Crypto Background

Blockchains and Cryptocurrencies
(Spring 2019)

This lecture

Crypto background

- hash functions
- random oracle model
- digital signatures
- … and applications

Cryptographic Hash Functions

Hash function

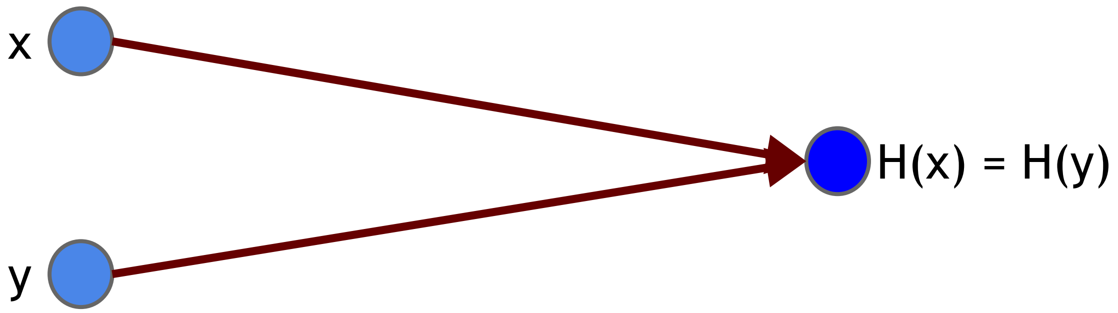
- takes a string of arbitrary length as input
- fixed-size output (i.e., hash function “compresses” the input)
- efficiently computable

Security properties:

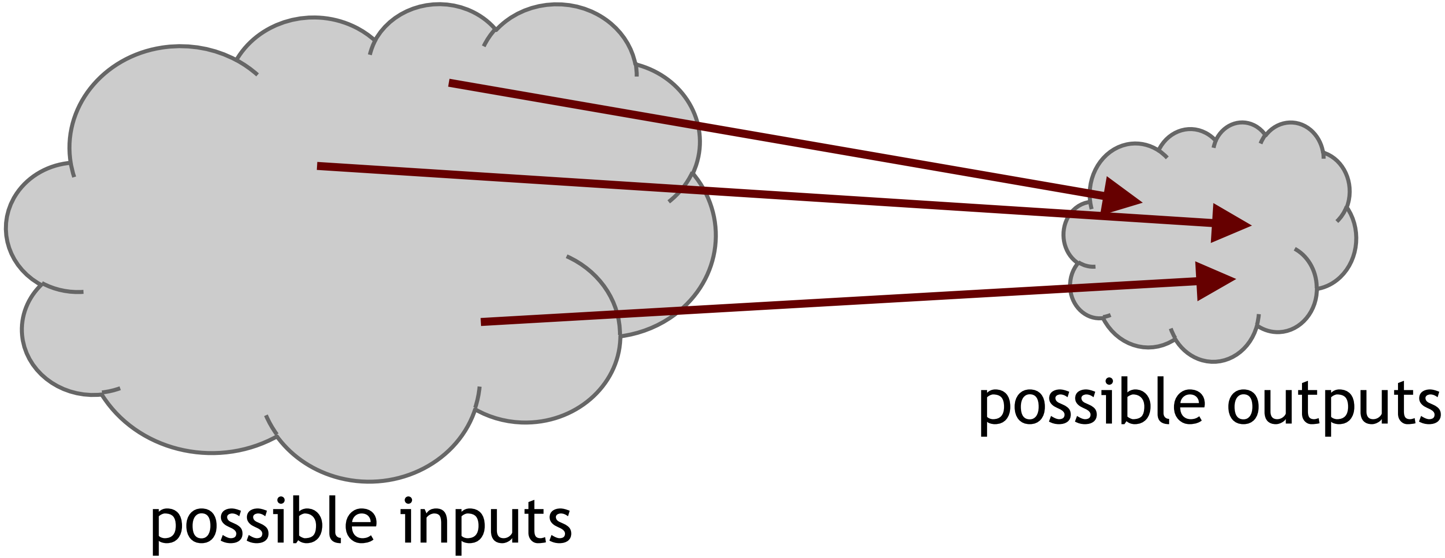
- Collision resistance
- Preimage resistance (one-way)

Property 1: Collision resistance

No efficient adversary can find x and y such that x != y and H(x)=H(y)



Collisions do exist ...



