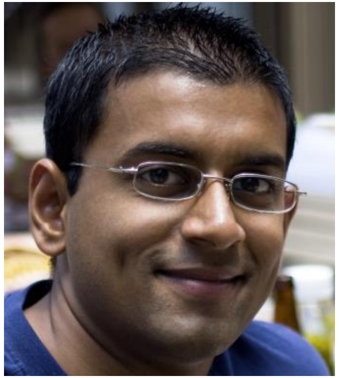
Lecture 1

Intro to Crypto and Cryptocurrencies

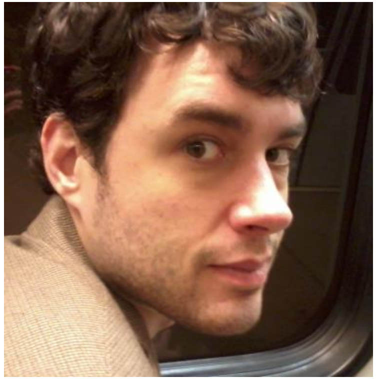
Welcome

Joseph Bonneau

Ed Felten



Arvind Narayanan



special guest:
Andrew Miller

This lecture

Crypto background

- hash functions
- digital signatures
- … and applications

Intro to cryptocurrencies

- basic digital cash

Lecture 1.1:

Cryptographic Hash Functions

Hash function:

- takes any string as input
- fixed-size output (we’ll use 256 bits)
- efficiently computable

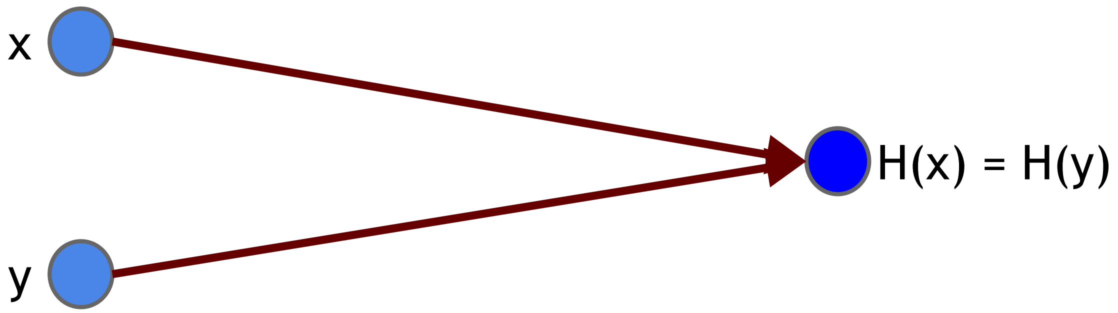
Security properties:

- collision-free
- hiding
- puzzle-friendly

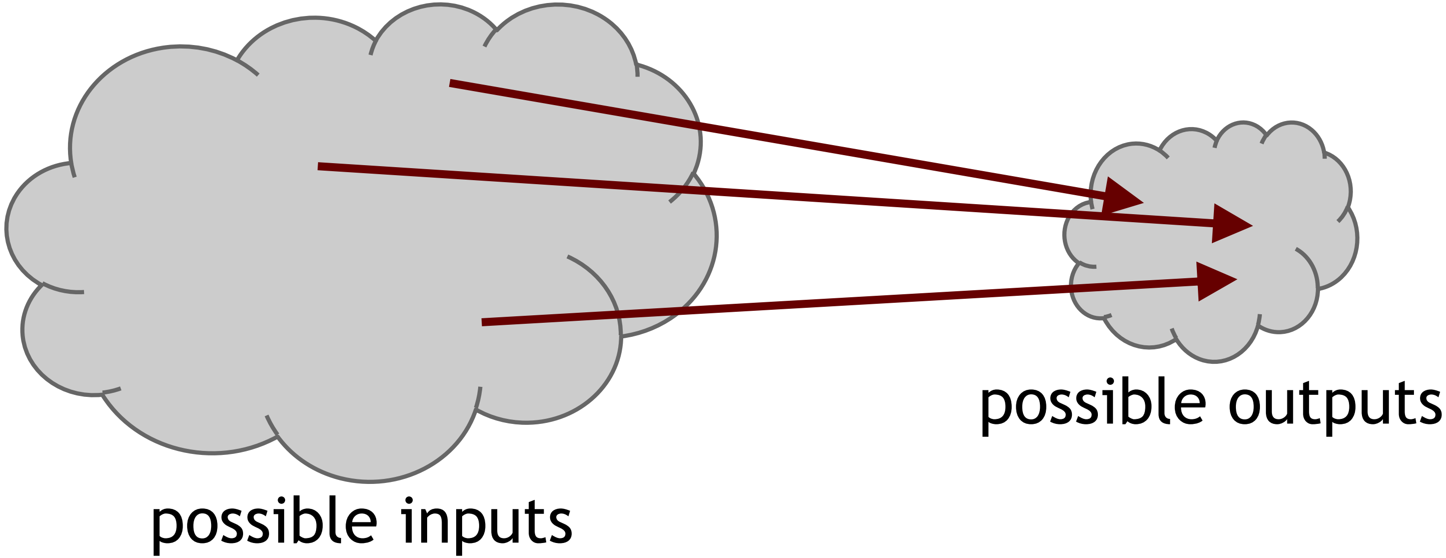
Hash property 1: Collision-free

Nobody can find x and y such that

x != y and H(x)=H(y)



Collisions do exist …



… but can anyone find them?

