CSE446: Blockchain & Cryptocurrencies

Lecture – 9: Bitcoin-3

Agenda

- Bitcoin components
  - Bitcoin Blockchain
    - Transaction
    - Block

Transaction input

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f
    "sequence": 4294967295
  }
]
```

https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc

- Transaction inputs identify (by reference) which UTXO will be consumed and provide proof of ownership through an unlocking script (scriptSig)
- To build a transaction
  - a wallet selects from the UTXO it controls, UTXO with enough value to make the requested payment
  - Sometimes one UTXO is enough, other times more than one is needed
  - For each UTXO that will be consumed to make this payment, the wallet creates one input pointing to the UTXO and unlocks it with an unlocking script (scriptSig)

Transaction input

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f
    "sequence": 4294967295
  }
]
```

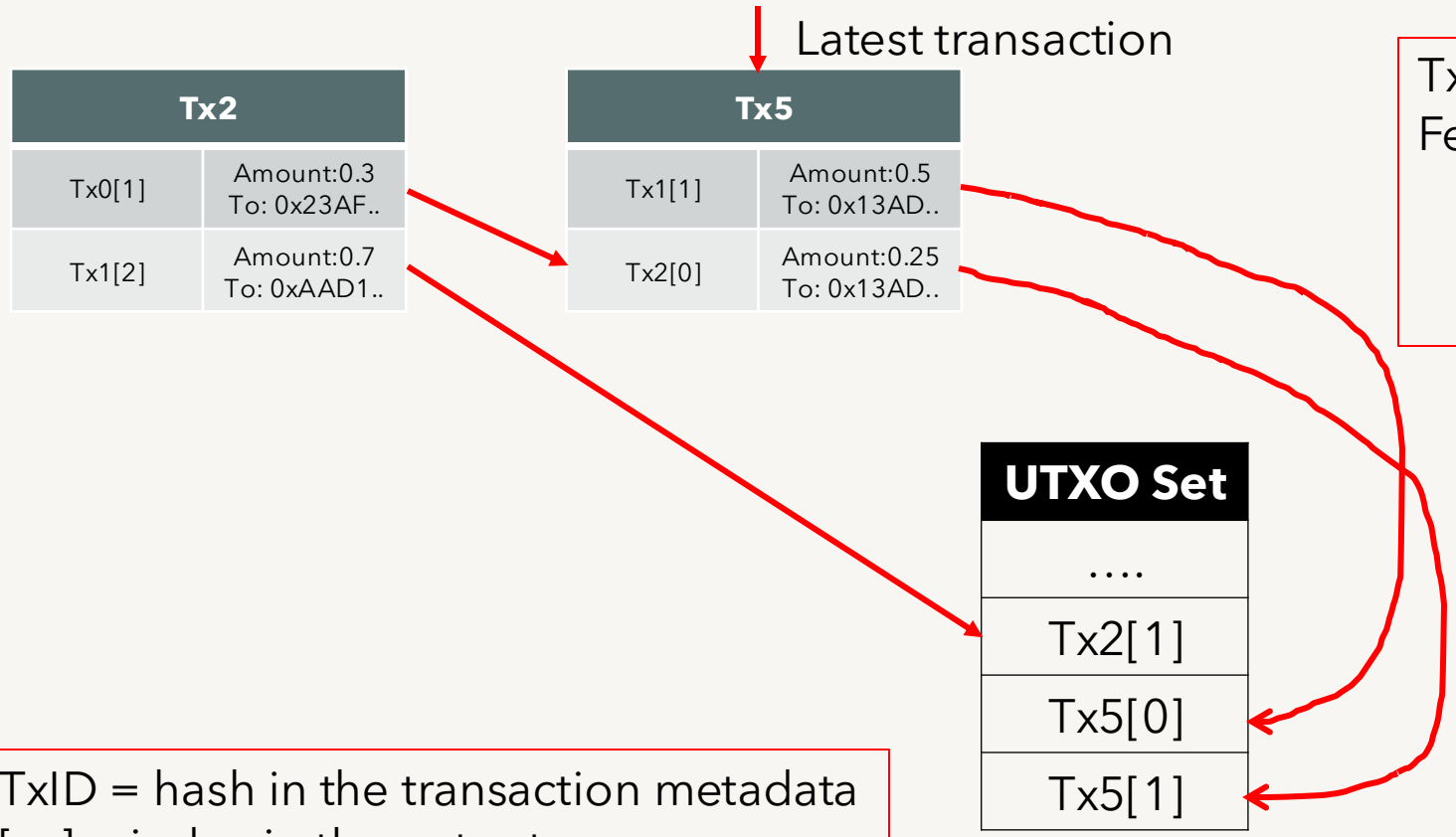
https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc

- The input contains three main elements:
  - A transaction ID, referencing the transaction that contains the UTXO being spent
  - An output index (vout), identifying which UTXO from that transaction is referenced (first one is zero)
  - A *scriptSig*, which satisfies the conditions placed on the UTXO, unlocking it for spending

Transaction id & fee

- The data structure of transactions does not have a field for fees
- Instead, fees are implied as the difference between the sum of inputs and the sum of outputs
- Any excess amount that remains after all outputs have been deducted from all inputs is the fee that is collected by the miners:
  - Fees = Sum(Inputs) – Sum(Outputs)
- Each transaction is identified by an identifier, called *transaction id* (TxID)
- TxID is created by double hashing the serialized all inputs and outputs and other data within the transaction
  - id = SHA256(SHA256(serialized input + output + other data))

