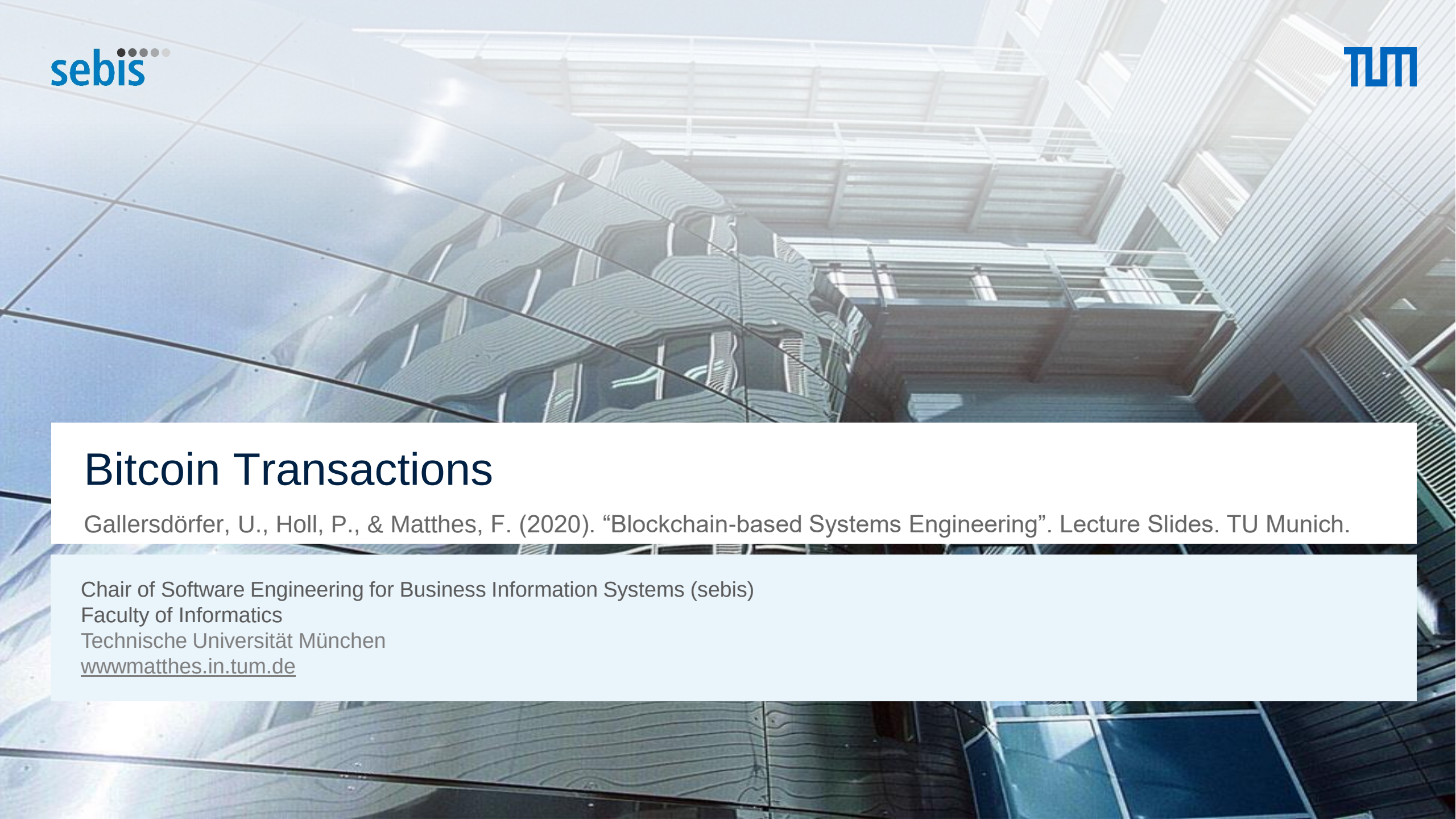


Bitcoin Transactions

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

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Outline

1. Types of ledger
   - Account-based ledger
   - Transaction-based ledger
   - Data structure and Bitcoin script

2. Transactions in Bitcoin
   - Data structure
   - Transaction verification

3. Bitcoin Script

4. Use cases of Bitcoin Script

This chapter is heavily inspired by and uses examples from „Bitcoin and Cryptocurrency Technologies“ by Arvind Narayanan

Account-based ledger

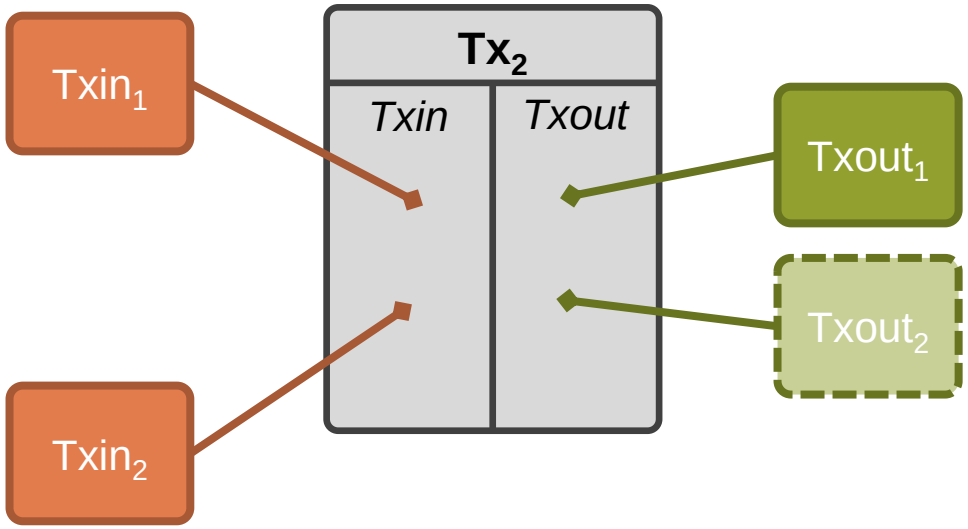
Transactions
Create 25 coins and credit to Alice signed by miners
Transfer 17 coins from Alice to Bob signed by Alice
Transfer 8 coins from Bob to Carol signed by Bob
Transfer 5 coins from Carol to Alice signed by Carol
Transfer 15 coins from Alice to Bob signed by Alice

Alice	Bob	Carol
25	0	0
8	17	0
8	9	8
13	9	3
-2	24	3

World State

- *Intuitively*: We consider Bitcoin to use an account-based ledger. However, an account-based approach takes a lot of effort to track the balances of every account.
- In an account-based ledger, transactions can transfer arbitrary amounts of coins between accounts.
- Transactions lead to a “world-state” of accounts and account balances.
- In Ethereum, the hash of the Merkle root of the Merkle tree of all accounts and their balances is stored in the block.
- By using a transaction-based ledger, Bitcoin enables wallet owners to define conditional transactions using Bitcoin Script.

Transaction-based ledger



- Transactions (**Tx**) have a number of inputs and a number of outputs.
  - **Inputs (Txin)**: Former outputs, that are being consumed
  - **Outputs (Txout)**: New creation of coins

- In transactions where **new coins** are created, **no Txin** is used (no coins are consumed)

- Each transaction has a unique identifier (**TxID**). Each output has a unique identifier within a transaction. We refer to them (in this example) as *#TX[#txout]*, e.g., 1[1], which is the second Txout of the second transaction.

Transaction-based ledger

0	Txin: ∅ Txout: 25.0 → Alice
1	Txin: 0[0] Txout: 17.0 → Bob, 8.0 → Alice signed by Alice
2	Txin: 1[0] Txout: 8.0 → Carol, 9.0 → Bob signed by Bob
3	Txin: 1[1] Txout: 6.0 → David, 2.0 → Alice signed by Alice

Example:

0. No input required, as coins are created.
1. The Tx is used as an Txin. Two Txout are created, one to Bob and one to Alice. (1[0] and 1[1]) The Tx is signed by Alice.
2. Uses first Txout of Tx1. Creates two Txout to Carol and Bob, signed by Bob.
3. Uses second Txout of Tx1. Creates two Txout to David and Alice, signed by Alice.

Further remarks

Change Address:
Why does Alice have to send money back to herself? In Bitcoin, either all or none of the coins have to be consumed by another transactions. The address the money is sent back to is called a *change address*. This enables an efficient verification, as one only has to keep a list of **unspent transaction outputs** (**UTXO**).

Consolidating funds:
Instead of having many unspent transaction outputs, a user can create a transaction that uses all UTXO she has and creates a single UTXO with all the coins in it.

Joint payments:
Two or more people can combine their inputs and create one output. Of course, it requires signatures from all involved people.

