Blockchain Principles and Applications

Amir Mahdi Sadeghzadeh, PhD

Data and Network Security Lab (DNSL)
Trustworthy and Secure AI Lab (TSAIL)

Most slides have been adapted from ECE/COS 470, Pramod Viswanath, Princeton 2024 and CS251, Dan Boneh, Stanford 2023.

Recap

Bitcoin blockchain: a sequence of block headers, 80 bytes each

H

BH_1

version	(4 bytes)
prev	(32 bytes)
time	(4 bytes)
bits	(4 bytes)
nonce	(4 bytes)
Tx root	(32 bytes)

80 bytes

H

BH_2

prev

Tx root

H

BH_3

prev

Tx root

...

coinbase Tx

coinbase Tx

Bitcoin blockchain: a sequence of block headers, 80 bytes each

time: time miner assembled the block. Self reported.
(block rejected if too far in past or future)

bits: proof of work difficulty
nonce: proof of work solution
} for choosing a proposer

Merkle tree: payer can give a short proof that Tx is in the block

new block every ≈10 minutes.

An example

Tx data

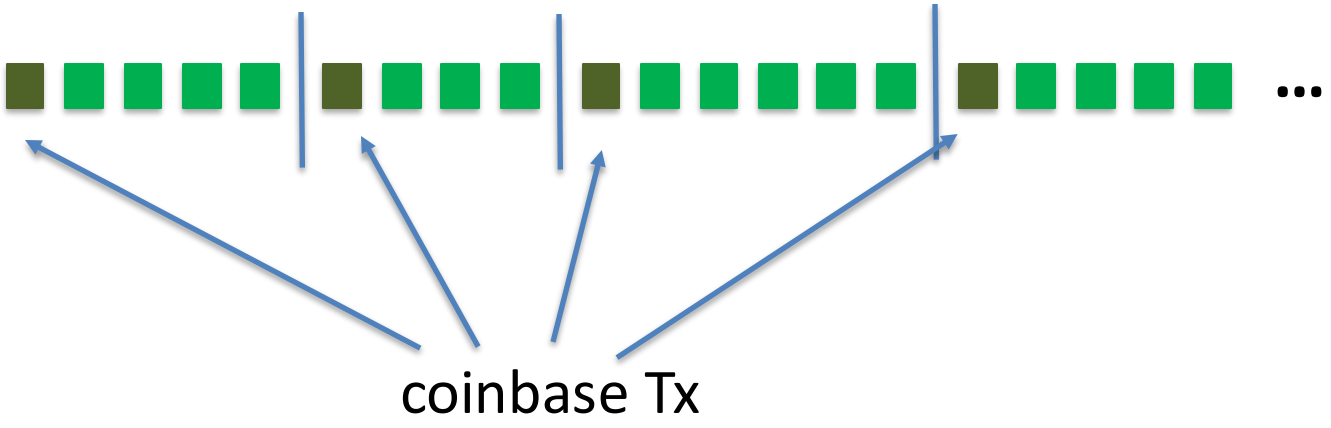
Height	Mined	Miner	Size	#Tx
648494	17 minutes	Unknown	1,308,663 bytes	1855
648493	20 minutes	SlushPool	1,317,436 bytes	2826
648492	59 minutes	Unknown	1,186,609 bytes	1128
648491	1 hour	Unknown	1,310,554 bytes	2774
648490	1 hour	Unknown	1,145,491 bytes	2075
648489	1 hour	Poolin	1,359,224 bytes	2622

Block 648493

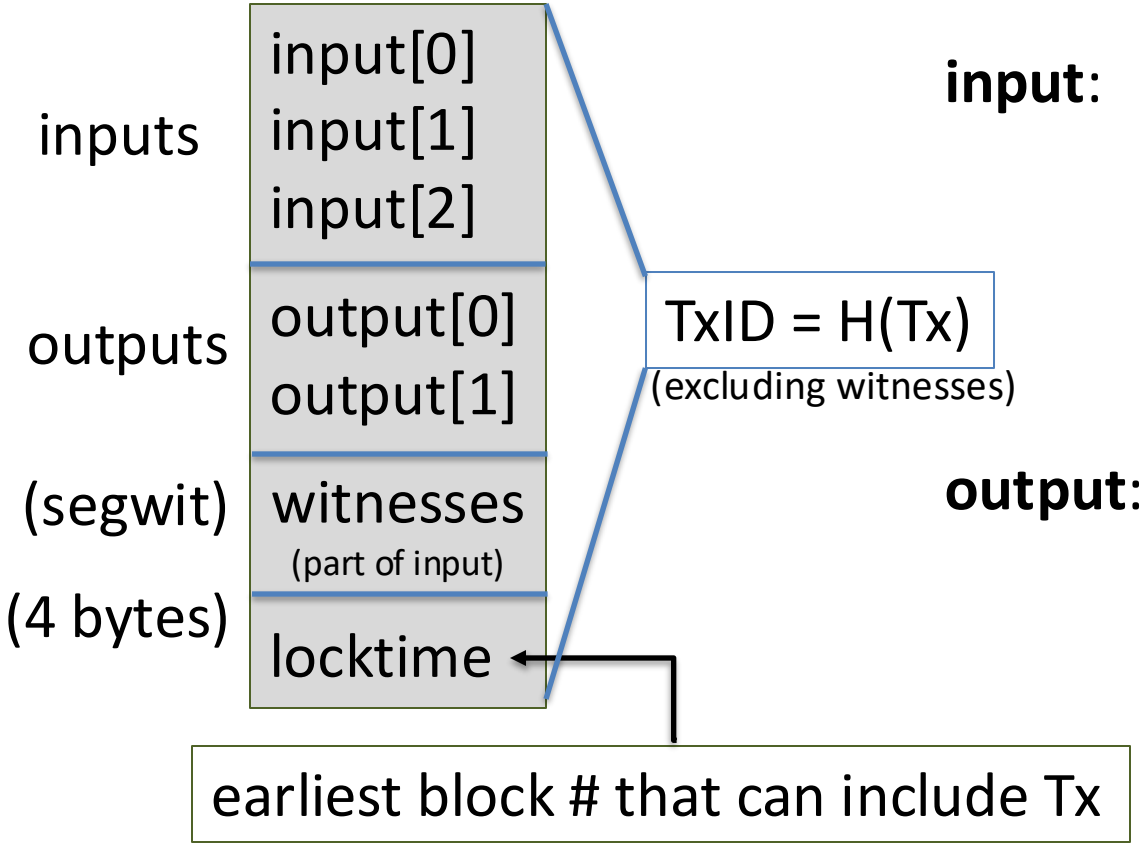
Timestamp	2020-09-15 17:25	
Height	648493	
Miner	SlushPool	(from coinbase Tx)
Number of Transactions	2,826	
Difficulty (D)	17,345,997,805,929.09	(adjusts every two weeks)
Merkle root	350cbb917c918774c93e945b960a2b3ac1c8d448c2e67839223bbcf595baff89	
Transaction Volume	11256.14250596 BTC	
Block Reward	6.25000000 BTC	
Fee Reward	0.89047154 BTC	(Tx fees given to miner in coinbase Tx)

This lecture

View the blockchain as a sequence of Tx (append-only)



Tx structure (non-coinbase)



input:

