Blockchain Technology and Applications

CS 989

Transactions and Scripts in Bitcoin

Dr. Ir. Angshuman Karmakar

IIT Kanpur

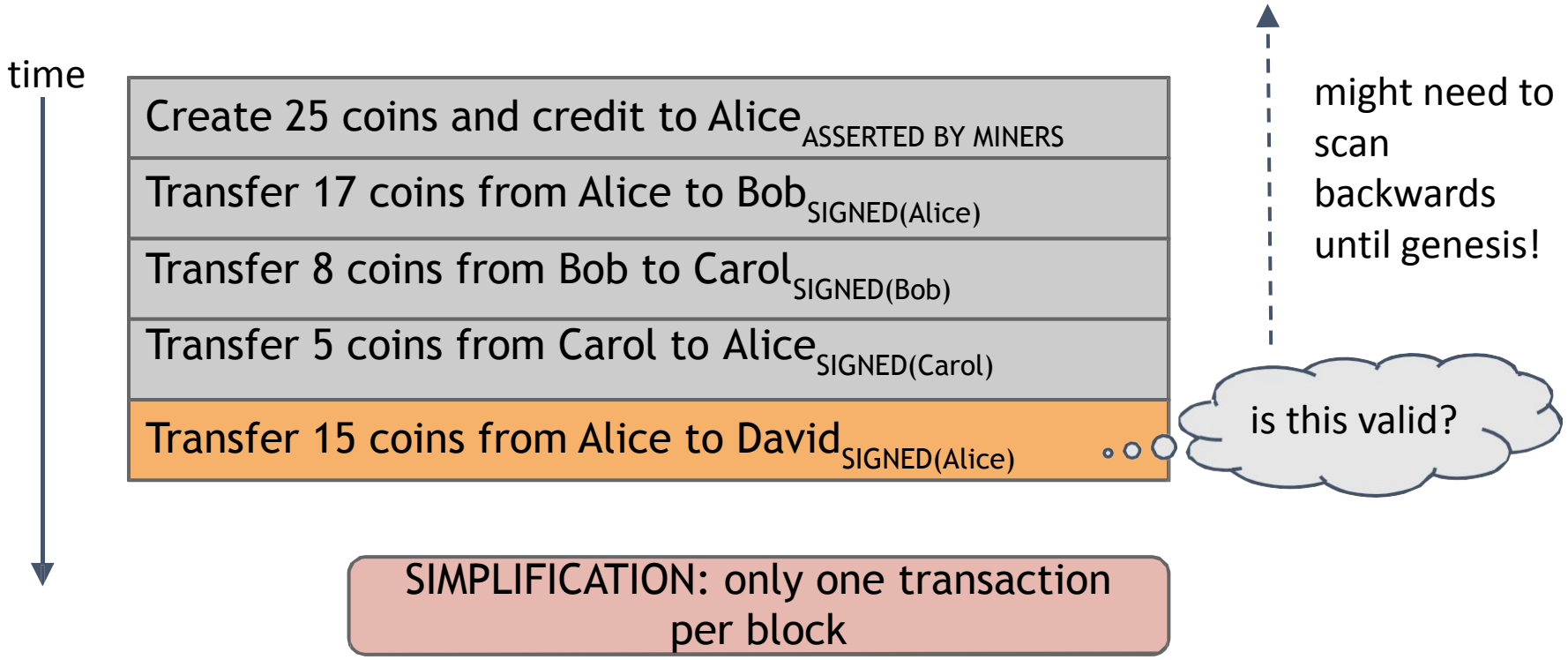
Teaching assistants

- **Sumit Lahiri** (sumitl@cse.iitk.ac.in)
- **Chavan Sujeet** (sujeetc@cse.iitk.ac.in)

Transactions in Bitcoin

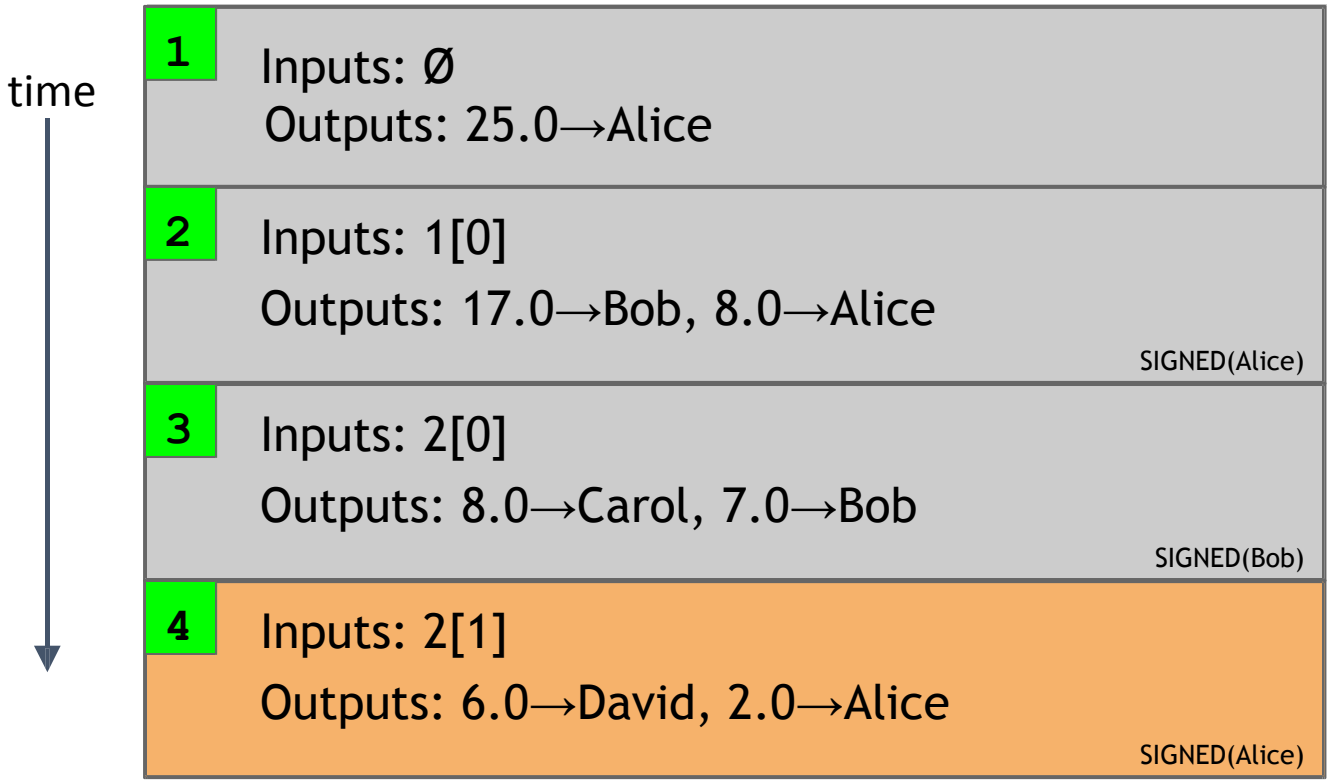
- Blockchain is a distributed ledger
- How to make a currency out of this ledger?
  - Addition of funds
  - Payments
  - Transfer of funds, etc.
- Maintain an account for each user
  - Most intuitive
  - Let's see an example

Account-based ledger



- We can make this process more efficient
  - More complex design

Transaction-based ledger

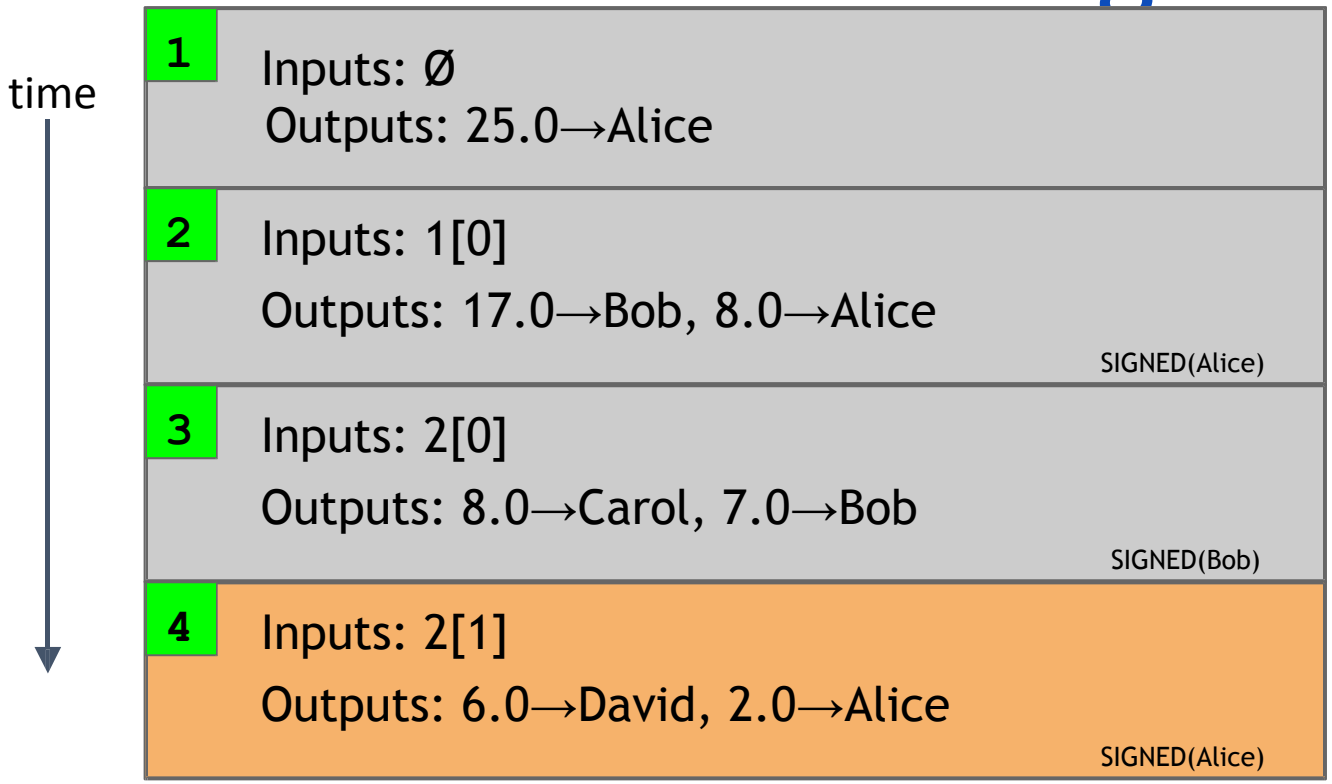


we implement this with hash pointers

SIMPLIFICATION: only one transaction per block

- Every transaction has one/multiple inputs and one/multiple outputs
- Alice wants to pay Bob

Transaction-based ledger



we implement this with hash pointers

SIMPLIFICATION: only one transaction per block

- Each input transaction refers to a previous output transaction
  - Source of *funds*
- First transaction is a coinbase transaction
  - No Input

