Cryptography & Cryptocurrencies

Contents

Crypto background
- Hash functions
- Digital signatures
- Applications

Introduction to cryptocurrencies
- Basic digital cash

Cryptocurrencies

Cryptography is used to:

provide a mechanism for securely encoding the rules of a cryptocurrency system in the system itself.

prevent tampering and equivocation.

encode the rules for creation of new units of the currency into a mathematical protocol.

Cryptographic Hash Functions

Hash function:

- Takes any string as input
- Produces a fixed-size output
- Is efficiently computable (running time that is O(n))

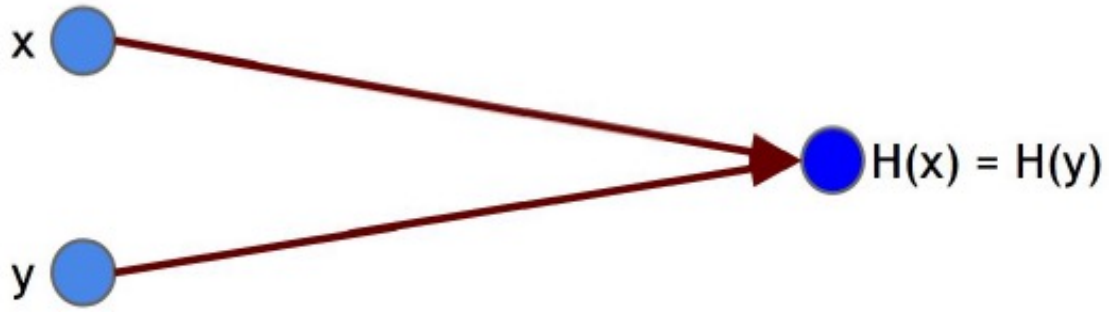
Properties of a cryptographically secure hash function:

- Collision-resistance
- Hiding
- Puzzle-friendly

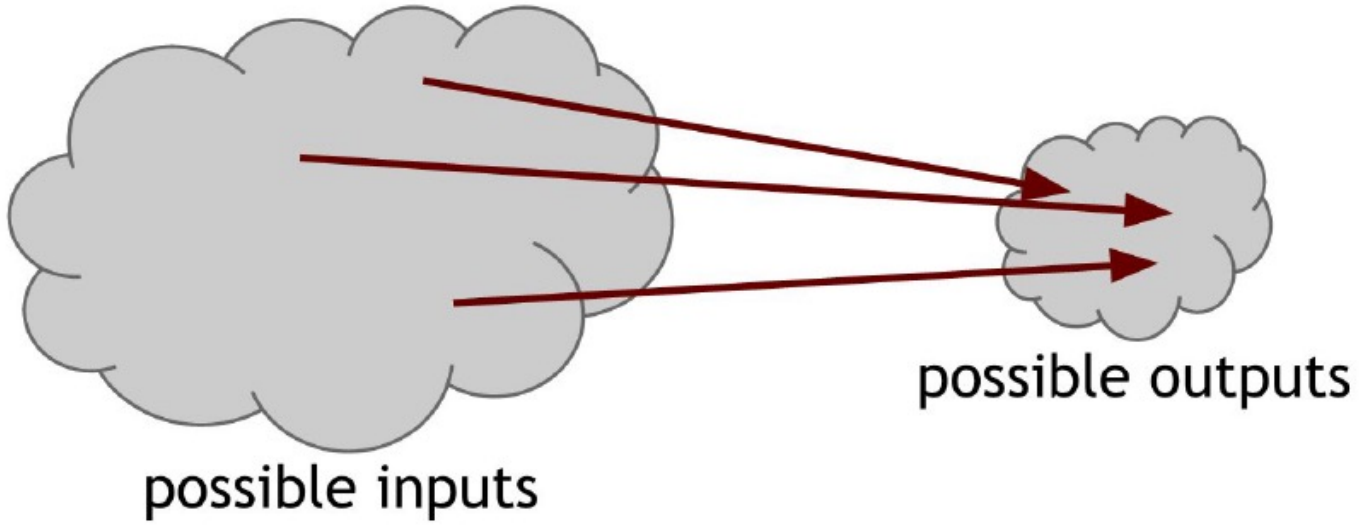
Property 1: Collision-resistance

It is infeasible to find two values, x and y , such that:

$x \neq y$, yet $H(x) = H(y)$.



…but collisions exist



Finding a collision

pick $2^{256} + 1$ distinct values and compute the hashes of each of them.

In fact, if we randomly choose just $2^{130} + 1$ inputs, it turns out there's a 99.8% chance that at least two of them are going to collide.

We can find a collision by only examining roughly the square root of the number of possible outputs.

…but

It takes a very, very long time to find a collision with this generic collision detection algorithm.

For 256-bit output compute the hash function

$2^{256} + 1$ times in the worst case

2^{128} times on average

If a computer calculates 10,000 hashes per second, it would take more than one octillion (10^{27}) years to calculate 2^{128} hashes!

If every computer ever made by humanity was computing since the beginning of the entire universe, up to now, the odds that they would have found a collision is still infinitesimally small

Non-generic collision detection

$H(x) = x \bmod 2^{256}$

- ✓ accepts inputs of any length
- ✓ returns a fixed sized output (256 bits)
- ✓ is efficiently computable

But this function also has an efficient method for finding a collision.

check the function here

Collision resistance

There are no hash functions *proven* to be collision-resistant.

We use hash functions that people have not yet succeeded on finding collisions.

MD5 is not one of them anymore!