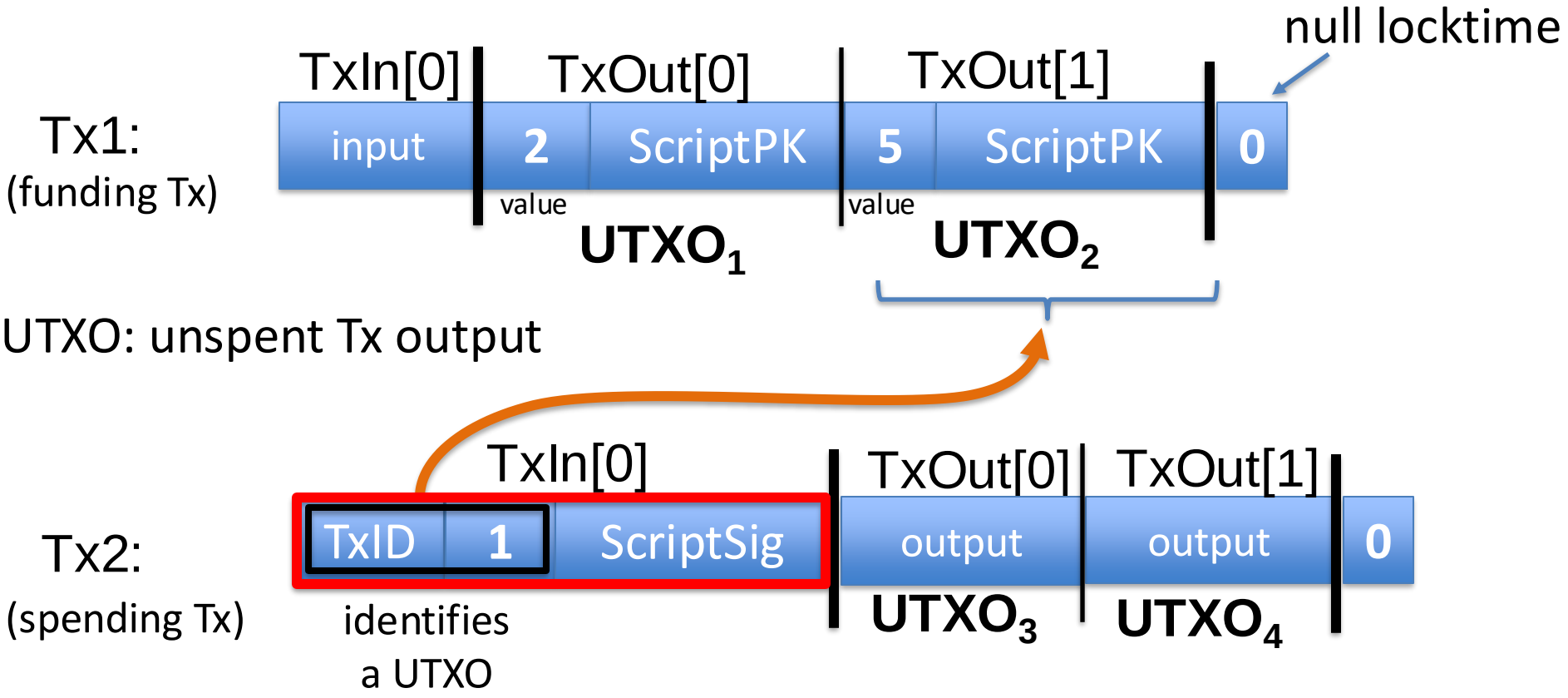
TxID	32 byte hash
out-index	4 byte index
ScriptSig	program
seq	ignore

output:

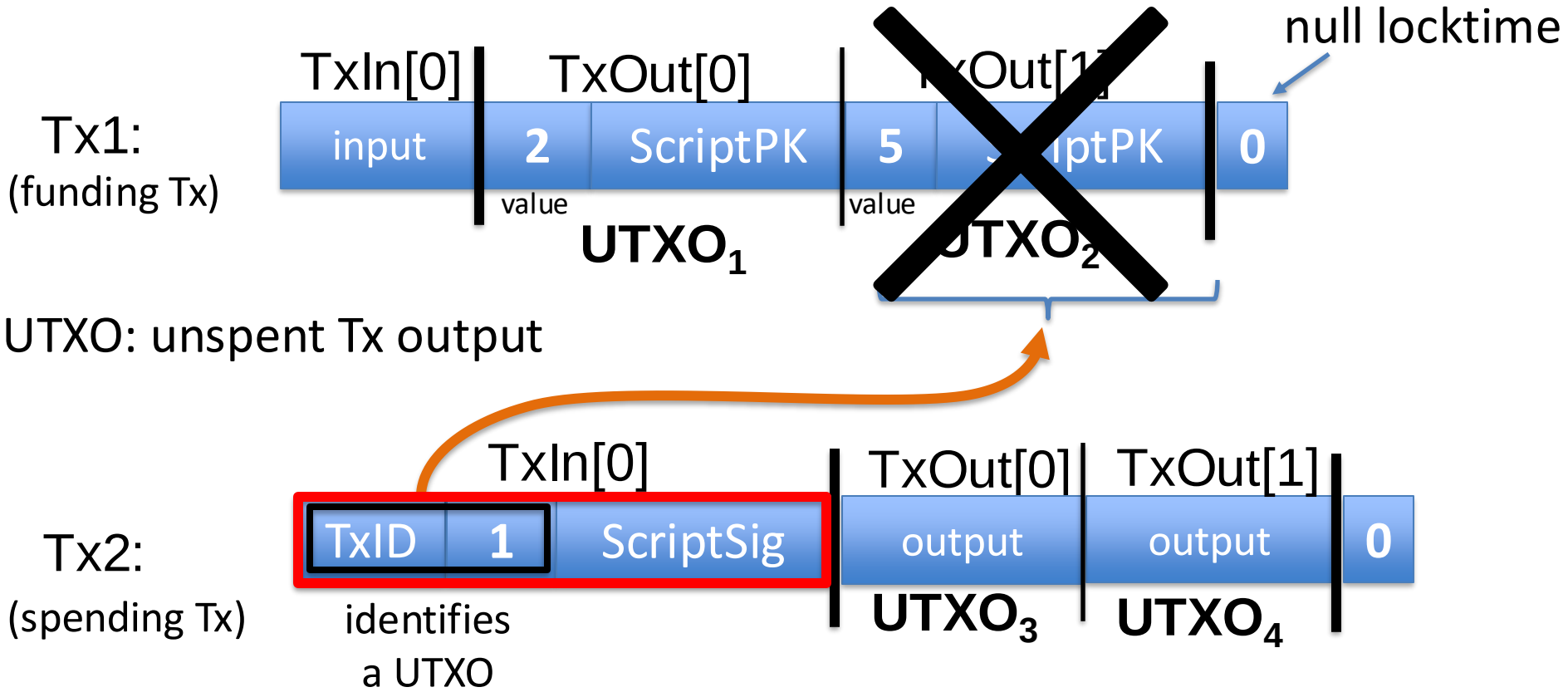
value	8 bytes
ScriptPK	program

#BTC = value/10^8

Example


Tx1:
(funding Tx)
TxIn[0]
TxOut[0]
TxOut[1]
null locktime
input
2
ScriptPK
5
ScriptPK
0
value
value
UTXO1
UTXO2
UTXO: unspent Tx output
Tx2:
(spending Tx)
TxIn[0]
TxID
1
ScriptSig
identifies
a UTXO
TxOut[0]
TxOut[1]
output
output
0
UTXO3
UTXO4

Example


Tx1:
(funding Tx)
TxIn[0]
TxOut[0]
TxOut[1]
null locktime
input
2
ScriptPK
5
ScriptPK
0
value
value
UTXO1
UTXO2
UTXO: unspent Tx output
Tx2:
(spending Tx)
TxIn[0]
TxID
1
ScriptSig
identifies
a UTXO
TxOut[0]
TxOut[1]
output
output
0
UTXO3
UTXO4

