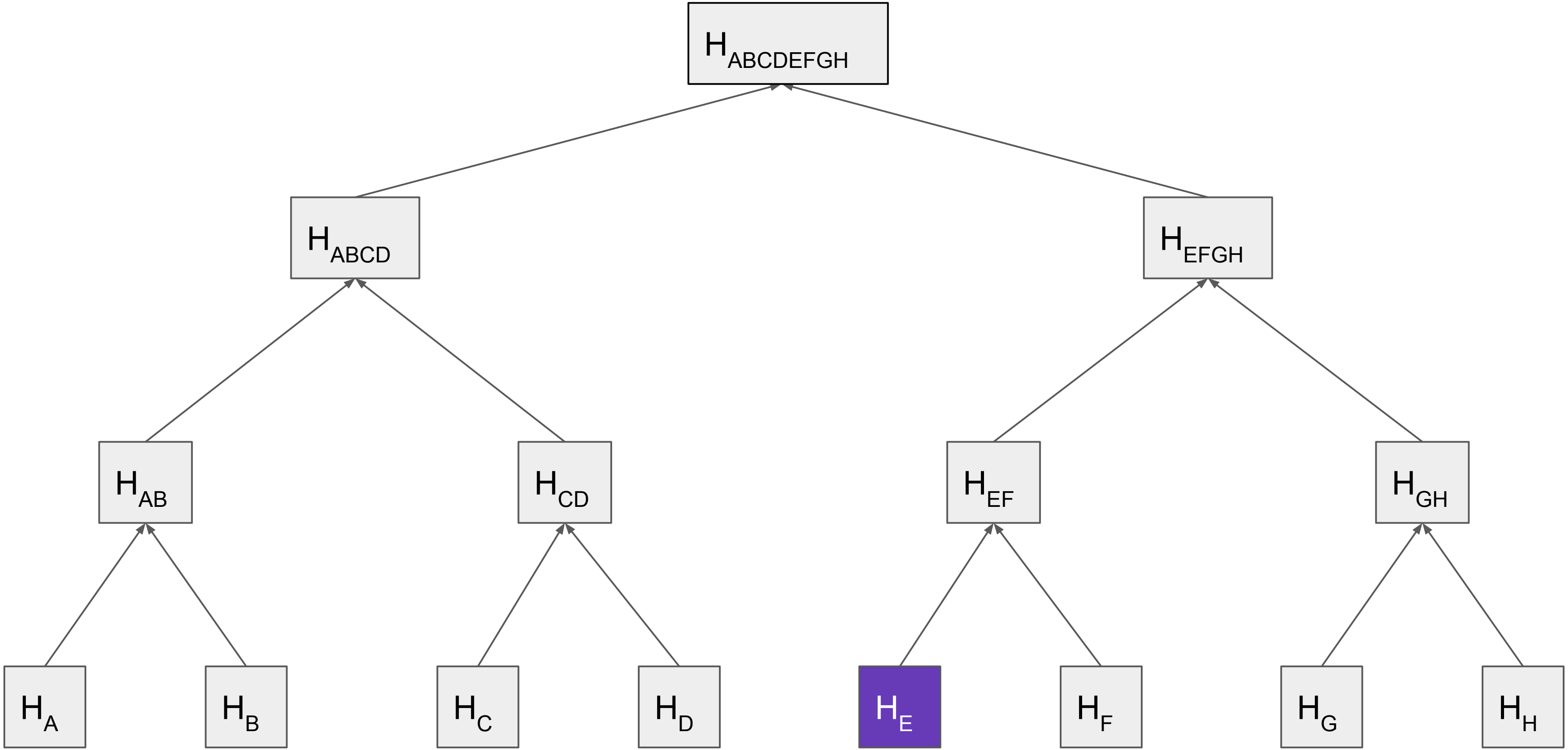
Proofs-of-inclusion

- 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

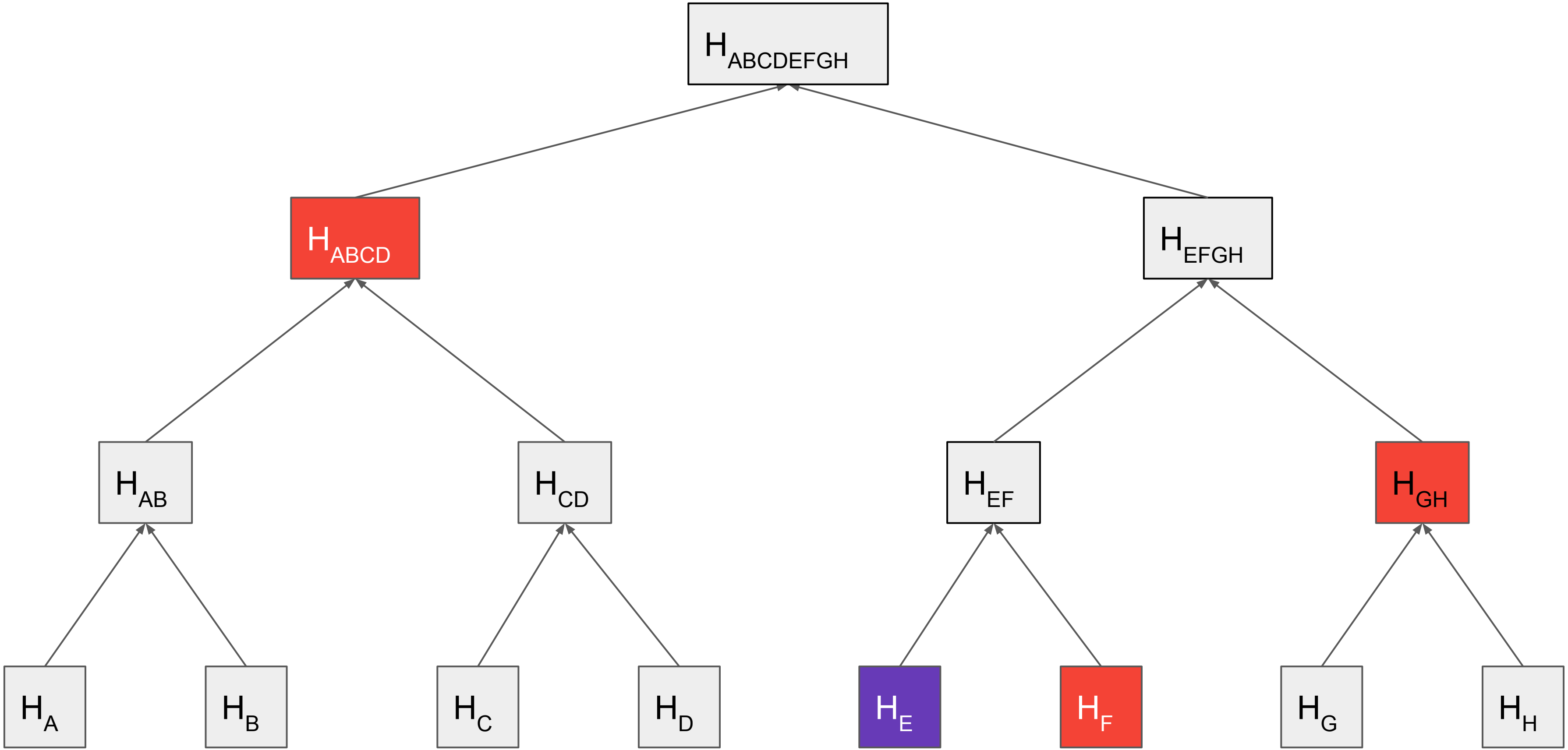
Merkle tree: proof of inclusion



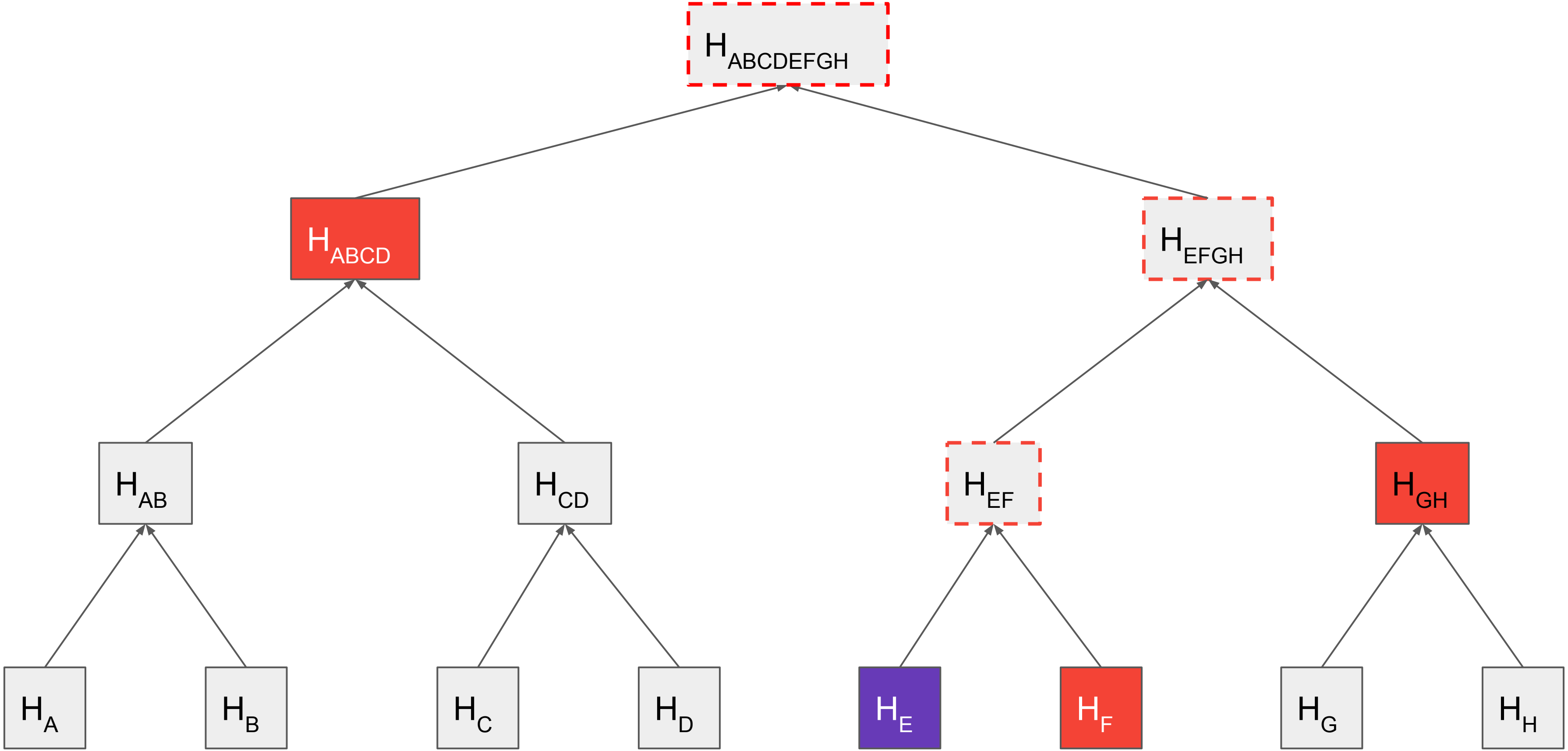
Merkle tree: proof of inclusion



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

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

Merkle Tree proof-of-inclusion security

- If adversary can present proof-of-inclusion for incorrect leaf, then we can break the hash function
- Proof is by computational reduction (whiteboard)

Merkle Tree protocol

`MT-construct(D)`

- Construct a Merkle Tree with given data D
- Returns the Merkle Tree root
- If |D| = chunk size, then: $\texttt{MT-construct(D) = H(D)}$
- Otherwise:
  $\texttt{MT-construct(D) = H(MT-construct(}D_1\texttt{) || MT-construct(}D_2\texttt{))}$
  where $D = D_1 \mathbin{||} D_2$

Merkle Tree protocol

`MT-prove(D, x)`

- Given data D and element x in D, construct proof-of-inclusion
- Returns the proof-of-inclusion π to be used with `MT-construct(D)`
- Proof contains:
  - Siblings on path connecting x to root
  - A bit for each sibling indicating whether the path we are taking is left or right

Merkle Tree protocol

`MT-verify(r, π, x)`

- Given Merkle Tree root r, element x, and proof-of-inclusion π
- Outputs true/false based on whether verification was successful

Correctness

For all D, x:

`MT-verify(MT-construct(D), MT-prove(D, x), x) = True`

(Proof by direct application of hashes on path)

Proof-of-inclusion security

- Assume the hash function is **collision-resistant**
- Recall collision resistance:
- $\forall$ PPT A: $\exists$ negl: $\Pr[\text{coll-find}_{A,H}(\lambda)] \leq \text{negl}(\lambda)$
- Where coll-find is the collision finding game:

```
def coll-find_{A,H}(λ):