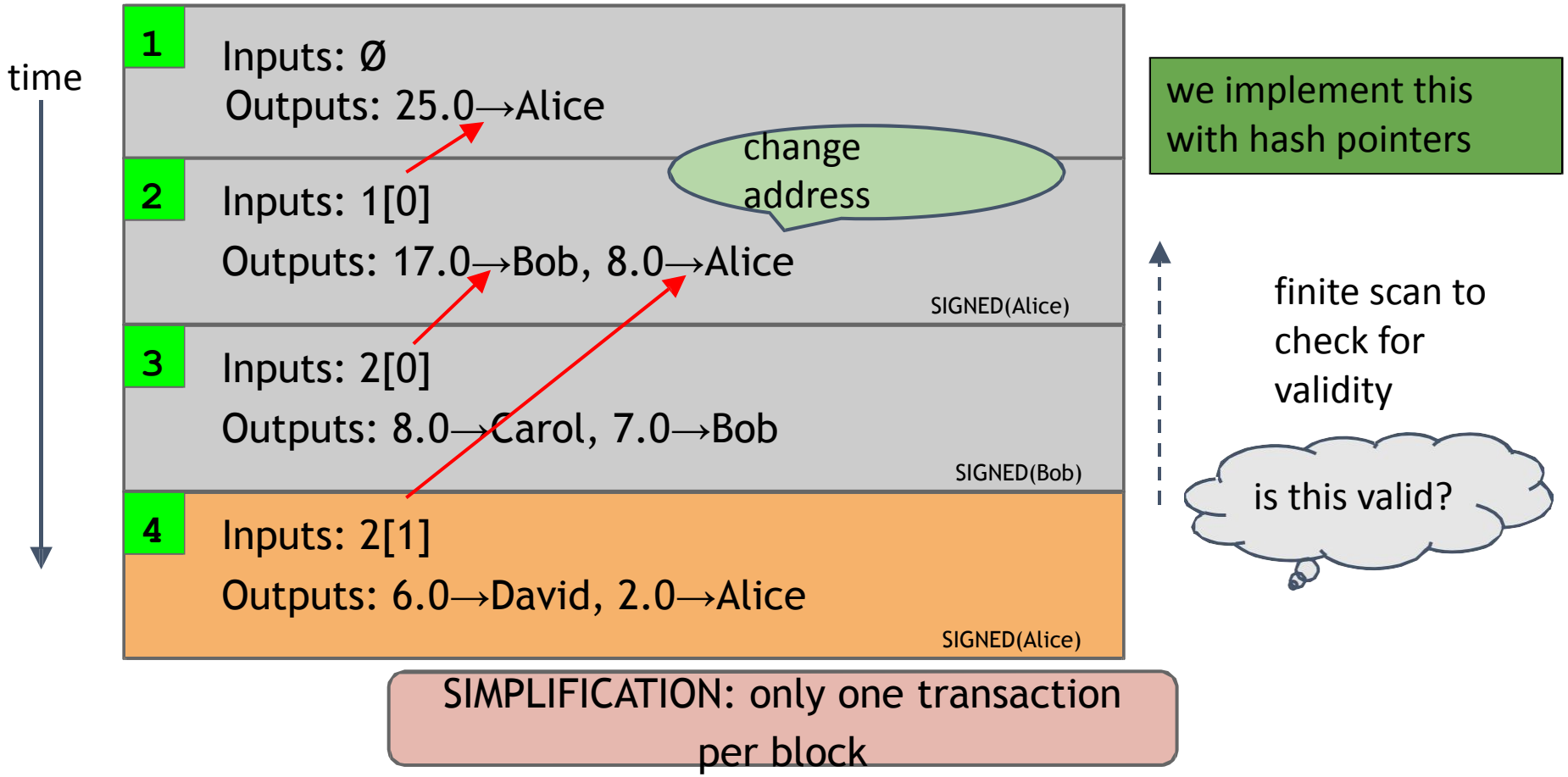
Transaction-based ledger

Change address

- Alice has 25 coins
  - Coins are immutable
- Alice wants to pay Bob 8 coins
- Alice creates a transaction
  - 1 input and two outputs

1. 8 to Bob's address
2. 17 back to herself

Transaction-based ledger



- Remember all addresses are pk of payee/payer
  - Alice creates new identities of herself
- Validity check is small numbers of scans

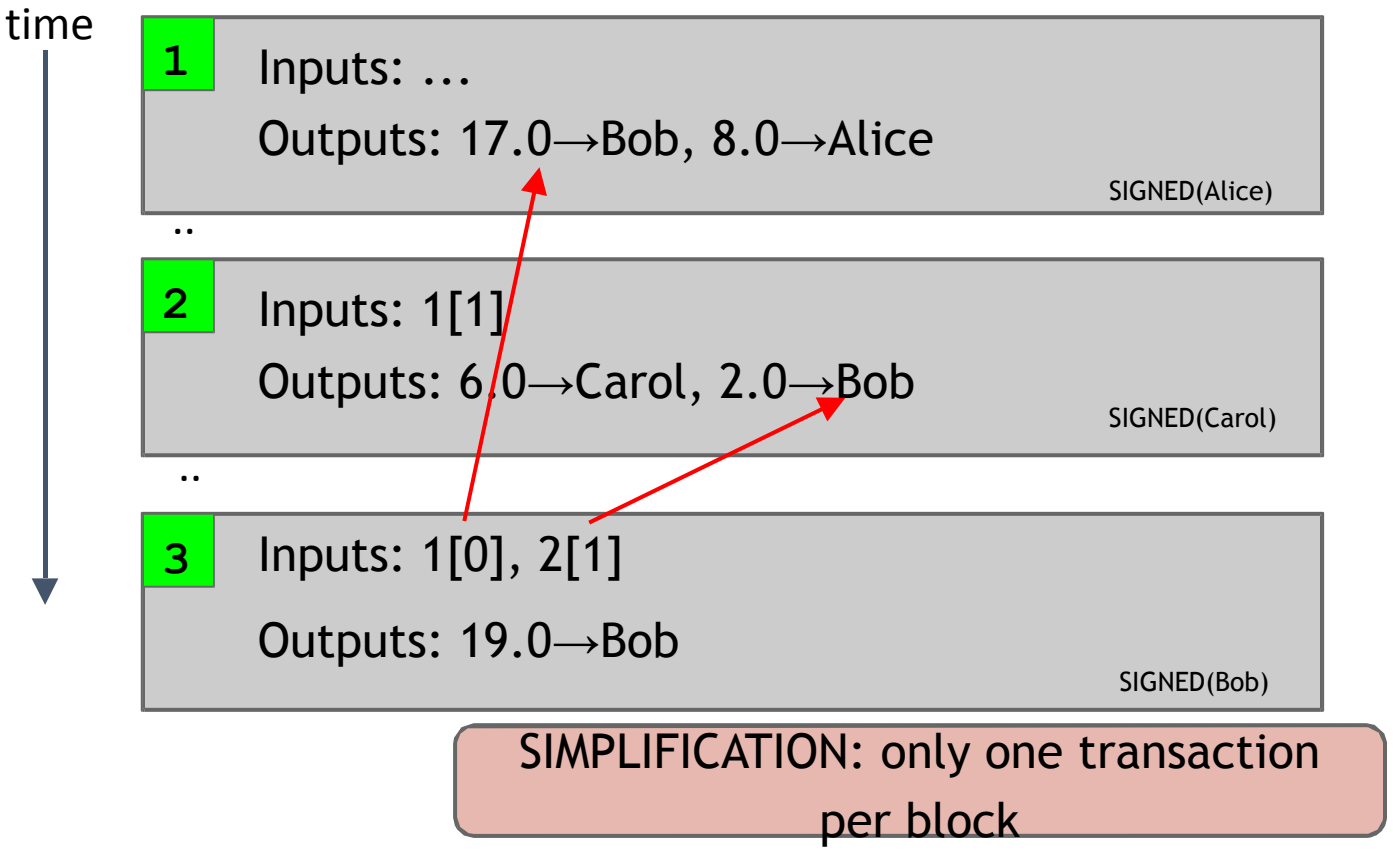
Transaction-based ledger

Change address and efficient verification

- Why?
  - Simplicity
  - Anonymity
  - Replay attacks
- Bonus
  - What happens when the payer omits the return address?
  - Alice--> Bob (8 coins)
- Bitcoin dust

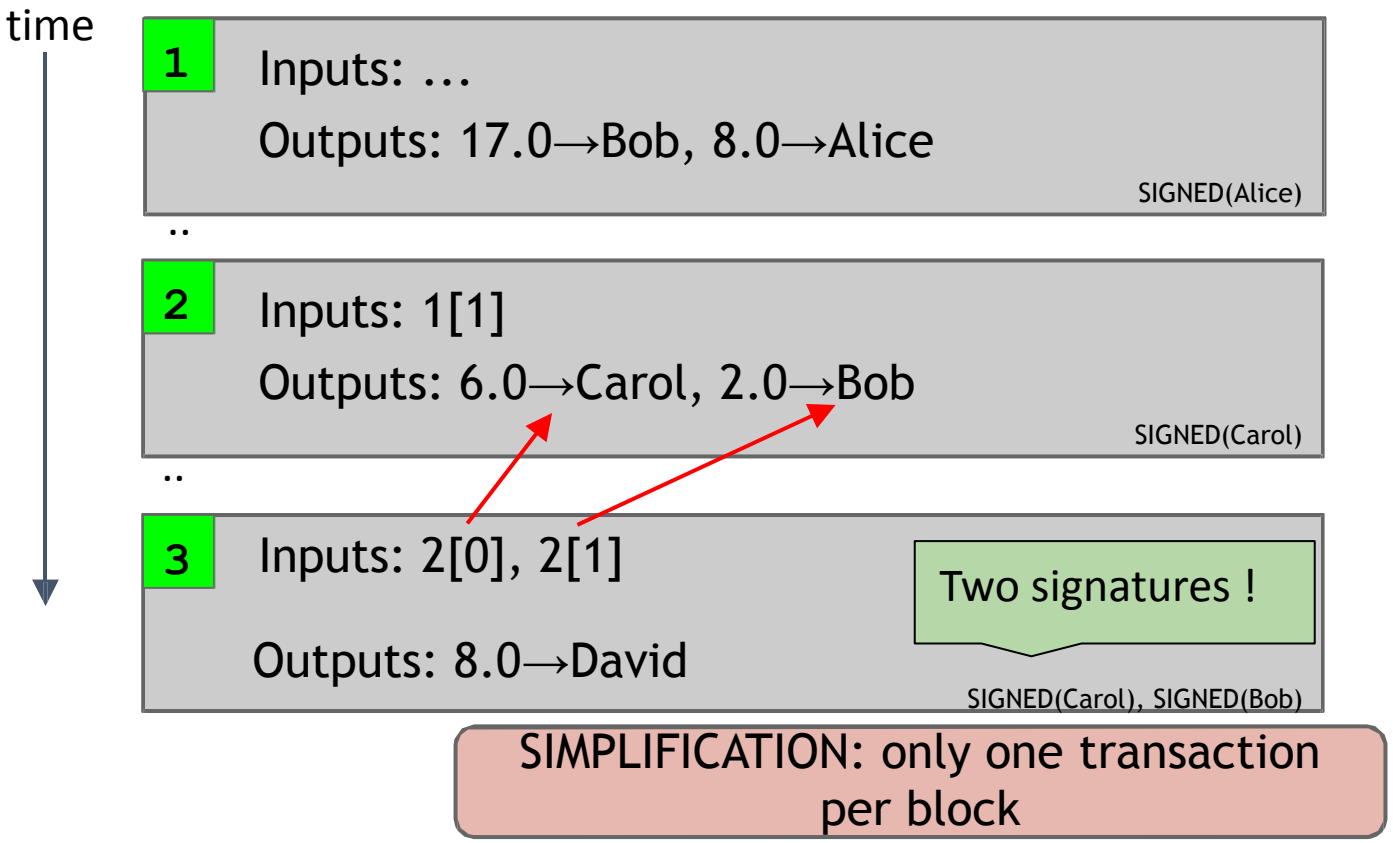
Transaction-based ledger

Merging values



Transaction-based ledger

Joint payments



Transaction-based ledger

Real example

metadata

input(s)

output(s)

