Intro to Cryptography and Cryptocurrencies

- Cryptographic Hash Functions
- Hash Pointers and Data Structures
  - Block Chains
  - Merkle Trees
- Digital Signatures
- Public Keys and Identities
- Let's design us some Digital Cash!

Intro to Cryptography and Cryptocurrencies

- Cryptographic Hash Functions
- Hash Pointers and Data Structures
  - Block Chains
  - Merkle Trees
- Digital Signatures
- Public Keys and Identities
- Let's design us some Digital Cash!

Cryptographic Hash Function

Hash Function: Mathematical Function with following 3 properties:

- The input can be any string of any size.
- It produces a fixed-size output. (say, 256-bit long)
- Is efficiently computable. (say, $O(n)$ for n-bit string)

Such general hash function can be used to build hash tables, but they are not of much use in cryptocurrencies. What we need are **cryptographic** hash functions.

Cryptographic Hash Functions

A Hash Function is **cryptographically secure** if it satisfies the following 3 **security properties**:

- Property 1: Collision Resistance
- Property 2: Hiding
- Property 3: "Puzzle Friendliness"

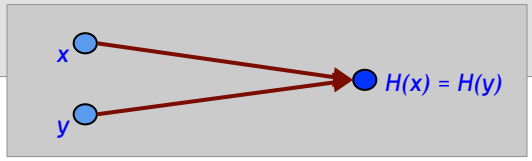
Cryptographic Hash Functions

A Hash Function is **cryptographically secure** if it satisfies the following 3 **security properties**:

- Property 1: Collision Resistance
- Property 2: Hiding
- Property 3: "Puzzle Friendliness"

Crypto Hash Property 1: Collision Resistance

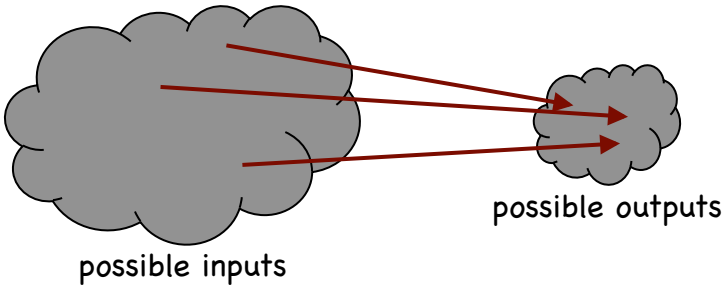
Collision Resistance: A hash function *H* is said to be **collision resistant** if it is infeasible to find two values, *x* and *y*, such that *x != y*, yet *H(x) = H(y)*.



In other words: If we have *x* and *H(x)*, we can "never" find an *y* with a matching *H(y)*.

Collision Resistance ?!

Collisions do exist …



… but can anyone find them?

Collision Resistance ?! (2)

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

Collision Resistance ?! (3)

Q: Is there a faster way to find collisions?

A: For some possible *H*'s, yes.
For others, we don't know of one.

No *H* has been proven collision-free.

Collision Resistance

Application: Hash as a **Message Digest**

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

Example: To recognize a file that we saw before, just remember its hash.

This works because hash is small.

Cryptographic Hash Functions

A Hash Function is **cryptographically secure** if it satisfies the following 3 **security properties**:

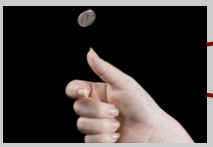
- Property 1: Collision Resistance
- Property 2: Hiding
- Property 3: "Puzzle Friendliness"

Crypto Hash Property 2: Hiding

We want something like this:

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

Example:



The value for x is easy to find because the distribution is not "spread out" (only two values!)

Crypto Hash Property 2: Hiding (cont)

Hiding: A hash function *H* is said to be **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*.

"*r || x*" stands for "*r* concatenated with *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 of Hiding Property: Commitment

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

Commit to a value, reveal it later.

Application of Hiding Property: Commitment

Commitment Scheme consists of two algorithms:

- com := commit(msg,key) takes message and secret key, and returns commitment
- verify(com,msg,key) returns true if com = commit(msg,key) and false otherwise.