Validating Tx2

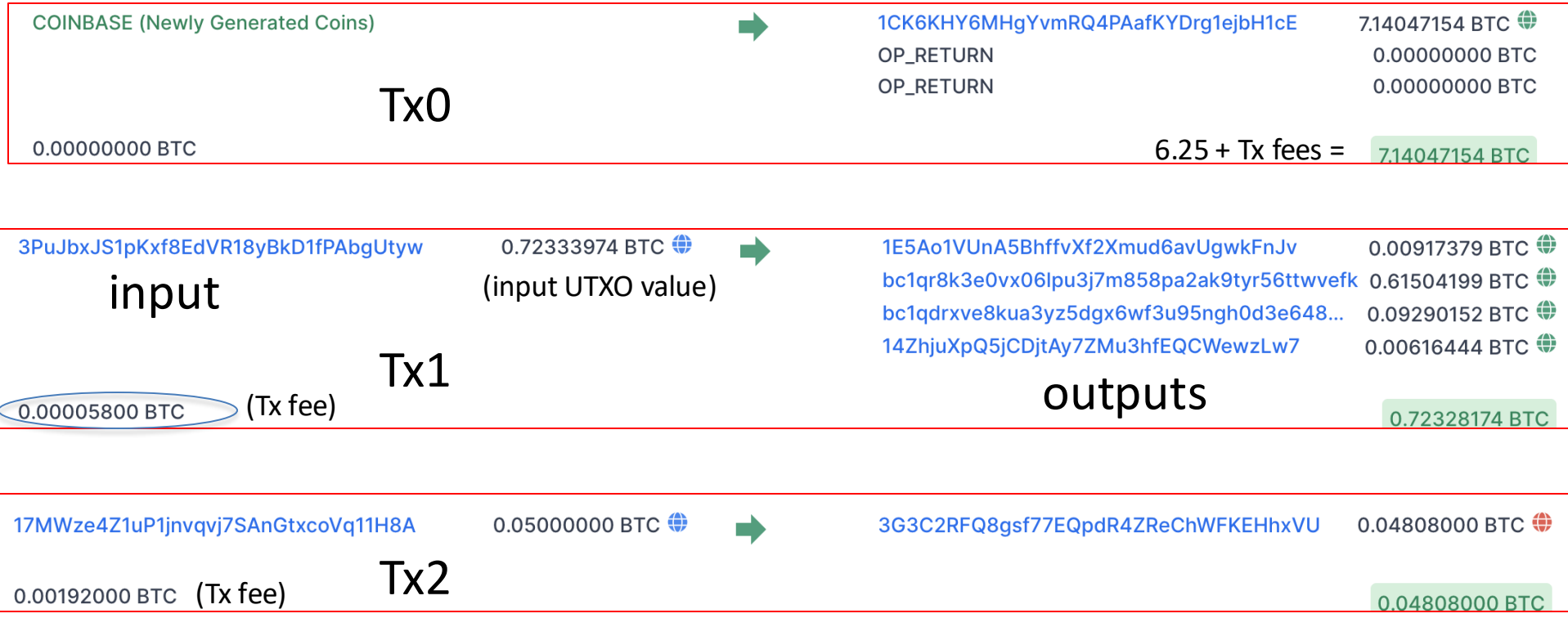
Miners check (for each input):

program from funding Tx: under what conditions can UTXO be spent

1. The program ScriptSig | ScriptPK returns true
2. TxID | index is in the current UTXO set
3. sum input values ≥ sum output values

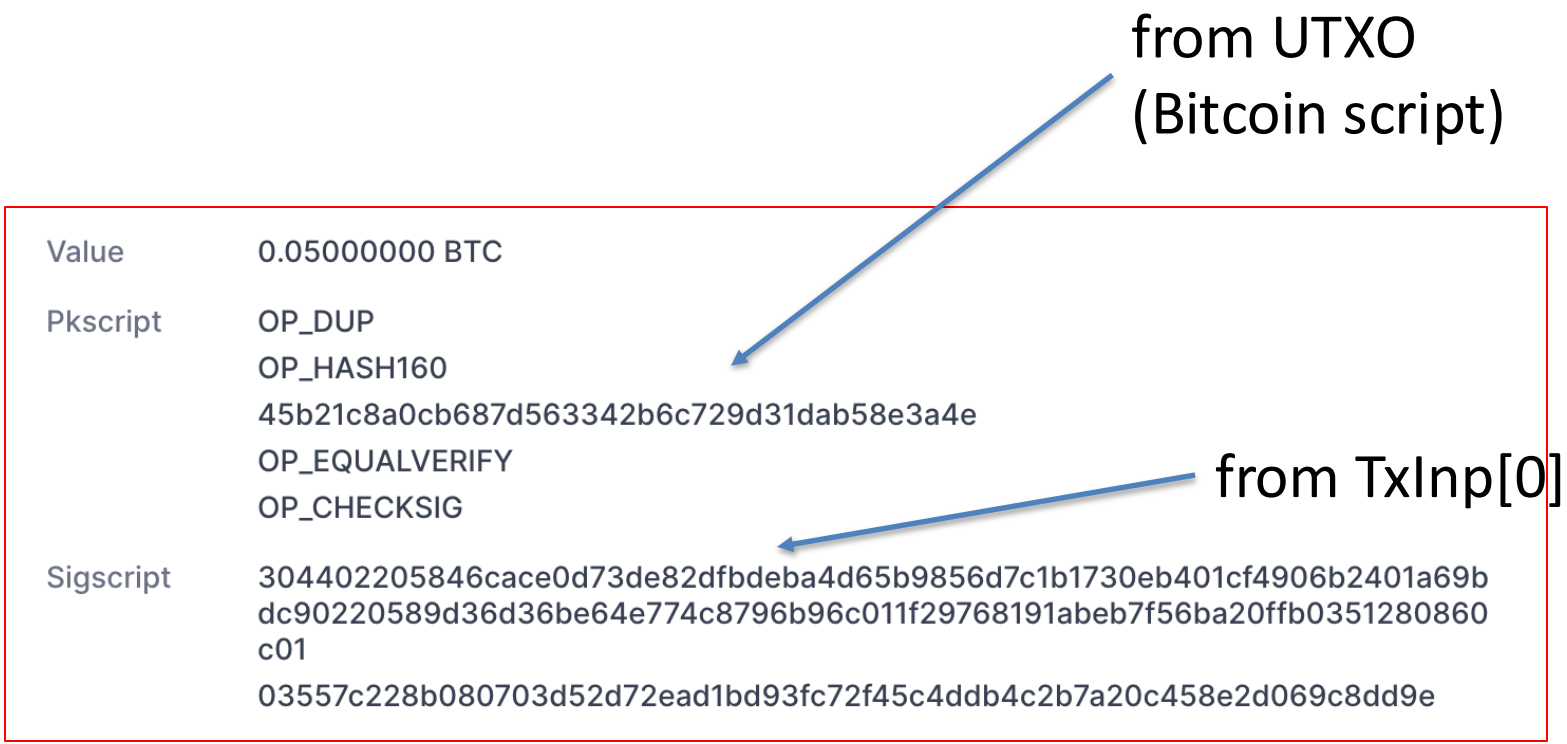
After Tx2 is posted, miners remove $UTXO_2$ from UTXO set

An example (block 648493) [2826 Tx]



sum of fees in block added to coinbase Tx

Focusing on Tx2: TxInp[0]



Transaction Script

What is Script?

