DAT159
Module3 – Blockchain technology

L17 - Bitcoin Mechanics 2

Lars-Petter Helland, 16.10.2018

Today

- Recap of last lecture

- Signing a transaction
- Validating a transaction
- Locking and unlocking scripts
- Collecting transactions into a block
- Merkle trees and -root
- Simplified Payment Verification (SPV)
- Validating new blocks

Reading material

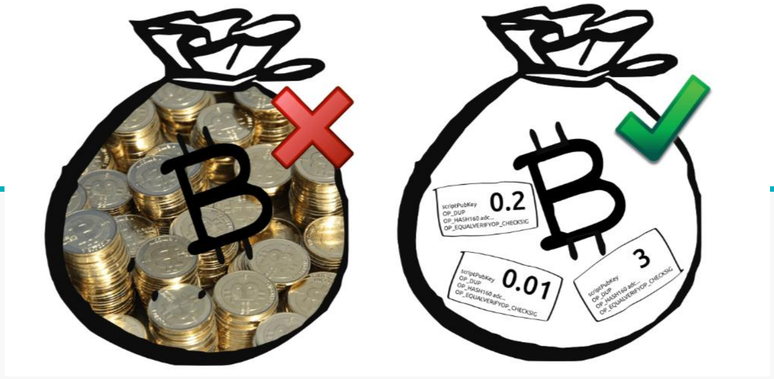
- **[AA Ch6] - Chapter 6 Transactions** ... forts.
- **[AA Ch6] - Chapter 9 The Blockchain** from Antonopoulos, Andreas M.. Mastering Bitcoin: Programming the Open Blockchain

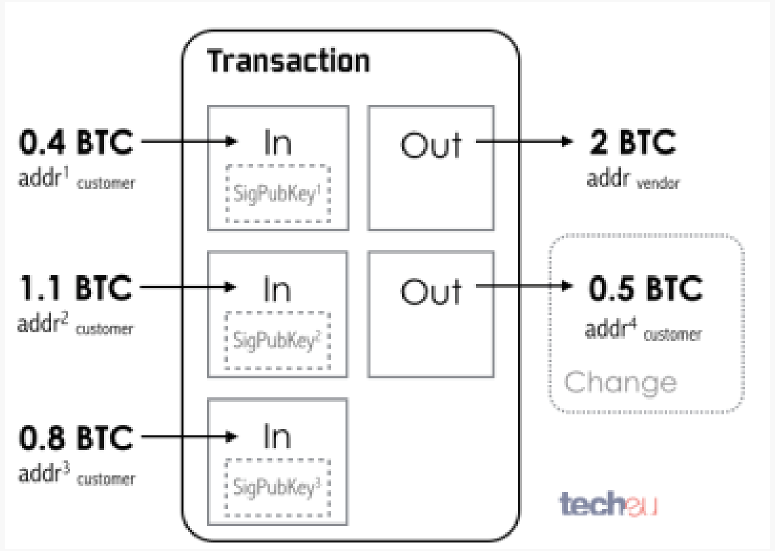
  [Some of the text in this presentation is taken directly from this book]

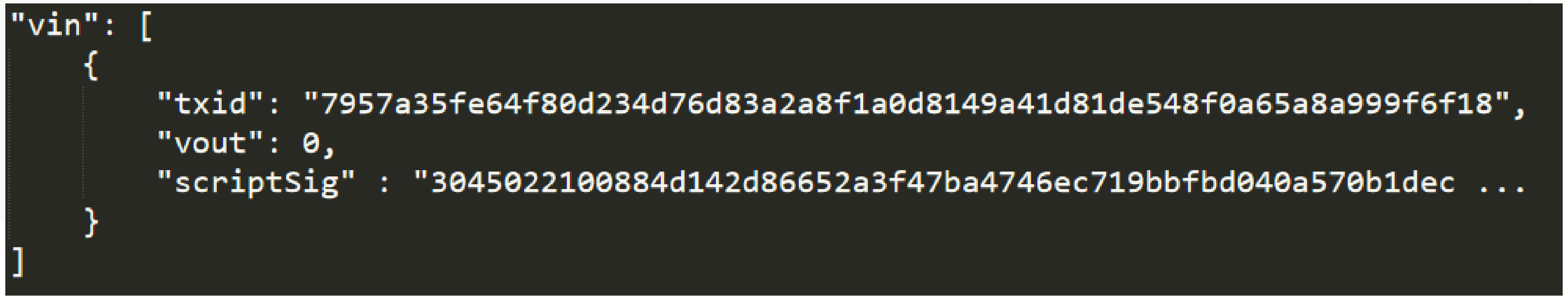
- **[NA Ch3] - Chapter 3 Mechanics of Bitcoin** from Narayanan, Arvind. Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction.