... but can a real-world adversary find them?

How to find a collision (for 256 bit output)

- try 2^{130} randomly chosen inputs
- 99.8% chance that two of them will collide

This works no matter what H is, but it takes too long to matter

- If a computer calculates 10,000 hashes/sec, it would take 10^{27} years to compute 2^{128} hashes

Is there a faster way to find collisions?

- For some possible H’s, yes.
- For others (like SHA-256), we don’t know of one.

Provably secure collision-resistant hash functions can be constructed based on “hard” number-theoretic problems.

Defining Collision Resistance

- Real-world adversaries
  - In practice, everyone has bounded resources
  - Therefore, reasonable to model a real-world adversary as such an entity
  - However, we do not make any assumptions about the adversarial strategy. He can use its (bounded) resources in any possible way

Cryptographic adversary: A probabilistic polynomial-time (PPT) algorithm

Defining Collision Resistance…

- Collision Resistance (informal): A hash function H is collision-resistant if for all PPT adversaries A,

Pr[A outputs x,y s.t. x!=y and H(x)=H(y)]
= “very small”

Defining Collision Resistance…

- Collision Resistance (informal): A hash function H is collision-resistant if for all PPT adversaries A,

  Pr[A outputs x,y s.t. x!=y and H(x)=H(y)]

  = "very small"

- "Very small" captured via a function that tends to 0. Formal definition: Modern Cryptography (next semester)

Application: Hash as message digest

If we know H(x) = H(y),

it's safe to assume that x = y.

To recognize a file that we saw before,

just remember its hash.

Useful because the hash is small.

Property 2: Pre-image Resistance

Intuition: Given H(x), no efficient adversary can find x, except with very small probability

Problem: What if input space of x is very small, or some inputs are much more likely than others?

H("heads")

H("tails")

easy to find x!

Defining Preimage Resistance

- **Preimage Resistance**: A hash function H is preimage-resistant if for all PPT adversaries A,

$$\Pr[x \leftarrow \{0,1\}^k,\ A(H(x)) \text{ outputs } x' \text{ s.t. } H(x')=H(x)] = \text{small}$$

x is drawn from uniform distribution over $\{0,1\}^k$ for some sufficiently large k

Preimage Resistance (contd.)

- If x is drawn from the uniform distribution, then inverting H(x) is hard
- But what if x is drawn from low-entropy distribution?
- Can append a random string r to x and then compute H(r | x) to prevent enumeration attacks

Theorem: Collision resistance implies preimage resistance if the hash function is sufficiently compressing

Application: Commitment

Want to "seal a value in an envelope", and
"open the envelope" later.

Commit to a value, reveal it later.

Commitment Schemes

(*com*, *key*) := commit(*msg*)

match := verify(*com*, *key*, *msg*)

To seal *msg* in envelope:

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

To open envelope:

publish *key*, *msg*

anyone can use verify() to check validity

Commitment Schemes

(*com*, *key*) ← commit(*msg*)

match ← verify(*com*, *key*, *msg*)

Security properties:

- Hiding: Given *com*, no PPT adversary can find* *msg*
- Binding: No PPT adversary can find* *msg* != *msg'* such that verify(commit(*msg*), *msg'*) == true

* Except with very small probability

Commitment Schemes

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: If H is a ***random oracle***, given H(*key* | *msg*), hard to find *msg*.
- Binding: Collision-reistance → Hard to find *msg* != *msg'* such that H(*key* | *msg*) == H(*key* | *msg'*)

Random Oracle (RO)

