Cryptography Engineering

- Lecture 12 (Jan 27, 2026)
- Today's notes:
  - Case study: Hash functions and Digital signature in Blockchain

Previous Topics

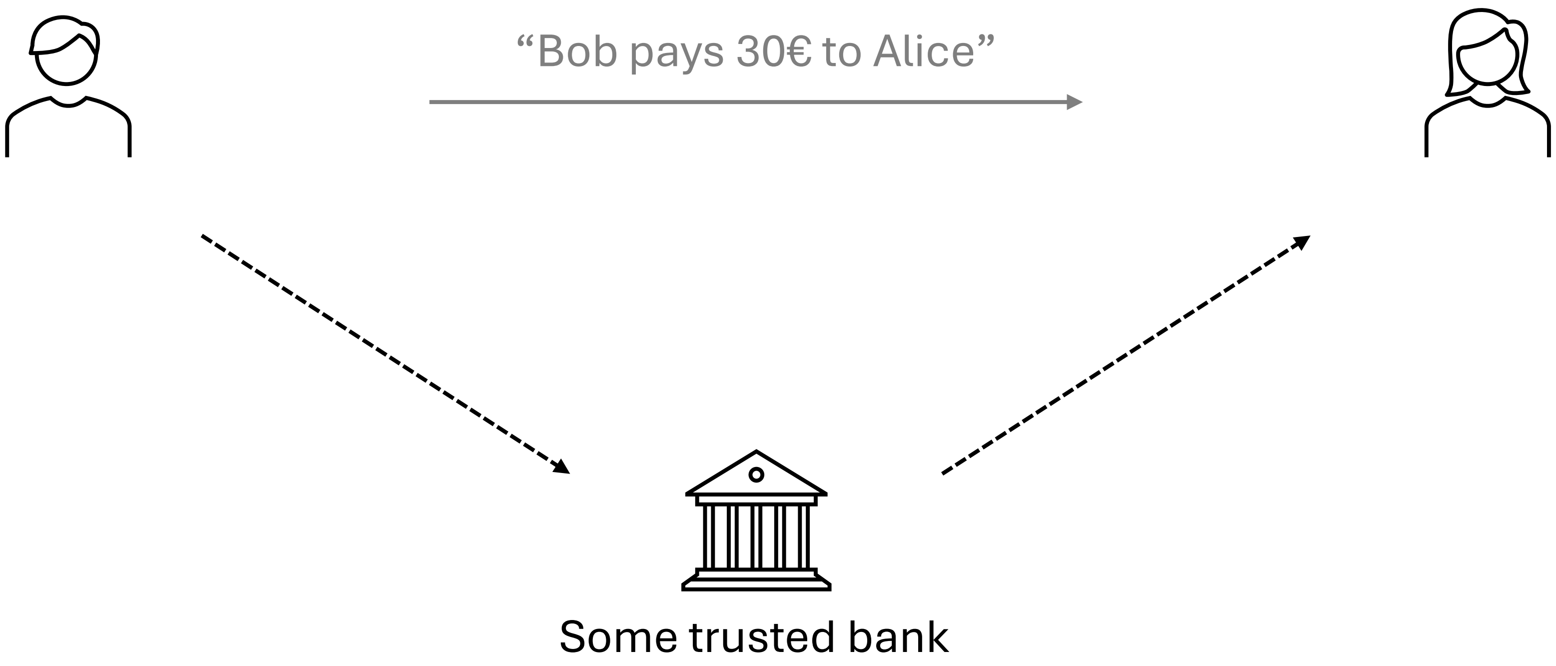
- Symmetric primitives: Hash functions, HKDF, HMAC, AEAD, ...
- Diffie-Hellman key exchange (DHKE), digital signature
- Certificate, TLS handshake
- Fujisaki-Okamoto Transform, CCA security
- Key encapsulation mechanism(KEM), post-quantum TLS (PQ-TLS), KEM-TLS
- Case study: X3DH + Double Ratchet => Secure messaging
- Password authentication, password storage (hashed + salted password)
- Password over TLS, SCRAM
- OPAQUE: Oblivious PRF + 3DH, against pre-computation attack

