Bitcoin & Blockchains, Cryptography I

Alexander Schönhuth

UNIVERSITÄT BIELEFELD
Faculty of Technology

Bielefeld University
May 4, 2022

Bitcoin
&
Blockchains

Hash Functions
–
Introduction

Hash Functions
–
Central
Properties

The Merkle-
Damgard
Transform

Bitcoin
&
Blockchains

Hash Functions
–
Introduction

Hash Functions
–
Central
Properties

The Merkle-
Damgard
Transform

Bitcoin: Things to Consider

ELECTRONIC CASH: PRESERVING VALUE

Issue

- *Question:* How to preserve the value of the cash?
- Generation of new electronic cash necessary for particular purposes

Solution – Mining

- Adopt idea that renders gold or diamonds valuable
  ☞ make electronic cash *sparse*
- New cash relates to computational puzzles
  - Solving puzzles = ”mining”
  - Requires time / electricity
  - Requires computational hardware resources

Electronic Cash: Ledger

Issue

- How to keep track of transactions?
- Requires ledger (= account book) to be accessed by anyone having permissions
- Data structure that supports such ledger?

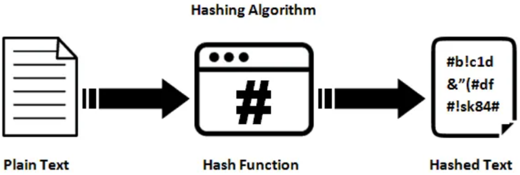
Solution: Blockchain

- Enable timestamping to establish order of transactions
- Preserve integrity of earlier transactions, so fraud impossible ☞ make use of electronic signatures

The Bitcoin Blockchain

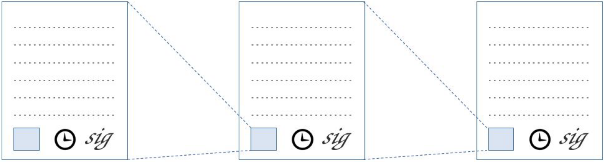
HASHING

- DEFINITION: A *hash function* maps data of arbitrary size to fixed-size output values, called hash (values)
- A hash function should
  - be (very) fast to compute
  - minimize collisions, i.e. cases where two different inputs get mapped to identical hashes



Hashing Algorithm

Blockchain: Ledger for Electronic Cash

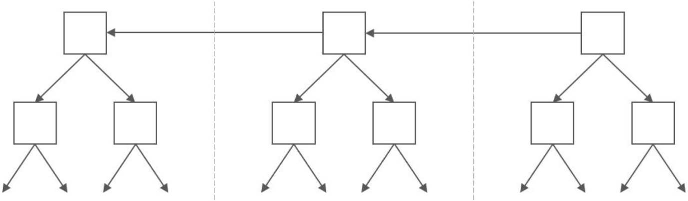


Linked Lists of Documents

From `https://bitcoinbook.cs.princeton.edu`

- Documents contain
  - Transactions, like "Alice transfers 10 coins to Bob"
  - (Hash) pointer to previous document
  - Timestamp
  - Electronic signature
- Preserves integrity of earlier transactions, so fraud impossible

Blocks of Documents



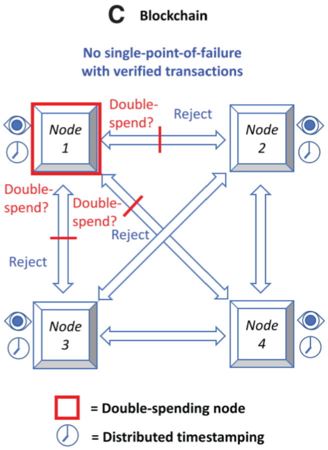
Blocks of Transaction Documents

From `https://bitcoinbook.cs.princeton.edu`

- Documents from various users are collected into blocks, receiving the same timestamp (separated by dotted lines)
- Blocks are structured using hash pointers (arrows)
- Enables linking large, mixed blocks of transactions

