BITS Pilani
Pilani Campus

Blockchain Technology
(BITS F452)

Dr. Ashutosh Bhatia, Dr. Kamlesh Tiwari
Department of Computer Science and Information Systems

innovate achieve lead

BITS Pilani
Pilani Campus

Introduction to Crypto and Cryptocurrency

LECTURE OUTLINE

- Crypto Background
  - Hash Functions
  - Digital Signatures and its Applications
- Introduction to cryptocurrency
  - Basic digital cash

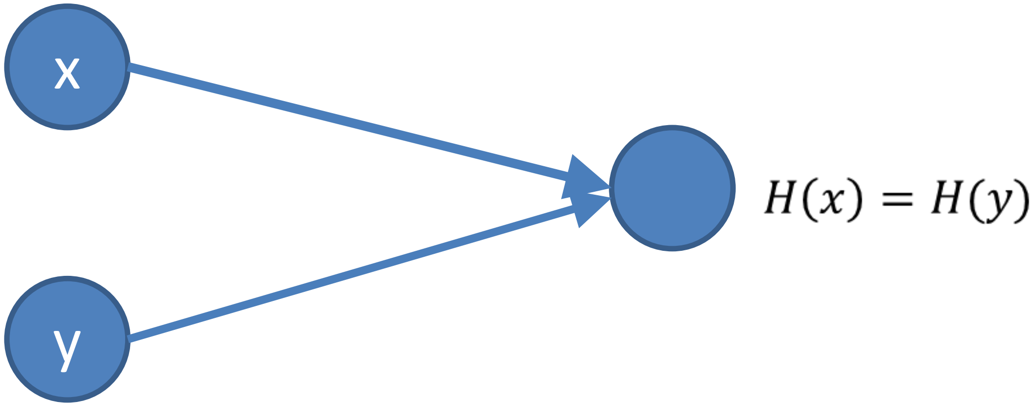
Hash Functions

- Takes arbitrarily length of string as input
- Produces a fixed sized output
- Efficiently Computable

- Security Properties
  - Collision Free
  - Hiding
  - Puzzle friendly

Hash Properties 1: Collision Resistant

- A collision occurs when two distinct inputs produce the same output
- **Collision-resistance**: A hash function H is said to be collision resistant if it is infeasible to find two values, x and y, such that $x \neq y$ and $H(x) = H(y)$



- However, collision do exist

How to find a collision?

- Try 2^{130} randomly chosen inputs and assuming that hash output is 256 bits, 99.8% chance that two of them will collide

- This works, no matter what the hash function is. (**Birthday Paradox**)

- However, 2^{130} is a so large number and any computer ever made by the humanity was trying to find a collision since the beginning of the universe till now, the probability of it finding a collision is infinitesimally small.

How to find a collision?

- Try 2^{130} randomly chosen inputs and assuming that hash output is 256 bits, 99.8% chance that two of them will collide

- This works, no matter what the hash function is. (**Birthday Paradox**)

- However, 2^{130} is a so large number and any computer ever made by the humanity was trying to find a collision since the beginning of the universe till now, the probability of it finding a collision is infinitesimally small.

Birthday Paradox

Find the probability that at-least two people in a room have the same birthday

Event A: at least two people in the room have the same birthday

Event A' : No people in the room have the same birthday

$$\Pr[A] = 1 - \Pr[A']$$

$$\Pr[A'] = 1 \times \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \left(1 - \frac{3}{365}\right) \cdots\cdots \left(1 - \frac{Q-1}{365}\right)$$

$$= \prod_{i=1}^{Q-1} \left(1 - \frac{i}{365}\right)$$

$$\Pr[A] = 1 - \prod_{i=1}^{Q-1} \left(1 - \frac{i}{365}\right)$$

$$\sqrt{Q \approx 2M \ln \frac{1}{1-\epsilon}}$$

M = 365, ϵ is the desired

if $\epsilon = .5$ *then* $Q \approx 1.17\sqrt{M}$

Thus to achieve 128 bit security against collision attacks, hashes of length at-least 256 is required

Is there a better way?

- For some possible Hash functions, YES
  - Example $H(x) = x \bmod 2^{256}$
- For others we don't know one
- No Hash Function is <u>proven</u> to be collision resistant

Application: Hash as a message digest

- If we know that H(x) = H(y)

  it is safe to assume that x = y

- To recognize a file that we saw before

  just remember its hash

- Useful as the hash is small

Hash Property 2: Hiding

- We want something like this

  given H(x) it is infeasible to find x.

- The problem is that this property can not be true in the stated form if the number of possible input values is small

- **Hiding:** A hash function H is hiding if: when a secret value r is chosen from a probability distribution that has **high min-entropy**, then given H(r | x) it is infeasible to find x.

- High min-entropy means that the distribution is very spread out and no particular value is chosen with negligible entropy.

Application: Commitment

We want to ”seal a value” in the envelop and ”open the envelop” later

Commit to a value and reveal it later

Commitment API

(com, key) := commit(msg)

match := verify(com, key, msg)

To seal msg in envelop

(com, key) := commit(msg), then publish com

To open envelop

publish key, msg

Anyone can use verify() to check the message

Commitment API

(com, key) := commit(msg)

match := verify(com, key, msg)

Security Properties

Hiding: Given com, infeasible to find msg

Binding: Infesible to find msg != msg’ s.t.

verify(commit(msg), msg’) = true

Commitment API

commit(msg) := (H(key | msg), key))

`where key is a random 256 bit value

verify(com, key, msg) = (H(key | msg) == com)

Security Properties

Hiding: Given H(key | msg), infeasible to find msg

Binding: Infeasible to find msg != msg' s.t.