Today's Contents

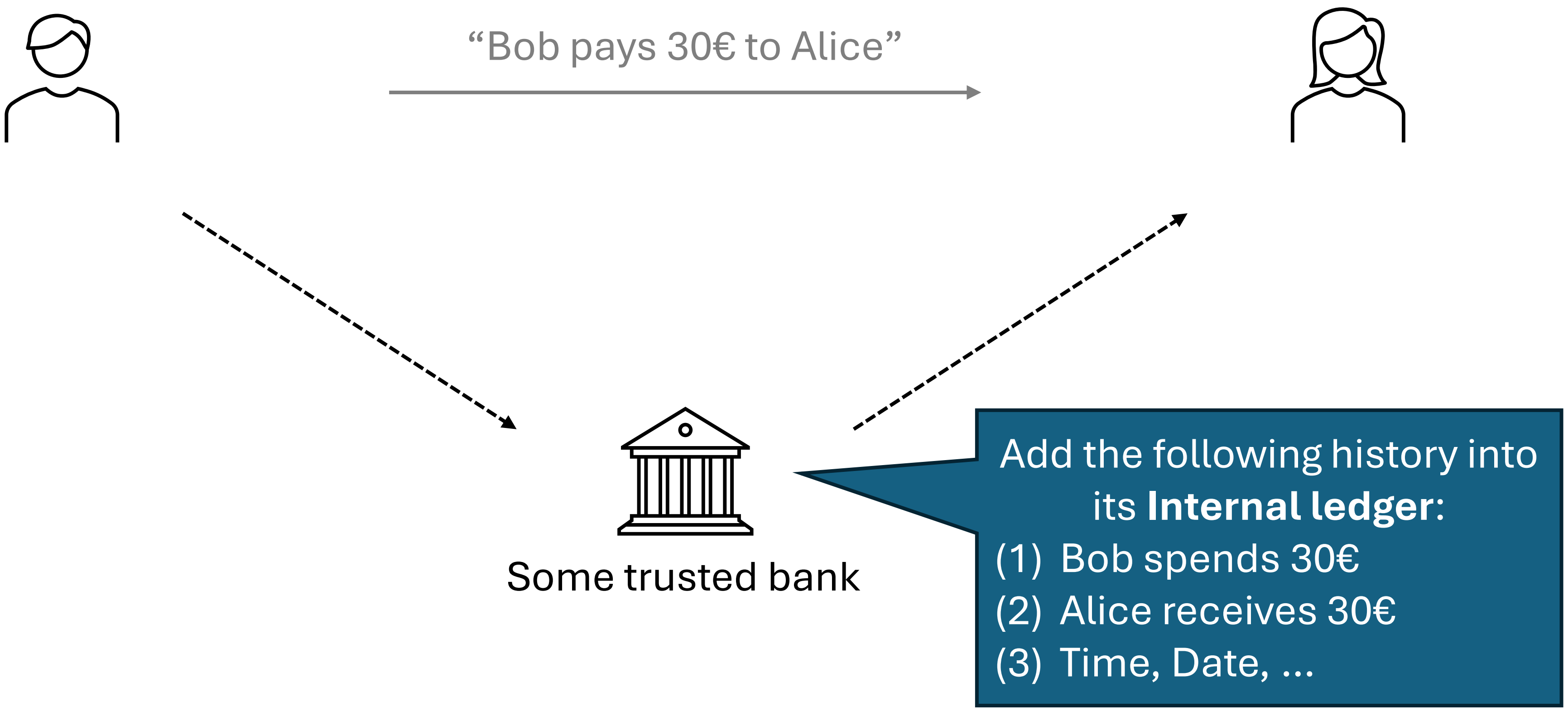
- Hash & Signature in blockchains:
  - Brief background: public authenticated ledger and blockchain
  - Hash functions: Hash-linked blocks
  - Merkle tree: Commitment, inclusion proofs
  - ECDSA: Authorizing transaction

Trusted Ledger

Trusted Ledger



Trusted Ledger



Trusted Ledger

Some trusted bank

- Centralized trusted ledger
  - The ledger (e.g., history of transactions) is maintained by some authority
  - Centralized system

Trusted Ledger

Some trusted bank

- Centralized trusted ledger
  - The ledger (e.g., history of transactions) is maintained by some authority
  - Centralized system
    - Efficient: Fast confirmation
    - Simplicity and strong consistency
    - Cost-effective in many settings

Trusted Ledger

- Centralized trusted ledger
  - The ledger (e.g., history of transactions) is maintained by some authority
  - Centralized system

- Efficient: Fast confirmation
- Simplicity and strong consistency
- Cost-effective in many settings

- Require trusted parties
- Single point of trust/failure
- Limited transparency / verifiability
- Potential censorship & Insider attacks

Decentralized public authenticated ledger

- Can we maintain a ledger **without trusting a single authority**, while still keeping it consistent and verifiable?

- Goal: Decentralized ledger that achieves
  - No single trusted maintainer
  - Public and transparent (e.g., anyone can verify)
  - Authenticated (e.g., transaction history is tamper-evident)

Decentralized public authenticated ledger

- Can we maintain a ledger **without trusting a single authority**, while still keeping it consistent and verifiable?

Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

- Goal: Dec
  - No sin
  - Public
  - Authenticated (e.g., transaction history is tamper-evident)

Decentralized public authenticated ledger

- Use **blockchain** to build decentralized public authenticated ledger

- A blockchain system includes many components:
  - E.g., Consensus mechanisms, peer-to-peer networking, smart contracts…

- In this lecture, we focus on how to:
  - Use hash functions to build blocks and link them into a chain
  - Use Merkle trees to achieve tamper-evident commitment to transaction history
  - Use signatures to authorize transactions and prove ownership
  - Make them publicly verifiable

Hash Chain

- Let H be a hash function

- Motivated question: Suppose that we have many data blocks, how can we construct a **compact digest** to record their order?

Hash Chain

- Let H be a hash function

- Motivated question: Suppose that we have many data blocks, how can we construct a **compact digest** to record their order?

Block 1 Block 2 Block 3 Block 4

- ~~Trivial solution: Store all blocks and their order~~

Hash Chain

- Let H be a hash function

- Motivated question: Suppose that we have many data blocks, how can we construct a **compact digest** to record their order?

Block 1 | Block 2 | Block 3 | Block 4

- ~~Trivial solution: Store all blocks and their order~~
- Use hash function:

$$\textbf{Digest} = H(H(H(H(\text{prefix}, \text{Block 1}), \text{Block 2}), \text{Block 3}), \text{Block 4})$$

Hash Chain

- Let H be a hash function

- Motivated question: Suppose that we have many data blocks, how can we construct a **compact digest** to record their order?

Block 1	Block 2	Block 3	Block 4
(1) **Header 1**	(1) **Header 2**	(1) **Header 3**	(1) **Header 4**
(2) Data 1	(2) Data 2	(2) Data 3	(2) Data 4
…	…	…	…

- ~~Trivial solution: Store all blocks and their order~~
- Use hash function **(to hash the short headers):**

$$\textbf{Digest} = H(H(H(H(\text{prefix}, \text{Header 1}), \text{Header 2}), \text{Header 3}), \text{Header 4})$$