Outline

1. Types of ledger
   - Account-based ledger
   - Transaction-based ledger
   - Data structure and Bitcoin script

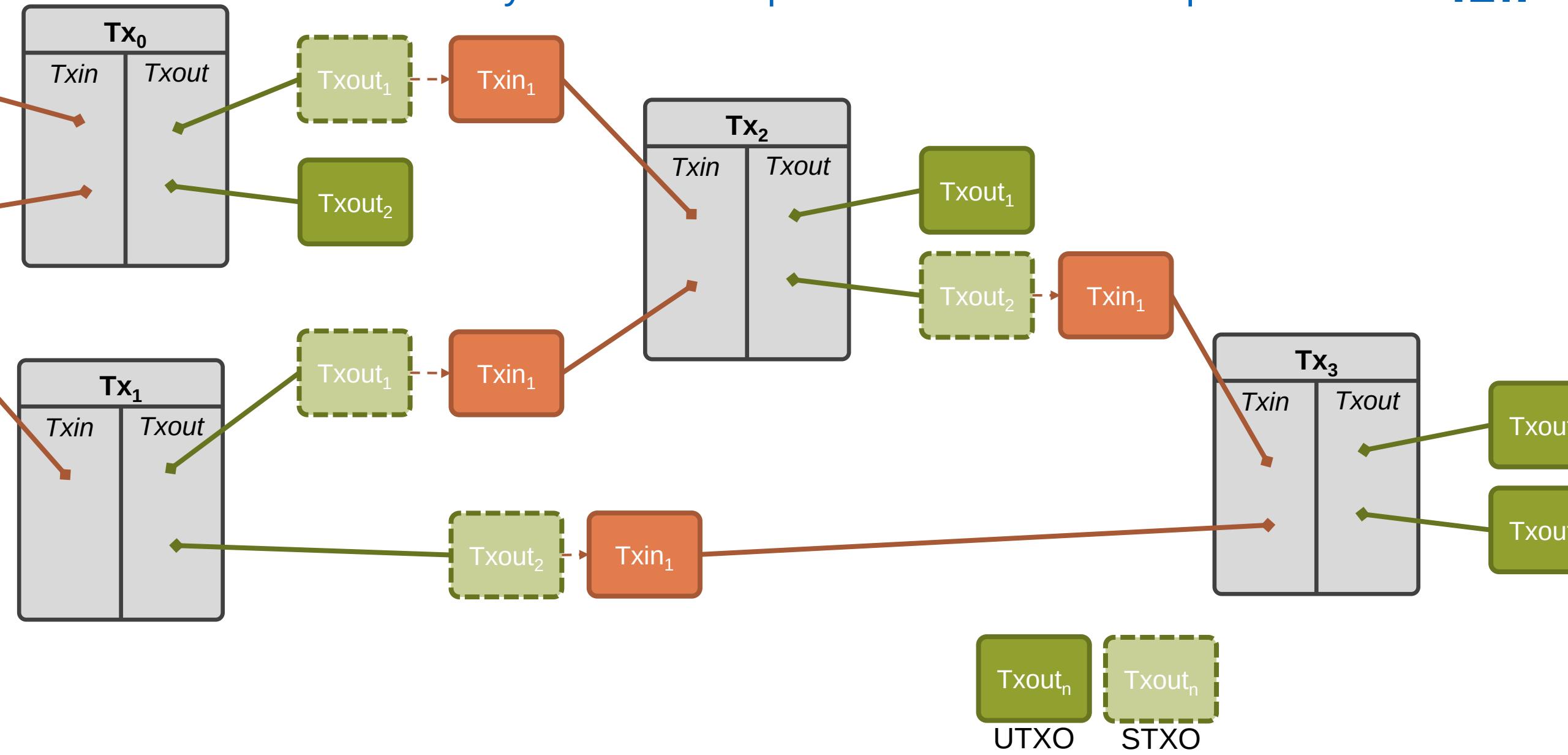
2. Transactions in Bitcoin
   - Data structure
   - Transaction verification

3. Bitcoin Script

4. Use cases of Bitcoin Script

This chapter is heavily inspired by and uses examples from „Bitcoin and Cryptocurrency Technologies“ by Arvind Narayanan

Transactions connected by transaction inputs and transaction outputs



Transactions – an advanced look

As previously stated, transactions consist of inputs and outputs following these principles:

- All inputs reference an existing unspent output or a coinbase transaction.
- Inputs and outputs **contain scripts** (scriptSig, scriptPubKey) **for verification**.
- Output scripts (scriptPubKey) **specify the conditions to redeem their value**.
- Input scripts (scriptSig) **provide a signature** to redeem the referenced output.
- Only **outputs store** the **BTC value** and the **receiver's address**.
- **All coins have a history (inputs/outputs) up to the original coinbase transaction that created them.**

Input format

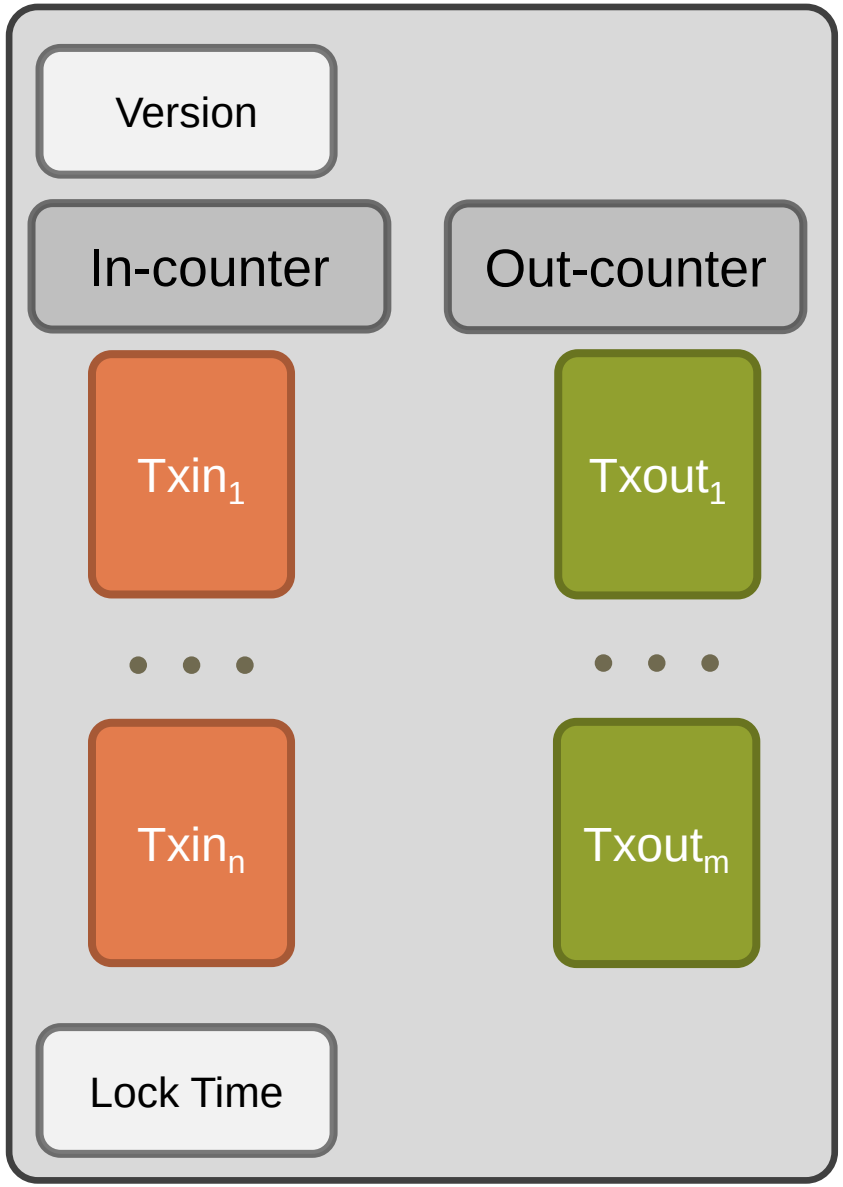
Txin

- previous transaction hash
- previous `Txout`-index
- script length
- *scriptSig*

Output format

Txout

- value in *Satoshi (=10^{-8} BTC)*
- script length
- *scriptPubKey*



Transactions – Logical data structure


```
"hash":"5a42590fbe0a90ee8e8747244d6c84f0db1a3a24e8f1b95b10c9e050990b8b6b",
"ver":1,
"vin_sz":2,
"vout_sz":1,
"lock_time":0,
"size":404,
"in":[
 {
  "prev_out":{
   "hash":"3be4ac9728a0823cf5e2deb2e86fc0bd2aa503a91d307b42ba76117d79280260",
   "n":0
  },
    "scriptSig":"30440..."
 },
 {
  "prev_out":{
   "hash":"7508e6ab259b4df0fd5147bab0c949d81473db4518f81afc5c3f52f91ff6b34e",
   "n":0
  },
  "scriptSig":"3f3a4ce81...."
 }
],
"out":[
 {
  "value":"10.12287097",
  "scriptPubKey":"OP_DUP OP_HASH160 69e02e18b5705a05dd6b28ed517716c894b3d42e OP_EQUALVERIFY OP_CHECKSIG"
 }
]
```


Metadata
It contains the **size of the transaction**, the **number** of **inputs** and **outputs**, the version and a lock-time. The hash is also contained, which can be referenced.