Application: Hash as message digests

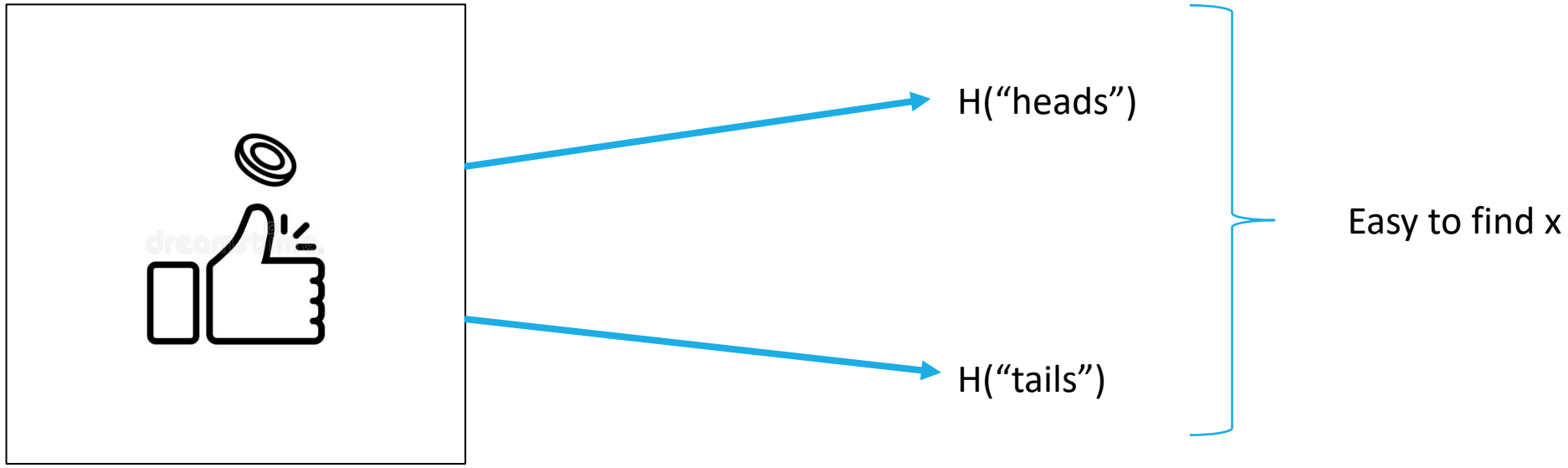
If we know H(x) = H(y), it is safe to assume that x =y

SecureBox: allows users to upload files and ensure their integrity when they download them.

Verify downloads on the internet, example

Property 2: Hiding

Given H(x), it is infeasible to find x



it needs to be the case that there’s no value of *x* which is particularly likely

Property 2: Hiding

x has to be chosen from a set that's *very spread out*

Solution: we can hide an input that's not spread out by concatenating it with another input that is spread out.

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: no particular value is chosen with more than negligible probability.

Application: Commitments

com := commit(msg, nonce) publish com

verify(com, msg, nonce) publish msg and nonce

- true if *com* == commit(*msg* , *nonce*)
- false otherwise

Security properties:

Hiding: Given com, infeasible to find msg

Binding: Infeasible to find msg != msg' such that commit(msg,nonce)==commit(msg',nonce')

Application: Commitments

Implementation:

commit(msg, nonce) := H(nonce ‖ msg) where nonce is a random 256-bit value

Hiding : Given H(*nonce* ‖ *msg)* , it is infeasible to find *msg*

Binding : It is infeasible to find two pairs *(msg, nonce)* and *(msg', nonce')* such that *msg* ≠ *msg'* and H(*nonce* ‖ *msg*) == H(*nonce'* ‖ *msg'*)

Property 3: Puzzle friendliness

For every possible n-bit 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 || x) = y

If someone wants to target the hash function to come out to some particular output value y , that if there's part of the input that is chosen in a suitably randomized way, it's very difficult to find another value that hits exactly that target.

Application: Search puzzle

Search puzzle. A search puzzle consists of

a hash function, H ,

a value, id (which we call the **puzzle-ID**), chosen from a high min-entropy distribution

and a target set Y

A solution to this puzzle is a value, x , such that

$H(id \| x) \in Y$.

Application: Search puzzle

If Y is the set of all n-bit strings the puzzle is trivial.

If Y has only 1 element the puzzle is maximally hard.

If a search puzzle is puzzle-friendly,

there's no solving strategy for this puzzle which is much better

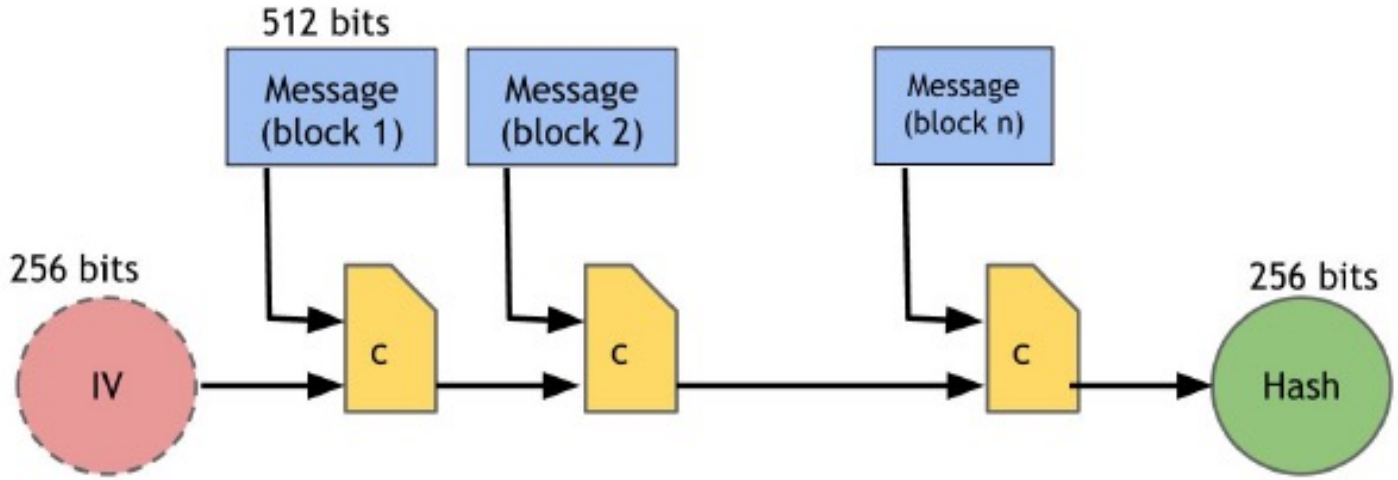
than just trying random values of x .

Important in mining!