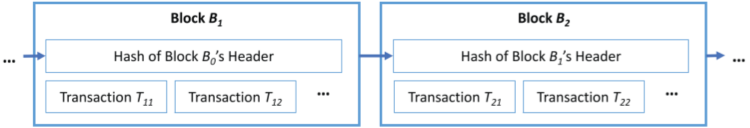
The Bitcoin Blockchain

- Nodes = bitcoin users/owners
- Every node maintains copy of the entire blockchain
- ☞ every node "is the bank"
- So, every node can verify every transaction
  ☞ "distributed verification"
- Approving / rejecting transactions: distributed timestamping mechanism



From Kuo et al., 2018

BITCOIN - A SIMPLIFIED BLOCKCHAIN EXAMPLE



From Kuo et al., 2018

Each block in the bitcoin blockchain contains (among others):

- Transactions
- The hash of the previous block (here 256 bits): if a transaction in block B1 is changed $\rightarrow$ the hash value stored in B2 does no longer match the hash of B1

Deterministic order of blocks

- Each block serves as a timestamp of the enclosed transactions
- Prevents double spending thanks to linear (chain like) structure

Block Creation I

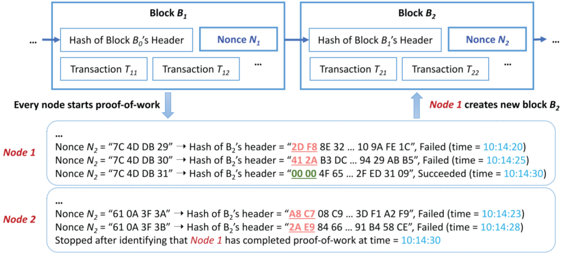
- *Issue:* We need to determine someone (i.e. one node) to generate a new block
- Optimally, that ”someone” should be picked randomly
- However, running random number generation across entire network impossible / insecure
- *Solution:* Principle of ”Proof of Work”

Block Creation II

Proof of Work

- *Nonce:* Additional data, added to and hashed with the block
- *Proof of work:* Determine nonce such that hashing nonce + block yields hash value below certain threshold
- *Mining:* Each node composes a block of transactions and searches for a nonce
- The block of the first node that determines such a suitable nonce is selected as the next block
- Once the block is verified, the successful miner receives a new coin as reward

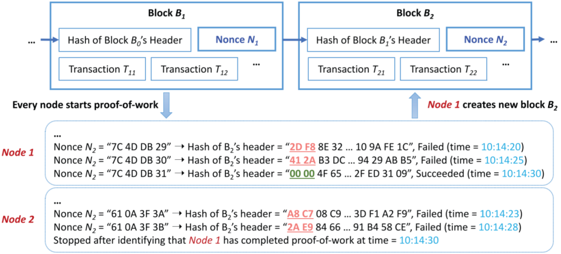
Block creation: Proof of work



From Kuo et al., 2018

- The winning node adds his/her block to the chain
- The new block is broadcast to the whole network
- Each node verifies the block; if successful
  - Block is added to chain
  - Creator receives "mining" reward

BLOCK CREATION: PROOF OF WORK



From Kuo et al., 2018

Important:

- Creating blocks (mining) is difficult
- Verifying is easy

Bitcoin
&
Blockchains

Hash Functions
–
Introduction

Hash Functions
–
Central
Properties

The Merkle-
Damgard
Transform

UNIVERSITÄT
BIELEFELD

INTRODUCTION

Traditional Currencies: Control

- Central banks control money supply
- Physical currencies have anti-counterfeiting features
- Law enforcement stops people from breaking rules

Decentralized Online Currencies: Evildoing

- Prevent malicious users from taking over
- Prevent individual users from counterfeiting / double spending
- Prevent loss of value

Cryptographic principles can warrant this

...