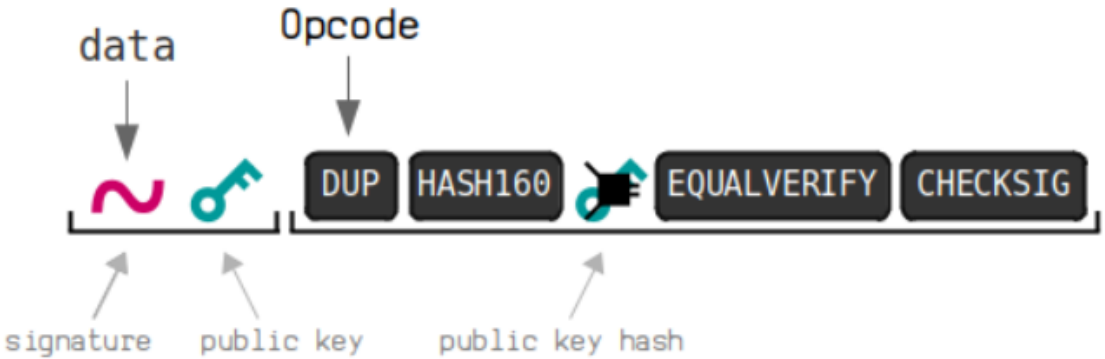
- Script is a mini programming language used as a locking mechanism for outputs in bitcoin transactions.
  - A locking script (ScriptPubKey) is placed on every transaction output.
  - An unlocking script (ScriptSig or Witness) must be provided to unlock an output (i.e. when used as an input to a transaction).
- If a full script (unlocking + locking) is valid, the output is "unlocked" and can be spent.

Script Language

Script is a very basic programming language. It consists of two things:

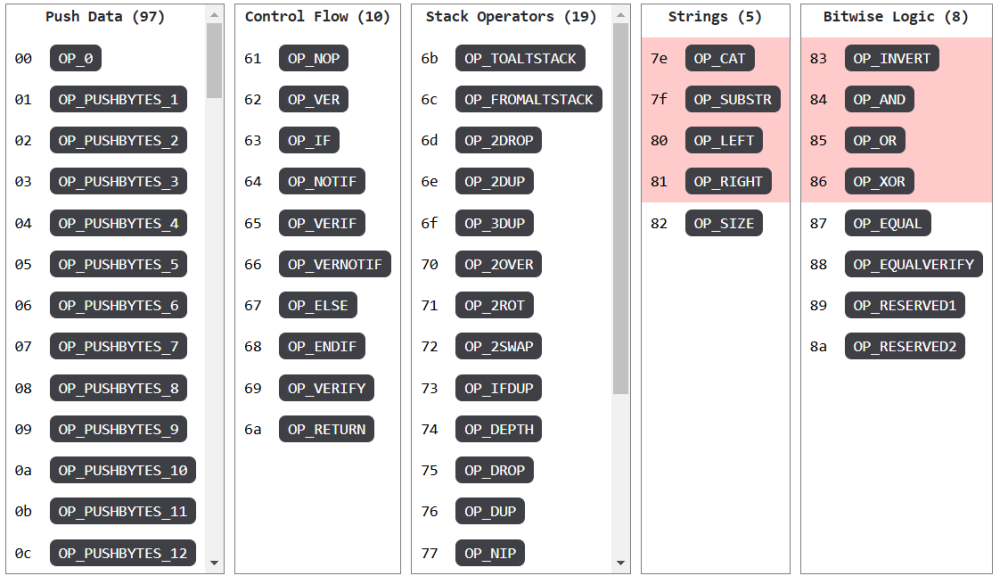
1. **Opcodes** – Simple functions that *operate* on data.
2. **Data** – Such as public keys and signatures.

Here's a simple diagram of a typical P2PKH script used in Bitcoin:



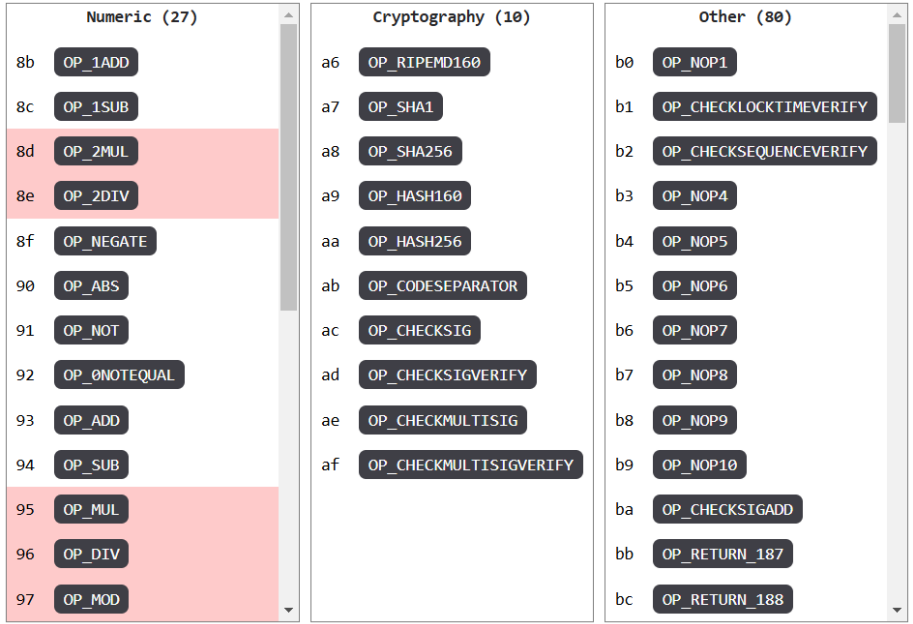
Opcodes

- Here's a quick list of all the opcodes available in the Script language (along with the corresponding hexadecimal byte used to represent each one).



Opcodes

- Here's a quick list of all the opcodes available in the Script language (along with the corresponding hexadecimal byte used to represent each one).



Numeric (27)

8b	OP_1ADD
8c	OP_1SUB
8d	OP_2MUL
8e	OP_2DIV
8f	OP_NEGATE
90	OP_ABS
91	OP_NOT
92	OP_0NOTEQUAL
93	OP_ADD
94	OP_SUB
95	OP_MUL
96	OP_DIV
97	OP_MOD

Cryptography (10)

a6	OP_RIPEMD160
a7	OP_SHA1
a8	OP_SHA256
a9	OP_HASH160
aa	OP_HASH256
ab	OP_CODESEPARATOR
ac	OP_CHECKSIG
ad	OP_CHECKSIGVERIFY
ae	OP_CHECKMULTISIG
af	OP_CHECKMULTISIGVERIFY

Other (80)

b0	OP_NOP1
b1	OP_CHECKLOCKTIMEVERIFY
b2	OP_CHECKSEQUENCEVERIFY
b3	OP_NOP4
b4	OP_NOP5
b5	OP_NOP6
b6	OP_NOP7
b7	OP_NOP8
b8	OP_NOP9
b9	OP_NOP10
ba	OP_CHECKSIGADD
bb	OP_RETURN_187
bc	OP_RETURN_188

Data

The data elements inside a Script (e.g. public keys, signatures) have to be **manually pushed on to the stack** using an opcode.

There are a few specific opcodes you can choose from to do this, and the one you use depends on *how many bytes* you want to push on to the stack.

`OP_0` to `OP_16`: (single byte representing numbers 0 to 16) [show]

`OP_PUSHBYTES_X`: 1 to 75 bytes [show]

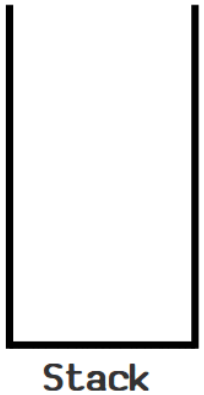
`OP_PUSHDATA1`: 76 to 255 bytes [show]

`OP_PUSHDATA2`: 256 to 65535 bytes [show]

`OP_PUSHDATA4`: 65536 to 4294967295 bytes [show]

Execution

- A complete script is run from left-to-right. As it runs, it makes use of a data structure called a stack.
- Data is pushed on to the stack.



Bitcoin Script

A stack machine. Not Turing Complete: no loops.

