Bitcoin:
Programming the Future of Money

Topics in Computer Science - ITCS 4010/5010, Spring 2025

Dr. Christian Kümmerle

Lecture 18

Bitcoin Script II: Pay-to-Script-Hash

Some figures are taken from:

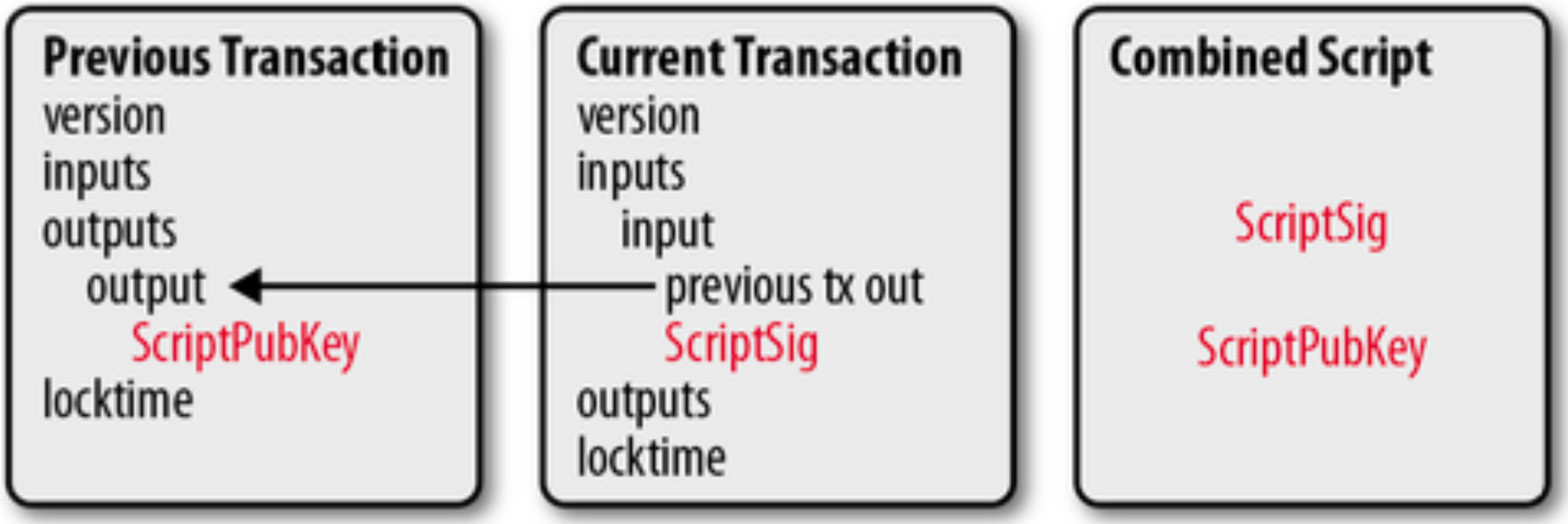
- “Mastering Bitcoin: Programming the Open Blockchain”, (Andreas Antonopoulos, David Harding), 3rd Edition, O’Reilly, 2023.

Bitcoin Script

RECAP: BITCOIN SCRIPT

- “Bitcoin Script” or “Script” is the internal programming language used for **prescribing conditions under which a output can be spent**, and for **providing flexible, yet secure validation rules** to satisfy the stated spending conditions.
- It is akin to the Forth programming language.
- Stack-based: Pushing and popping values of a stack.
- No ability to execute loops
- **Not a Turing-complete** programming language!
- Designed to be simple and operate without bugs -> Crucial for Security

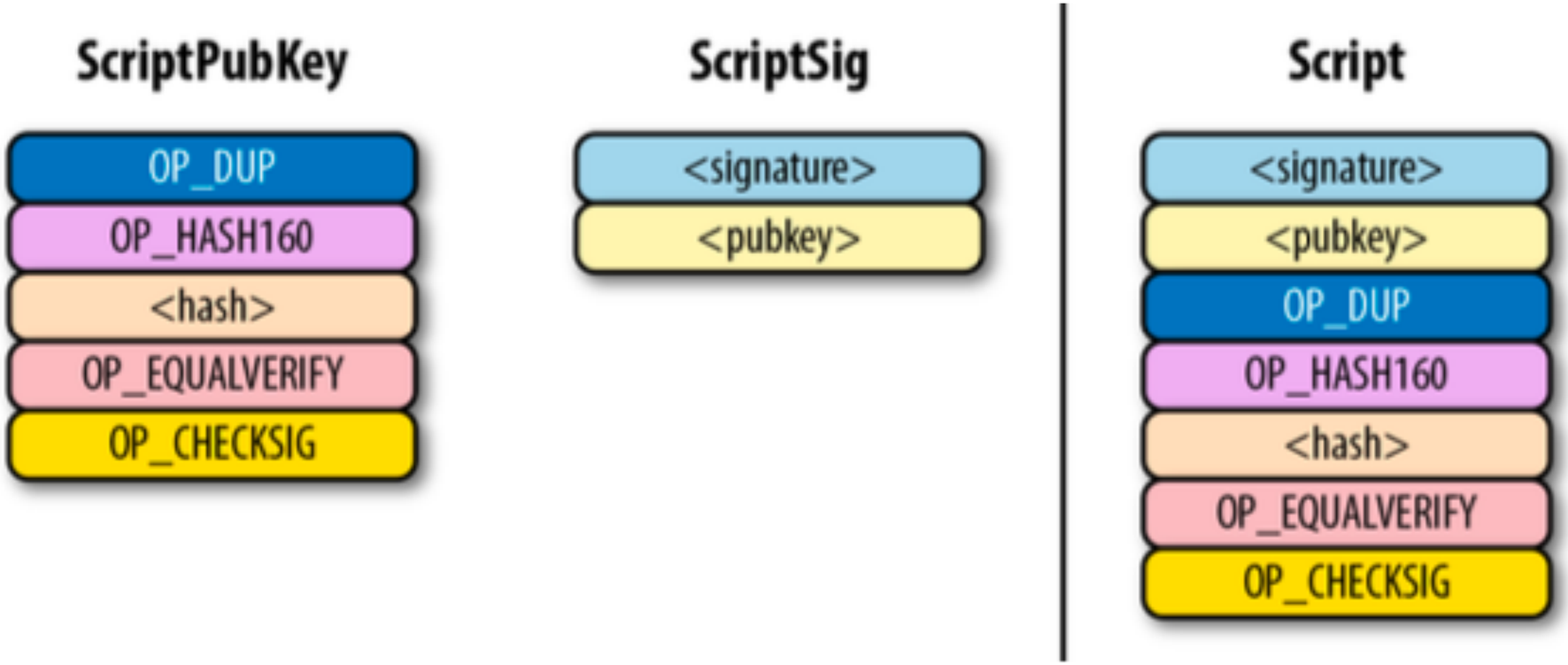
OUTPUT SCRIPT, INPUT SCRIPT AND COMBINED SCRIPT



Bitcoin Script execution rules:

- Input Script (from “ScriptSig” field of input) is executed (resulting in a stack)
- If executed without errors -> copy stack of input script, executed output script on it. If executed with errors -> Return FALSE
- Output Script (from “ScriptPubKey” field of output) is then executed.
- If resulting final stack only contains TRUE -> spending of output is valid.
- For SegWit outputs: “ScriptSig” of corresponding input empty, but corresponding “Witness” field of input used instead

EXAMPLE: BITCOIN SCRIPT FOR P2PKH



Note:

- The evaluation of P2SH scripts is slightly different than the one of other scripts due to the existence of the redeem script. This is specified in BIP0016.
- In particular, the rules are:
  - Validation fails if there are any operations other than "push data" operations in the ScriptSig.
  - Normal validation is done: an initial stack is created from the signatures and <RedeemScript>, and the hash of the script is computed and validation fails immediately if it does not match the hash in the outpoint.
  - <RedeemScript> is popped off the initial stack, and the transaction is validated again using the popped stack and the deserialized <RedeemScript> as the ScriptPubKey

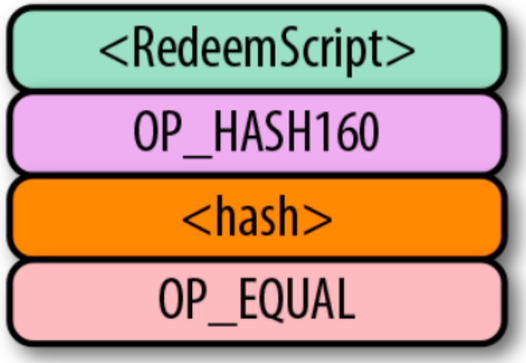
- See Chapter 8 on “Pay-to-Script-Hash” in Programming Bitcoin for a detailed discussion

Key Features of P2SH

- **Easy to work with** as user:
  We have “address” format, unlike for P2MS
- **More flexible script** patterns possible.
- **Burden of data storage** of script shifted from output to **input script** (i.e., only on-chain if output is actually spendable & spent)

SPECIAL TREATMENT OF REDEEM SCRIPT DATA IN P2SH

Special Script pattern in P2SH:



- If the script pattern above is executed:
  - If result on stack is “0” (OP_0 as opcode) -> Validation fails immediately
  - If result on stack is “1” (OP_1 as opcode) -> “1” is removed from stack, and <RedeemScript> is parsed and added to the Script command set / opcode set. The command set is then applied to the remaining stack.