How to find a collision

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

Is there a faster way to find collisions?

For some possible H's, yes.

For others, we don't know of one.

No H has been proven collision-free.

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.

Hash property 2: Hiding

We want something like this:
Given H(x), it is infeasible to find x.

H("heads")

H("tails")

easy to find x!

Hash property 2: Hiding

<u>Hiding property</u>:
If 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”, so that no particular value is chosen with more than negligible probability.

Application: Commitment

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

Commit to a value, reveal it later.

Commitment API

(*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 API

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

Security properties:

Hiding: Given *com*, infeasible to find *msg*.

Binding: Infeasible to find *msg != msg'* such that verify(commit(*msg*), *msg'*) == true

Commitment API

commit(*msg*) := (H(*key* | *msg*), H(*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'* such that

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

Hash property 3: Puzzle-friendly

Puzzle-friendly:

For every possible output value y,
if k is chosen from a distribution with high min-entropy,
then it is infeasible to find x such that $H(k \mid x) = y$.

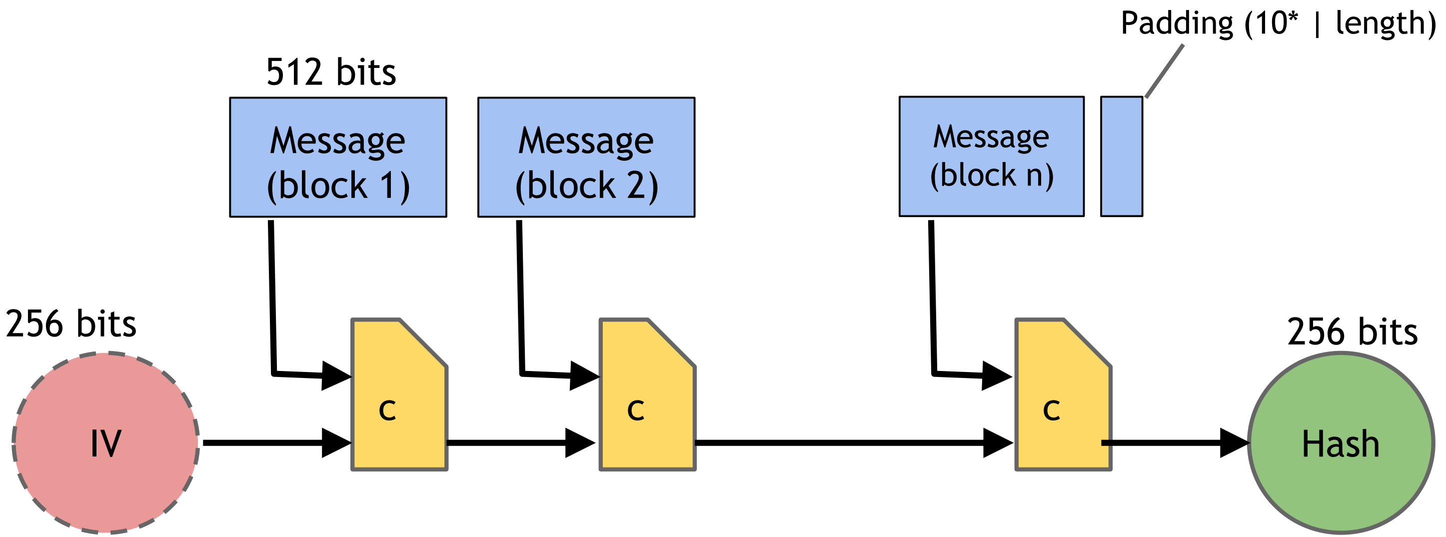
Application: Search puzzle

Given a "puzzle ID" *id* (from high min-entropy distrib.),
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 is collision-free, then SHA-256 is collision-free.