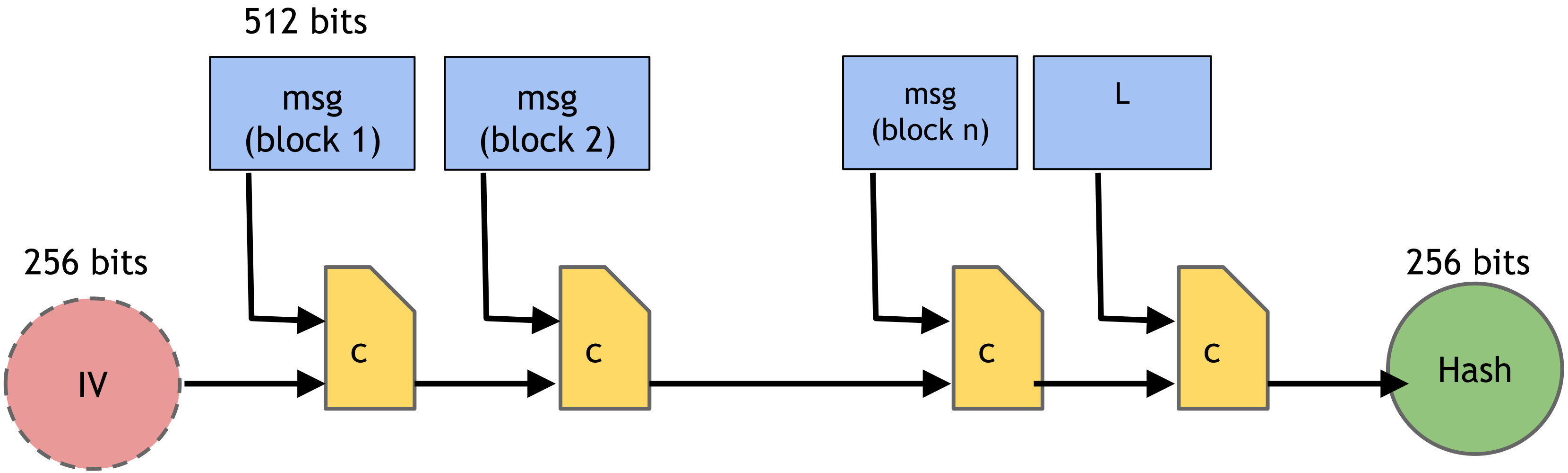
- Imagine an elf in a box with an infinite writing scroll
- Upon receiving an input x, the elf checks the scroll if there is an entry y corresponding to x. If yes, it returns y.
- Otherwise, elf chooses a random value y (from the output space) and returns it. It adds an entry (x,y) to the scroll.

Random Oracle (RO)

- In practice-oriented provable security, hash functions are often modeled as a random oracle
- Each party (including adversary) is given black-box access to the random oracle. They can query the random oracle any polynomial number of times
- By definition, the answers of random oracle answers are unpredictable
- Random oracle captures many security properties such as one-wayness, collision-resistance .

SHA-256 hash function

Suppose msg is of length L s.t. L is a multiple of 512 (pad with 0s otherwise)



Theorem [Merkle-Damgard]: If c is collision-resistant, then SHA-256 is collision-resistant.