H(key | msg) == H(key | msg')

Hash Property 3: Puzzle friendly

For every possible out put value y,

if k is chosen randomly from a distribution with high min entropy,

then it is infeasible to find x such that H(k | x) = y

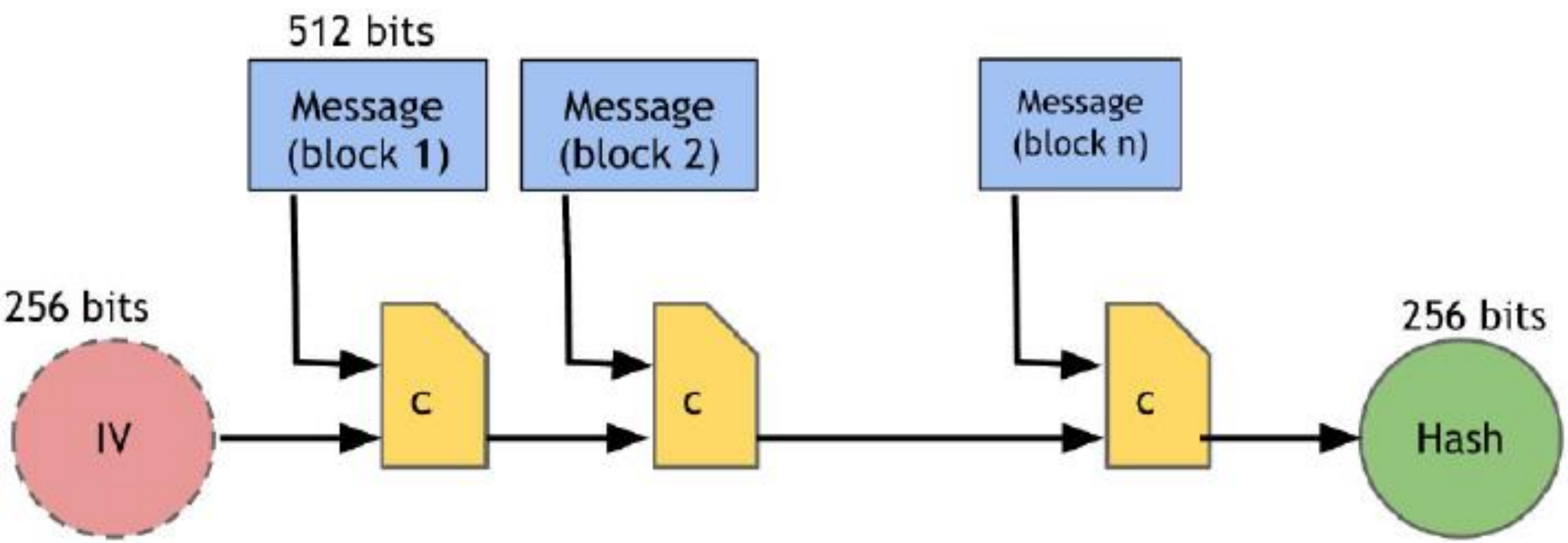
Application: Search Puzzle

Given a puzzle ID, id (from high min-entropy dist.) and a target set Y

Try to find a solution x, such that $H(id \mid x) \in Y)$

Puzzle friendly property implies that no solving strategy is much better than trying random values of x.

SHA 256 hash function



Theorem: If c (the compression function) is collision-free than SHA-256 is collision free

[Blockchain Demo (andersbrownworth.com)](https://andersbrownworth.com)

Hash Pointer

- Hash Pointer is :
  - pointer to where some information is stored
  - cryptograhic hash of the information

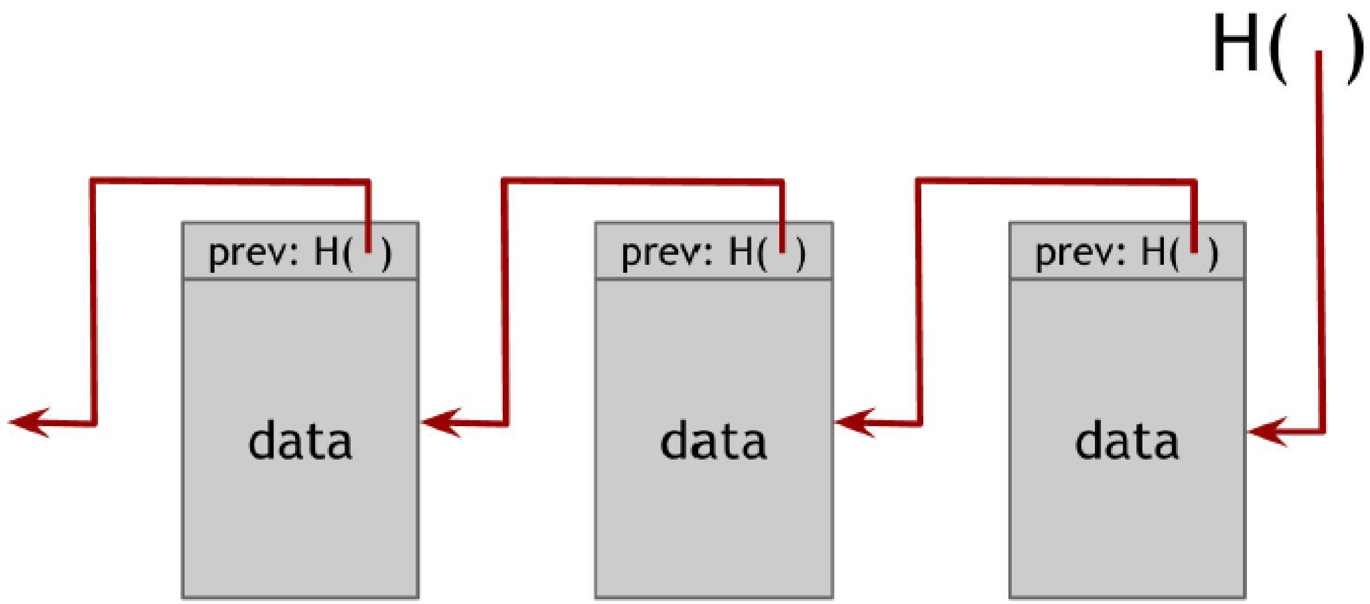
- If we have a hash pointer, we can
  - ask to get the info back
  - verify that it has not changed