EXAMPLE: IMPLEMENT T-OF-K MULTISIGNATURE IN P2SH

Input Script:

- E.g., <Signature 2> <Signature k> <Redeem Script>

Output Script:

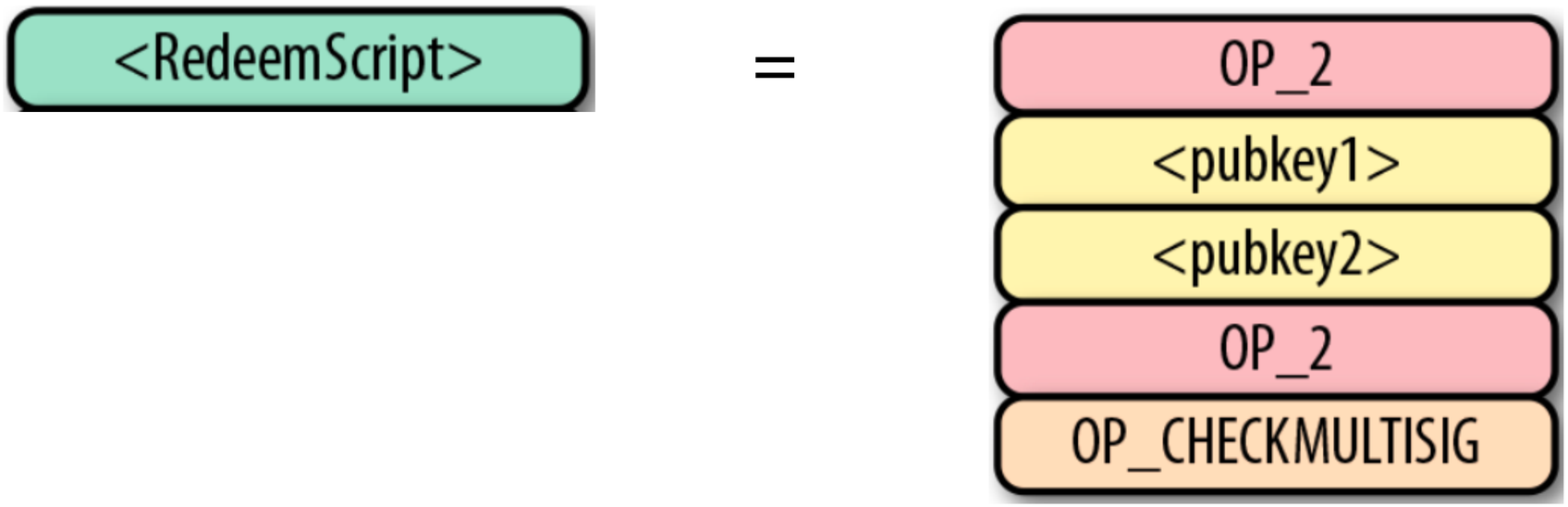
- OP_HASH160 <20-byte hash of redeem script> OP_EQUAL

Redeem Script:

- t <Public Key 1> <Public Key 2> ... <Public Key k> k OP_CHECKMULTISIG

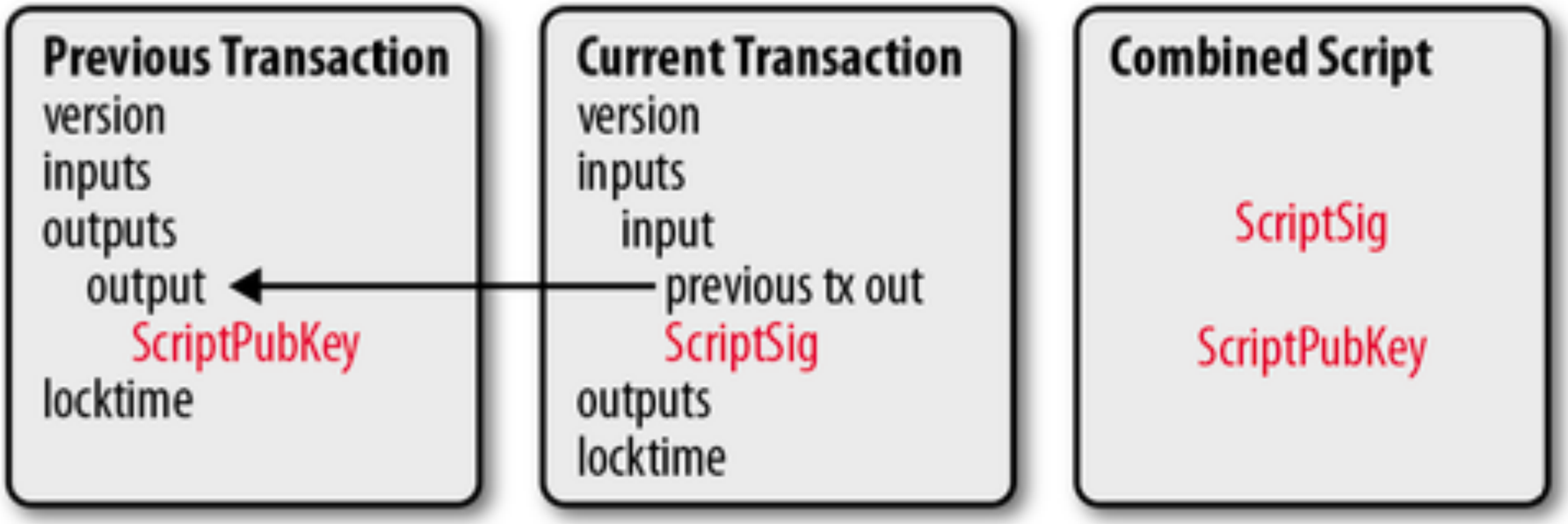
Now: Take 2-of-2 Multisig.

EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH

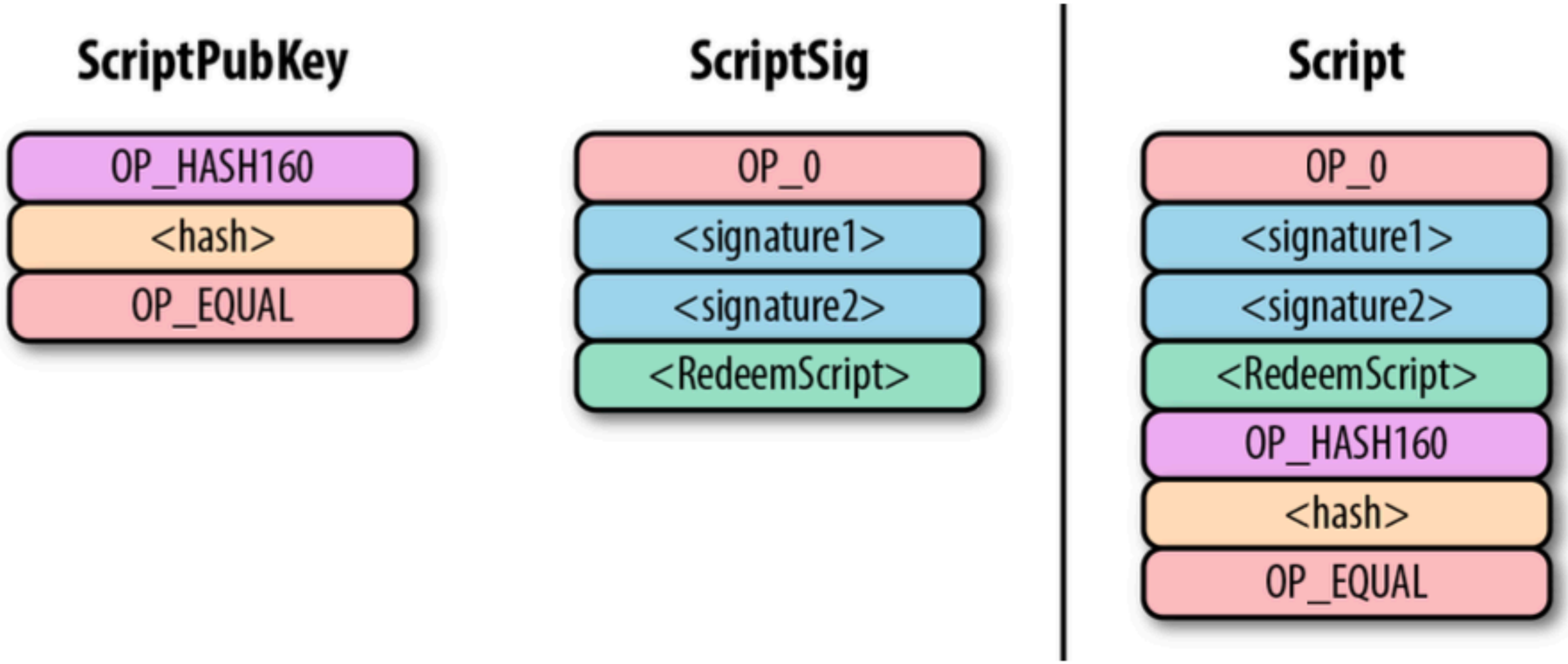


Redeem Script of a 2-of-2 Multisignature spending condition

OUTPUT SCRIPT, INPUT SCRIPT AND COMBINED SCRIPT

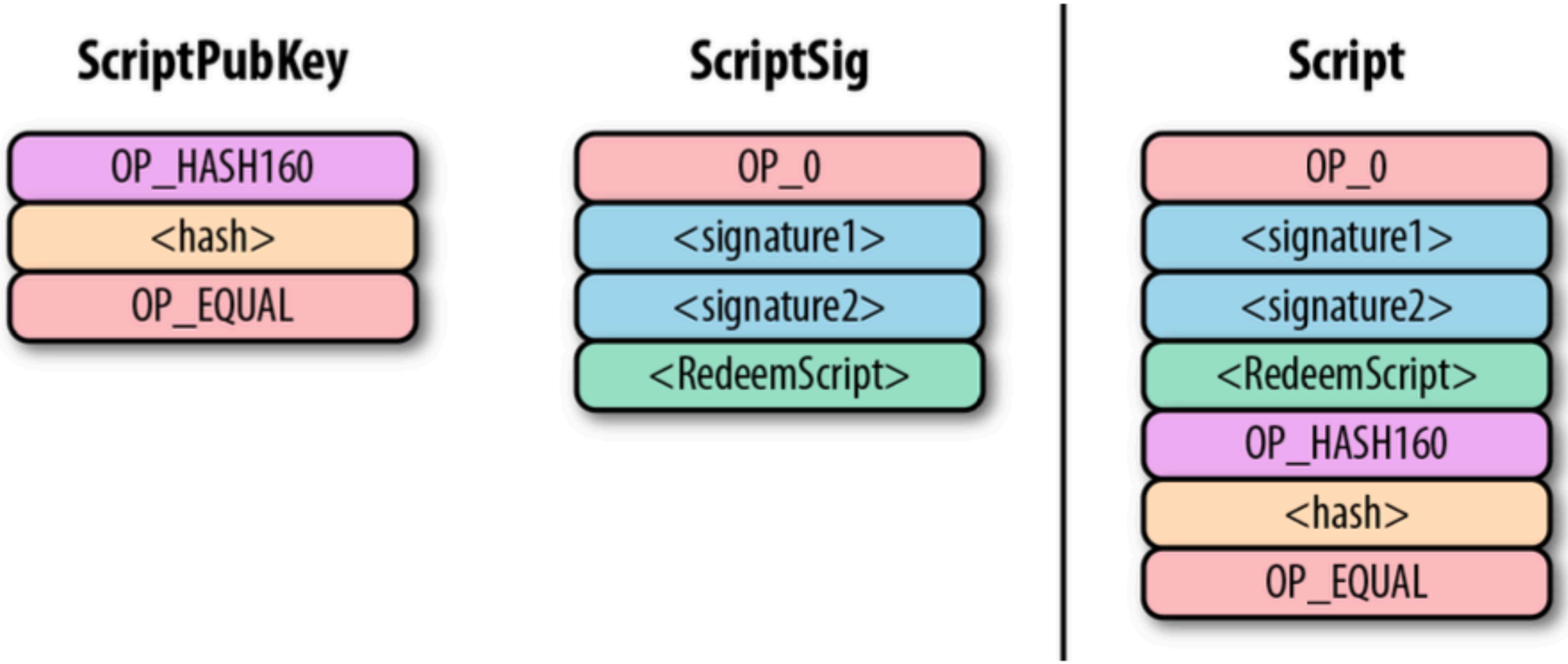


EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH



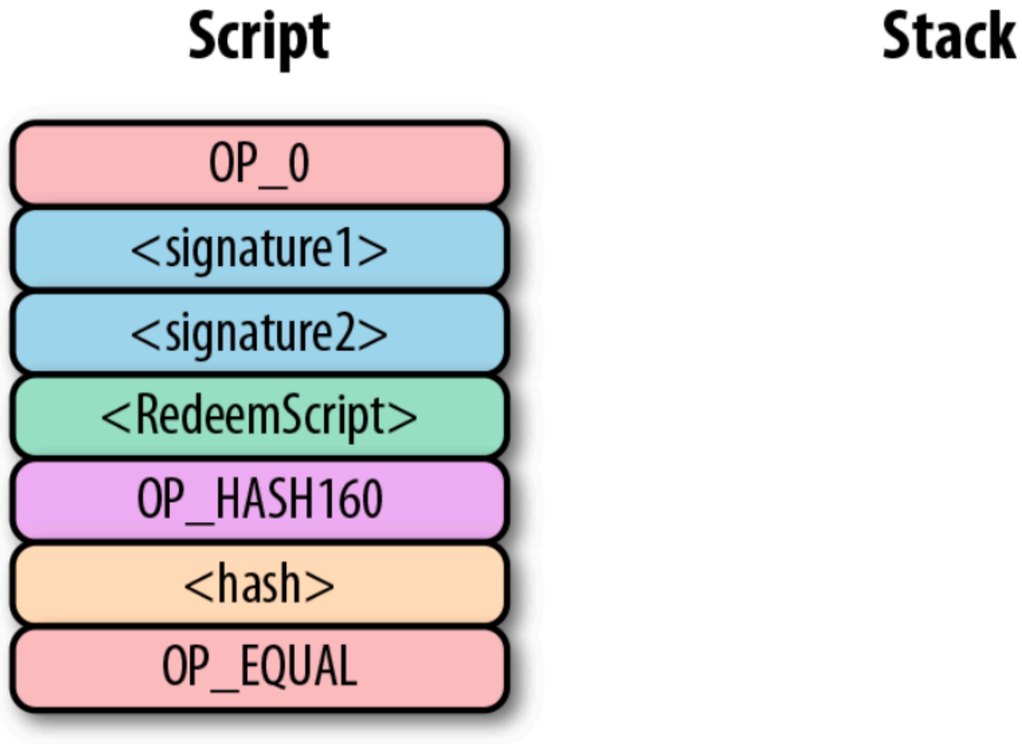
Combined Input & Output Script of P2SH address format

EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH

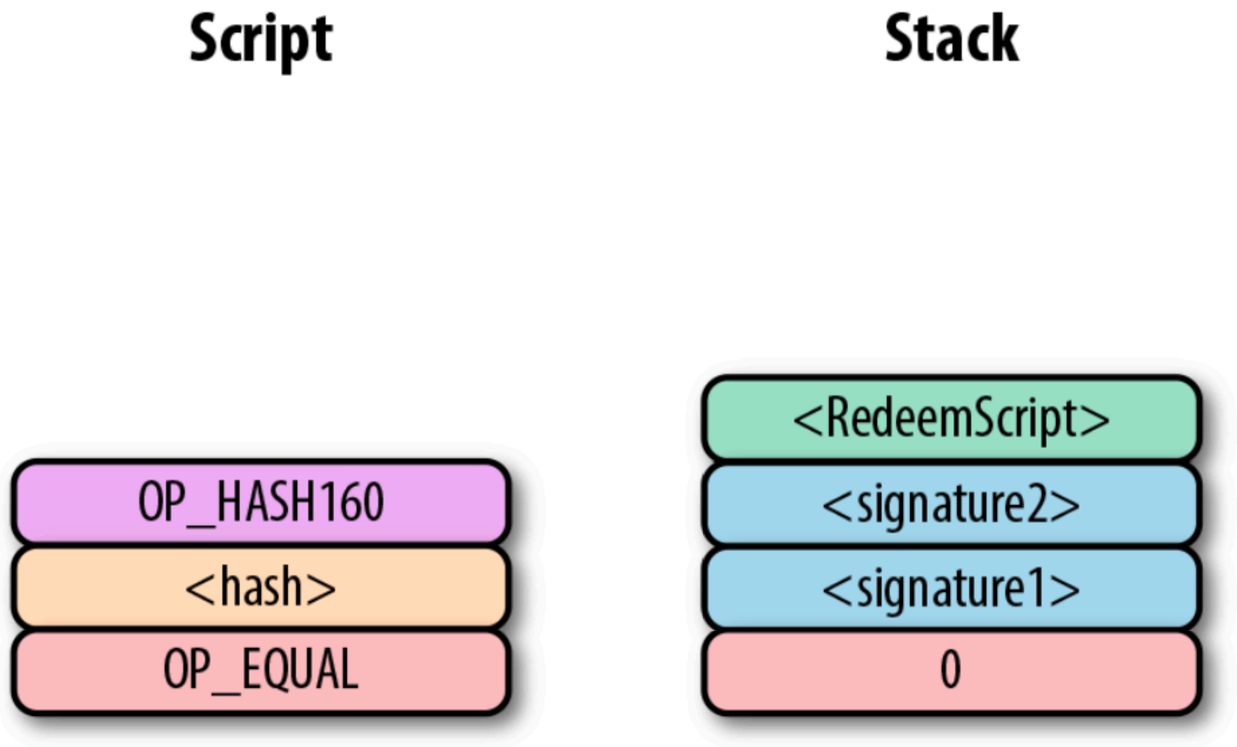


Combined Input & Output Script of P2SH address format

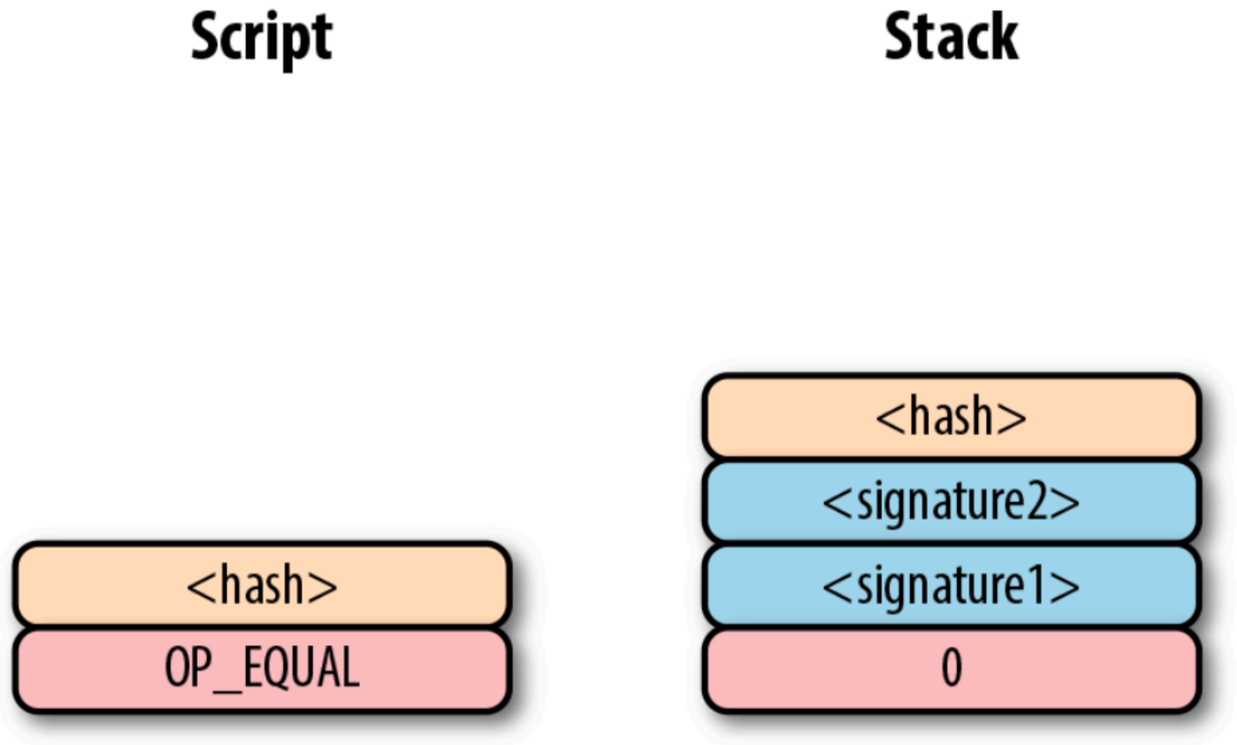
EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH



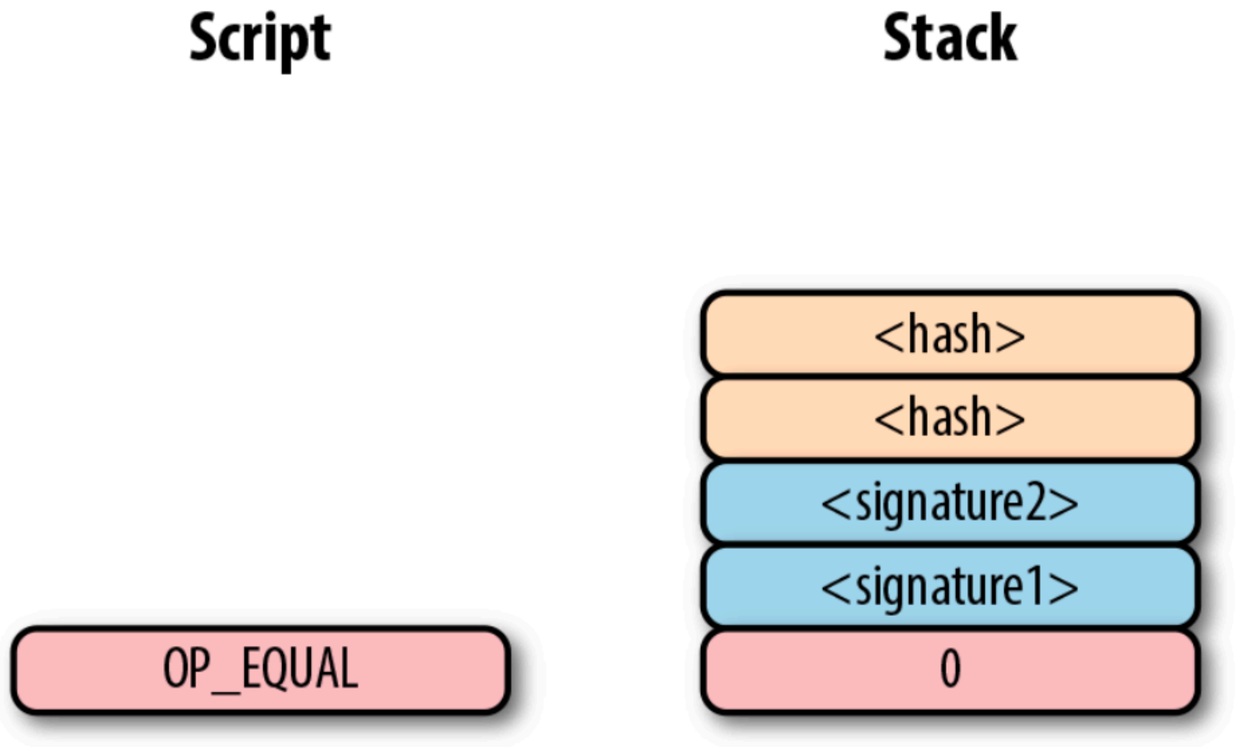
EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH



EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH



EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH



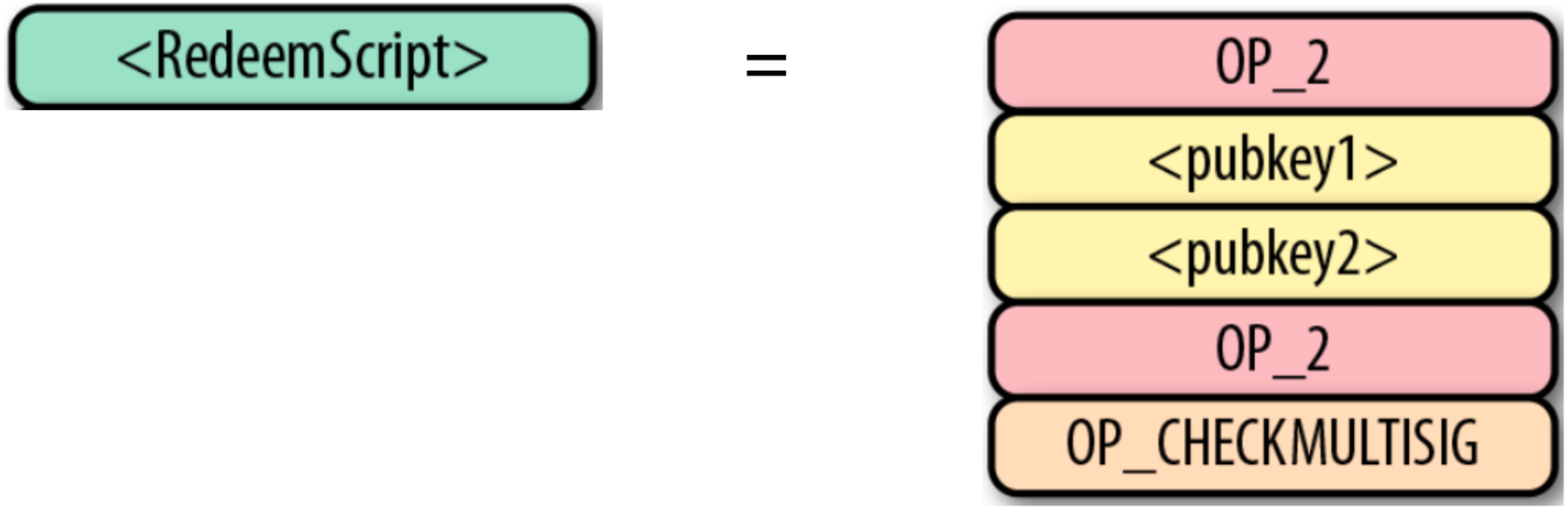
EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH

Script

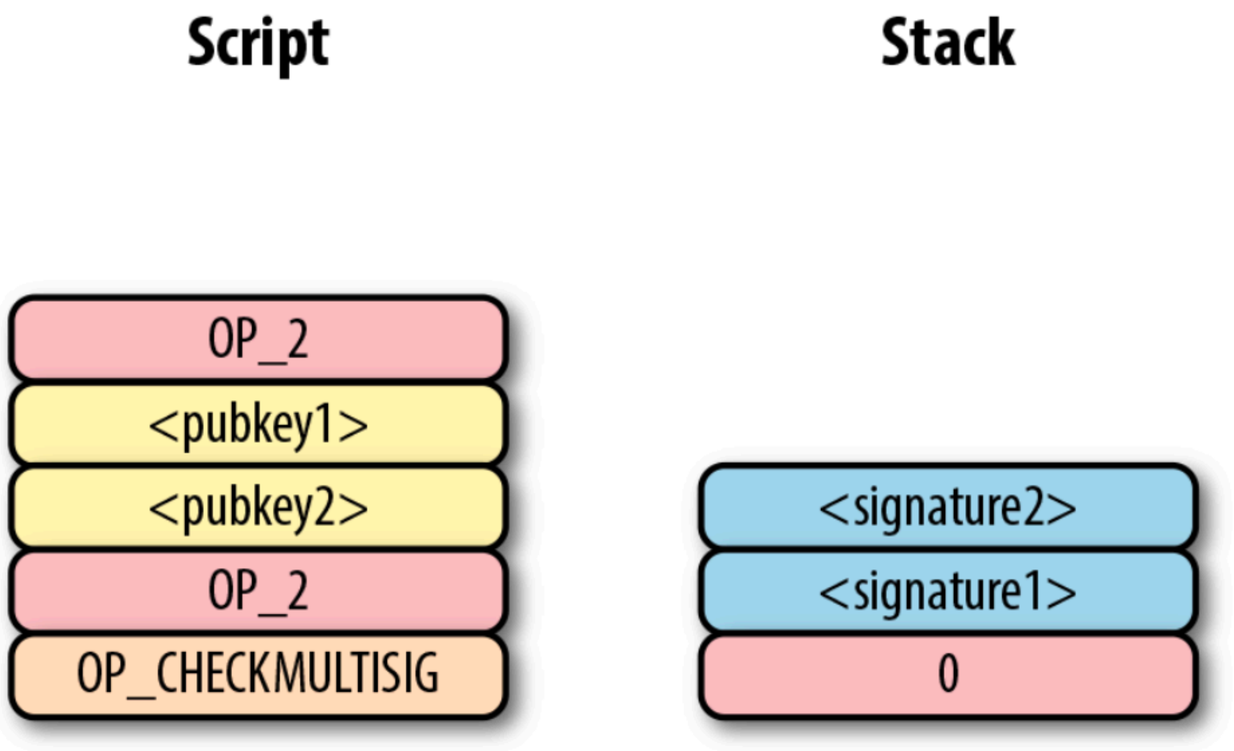
Stack

EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH

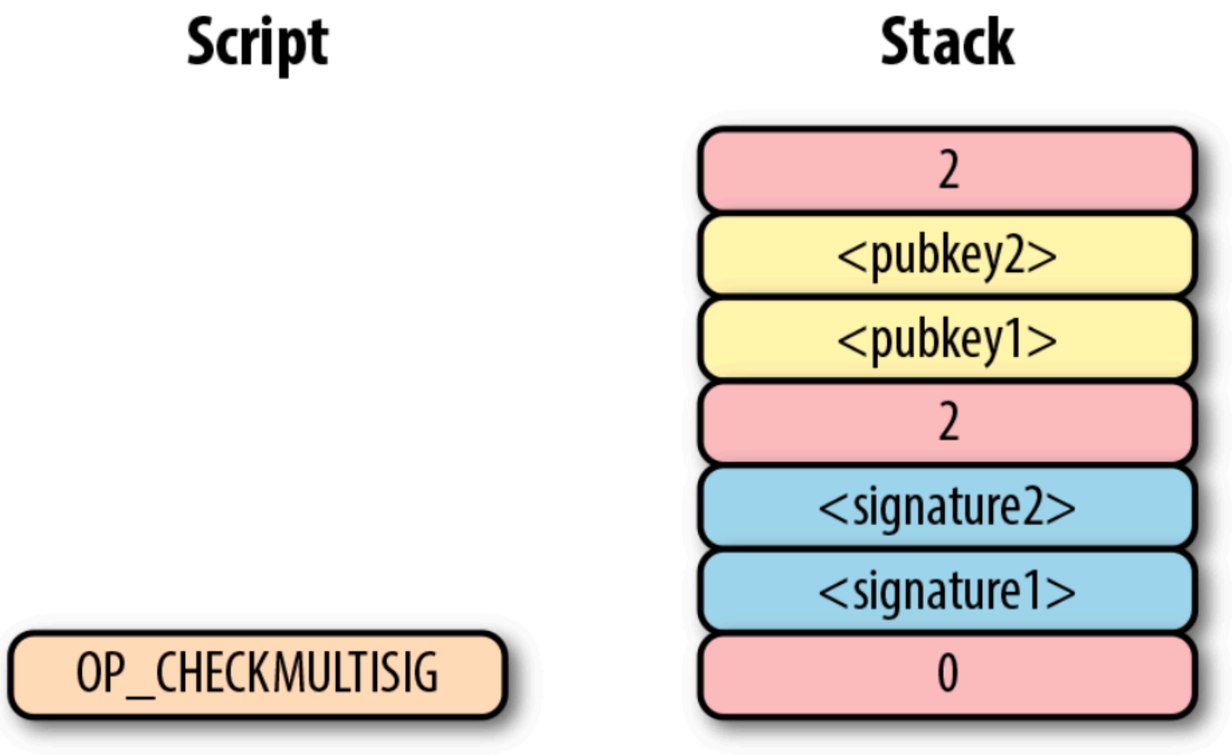
Recall:



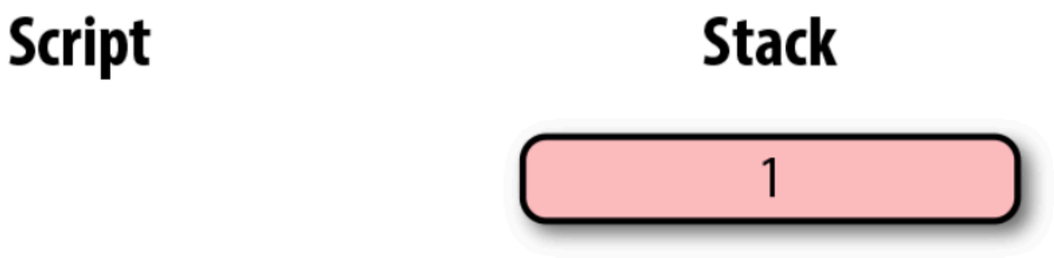
EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH



EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH



EXAMPLE: 2-OF-2 MULTISIGNATURE IN P2SH



BITCOIN SCRIPT: IF-ELSE STATEMENTS

```
condition
IF
  code to run when condition is true
OP_ELSE
  code to run when condition is false
OP_ENDIF
code to run in either case
```

BITCOIN SCRIPT: IF-ELSE STATEMENTS - RELEVANT OPCODES

- OP_0 (also called OP_FALSE): 0x00.
  Empty array of bytes is pushed onto the stack.
- OP_1 (also called OP_TRUE): 0x51.
  Pushes number 1 to stack>
- OP_IF: 0x63. Pops the top stack item, if item is not 0, executes following opcodes until OP_ELSE or OP_ENDIF. Otherwise, skip following opcodes until OP_ELSE or OP_ENDIF is encountered.
- OP_ELSE: 0x67.
- OP_ENDIF: 0x68.
- OP_CHECKSIG: 0xac. The entire transaction's outputs, inputs, and script are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwise.