    x_1, x_2 ← A(1^λ)
    if x_1 ≠ x_2 ∧ H(x_1) = H(x_2):
        return 1
    return 0
```

Threat modelling with bad events

- When defining a security property precisely, specify what **bad event** we are trying to avoid
- In this case, the construction of a proof about a non-existent element
- It is important to allow the adversary to **choose** which Merkle Tree to attack
- It is possible that the vast majority of trees are not attackable…
- Hence, we define a **game** where the adversary chooses a data set D to construct the tree from, an element x, and a proof of π
- The adversary can construct these arbitrarily. π does not need to be produced out of a tree!

The Merkle-Tree forgery game

```
def MT-forgery_{A,Π(H)}(λ):
    (D, x, π) ← A(1^λ)
    if MT-verify(MT-construct(D), π, x) ∧ x ∉ D:
        return 1
    return 0
```

The Merkle Tree security

$\forall$ PPT A: $\exists$ negl: $\Pr[\text{MT-forgery}_{A,\Pi(H)}(\lambda)] \leq \text{negl}(\lambda)$

The theorem: Assumption → Desirable

Theorem: If H is collision-resistant, then the MT constructed from H is secure:

$$\forall \text{ PPT } A: \exists \text{ negl}: \Pr[\text{coll-find}_{A,H}(\lambda)] \leq \text{negl}(\lambda)$$

$$\rightarrow$$

$$\forall \text{ PPT } A: \exists \text{ negl}: \Pr[\text{MT-forgery}_{A,\Pi(H)}(\lambda)] \leq \text{negl}(\lambda)$$

Proof strategy: Contraposition

By reductio ad absurdum using contraposition:

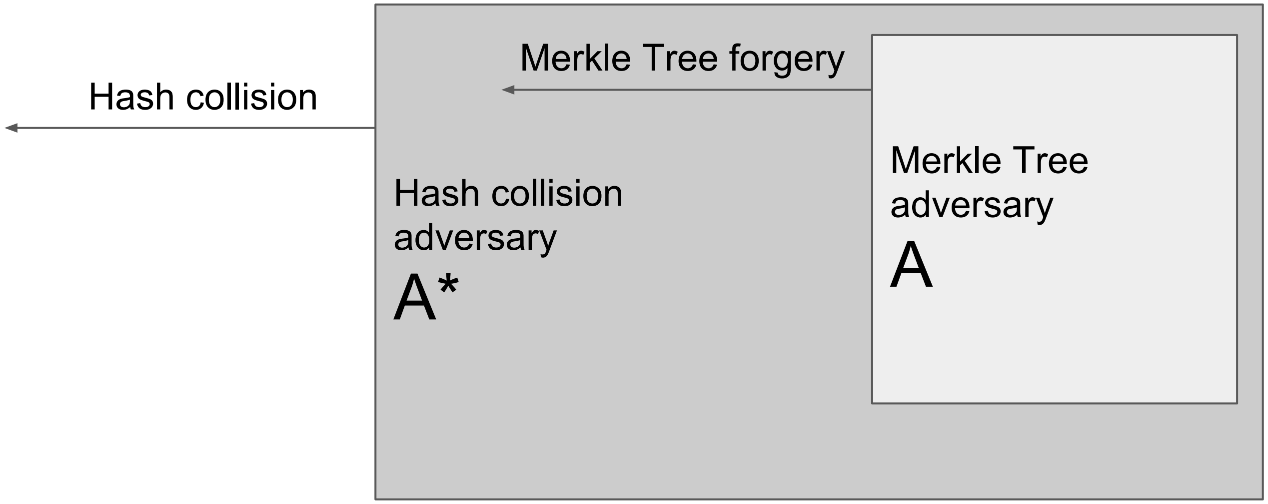
- Suppose for contradiction that
  not $\forall$ `PPT A:` $\exists$ `negl: Pr[MT-forgery`$_{A,\Pi(H)}$`(λ)] ≤ negl(λ)`
  i.e., the Merkle Tree construction is **not** secure
- It suffices to show that
  not $\forall$ `PPT A:` $\exists$ `negl: Pr[coll-find`$_{A,H}$`(λ)] ≤ negl(λ)`
  i.e., the hash function is **not** collision-resistant

