Blockchain and Cryptocurrencies

Chapter 1: Basic Tools — Hashed Datastructures

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2020

Literature for this lecture

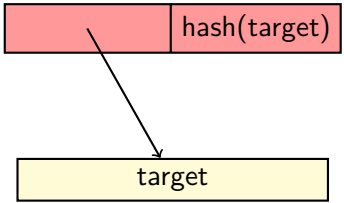
- Chapter 1 of Bitcoin and Cryptocurrency Technologies

Contents

Contents

Hash Pointers

- A self verifying data structure
- Consists of a reference to a target paired with the hash of the target
- If the target changes, we can find out by recomputing the hash and comparing with the stored hash.



Hash Pointers in Python

```
1  import hashlib
2  ## library with various hash functions
3  class HashPointer:
4      def __init__ (self, target : bytearray):
5          self._target = target
6          self._hash = self.digest()
7
8      def verify(self):
9          return self._hash == self.digest()
10
11     def digest(self):
12         m = hashlib.sha256(self._target)
13         return m.hexdigest()
```

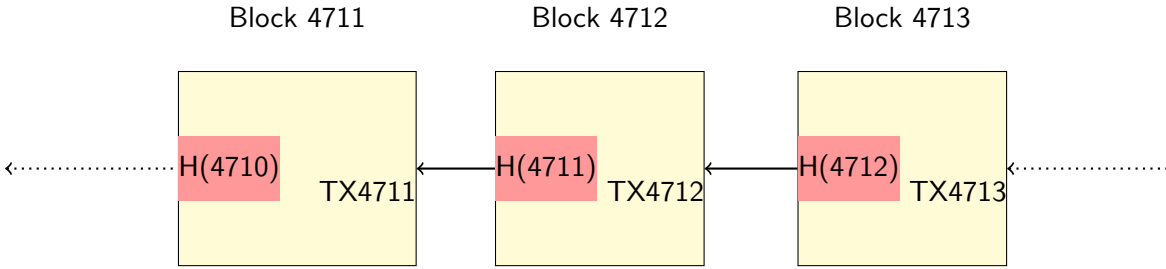
- hashlib.sha256 computes the SHA-256 hash of its bytearray argument
- m.hexdigest() finalizes the hash function and converts the result into a string

Using Hash Pointers

```
1 ba = bytearray ("A Few Good Men", 'utf-8')
2
3 hp = HashPointer(ba)
4
5 hp.verify() ## returns True
6
7 ba[6] = 70 ## tamper with the target
8
9 hp.verify() ## returns False
```

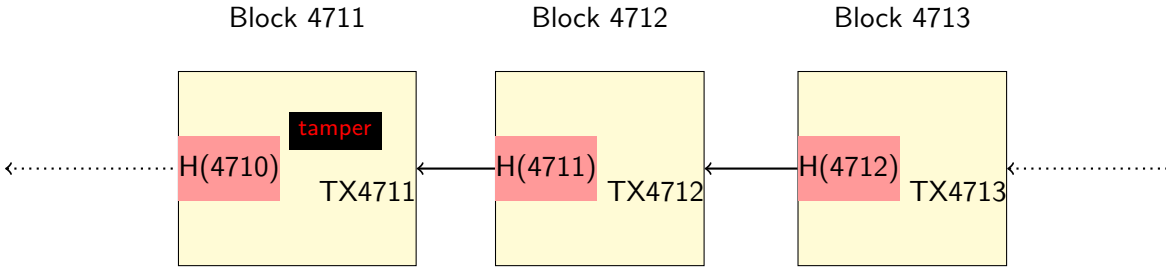
Contents

From Hash Pointer to Blockchain



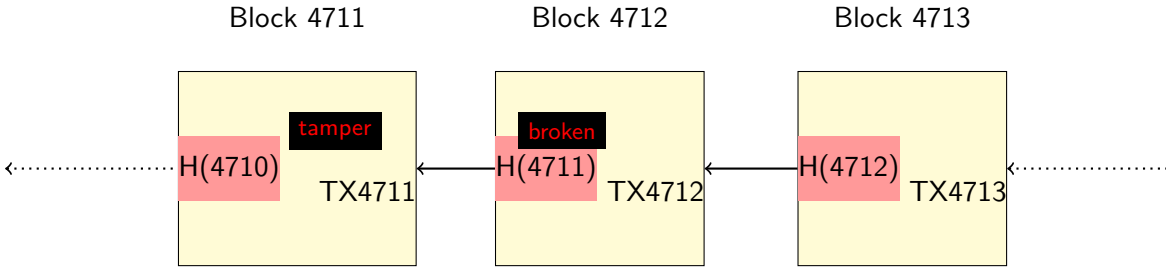
- Every block consists of the text of the transactions and the hash of the preceding block. (simplified)