We require two security properties:

- Hiding: Given com, it is infeasible to find msg.
- Binding: It is infeasible to find two pairs *(msg,key)* and *(msg′,key′)* s.t. *msg != msg′* and *commit(msg,key) == commit (msg′,key′)*.

Implementation of Commitment

- commit(msg,key) := H(key || msg)
- verify(com,msg,key) := (H(key || msg) == com)

Proof of security properties:

- Hiding: Given H(key || msg), it is infeasible to find msg.
- Binding: It is infeasible to find *msg != msg′* such that *H(key || msg) == H(key || msg′)*

Cryptographic Hash Functions

A Hash Function is **cryptographically secure** if it satisfies the following 3 **security properties**:

- Property 1: Collision Resistance
- Property 2: Hiding
- Property 3: "Puzzle Friendliness"

Crypto Hash Property 3: "Puzzle Friendliness"

Puzzle Friendliness: A hash function H is said to be **puzzle friendly** if 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 $H(k \,||\, x) = y$, in time significantly less than 2^n.

If a hash function is puzzle friendly, then there is no solving strategy for this type of puzzle that is much better than trying random values of x.

Bitcoin mining is just such a computational puzzle.

Intro to Cryptography and Cryptocurrencies

- Cryptographic Hash Functions
- Hash Pointers and Data Structures
  - Block Chains
  - Merkle Trees
- Digital Signatures
- Public Keys and Identities
- Let's design us some Digital Cash!

Hash Pointers

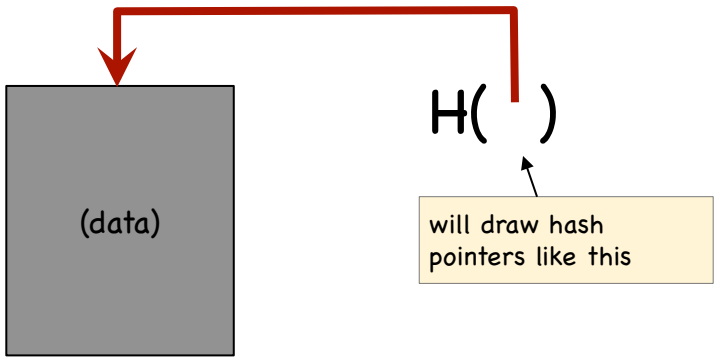
Hash Pointer is:

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

Given a Hash Pointer, we can

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

Hash Pointers

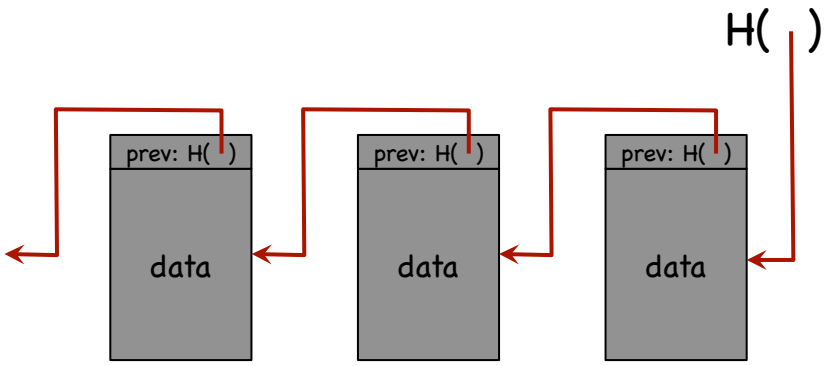


Hash Pointers

Key Idea:

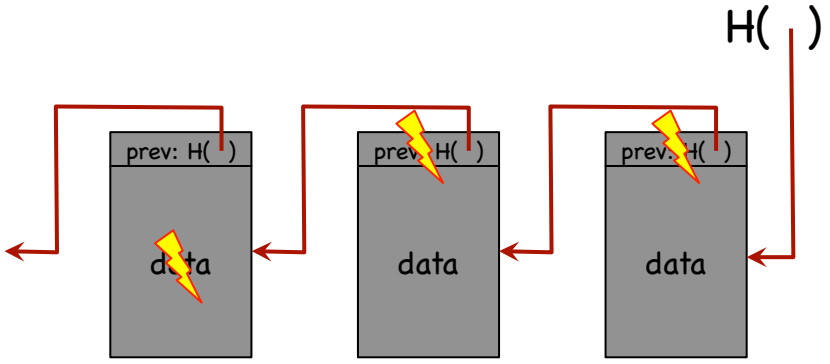
Build data structures with hash pointers.