Proof strategy: Contraposition

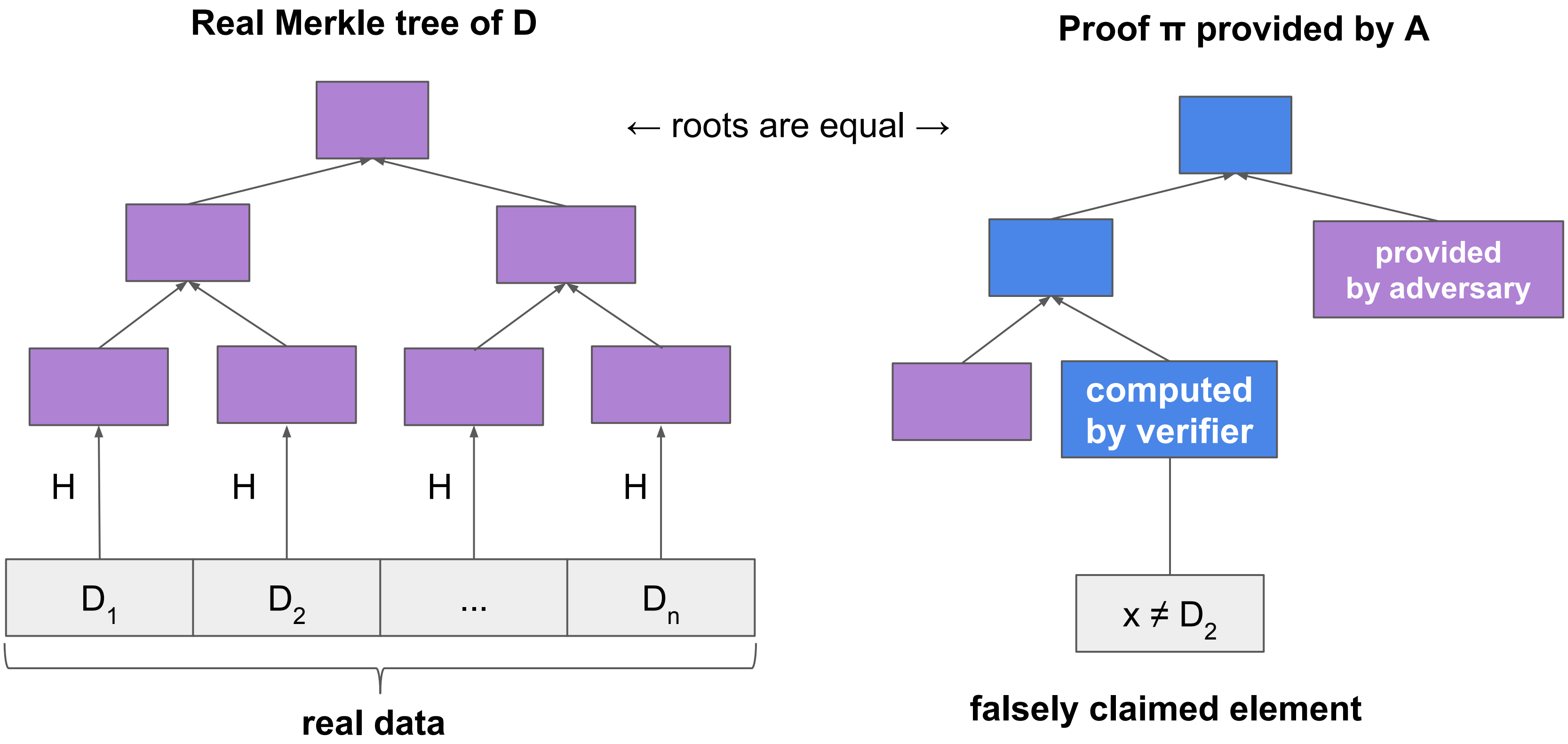
- Suppose for contradiction that
  $\exists$ PPT A: $\Pr[\text{MT-forgery}_{A,\Pi(H)}(\lambda)]$ is non-negl
- It suffices to show that
  $\exists$ PPT A*: $\Pr[\text{coll-find}_{A,H}(\lambda)]$ is non-negl

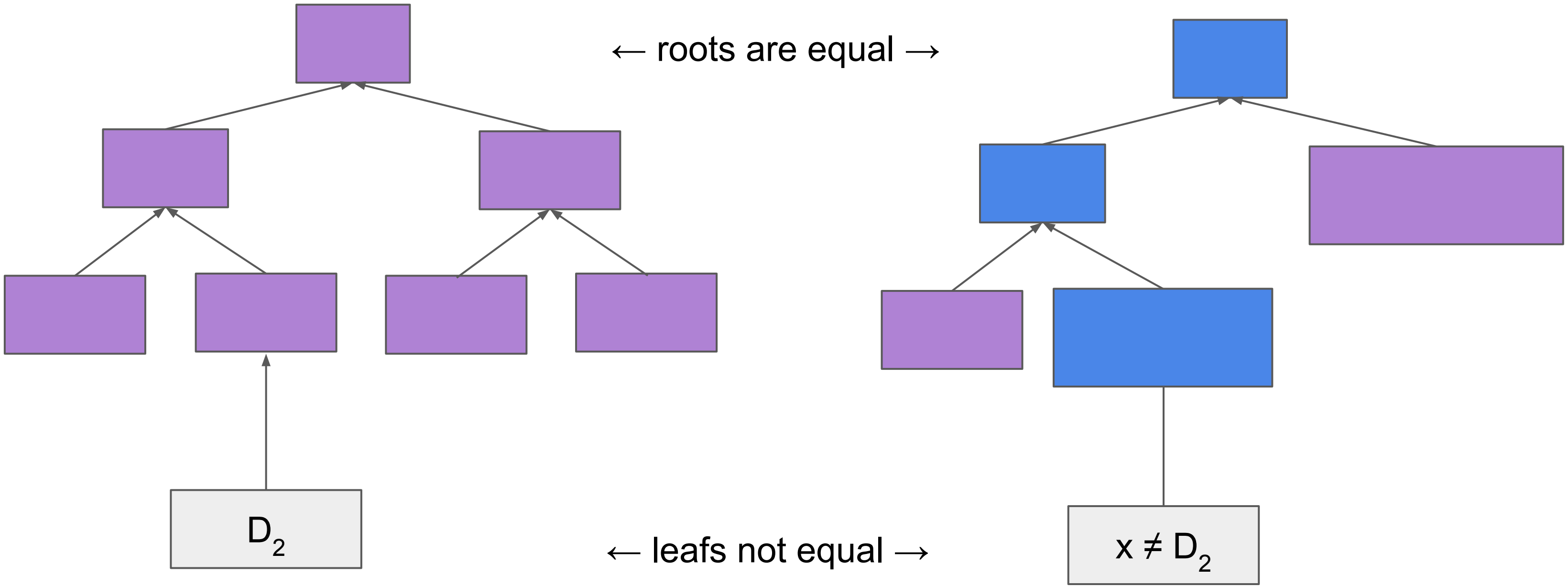
The PPT A is arbitrary, so we must use it as black box.
We show the existence of A* by construction.
Since A is a machine, we can have A* call A in its code.