The Chain Structure in Bitcoin

Genesis block

- Anchor of the chain, no previous hashes
- Hardcoded into Bitcoin applications
- Publicly known

```
GetHash()       = 0x000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
hashMerkleRoot = 0x4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b
txNew.vin[0].scriptSig     = 486604799 4 0x736B6E616220726F662074756F6C69616220646E6F63657320666F206B6E697262206E6F20726F6C6C65636E61684320393030322F6E614A2F33302073656D695420656854
txNew.vout[0].nValue       = 5000000000
txNew.vout[0].scriptPubKey = 0x5F1DF16B2B704C8A578D0BBAF74D385CDE12C11EE50455F3C438EF4C3FBCF649B6DE611FEAE06279A60939E028A8D65C10B73071A6F16719274855FEB0FD8A6704 OP_CHECKSIG
block.nVersion = 1
block.nTime     = 1231006505
block.nBits     = 0x1d00ffff
block.nNonce    = 2083236893

CBlock(hash=000000000019d6, ver=1, hashPrevBlock=00000000000000, hashMerkleRoot=4a5e1e, nTime=1231006505, nBits=1d00ffff, nNonce=2083236893, vtx=1)
  CTransaction(hash=4a5e1e, ver=1, vin.size=1, vout.size=1, nLockTime=0)
    CTxIn(COutPoint(000000, -1), coinbase 04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e63656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c6f757420666f722062616e6b73)
    CTxOut(nValue=50.00000000, scriptPubKey=0x5F1DF16B2B704C8A578D0B)
  vMerkleTree: 4a5e1e
```

(Image from Bitcoin Wiki)

The Chain Structure in Bitcoin

Block 0
(Genesis block)

Block 1:
Prev_hash = $\boldsymbol{H(\text{Block 0 Header})}$
Merkle_root = … (Explained later)
Other header fields…

Block body (Transactions)…

The header of a block includes:

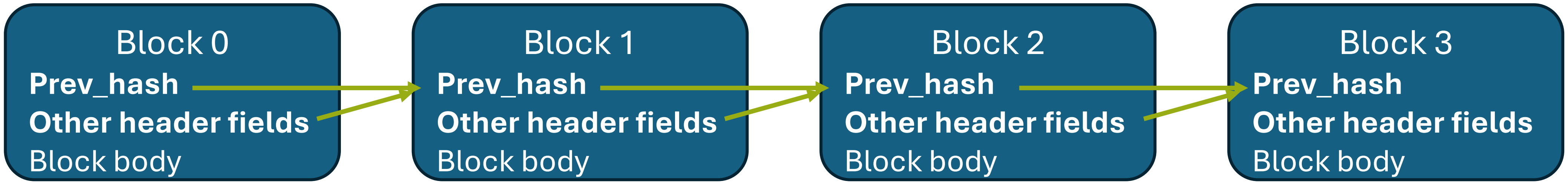
- **Prev_hash and Merkle_root,**
- nVersion, nTime, nBits, and nNonce

The block body records the transactions included in the block (and related metadata).

The Chain Structure in Bitcoin

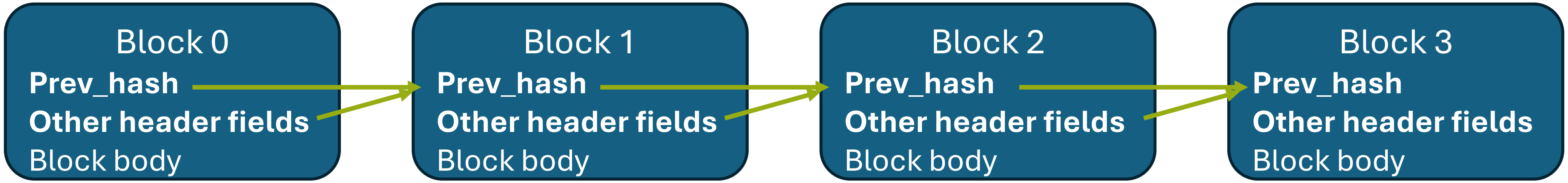
Block 0	Block 1	Block 2	Block 3
Prev_hash	Prev_hash	Prev_hash	Prev_hash
Other header fields	Other header fields	Other header fields	Other header fields
Block body	Block body	Block body	Block body

The Chain Structure in Bitcoin



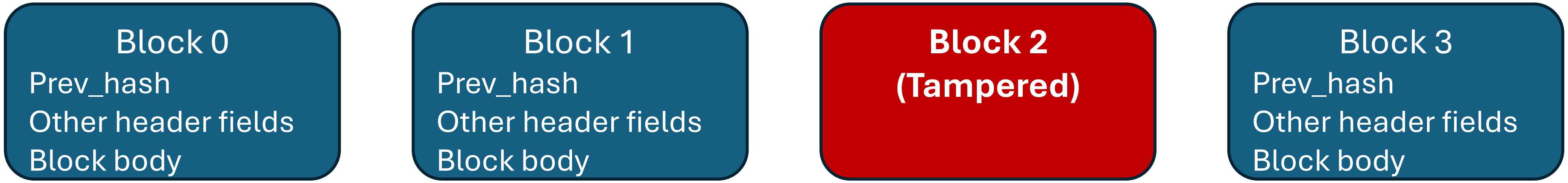
- “Chain” all blocks in order using hash function / pointer, i.e., prev_hash = H(prev_header)

The Chain Structure in Bitcoin



- "Chain" all blocks in order using hash function / pointer, i.e., prev_hash = H(prev_header)
- **What if a block is tampered?**

The Chain Structure in Bitcoin



The Chain Structure in Bitcoin

Block 0	Block 1	**Block 2**	Block 3
Prev_hash	Prev_hash	**Prev_hash**	Prev_hash
Other header fields	Other header fields	**Other header fields**	Other header fields
Block body	Block body	Block body	Block body

- If the header of block 2 was modified, then we can easily detect it via Block 3's prev_hash.

The Chain Structure in Bitcoin

Block 0	Block 1	**Block 2**	Block 3
Prev_hash	Prev_hash	Prev_hash	Prev_hash
Other header fields	Other header fields	Other header fields	Other header fields
Block body	Block body	**Block body**	Block body

- If the header of block 2 was modified, then we can easily detect it via Block 3’s prev_hash.
- **But what if only the block body of Block 2 was modified (e.g., inserting or changing transactions)?**

The Block Structure in Bitcoin

Block i
Prev_hash
Other header fields
Block body

Conceptually, a Bitcoin block body is an ordered list of transactions:

Block body = $(tx_1, tx_2, tx_3, \dots, tx_n)$

- Each transaction is a serialized objects (i.e., with specific data structure like (inputs, outputs, scripts)...)
- The **order matters**
- How can we detect if some transactions are not valid?