Quick survey of op codes:

1. **OP_TRUE** (OP_1), **OP_2**, …, **OP_16**: push value onto stack

   81 82 96

2. **OP_DUP**: push top of stack onto stack

   118

Bitcoin Script

3. control:

99 **OP_IF** <statements> **OP_ELSE** <statements> **OP_ENDIF**

105 **OP_VERIFY**: abort fail if top = false

106 **OP_RETURN**: abort and fail

what is this for? ScriptPK = [OP_RETURN, <data>]

136 **OP_EQVERIFY**: pop, pop, abort fail if not equal

Bitcoin Script

4. arithmetic:

   OP_ADD, **OP_SUB**, **OP_AND**, …: pop two items, add, push

5. crypto:

   OP_SHA256: pop, hash, push

   OP_CHECKSIG: pop pk, pop sig, verify sig. on Tx, push 0 or 1

6. Time: **OP_CheckLockTimeVerify** (CLTV):

   fail if value at the top of stack > Tx locktime value.

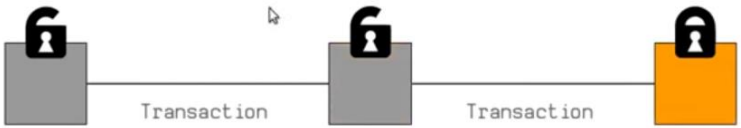
   usage: UTXO can specify min-time when it can be spent