Proof strategy: Computational reduction

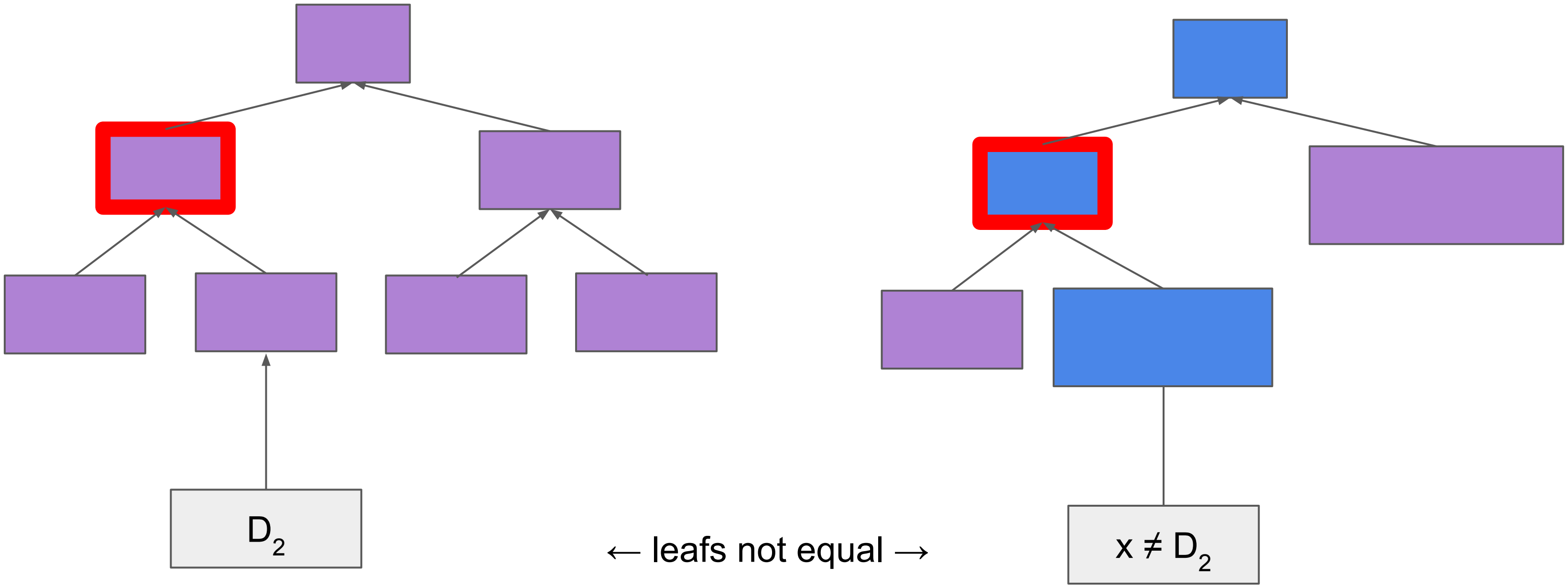


Situation if adversary A wins



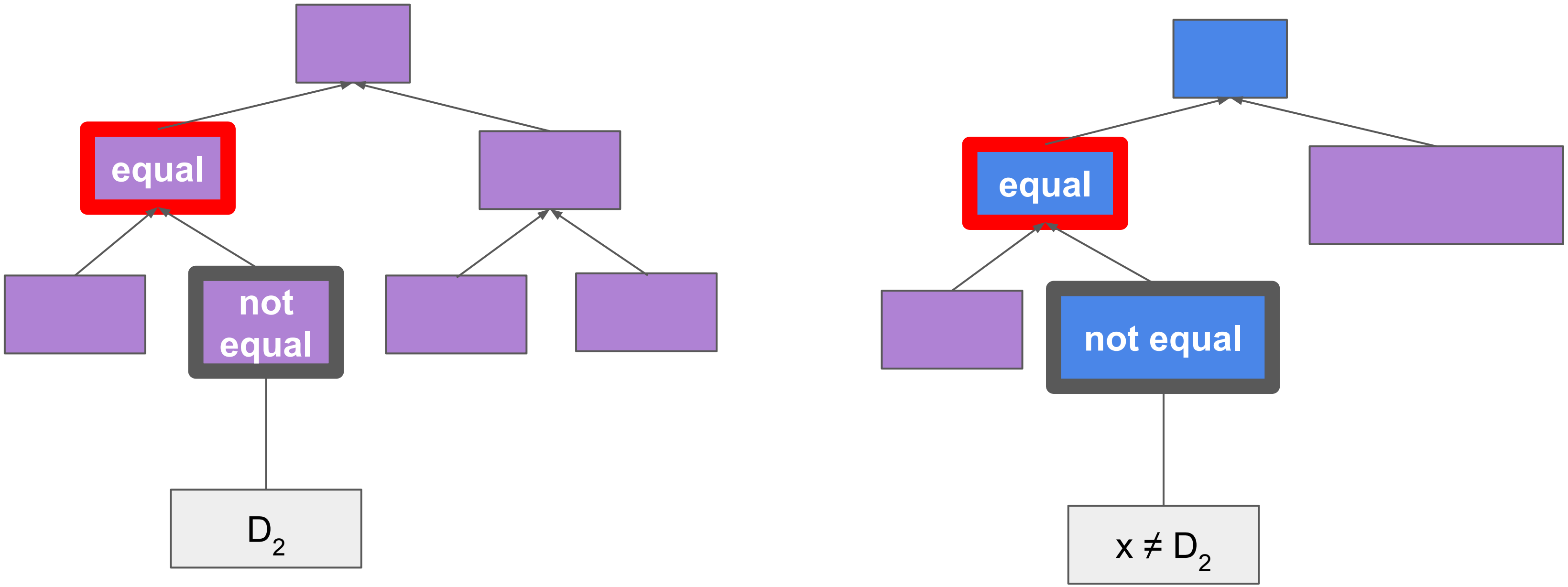

← roots are equal →
D2
← leafs not equal →
x ≠ D2

Induction:
Take **lowest** tree level where nodes are **equal**
That level must exist, as roots are equal
That level cannot be a leaf, as leafs are not equal



$H(L^a \mid\mid R^a) = H(L^b \mid\mid R^b)$ but $R^a \neq R^b$

We can extract a hash collision!



Proof conclusion

A* works as follows:

- Checks if A has found forgery
- If not, aborts
- If yes, finds minimum level where hashes are equal
- This gives a hash collision

If A finds a MT forgery, then A* finds a hash collision

$$\underbrace{\mathtt{Pr[MT\text{-}forgery}_{A,\Pi(H)}(\lambda)]}_{\text{non-negligible by contradiction assumption}} \overbrace{=}^{\text{equal by computational reduction}} \underbrace{\mathtt{Pr[coll\text{-}find}_{A^*,H}(\lambda)]}_{\text{non-negligible, therefore contradiction}}$$

Proof of security is nuanced

Proof works only if Merkle Tree size is fixed
(i.e., elements of D are some constant).
Otherwise construction is not secure!
Extreme care is needed when inventing or even slightly modifying such structures!

Don't roll your own crypto.
Use standard code by others which you know is secure.

Merkle tree applications

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

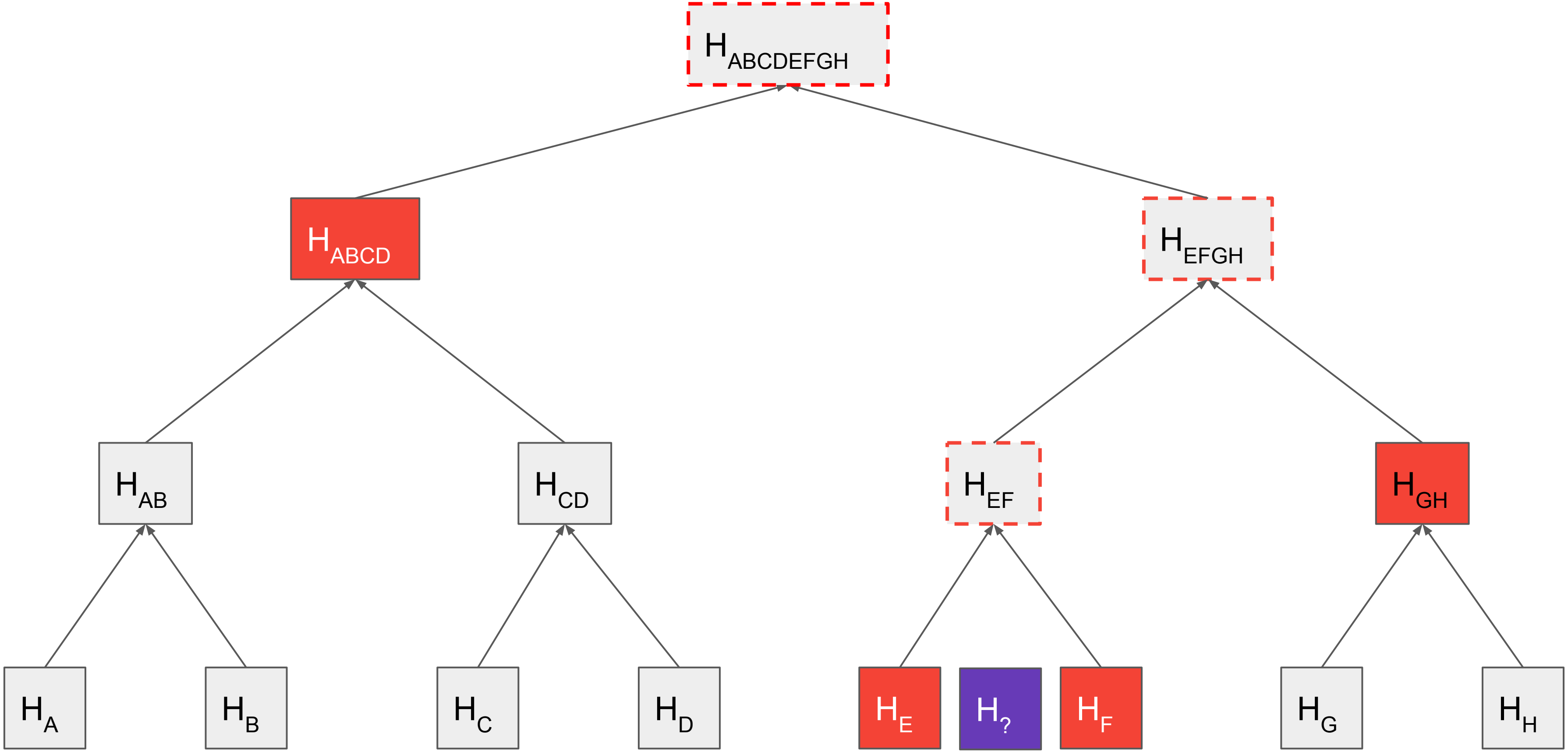
Storing *sets* instead of lists

- 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



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

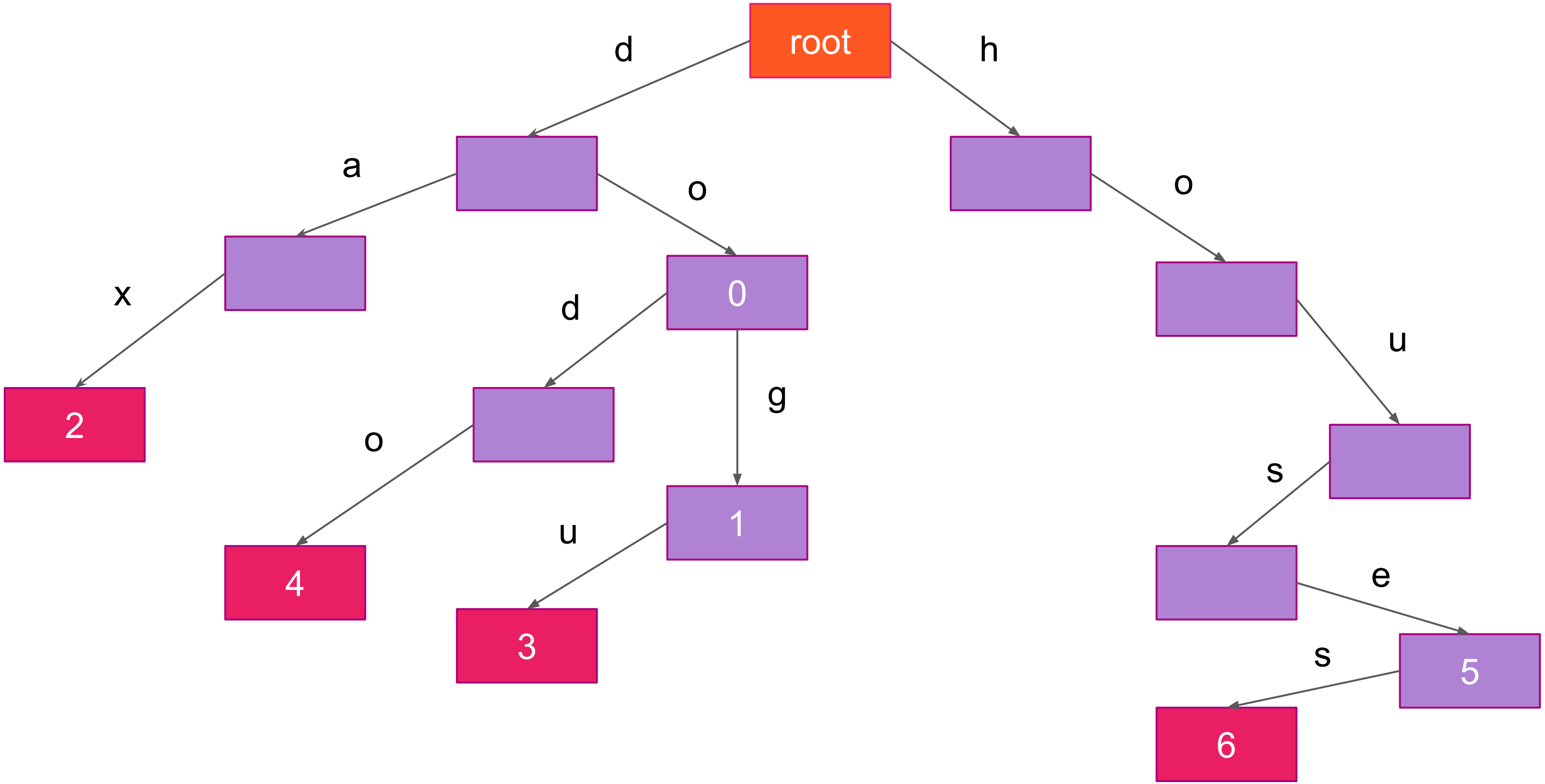
Tries: query(string)