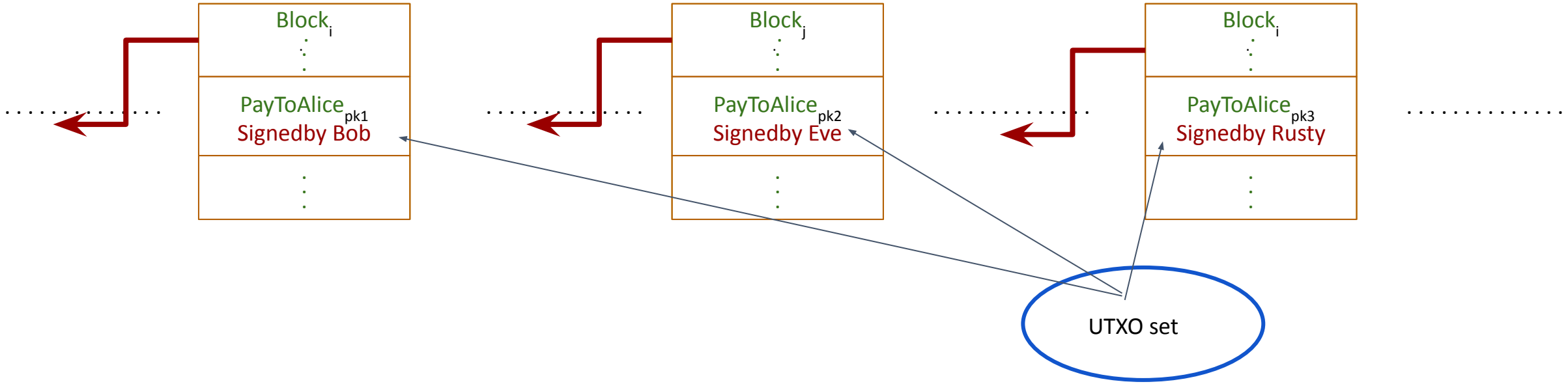

```
{
    "hash":"5a42590fbe0a90ee8e8747244d6c84f0db1a3a24e8f1b95b10c9e050990b8b6b",
    "ver":1,
    "vin_sz":2,
    "vout_sz":1,
    "lock_time":0,
    "size":404,
    "in":[
     {
      "prev_out":{
       "hash":"3be4ac9728a0823cf5e2deb2e86fc0bd2aa503a91d307b42ba76117d79280260",
       "n":0
  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


Recipient address

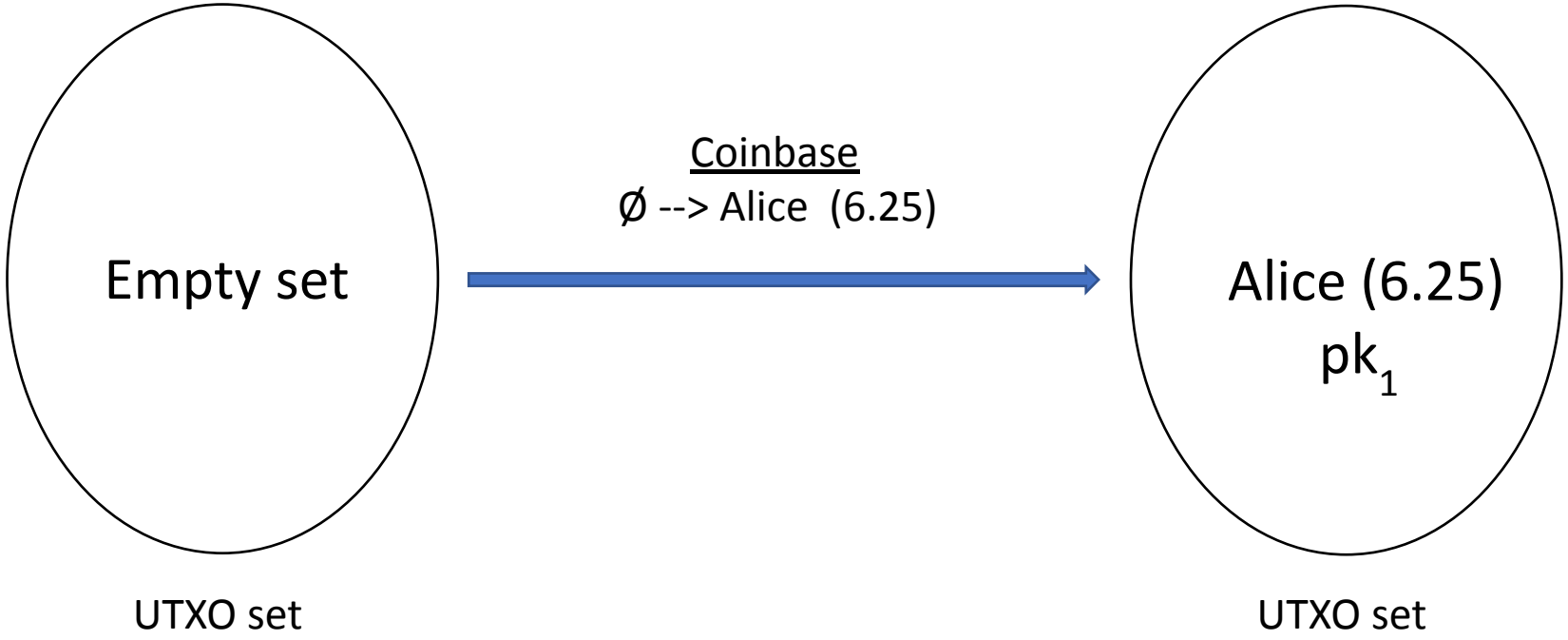
- More examples later

UTXO

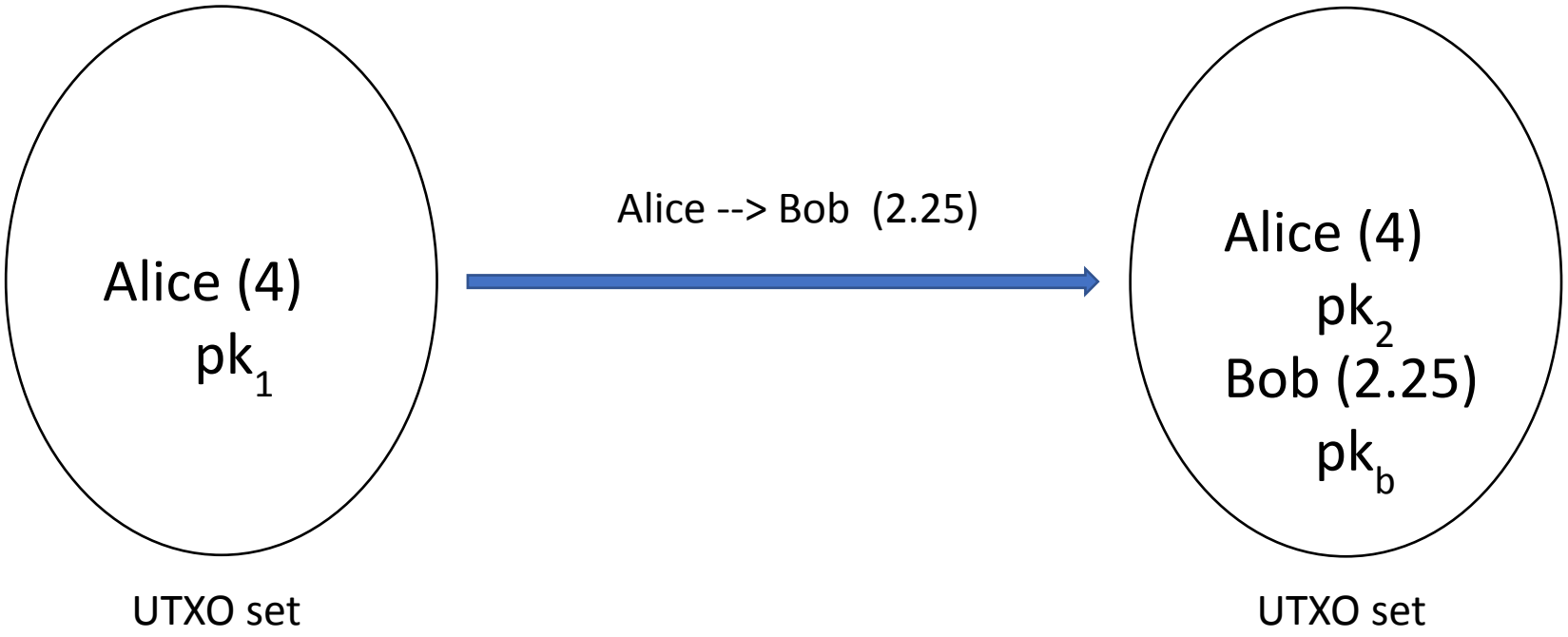
- Unspent transaction output
- A model to track all unspent outputs
- Bitcoin uses UTXO
- Miners track these
  - Faster verification
- Ethereum uses account-based model not UTXO



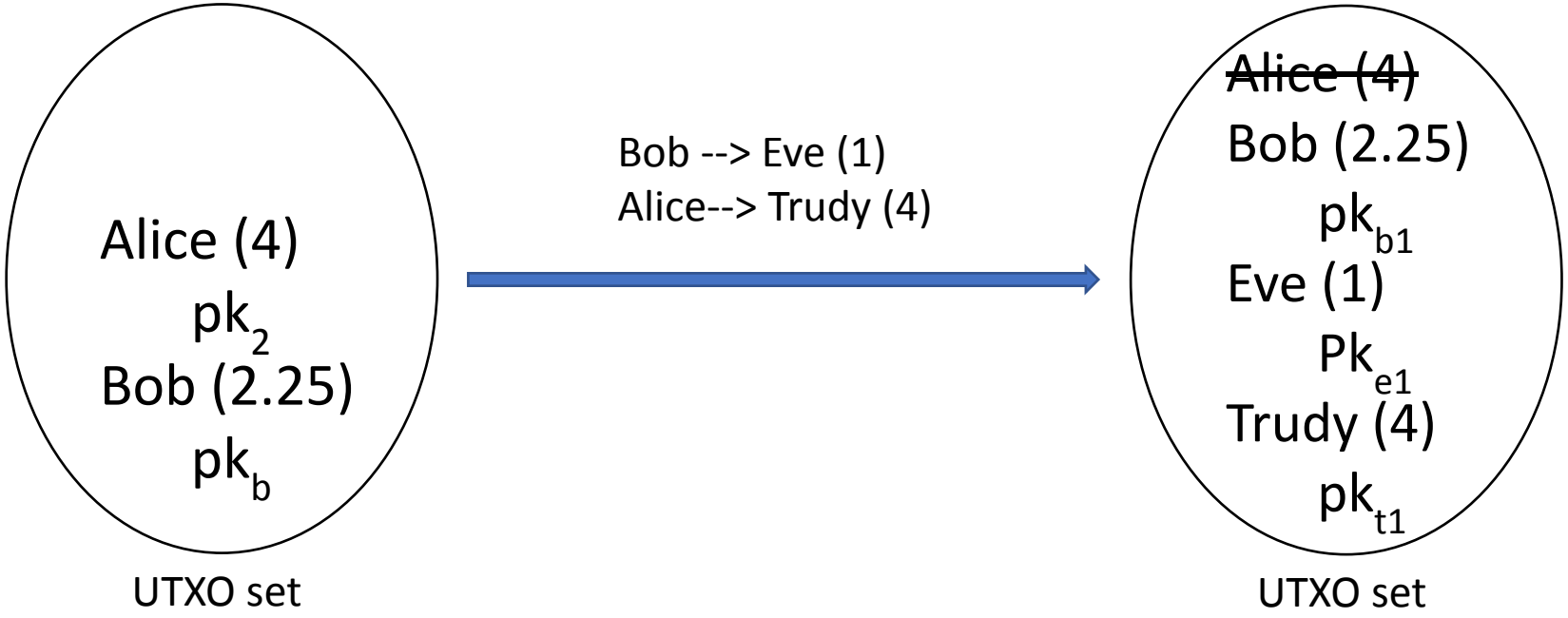
UTXO


Empty set
Coinbase
Ø --> Alice (6.25)
Alice (6.25)
pk_1
UTXO set
UTXO set

UTXO


Alice (4)
pk_1
Alice --> Bob (2.25)
Alice (4)
pk_2
Bob (2.25)
pk_b
UTXO set
UTXO set

UTXO



- Double-spend transactions?
- Who stores the UTXO set?

Bitcoin Dust

- In the Bitcoin network,
  - The transaction fee is is proportional to number of bytes it occupies on the blockchain
  - The larger the transaction the larger the fee
- Each UTXO requires some minimum number of bytes
- When paying someone the wallet will try to minimize the number if input transactions, to save cost
  - E.g. : For a payment of 1 BTC, If an user has 3 UTXOs of .5, .5, and 1 BTC, It will cost the user less to spend the UTXO with 1 BTC UTXO