Recap of last lecture

- The accounting model
- Outputs
- Inputs
- Transactions
- Coinbase transaction
- The UTXO-set
- Wallet software







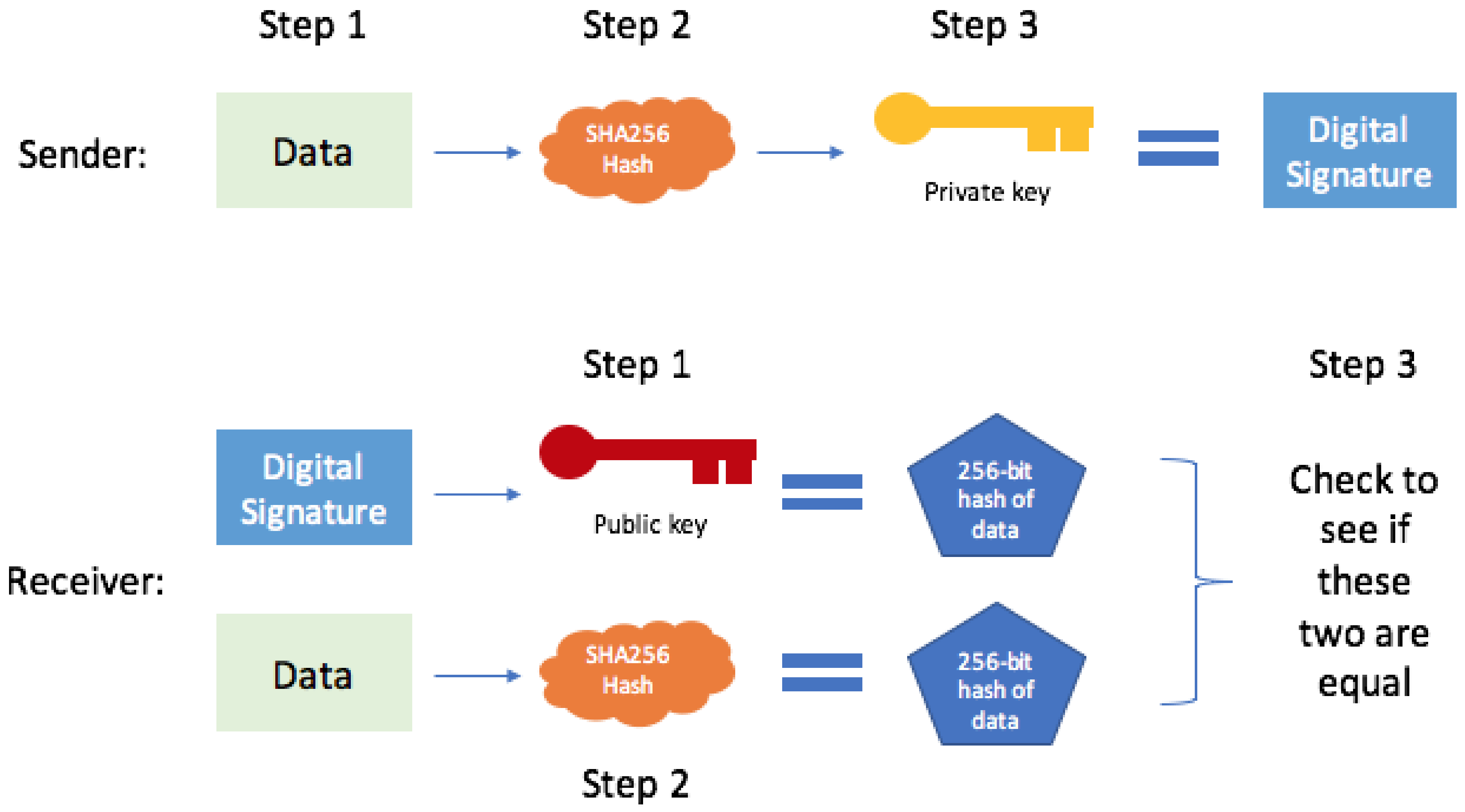
```
"vin": [
    {
        "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
        "vout": 0,
        "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1dec ...
    }
]
```

Signing a transaction

- A Bitcoin transaction is signed using ECDSA
- A transaction contains many signatures, one per input. *(Simplified in Oblig3)*
- Each input is signed using the private key for the corresponding output.
- The message that is signed is "the entire" transaction (a hash/fingerprint of all the important information)

- By doing it like this, we ensure that (for each input):
  - **The owner of the input (referenced output) is authenticated** (only he knows the private key belonging to the output address)
  - **We can verify that the owner agrees to spend the inputs** as described in the list of outputs (the message that is signed can not be modified)

Just to refresh: Creating and verifying a signature



Validating transactions

- **End users (like you and me) create the transactions** with wallet software. The transactions are then sent to the Bitcoin network for processing.
- **The network nodes must validate all incoming transactions** to make sure that money are spent correctly.
- We will list some of the things network nodes must check for in every transaction.