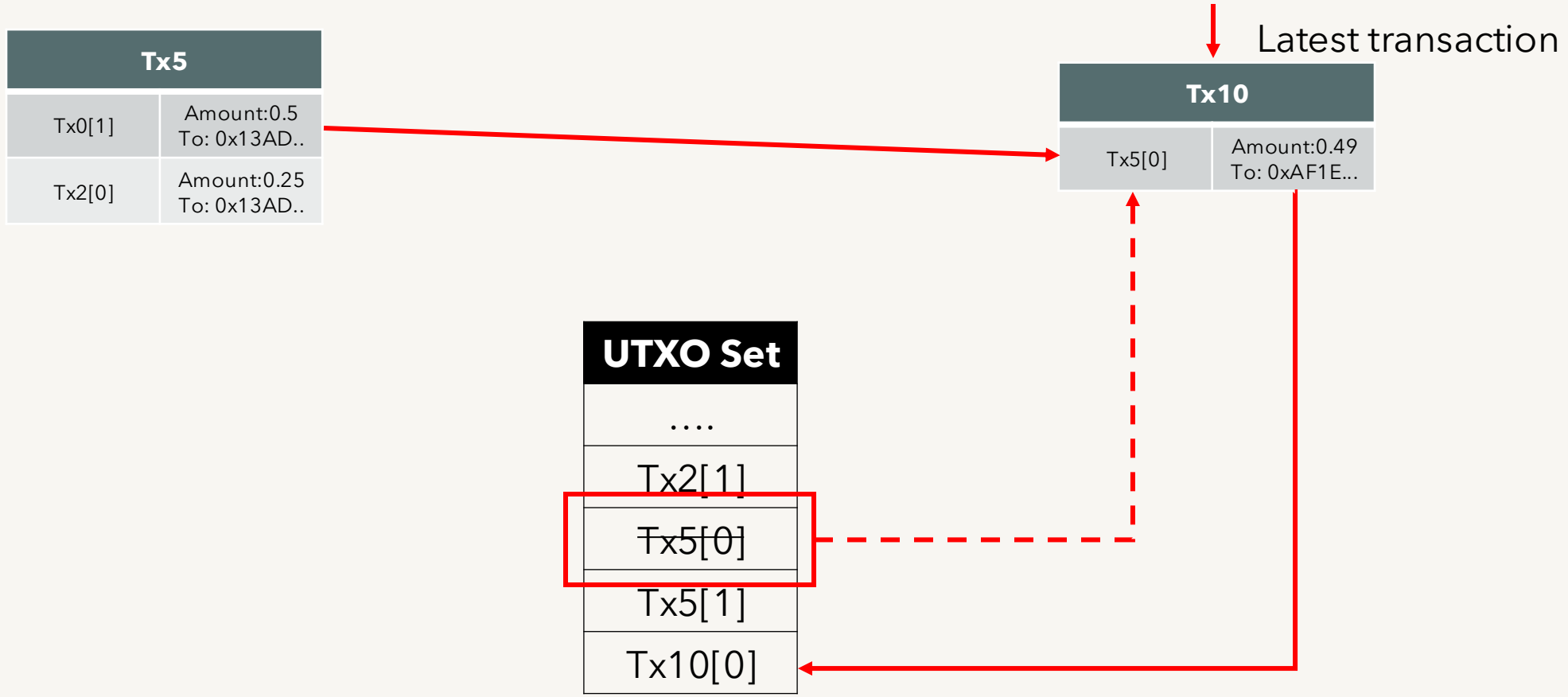
Transaction



Tx1[1] = 0.5, Tx2[0] = 0.3
Fee = sum(input) – sum(output)
= (0.5 + 0.3) – (0.5 + 0.25)
= 0.8 – 0.75
= 0.05

TxID = hash in the transaction metadata
[…] = index in the output

Transaction



Bitcoin script

- Every transaction input and output use the scripting capability in Bitcoin
- Bitcoin script allows a user to impose a restriction (condition) as to which user can use it
- scriptPubkey is called the locking script as it locks the value in the output to a certain bitcoin address
  - Only who can prove that they own that address can use this bitcoin

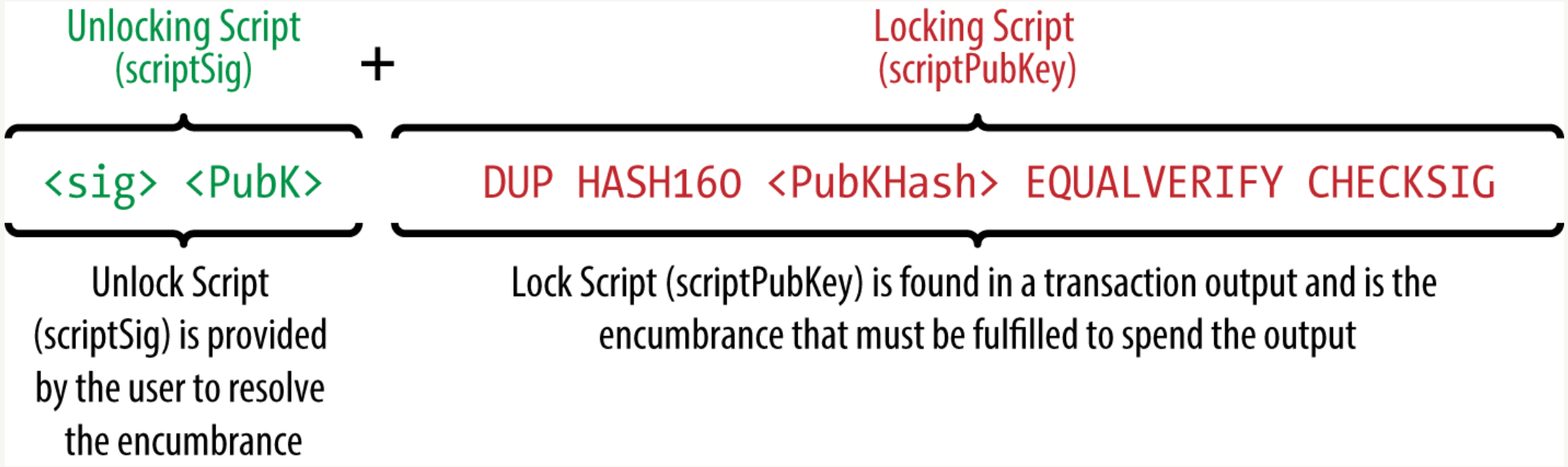
https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc

Bitcoin script

- When someone would like to use a tx output, they need to use it as a transaction input
- A proof is needed to claim the ownership of the tx input
- scriptSig is the required proof
- scriptSig contains a digital signature which unlocks the locking script
  - Digital signature proves the ownership over the locked Bitcoin address