Lecture 1.2:

Hash Pointers and Data Structures

hash pointer is:

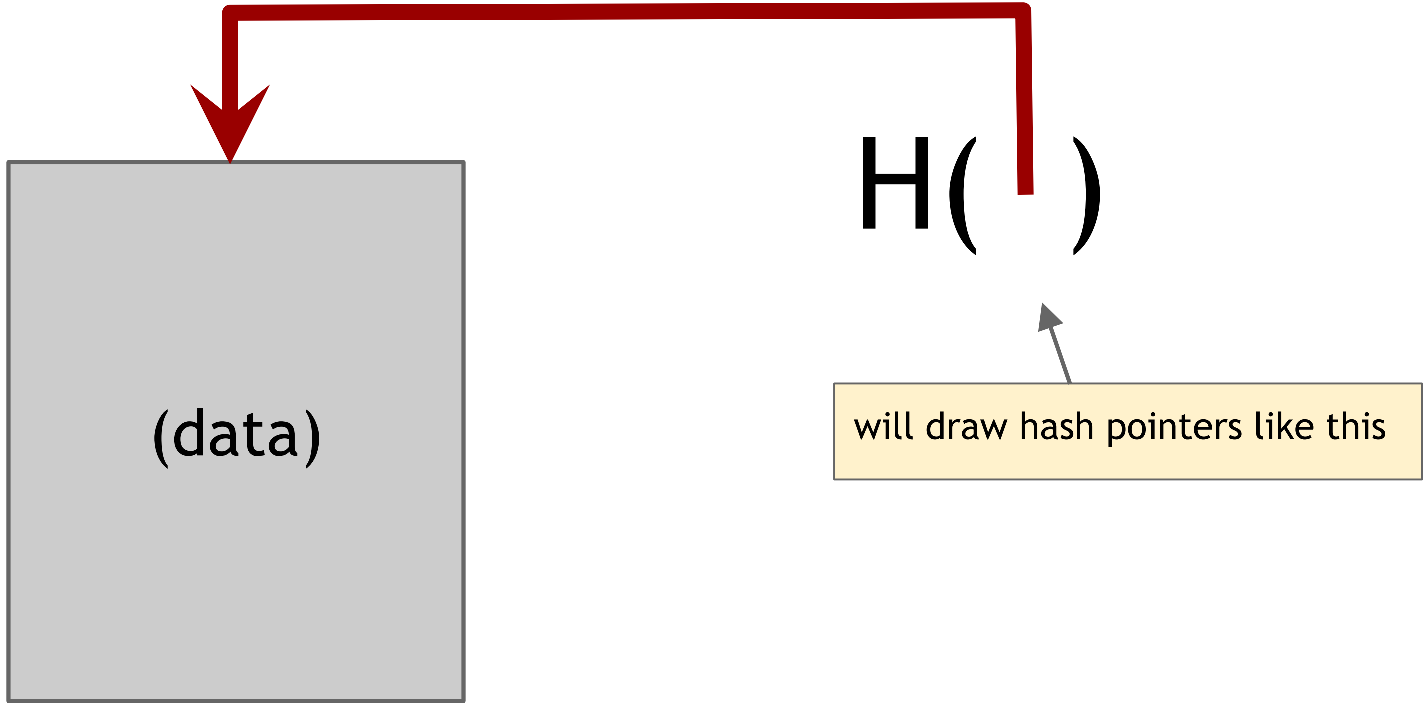
* pointer to where some info is stored,

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