- However, due to change address, some UTXO has very tiny BTC left
- It will take more BTC as transaction fee, than the amount they hold
- Such minuscule UTXOs are scattered in the Bitcoin network that will never be spent
- These are called Bitcoin dust

Scripts

Scripts

- What really is in the output of a transaction?
  - Public-key of the payee?
  - Signature of the payer?
- A script
  - A simple example
  - *"These coins can be redeemed by the owner of this public-key along with the signature which proves the ownership of the key"*
  - A *smart contract*
  - Can be more complex

Scripts

Language

- Stack based language (zero address instructions)
- Written specifically for bitcoin
  - Inspired by *Forth* language
- Very small
  - Only 256 instructions
  - 15 disabled
  - 75 reserved
- Basic logic if-then, throwing errors, return, etc
- Cryptographic instructions
  - Signature verification, hash computation, etc

Scripts

Language

- No loops
- Every instruction is executed only once in a linear fashion
- Upper bound of execution time and memory
  - Why?
- Not Turing complete !
  - Cannot compute arbitrarily complex function

Scripts

An example

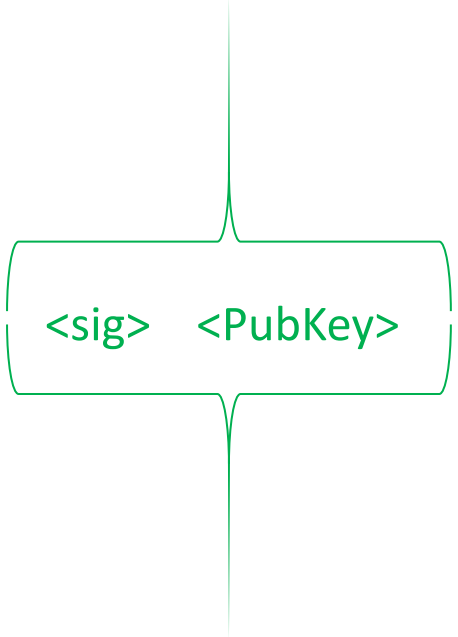
- A common transaction
  - Pay-to-PubKeyHash or P2PKH
- Output of a transaction is a script
  - ScriptPubKey
  - Locking script

OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

Scripts

An example

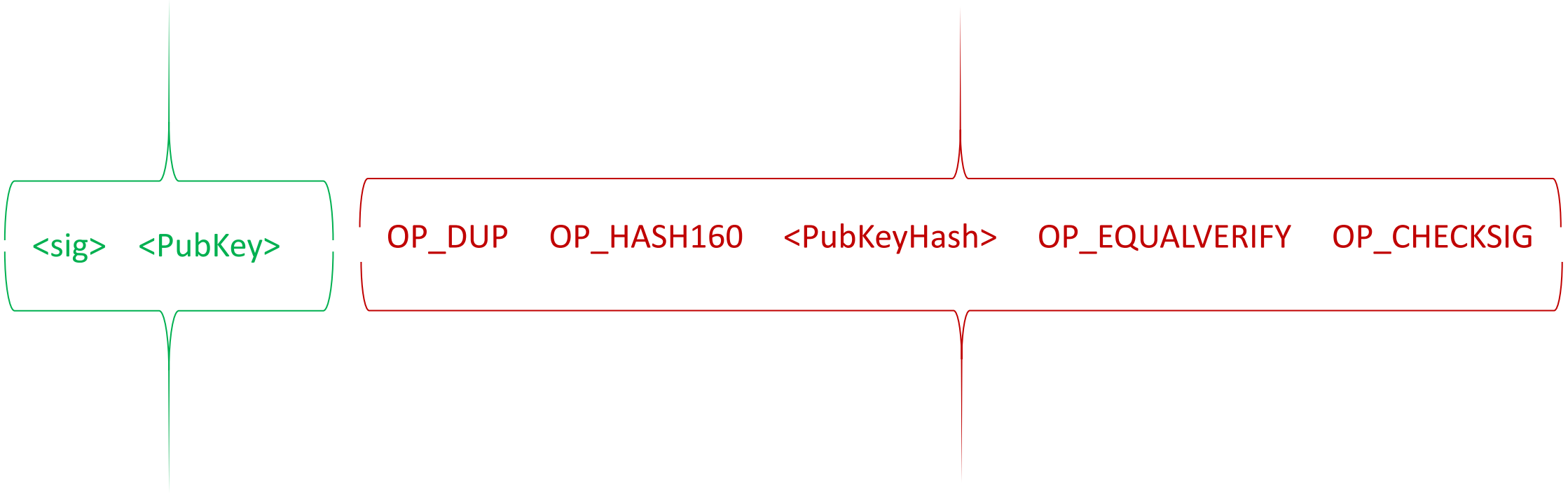
- Input of a transaction is also a script