innovate achieve lead

Key Idea

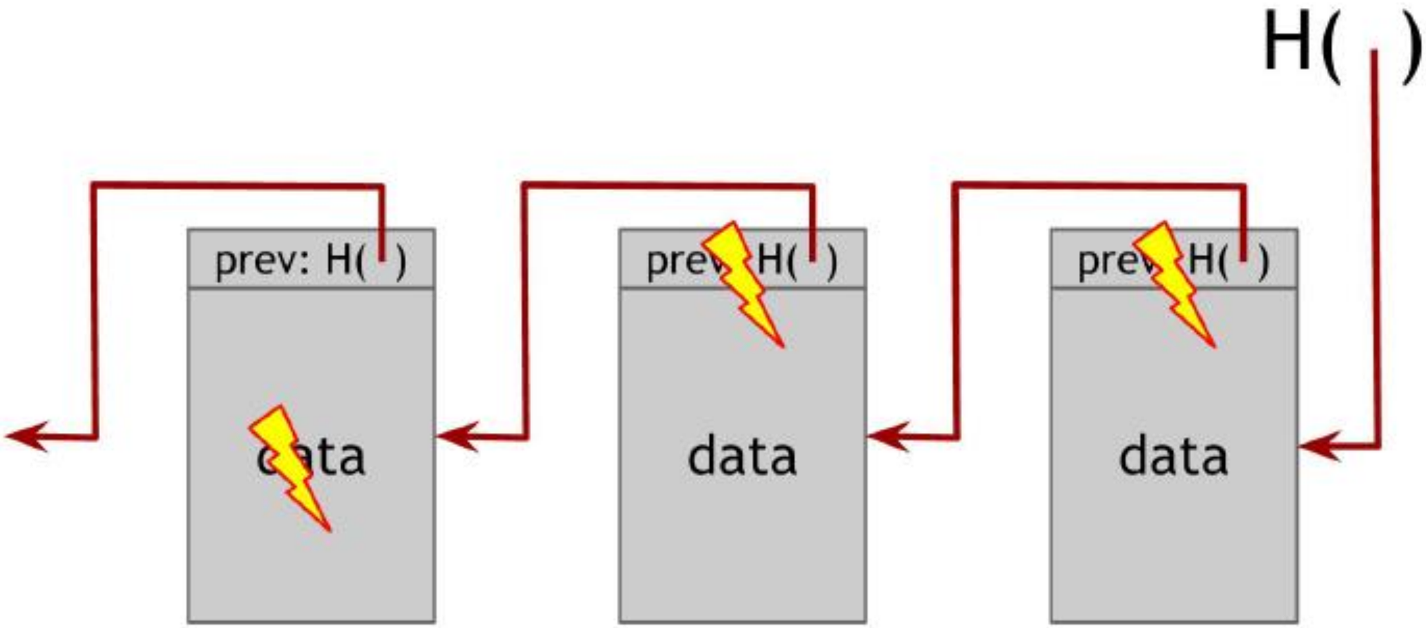
Build Data Structures with Hash Pointers