with great probability

Cryptographic Techniques

- Hash Functions
- Hash Pointers and Data Structures
- Digital Signatures
- Public Keys as Identities

CRYTOGRAPHIC HASH FUNCTIONS

- Collision Resistance and Message Digests
- Hiding and Commitments
- Puzzle Friendliness and Search Puzzles
- SHA-256 and Merkle-Damgard Transform

HASH FUNCTIONS

- A hash function takes a *hash-key* x as input and maps it to a bucket number.
- The bucket number is a an integer in the range from 0 to $B-1$, where B is the number of buckets. Often B is a prime.
- Here in the following, $B = 2^{256}$, reflecting having numbers coded as 256-bit strings
- *Simple Example*: Hash-keys are positive integers.

$$h(x) \equiv x \mod B \tag{1}$$

  which is the remainder of x when dividing it by B. Often, B is a prime.
- If $B = 2^{256}$, that is $h(x) \equiv x \mod 2^{256}$, hashing amounts to keeping the last 256 bits of arbitrarily sized input.

Hash Functions

- If hash-keys are not integers, they are often converted to integers.
- Example: if hash-keys are strings, one can map each character to its ASCII code, and sum them up, before dividing them by B.
- If hash-keys have several components (such as arrays), convert each component to integer, and sum them up.
- Let $h(x) \equiv x \mod 5$. *Example:*

$$h("AB") = h(ord('A') + ord('B')) = h(65 + 66) = h(131) = 1$$

CRYPTOGRAPHIC HASH FUNCTIONS

General Properties Assumed Here

- *Input:* string of arbitrary size
- *Output:* Fixed size, commonly 256 bits
- All hash functions are efficiently computable

Key Properties

- Collision-resistance
- Hiding
- Puzzle-Friendliness

Blockchains
–
Motivation

Hash Functions
–
Introduction

Hash Functions
–
Central
Properties

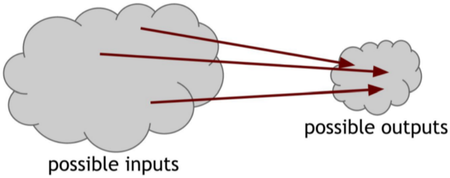
The Merkle-
Damgard
Transform

Collistion Resistance

Collision Resistance: Introduction

General Purpose

- *Input space:* too large, so difficult to grasp computationally
- *Output space:* small and sufficiently structured to serve as a computational foundation
- *Drawback:* Unavoidably, two different inputs can be mapped to the same output ☞ *Collision!*
- *Example:* For a 256-bit hash function, $2^{256} + 1$ different inputs guarantee a potential collision



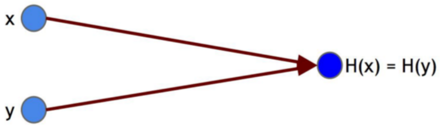
Hash function: output space is smaller than input space

From `https://bitcoinbook.cs.princeton.edu`

COLLISION RESISTANCE: DEFINITION

A hash function H is said to be **collision resistant**, if it is *computationally infeasible* to find

$$x \neq y \quad \text{such that} \quad H(x) = H(y)$$



Hash collision. Different x and y are hashed to same value

From `https://bitcoinbook.cs.princeton.edu`

Collision Resistance: Computational Infeasibility

- If hash function is sufficiently random, finding collisions amounts to trying inputs – and there are too many
- If not well defined, however, finding collisions can bea easy
- *For example:* $H(x) \equiv x\ mod\ 2^{256}$, returning the last 256 bits of the input, is not collision resistant
- *Birthday paradox:* For an output space of size 2^{256}, already 2^{130} random inputs yield a collision with probability 99.8%.
- God thank that computing 2^{128} hash values takes 2^{27} years

Good hash function design is key

Collision Resistance: Summary

- In the following, we assume that our hash functions are collision resistant
- Still, this means that collisions are theoretically possible
- So, what is the real life effect of this assumption?
- *Answer:* No collisions have been found so far for the hash functions in use here

Application: Message Digests

- When working with a collision resistant hash function H, one can assume that $H(x) \neq H(y)$ for $x \neq y$

Message Digest

- *Problem:* Alice uploads a huge file.
  - She would like to ensure the identity of the file when downloading it later
  - Keeping a copy for comparing it is no option
- *Solution:* Hash the file using a collision resistant hash function
  - Before uploading it
  - After downloading it

If the two hashes agree, files are identical!