Script

Script

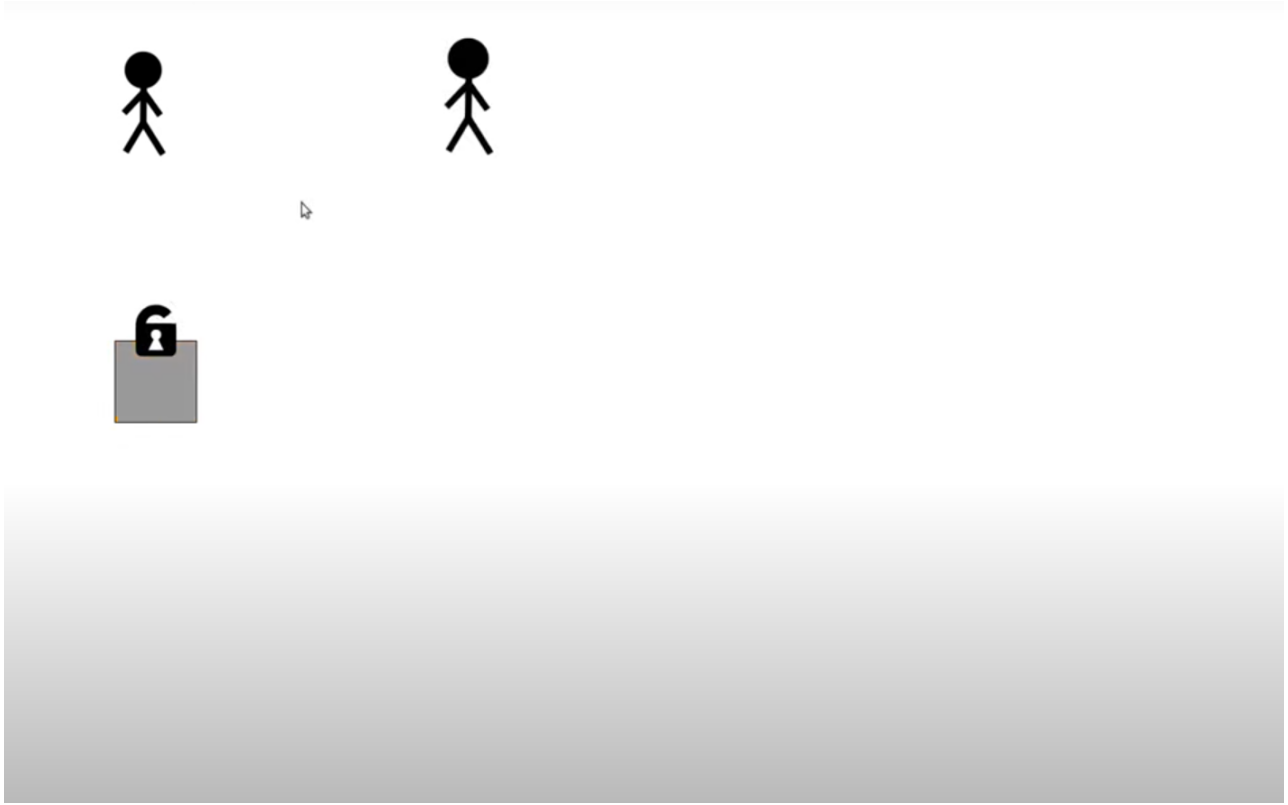
Script

Script

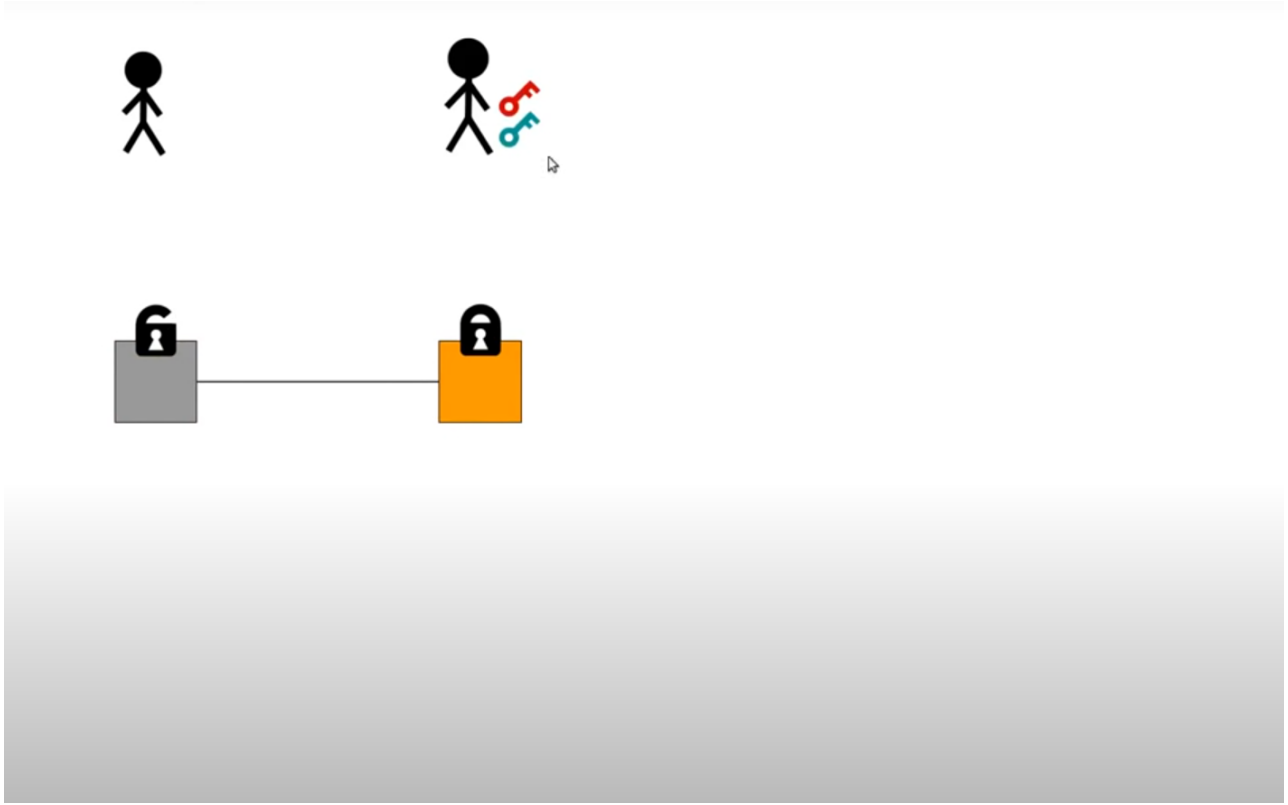


Script

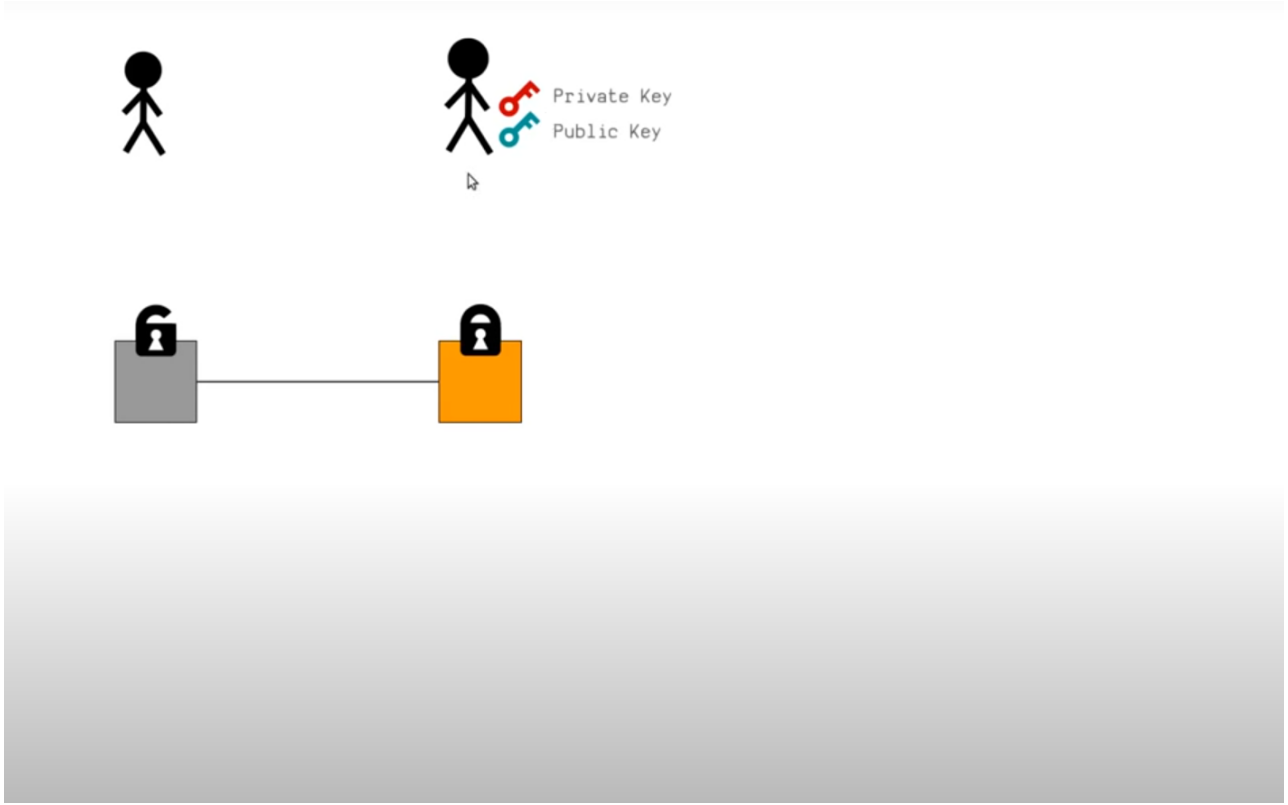