Transaction validation checklist (simplified)

- The transaction’s syntax and data structure must be correct.
- Neither lists of inputs or outputs are empty.
- Each output value, as well as the total, must be within the allowed range of values (less than 21m coins, more than the dust threshold).
- For each input, if the referenced output exists in any other transaction in the pool, the transaction must be rejected.
- For each input, the referenced output must exist and cannot already be spent.
- Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in the allowed range of values (less than 21m coins, more than 0).
- Reject if the sum of input values is less than sum of output values.
- The unlocking scripts for each input must validate against the corresponding output locking scripts.

Locking and unlocking scripts

- I have been "lying" a little bit about how the spending of outputs is secured by providing a signature for the transaction (or for the inputs).
- I told you that:
  - An **output** contains an **address** (linked to the **public key**)
  - An **input** contains a **signature** (+ a **public key** for verification)
  - The **transaction** represents the **message** that is signed
  - Verification is done by:
    1. Matching the public key with the address of the referenced output
    2. Checking that the signature is valid
- In real life, it is actually a little more complicated ...

Locking and unlocking scripts

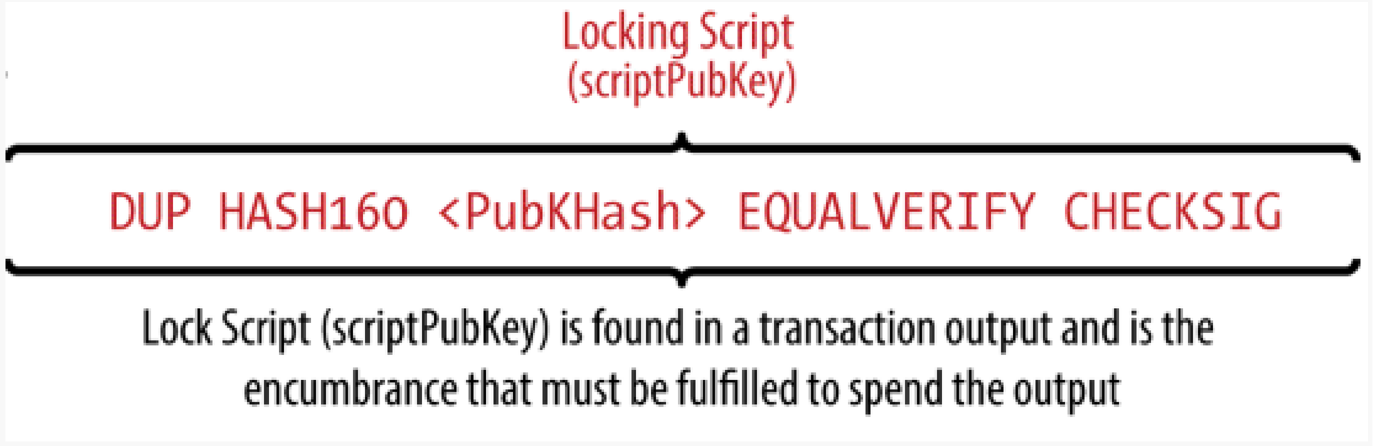
- The truth:
  - An **output** contains an **locking script** (most often containing a hash of the public key)
  - An **input** contains a **unlocking script** (most often containing a signature and a public key)

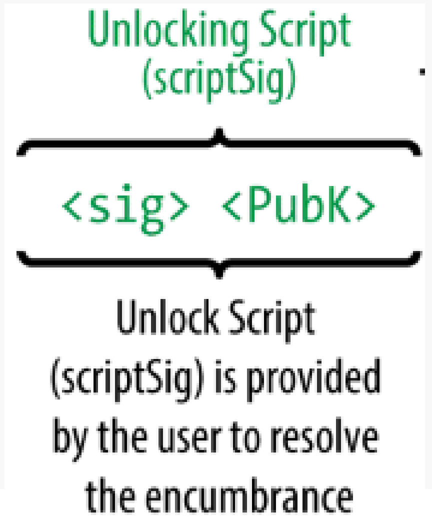
- The script language in Bitcoin is called **Script**, and is a stack based language. *(Similar to Forth, or RPN used on calculators like HP-15C)*

- The scripts (unlocking+locking) are run together as a one script
- We must look at how it works.