https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc

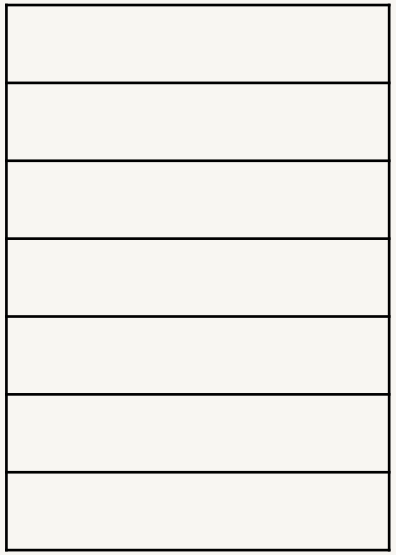
Bitcoin script locking + unlocking



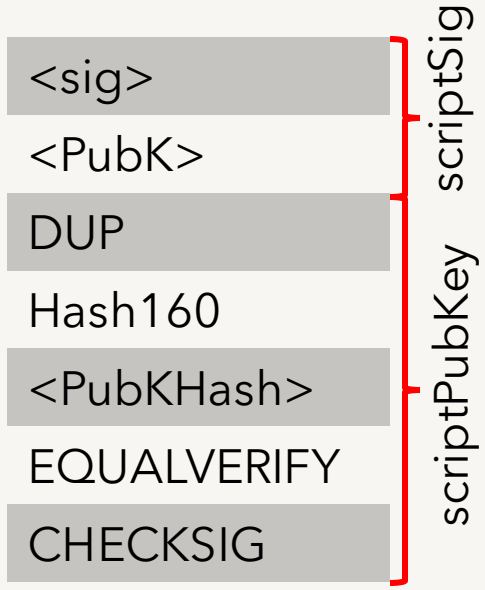
Move from left to right

Source: http://chimera.labs.oreilly.com/books/1234000001802/ch05.html

Bitcoin script locking + unlocking

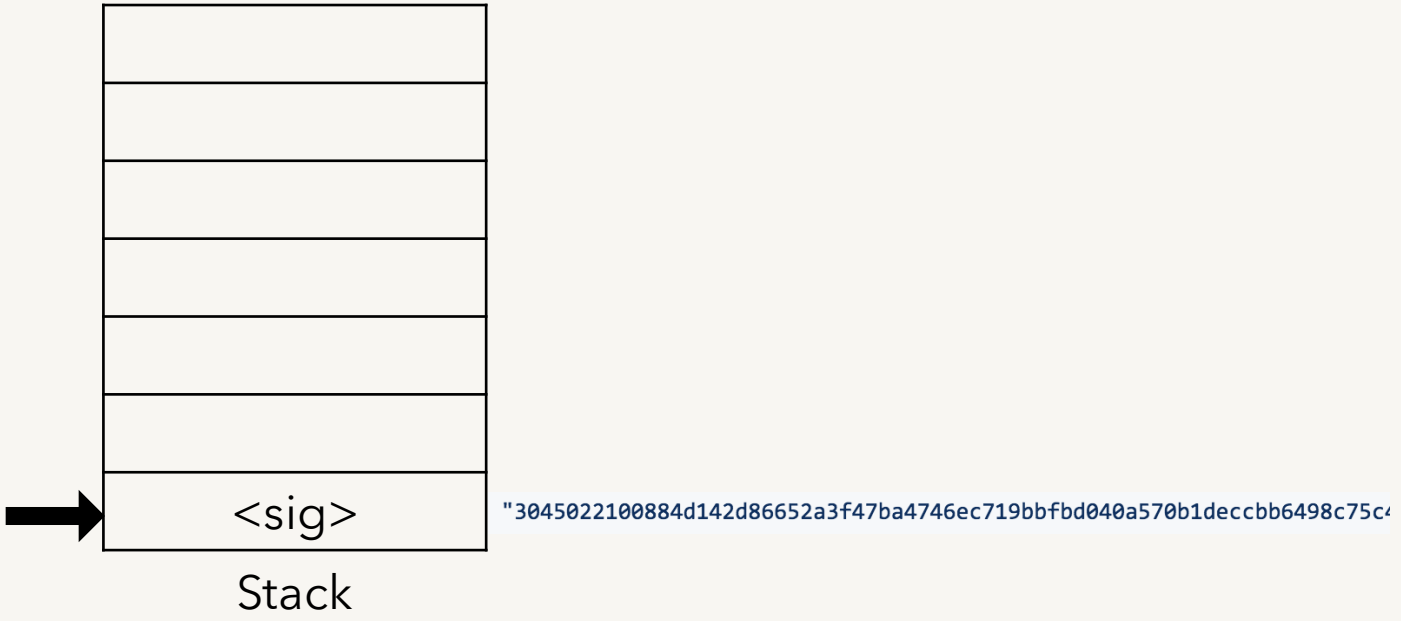


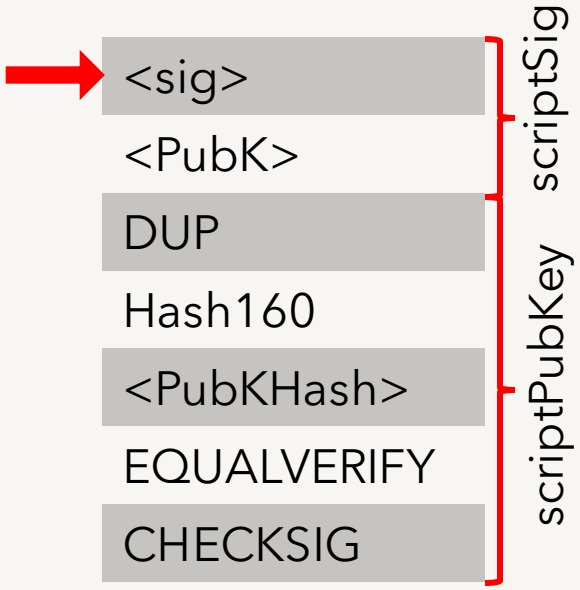
Stack



- Bitcoin uses a stack as presented in the left
- Locking and unlocking script are placed as shown in the right
- The script is executed line by line, top to bottom for this figure
- Explanation is added here

Bitcoin script locking + unlocking

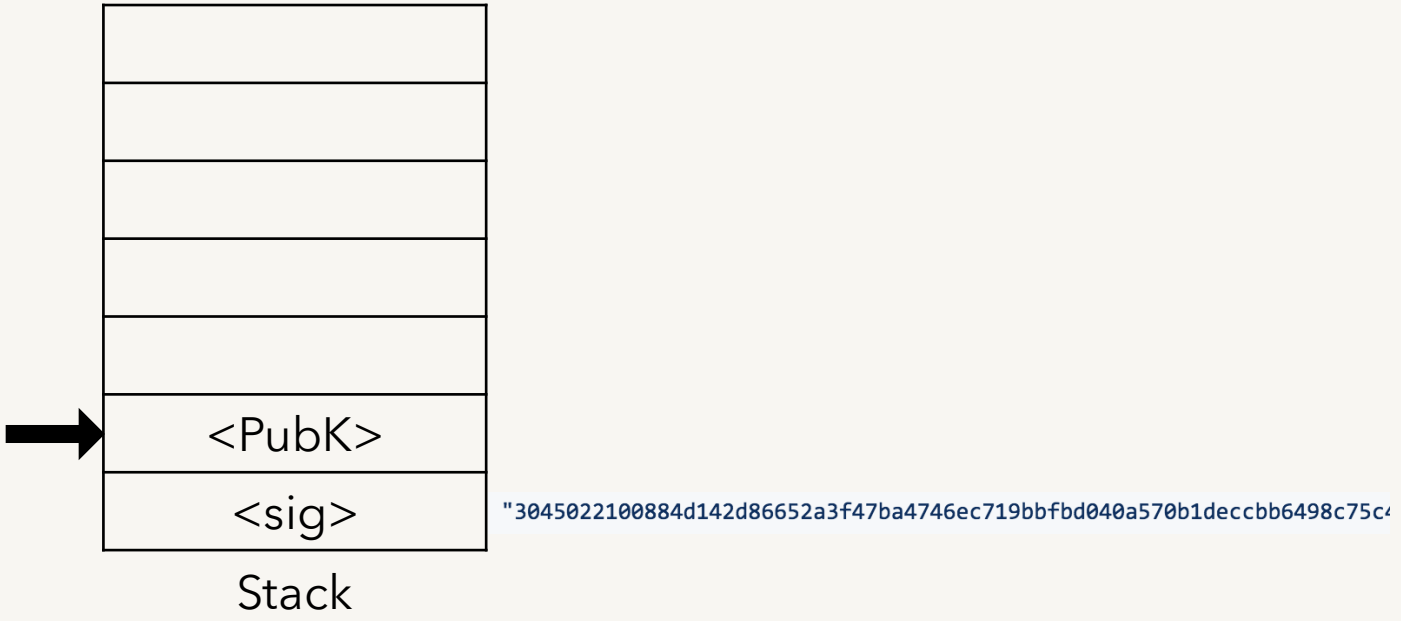




Current command: <sig> represents the signature in the input

- The signature from the input is pushed in the stack
- The black arrow represents the current top item in the stack