- 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



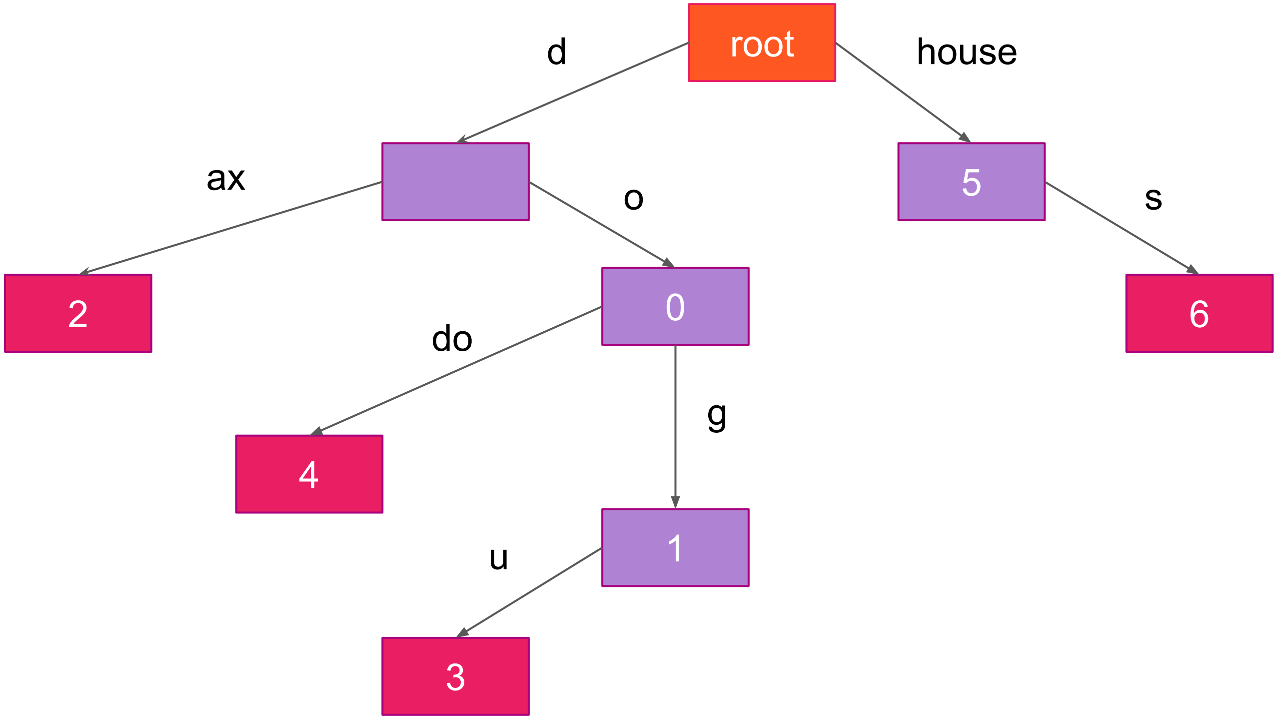
Patricia (or radix) trie

- 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 does not need to be yes/no
- Can contain arbitrary value
- This allows us to map keys to values
- **add(key, value)**
- **query(key) → value**