The Block Structure in Bitcoin

Block i
Prev_hash
Other header fields
Block body

Conceptually, a Bitcoin block body is an ordered list of transactions:

Block body = $(tx_1, tx_2, tx_3, \dots, tx_n)$

- Each transaction is a serialized objects (i.e., with specific data structure like (inputs, outputs, scripts)...)
- The order matters
- How can we detect if some transactions are not valid?

- A straight-forward solution: Compute

$$\text{tx\_commitment} = H(tx_1, tx_2, \dots, tx_n)$$

and store the hash in the block header.

The Block Structure in Bitcoin

Block i
Prev_hash
Other header fields
Block body

- How can we detect if some transactions are not valid?

  Block body = $(tx_1, tx_2, tx_3, \dots, tx_n)$

- A straight-forward solution: Compute

  tx_commitment = $H(tx_1, tx_2, \dots, tx_n)$

  and store the hash in the block header.

- **Drawbacks:** To verify, we must download the whole block to get all transactions
  - Not friendly to light clients
  - Not friendly to limited bandwidth or high latency network
  - No efficient way to prove inclusion of a single transaction

The Block Structure in Bitcoin

Block i
Prev_hash
Other header fields
Block body

- How can we detect if some transactions are not valid?

  Block body = $(tx_1, tx_2, tx_3, \ldots, tx_n)$

- A straight-forward solution: Compute

  tx_commitment = $H(tx_1, tx_2, \ldots, tx_n)$

  and store the hash in the block header.

- Drawbacks: To verify, we must download the whole block to get all transactions
  - Not friendly to light clients
  - Not friendly to limited bandwidth or high latency network
  - No efficient way to prove inclusion of a single transaction
- In Bitcoin, we use **Merkle tree** to provide a compact and efficient commitment

Merkle Tree

- Example: Generate a commitment of 8 transactions A,B,C,D,E,F,G, and H.

tx_A tx_B tx_C tx_D tx_E tx_F tx_G tx_H

Merkle Tree

- Example: Generate a commitment of 8 data blocks A,B,C,D,E,F,G, and H.

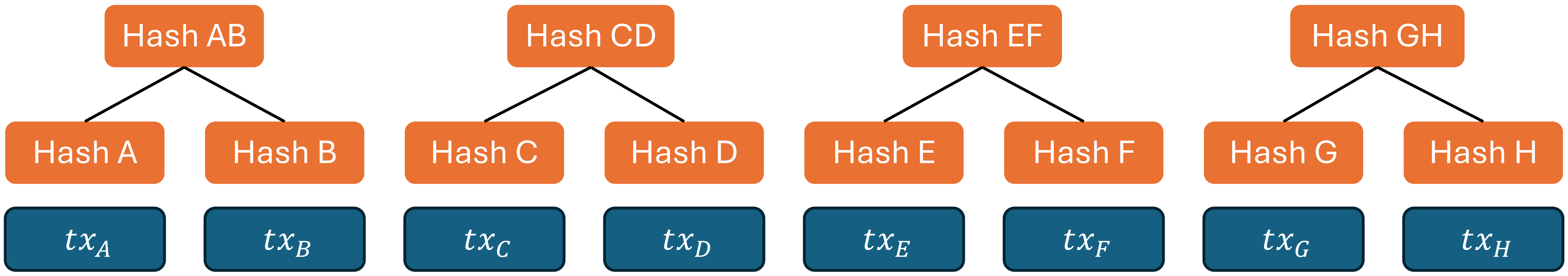
$$\text{Hash A} = H(\text{transactions } A)$$

Hash A	Hash B	Hash C	Hash D	Hash E	Hash F	Hash G	Hash H
tx_A	tx_B	tx_C	tx_D	tx_E	tx_F	tx_G	tx_H

Merkle Tree

- Example: Generate a commitment of 8 data blocks A,B,C,D,E,F,G, and H.

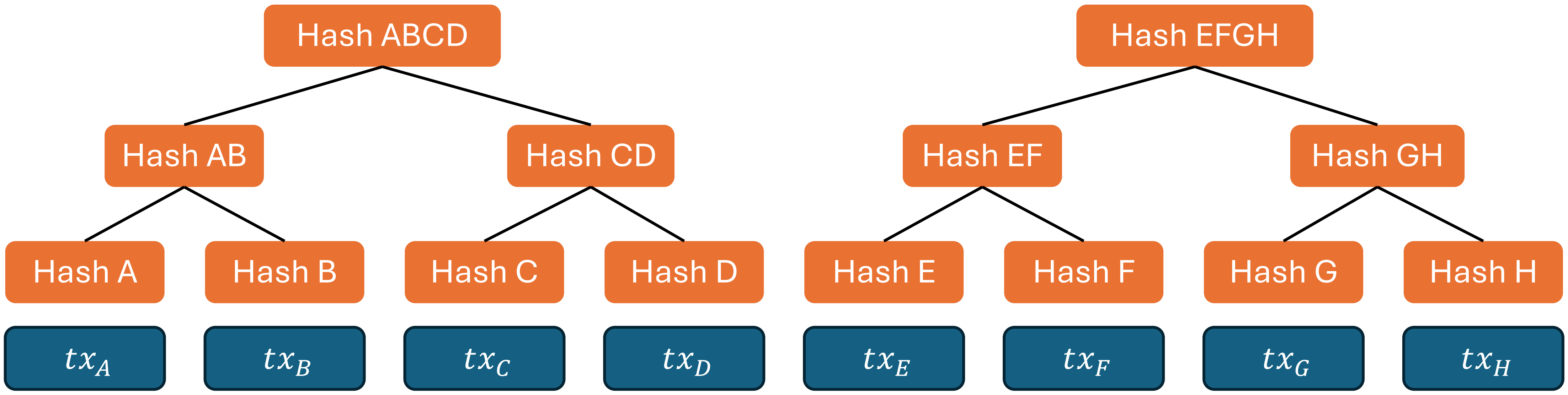
$$\text{Hash AB} = H(\text{Hash A} \mathbin{||} \text{Hash } B)$$



Merkle Tree

- Example: Generate a commitment of 8 data blocks A,B,C,D,E,F,G, and H.