SHA-256 hash function

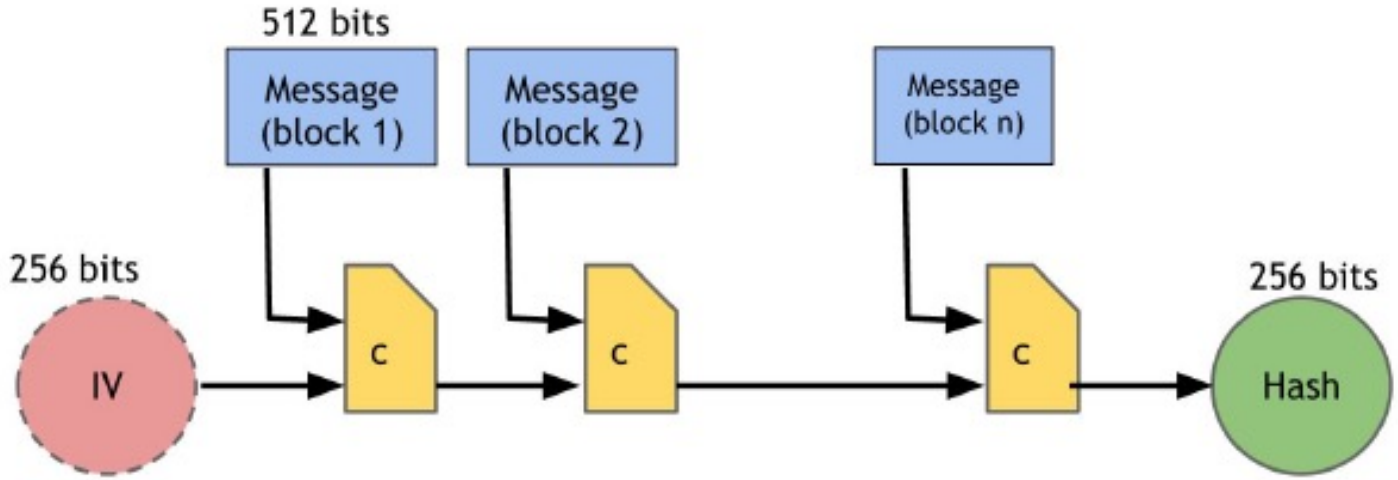
We require that our hash functions work on inputs of arbitrary length.

Merkle-Damgard transform: convert a hash function that works on fixed-length inputs into a hash function that works on arbitrary-length inputs



SHA-256 hash function

Theorem: if c is collision-resistant, then SHA-256 is collision-resistant



Hash Pointers and Data Structures

A hash pointer is:

a pointer to where some information is stored and

a cryptographic hash of the information.

A regular pointer gives you a way to retrieve the information

a hash pointer also gives you a way to verify that the information hasn’t changed.

Hash Pointers and Data Structures

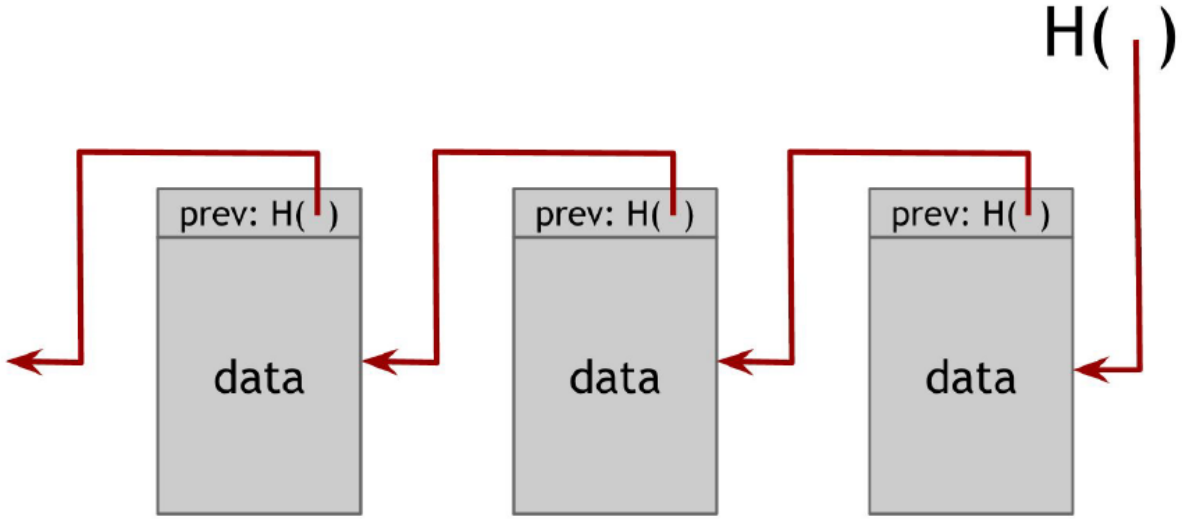
We can use hash pointers to build all kinds of data structures.

- linked list
- binary search trees

A block chain is a linked list that is built with hash pointers instead of pointers.