EXERCISE

Write an appropriate redeem script for a 1-of-2 multi signature spending condition without the use of OP_CHECKMULTIGSIG (but relying on an if-else-statement)

EXERCISE

Write an appropriate redeem script for a 1-of-2 multi signature spending condition without the use of OP_CHECKMULTIGSIG (but relying on an if-else-statement)

REDEEM SCRIPT FOR 1-OF-2 MULTISIGNATURE SPENDING CONDITION

<RedeemScript>:

Script

Stack

```
OP_IF
 <Pubkey1>
OP_CHECKSIG
OP_ELSE
 <Pubkey2>
OP_CHECKSIG
OP_ENDIF
```

Q: How do we need to choose rest of ScriptSig to satisfy redeem script in either case?

CORRESPONDING SCRIPTSIG: 1-OF-2 MULTISIGNATURE SPENDING CONDITION

Case 1: We have a signature for <Pubkey 1>.

ScriptSig:

```
<Signature 1>
 OP_1
<RedeemScript>
```

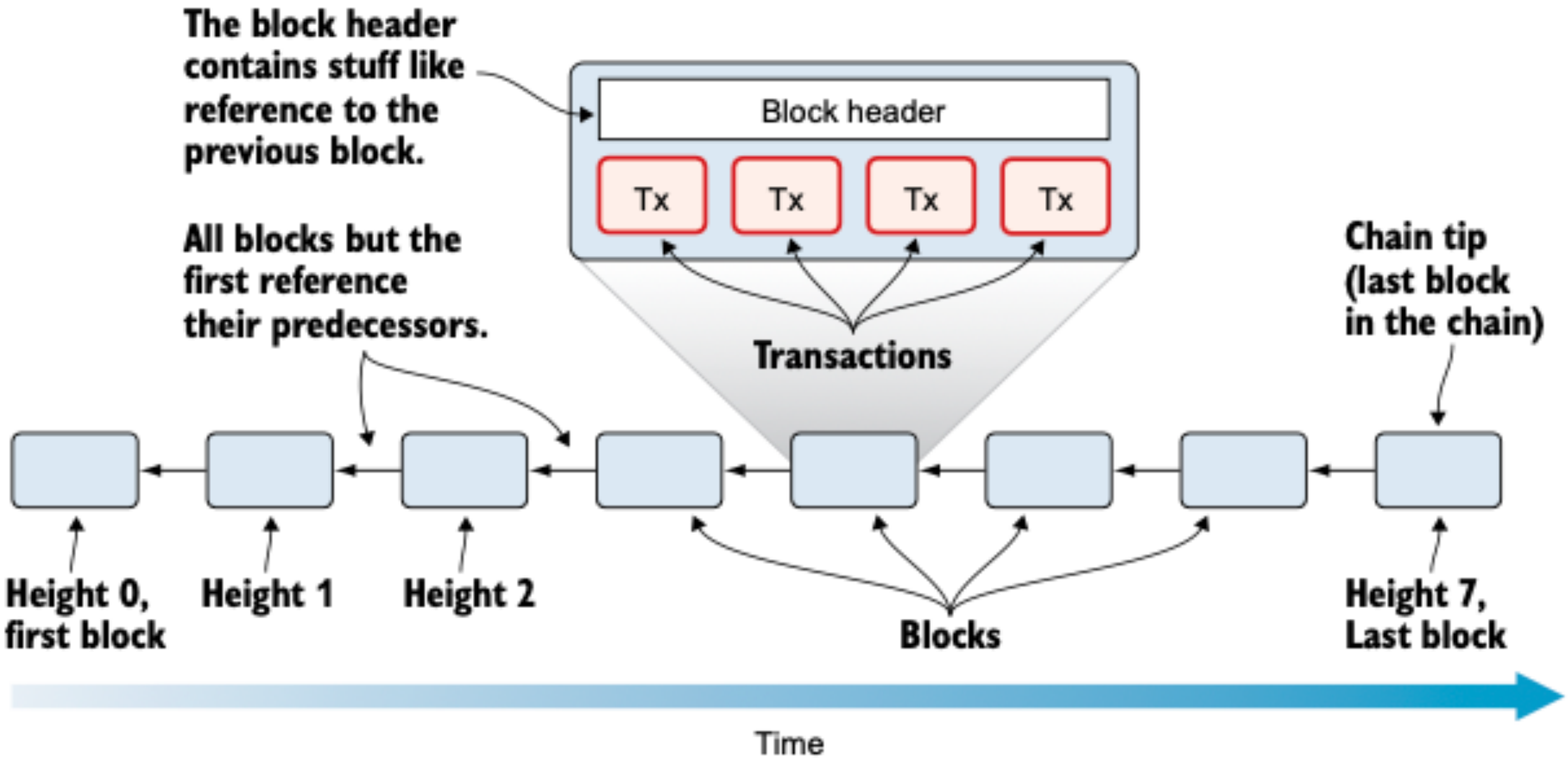
Case 2: We have a signature for <PubKey2>

ScriptSig:

```
<Signature 2>
 OP_0
<RedeemScript
```

Bitcoin Blocks

STRUCTURE OF THE BITCOIN BLOCKCHAIN



A BLOCK HEADER EXAMPLE

```
020000208ec39428b17323fa0ddec8e887b4a7c53b8c0a0
a220cfd0000000000000000005b0750fce0a889502d4050
8d39576821155e9c9e3f5c3157f961db38fd8b25be1e77a
759e93c0118a4ffd71d
```

- 02000020 - version, 4 bytes, LE
- 8ec3...00 - previous block, 32 bytes, LE
- 5b07...be - merkle root, 32 bytes, LE
- 1e77a759 - timestamp, 4 bytes, LE
- e93c0118 - bits, 4 bytes
- a4ffd71d - nonce, 4 bytes

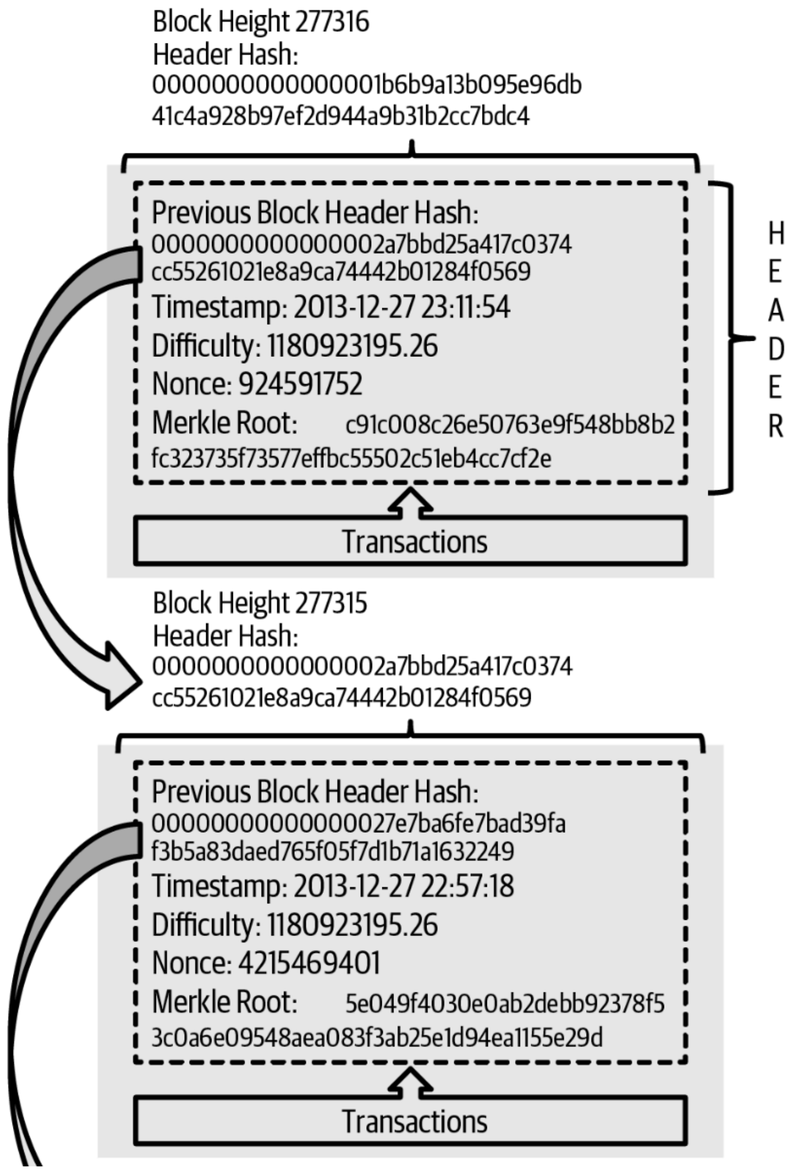
A Bitcoin block header and its breakdown.