Inputs
An **array** of **all inputs**. Each input contains the previous transaction and the index of Txout. Also a signature *script* is provided.

Outputs
An **array** of **all outputs**. One output has two fields: the amount of the transferred coins[1] and the *scriptPubKey*.

An example transaction with two Txin and one Txout.

Example taken from „Bitcoin and Cryptocurrency Technologies“ by Arvind Narayanan

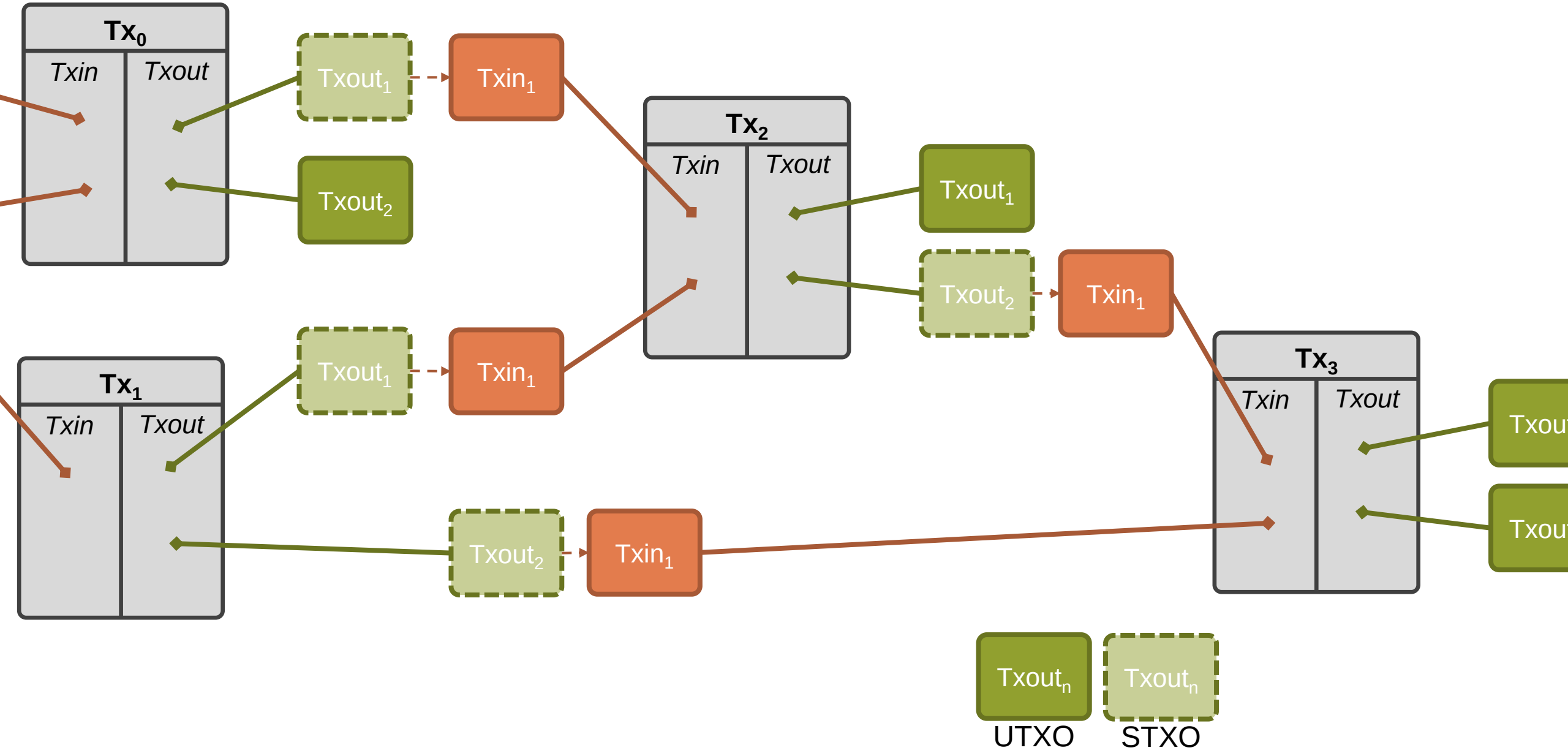
[1]Note, that the sum of all output amounts has to be the same or smaller than the sum of all inputs. The difference is the transaction fee, which is covered in the chapter on Bitcoin consensus.

How are transactions validated?

- Check syntactic correctness
- Make sure neither in or out lists are empty
- Size in bytes <= MAX_BLOCK_SIZE
- Each output value, as well as the total, must be in legal money range
- Make sure none of the inputs have hash=0, n=-1 (*coinbase* transactions)
- Check that nLockTime <= INT_MAX[1], size in bytes >= 100[2], and sig opcount <= 2[3]
- Reject "nonstandard" transactions: scriptSig doing anything other than pushing numbers on the stack, or scriptPubkey not matching the two usual forms[4]
- Reject if we already have matching tx in the pool, or in a block in the main branch
- For each input, if the referenced output exists in any other tx in the pool, reject this transaction.[5]
- For each input, look in the main branch and the transaction pool to find the referenced output transaction. If the output transaction is missing for any input, this will be an orphan transaction. Add to the orphan transactions, if a matching transaction is not in there already.
- For each input, if the referenced output transaction is coinbase (i.e. only 1 input, with hash=0, n=-1), it must have at least COINBASE_MATURITY (100) confirmations; else reject this transaction
- For each input, if the referenced output does not exist (e.g. never existed or has already been spent), reject this transaction[6]
- Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in legal money range
- Reject if the sum of input values < sum of output values
- 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
- Add to transaction pool[7]
- "Add to wallet if mine"
- Relay transaction to peers
- For each orphan transaction that uses this one as one of its inputs, run all these steps (including this one) recursively on that orphan

The Bitcoin wiki provides an overview about Bitcoin protocol rules for clients.

Transactions and their connection in between each other



Outline

1. Types of ledger
   - Account-based ledger
   - Transaction-based ledger
   - Data structure and Bitcoin script

2. Transactions in Bitcoin
   - Data structure
   - Transaction verification

3. Bitcoin Script

4. Use cases of Bitcoin Script

This chapter is heavily inspired by and uses examples from „Bitcoin and Cryptocurrency Technologies“ by Arvind Narayanan

Bitcoin Script

What does

OP_DUP OP_HASH160 8a014218a5a42e2c6fc5d573ab54a91ff555d1de OP_EQUALVERIFY OP_CHECKSIG

mean?

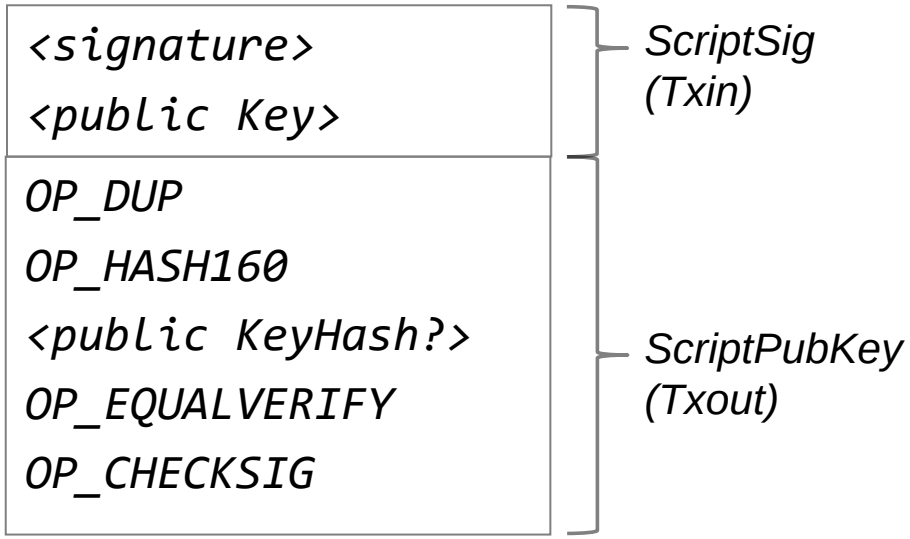
- A transaction in Bitcoin does not simply specify a public key the money gets sent to.
- The **transaction contains** a **script**. The scripting language is called *Bitcoin Script*.
- As stated previously, the identity is not a public key, but a hash of a public key (P2PKH). A person to redeem the UTXO needs to provide a public key that hashes to the P2PKH and a signature which belongs to this public key. Therefore, we need to have a way to check if an entity is allowed to spend the TxOut.