Patricia trie



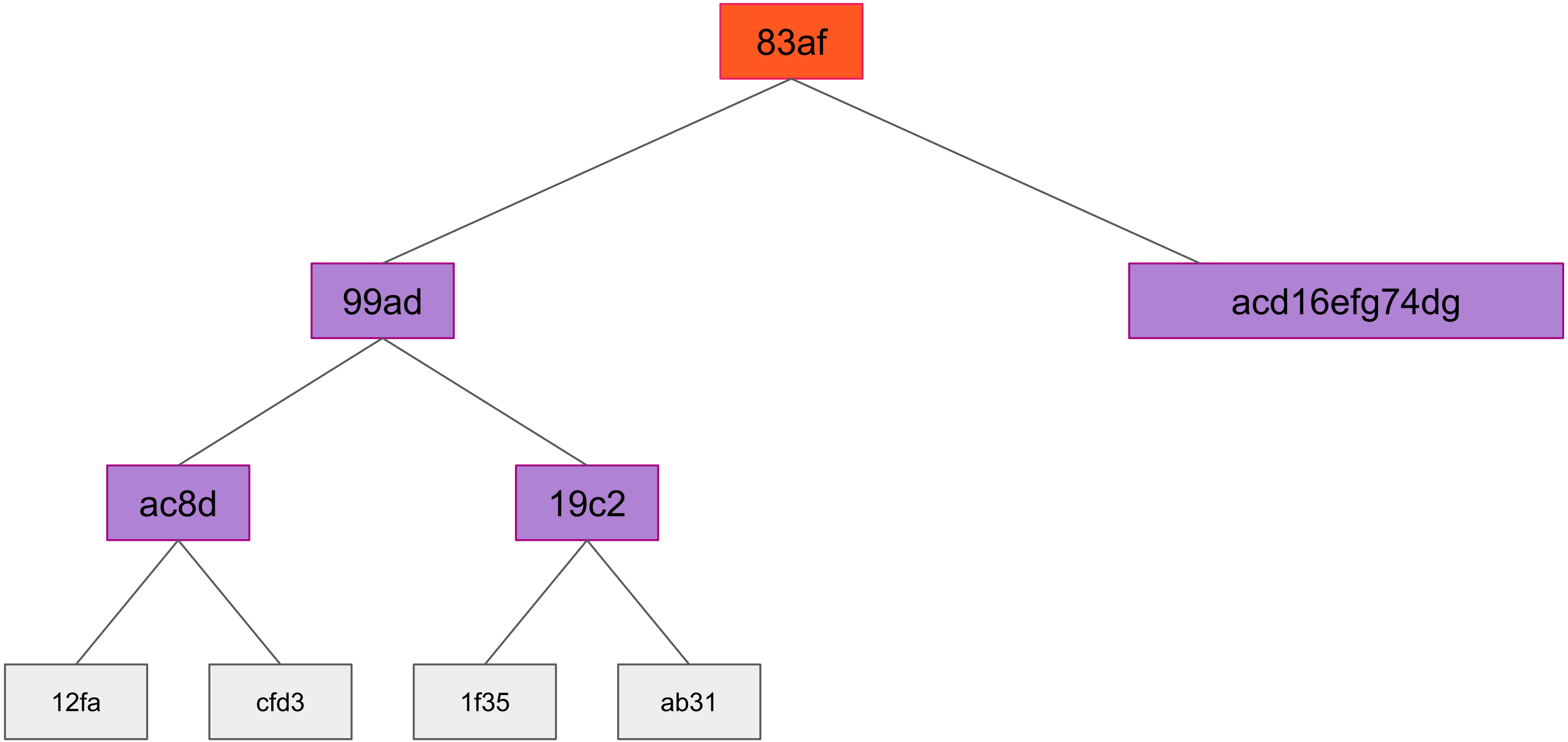
Merkle Patricia trie

- 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

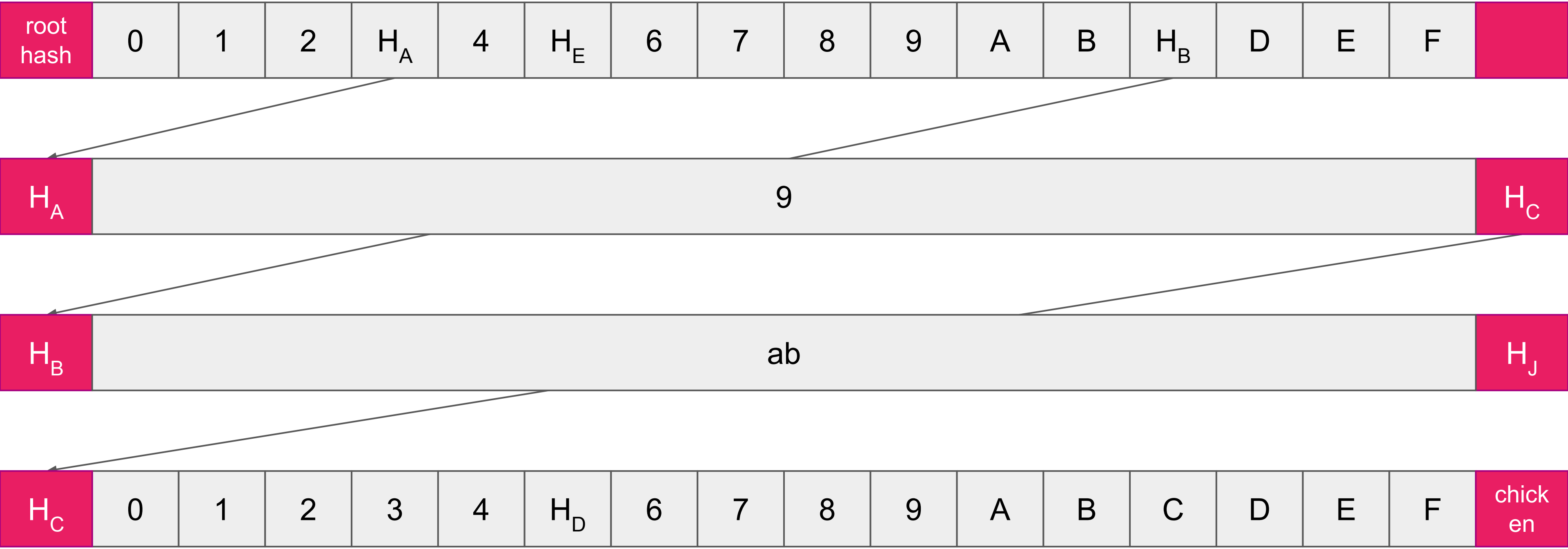


Merkle patricia trie: node

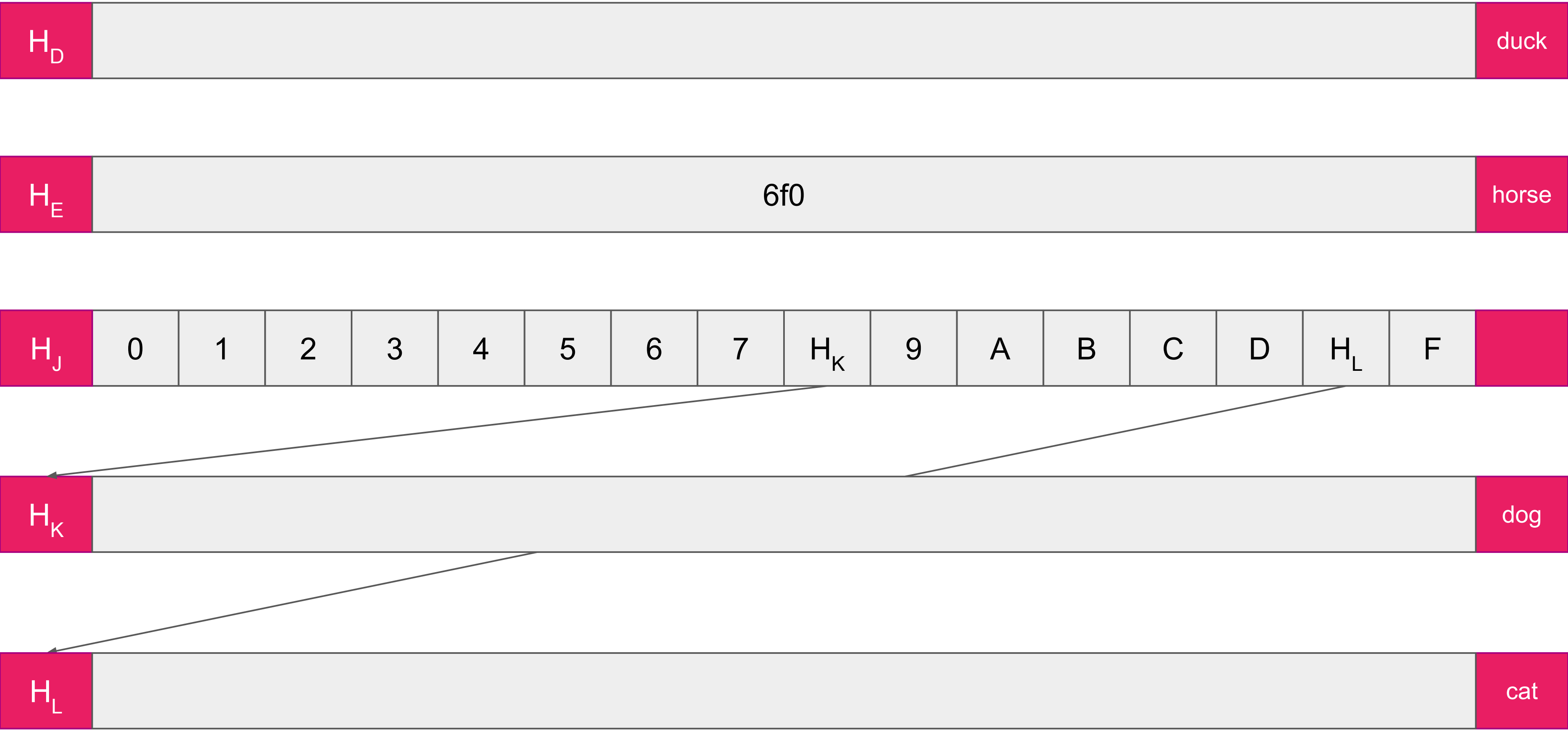
key	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	value

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

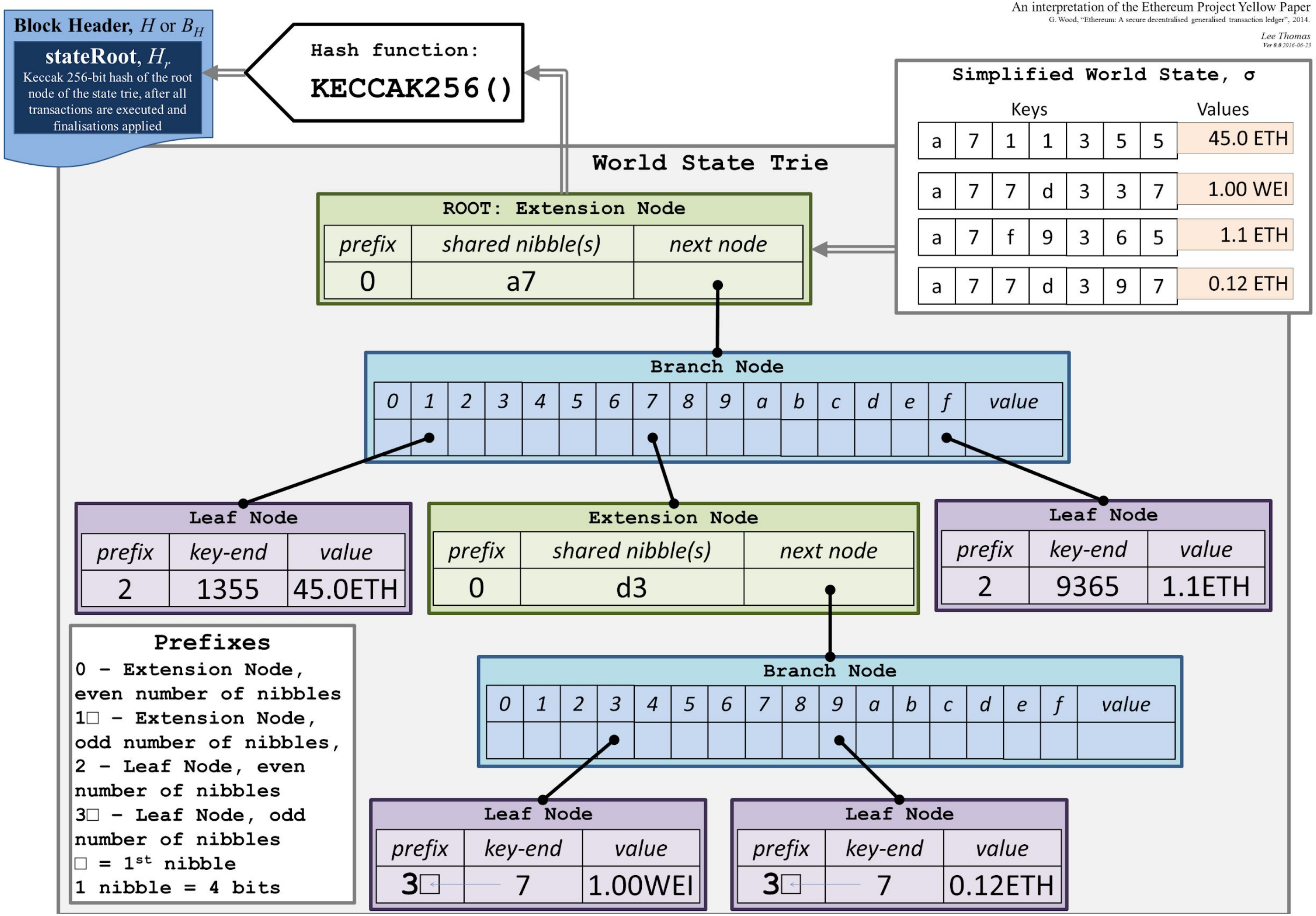
Ethereum Modified Merkle-Paricia-Trie System

An interpretation of the Ethereum Project Yellow Paper

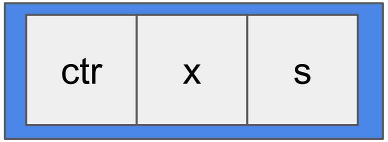
G. Wood, “Ethereum: A secure decentralised generalised transaction ledger”, 2014.