Merkle-Damgard

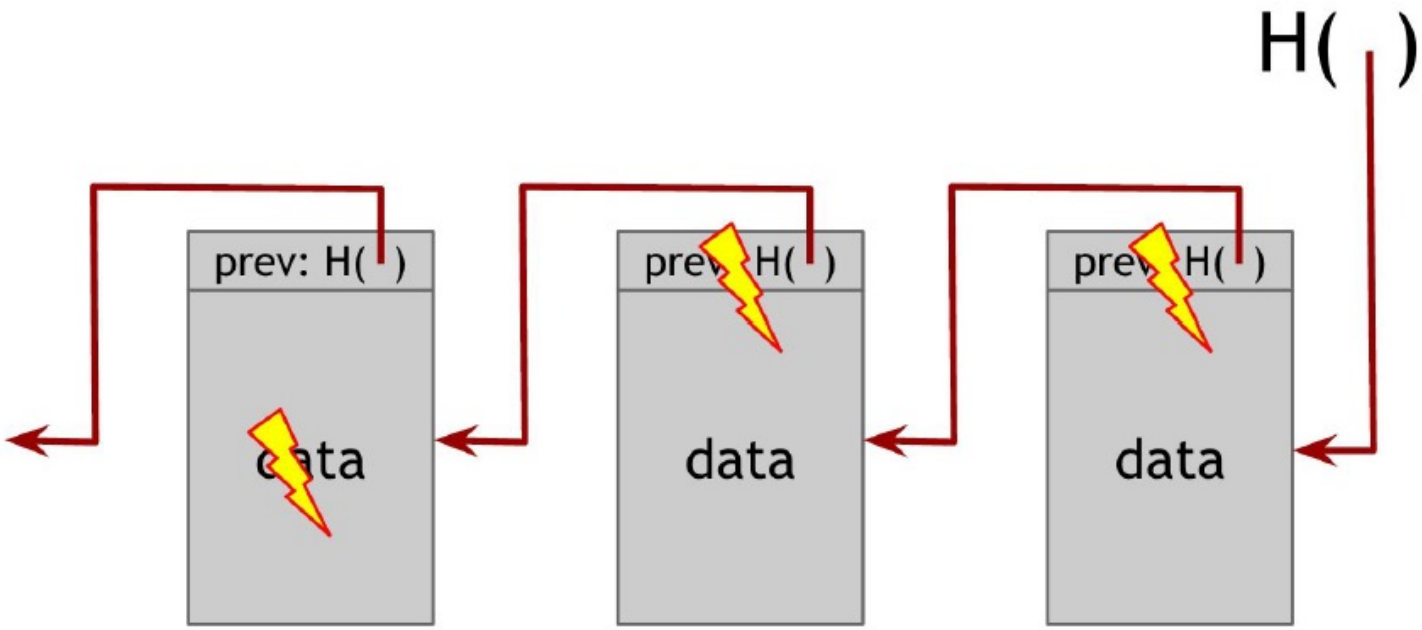
construction



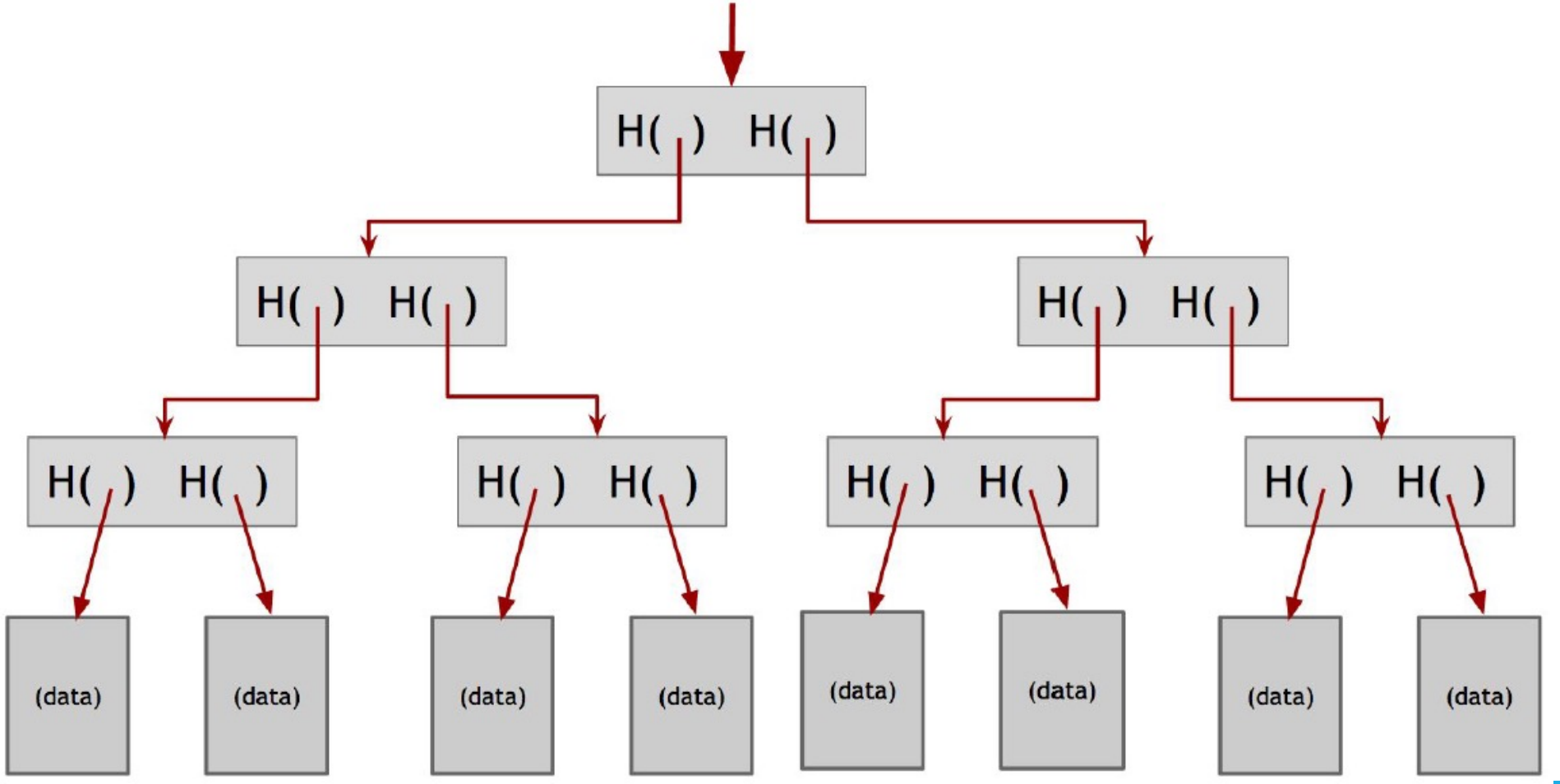
Hash Pointers and Data Structures

A use case for a block chain is a ***tamper-evident log.***

A log data structure that stores a bunch of data, and allows us to append data onto the end of the log.