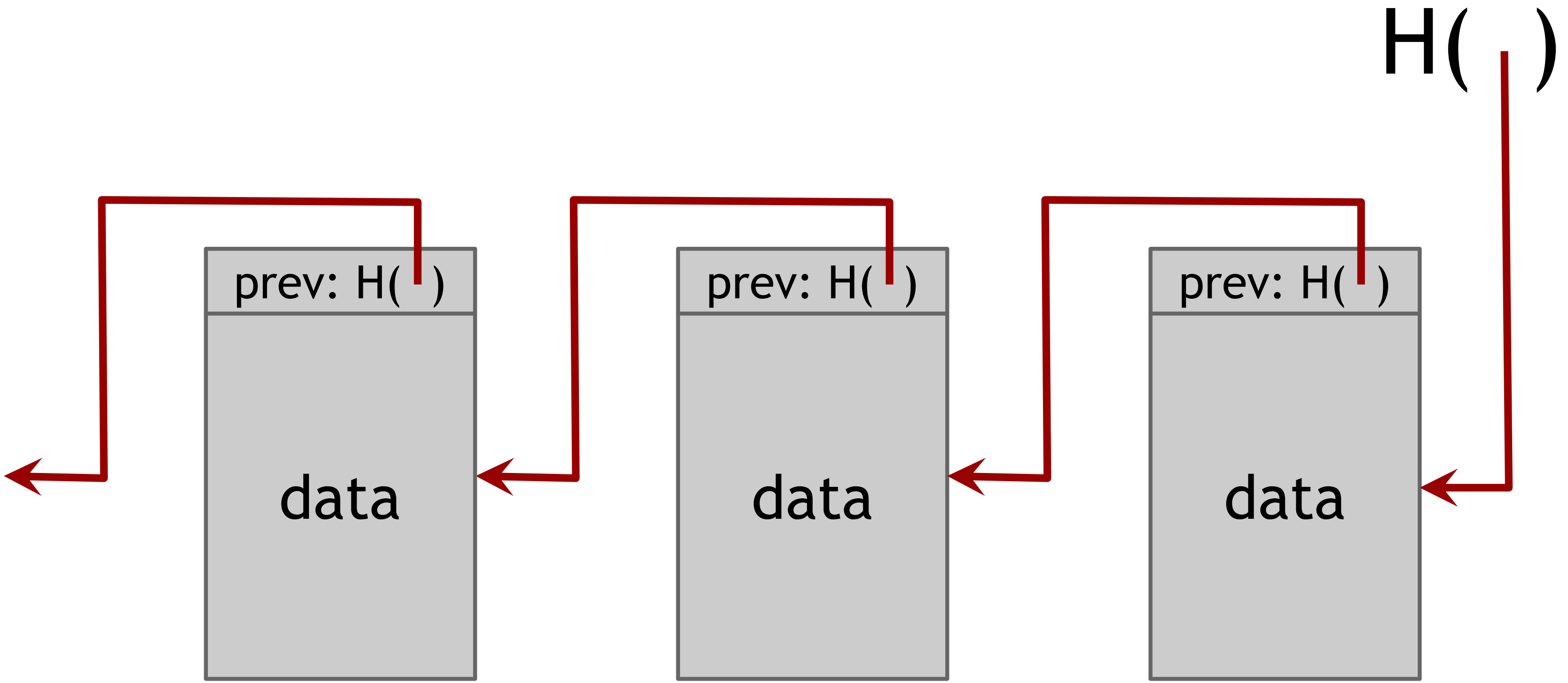


H( )
(data)
will draw hash pointers like this

key idea:

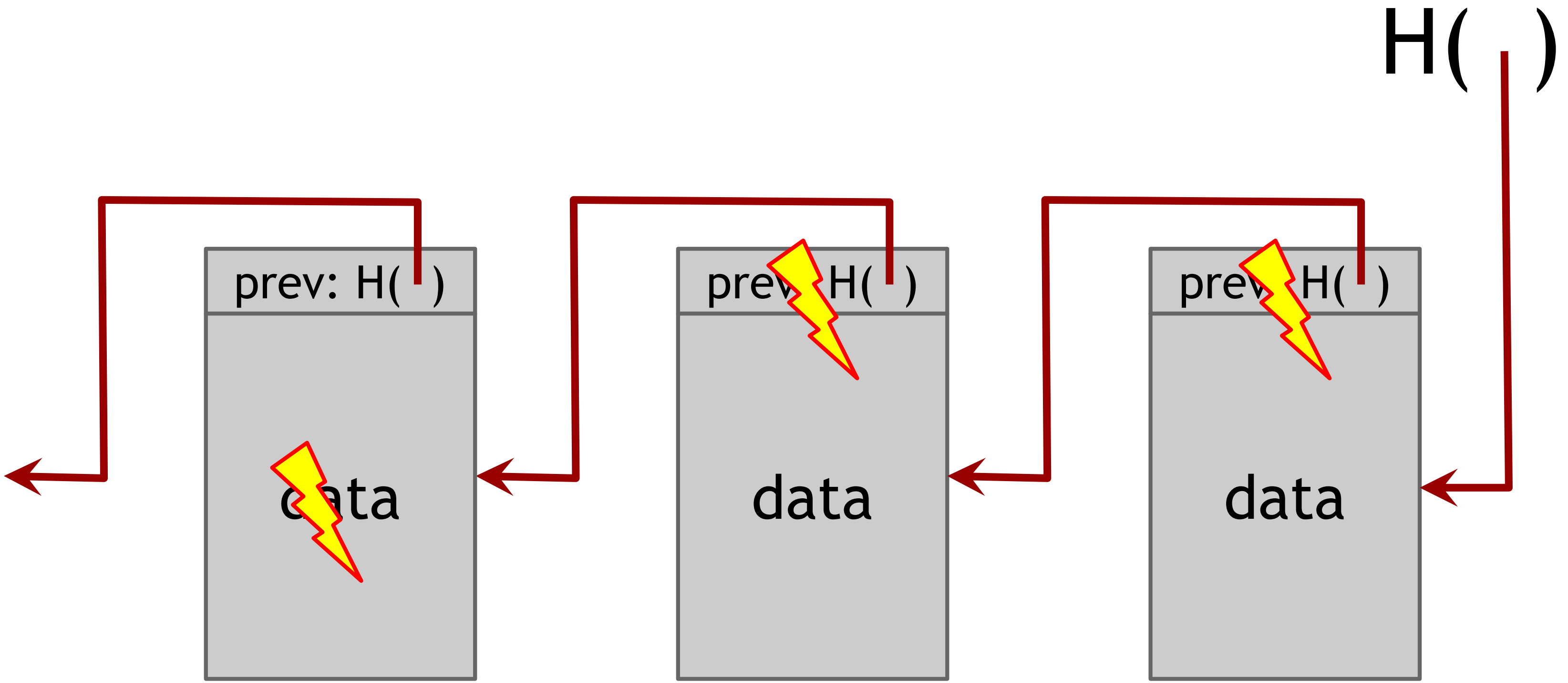
build data structures with hash pointers

linked list with hash pointers = “block chain”