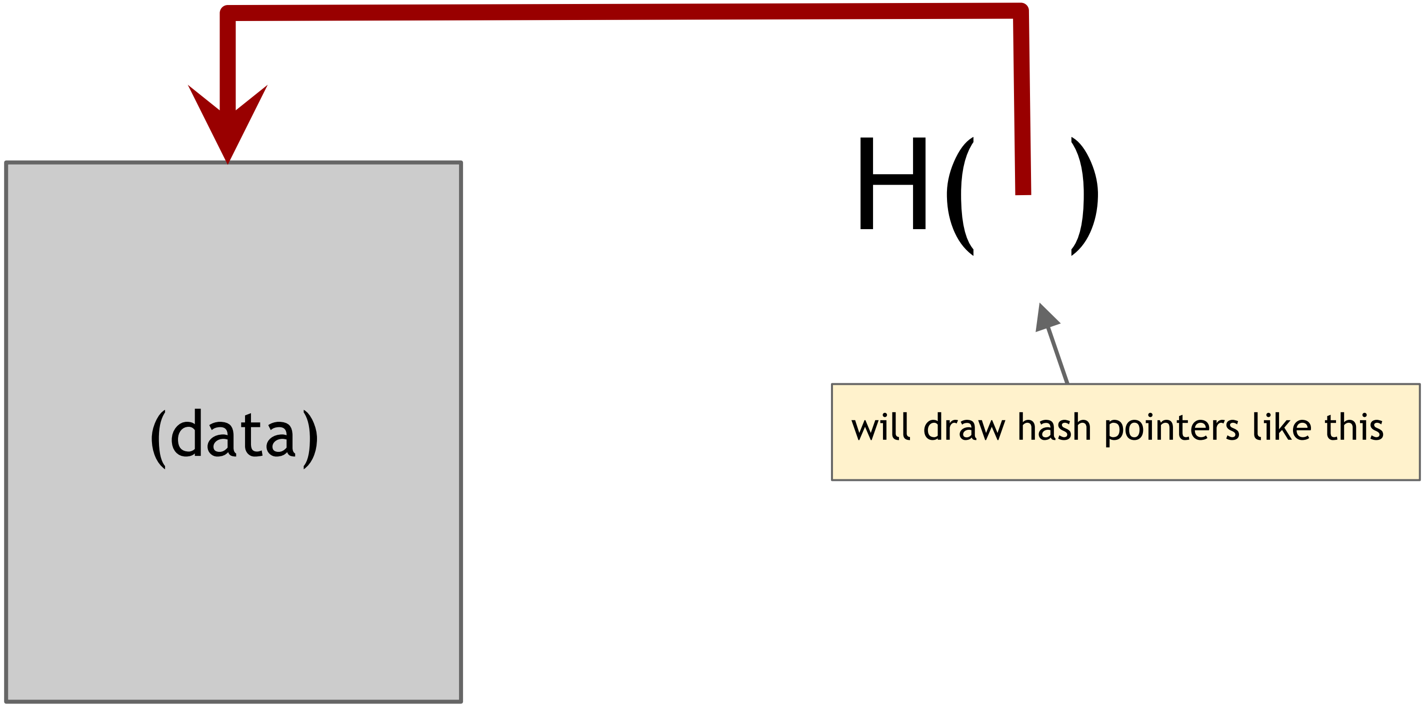
Hash Pointers and Data Structures

Hash pointer

- pointer to where some info is stored, *and*
- cryptographic hash of the info