How does the script work?

- The script is executed and if the result is true, the UTXO can be spent, otherwise not.
- Idea: not only the *scriptPubKey* contains *Bitcoin Script*, but also *scriptSig* contains *Bitcoin Script.*
- The **scriptSig** is **concatenated** with the **scriptPubKey** and then executed.

Bitcoin Script (cont.)

Stack



Transaction script

Explanation
On the left the stack is displayed. On the right, the transaction script is specified. The script is executed line by line and the stack displays the items remaining **after** the execution of the line.

This box explains every step.

Bitcoin Script (cont.)

Current Command: *<signature>*

The signature is pushed onto the stack.

Bitcoin Script (cont.)

Stack

Transaction script

Current Command: *<public Key>*

The public key is pushed onto the stack.

Bitcoin Script (cont.)

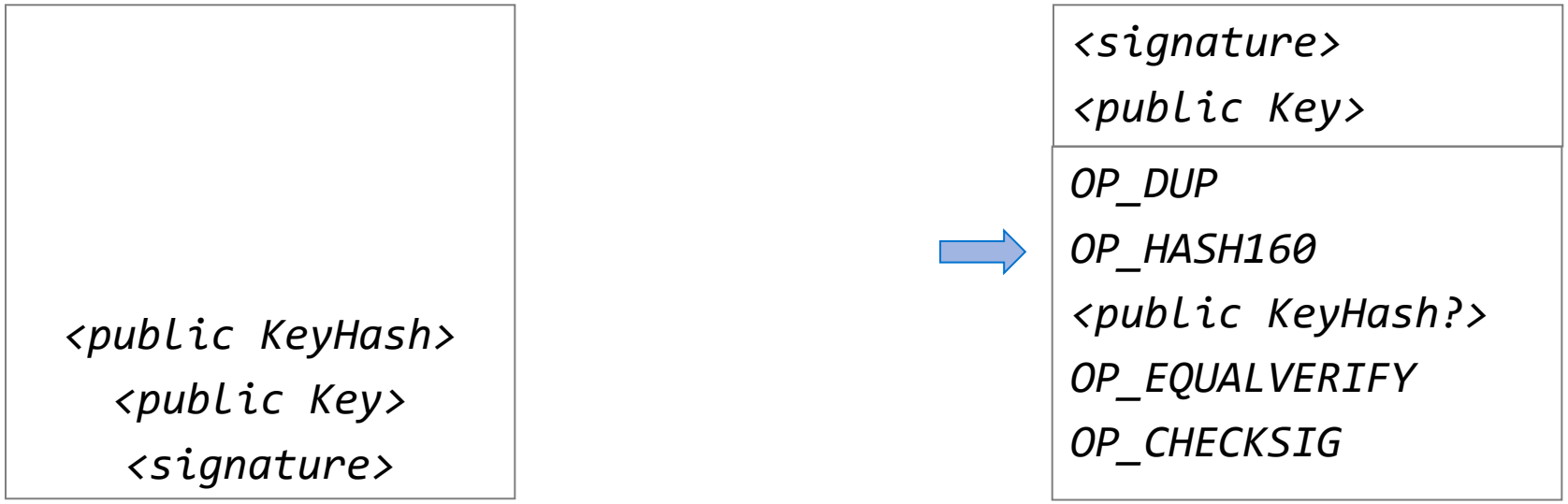
Stack

Transaction script

Current Command: *OP_DUP*

OP_DUP duplicates the current top item on the stack. Therefore, *<public Key>* is placed on top of the stack again.

Bitcoin Script (cont.)



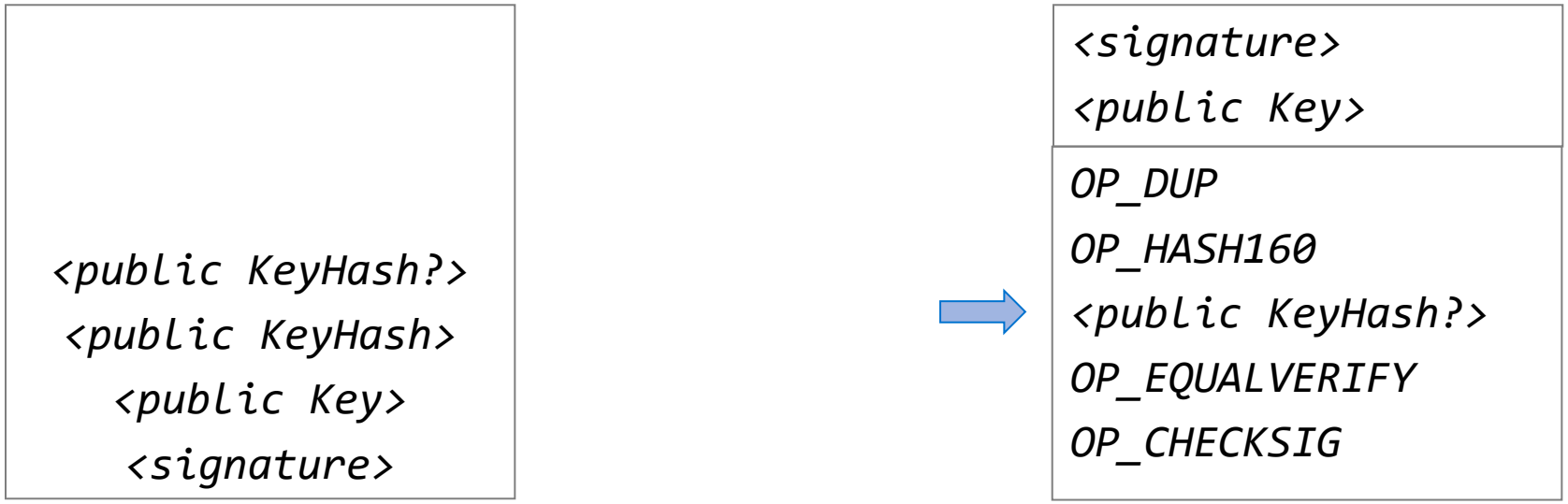
Stack

Transaction script

Current Command: *OP_HASH160*

The command tells us to pop the top stack value, compute its cryptographic hash and push the result top the top of the stack.

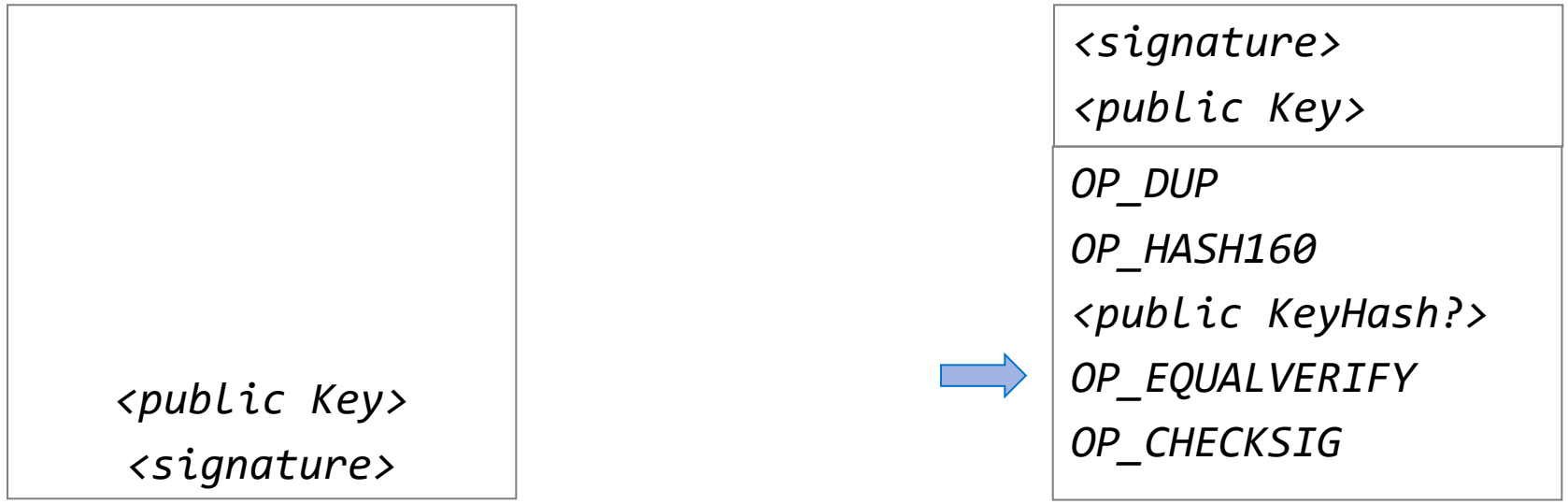
Bitcoin Script (cont.)