From Hash Pointer to Blockchain



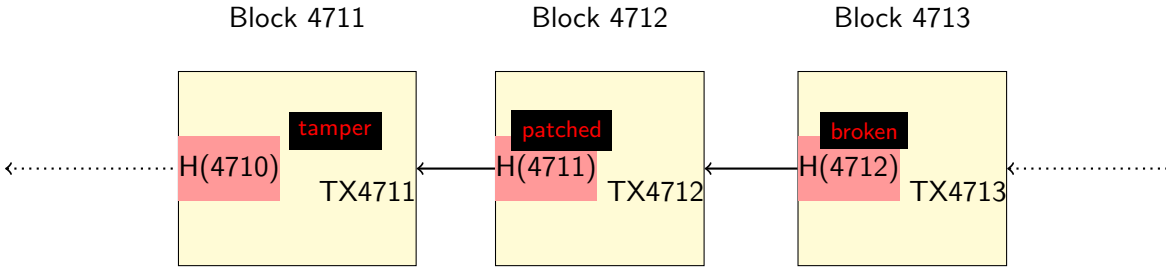
- Every block consists of the text of the transactions and the hash of the preceding block. (simplified)
- Tampering with data

From Hash Pointer to Blockchain



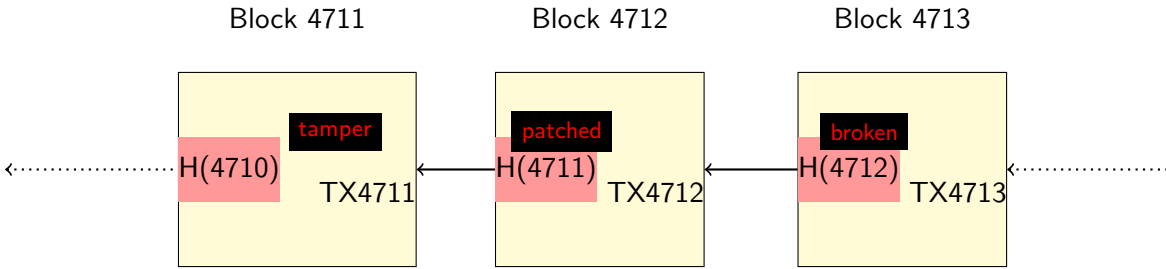
- Every block consists of the text of the transactions and the hash of the preceding block. (simplified)
- Tampering with data breaks hash in subsequent block

From Hash Pointer to Blockchain



- Every block consists of the text of the transactions and the hash of the preceding block. (simplified)
- Tampering with data breaks hash in subsequent block
- Patching that hash breaks hash in next block

From Hash Pointer to Blockchain



- Every block consists of the text of the transactions and the hash of the preceding block. (simplified)
- Tampering with data breaks hash in subsequent block
- Patching that hash breaks hash in next block and so on up to the root which cannot be patched

A Simple Blockchain

```
1  class Block:
2      def __init__ (self, data : bytearray, prev = None):
3          self._prev = prev
4          self._data = data
5          self._prev_hash = prev.digest() if prev is not None else bytearray(256)
6
7  class Blockchain:
8      def __init__(self):
9          self._root_hash = bytearray (256)
10         self._list = None
```

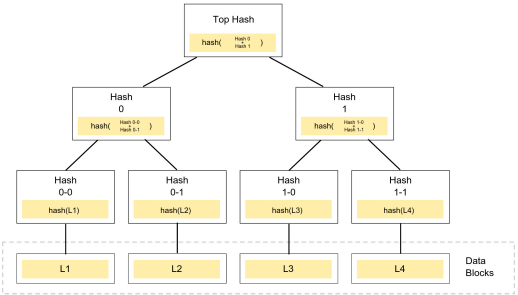
- See demo video and accompanying code

Contents

Merkle Tree

Definition (Wikipedia)

A **hash tree** or **Merkle tree** is a tree in which every leaf node is labeled with the cryptographic hash of a data block, and every non-leaf node is labeled with the cryptographic hash of the labels of its child nodes.