Linked List

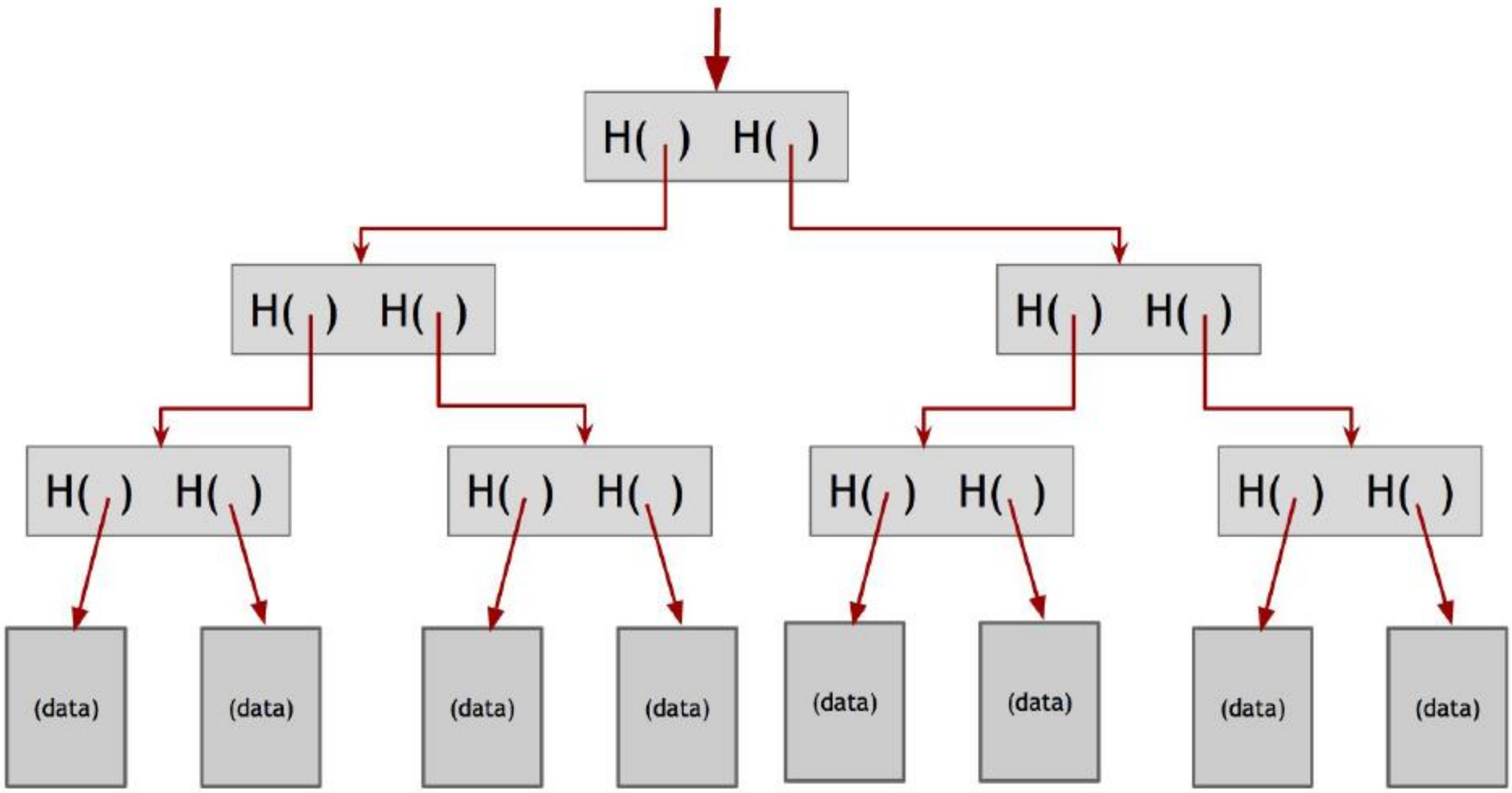


A **blockchain** is a linked list that is built using hash pointers instead of pointers