Hiding

HIDING

DEFINITION [HIDING PROPERTY – FIRST TRY]:
A hash function H has the *hiding property* if, when given $y = H(x)$, there is no feasible way to determine x.

Issue – Thought Experiment

- Flip a coin and hash the outcome, ″heads″ or ″tails″
- An adversary can determine the input by hashing both ″heads″ and ″tails″ and comparing with the hashed outcome
- ☞ It is easy to determine the input
- *Issue:* Certain input values are particularly likely to show
- *Idea:* ″Spread out″ input.
- What does that mean? How can we do that?

HIDING

DEFINTION [MIN-ENTROPY]:

A probability distribution P has *high min-entropy* if no particular value r has high probability $P(r)$ to show.

EXAMPLE: A distribution over a domain with many values, that assigns equal probability to each element of the domain has high min-entropy. For example, the probability distribution that assigns to each 256-bit string r equal probability ($= 1/2^{256}$) has high min-entropy.

Hiding

Concatenation of Strings:

- Let $s||t$ denote the concatenation of strings s and t
- *Example:* For $s =$ ”ab” and $t =$ ”yz”, we have $s||t =$ ”$abyz$”

Enforcing the Hiding Property: Idea

- Let x the input that you want to hide
- Select a probability distribution with high min-entropy
- Pick a random r according to this distribution
- Consider $H(r||x)$, that is, hash the concatenation of r and x

HIDING

Enforcing the Hiding Property: Idea

- Pick a random r from high min-entropy distribution
- Compute $H(r||x)$, that is, hash the concatenation of r and x

DEFINITION [HIDING PROPERTY – BETTER TRY]:
A hash function H is *hiding* if

- for r drawn from a high min-entropy distribution
- it is infeasible to determine any input x from $H(r||x)$.

Commitments

Definition [Commitment]:
A *commitment* is the digital analog of taking a value (or message), sealing it in an envelope.

- The value/message is yours, that is, you *committed* to the contents of the envelope
- The value/message remains a secret from everyone else
- You can open the envelope and reveal the value/message to everyone, any suitable moment
- Once open, others can verify that you commit to the value/message in the envelope

Commitment Scheme I

Committing

- Generate a random ”nonce” (= ”number used only once”) from a distribution of high min-entropy
- Hash the concatenation of *nonce* with the message *msg*, to which you commit, with a hash function H, representing the commit function
- Publish the commitment, i.e. the hash

$$com = H(nonce||msg)$$

- *com* is the envelope; everyone can see *com*

Commitment Scheme II

Opening the Envelope / Verification

- Publish the *nonce* and the message *msg*
- Everybody can check whether

$$com = H(nonce||msg)$$

If yes, you genuinely committed to the message

Commitment Scheme

Commitment scheme. A commitment scheme consists of two algorithms:

- **com := commit(*msg, nonce*)** The commit function takes a message and secret random value, called a nonce, as input and returns a commitment.
- **verify(*com, msg, nonce*)** The verify function takes a commitment, nonce, and message as input. It returns true if *com* == commit(*msg*, *nonce*) and false otherwise.