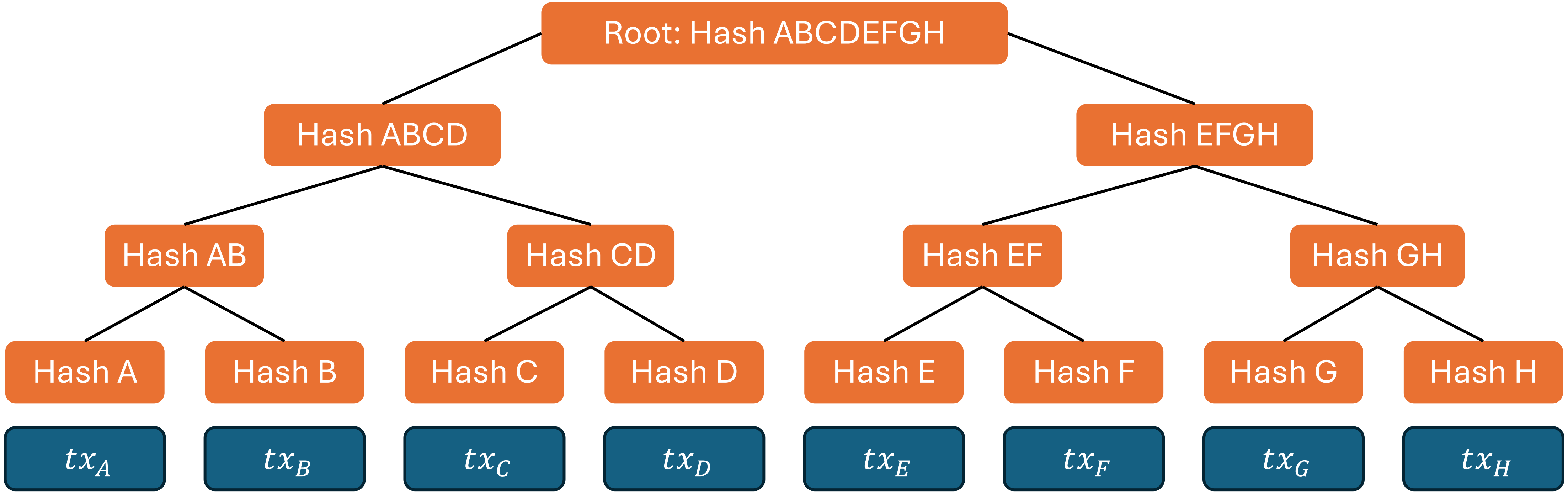
$$\text{Hash ABCD} = H(\text{Hash AB} \,||\, \text{HashCD})$$



Merkle Tree

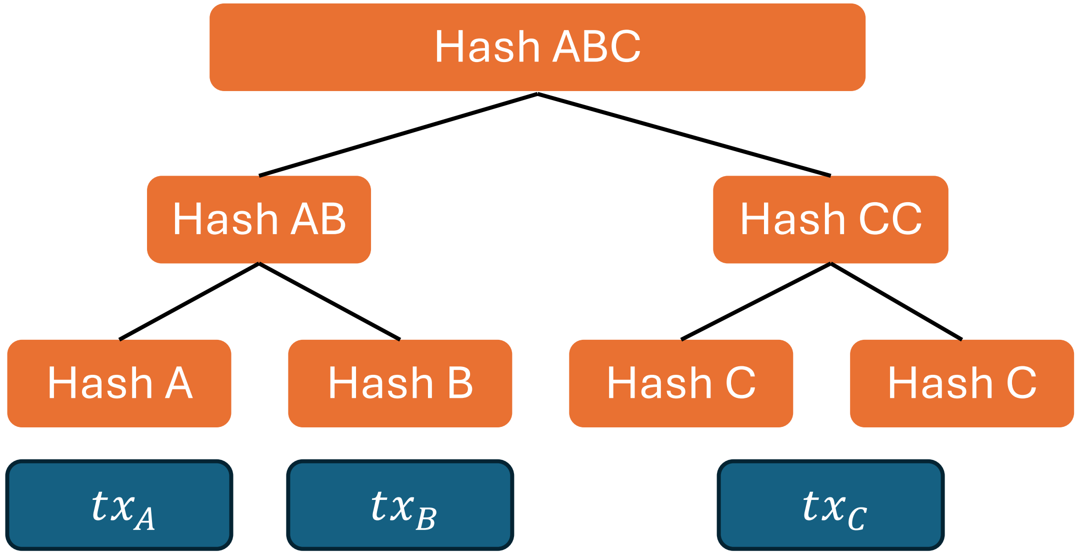
- Example: Generate a commitment of 8 data blocks A,B,C,D,E,F,G, and H.