Linked List: Tampering Detection

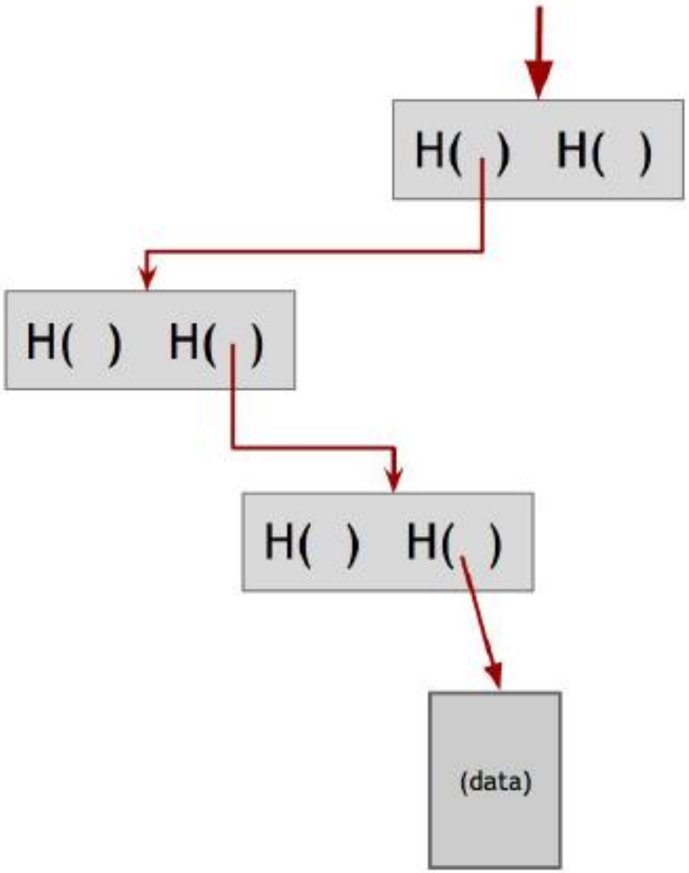


Binary Tree With Hash Pointers: Merkle Tree



https://prathamudeshmukh.github.io/merkle-tree-demo/

Proof of Membership in a Merkle Tree



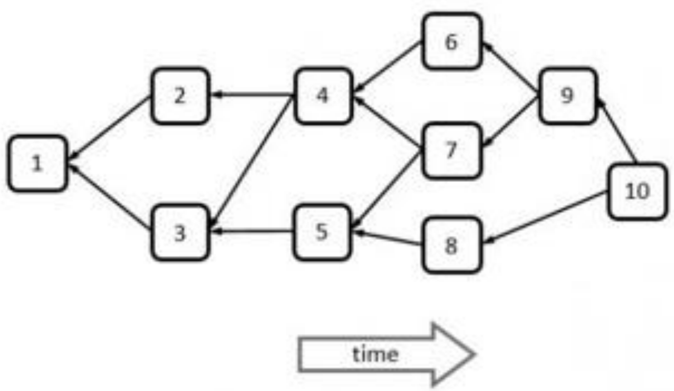
Advantages of Merkle Tree

Tree holds many items but just need to remember the root hash

Can verify the membership in $O(\log\ n)$ time

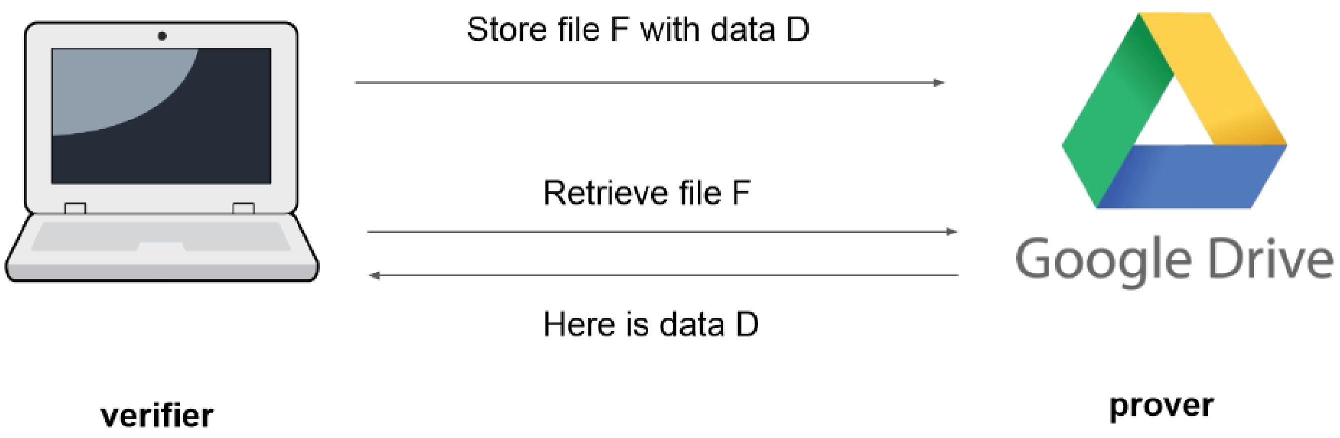
More generally we can use hash pointers in any pointer based data structure that has no cycle.

In case of cycles there is no node to start



The storage problem

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



The storage: Basic Protocol

- Client sends F with Data D to server
- Server stored (F, D)
- Client deletes D
- Client requests F from server
- Server returns D
- Client has recovered D

The storage protocol Against Adversaries

- What if server is adversial and returns D’ != D
- Trivial solution
  - Client does not delete D
  - Whenever server return D’ client can compare D and D’

What is client does not have memory to store data for a long time?

The storage : Hash based protocol

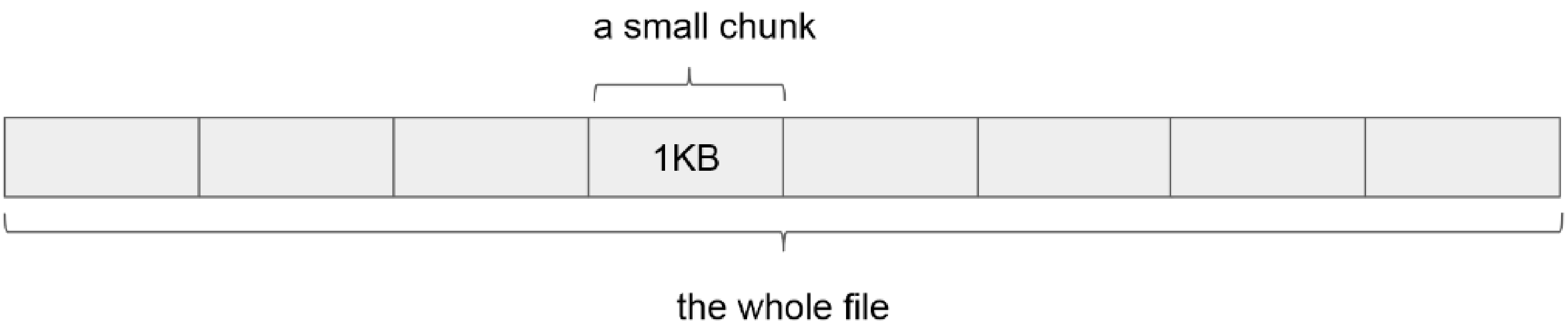
- Client send file F with Data D to the server
- Server stores (F,D)
- Client stored H(D), deletes D
- Client requests F from server
- Server returns D’
- Clinet compares H(D) = H(D’)

The storage : File chunks

- What if client wants to retrieve the 19007th byte of the file
- Must download the whole file
- Merkle tree to rescue.

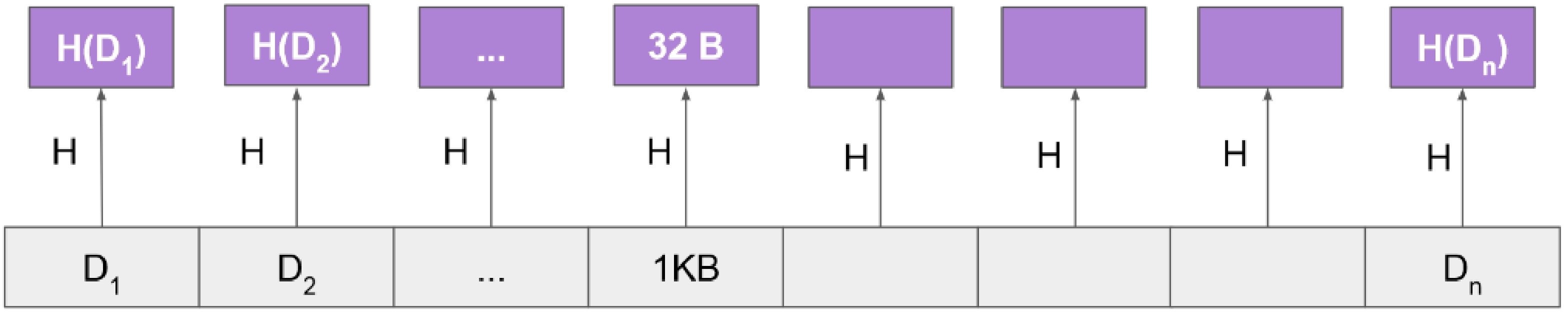
Merkle Tree:

- Splits file into chunks (say 1 KB)