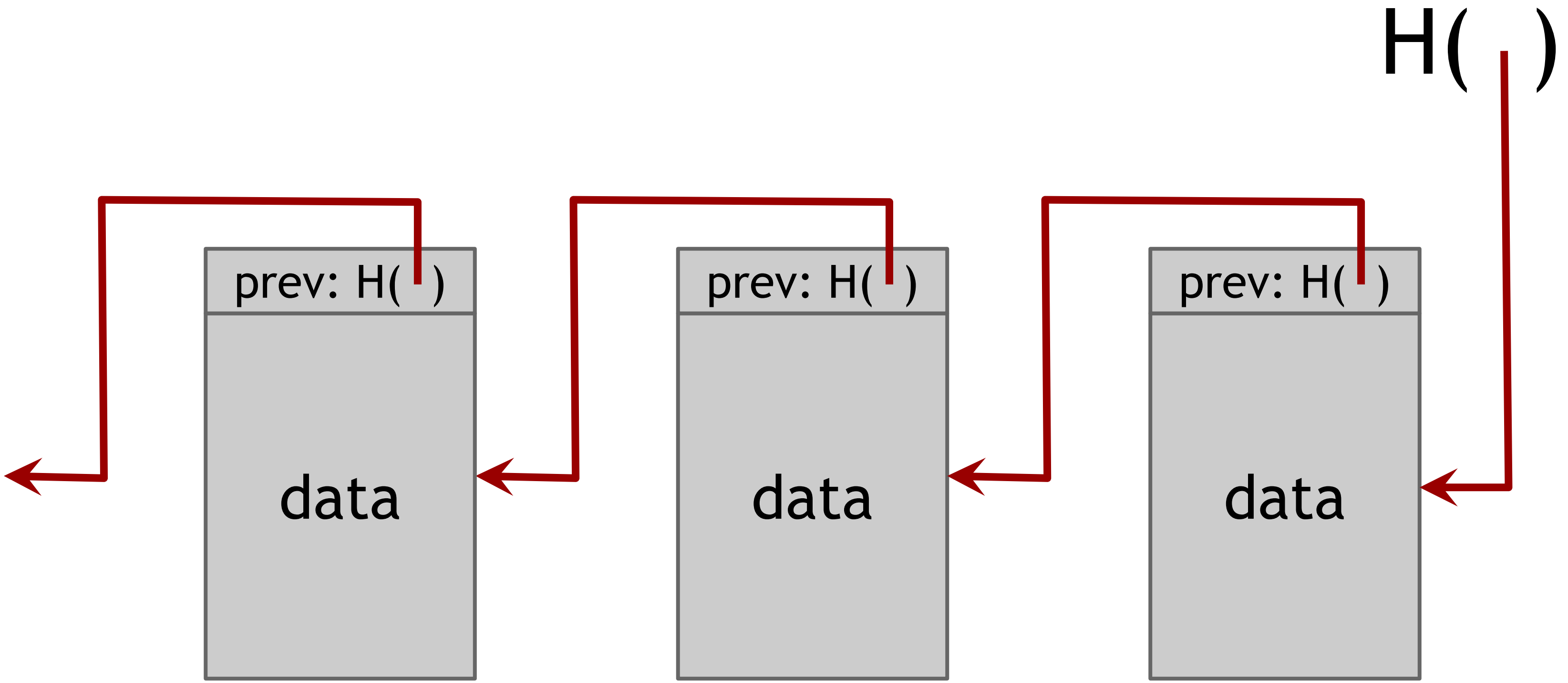
If we have a hash pointer, we can

- ask to get the info back, and
- verify that it hasn't changed


(data)
H( )