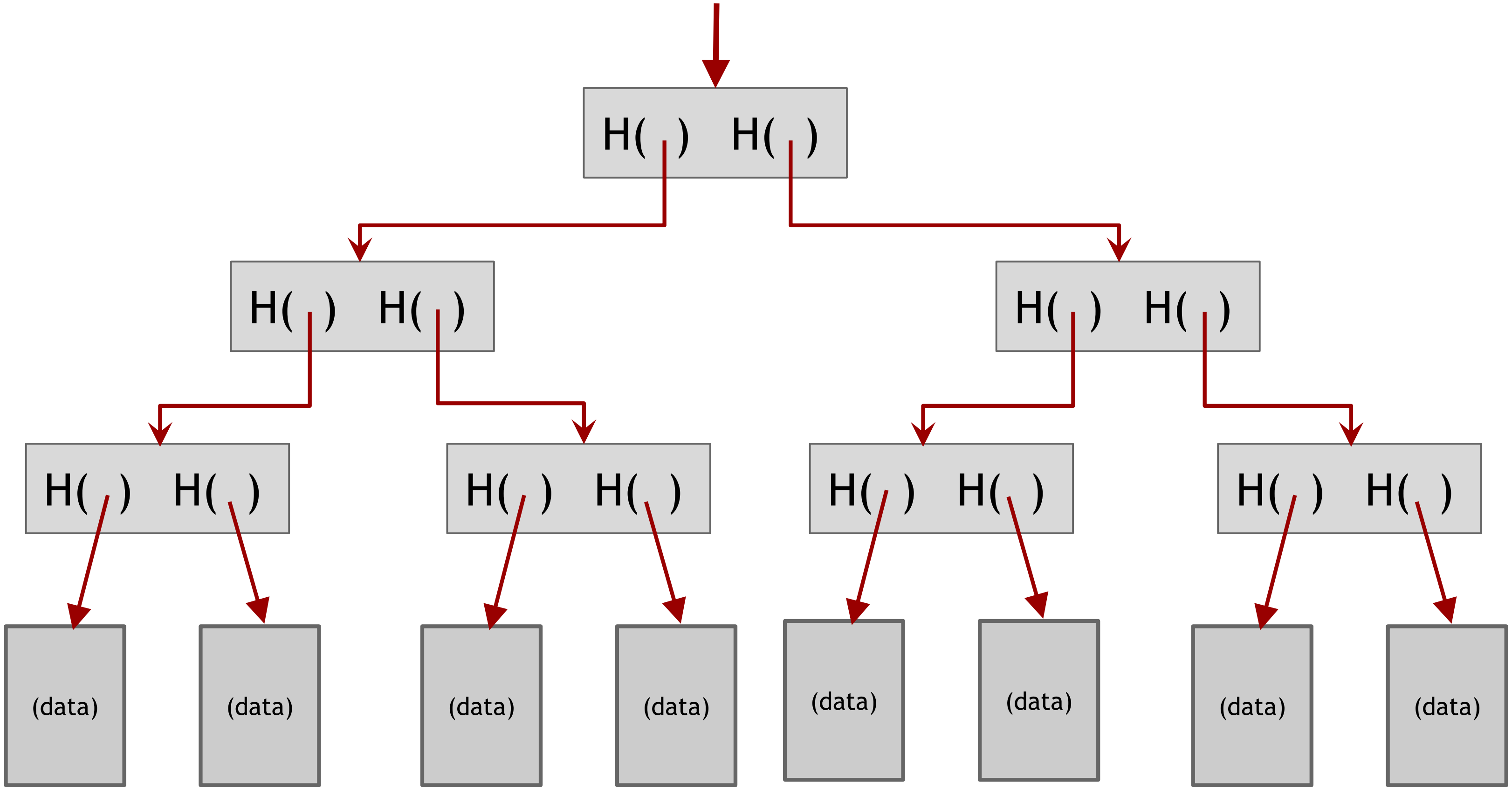
use case: tamper-evident log

detecting tampering

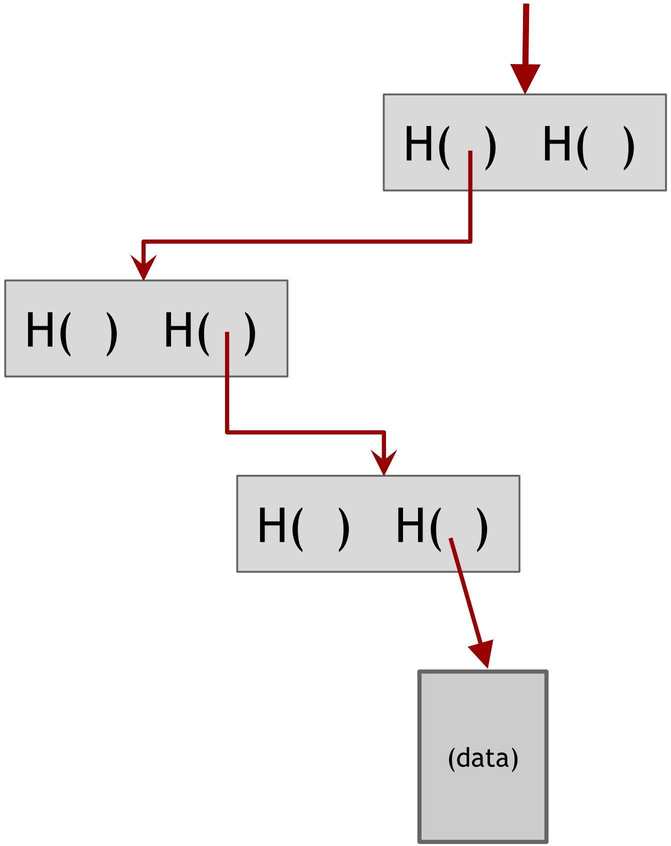


use case: tamper-evident log

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

Lecture 1.3:

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

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)

can be randomized algorithms

Requirements for signatures