We require that the following two security properties hold:

- ***Hiding***: Given *com*, it is infeasible to find *msg*
- ***Binding***: It is infeasible to find two pairs *(msg, nonce)* and *(msg', nonce')* such that *msg* ≠ *msg'* and commit(*msg, nonce*) == commit(*msg', nonce'*)

From `https://bitcoinbook.cs.princeton.edu`

- *Hiding:* hash function *commit* has the hiding property
- *Binding:* hash function *commit* is collision resistant

Puzzle-Friendliness

PUZZLE-FRIENDLINESS: DEFINITION

DEFINITION [PUZZLE-FRIENDLINESS]:

Let H be a hash function, and

- k be drawn from a high min-entropy distribution
- Y be a set of output values, defined by
  - n bits being predetermined
  - the remaining bits being arbitrary

Then H is *puzzle-friendly* if it is infeasible to find x such that

$$H(k||x) \in Y$$

in significantly less than 2^n trials.

Puzzle-Friendliness: Explanation

Example:

- Let k be from high min-entropy distribution
- Let H have a 256-bit output
- Let
$$Y := \{x_1 ... x_{256} \mid x_1 = ... = x_n = 0\}$$
be all bit strings of length 256 whose first n positions are zero

H is puzzle-friendly, if one needs 2^n trials for finding x such $H(k||x) \in Y$.

Puzzle-Friendliness: Explanation

Intuition: Let

- S be the set of output values overall
- $Y \subset S$ a particular subset of output values
- r be sufficiently random

The smaller Y, the longer it takes to find x such that $H(r||x) \in Y$.

Puzzle-Friendliness versus Hiding: Let

- S consist of sufficiently many elements, e.g. all 256-bit strings
- H be puzzle-friendly

Then H is also hiding.

Proof: Hiding translates into considering $Y = \{y\}$. Puzzle-friendliness implies requiring (about) 2^{256} trials for finding x such $H(k||x) = y$, which means hiding. $\square$

UNIVERSITÄT BIELEFELD

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

$$H(id \parallel x) \in \boldsymbol{Y}.$$

- If H has n-bit output (e.g. $n = 256$), H can take any of 2^n different values
- The smaller Y, the harder the puzzle
- Puzzle *id* is sufficiently random
- For puzzle-friendly H, finding x requires maximum amount of time possible
- Puzzle-friendly hash functions give rise to *hard* search puzzles

Blockchains
–
Motivation

Hash Functions
–
Introduction

Hash Functions
–
Central Properties

The Merkle-Damgard Transform

SHA - Secure Hash Algorithm

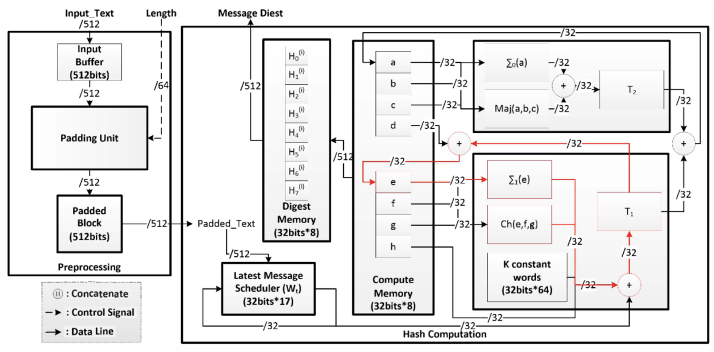
Bitcoin and SHA-256

- Bitcoin uses the *SHA-256* as the central hash function
- SHA-256 was invented in 2001 by the NSA
- The SHA-256 has all properties required:
  - It is collision-resistant; so far, no collision observed
  - It has the hiding property
  - It is puzzle-friendly

SHA-256: Technical Properties

- The SHA-256 can take inputs of arbitrary length and generates 256-bit output
- It builds on a *compression function* that takes fixed-length input, and
- the *Merkle-Damgard transform*, which enables input of arbitrary length for fixed-length input functions