By Azaghal - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=18157888

Ralph Merkle. By david.orban - https://www.flickr.com/photos/davidorban/1347574959/, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=18167859

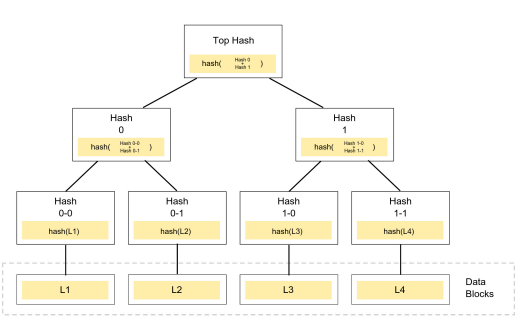
Features of Merkle Tree

- The root hash (Top hash) summarizes all information in the tree
- There is a concise proof of membership of a data block in the tree
- There is a concise proof of non-membership in a **sorted** tree
- Sorted means that an in-order traversal of the tree visits the data blocks in ascending order (e.g., $L1 < L2 < L3 < L4$ in the picture).

Proof of Membership in a Merkle Tree

Membership

To prove membership of a data block it is sufficient to provide the hash of the data block and all hashes along the path from that data block to the root of the tree.
In a tree of size n the proof consists of $O(\log n)$ hash values.



By Azaghal - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=18157888

Example

In the picture, the proof that $L2$ is member of the tree consists of the list of red items:

- (Hash 0-0, Left)
- Hash 0-1 = hash (L2),
- Hash 0 = hash (Hash 0-0 ‖ Hash 0-1),
- (Hash 1, Right)
- Top Hash = hash (Hash 0 ‖ Hash 1)

Proof of Non-Membership in a Merkle Tree

Non-Membership

To prove non-membership of a data block x in a sorted tree, we need

- proof of membership of the largest x_1 such that $x_1 < x$ in the tree,
- proof of membership of the smallest x_2 such that $x < x_2$ in the tree, and
- demonstrate that x_1 and x_2 are direct neighbors by comparing the proofs.

(We leave the corner cases where x is smaller or bigger than all elements in the tree for the exercise.)

Comparing Proofs

Let P_i be the proof that x_i is in the tree (for $i = 1, 2$). We process both proof lists backwards starting from the root.

Algorithm Direct Neighbors

DirectNeighbors (P_1, P_2):

- If P_1 or P_2 empty, then fail.
- Consider the last step: $P_1 = (P_1'.(H_1, d_1))$, $P_2 = (P_2'.(H_2, d_2))$ where $d_i \in \{Left, Right\}$.
- If $H_1 = H_2$ and $d_1 = d_2$, then DirectNeighbors (P_1', P_2').
- If $d_1 = Left$ and $d_2 = Right$, then Leftmost (P_1') and Rightmost (P_2').
- If $d_1 = Right$ and $d_2 = Left$, then Rightmost (P_1') and Leftmost(P_2').

Leftmost(P):

- All directions in P are $Right$.

Rightmost(P):

- All directions in P are $Left$.

Comparing Proofs

Example

- Proof for L2 is [(Hash 0-0, Left), (Hash 1, Right)].
- Proof for L3 is [(Hash 1-1, Right), (Hash 0, Left)].
- DirectNeighbors considers (Hash 1, Right) and (Hash 0, Left) first.
- As Right $\neq$ Left, it remains to check that
  - Rightmost ([(Hash 0-0, Left)]) which is true
  - Leftmost ([Hash 1-1, Right)]) which is also true
- Hence, L2 and L3 are direct neighbors.
- If the example tree is sorted, then any value L such that $L2 < L < L3$ cannot be in the tree.

Constructing a Merkle Tree

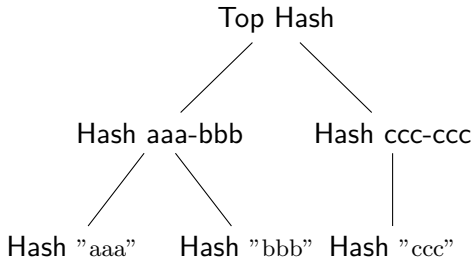
Issue: a balanced binary tree requires $n = 2^k$ data items. What if n is not a power of two?

Algorithm

Input: a list of $n > 0$ data items

1. Construct the working list of all leaf nodes by computing the hash of each data item
2. Construct the new working list of inner nodes by taking two adjacent nodes from the working list as children and computing the hash of the concatenation of their hashes. If only one node is left, duplicate it.
3. If more than one node remains in the working list, repeat from step 2
4. Otherwise, the remaining node is the root of the Merkle tree

Concrete Example



- The data items are the strings "aaa", "bbb", "ccc" using encoding 'utf−8'
- The hash function is SHA-256.

Thanks!