“valid signatures verify”

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

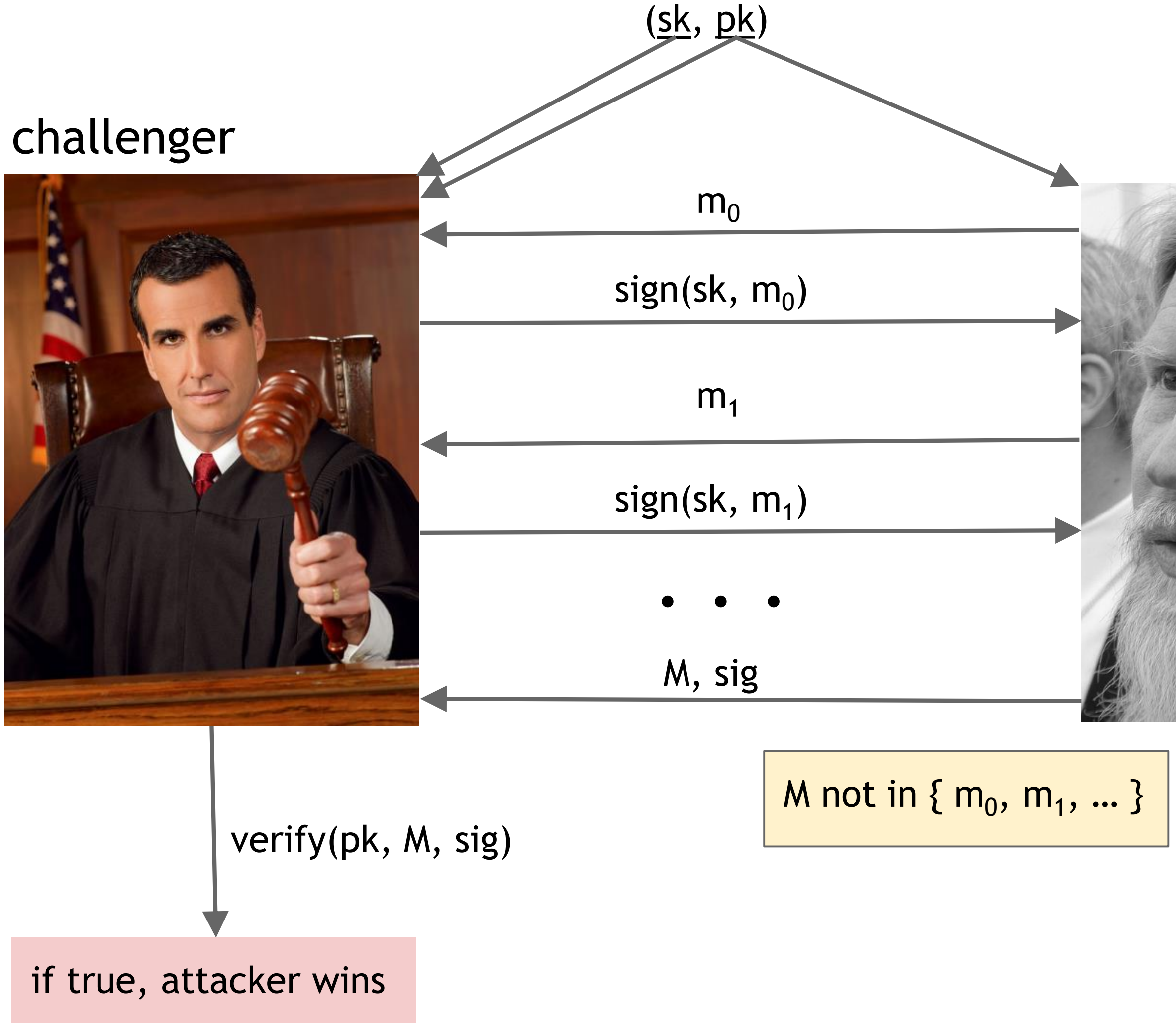
“can’t forge signatures”

adversary who:

knows pk

gets to see signatures on messages of his choice

can’t produce a verifiable signature on another message


(sk, pk)
challenger
attacker
m0
sign(sk, m0)
m1
sign(sk, m1)
. . .
M, sig
M not in { m0, m1, ... }
verify(pk, M, sig)
if true, attacker wins