Script



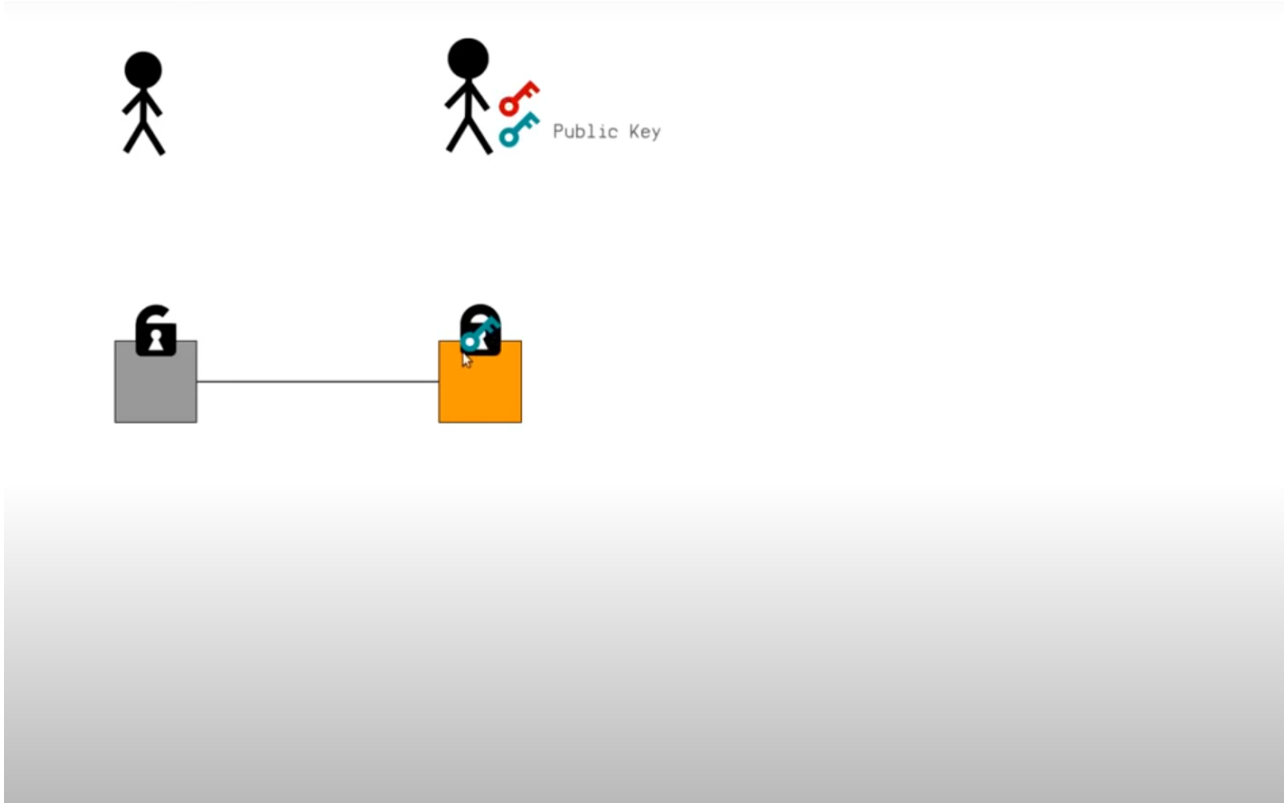
Script



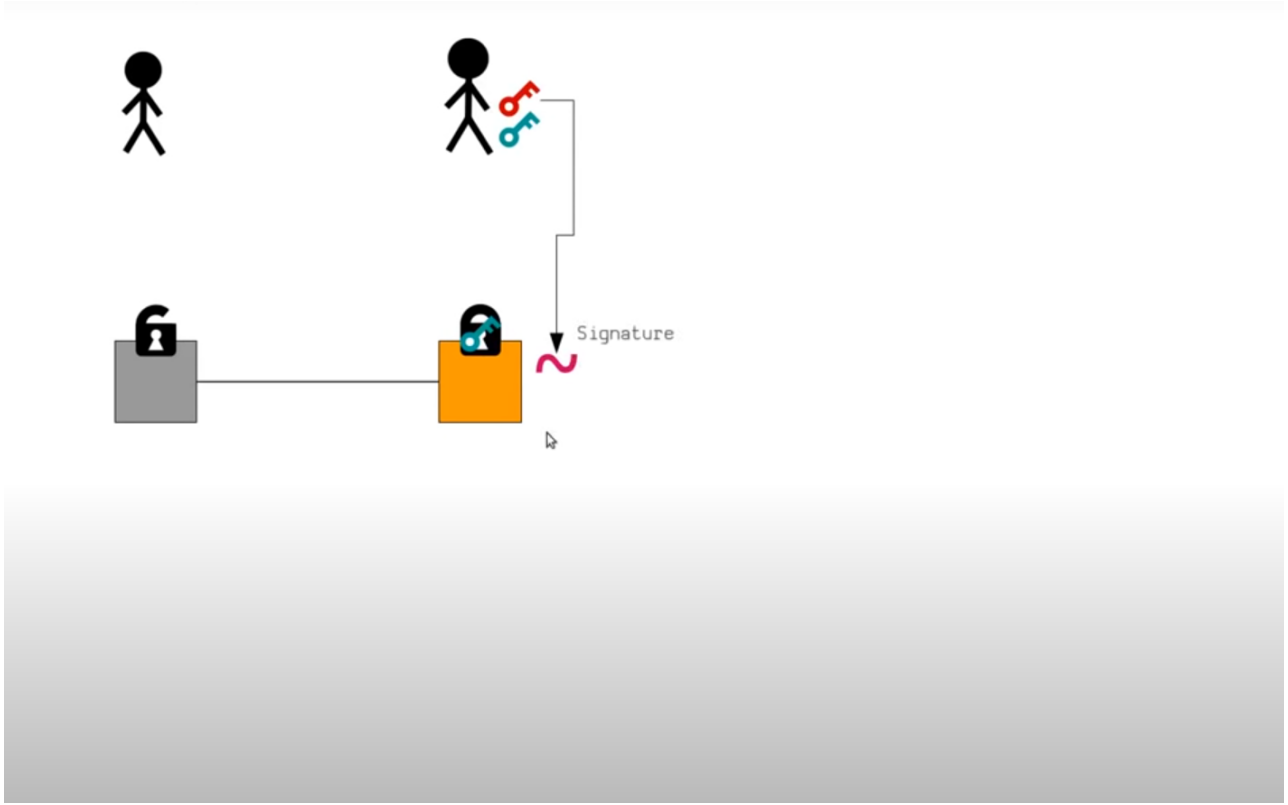
Script



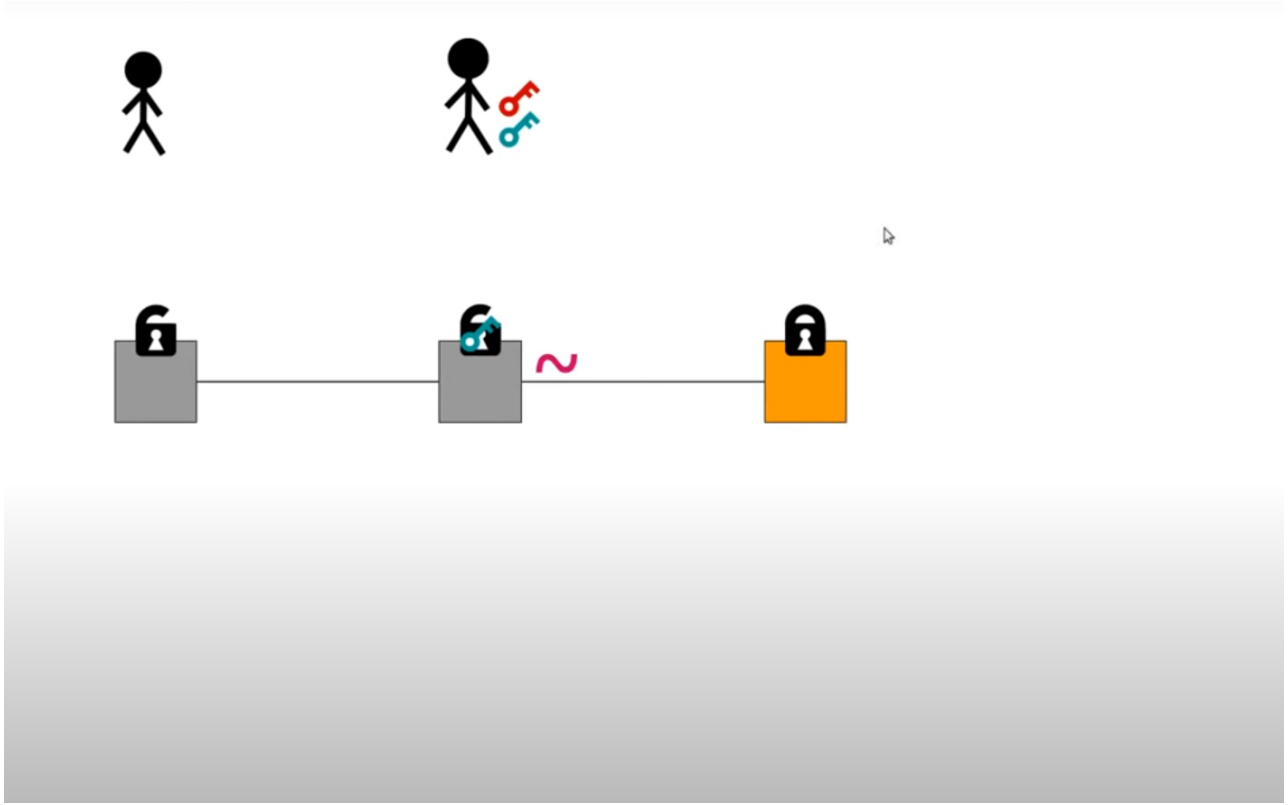