will draw hash pointers like this

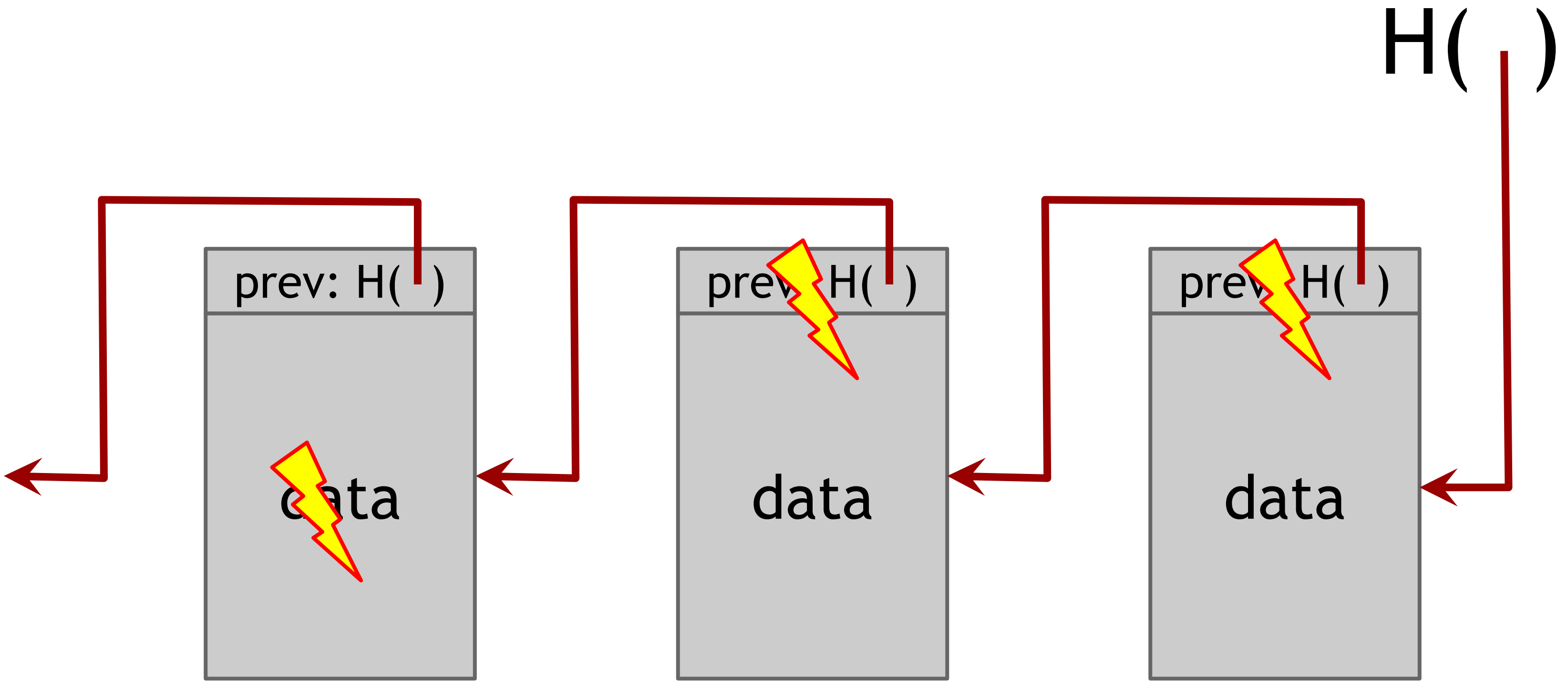
Building data structures with hash pointers

Linked list with hash pointers = “Blockchain”