$$\text{Merkle\_root} = H(\text{Hash ABCD} \mid\mid \text{HashEFGH})$$



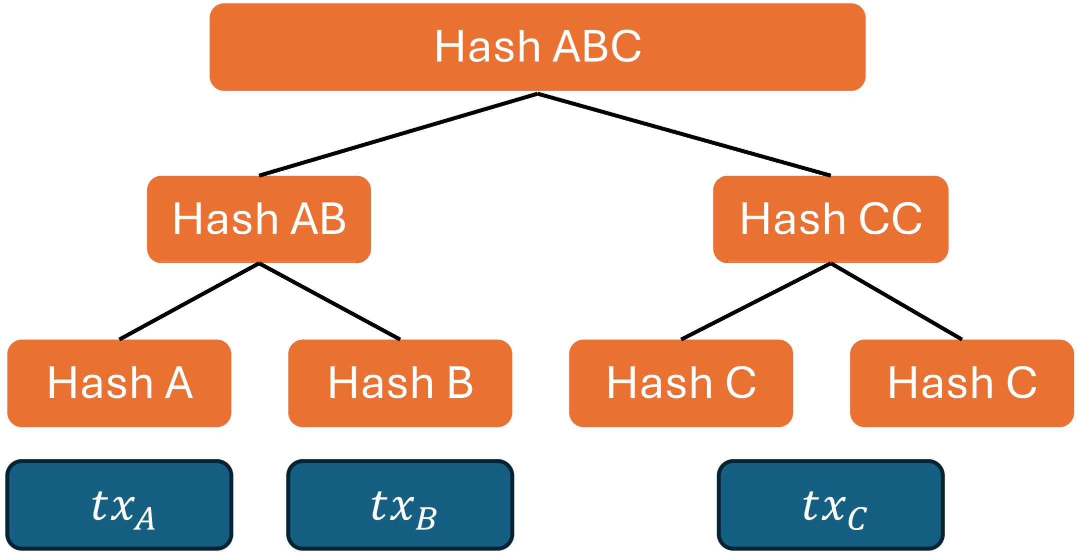
Merkle Tree

- Merkle tree ensures that all leaf nodes (transactions) contribute to the root hash
  - One leaf node modified => Different root hash
- How do we handle odd numbers of nodes?
  - Duplicate the last leaf (which still preserves the property that every leaf influences the root hash)



Merkle Tree

- How can we verify a leaf is included in the committed set (i.e., included in the tree)?

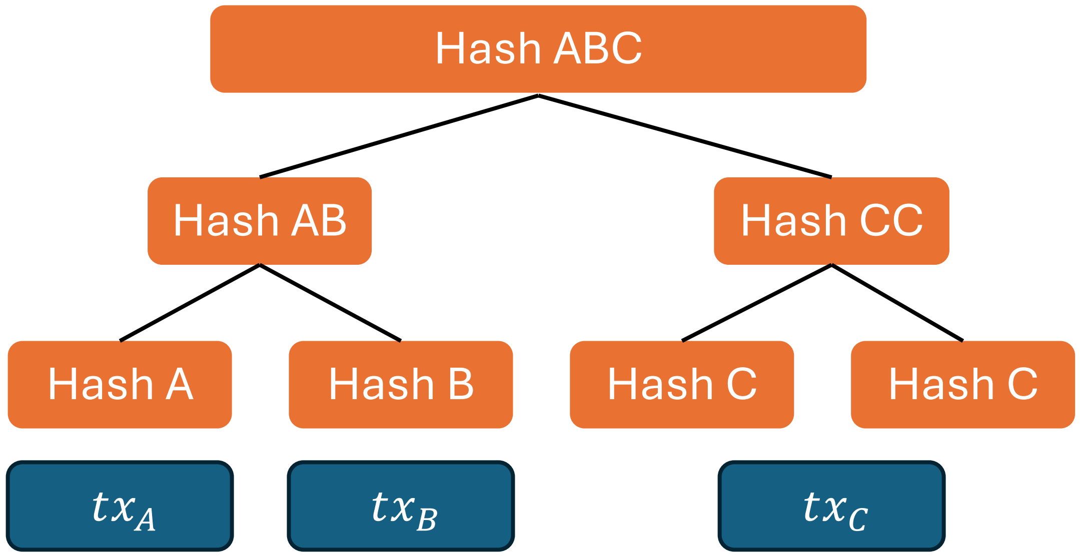


Merkle Tree

- How can we verify a leaf is included in the committed set (i.e., included in the tree)?

- Trivial inefficient solution: Request all transactions and re-construct the tree

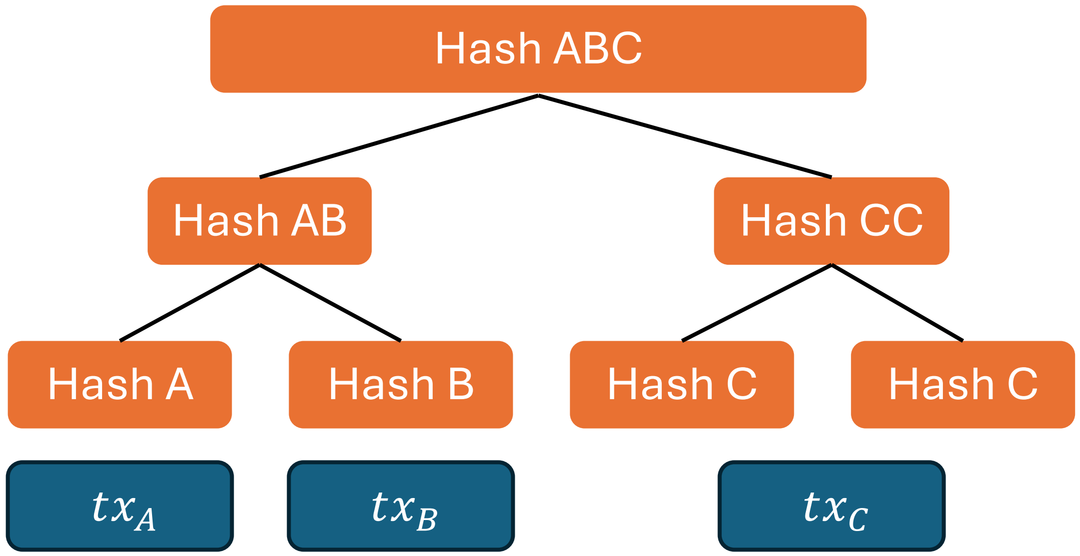
- Better solution: Merkle proof
- Example: Verify tx_A in Hash ABC



Merkle Tree

- Merkle proof: Leaf node + Merkle path => Recompute the root hash

- Example: Verify tx_A in Hash ABC
  - Leaf node: tx_A
  - Merkle path: Hash B, Hash CC