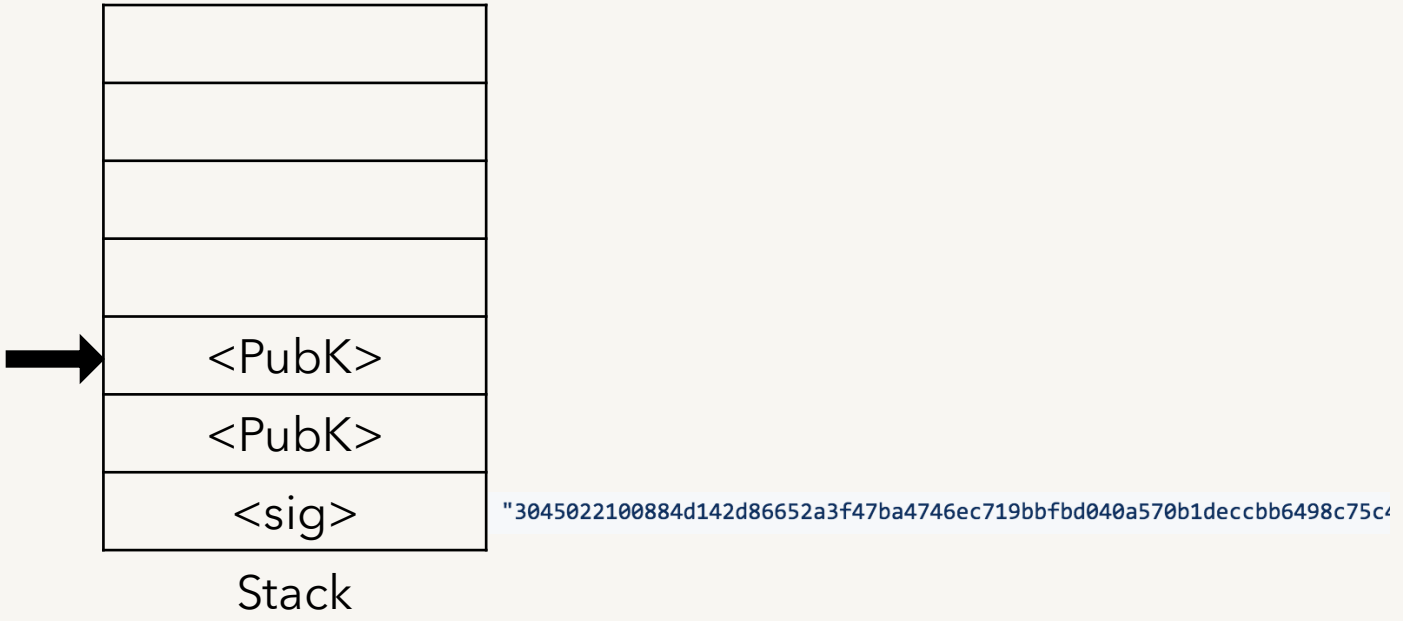
Bitcoin script locking + unlocking

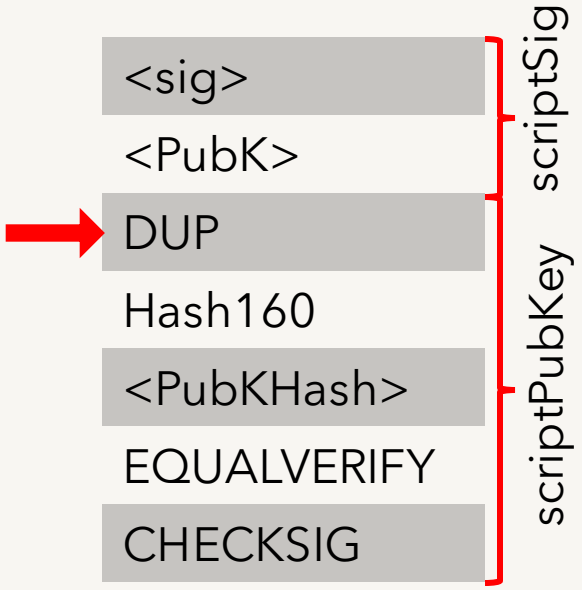


Current command: <PubK>

- The public key of the input is pushed in the stack

Bitcoin script locking + unlocking





Current command: <DUP>

- The DUP command duplicates the value from the top of the stack and pushes it in the stack
- In real bitcoin script, <DUP> is represented with <OP_DUP>

Bitcoin script locking + unlocking

Current command: <Hash160>

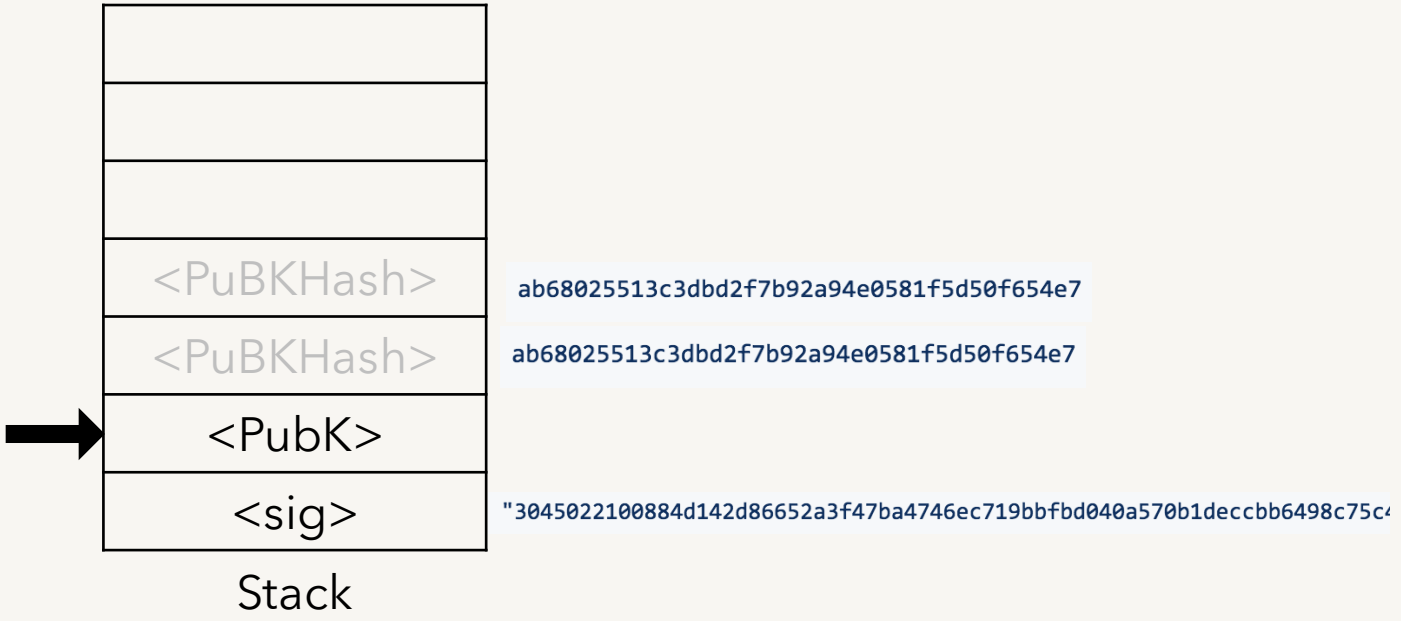
- The Hash160 command pops the top item from the stack, creates its hash and pushes the hash back in the stack
- In real bitcoin script, <Hash160> is represented with <OP_Hash160>