Merkle Tree:

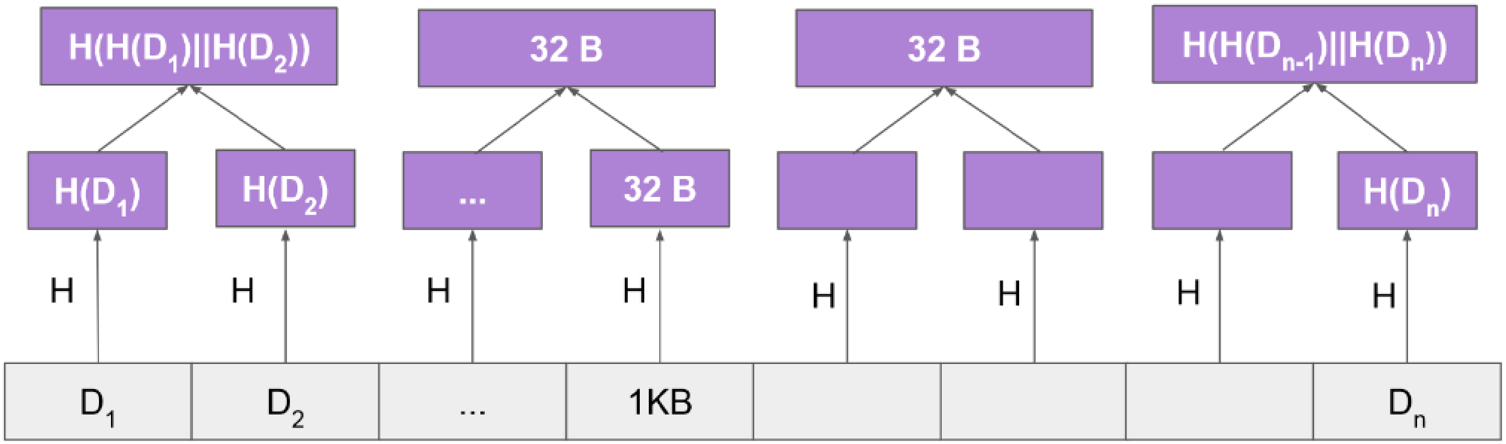
- Hash each chunk using cryptographic hash function



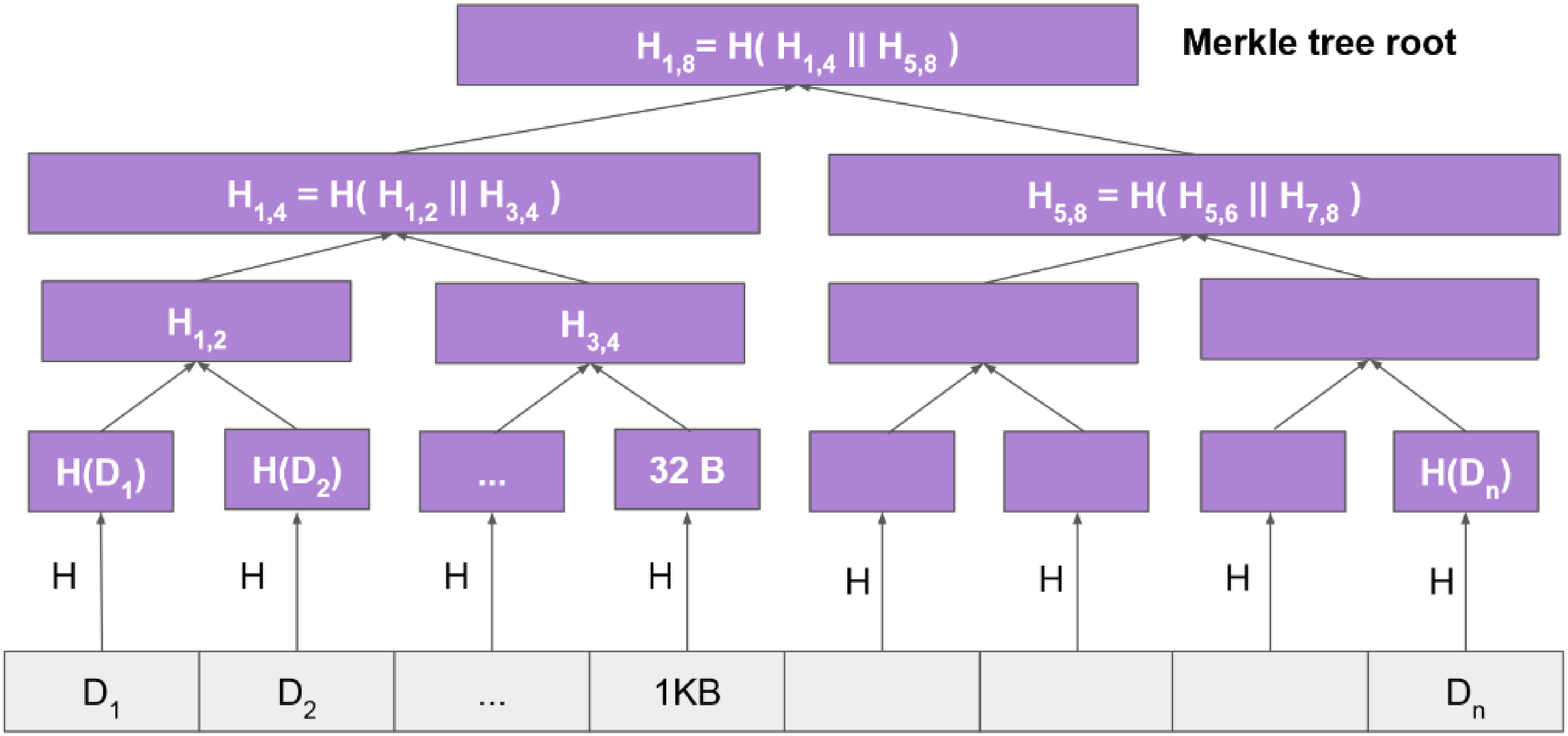
Arrows show direction of hash function application