Current Command: *<public KeyHash?>*

The public key specified in the Txout is pushed onto the stack. This was the hash specified by the sender of the transaction.

Bitcoin Script (cont.)



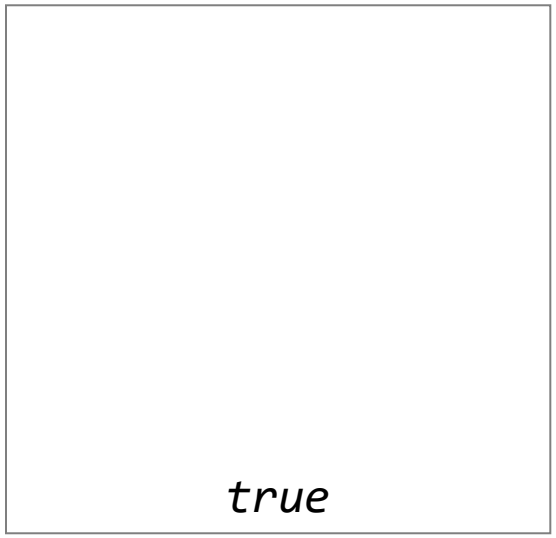
Stack

Transaction script

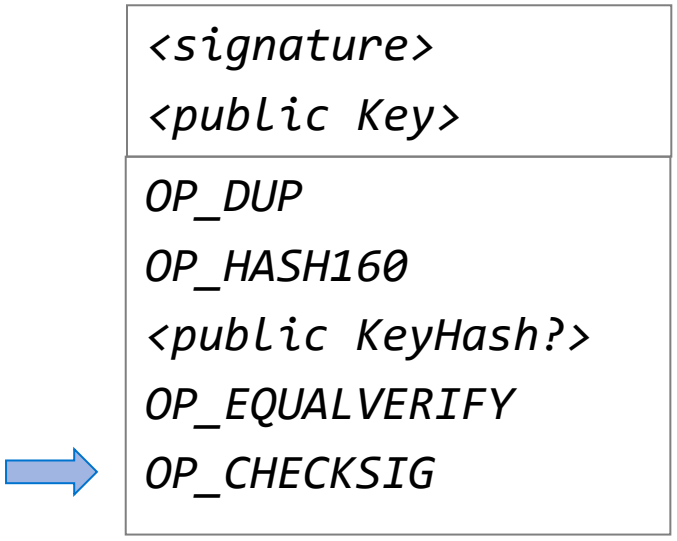
Current Command: *OP_EQUALVERIFY*

The *OP_EQUALVERIFY* command checks whether the two top items on the stack are equal or not. If they are not equal, the program stops and an error is thrown. Otherwise, both items are consumed. (In this case: Success!)

Bitcoin Script (cont.)



Stack



Transaction script

Current Command: *OP_CHECKSIG*

The command *OP_CHECKSIG* pops two items of the stack and does a entire signature verification. If the signature verification is valid, then both items are consumed and true is pushed on the stack. The stack contains true and the transaction valid.

What exactly is signed by the entity? find out more.

Bitcoin Script (cont.)

Bitcoin script offers domain-specific opcodes.

Bitcoin script supports:

- **Flow control**: *OP_IF*, *OP_NOTIF*, …
  - However, no loops.
- **Stack control**: *OP_DUP*, *OP_DROP*, …
- **Splicing**: *OP_CAT* (concatenates two strings), …
  - Currently disabled.
- **Bitwise logic**: *OP_AND*, *OP_OR*, *OP_XOR*, …
  - Currently disabled.
- **Arithmetic**: *OP_1ADD*, *OP_1SUB*, …
  - Multiplication / division disabled.
- **Cryptographic Operations**: *OP_SHA1*, *OP_SHA256*, …
- **Locktime**: *OP_CHECKLOCKTIMEVERIFY*

Therefore: *limited usage!*

Outline

1. Types of ledger
- Account-based ledger
- Transaction-based ledger
- Data structure and Bitcoin script

2. Transactions in Bitcoin
- Data structure
- Transaction verification

3. Bitcoin Script

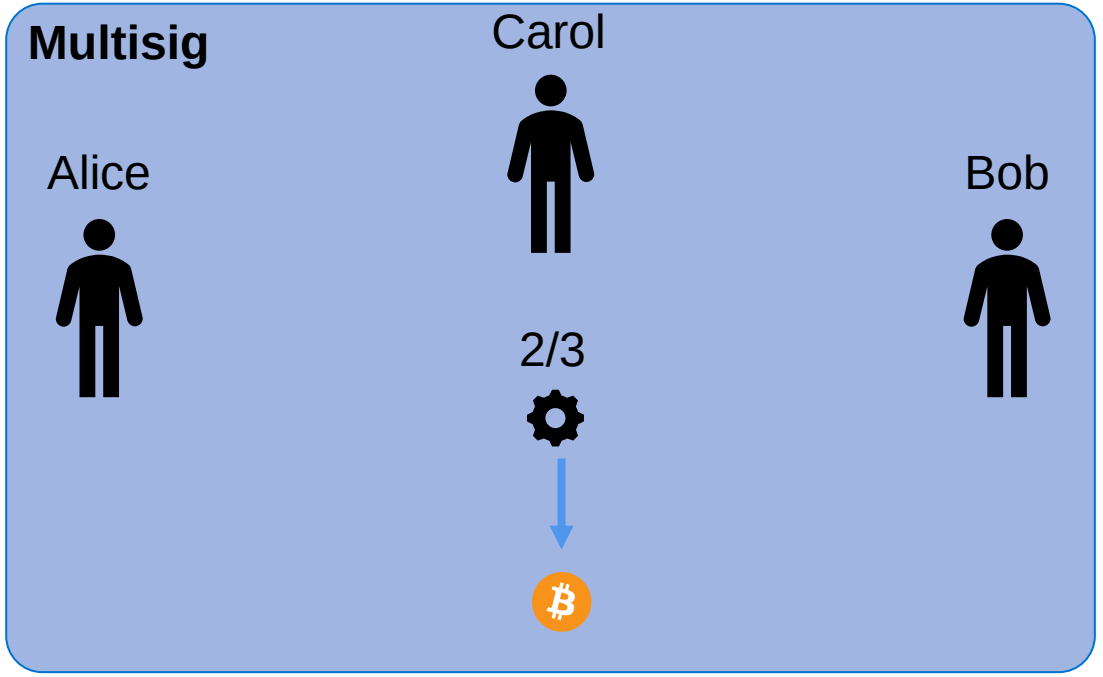
4. Use cases of Bitcoin Script

This chapter is heavily inspired by and uses examples from „Bitcoin and Cryptocurrency Technologies“ by Arvind Narayanan

Use cases of Bitcoin Script – MultiSig address

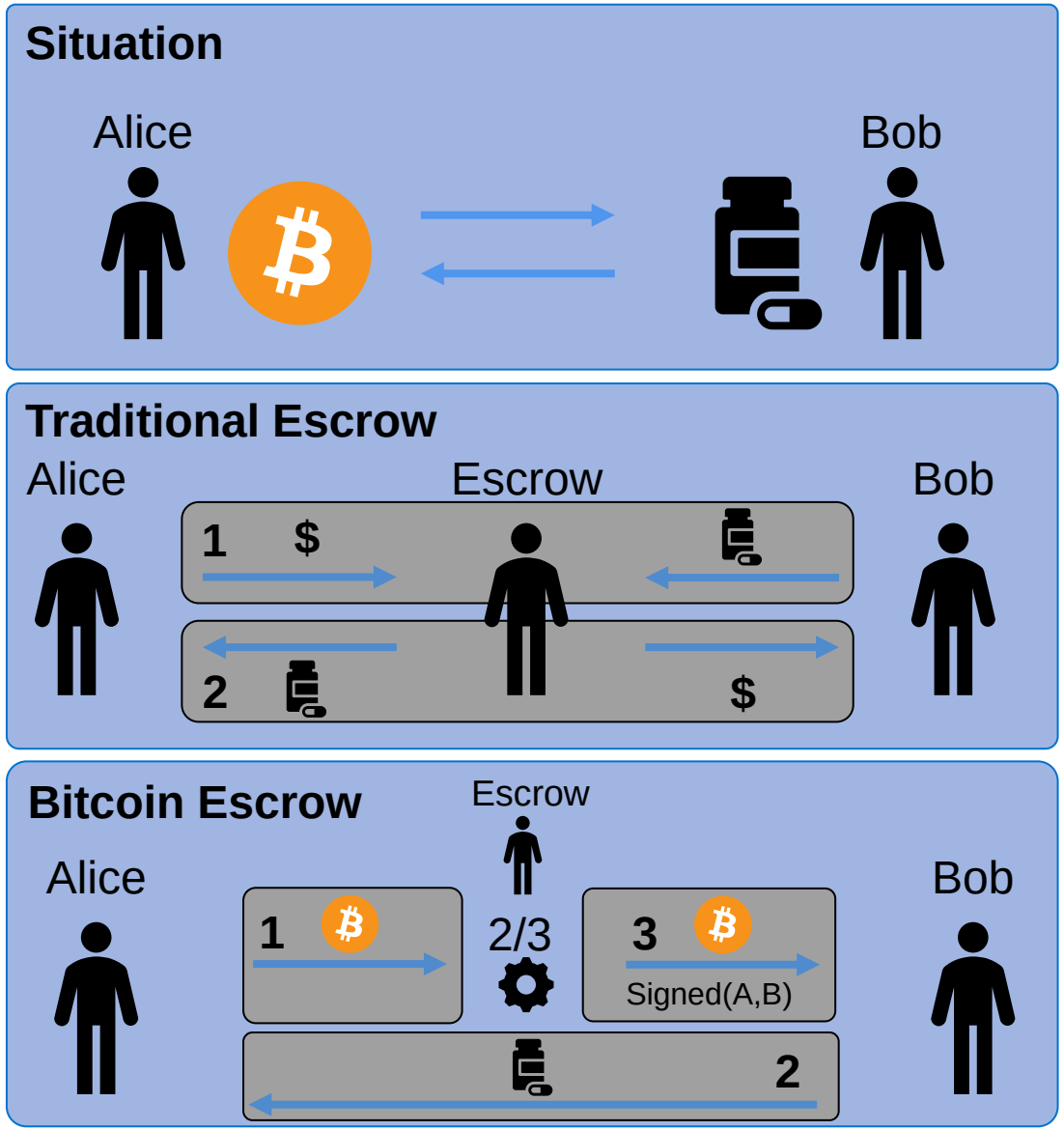
- A N/M multisig address is an address which has following properties
  - It requires more than one signature
  - N defines the number of required signatures
  - M defines the number of maximum signatures