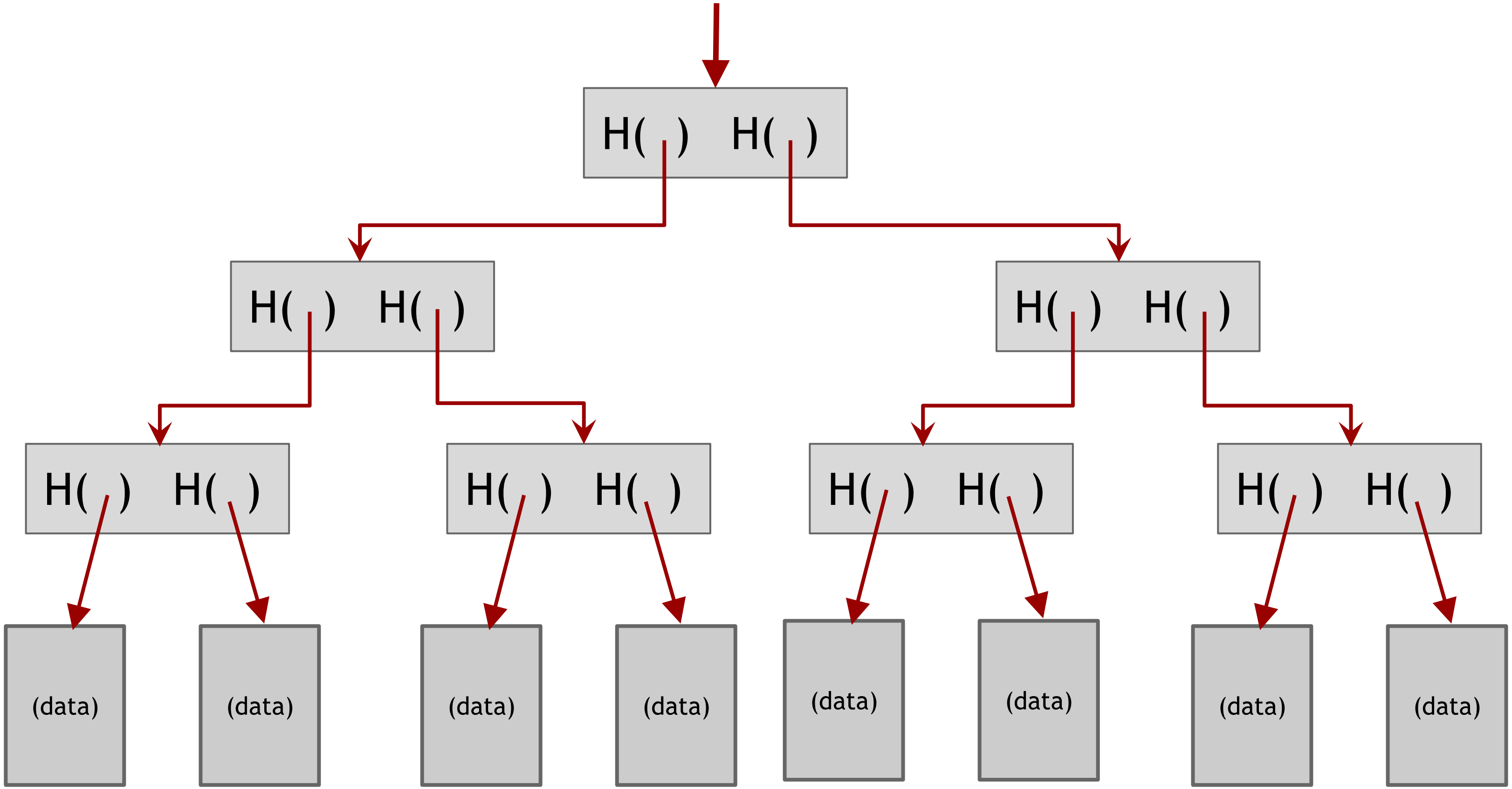
use case: tamper-evident log

detecting tampering



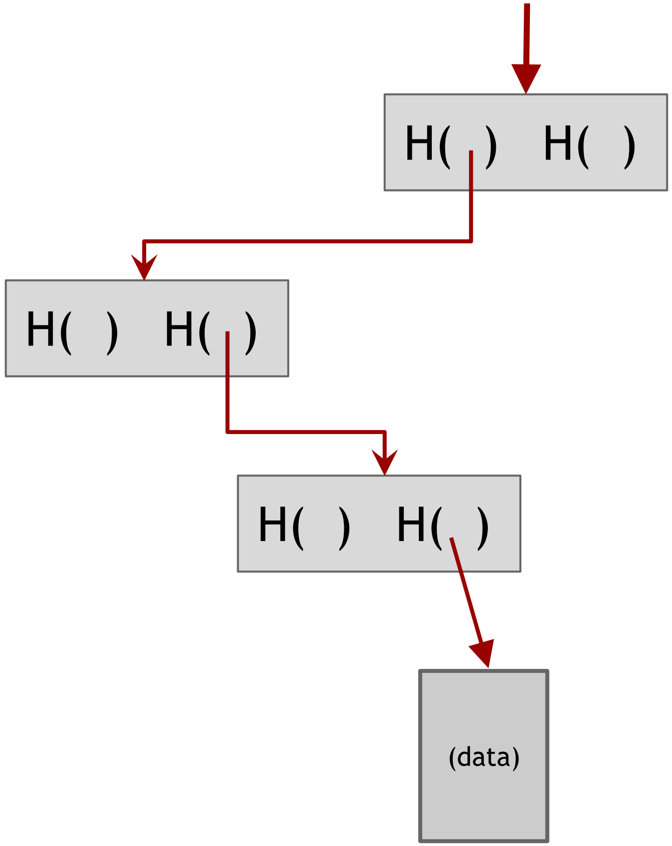
use case: tamper-evident log

binary tree with hash pointers = “Merkle tree”



proving membership in a Merkle tree

show O(log n) items



Advantages of Merkle trees

- Tree holds many items, but just need to remember the root hash
- Can verify membership in O(log n) time/space

Variant: *sorted* Merkle tree

- can verify non-membership in O(log n)
- show items before, after the missing one

More generally ...