Merkle Tree:

- Combine them to create a binary tree
- Each node stores the hash of the concatenation of their children



Merkle Tree:

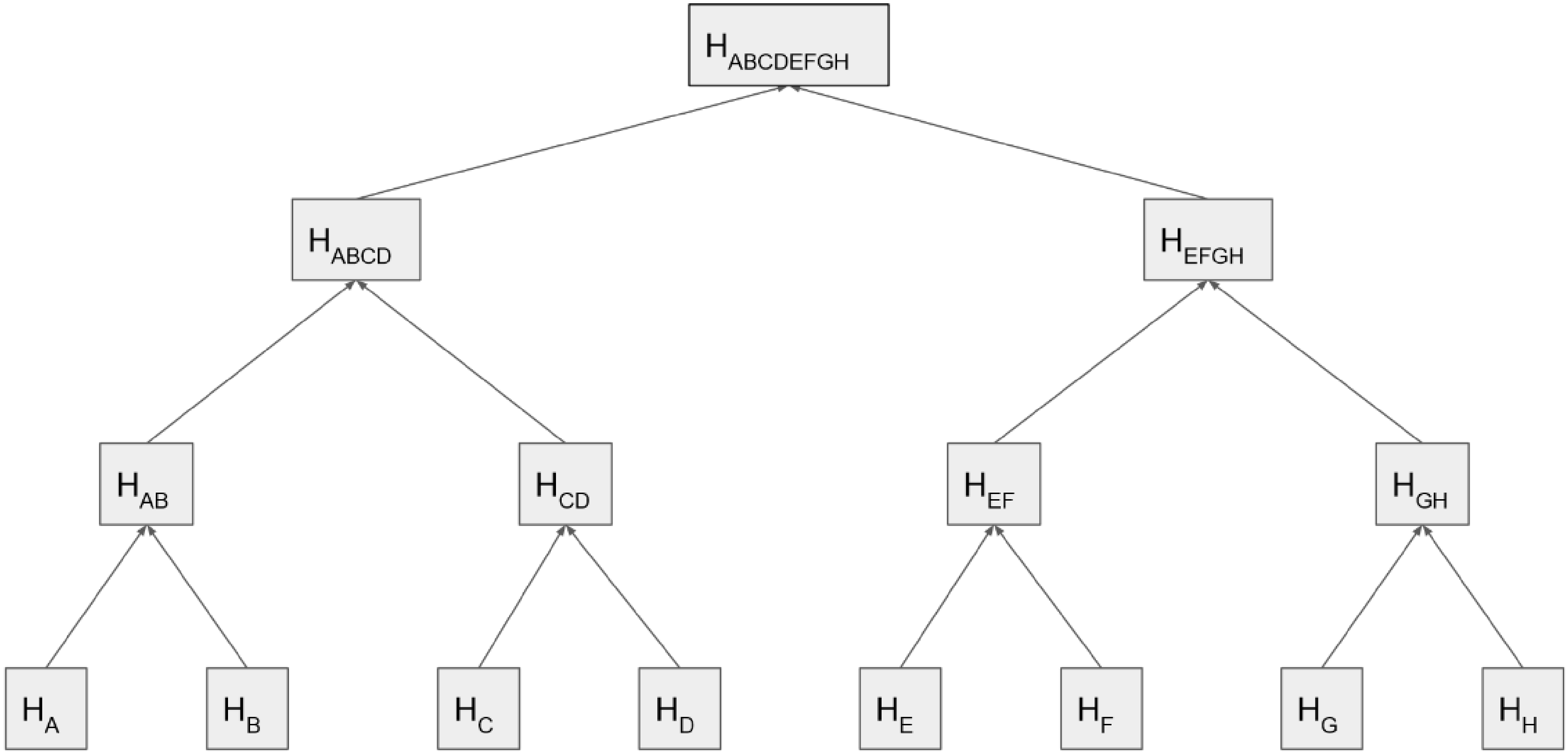


$H_{1,8} = H(H_{1,4} \| H_{5,8})$
Merkle tree root
$H_{1,4} = H(H_{1,2} \| H_{3,4})$
$H_{5,8} = H(H_{5,6} \| H_{7,8})$
$H_{1,2}$
$H_{3,4}$
$H(D_1)$
$H(D_2)$
...
32 B
$H(D_n)$
H
H
H
H
H
H
H
H
D_1
D_2
...
1KB
D_n