- A 2/3 multisig address requires two arbitrary signatures from a pool of three predefined signatures.



Use cases of Bitcoin Script – Escrow

- Alice and Bob want to do business together. Alice wants to buy some real-world product from Bob with Bitcoin. However, Alice does not want to send the Bitcoin before having the physical good, but Bob does not want to send the physical good before having the Bitcoin in his wallet.
- What to do? In traditional systems, an escrow would enable the transaction. The escrow would receive both the good and the money and exchange them.
- With Bitcoin (or other cryptocurrencies), this process can be enhanced. Alice creates a 2/3 multisig-address and transfers the money to it. Bob can now safely send his good.
- When the good arrives, both Alice and Bob can sign the transaction and Bob receives his money.
  → If the exchange is correctly executed, the escrow is not involved.



Use cases of Bitcoin Script – Micropayments

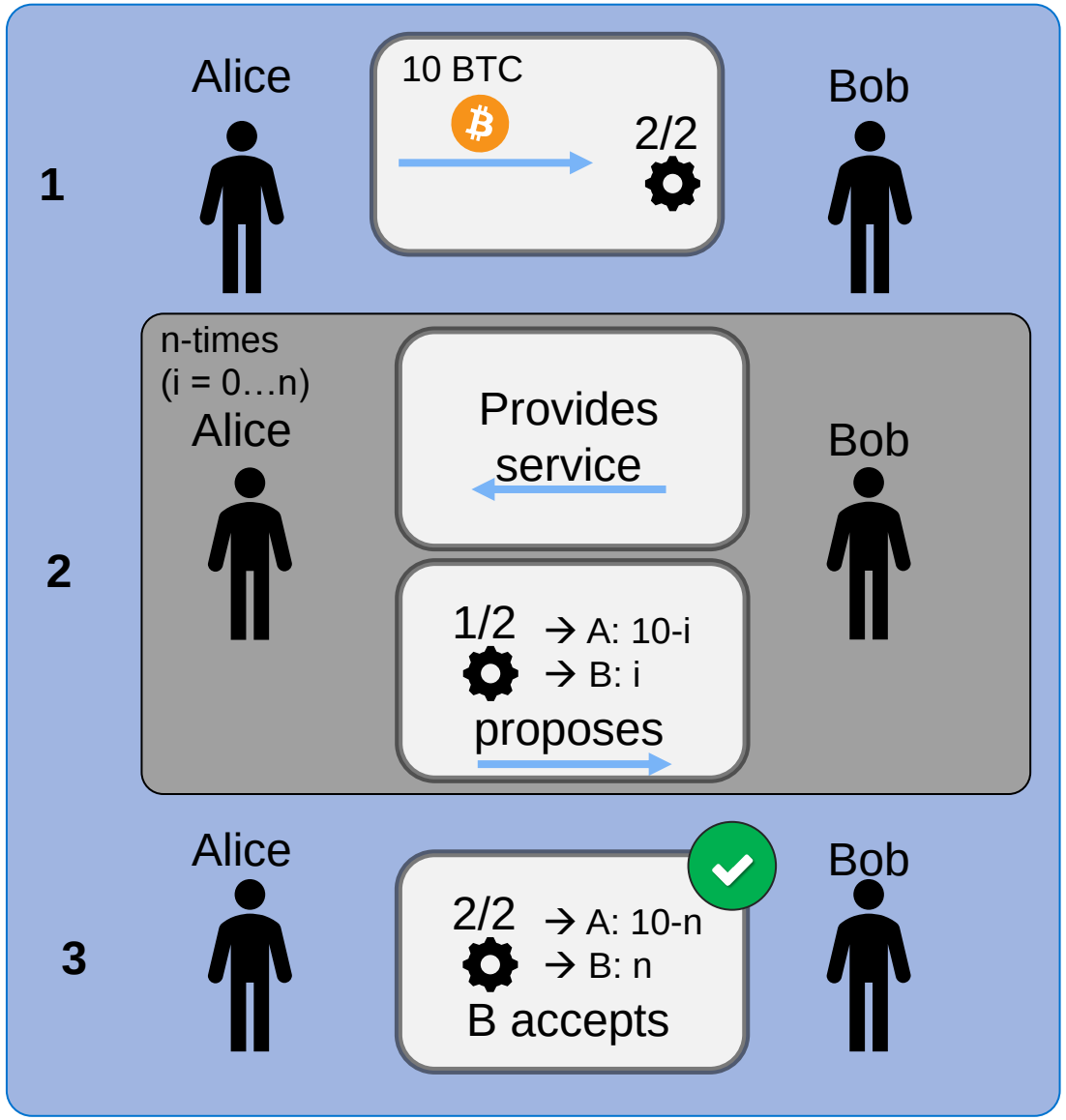
Alice wants to continuously use a service from Bob and pay a certain small amount of Bitcoin for each usage.

Creating new transactions every time won't work as:

- Too many transactions get created
- The fees for the transactions would be too high

An approach would be following:

- At the beginning, Alice send her maximum spendable amount to a 2/2 multisig address.
- Every time she consumes the service by Bob she signs a transaction locally for the multisig address, sending the accumulated amount to Bob and the change to herself. A message containing this incomplete transaction[1] is sent to Bob.
- If she is finished using the service, Bob will sign the last transaction he received and publish it on the network.



→ What is the problem?

[1] The transaction at this moment is rejected by the network, because the second signature is missing.