STRUCTURE OF A BITCOIN BLOCK

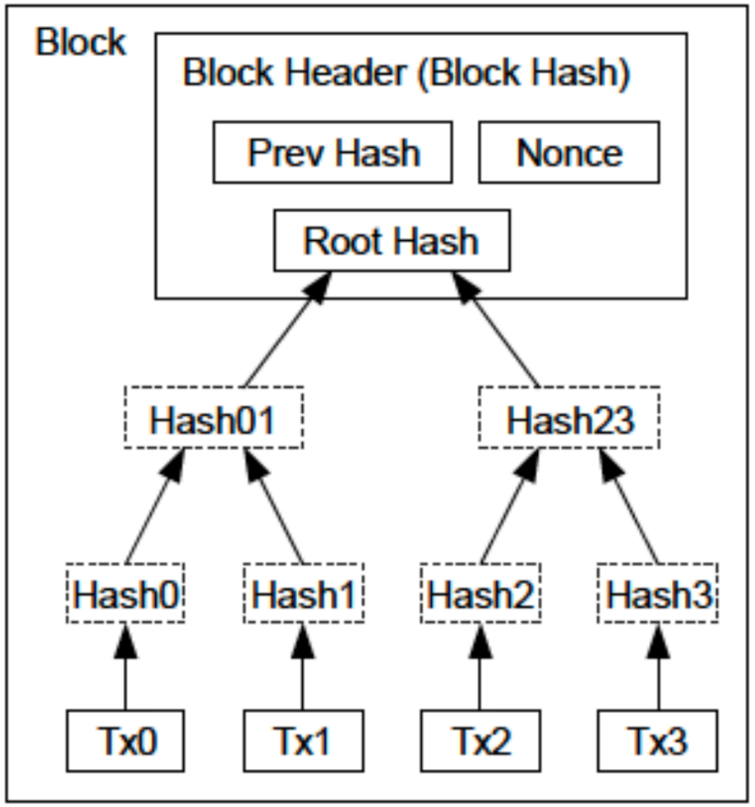
Block			
Field	**Size**	**Format**	**Description**
Version ↘	4 bytes	little-endian	The version number for the block.
Previous Block ↘	32 bytes	natural byte order	The block hash of a previous block this block is building on top of.
Merkle Root ↘	32 bytes	natural byte order	A fingerprint for all of the transactions included in the block.
Time ↘	4 bytes	little-endian	The current time as a Unix timestamp.
Bits ↘	4 bytes	little-endian	A compact representation of the current target.
Nonce ↘	4 bytes	little-endian	
Transaction Count	compact	compact size	How many upcoming transactions are included in the block.
Transactions	variable	transaction data	All of the raw transactions included in the block concatenated together.

In grey background: This part is called the **block header** of a Bitcoin block.

A BLOCK IN THE BITCOIN BLOCKCHAIN



Two Bitcoin blockchain blocks

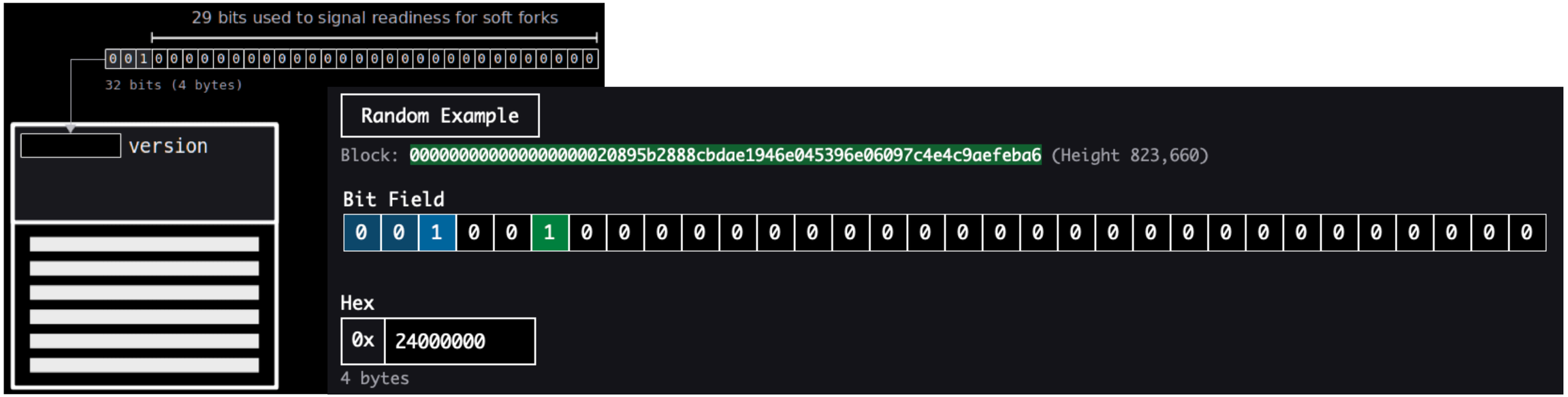


Transactions Hashed in a Merkle Tree

Schematic structure of a Bitcoin blockchain block

BITCOIN BLOCK: VERSION FIELD

- Length: 4 bytes
- Contains information about which Bitcoin protocol update (“soft fork”) the miner of the block is expressing support for.
- BIP 9: Standardizes how to miner can express support for multiple proposals, if first 3 bits of first byte are 001.
- Can also be used freely as part of “nonce” to alter the resulting block header hash.



BITCOIN BLOCK: PREVIOUS BLOCK

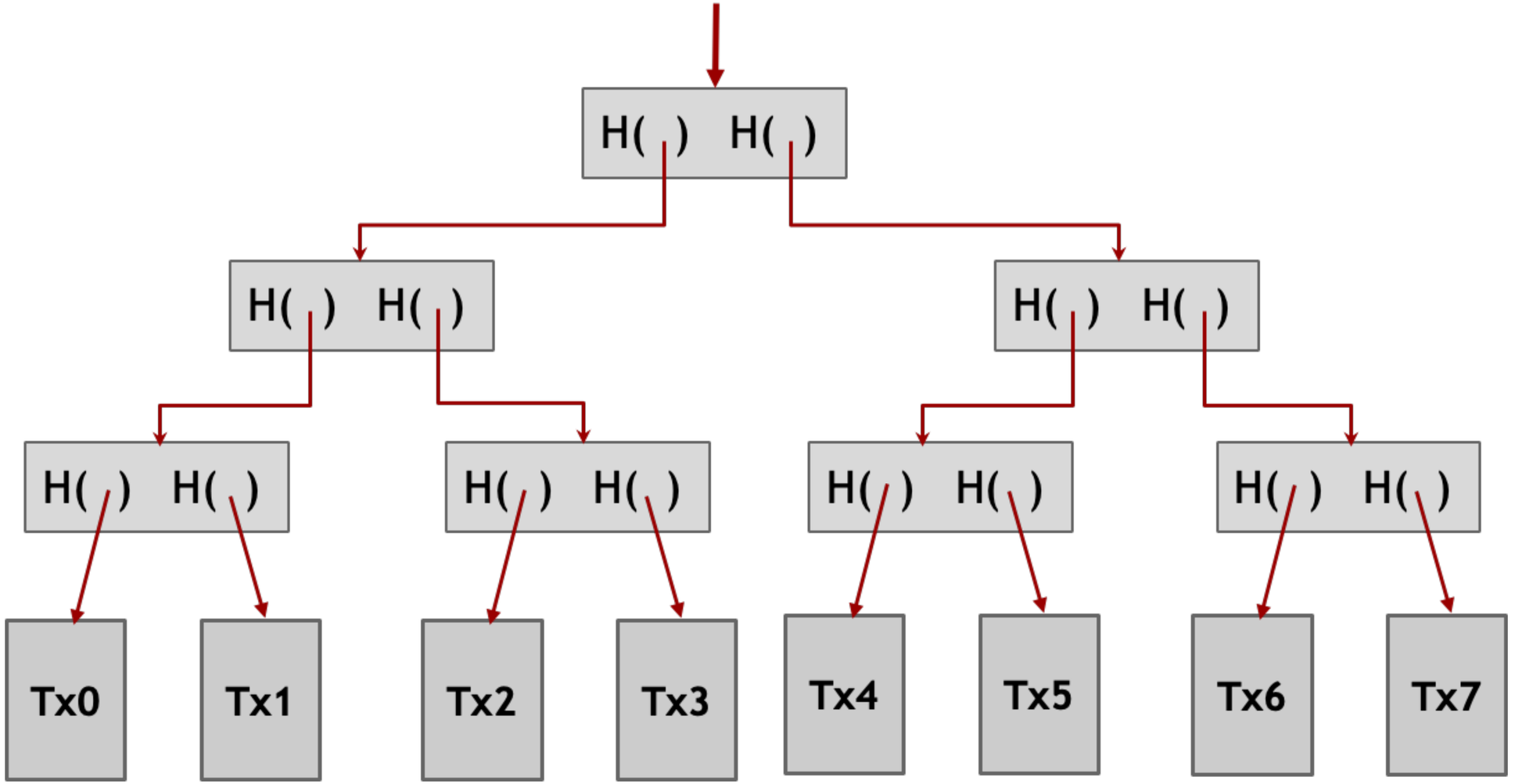
- Length: 32 bytes
- Hash of previous block in the Bitcoin blockchain.
- Contains information about the mining difficulty.