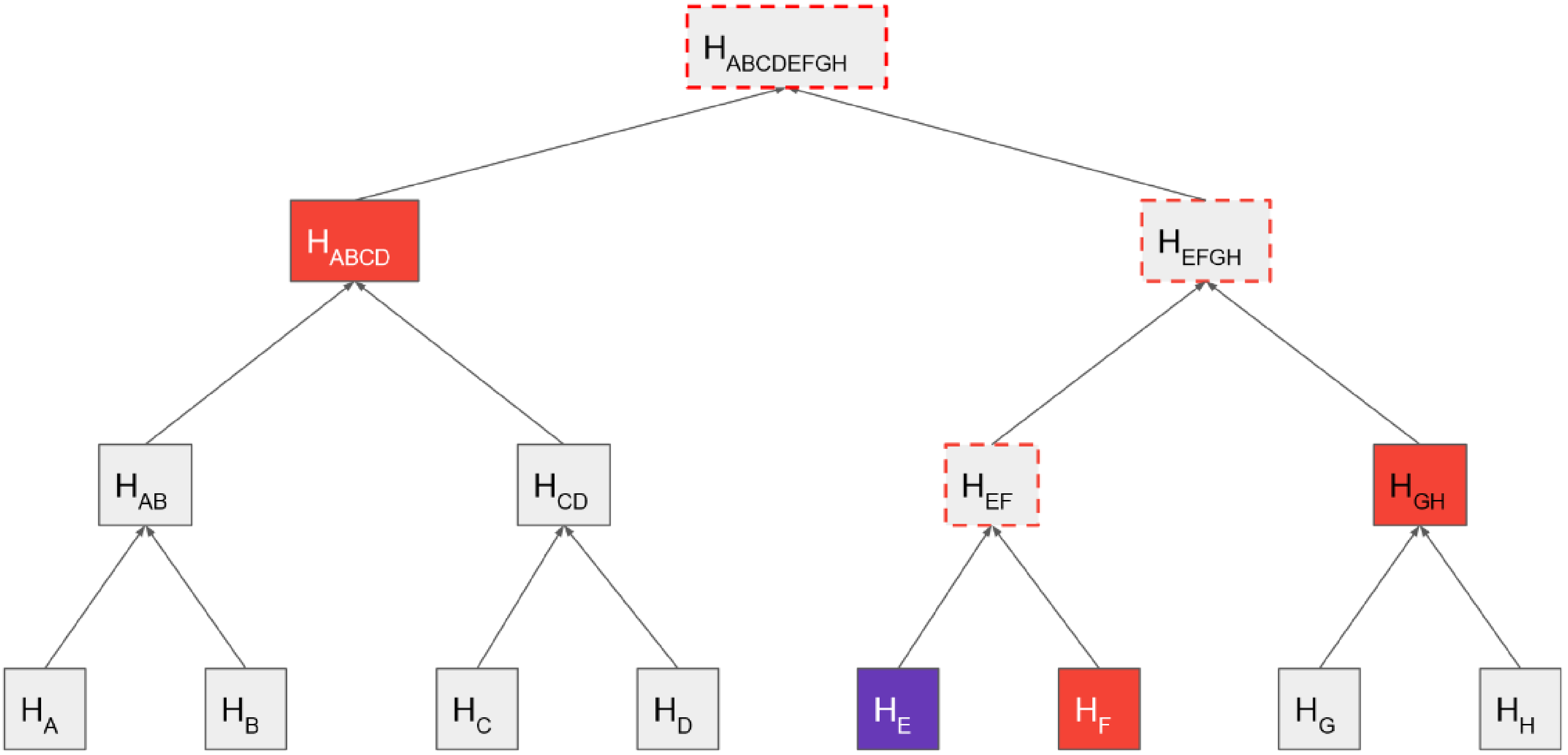
Proof of inclusion

- Client creates Merkle Tree with root MRT from file data D.
- Client send file data D to server.
- Client deletes data D but stores MTR
- Client request chunk X from the server
- Server returns chunk X and a short proof of inclusion π
- Client checks that chunk X is include in MTR using proof π

Proof of inclusion



Proof of inclusion



Proof of inclusion

- Prover sends chunks
- Prover sends siblings along path connecting leaf to MTR
- Verifier computes hashes along the path connecting leaf to MTR
- Verifier checks that computer root is = MTR
- The proof of inclusion is O(logn)
- If adversay can present proof-of-inclusion for incorrect leaf then we can break the hash function

Merkle Tree Protocol (Optional)

MT-Construct(D)

// Constructs a Merkle Tree with given Data D

//Return the Merkle tree root

If |D| = *chunk size* **then**

MT-Construct(D) = H(D)

Else

MT-Construct(D) = H(MT-Construct(D1) || MT-Construct(D2), where D = D1 || D2

Merkle Tree Protocol (Optional)

MT-Prove(D,x)

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

Merkle Tree Protocol (Optional)

MT-Verify(r, π ,x)

- Given Merkle root r, element x and proof-of-inclusion π
- Output True/False based on whether the verification was successful

Correctness

For all D, x :

MT-Verify(MT-Construct(D), MT-prove(D,x), x) = True

Merkle Tree Applications

- Bitcoin uses Merkle Tree to store the transactions
- Bit-Torrent uses Merkle tree to exchange file
- Etheriun Blockchain uses Merkle-Patricia tries for storage and transactions

Digital Signatures

What we want from Digital Signatures?

Only you can sign but any one can verify.

Signature is tied to a particular document

Can’t be cut and paste to another document.

API for digital signatures

(sk ; pk) := generateKeys(keySize)

sk : secret signing key

pk : Public verification key

sig := sign(sk ; message)

isValid : = verify(pk ; message; sig)

Requirements for Signatures

Valid Signatures Verify

verify(pk ; message; sign(sk ; message)) == true

Can’t forge signatures

Adversary who, knows pk , gets to see the signature of his own choice, can’t produce a verifiable signature on another message.

Practical Stuff ...

Algorithms to generate keys need to be randomized

So, we need a good source of randomness

Limit of message size

fix: use Hash(message) rather than message.

Fun Trick: Sign a hash pointer

Signature covers the whole structure

BITCOIN uses ECDSA standard for Digital Signatures

Useful trick: Public key == Identity

If you see sig such a verify(pk; msg; sig) == true

Think of it as

pk says “[msg]”

To speak for **pk** you must know **sk**

Decentralized Identity Management

Anybody can make a new identity at anytime

make as many as you want

No central point of coordination

<u>These identities are called “addresses” in Bitcoin</u>

Privacy

Addresses not directly connected to real world identity

But observer can link together an address's activity over time