Merkle trees

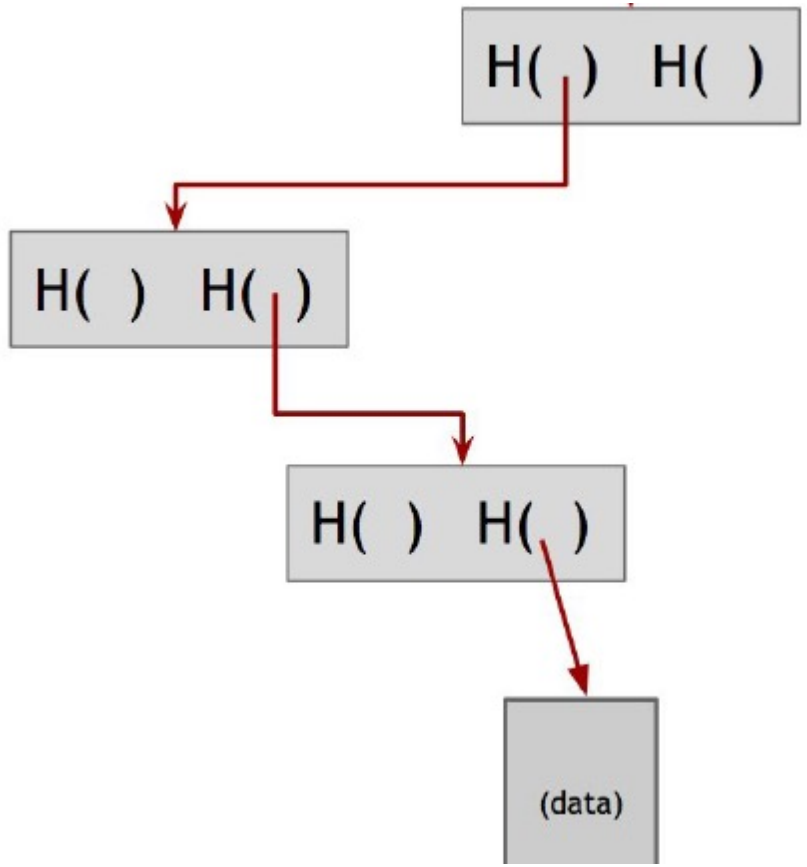


Merkle trees - Proof of Membership

Someone wants to prove that a certain data block is a member of the Merkle Tree.

Only about *log(n)* items need to be shown.

It takes about *log(n)* time for us to verify it.



Merkle trees - Proof of non-Membership

With a **sorted Merkle tree**, it becomes possible to verify non-membership in a logarithmic time and space.

Show a path to the item that's just before where the item in question would be.

Show the path to the item that is just after where it would be.

If these two items are consecutive in the tree, then this serves as a proof that the item in question is not included.

We can use hash pointers in any pointer-based data structure that has no cycles.

Digital Signatures

A digital signature is supposed to be the digital analog to a handwritten signature on paper.

Its properties:

1. Only you can make your signature, but anyone who sees it can verify that it's valid.
2. The signature is tied to a particular document so that the signature cannot be used to indicate your agreement or endorsement of a different document.

API for Digital Signatures

(sk, pk) := generateKeys(keysize)

sig := sign(sk , message)

isValid := verify(pk , message , sig)

Properties of Digital Signatures

Valid signatures must verify

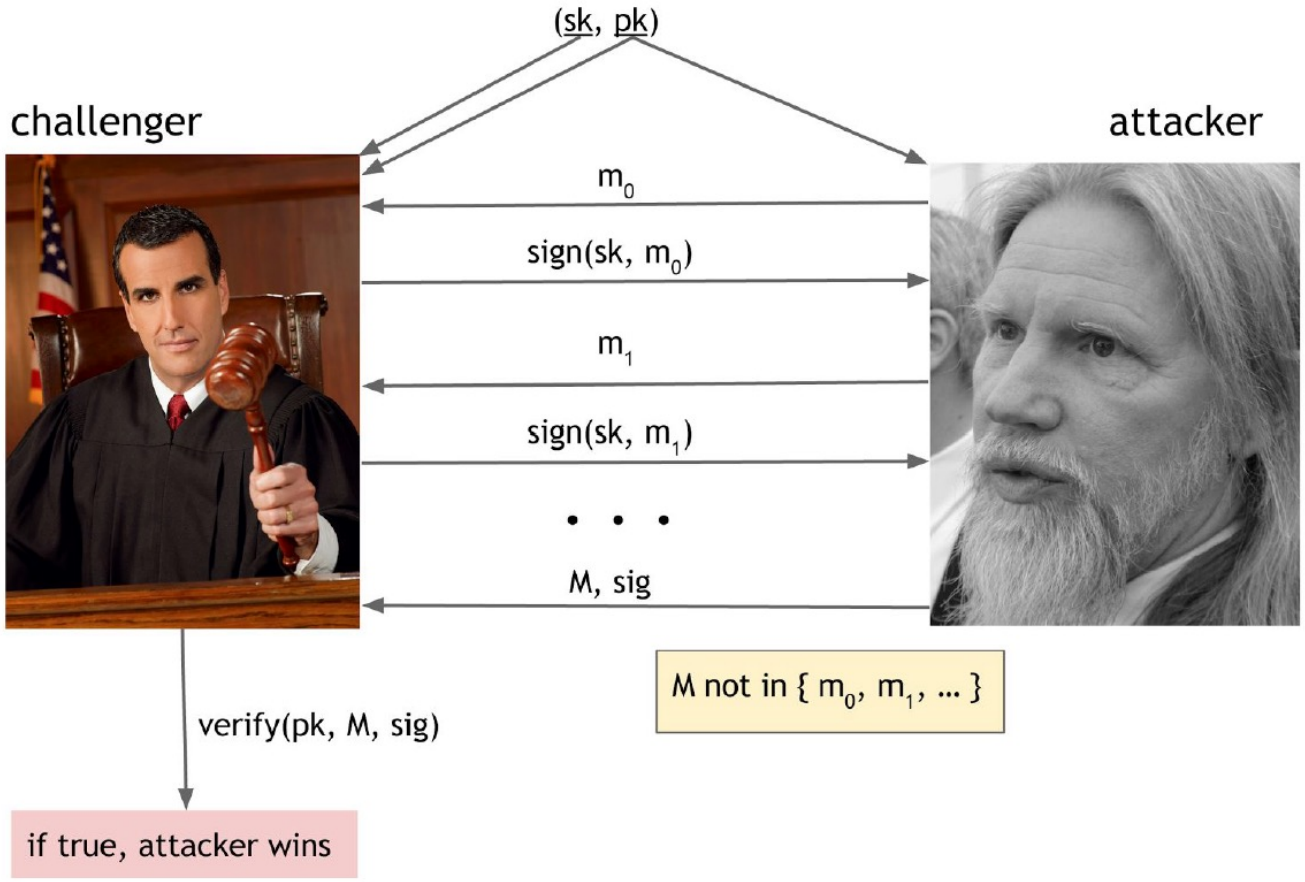
verify (*pk* , *message* , **sign** (*sk* , *message*)) == **true**

Signatures are ***existentially unforgeable,*** it's computationally infeasible to forge signatures. An adversary who:

- knows your public key
- gets to see your signatures on some other messages

  can't forge your signature on a new message

The unforgeability game



Practical Concerns

Algorithms are randomized

we need good source of randomness

Limit on the message size

use Hash(message) rather than message

A trick: signing the hash pointer the signature covers that whole structure.