Use cases of Bitcoin Script – Lock time

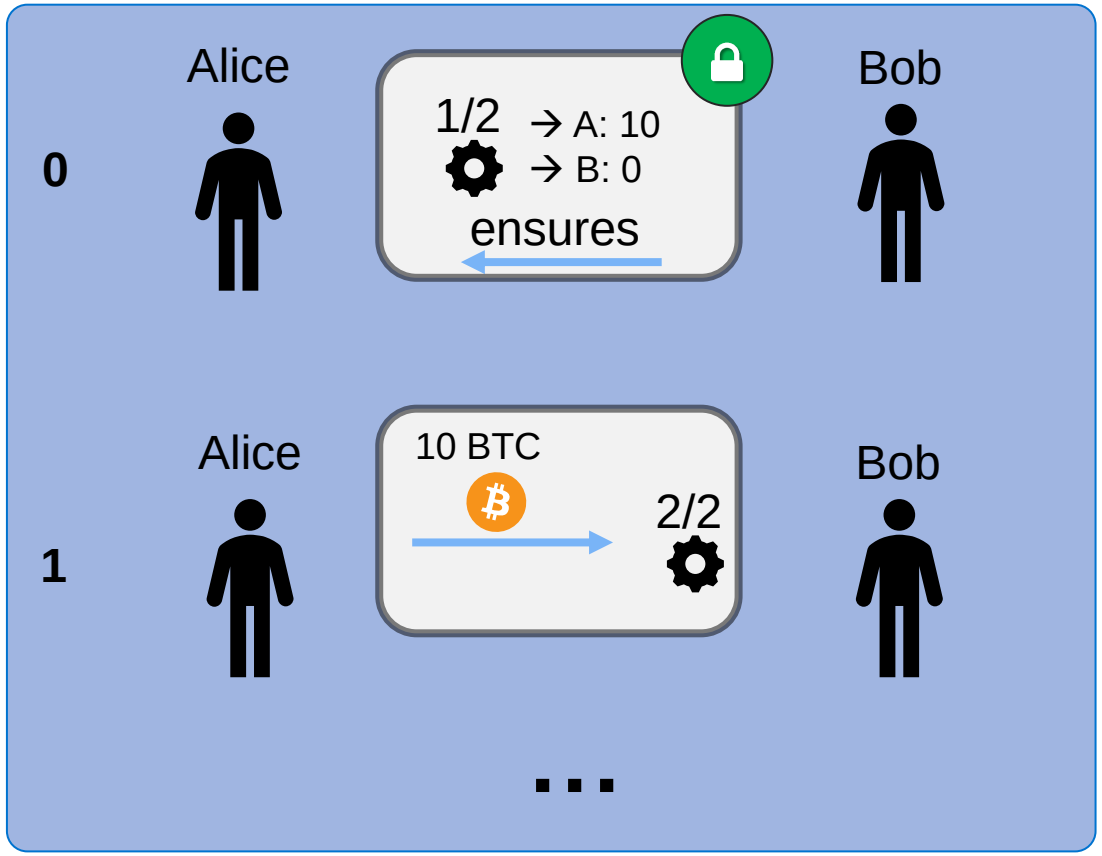
The problem is that Bob could decide **not to sign** the last transaction (or any transaction at all), leaving Alice's money stuck inside the multisig address.

However, there is a way around it:

On the slide describing the contents of a transaction (Slide 8 of this deck), we did not yet explain the purpose of the field *lock time*.

In order to ensure that Alice receives her money back, she can demand from Bob to create a transaction from the multisig-address, transferring the complete funds back to Alice before creating it. This transaction contains the lock time[1] as the transaction should only be spendable after a certain amount of time *t* and if the output is not spent otherwise.

With this, it is safe to create a multisig address without having to worry if one receives the money back.



Time locked

[1]As there is no notion of time in Bitcoin, the lock time is given as the block height. Until this block height is reached, the transaction is invalid.

Use cases of Bitcoin Script – Proof of burn / data storage

A special op code is the *OP_RETURN* command.

- If it is ever reached, it **throws** an **error**.
- Therefore, no matter where *OP_RETURN* is placed in the script, the transaction will not be redeemable afterwards.
- → The **money** / specified amount of Bitcoins is therefore **lost** or burnt forever.

This can be done out of different reasons:

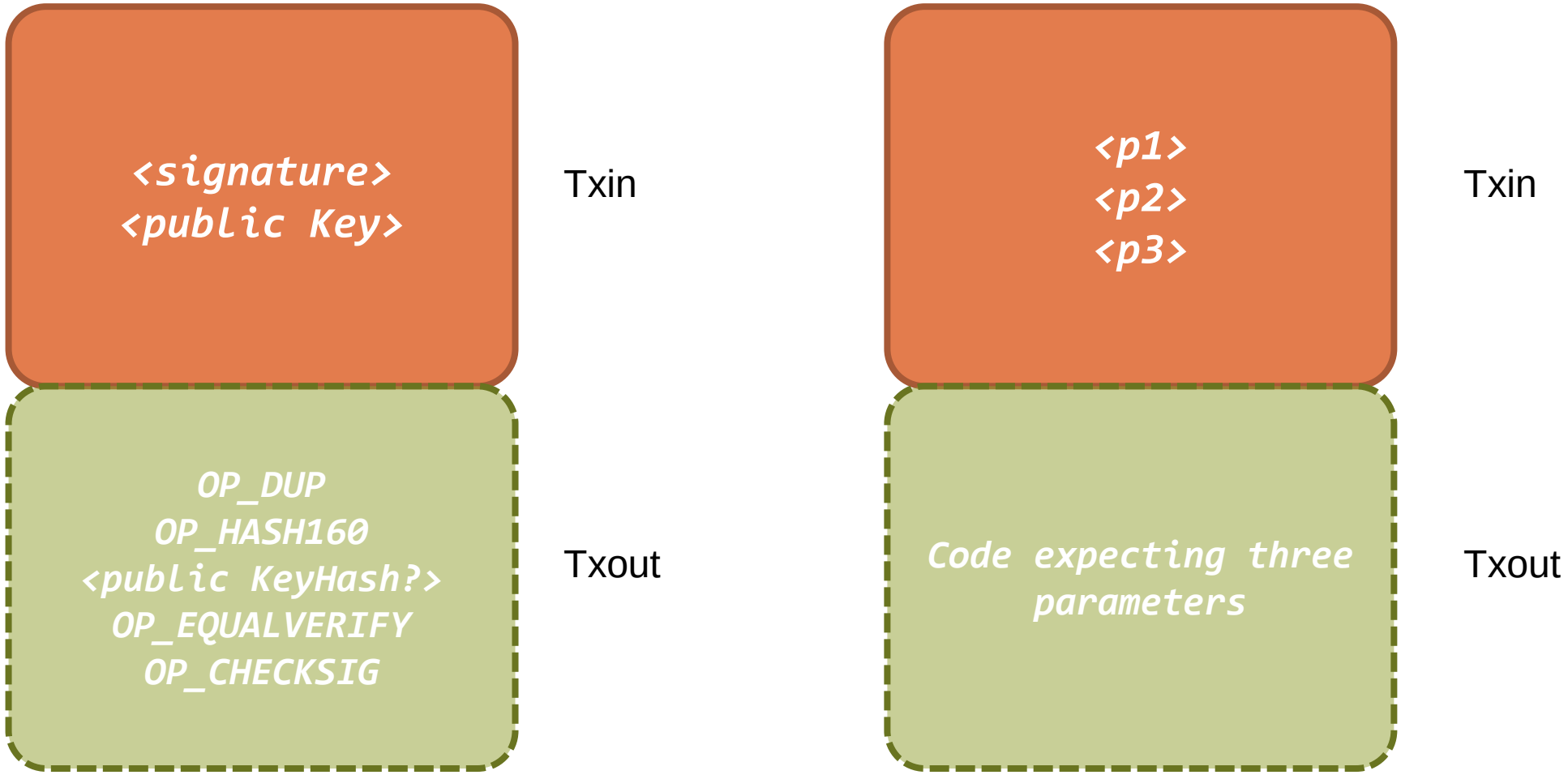
- **Bootstrapping for other Blockchain systems**: In order to receive a certain amount of coins on another Blockchain, you can burn your Bitcoins.

- **Storing data in the Bitcoin Blockchain**: As the code after an *OP_RETURN* statement won't be executed, you can place arbitrary data after that command. With this, data can be stored.

```
OP_RETURN
<Information>
<...>
<...>
```

Possible scriptPubKey

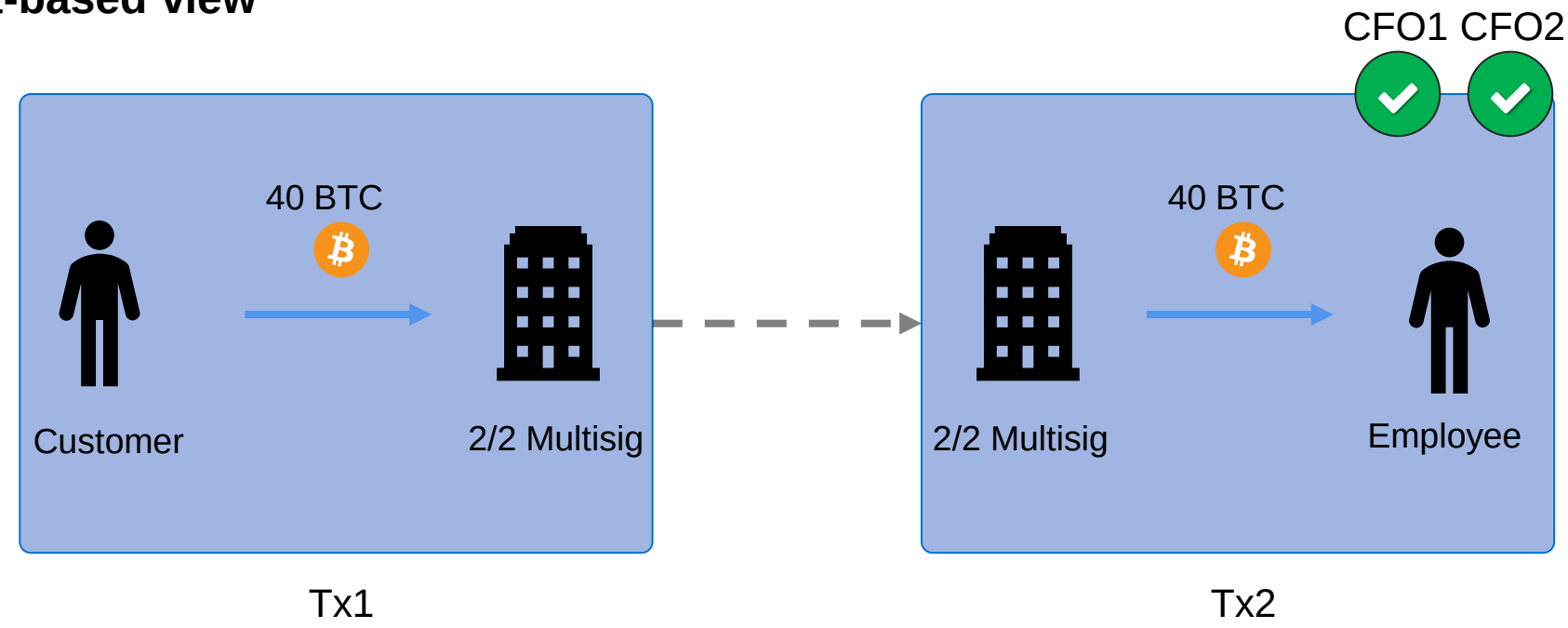
Issue: Data provided by Txin has to match the parameters expected by the Txout script