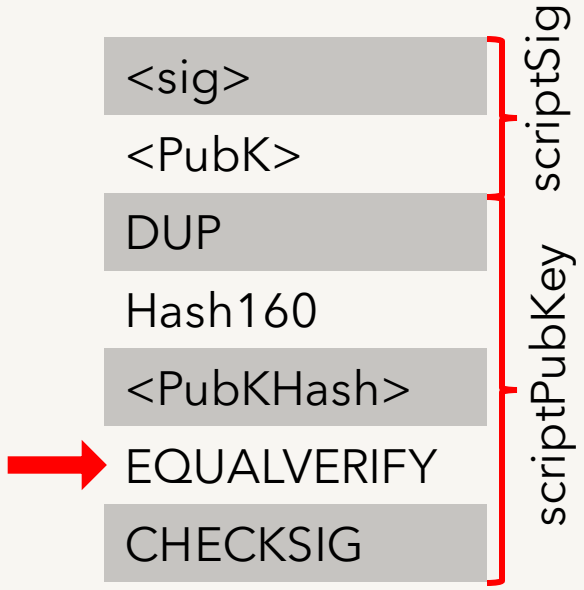
Bitcoin script locking + unlocking

Current command: <PubKHash>

- The public key hash specified in the tx output is pushed to the stack

Bitcoin script locking + unlocking

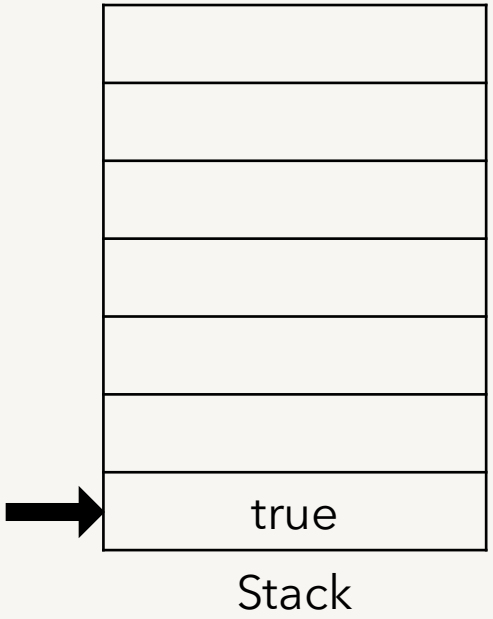


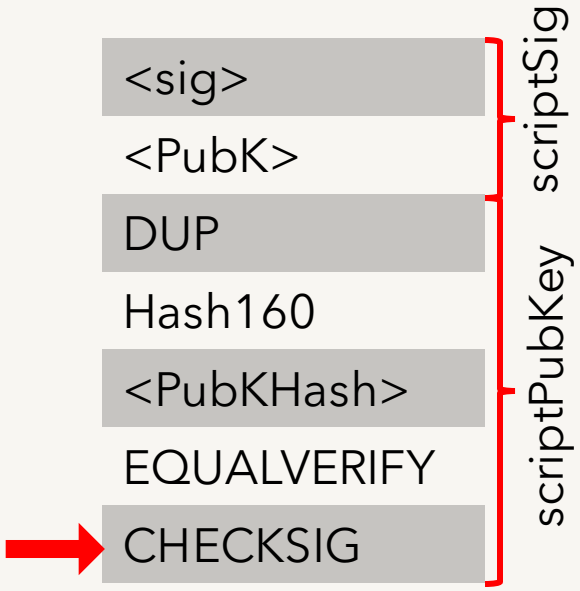


Current command: <EQUALVERIFY>

- This command checks if the top two items in the stack are equal or not. If not equal, an error is thrown. Otherwise two items are popped from the stack
- In real bitcoin script, <EQUALVERIFY> is represented with <OP_EQUALVERIFY>

Bitcoin script locking + unlocking





Current command: <CHECKSIG>

- This command pops two items from the stack and verifies the signature using the public key. If the verification is successful, a true value is pushed otherwise a false value is pushed. In real bitcoin script, <CHECKSIG> is represented with <OP_CHECKSIG>

Transaction verification

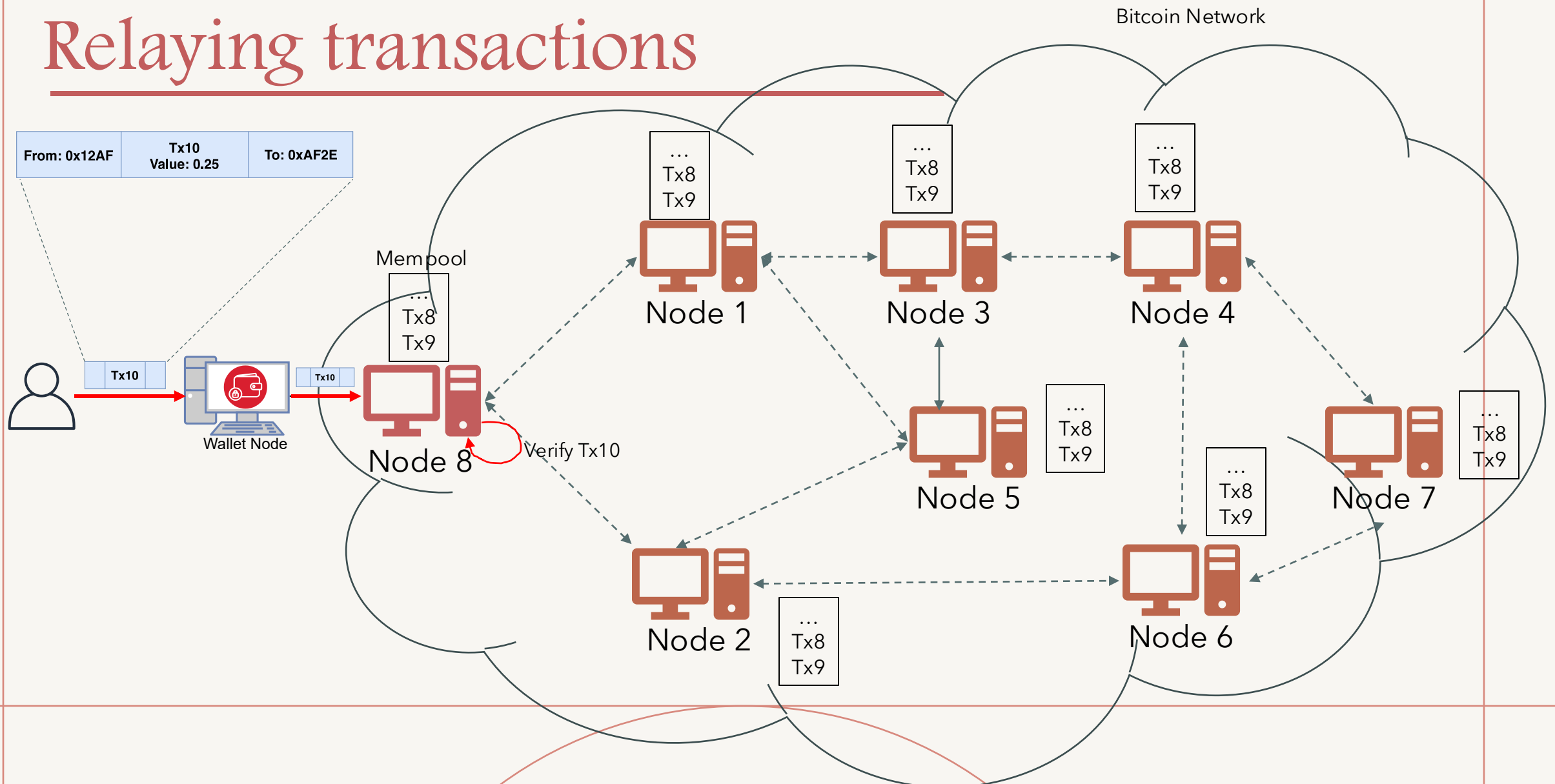
- When a node receives a transaction, it verifies the transaction with a number of steps
  - Check syntactic correctness
  - Make sure neither in (except for a coinbase tx) or out lists are empty
  - Reject if we already have matching tx in the pool, or in a block in the blockchain
    - Every node maintains a transaction pool called mempool, consisting of txs not inserted in a block yet
  - For each input, if the referenced output does not exist (e.g. never existed or has already been spent), reject this transaction
  - Reject if the sum of input values < sum of output values