  - ScriptSig + ScriptPubKey
- Unlocking script



Scripts

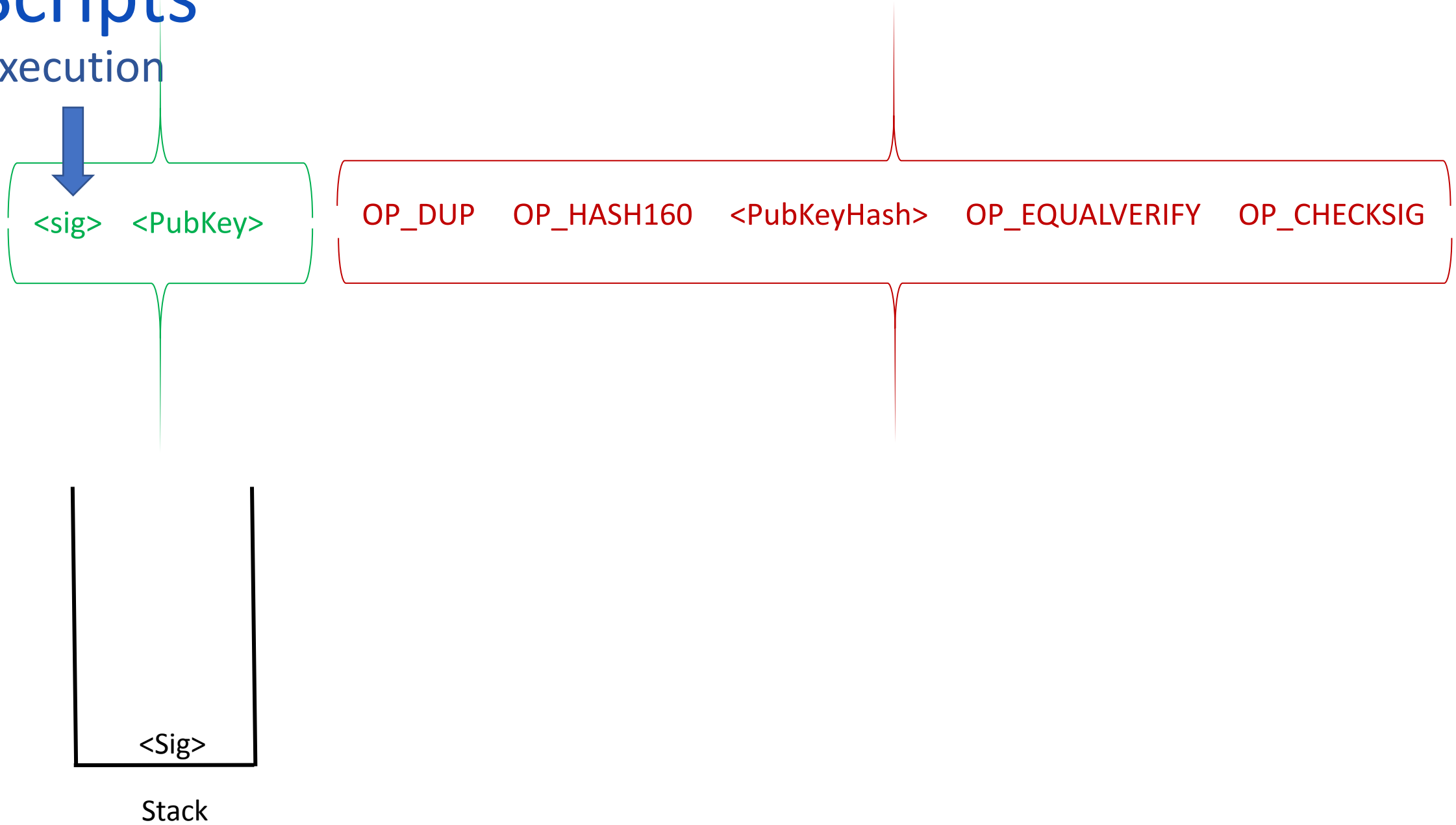
Execution

- While executing
  - ScriptSig + ScriptPubKey
- Only two outcomes
  - Success or Failure



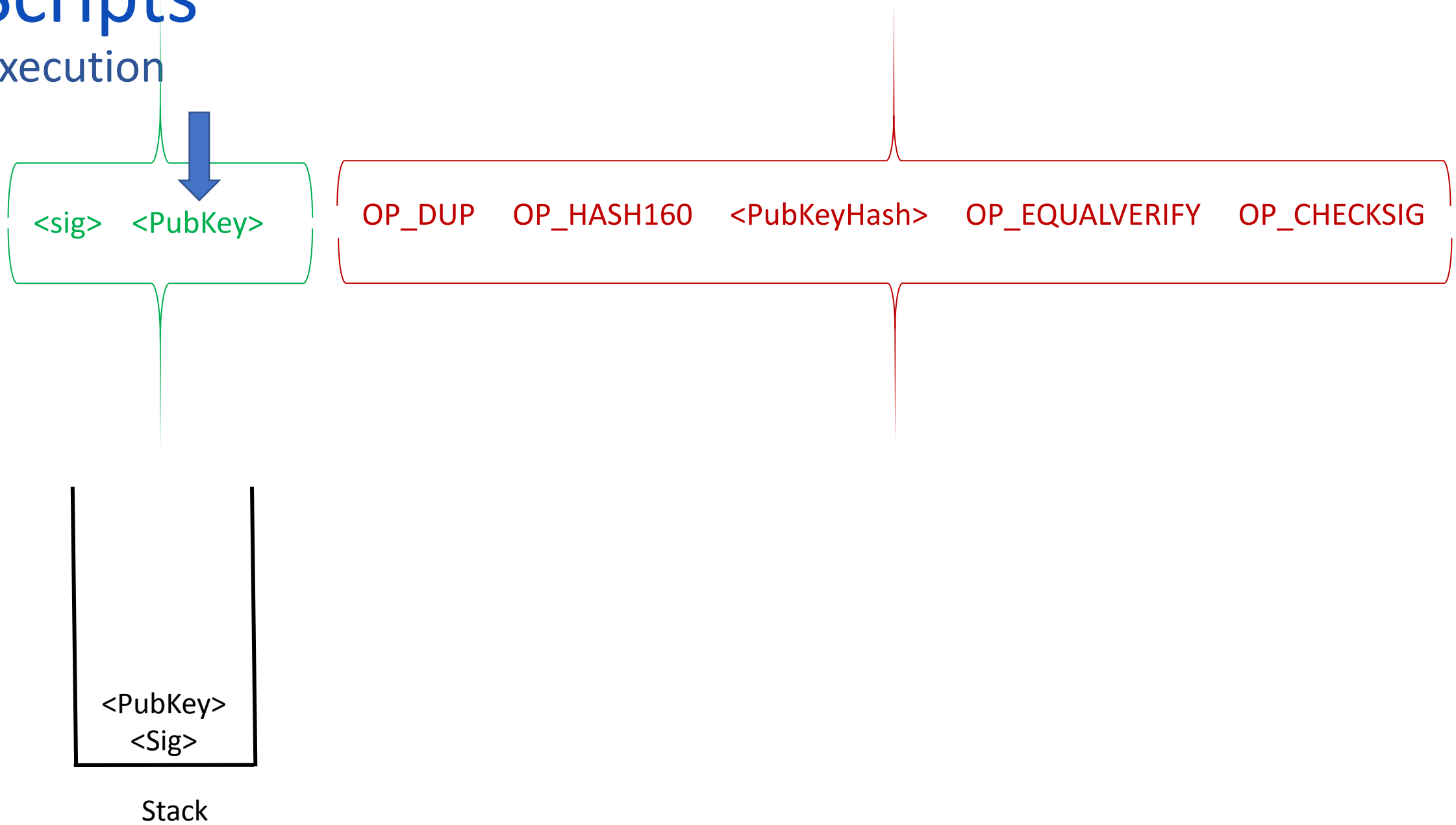
Scripts

Execution



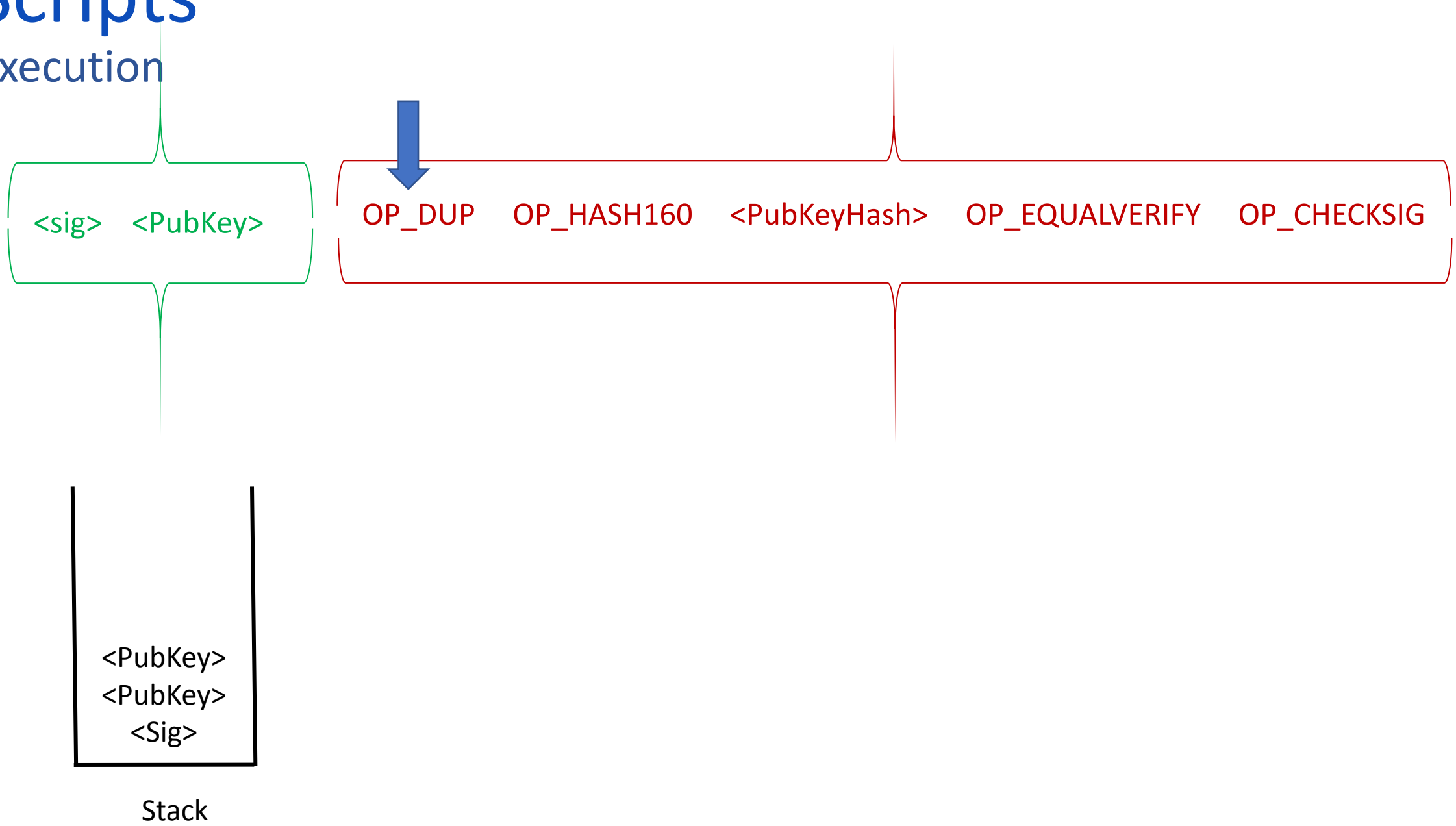
Scripts

Execution



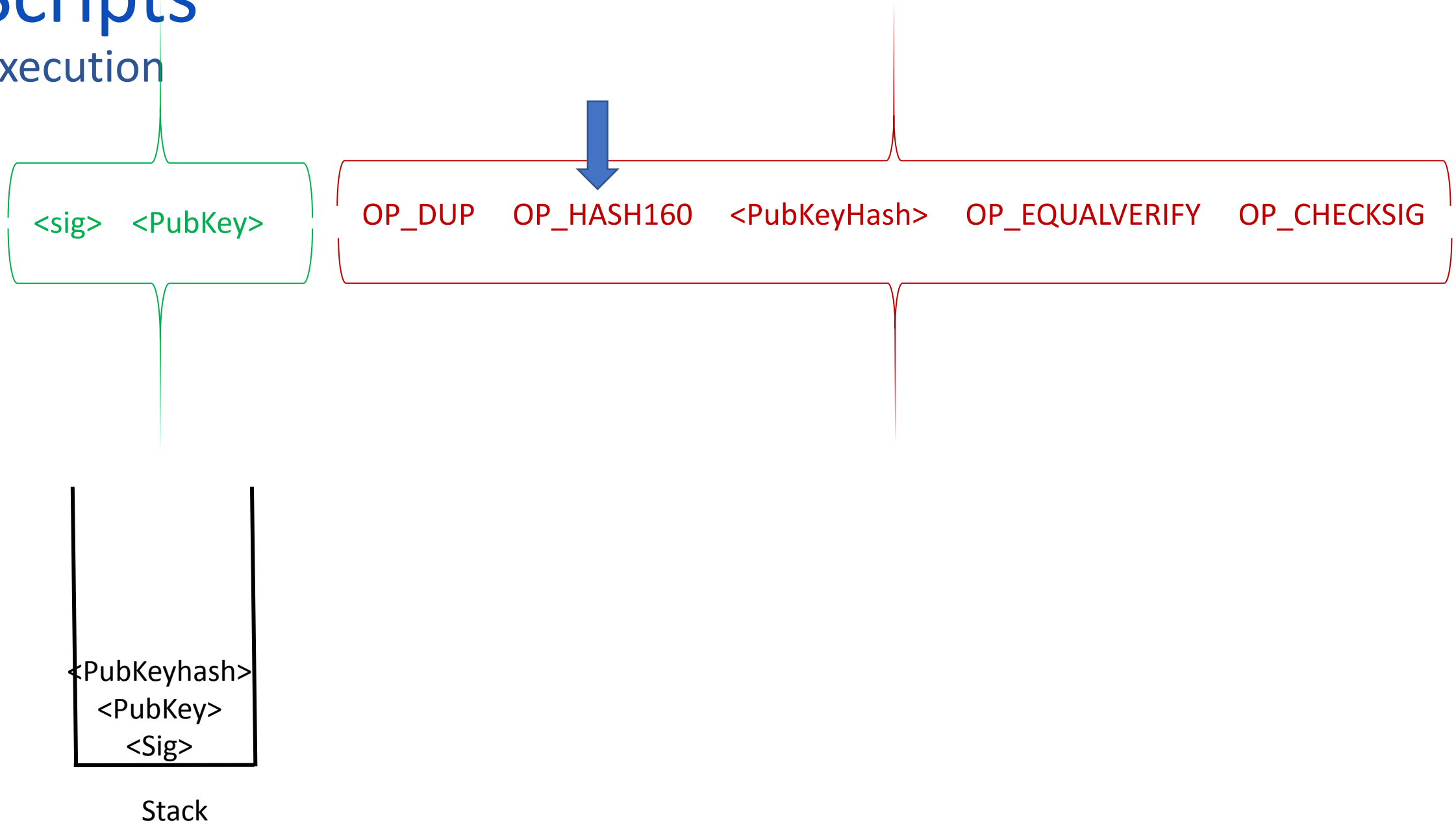
Scripts

Execution



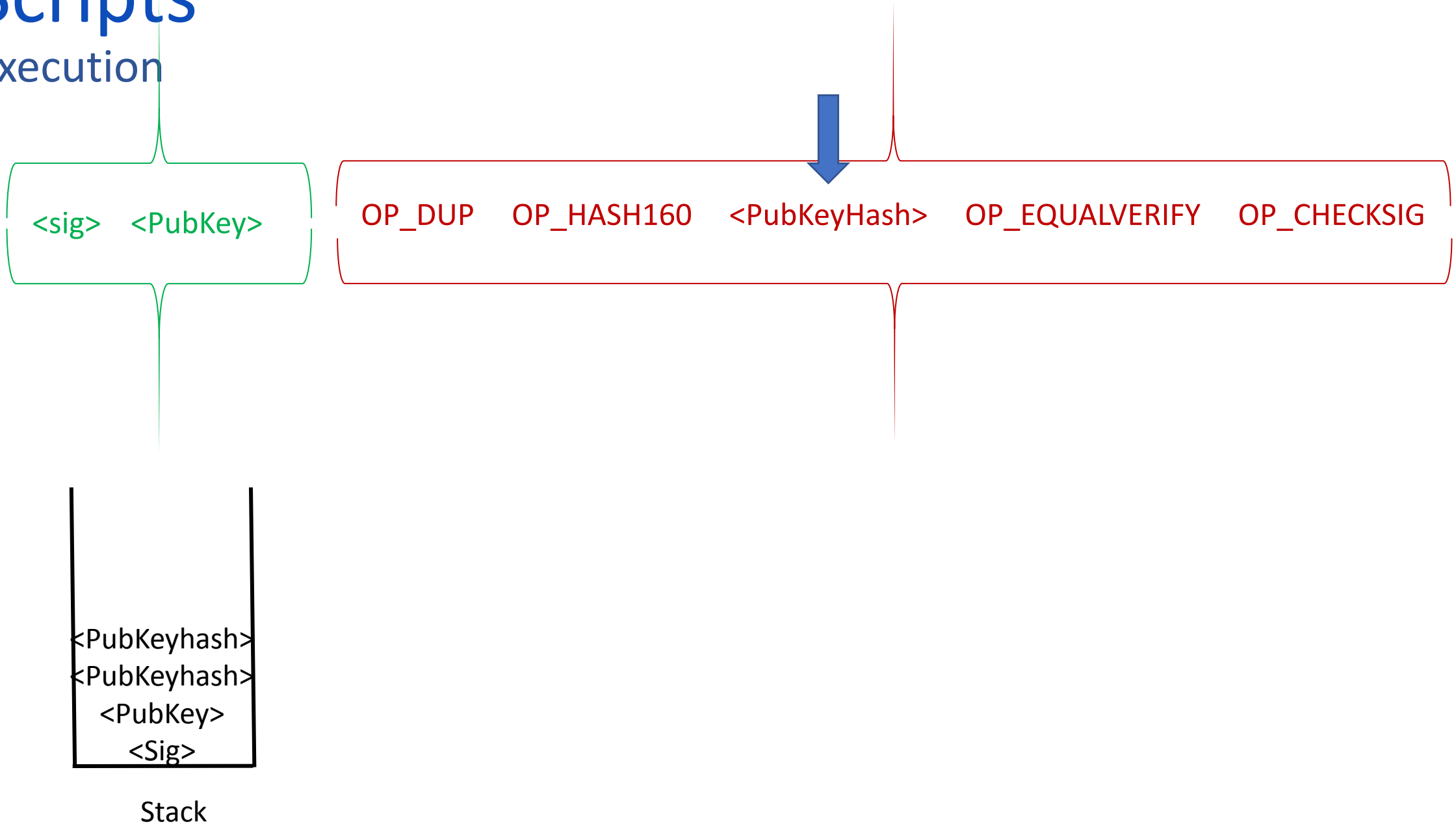
Scripts

Execution



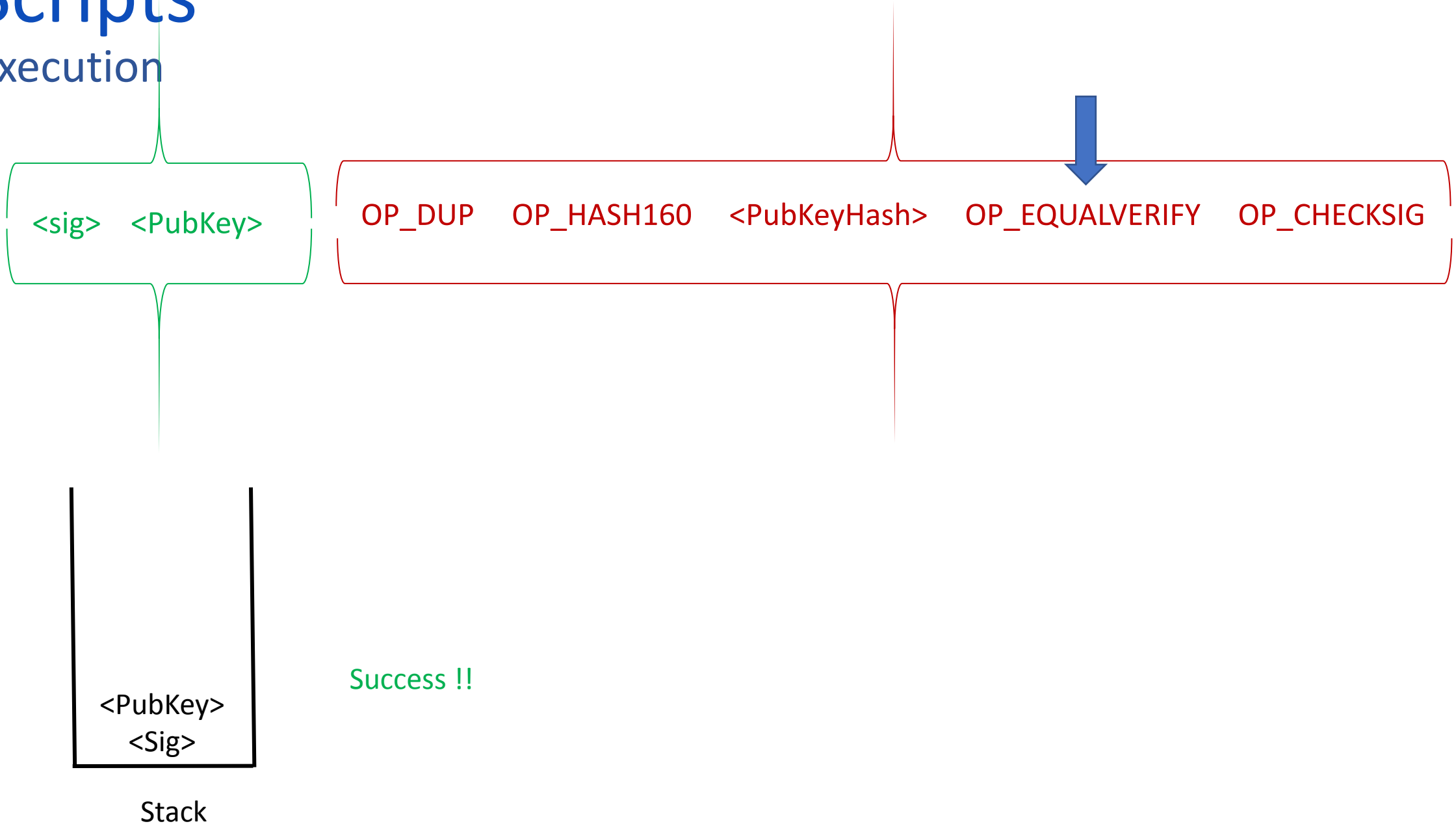
Scripts

Execution



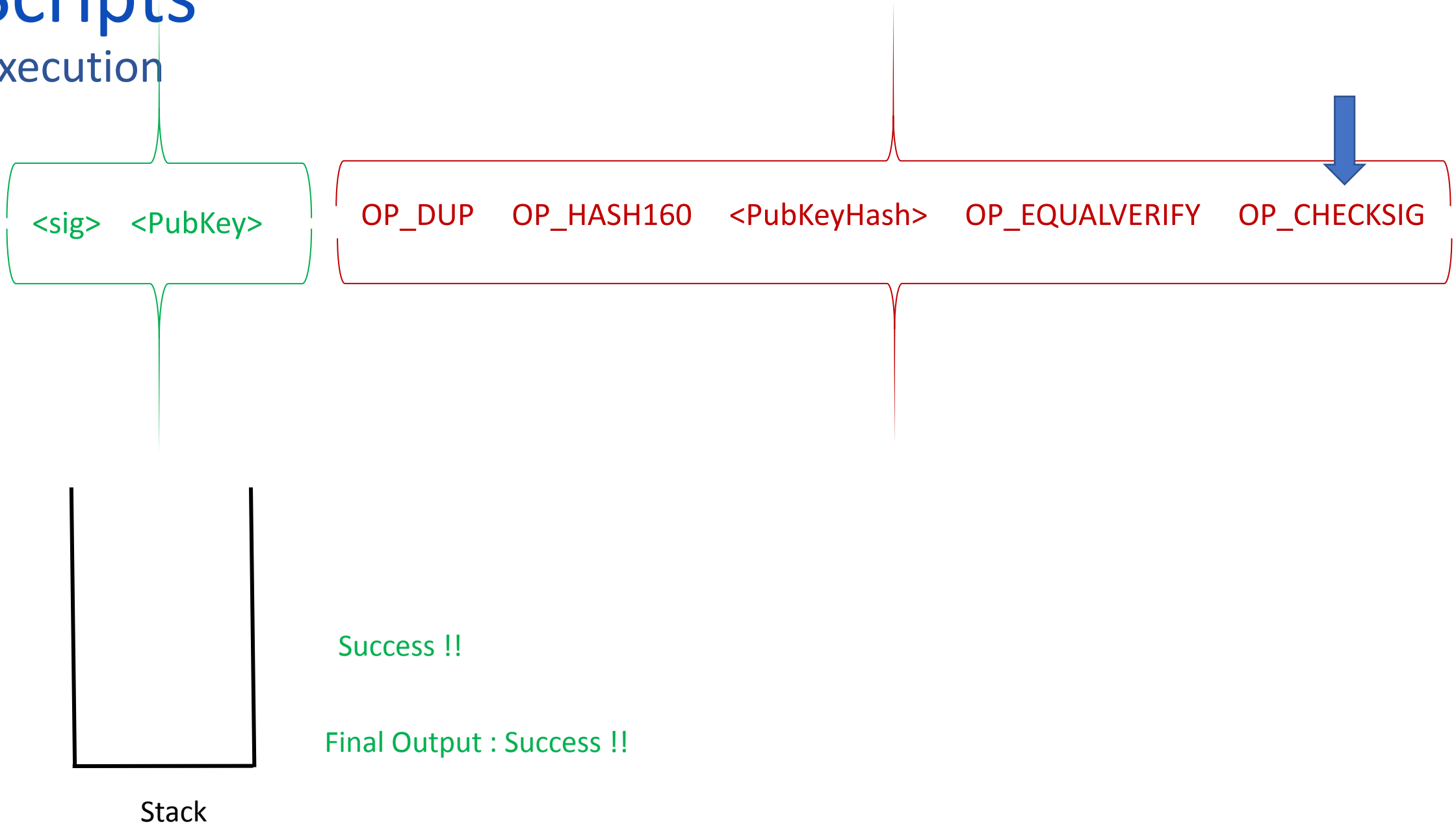
Scripts

Execution



Scripts

Execution



Scripts

Scripts

- P2MS
- Pay to MultiSig
- Lock the coins to multiple public-key
- Unlocking requires some or all signatures corresponding to the public-keys
- t of n MultiSig
- t valid signatures requires
- Known error : pops one extra element from the stack
  - Off-by-one error
  - Alleviated by pushing a dummy variable to the stack

Scripts

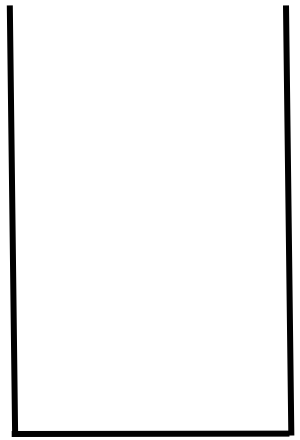
Execution

OP_0 <Sig1> <Sig2>

scriptSig/Unlocking

OP_2 <PubKeyHash_a> <PubKeyHash_b> <PubKeyHash_c> OP_3 CheckMultiSig

scriptPubKey/locking

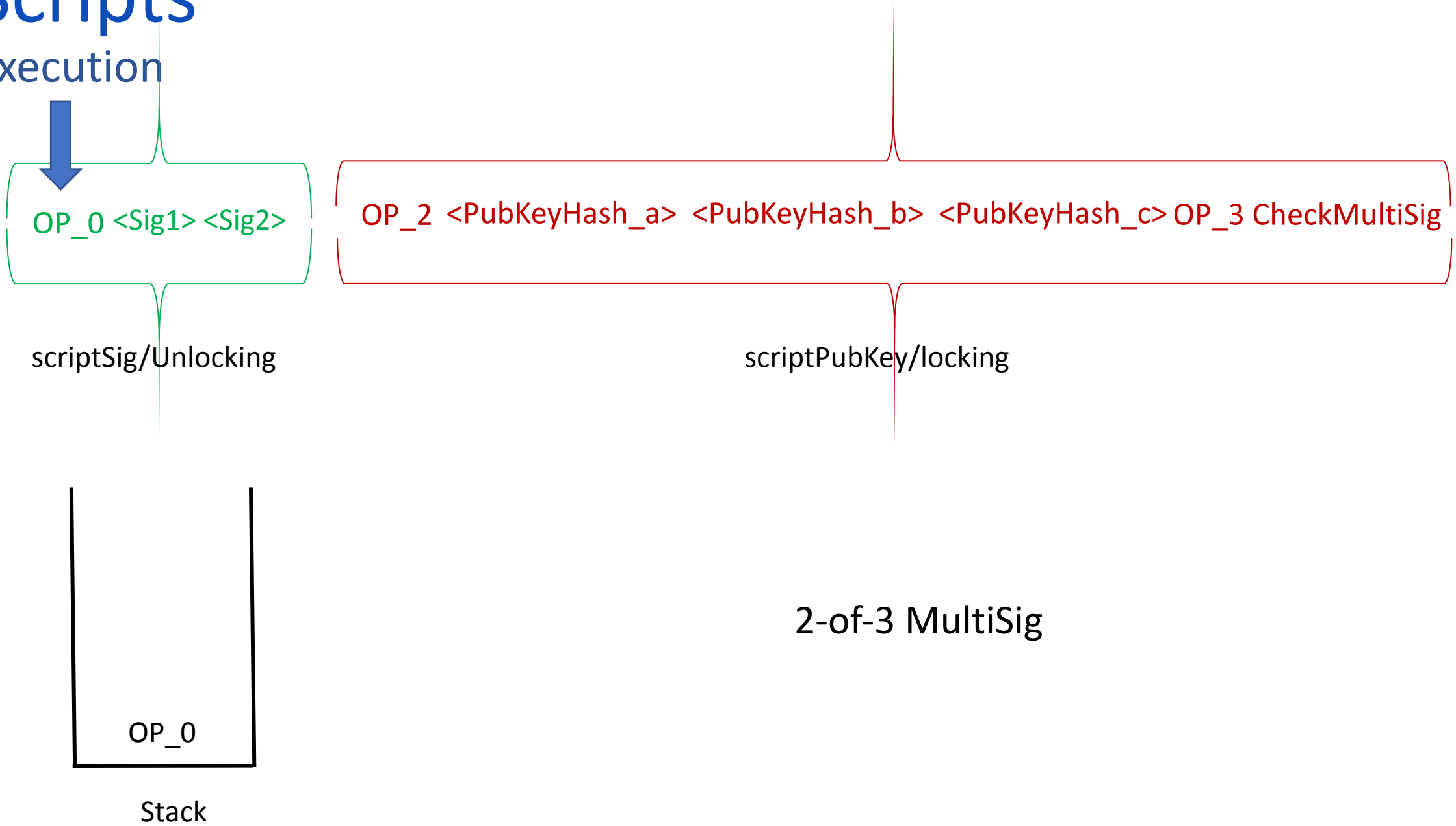


Stack

2-of-3 MultiSig

Scripts

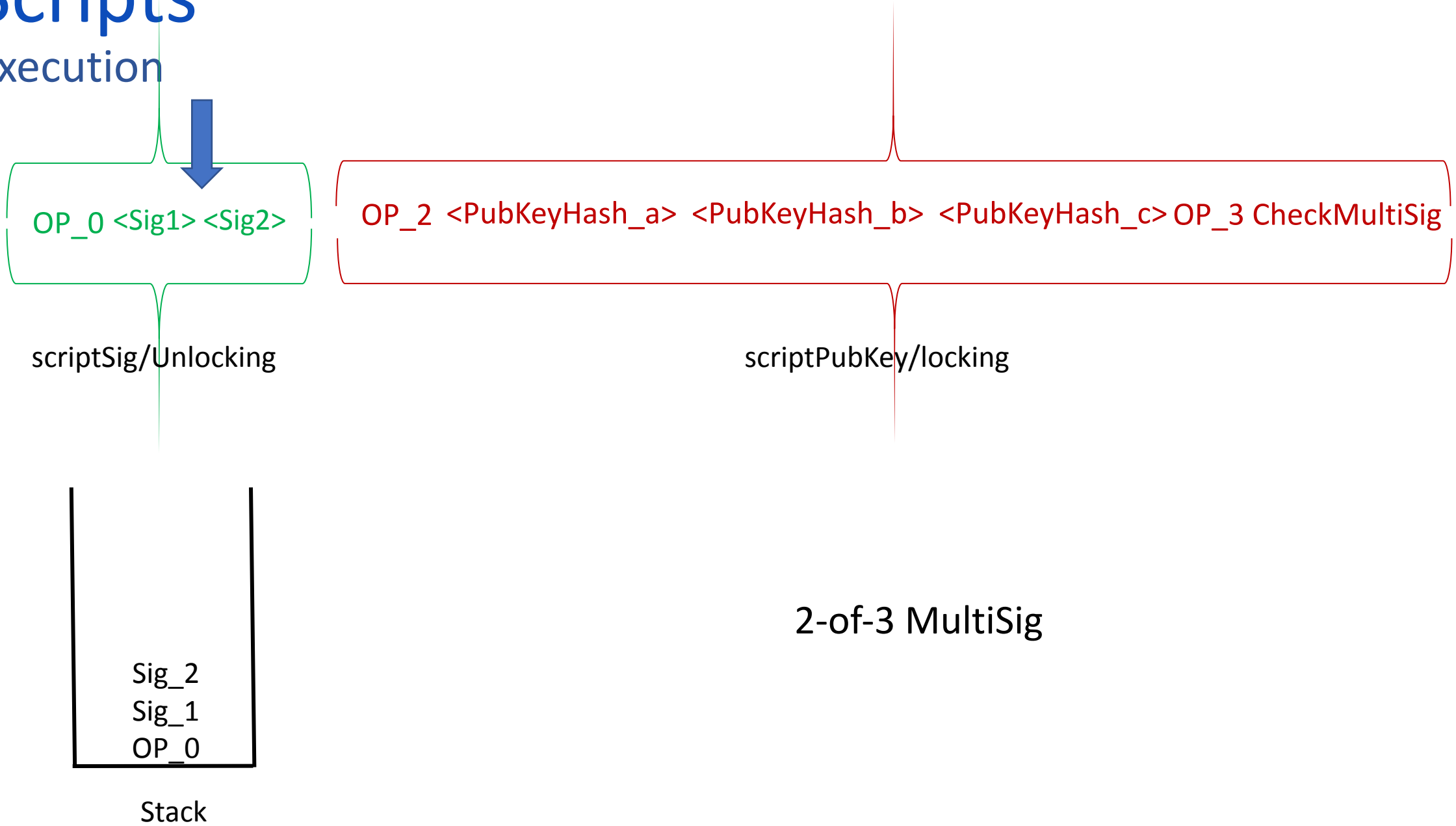
Execution



2-of-3 MultiSig

Scripts

Execution



2-of-3 MultiSig

Scripts

Execution

OP_0 <Sig1> <Sig2>

scriptSig/Unlocking

OP_2 <PubKeyHash_a> <PubKeyHash_b> <PubKeyHash_c> OP_3 CheckMultiSig

scriptPubKey/locking

OP_2
Sig_2