Locking and unlocking scripts

- The top script is a typical example of a locking script (located in an **Output**). The **<PubKHash>** is a hash of the public key, similar to the **address**.
- The bottom script is a corresponding unlocking script (located in an **Input**). The **<sig>** is the provided signature, and the **<PubK>** is the public key.


Locking Script
(scriptPubKey)
DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
Lock Script (scriptPubKey) is found in a transaction output and is the encumbrance that must be fulfilled to spend the output


Unlocking Script
(scriptSig)
<sig> <PubK>
Unlock Script (scriptSig) is provided by the user to resolve the encumbrance

A simple Script example

`2 3 OP_ADD 5 OP_EQUAL`

- How is this evaluated, and what is the result?

- 2 is pushed on the stack. Stack: `2`
- 3 is pushed on the stack. Stack: `2 3`
- OP_ADD removes the top two, adds, and pushes the result. Stack: `5`
- 5 is pushed on the stack. Stack: `5 5`
- OP_EQUAL removes the top two, compares, and pushes ... Stack: **`TRUE`** (=1)

- Script was run successfully with **`TRUE`** as the result!

Another Script example (`unlocking` + `locking`)

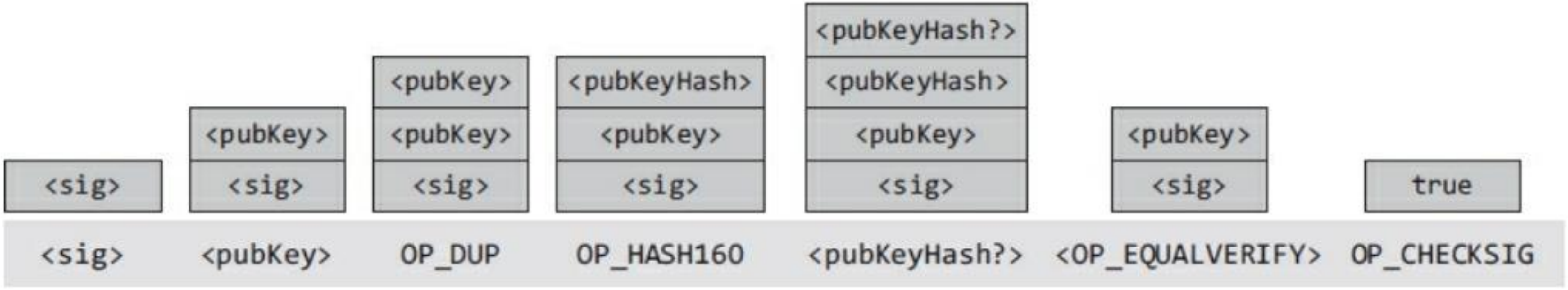
```
<MySignature> <MyPublicKey> OP_DUP OP_HASH160
<MyPublicKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

Execution:

```
<MySignature>---> <MySignature>
<MyPublicKey>---> <MySignature> <MyPublicKey>
OP_DUP       ---> <MySignature> <MyPublicKey> <MyPublicKey>
OP_HASH160   ---> <MySignature> <MyPublicKey> Hash(<MyPublicKey>)
<MyPubl...>  ---> <MySignature> <MyPublicKey> Hash(<MyPublicKey>) <MyPublicKeyHash>
OP_EQUALVERIFY ---> <MySignature> <MyPublicKey>
OP_CHECKSIG  ---> TRUE
```

After I did the previous slide, of course I found

- A much nicer presentation of the stack:



Locking and unlocking scripts

- The example we saw is called a Pay-to-Public-Key-Hash or “P2PKH” script.
- It is by far the most common script.
- Verification was done by:
    1. Matching the public key with the "address" from the referenced output
    2. Checking that the signature is valid
- Exactly like I told you :)
- But using a script language for locking and unlocking outputs opens up for more sophisticated ways of spending the money, f.ex.:
    - Multisignature scripts
    - Escrow transactions
    - Smart contracts

pause

Collecting transactions into a block

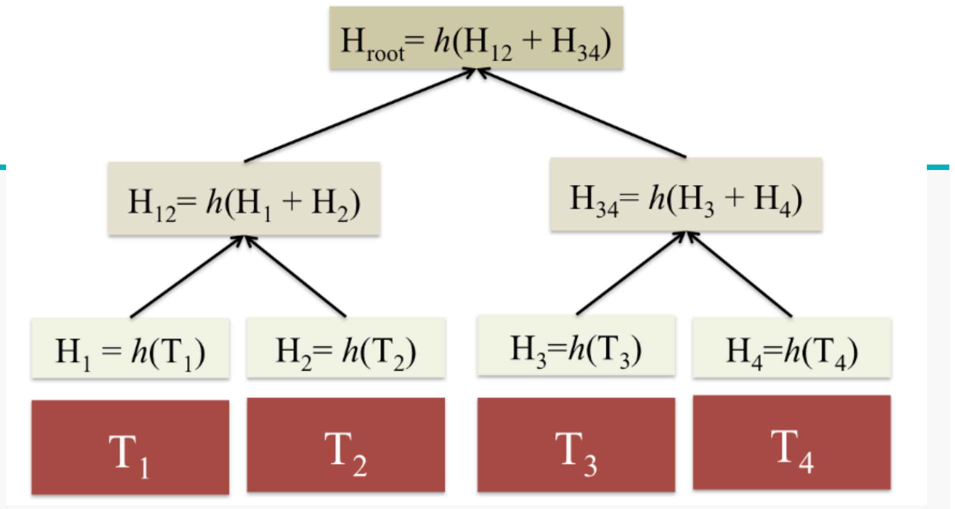
- Now, we will look more in detail on how a Bitcoin block looks like

- We know from the introductory video that a block contains a **prev**, a **nonce**, some **data**, and a **hash**.
- In Bitcoin, the **data** consists of all the **transactions**, maybe more that a thousand transactions in a block.
- How are they stored, and how do we keep track of all the transactions in an efficient way?
- (The blockchain can be stored as a flat file, or in a simple database. The Bitcoin Core client stores the blockchain metadata using Google's LevelDB database.)

The structure of a Bitcoin block

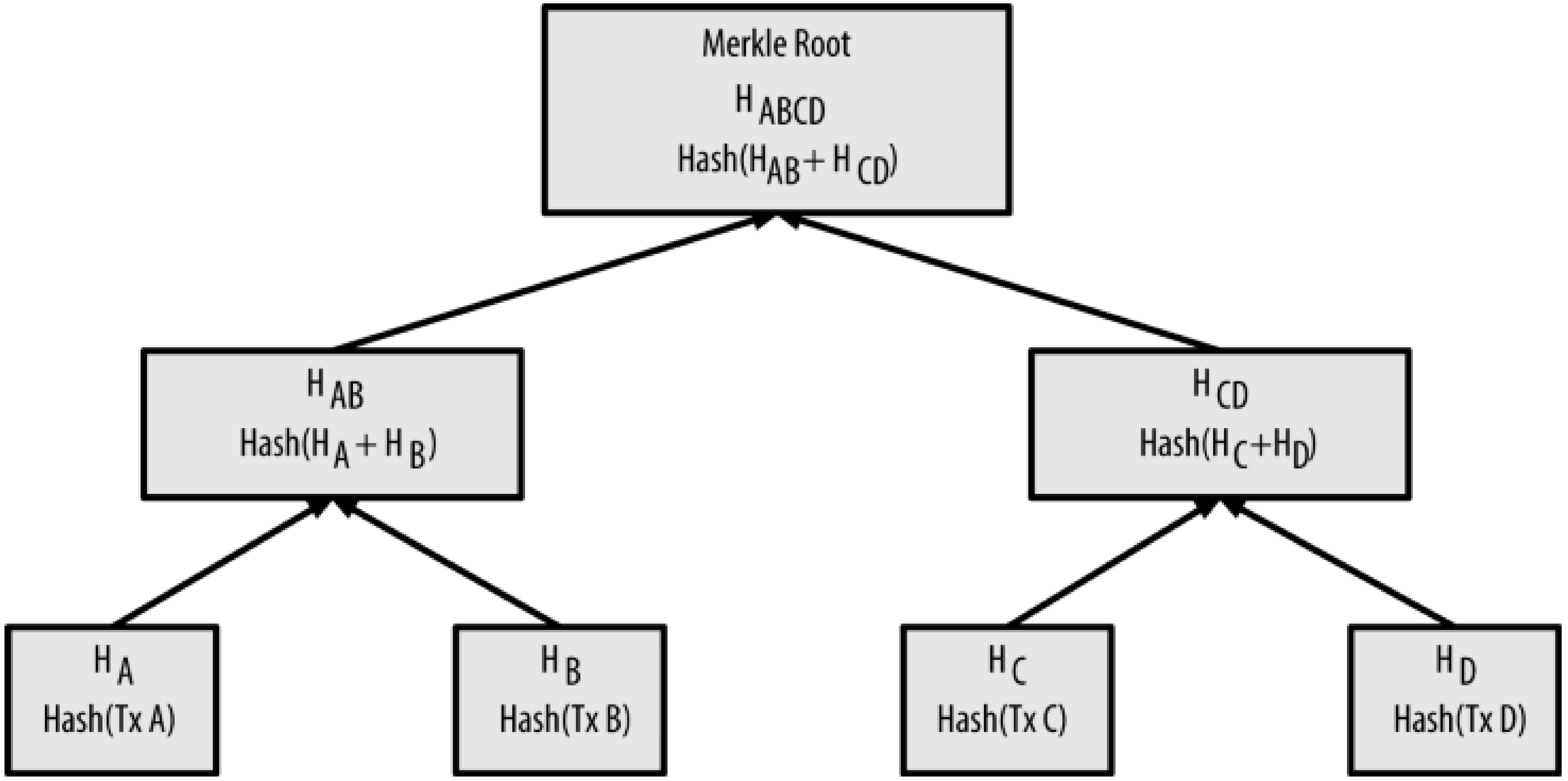
- Due to the amount of data, a block is divided into a **block header**, and a block body. **The block header contains a summary of what is in the block**:
  - Previous Block Hash (as before)
  - Difficulty (related to mining)
  - Timestamp (related to mining)
  - Nonce (as before)
  - **Merkle tree root (a "summary" of all the transactions)**
- The block body stores all the transactions
- Not stored directly, but kept for lookup by nodes:
  - **Block Hash !!!** (the hash of the block header, the unique identifier)
  - Block Height (the block number in the the chain, starting with 0)
- So, what is this Merkle tree root?

Merkle trees

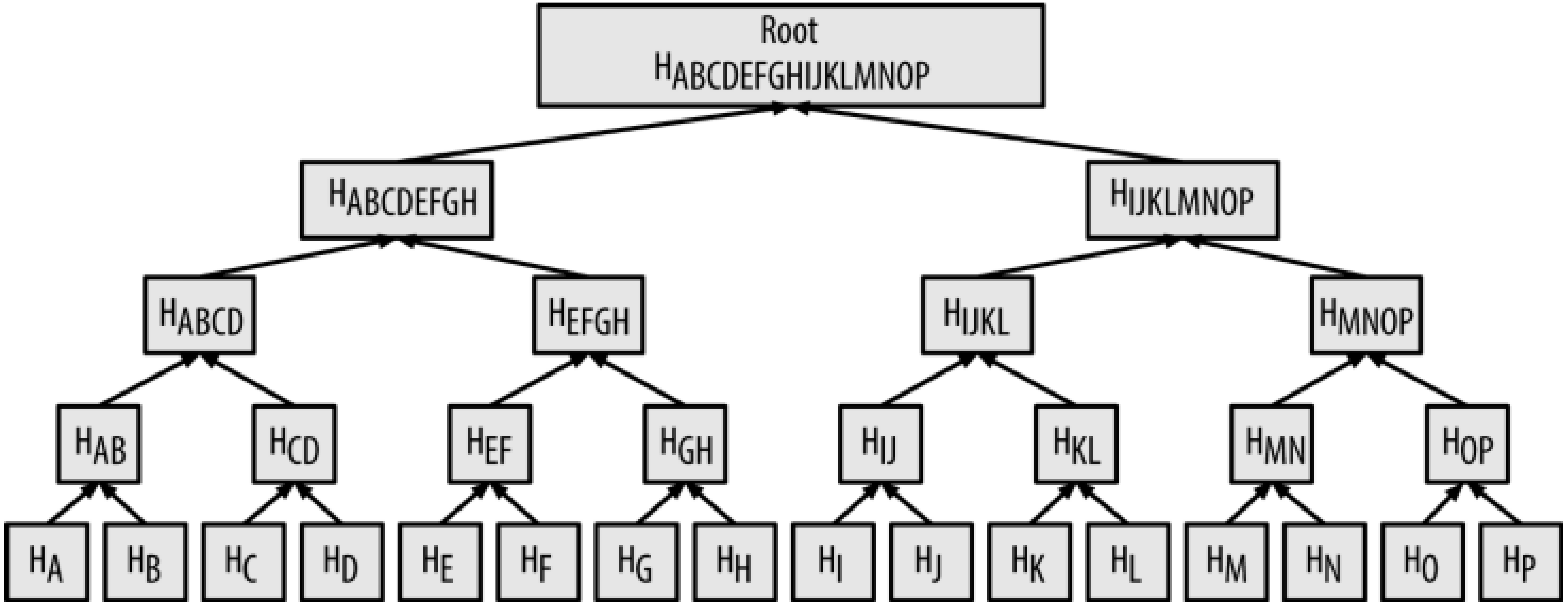


- A Merkle tree is a kind of binary tree that uses hash pointers. The nodes contain the hash values of the hashes of its children.
- The root is a "fingerprint" of all the data used for the leaf nodes.
- In Bitcoin, in stead of basing the block hash on all the transaction data (can be much), it is based on the Merkle root of the Merkle tree of transactions.
- It also enables fast verification of inclusion or exclusion of a transaction in a block.

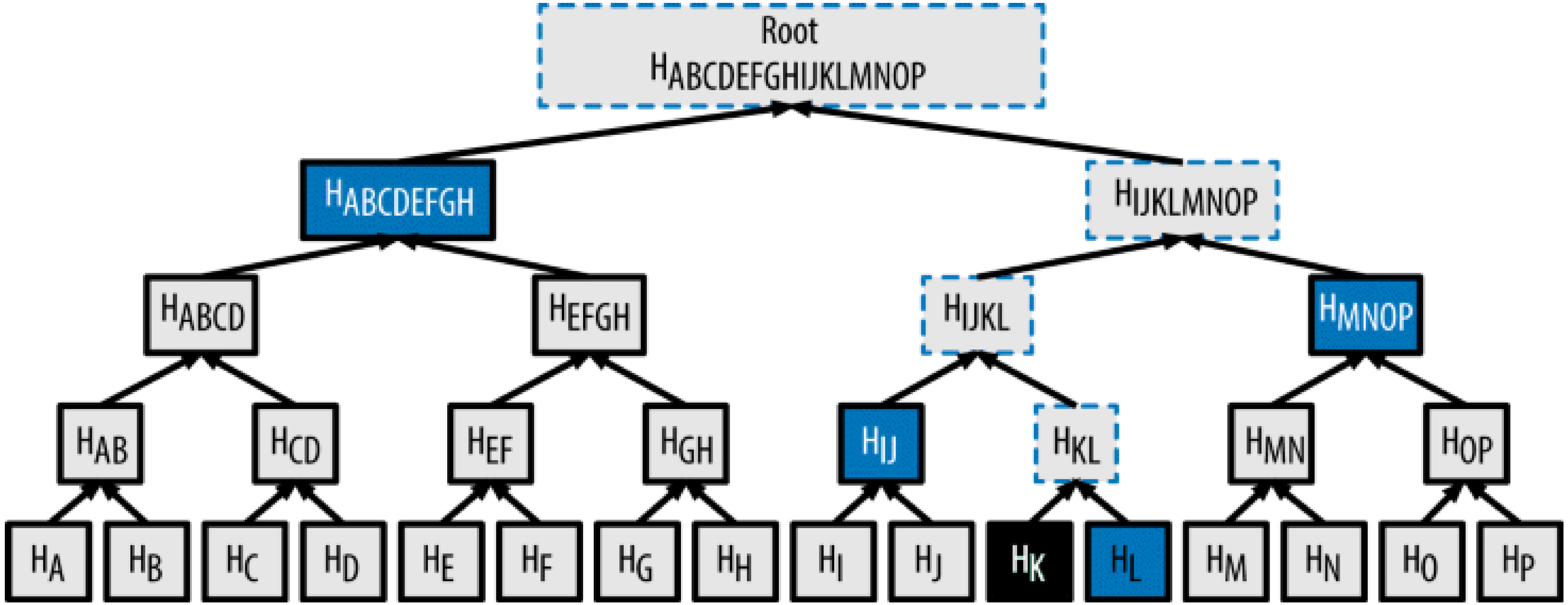
Calculating the nodes



A larger tree

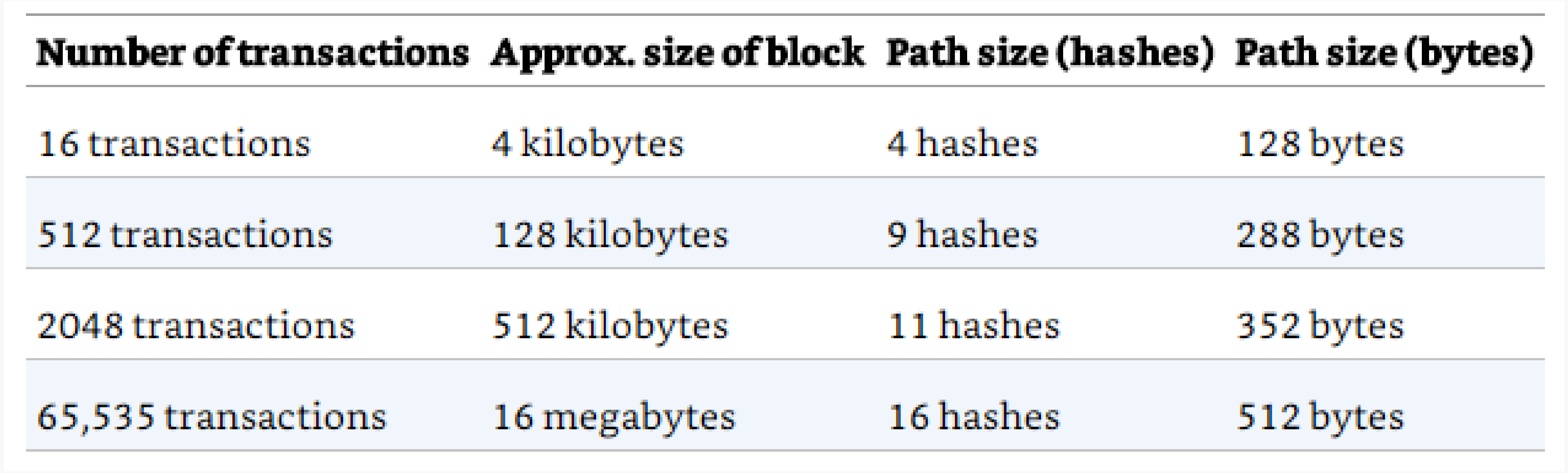


Proving inclusion - Merkle path



Merkle tree efficiency

- Searching for inclusion is O(log(n))
- The table below shows clearly how efficient it is



Number of transactions	Approx. size of block	Path size (hashes)	Path size (bytes)
16 transactions	4 kilobytes	4 hashes	128 bytes
512 transactions	128 kilobytes	9 hashes	288 bytes
2048 transactions	512 kilobytes	11 hashes	352 bytes
65,535 transactions	16 megabytes	16 hashes	512 bytes

Merkle tree summary

- A Merkle tree is a binary tree using hash pointers to "sum up" what is in the input of all the leaf nodes.
- A Merkle (tree) root contains a combined fingerprint for all the leaf nodes.
- Properties:
  - The root is a compact representation of a large amount of transactions
  - If any of transactions are modified, added or removed, the Merkle root will not be valid, and must be recalculated.
  - Proving inclusion of a transaction i done in O(log(n)) time / space.
- Bitcoin blocks use the Merkle root as part of the (mined) block header.

Simplified Payment Verification (SPV)

- A lightweight node (like a mobile wallet) does not download full blocks, just block headers.
- On request, it can download a Merkle path associated with a transaction related to a specific address.
- The lightweight node can then:
  - Use the Merkle path to verify that the transaction is in the block
  - Use the Block header to link the block to the rest of the blockchain
- The amount of data needed for such a verification is typically 1kB, whereas the data in a block is 1MB. So a thousand times less.

Summary of the Bitcoin block structure

- A block is divided into a **block header**, and a block body (containing the transactions). The block header contains a summary of what is in the block:
  - Previous Block Hash (as before)
  - Difficulty (related to mining)
  - Timestamp (related to mining)
  - Nonce (as before)
  - **Merkle tree root (a "summary" of all the transactions)**
- The block body stores all the transactions
- Not stored directly, but kept for lookup by nodes:
  - Block Hash (the hash of the block header, the unique identifier)
  - Block Height (the block number in the the chain, starting with 0)

Creating blocks - Mining

- **Miners aggregate transactions into blocks** and then "mines" the blocks.
- If a miner is successful in finding a nonce that makes the block hash match (is less than) the mining target, the block is sent to the rest of the network to be included in the blockchain.