Transaction verification

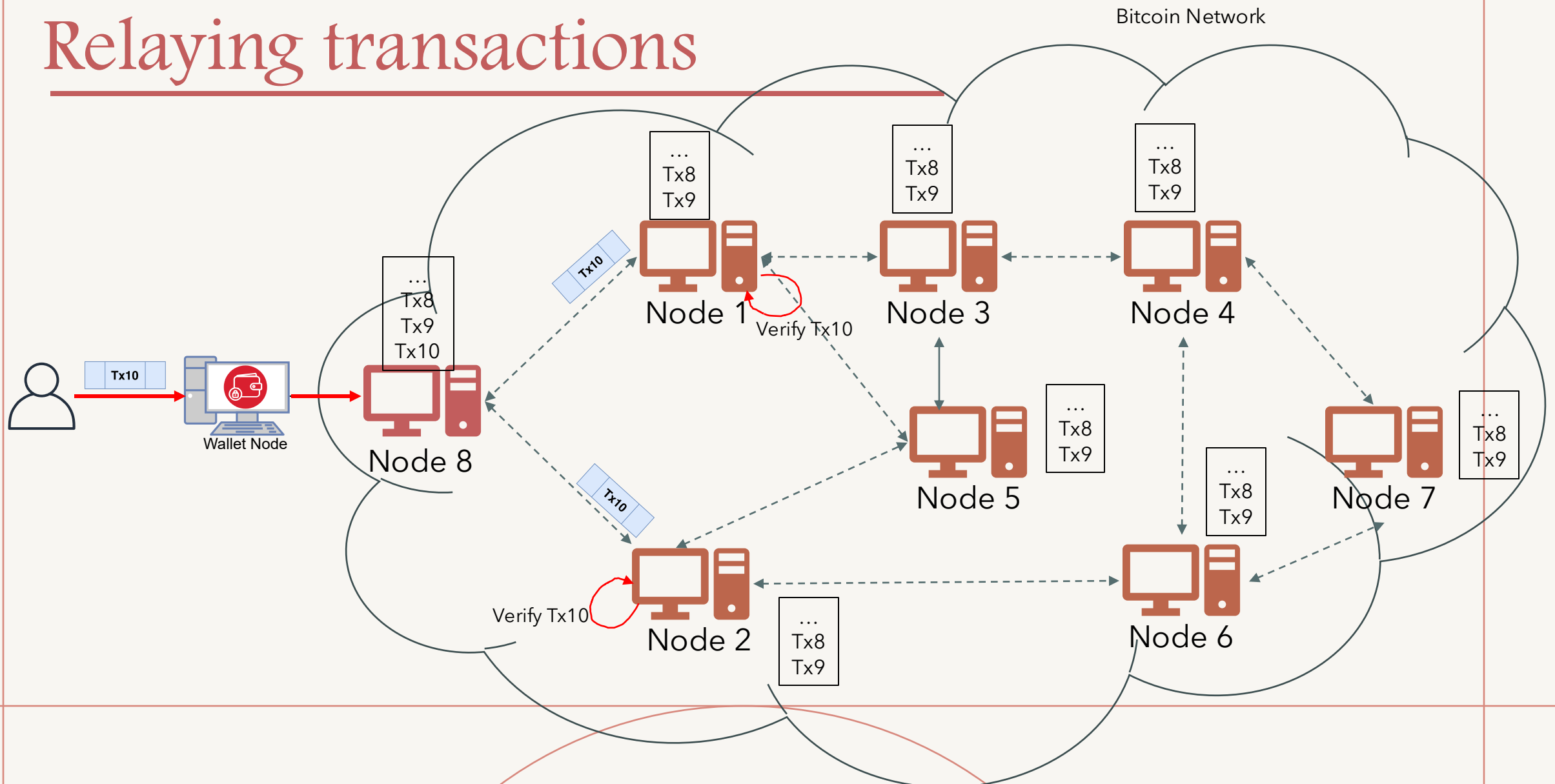
- Reject if transaction fee (defined as sum of input values minus sum of output values) would be too low to get into an empty block
- Verify the scriptPubKey accepts for each input; reject if any are bad

- If the verification is successful:
  - Add it to a transaction pool inside the node (mempool)
  - Add it to the wallet if the output belongs to the addresses controlled by the user's wallet
  - Relay the transaction to peers