Sig_1
OP_0

Stack

2-of-3 MultiSig

Scripts

Execution

OP_0 <Sig1> <Sig2>

scriptSig/Unlocking

OP_2 <PubKeyHash_a> <PubKeyHash_b> <PubKeyHash_c> OP_3 CheckMultiSig

scriptPubKey/locking

PubKeyHash_a
OP_2
Sig_2
Sig_1
OP_0

Stack

2-of-3 MultiSig

Scripts

Execution

OP_0 <Sig1> <Sig2>

scriptSig/Unlocking

OP_2 <PubKeyHash_a> <PubKeyHash_b> <PubKeyHash_c> OP_3 CheckMultiSig

scriptPubKey/locking

PubKeyHash_c
PubKeyHash_b
PubKeyHash_a
OP_2
Sig_2
Sig_1
OP_0

Stack

2-of-3 MultiSig

Scripts

Execution

OP_0 <Sig1> <Sig2>

scriptSig/Unlocking

OP_2 <PubKeyHash_a> <PubKeyHash_b> <PubKeyHash_c> OP_3 CheckMultiSig

scriptPubKey/locking

PubKeyHash_c
PubKeyHash_b
PubKeyHash_a
OP_2
Sig_2
Sig_1
OP_0

Stack

2-of-3 MultiSig

Scripts

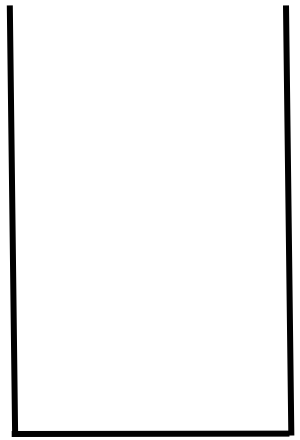
Execution

OP_0 <Sig1> <Sig2>

scriptSig/Unlocking

OP_2 <PubKeyHash_a> <PubKeyHash_b> <PubKeyHash_c> OP_3 CheckMultiSig

scriptPubKey/locking



Stack

PubKeyHash_c PubKeyHash_b PubKeyHash_a Sig_2 Sig_1

Scripts

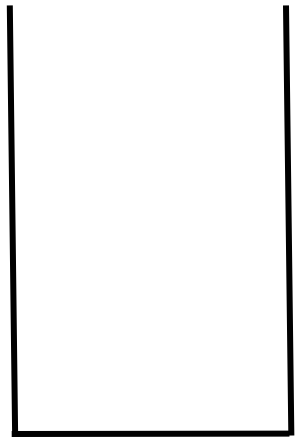
Execution

OP_0 <Sig1> <Sig2>

scriptSig/Unlocking

OP_2 <PubKeyHash_a> <PubKeyHash_b> <PubKeyHash_c> OP_3 CheckMultiSig

scriptPubKey/locking



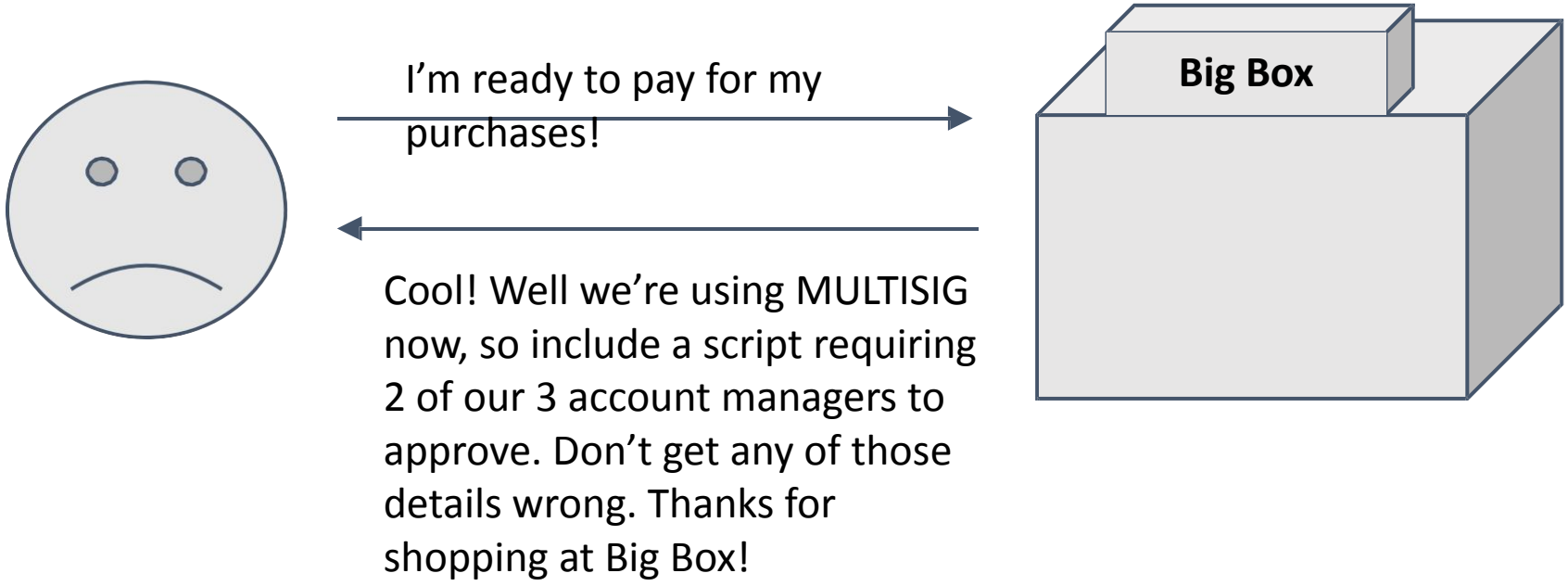
Stack

PubKeyHash_c PubKeyHash_b PubKeyHash_a Sig_2 Sig_1

- Repeats for all signatures provided
- Returns Success only 2 out of three signature matches

Scripts

P2MS application



Scripts

Scripts

- P2SH
- Pay to ScriptHash
- Lock the coins to hash of a script