Practical stuff…

algorithms are randomized

need good source of randomness

limit on message size

fix: use Hash(message) rather than message

fun trick: sign a hash pointer

signature “covers” the whole structure

Bitcoin uses ECDSA standard

Elliptic Curve Digital Signature Algorithm

relies on hairy math

will skip the details here --- look it up if you care

good randomness is essential

foul this up in generateKeys() or sign() ?

probably leaked your private key

Lecture 1.4:

Public Keys as Identities

Useful trick: public key == an identity

if you see *sig* such that *verify(pk*, *msg*, *sig)==true*,
think of it as

pk says, "*[msg]*".

to "speak for" *pk*, you must know matching secret key *sk*

How to make a new identity

create a new, random key-pair *(sk, pk)*

pk is the public "name" you can use

[usually better to use Hash(pk)]

sk lets you "speak for" the identity

you control the identity, because only you know *sk*

if *pk* "looks random", nobody needs to know who you are

Decentralized identity management

anybody can make a new identity at any time

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, make inferences.

Later: a whole lecture on privacy in Bitcoin ...

Lecture 1.5:

Simple Cryptocurrencies

GoofyCoin

Goofy can create new coins

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

New coins belong to me.

A coin's owner can spend it.

signed by pk_{Goofy}

Pay to pk_{Alice} : H( )

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

Alice owns it now.

The recipient can pass on the coin again.

signed by pk_{Alice}

Pay to pk_{Bob} : H( )

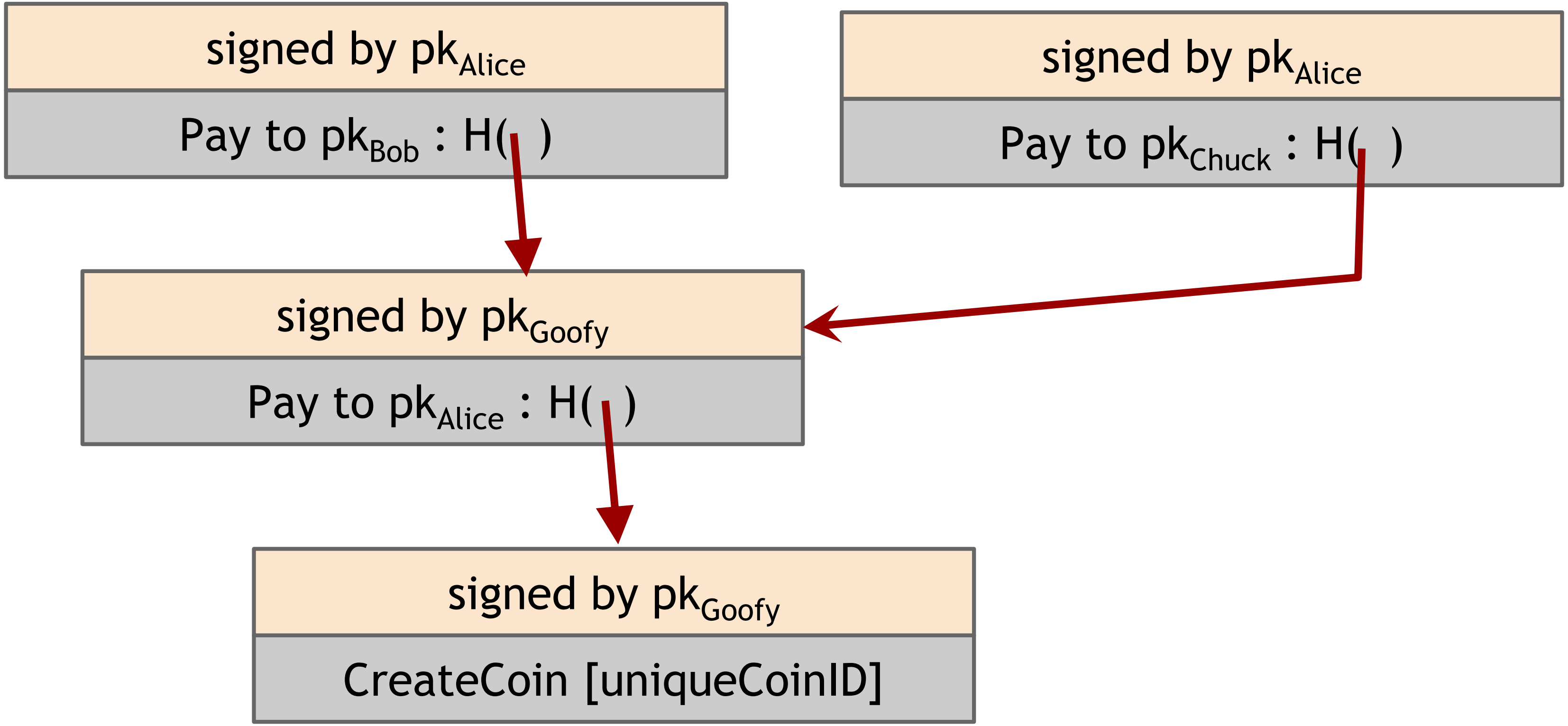
signed by pk_{Goofy}

Pay to pk_{Alice} : H( )