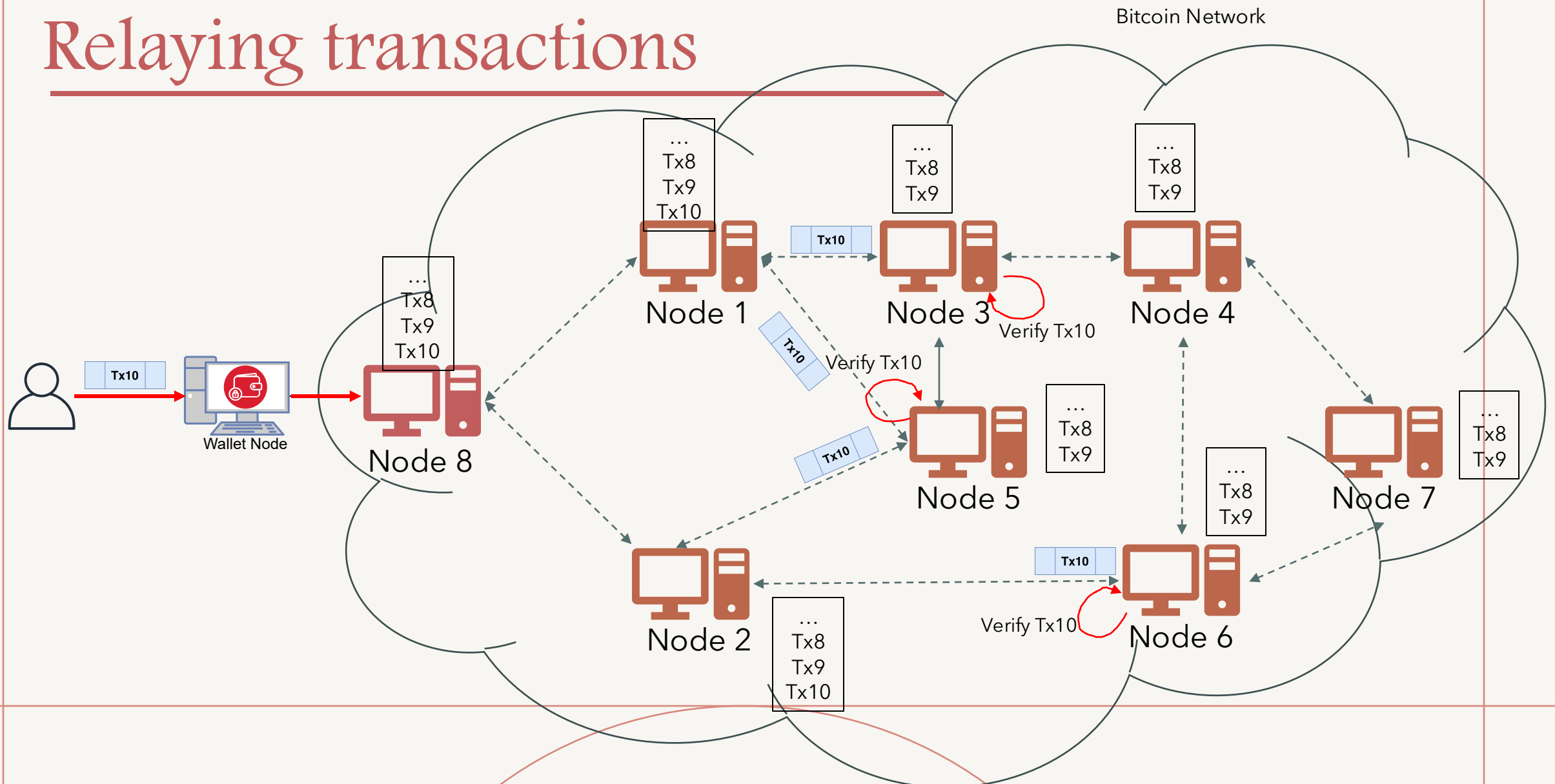
Relaying transactions



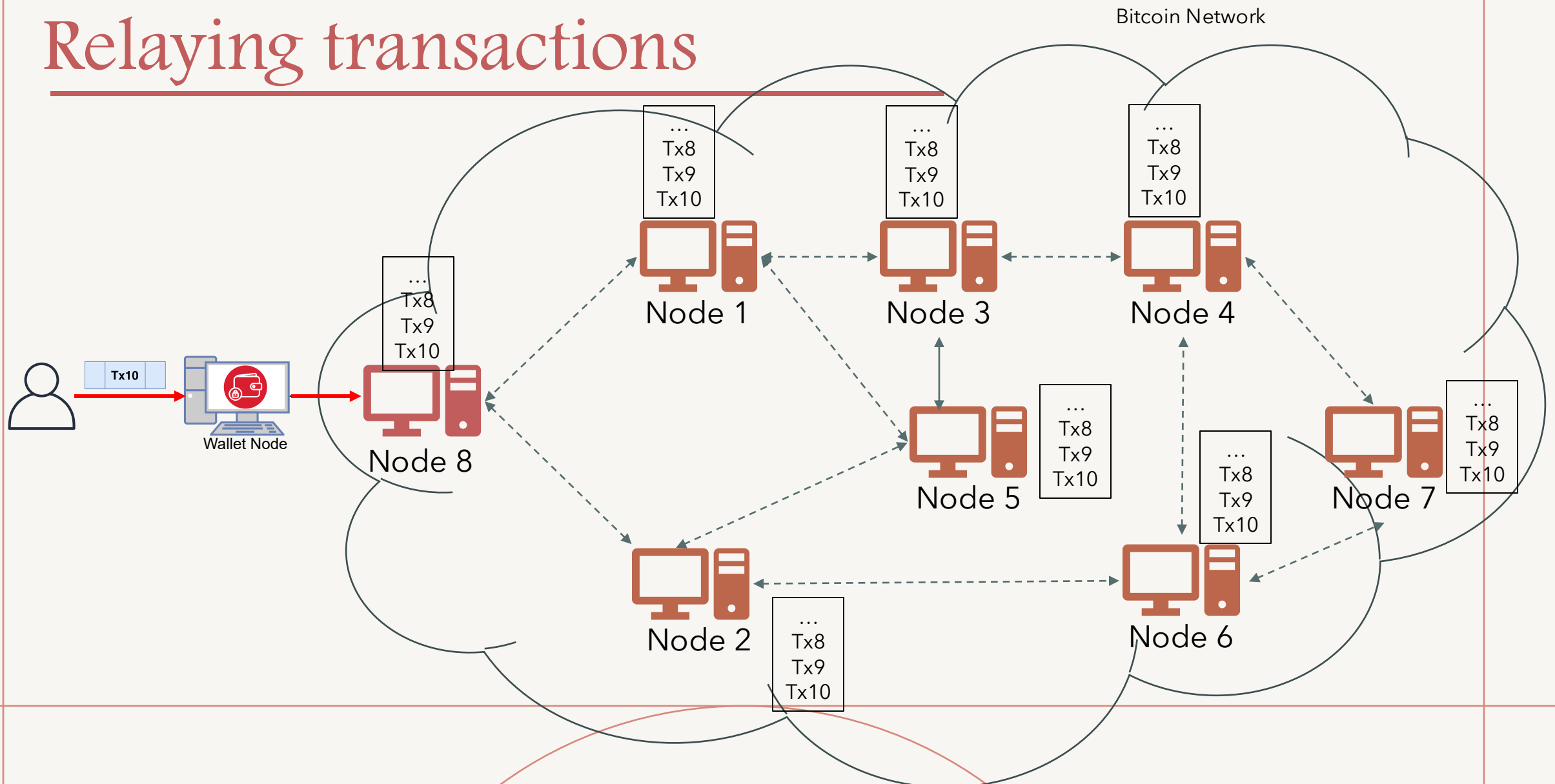
Relaying transactions



Relaying transactions



Relaying transactions



Block

- Relayed transactions are combined into a block
- A newly created block contains the most recent transactions that did not exist in previous blocks
  - Each transaction can appear **only once** in a block
- Each block is also propagated into the network
- Each block is identified by an identifier, called block id
  - The block id is created by double-hashing the block header (discussed later)
- The network is set to create **one block every 10 minutes**
- A transaction in a valid block is called confirmed

Block

Table 1. The structure of a block

Size	Field	Description
4 bytes	Block Size	The size of the block, in bytes, following this field
80 bytes	Block Header	Several fields form the block header
1–9 bytes (VarInt)	Transaction Counter	How many transactions follow
Variable	Transactions	The transactions recorded in this block

https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch09.asciidoc

Block

Table 2. The structure of the block header

Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the merkle tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)
4 bytes	Difficulty Target	The Proof-of-Work algorithm difficulty target for this block
4 bytes	Nonce	A counter used for the Proof-of-Work algorithm

Pointer to the previous block

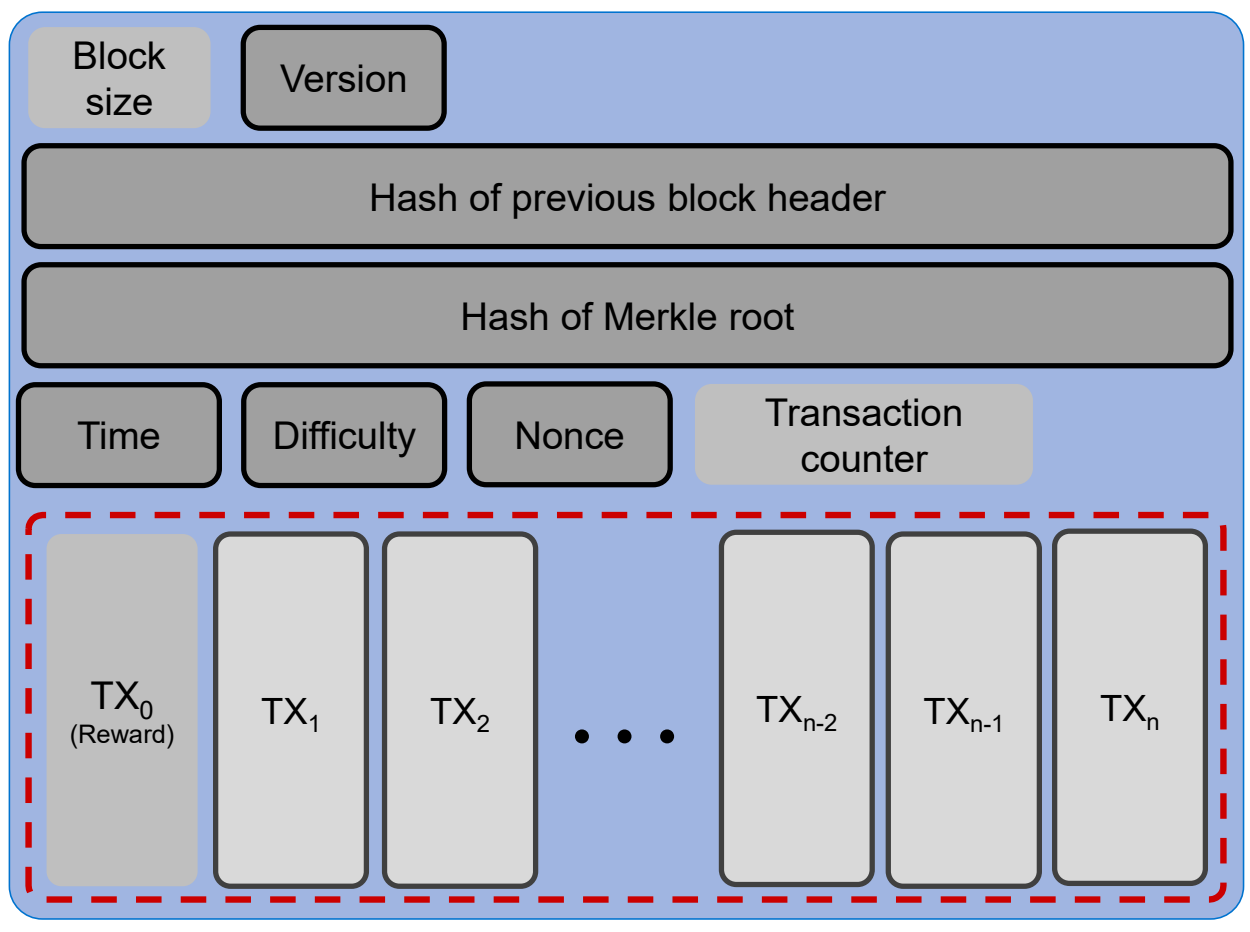
A hashed data structure of all Transactions in the block