- **The network nodes will (as with incoming transactions) validate an incoming block** before it is added to the blockchain.
- The checklist for block validation includes the checklist for each transaction, but there are more things to check.

Validating blocks and appending to the blockchain

- The block data structure is syntactically valid
- The block header hash is less than the target (enforces the Proof-of-Work)
- The header info, including the Merkle root, must calculate correctly
- There must not be duplicate transactions in the block
- No inputs must be included in more than one transaction
- The first transaction (and only the first) is a coinbase transaction
- The block size is within acceptable limits
- All transactions within the block are valid using the transaction checklist discussed earlier

Bridging Lab1 and Lab2 ??? (if you have extra time)

- In Lab1 you saw how we can build a simple blockchain (top-down)
- In Lab2 you saw (will see) how you create valid transactions (bottom-up)

- What needs to be done to bridge the two?

  - You need to define a slightly different block structure (including transactions and the Merkle root)
  - You need to implement a Merkle root calculation
  - You need to implement proper block validation

Networking, API and applications

- The "elephant" in the room now is of course the networking part. So far, we have only looked at a centralized solution running on one node.
- **But we are out of time. :(**
- If we had three more weeks, I think a goal could be to make a small simple peer-to-peer-network and a DAT159Coin.
- And if we had another three more weeks, we could have made a simple block explorer and a wallet. …..

Next

- Oblig3 ...
- Next week we will look other aspects of Blockchain Technology.
  - Blockchain applications ++
  - Smart contracts ++