Height	Block Hash
866,107	00000000000000000001af6280b98be52c4787b7b3520672bf0afd92ca89dad4
866,106	00000000000000000002d15114d27ccf790bdcb86e61c5f04b49f6098ebd832f
866,105	00000000000000000026f69db132bb21945fa2af4aa11bb5eb0b7211eaf533f
866,104	00000000000000000000fa53ea9f98a694ce28acefb3d1d12d82e15177843b45
866,103	00000000000000000010cecf6ea4d7c05b1912b69a3da87c8120f38302aa85f

Five examples

BITCOIN BLOCK: MERKLE ROOT

- Length: 32 bytes
- Encodes all transactions of block in one 32-byte hash.
- Can be used for simplified verification of transactions.

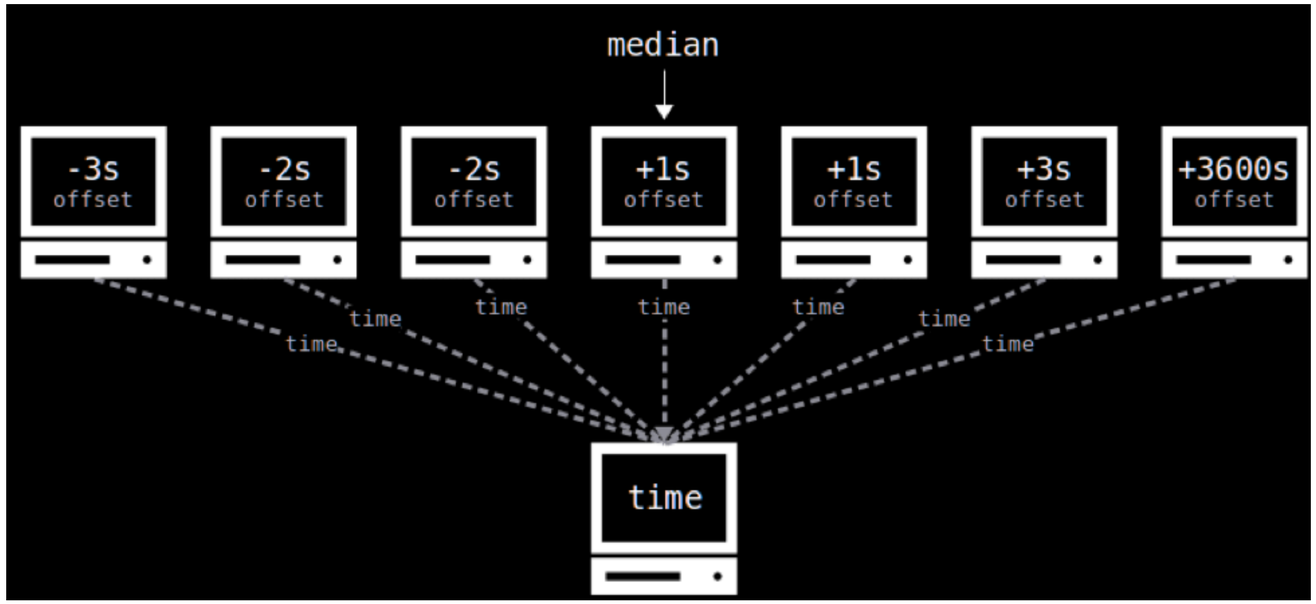


BITCOIN BLOCK: TIME(STAMP)

- Length: 4 bytes, little-endian.
- Indicates the time when the block was mined (in Unix time)
- Note: **There can be successive blocks with reverse order of timestamps!**
- **Rules:**
  - Must be greater than median time of last 11 blocks.
  - Less than “network adjusted time” + 2 hours.

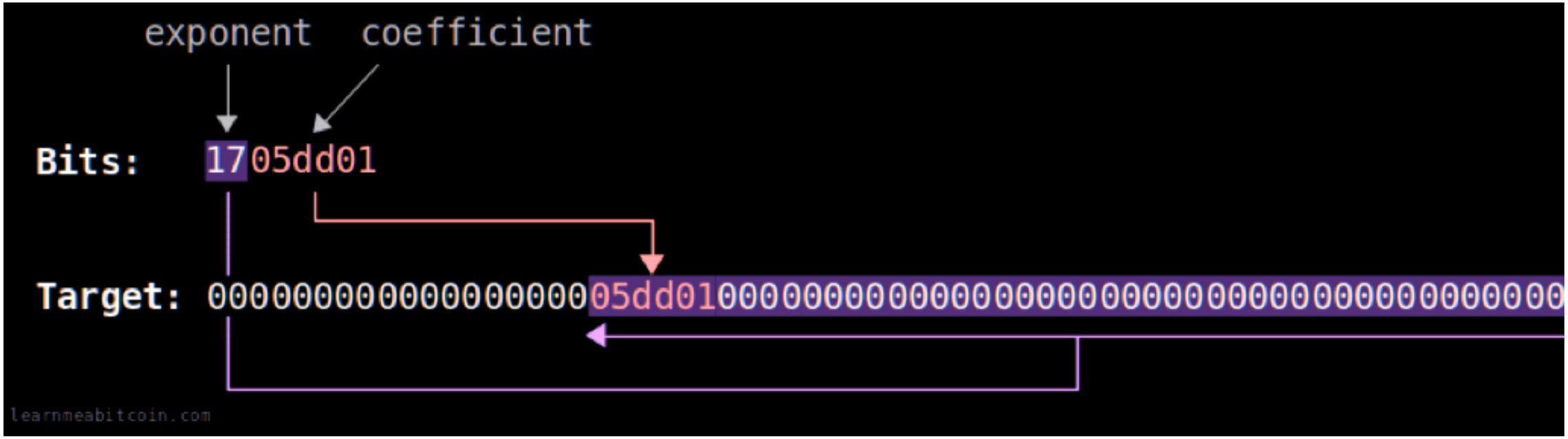
Usages:

- Compute difficulty target adjustment.
- To enforce transaction locktime.



BITCOIN BLOCK: BITS

- Length: 4 bytes
- Encodes **target for difficulty adjustment** in the mining process.
- Exponent (first byte): How many leading zeros.
- Coefficient (following 3 bytes): Specifies exact target value.



BITCOIN BLOCK: NONCE

- Length: 4 bytes, little-endian
- Is increased by miner successively to change the block hash.

BITCOIN BLOCK: TRANSACTION COUNT AND TRANSACTIONS

- We already learned what they are like!

COINBASE TRANSACTION

- Corresponds to transaction paying out the miner reward to the miner of the block.
- Currently 3.15 BTC / block (since April 2024).
- Must be first transaction in the block.
- TXID: Must be all zeros.
- Input index number (VOUT): Set to maximal value.
- Does NOT have anything to do with the company Coinbase!

```
01000000010000000000000000000000000000000000000000000000000000000000000000ffffffff0804233fa04e028b12ffffffff013049
0b2a010000004341047eda6bd04fb27cab6e7c28c99b94977f073e912f25d1ff7165d9c95cd9bbe6da7e7ad7f2acb09e0ced91705f7616af53
bee51a238b7dc527f2be0aa60469d140ac00000000

Transaction: 18a16d322b235f636ab90e62e79a9f20a0b9c14e8da51e9dc0974f99f82ee444
```

Example Coinbase Transaction

THE DIFFICULTY ADJUSTMENT

Part of the Bitcoin consensus rules.

Every 2016 blocks, mining difficulty is adjusted by updating value for “target” in the subsequent 2016 blocks based on:

$$\text{new target} = \text{old target} \cdot \frac{(\text{time of current block}) - (\text{time of (current} - 2015\text{th) block})}{20160 \text{ minutes}}$$

VALIDATION OF BLOCKS

For a block to be valid, the following rules need to be satisfied:

- Syntax of the block data structure needs to be correct (see also here).
- Block header hash is less than the target.
- Block time stamp is above the **Median Time Past** (See BIP113) (median time last 11 blocks in the chain).
- Block time stamp is below **Network Adjusted Time** plus two hours.
- Block size is below 1,000,000 vbytes.
- (Only) first transaction in transaction Merkle tree is the coinbase transaction.
- All transactions in block are valid.