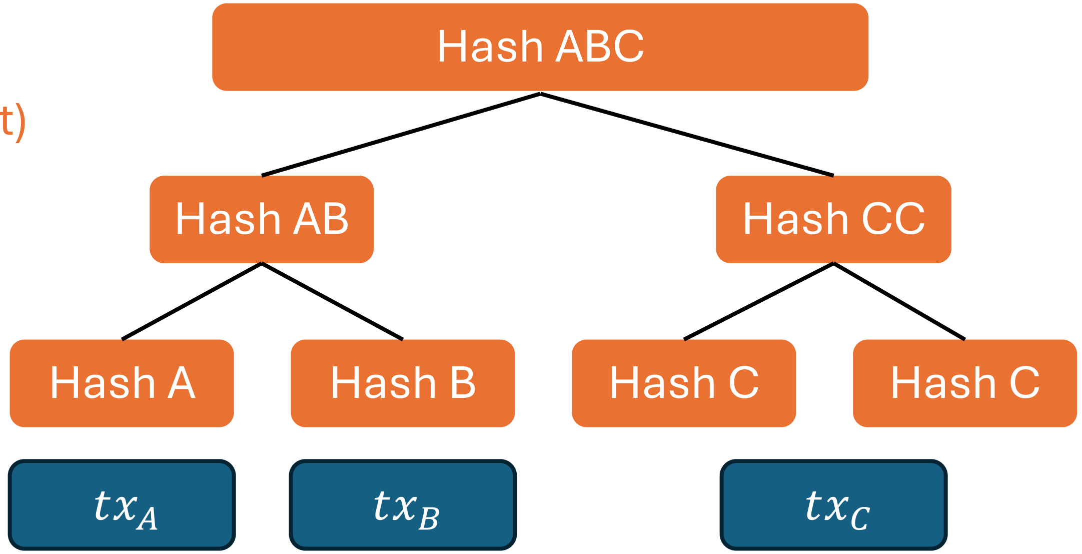


Merkle Tree

- Merkle proof: Leaf node + Merkle path => Recompute the root hash

- Example: Verify tx_A in Hash ABC
  - Leaf node: tx_A
  - Merkle path: Hash B (right), Hash CC (right)

- Example: Verify tx_B in Hash ABC
  - Leaf node: tx_B
  - Merkle path: Hash A (left), Hash CC (right)



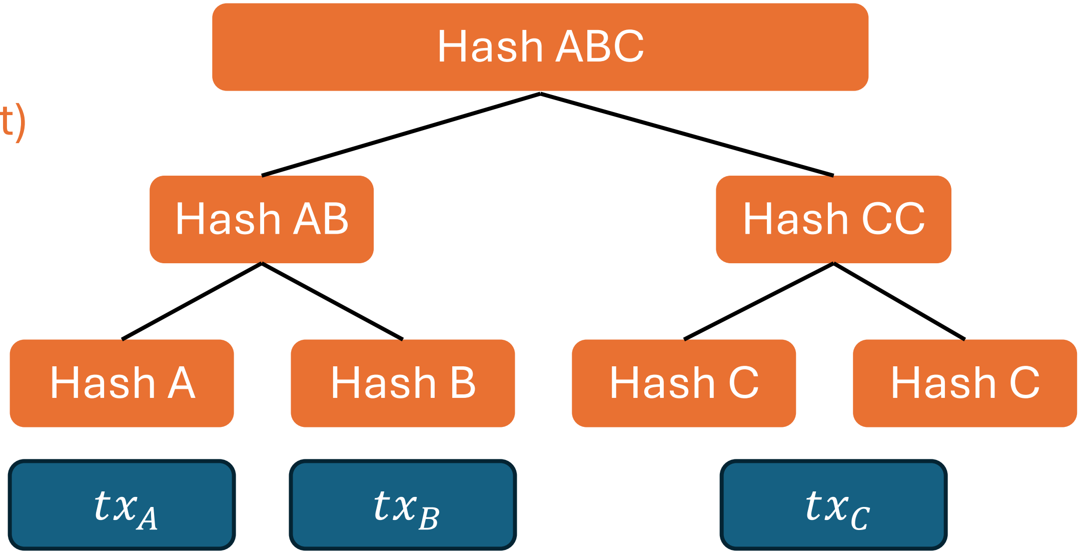
Merkle Tree

- Merkle proof: Leaf node + Merkle path => Recompute the root hash

- Example: Verify tx_A in Hash ABC
  - Leaf node: tx_A
  - Merkle path: Hash B (right), Hash CC (right)

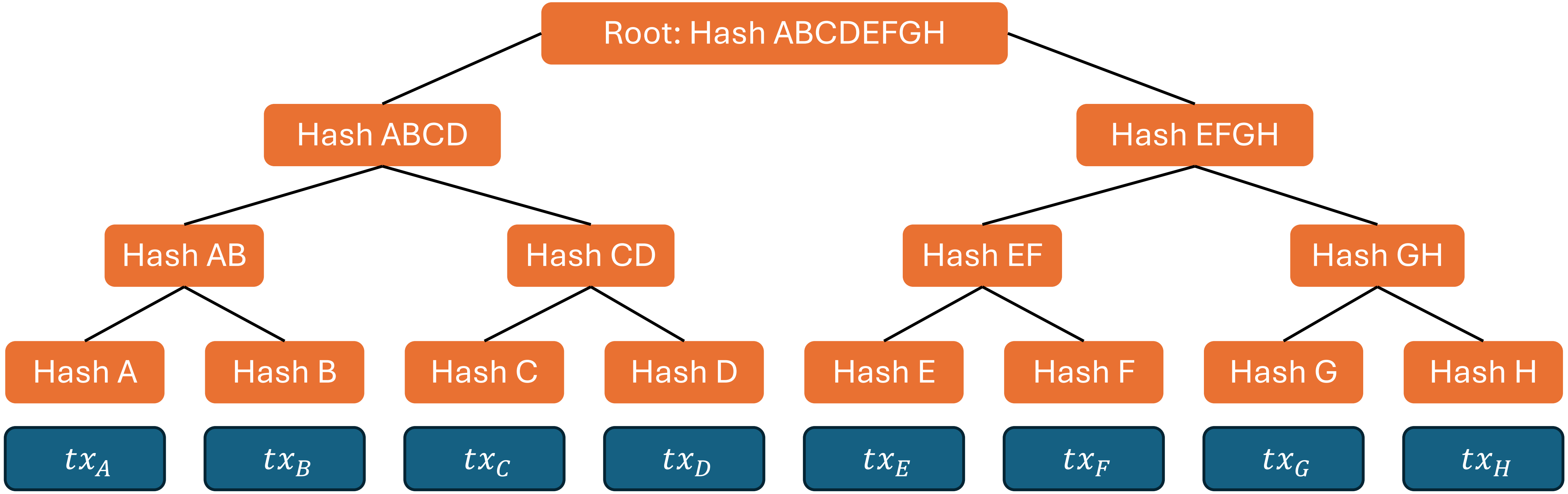
- Example: Verify tx_B in Hash ABC
  - Leaf node: tx_B
  - Merkle path: Hash A (left), Hash CC (right)