Real world example for script compatibility

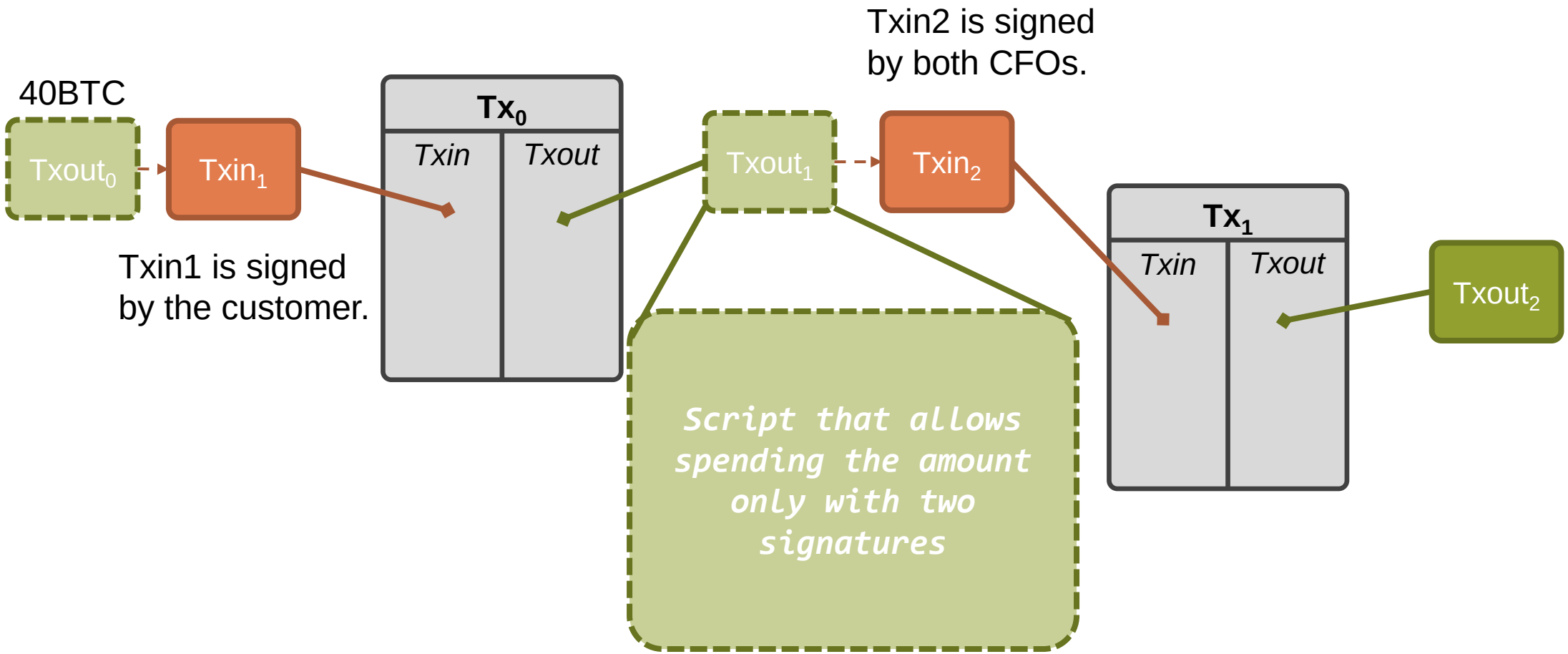
A company decides to be innovative and offers customers to pay with BTC. However, they want to be secure and want to store all their money on a 2/2 signature address to be signed by CFO1 and CFO2. A customer should directly pay to that certain script. However, this is not very user-friendly, as most users are not aware how to create a transaction with a certain script. That is because the sender has to define the script.

Account-based view



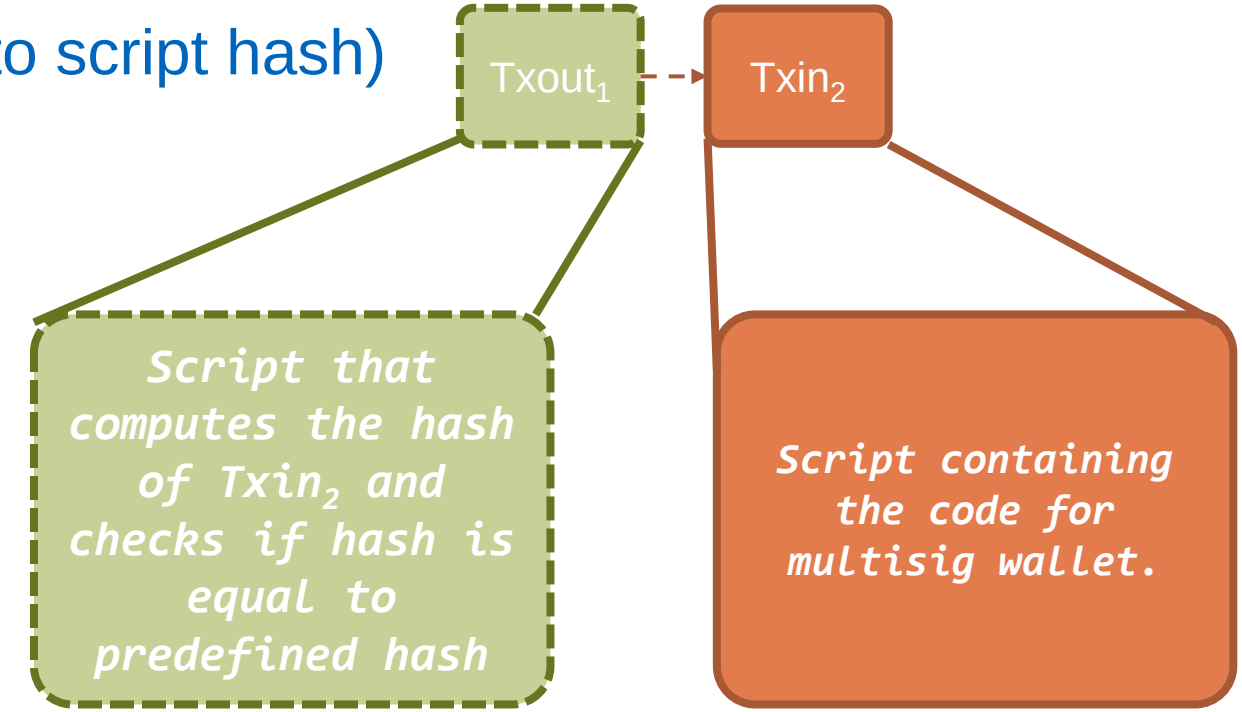
Real world example for script compatibility (cont.)

Transaction-based view



$Txout_1$ contains the script for the multisig-address.

Solution: P2SH (pay to script hash)



Similar to transactions “sending money to the hash of a public key”, you can specify a transaction like this: “send money to the hash of a *script*”. This is called Pay-to-Script-Hash (**P2SH**)

This has some advantages:

- It shifts the responsibility from the sender to the receiver, as the receiver gives a hash to the sender. The receiver has to make sure that it has a script that hashes to the hash.
- It has an efficiency gain. Miners have to store all UTXOs and with this, they get much smaller in means of storage requirements. All the scripts are used in the input and they don’t have to be stored by miners.

[1]A transaction which requires more than one signature to consume the Txout.