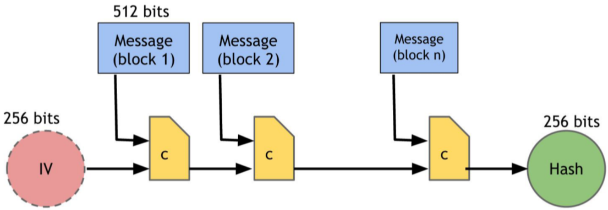
SHA-256: Compression Function



From Jeong & Kim, 2014

Take-Home-Message: It's complicated and works

Merkle-Damgard Transform I

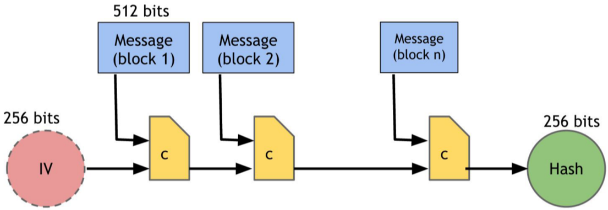


Merkle-Damgard: Iterated application of compression function *c*

From `bitcoinbook.cs.princeton.edu`

- The **Merkle-Damgard transform** turns a hash function of fixed-length input into one of arbitrary-length input
- It preserves collision resistance: if compression function is collision resistant, so is Merkle-Damgard transform

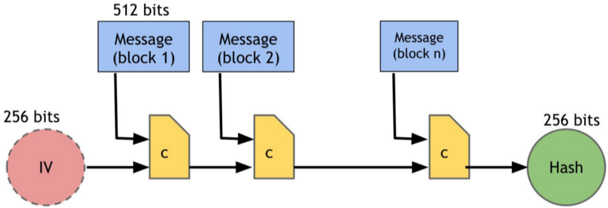
Merkle-Damgard Transform II



From bitcoinbook.cs.princeton.edu

- The compression function takes inputs of fixed length m and produces an output of length n, where $n < m$
- Divide the input of arbitrary length into blocks of length $m - n$
- Here, $m = 768, n = 256$, so $m - n = 512$

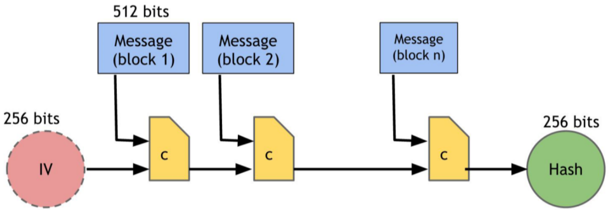
Merkle-Damgard Transform III



From bitcoinbook.cs.princeton.edu

- Pass each block together with the output of the previous block into the compression function.
- Input length $= m - n + n = m$, which is the fixed length of the input of the compression function.

Merkle-Damgard Transform IV



From `bitcoinbook.cs.princeton.edu`

- For the 1st block, we use an **Initialization Vector (IV)** as input, because there is no previous block output.
- The **IV** is reused for every call to the hash function.
- The output of the last block is the output that is returned

PADDING I

Issue

- *Observation:* The Merkle-Damgard Transform takes input of length $n \times 512$, where n is arbitrary
- Arbitrary-length input does not necessarily match this requirement
- When breaking the input into fixed sized blocks, the last block may be too small
- Padding addresses to get the last block to the right size

Padding II

Solution

- Add a ″1″ followed by as many ″0″s as necessary to the last block
- Also add a 64- or 128-bit integer that specifies the length of the entire message
  ☞ This prevents *length extension attacks*
- The length of the block and the length (64 or 128) of the integer determine the number of ″0″s
- Once padded, the input suits the Merkle-Damgard transform

MATERIALS / OUTLOOK

- See *Bitcoin and Cryptocurrency Technologies*, 1.1
- See `https://bitcoinbook.cs.princeton.edu/` for further resources
- Further: T. Kuo, H.Kim and L. Ohno-Machado (2017): *Blockchain ditributed ledger technologies for biomedical and health care applications*
- Next lecture: “Cryptography II”
  - See *Bitcoin and Cryptocurrency Technologies* 1.2–1.4