Lee Thomas

Ver 0.0 2016-06-23



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

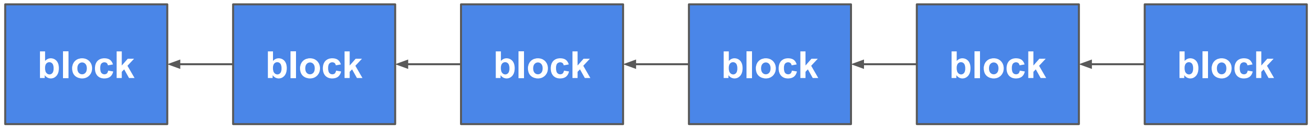
- Blocks must satisfy proof-of-work equation

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

- 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

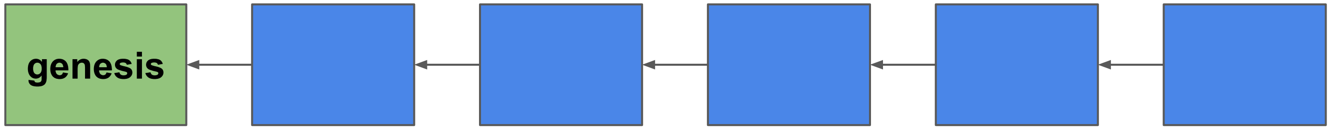
- Each block references a **previous** block
- This reference is by **hash** to its **previous** block, similar to Merkle Trees
- This linked list is called the **blockchain**
- Convention: Arrows show authenticated inclusion



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

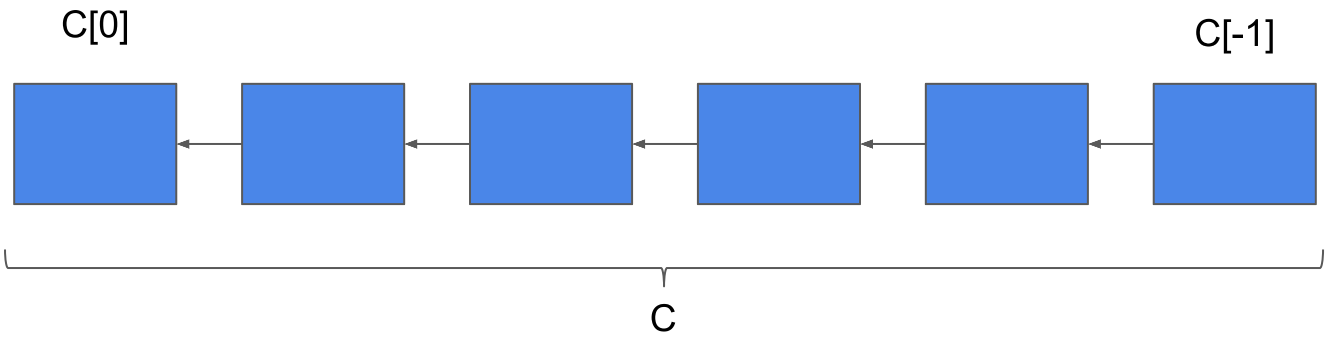
Blockchain

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



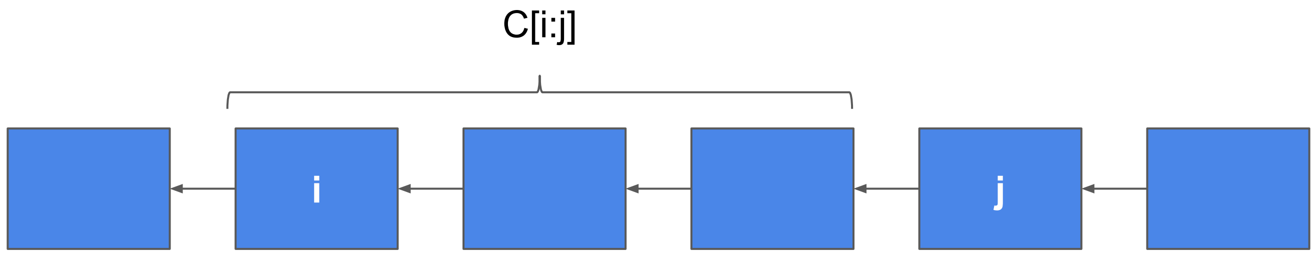
Notation conventions

- We use the symbol **C** to denote a blockchain
- C is a **sequence** of blocks
- We use C[i] to denote the i^{th} block (0-based)
- C[0] denotes genesis
- We use C[-i] to denote the i^{th} block from the end
- Chain property: For each i > 0: C[i].s = H(C[i - 1])



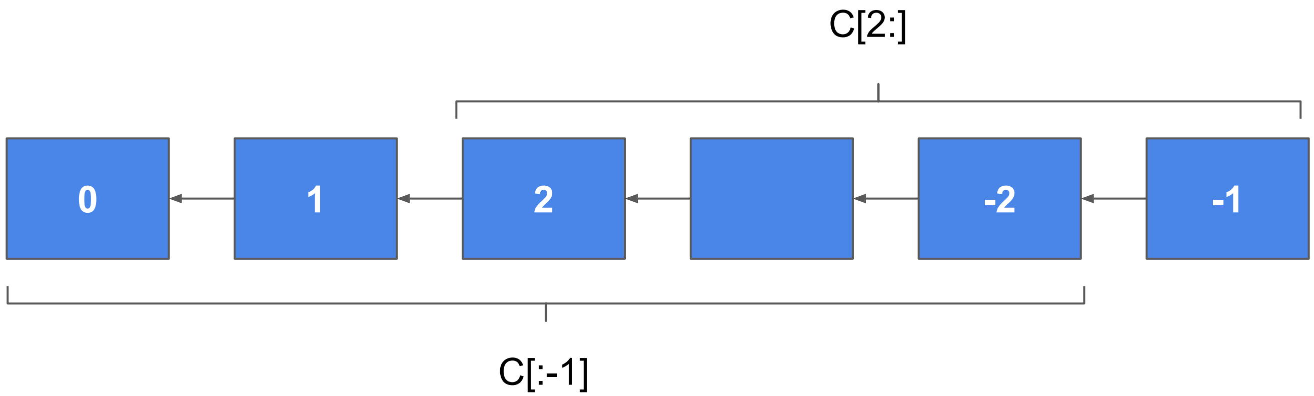
Notation conventions

- Range notation: C[i:j] denotes a **subsequence** from i (inclusive) to j (exclusive)
- Similarly C[-i:j], C[i:-j], C[-i:-j]