Linked List with Hash Pointers: "Block Chain"



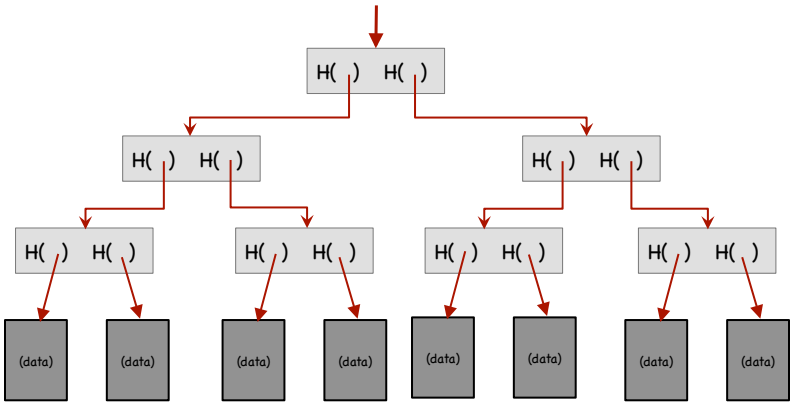
use case: tamper-evident log

Detecting Tampering in Block Chains



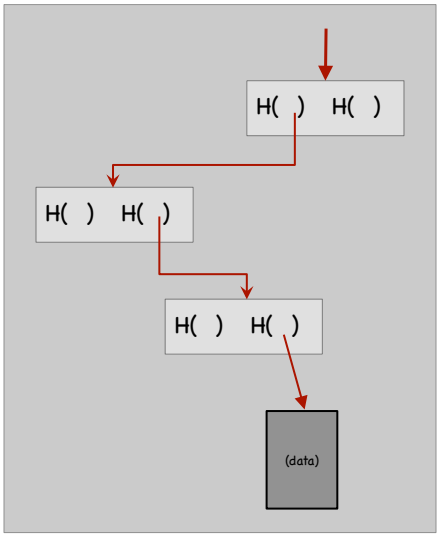
use case: tamper-evident log

Binary Trees with Hash Pointers: "Merkle Tree"



Used in file systems (IPFS, Btrfs, ZFS), BitTorrent, Apache Wave, Git, various backup systems, Bitcoin, Ethereum, and database systems.

Proving Membership in a Merkle Tree



Single branches of the tree can be downloaded at a time.

To prove that a data block is included in the tree only requires showing blocks in the path from that data block to the root.

Benefits 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)

Beyond Merkle Trees ..

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

Intro to Cryptography and Cryptocurrencies

- Cryptographic Hash Functions
- Hash Pointers and Data Structures
  - Block Chains
  - Merkle Trees
- **Digital Signatures**
- Public Keys and Identities
- Let's design us some Digital Cash!

Digital Signatures

Q: What do we want from signatures?

Only you can sign, but anyone can verify.

Signature is tied to a particular document, i.e., cannot be cut-and-pasted to another document.

Digital Signature Scheme

Digital Signature Scheme consists of 3 algorithms:

- (sk,pk) := generateKeys(keysize) generates a key pair
  - *sk* is secret key, used to sign messages
  - *pk* is public verification key, given to anybody
- sig := sign(sk, msg) outputs signature for *msg* with key *sk*.
- verify(pk,msg,sig) returns true if signature is valid and false otherwise.

Requirements for Digital Signature Scheme

Valid signatures must verify!