- Proof size: $\log(N)$ (the depth of the tree)

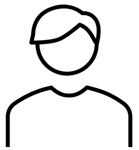


Merkle Tree

- Quick question: What is the Merkle path of tx_D

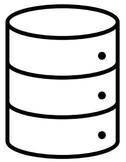


Transaction Inclusion Proof via Merkle Trees



The client tracks the longest *valid* chain (the most-work chain), and stores all block headers

Light client
(only stores block headers)



Full node
(store all blocks)

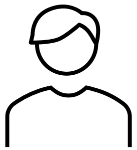
Transaction Inclusion Proof via Merkle Trees

Light client
(only stores block headers)

The block i's header already included the Merkle root hash, but no transactions

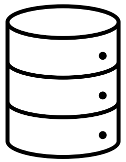
Full node
(store all blocks)

Transaction Inclusion Proof via Merkle Trees



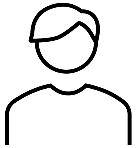
Light client
(only stores block headers)

Merkle_root_hash
of block i



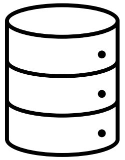
Full node
(store all blocks)

Transaction Inclusion Proof via Merkle Trees



Light client
(only stores block headers)

Merkle_root_hash
of block i



Full node
(store all blocks)

Request:
Prove that a transaction tx
is included in block i

Transaction Inclusion Proof via Merkle Trees

Light client
(only stores block headers)

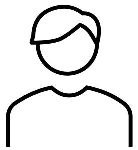
Merkle_root_hash
of block i

Request:
Prove that a transaction tx
is included in block i

Full node
(store all blocks)

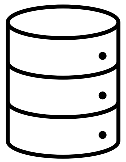
(1) Find tx
(2) Reconstruct the Merkle path of tx

Transaction Inclusion Proof via Merkle Trees



Light client
(only stores block headers)

Merkle_root_hash
of block i



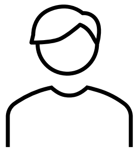
Full node
(store all blocks)

Request:
Prove that a transaction tx
is included in block i
→

(1) Find tx
(2) Reconstruct the Merkle path of tx

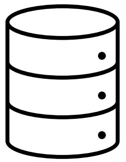
tx, the Merkle path of tx
←

Transaction Inclusion Proof via Merkle Trees



Light client
(only stores block headers)

Merkle_root_hash
of block i



Full node
(store all blocks)

Request:
Prove that a transaction tx
is included in block i

(1) Find tx
(2) Reconstruct the Merkle path of tx

tx, the Merkle path of tx

Recompute the root hash, and
compare with the Merkle_root_hash
stored in the local block header

Summary of Hash Functions in Blockchain

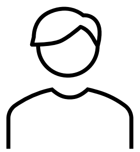
- Chain all blocks together via hash functions
  - prev_hash field in the block header
  - Preserve the order
  - Provide compact digests for each block
- Merkle tree:
  - Merkle_root field in the block header
  - Generate compact commitment of all transactions
  - Enable Merkle proofs for proving transaction inclusion (not full validity)
- Other parts involving hash functions but beyond today's scope:
  - Proof-of-Work mining: Mine a nonce such that H(block header) < Target → broadcast → others verify → chain extends → miner earns reward…
  - Identifiers, …

Authorizing Transactions in Bitcoin

- In Bitcoin, users sign their transaction via **ECDSA**

- The balance of a user is maintained via the **UTXO** model
  - Roughly, coins are stored as **Unspent Transaction Outputs (UTXOs)**
  - Balance(user) = sum of values of UTXOs that can be spent by their key(s)
- Conceptually, a Bitcoin transaction includes
  - Version
  - Inputs (includes unlocking data, e.g., the owner’s signatures of all UTXOs with their previous lock_script)
  - Outputs (includes locking data, e.g., new lock_script and the new owner’s address pk_hash)
  - Locktime
- To spend coins, a transaction
  - consumes some existing UTXOs as inputs (references to previous outputs)
  - creates new UTXOs as outputs (recipient + change)

Authorizing Transactions in Bitcoin



(pk, sk)

(1) Collect all UTXOs that he wants to spend

These UTXOs includes their references and previous lock_script (e.g., scriptPubKey)

(2) Sign transaction digest of each UTXO: $\boldsymbol{\sigma = \mathrm{ECDSA.Sign}(sk, [\mathrm{transaction\ digest}])}$

(3) Include all σ in the new UTXO's input

(4) Create the output of the new UTXO: New lock_script (recipient, change, ...

(5) Broadcast transaction

Further Topics

- Transaction Malleability & SegWit
- Proof-of-Work & Difficulty Adjustment
- Consensus Mechanism
- Smart Contract