Digital Signatures

Bitcoin uses a particular digital signature scheme that's called the Elliptic Curve Digital Signature Algorithm (**ECDSA**).

A U.S. government standard.

Bitcoin uses ECDSA over the standard elliptic curve "secp256k1" which is estimated to provide 128 bits of security.

> it is as difficult to break this algorithm as performing 2^{128} symmetric-key cryptographic operations

Public Keys as Identities

If you see a message with a signature that verifies correctly under a public key, pk , then you can think of this as pk is saying the message.

In order for someone to speak for the identity pk , they must know the corresponding 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
- sk lets you “speak for” this new identity

You control the identity, because only you know the sk.

Decentralized Identity Management

Anybody can make a new identity at any time,

you can make as many as you want!

There is no central point of coordination.

These identities are called “addresses” in Bitcoin.

A Simple Cryptocurrency – The GoofyCoin

Goofy can create new coins

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

A Simple Cryptocurrency – The GoofyCoin

A coin's owner can spend it.

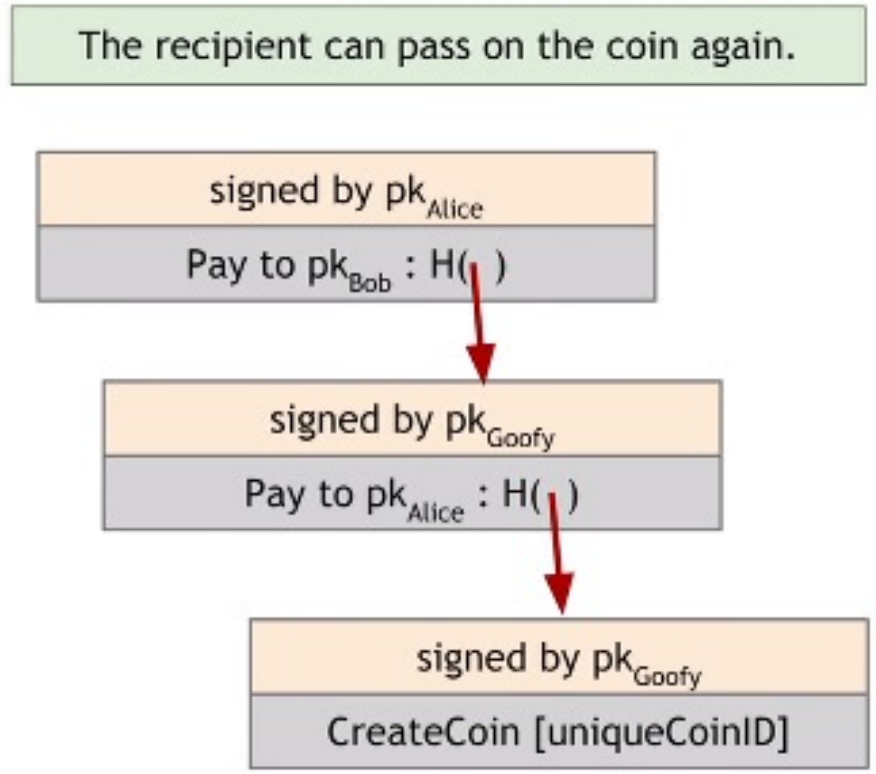
signed by pk_{Goofy}

Pay to pk_{Alice} : H( )

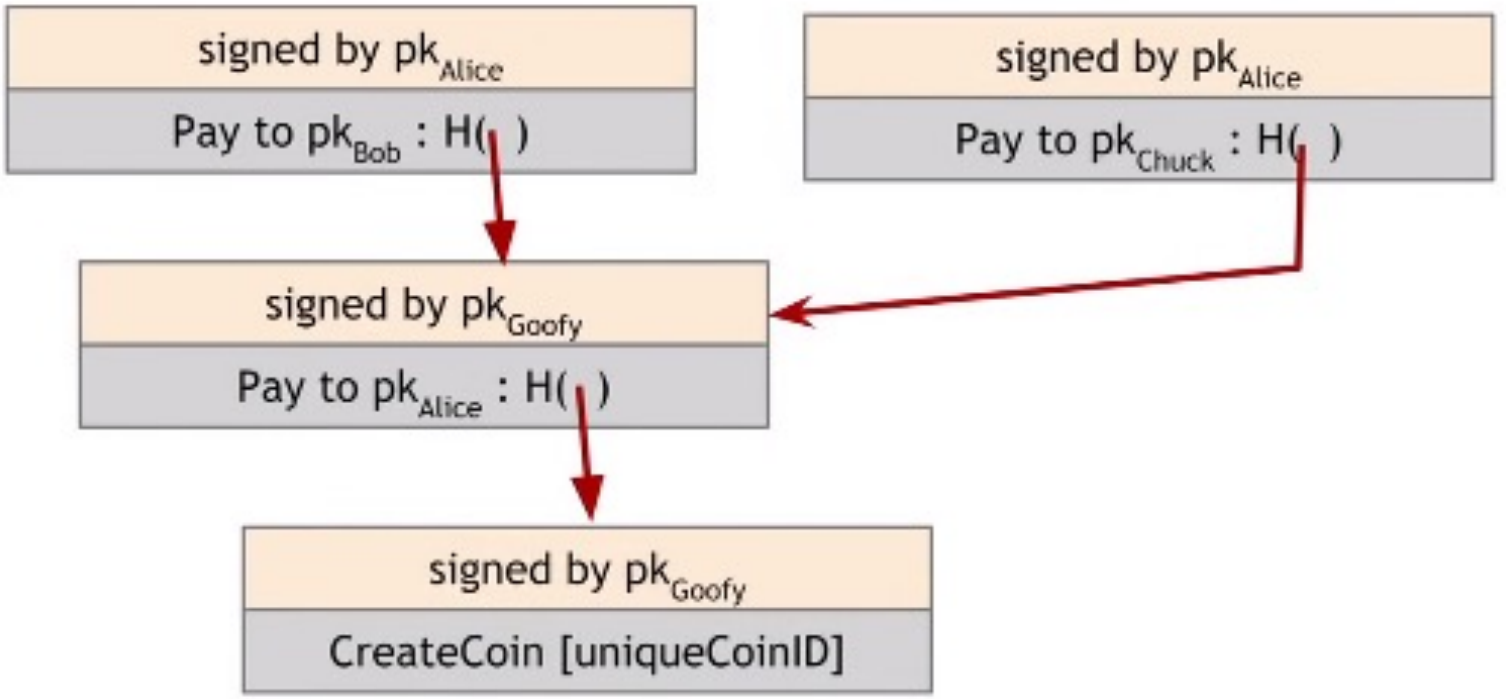
signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