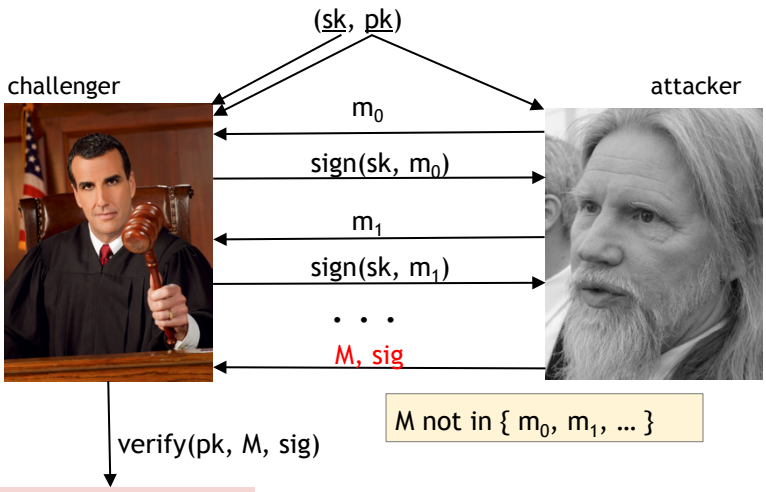
verify(pk, msg, sign(sk, msg)) == true

Signatures must be unforgeable!

An adversary who
- knows *pk*
- has seen signatures on messages of her choice

cannot produce a verifiable signature on a new message.

The "Unforgeability Game"



Digital Signatures in Practice

Key generation algorithms must be randomized.

.. need good source of randomness

Sign and verify are expensive operations for large messages.

Fix: use *H(msg)* rather than *msg*.

Check this out:

Signing a hash pointer "covers" the whole data structure!

Intro to Cryptography and Cryptocurrencies

- Cryptographic Hash Functions
- Hash Pointers and Data Structures
  - Block Chains
  - Merkle Trees
- Digital Signatures
- Public Keys and Identities
- Let's design us some Digital Cash!

Signatures, Public Keys, and Identities

If you see a signature *sig* such that

verify(pk, msg, sig)==true,

think of it as

pk says, "*[msg]*".

Why?

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

How to Create 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

By creating a key-pair,
anybody can make a new identity at any time.

Make as many as you want!

No central point of coordination.

These identities are called addresses in Bitcoin.

Identities and Privacy

Addresses are not directly connected to real-world identity.

But observer can link together an address' activity over time, and make inferences about real identity.

We will talk later about privacy in Bitcoin . . .

Intro to Cryptography and Cryptocurrencies

- Cryptographic Hash Functions
- Hash Pointers and Data Structures
  - Block Chains
  - Merkle Trees
- Digital Signatures
- Public Keys and Identities
- Let's design us some Digital Cash!

Vanilla Cryptocurrency Ver. 0.0

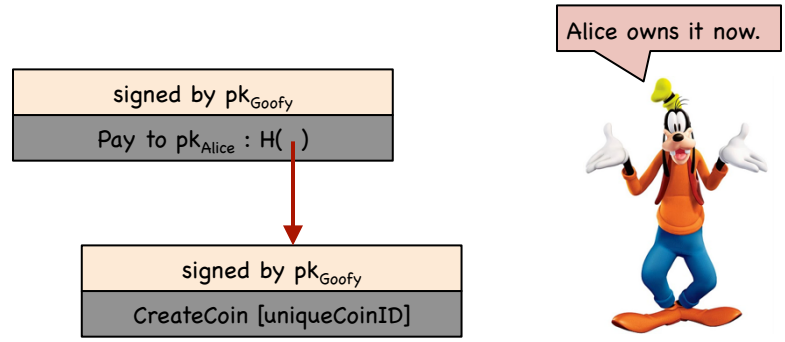
GoofyCoin

Goofy can create new Coins

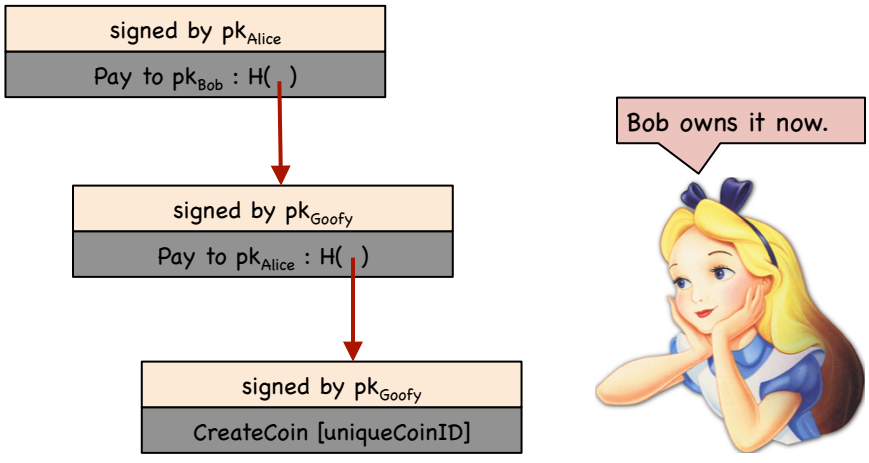
signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