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

double-spending attack

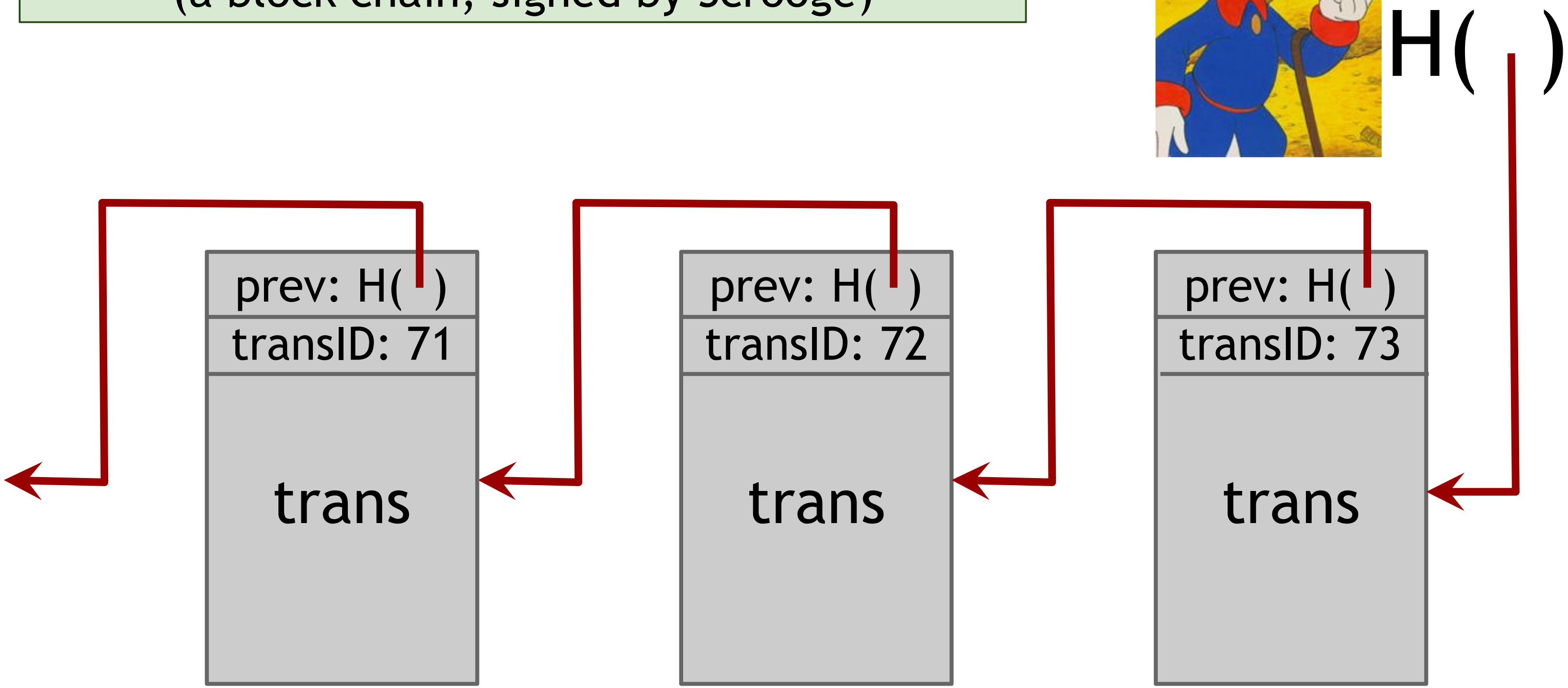


double-spending attack

the main design challenge in digital currency

ScroogeCoin

Scrooge publishes a history of all transactions
(a block chain, signed by Scrooge)



optimization: put multiple transactions in the same block

CreateCoins transaction creates new coins

transID: 73	type:CreateCoins	
coins created		
num	*value*	*recipient*
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

coinID 73(0)

coinID 73(1)

coinID 73(2)

Valid, because I said so.

PayCoins transaction consumes (and destroys) some coins, and creates new coins of the same total value

transID: 73 type:PayCoins

consumed coinIDs:
68(1), 42(0), 72(3)

coins created

num	*value*	*recipient*
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

signatures

Valid if:
- -- consumed coins valid,
- -- not already consumed,
- -- total value out = total value in, and
- -- signed by owners of all consumed coins

Immutable coins

Coins can’t be transferred, subdivided, or combined.

But: you can get the same effect by using transactions

to subdivide: create new trans

consume your coin

pay out two new coins to yourself

Don’t worry, I’m honest.

Crucial question:

Can we descroogify the currency, and operate without any central, trusted party?