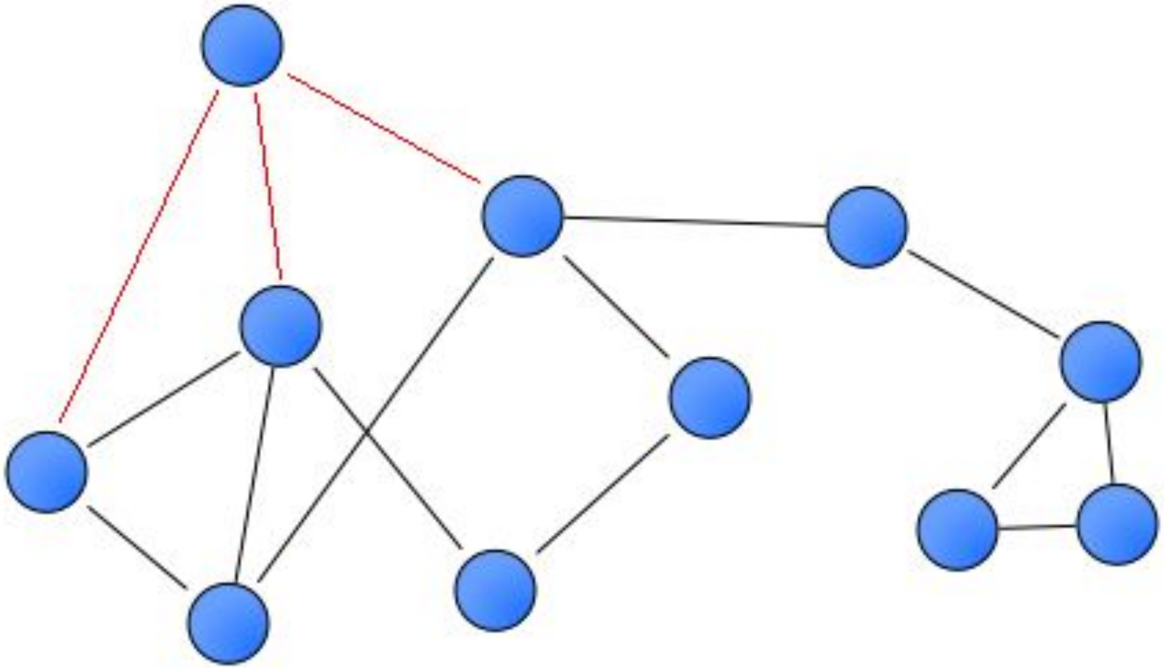
Notation conventions

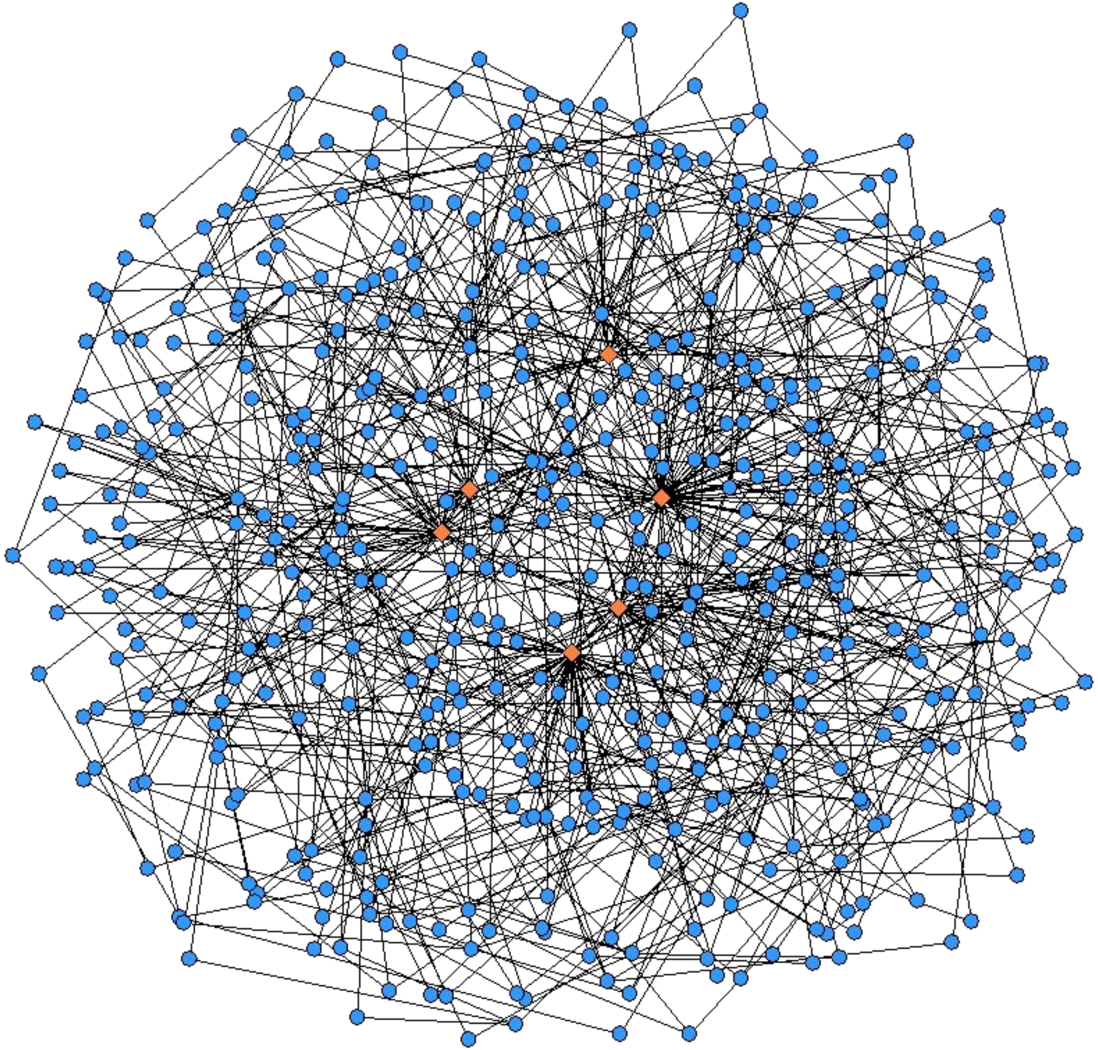
- Range notation:
  C[i:], C[-i:] denotes chain from i (or -i) inclusive to end inclusive
  C[:i], C[-i:] denotes chain from beginning inclusive to i (or -i) exclusive

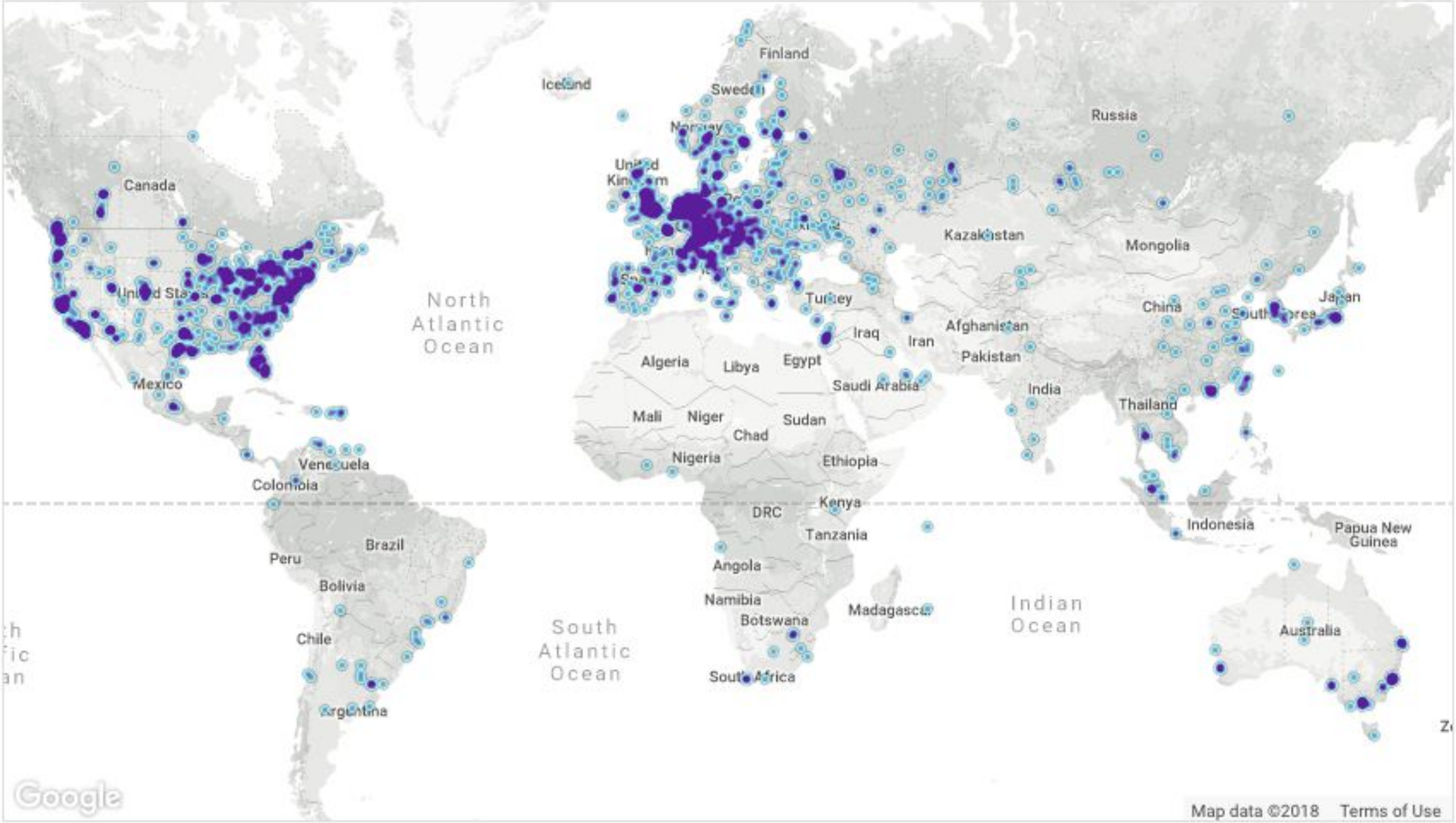


The bitcoin network

- 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
Iceland
Russia
Canada
Kazakhstan
Mongolia
North Atlantic Ocean
Turkey
China
Japan
Afghanistan
Iraq
Iran
Algeria
Libya
Egypt
Pakistan
Mexico
Saudi Arabia
India
Thailand
Mali
Niger
Sudan
Chad
Nigeria
Ethiopia
Colombia
Kenya
DRC
Indonesia
Tanzania
Papua New Guinea
Brazil
Peru
Angola
Bolivia
Namibia
Indian Ocean
South Atlantic Ocean
Botswana
Chile
Australia
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

- Financial data is encoded in the form of *transactions*
- Every transaction is broadcast on the network to everyone using the gossip protocol
- Financial data on cryptocurrencies are **common knowledge** among all participants

Transactions on blockchain.info

Eclipse attacks

- Cut the network in some honest nodes into two 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
- **We cannot hope to rectify this situation**
- 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