Time when this block is created

These two fields are used to achieve consensus, will be discussed later

https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch09.asciidoc

Block

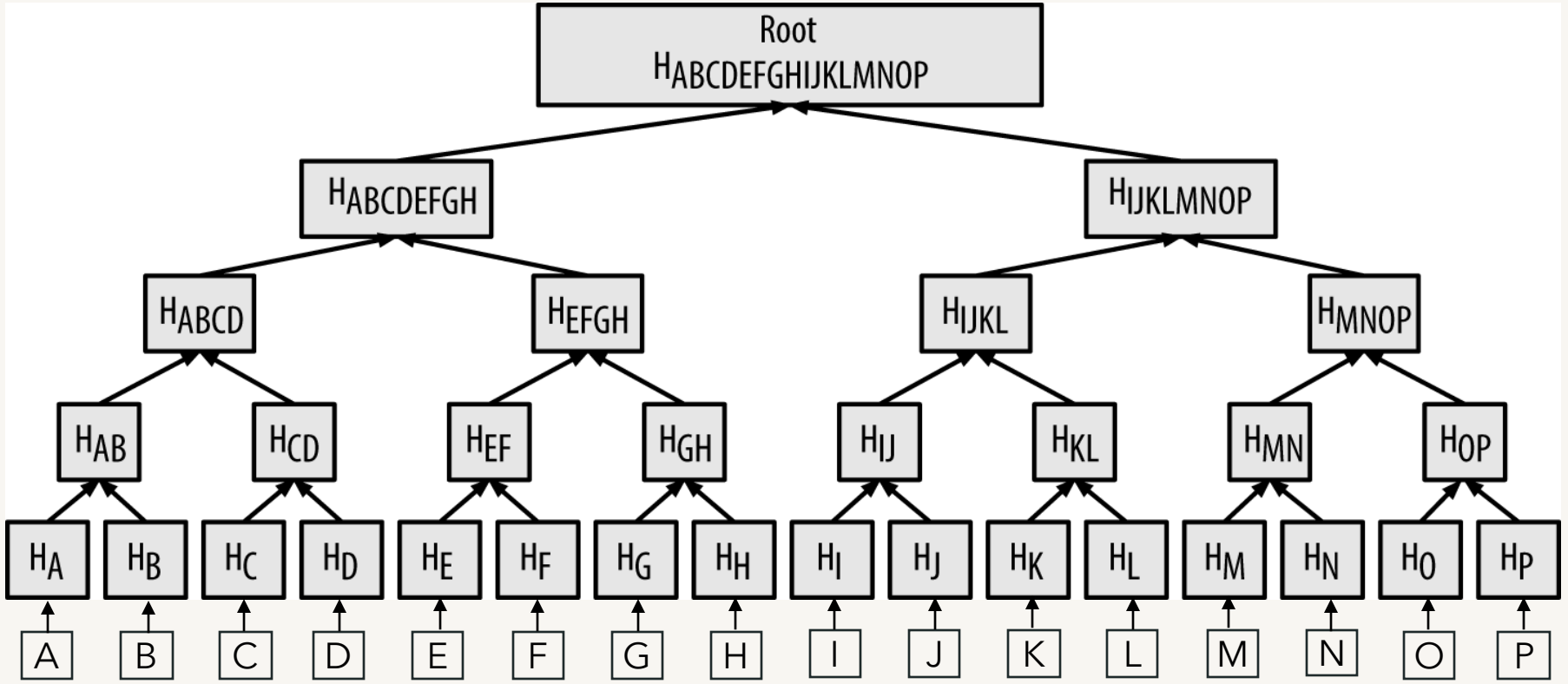


Bitcoin Basics- Gallersdörfer, U., Holl, P., & Matthes, F. (2020). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.

Block

- Merkle trees are used in bitcoin to summarise all the transactions in a block
  - producing an overall digital fingerprint of the entire set of transactions
  - providing a very efficient process to verify the inclusion of a transaction (proof of membership)
- A merkle tree is a balanced binary tree, containing an even number of leaf nodes
  - if a tree needs to be constructed with an odd number of leaf nodes, the last element is duplicated
- It is constructed by recursively hashing pairs of nodes until there is only one hash, called the *root*, or *merkle root*
- The cryptographic hash algorithm used in bitcoin's merkle trees is SHA256 applied twice, also known as double-SHA256

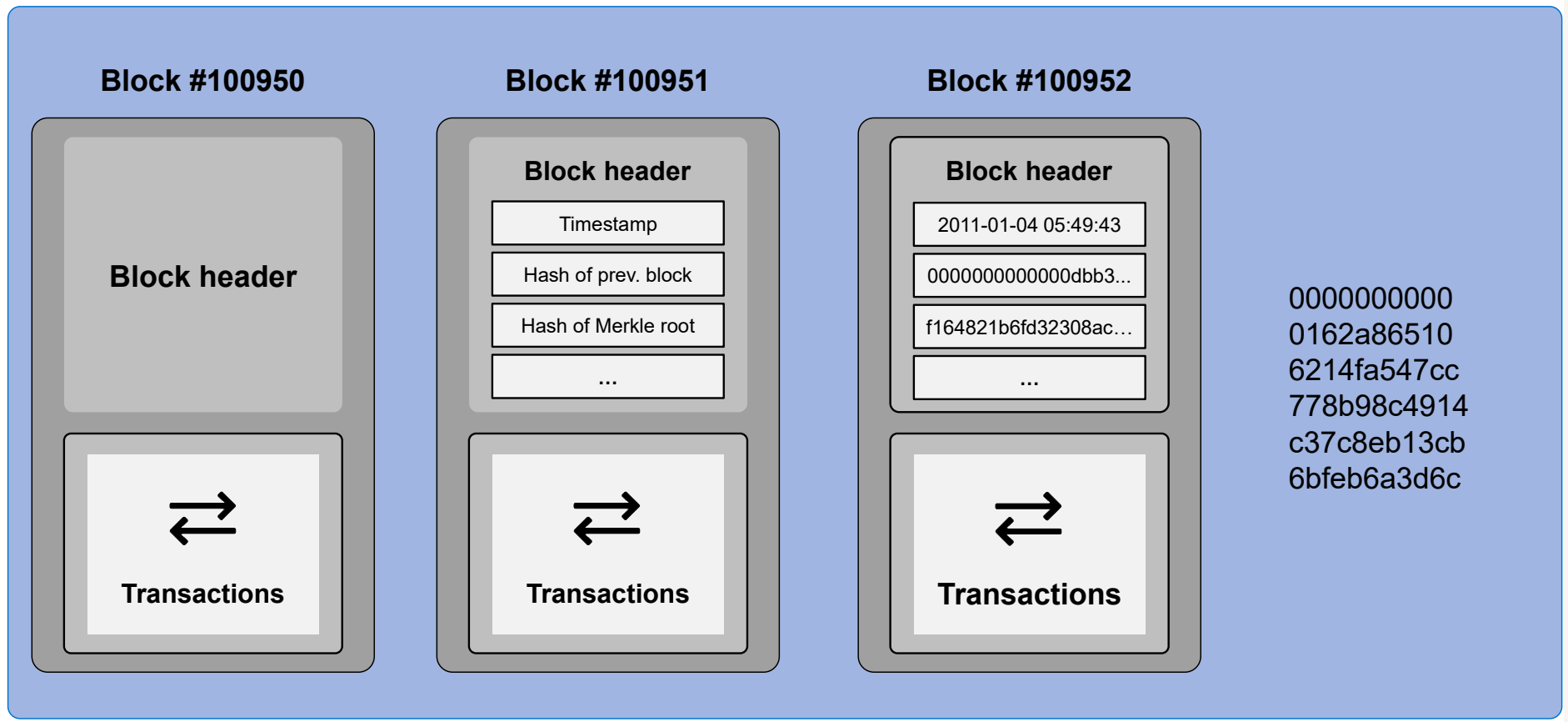
Block



$H_A = SHA256(SHA256(TX_A), H_{AB} = SHA256(SHA256(H_A \| H_B)$

https://raw.githubusercontent.com/bitcoinbook/bitcoinbook/develop/images/mbc2_0904.png

Blockchain



Bitcoin Basics- Gallersdörfer, U., Holl, P., & Matthes, F. (2020). "Blockchain-based Systems Engineering". Lecture Slides. TU Munich.


THANK YOU
QUESTIONS?