A Simple Cryptocurrency – The GoofyCoin

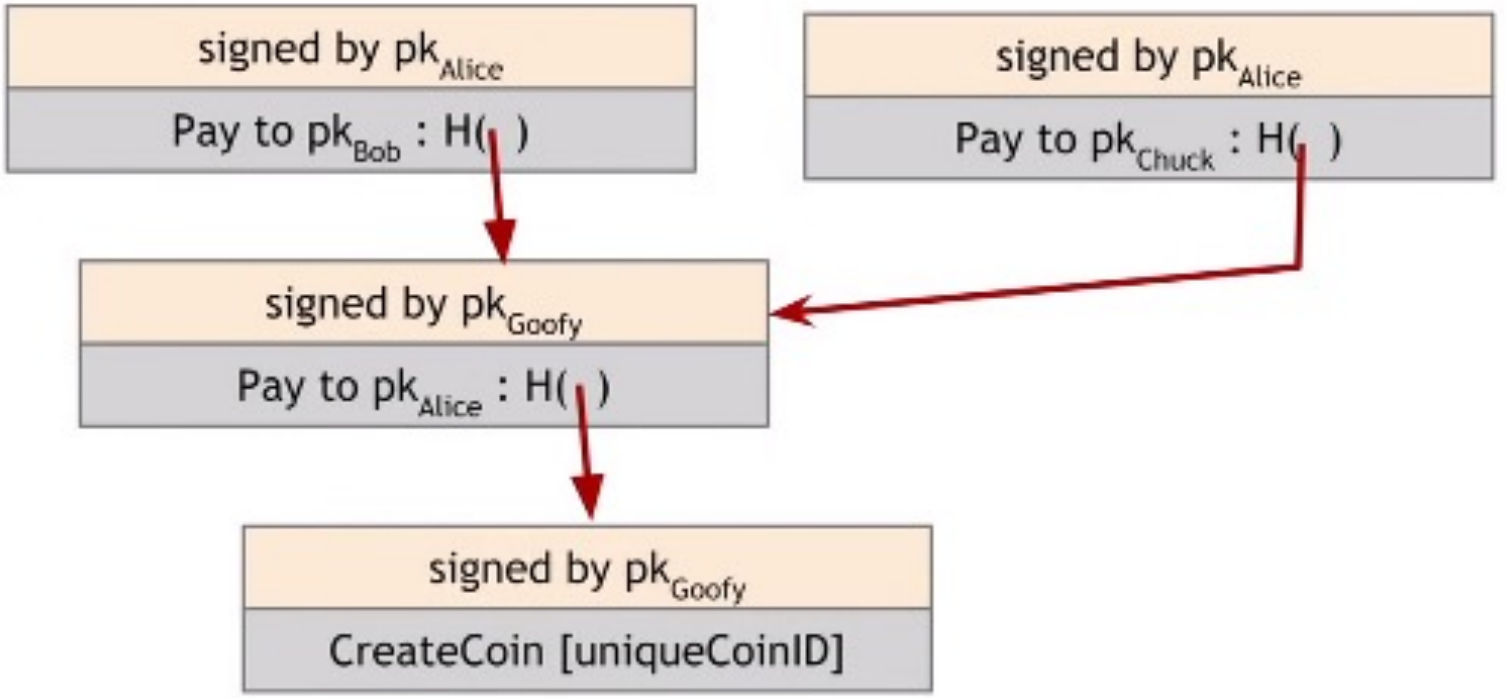


A Simple Cryptocurrency – The GoofyCoin

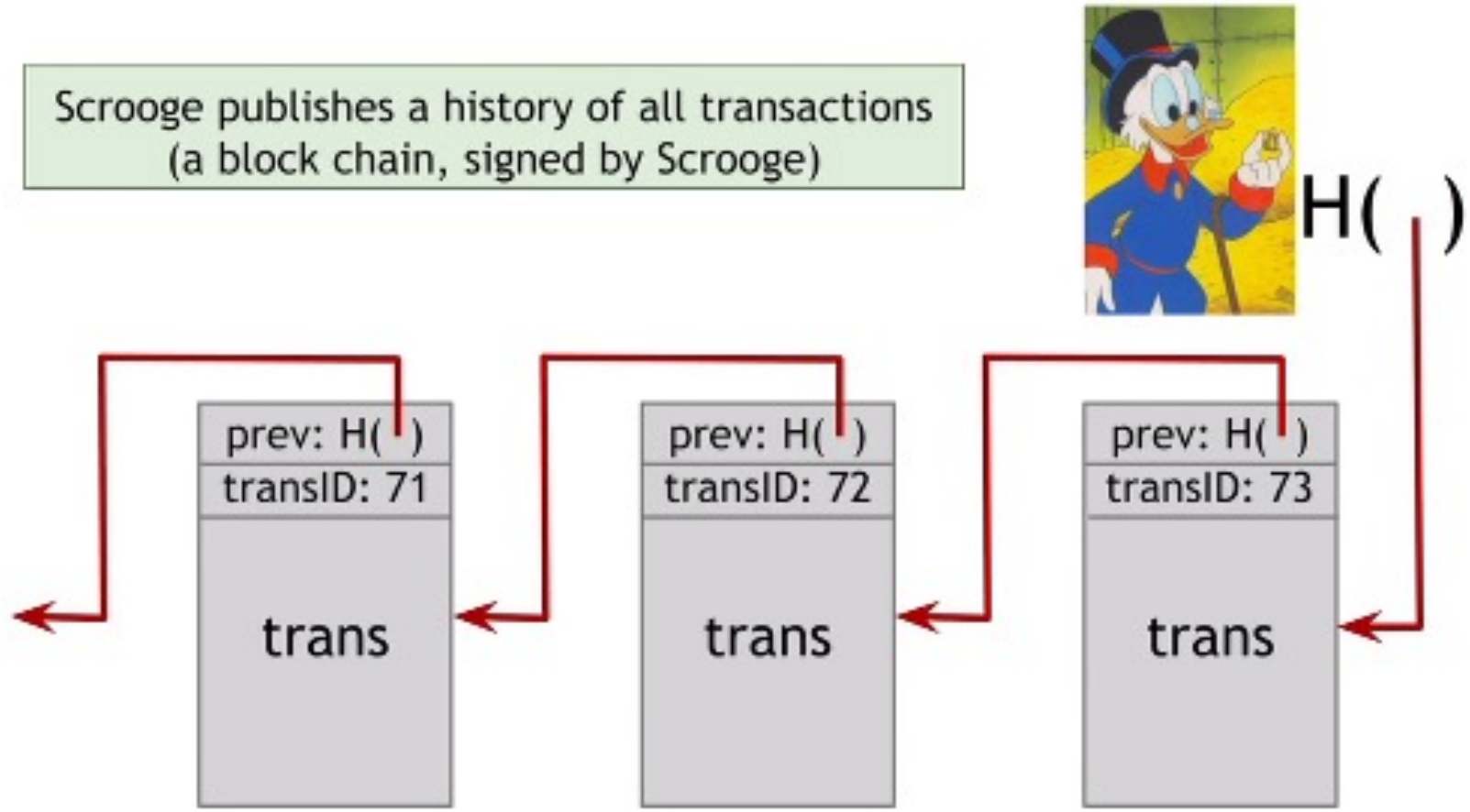


A Simple Cryptocurrency – The GoofyCoin

Double Spending Attack!

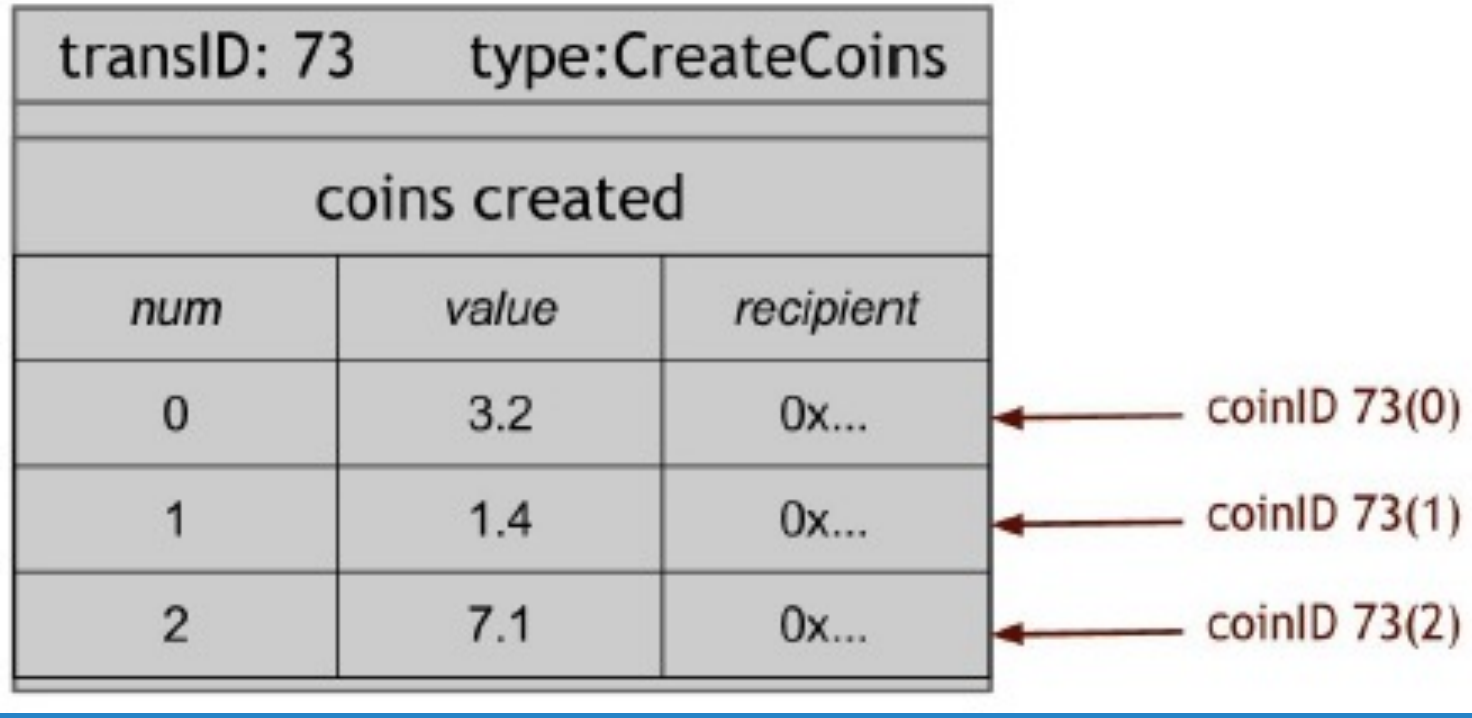


A Simple Cryptocurrency – The ScroogeCoin



A Simple Cryptocurrency – The ScroogeCoin

CreateCoins transaction creates new coins



A Simple Cryptocurrency – The ScroogeCoin

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

A Simple Cryptocurrency – The ScroogeCoin

Immutable coins.

Coins cannot be transferred, subdivided or combined.

Same effect by:

- Create new transactions
- Consume your coin
- Pay out two new coins to yourself

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		

A Simple Cryptocurrency – The ScroogeCoin

Crucial question:

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

A Simple Cryptocurrency – The ScroogeCoin

The problems to be solved:

Users must agree upon a single published block chain as the history of which transactions have happened.

They must all agree on which transactions are valid, and which transactions have actually occurred.

They also need to be able to assign IDs to things in a decentralized way.

The minting of new coins needs to be controlled in a decentralized way.

References

CS 4593/6463 – Bitcoins and Cryptocurrencies, Prof. Murtuza Jadliwala, University of Texas, San Antonio

Note: most of the slides used in this course are derived from those available for the book "Bitcoins and Cryptocurrencies Technologies – A Comprehensive Introduction", Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller & Steven Goldfeder, 2016, Princeton University Press.