- Unlocking has two steps
  - Provide the script that has the given hash
  - provide data (e.g., signatures) that will evaluate to success for the hash
- Allows to create any complex logic inside the unlocking script
  - Does not burden the payer

Scripts

Execution

Redeem script/logic and associated data which evaluates to SUCCESS

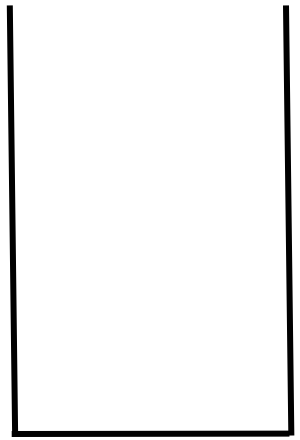
OP_0 <Sig1> <Sig2> <RedeemScript>

scriptSig/Unlocking

OP_HASH160 <ScriptHash> OP_EQUALVERIFY

scriptPubKey/locking

Assume the redeem script is 2-of-3 MultiSig



Stack

All the burden on payee to provide correct data !!!!

Scripts

Execution

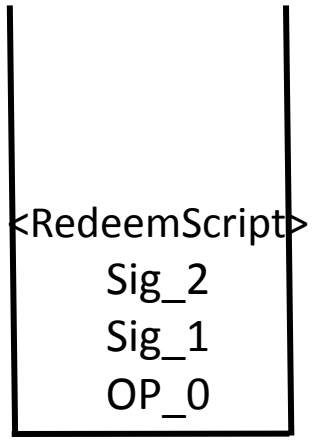
OP_0 <Sig1> <Sig2> <RedeemScript>

scriptSig/Unlocking

OP_HASH160 <ScriptHash> OP_EQUALVERIFY

scriptPubKey/locking

Assume the redeem script is 2-of-3 MultiSig



Stack

Scripts

Execution

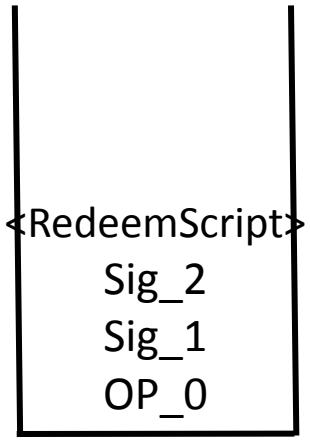
OP_0 <Sig1> <Sig2> <RedeemScript>

scriptSig/Unlocking

OP_HASH160 <ScriptHash> OP_EQUALVERIFY

scriptPubKey/locking

Assume the redeem script is 2-of-3 MultiSig



Stack

H(RedeemScript) $\stackrel{?}{=}$ <ScriptHash>

Scripts

Execution

OP_0 <Sig1> <Sig2> <RedeemScript>

scriptSig/Unlocking

OP_HASH160 <ScriptHash> OP_EQUALVERIFY

scriptPubKey/locking

Assume the redeem script is 2-of-3 MultiSig

OP_3
PubKeyHash_c
PubKeyHash_b
PubKeyHash_a
OP_2
Sig_2
Sig_1
OP_0

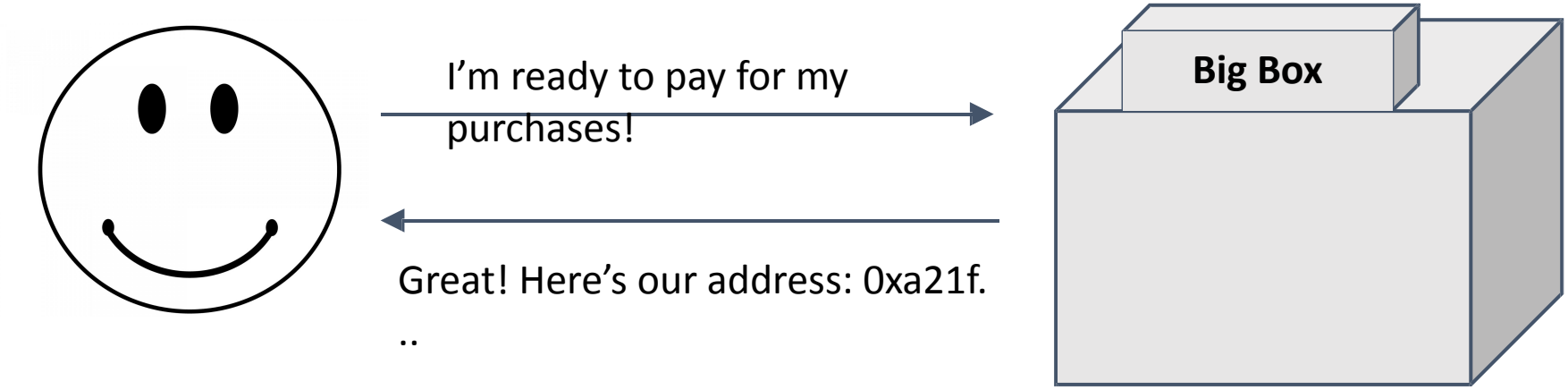
Stack

Like the 2-of-3 MultiSig shown before

Can anyone provide the <ScriptHash> ?