Can use hash pointers in any pointer-based
data structure that has no cycles

Digital Signatures

What we want from signatures

- Only you can sign, but anyone can verify
- Signature is tied to a particular document

  (*can't be cut-and-pasted to another doc)*
- Even if one can see your signature on some documents, he cannot "forge" it

Digital signatures

randomness

- (sk, pk) ← keygen(r)

  sk: secret signing key

  pk: public verification key

- sig ← sign(sk, message)

- isValid ← verify(pk, message, sig)

randomized algorithm

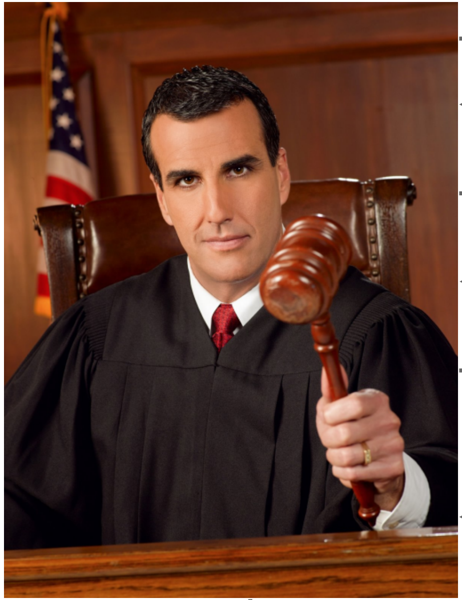
Typically randomized

Requirements for signatures

- Correctness: “valid signatures verify”
    - verify(pk, message, sign(sk, message)) == true
- Unforgeability under chosen-message attacks (UF-CMA): “can’t forge signatures”
    - adversary who knows pk, and gets to see signatures on messages of his choice, can’t produce a verifiable signature on another message

UF-CMA Security

(sk, pk) ← keygen(1^k)



pk

m_0

sign(sk, m_0)

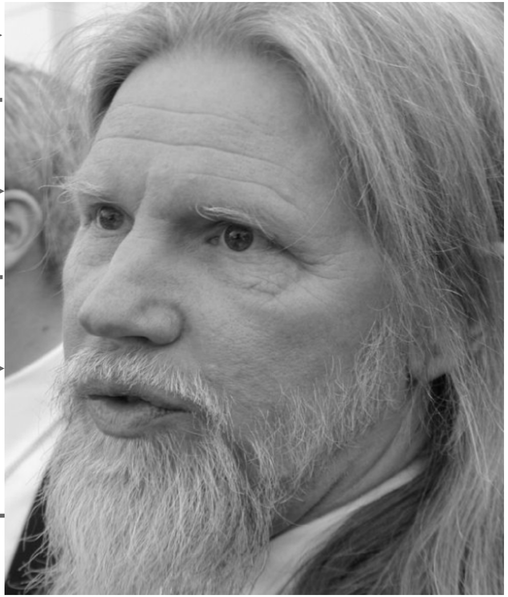
m_1

sign(sk, m_1)

• • •

M, sig

M not in { m_0, m_1, … }



Challenger

verify(pk, M, sig)

ifValid, attacker wins

Adversary

Definition: A signature scheme (keygen,sign,verify) is UF-CMA secure if for every PPT adversary A, Pr[A wins in above game] = very small

Notes

- Algorithms are randomized: need good source of randomness. Bad randomness may reveal the secret key
- fun trick: sign a hash pointer. signature “covers” the whole structure
- Bitcoin uses Elliptic Curve Digital Signature Algorithm (ECDSA), a close variant of Schnorr over Elliptic curves