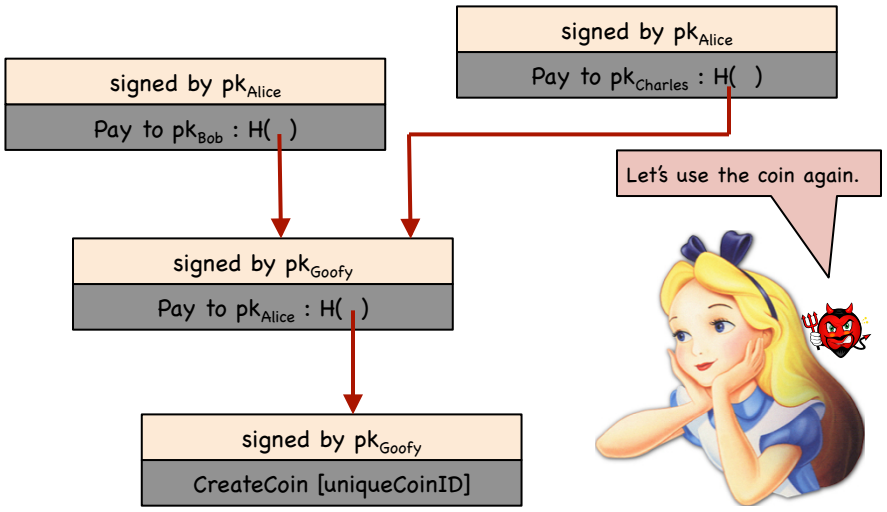
Goofy can spend the Coins



The Recipient can pass on the Coin again



The Recipient can also double-spend the coin!



Double-Spending

Main design challenge in all digital currencies

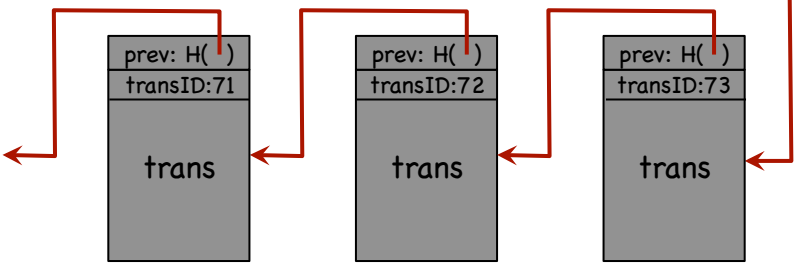
Vanilla Cryptocurrency Ver. 1.0

ScroogeCoin

Record Transactions in central Block Chain

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

H( )



optimization: put multiple transactions in the same block

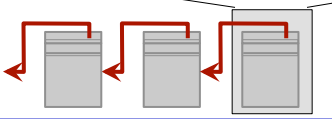
Creating new Coins in ScroogeCoin

`CreateCoins` transaction creates new coins

Valid, because I said so!

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)



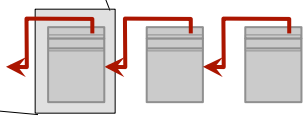
Pay Coins in ScroogeCoin

`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

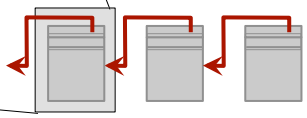


Coins in ScroogeCoin are Immutable

`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		

Coins are **Immutable**:
They cannot be
- transferred,
- subdivided, or
- combined



How to deal with Immutable Coins

Coins are **Immutable**:
They cannot be
- transferred,
- subdivided, or
- combined

But: You can get the same effect by using transactions.

Example - Subdivide Coin: are **Immutable**:

1. create new transaction
2. consume (destroy) your coin
3. pay out two new coins to yourself

The Problem with ScroogeCoin

Crucial question:

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