Scripts

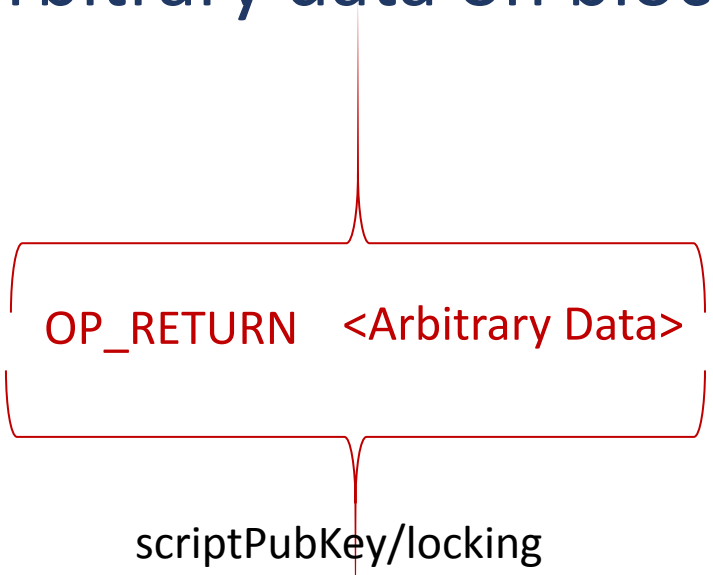
P2MS application



Scripts

NULL_DATA

- OP_RETURN
- Used to store arbitrary data on blockchains



- Always returns Success
- Data is forever stored in blockchain
- What will be the unlocking script?

Scripts

Scripts

- Why?
- People used to put arbitrary data in the <PKHASH> section of P2PKH

OP_DUP OP_HASH160 <arbitrary_data> OP_EQUALVERIFY OP_CHECKSIG

- Unredeemable !!!
- Increases the UTXO size

Links

1. Bitcoin opcodes
   https://wiki.bitcoinsv.io/index.php/Opcodes_used_in_Bitcoin_Script

https://bitcoin.clarkmoody.com/dashboard/

OP_return
https://www.blockchain.com/explorer/transactions/btc/d29c9c0e8e4d2a9790922af73f0b8d51f0bd4bb19940d9cf910ead8fbe85bc9b
Size : 10000 bytes
P2pk
https://www.blockchain.com/explorer/transactions/btc/ab3f542ad5add941f8ba4282cebab0cca81266a83cab6b98ebb8ae730477a570
Multisig
https://www.blockchain.com/explorer/transactions/btc/949591ad468cef5c41656c0a502d9500671ee421fadb590fbc6373000039b693
Change address
https://www.blockchain.com/explorer/transactions/btc/0a1c0b1ec0ac55a45b1555202daf2e08419648096f5bcc4267898d420dffef87
Forgets to put his own address back
https://www.blockchain.com/explorer/transactions/btc/d80feae624c18067044e1bca0917bdb0c42f81ff1e7b9b2febacfced1ed7f691

Acknowledgement

- The material of this lecture is mostly due to Prof. Arvind Narayanan's Lecture at Princeton and his book named Bitcoin and cryptocurrency technologies (chapter 3)

The end !!!