Script



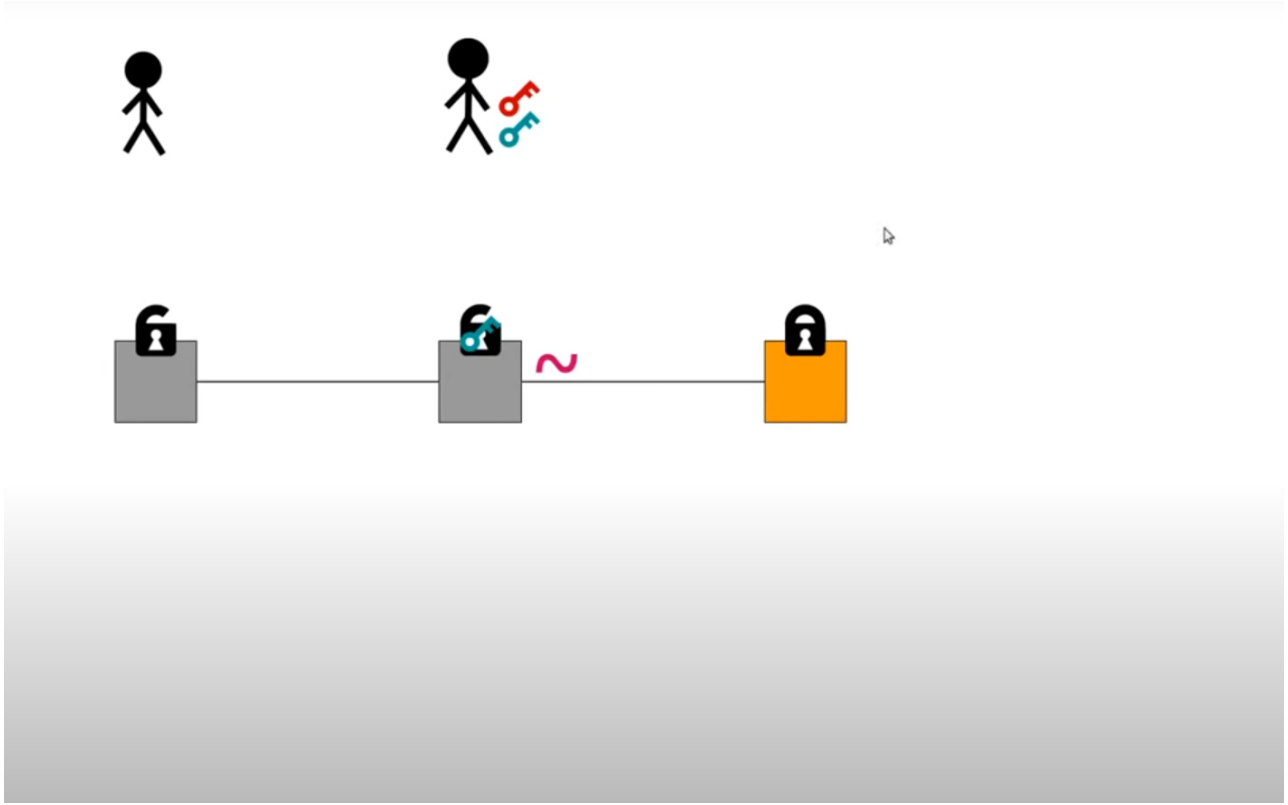
Script



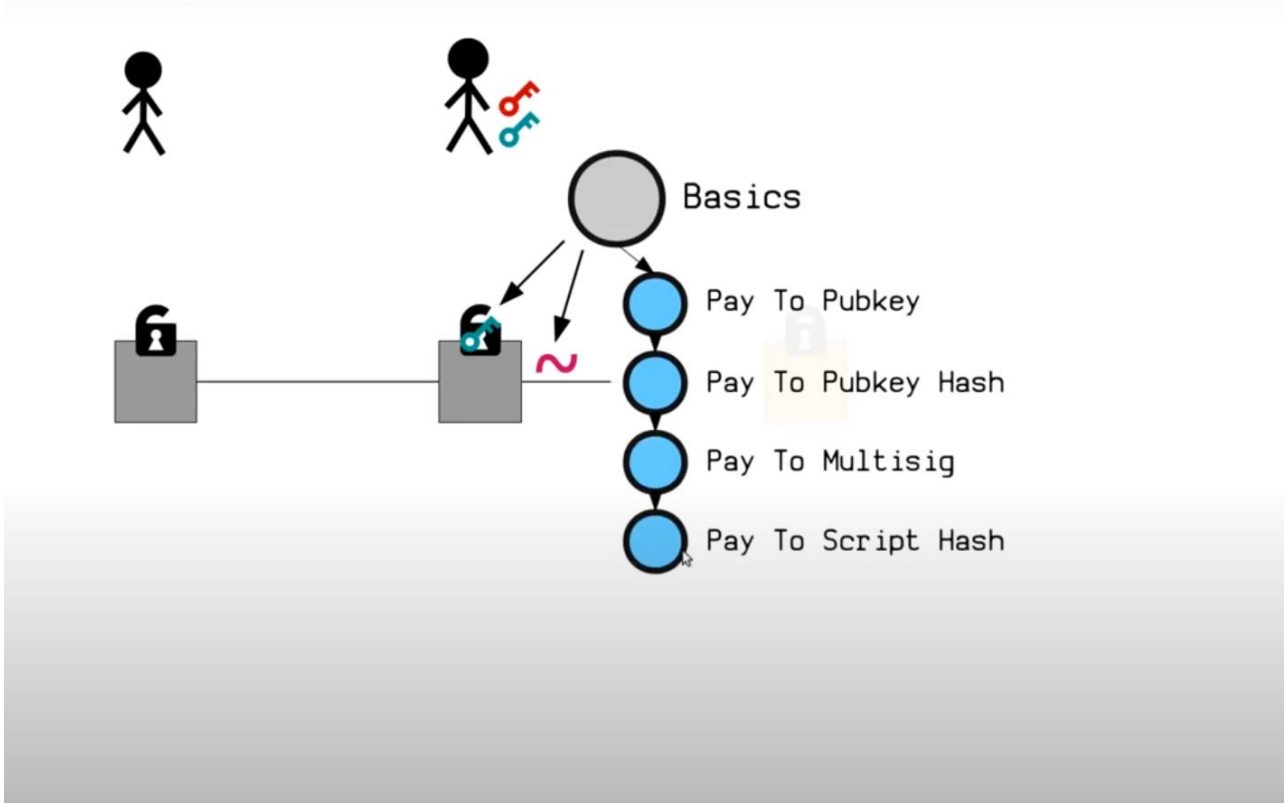
Script



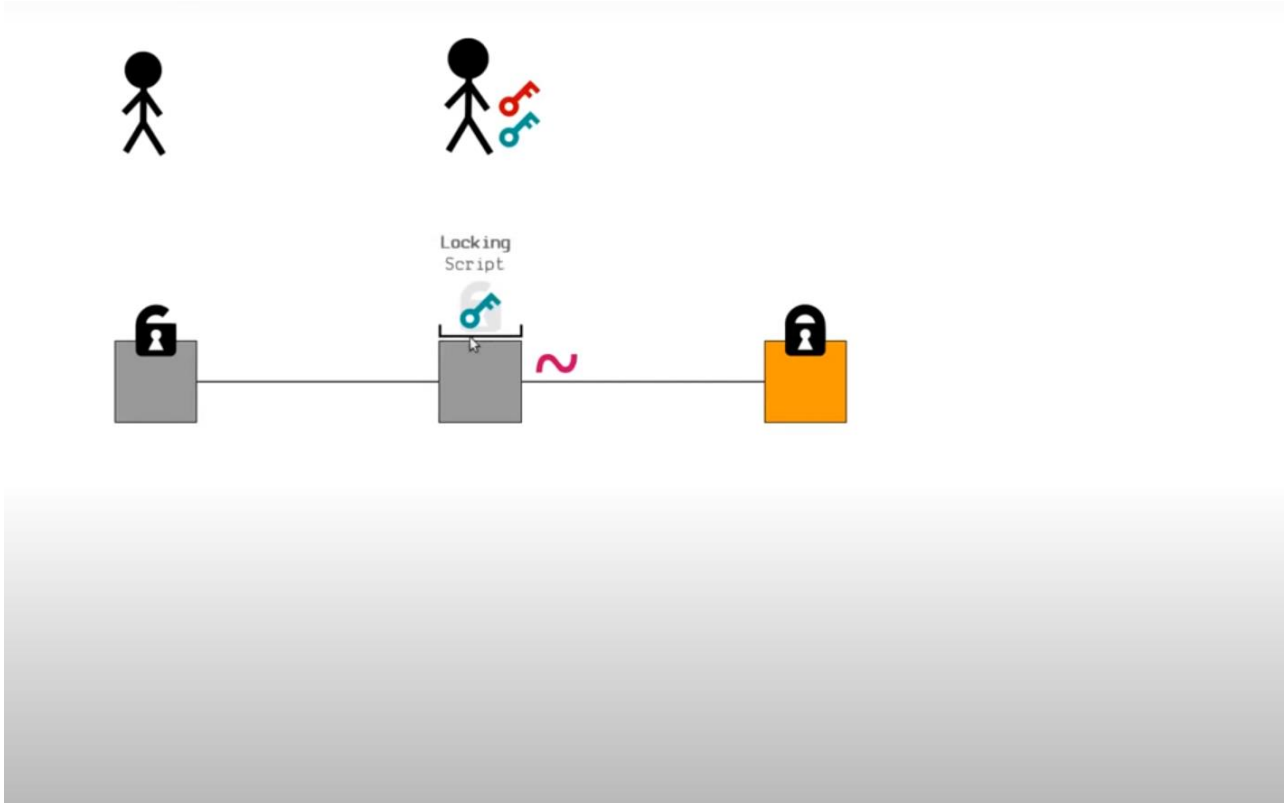
Script



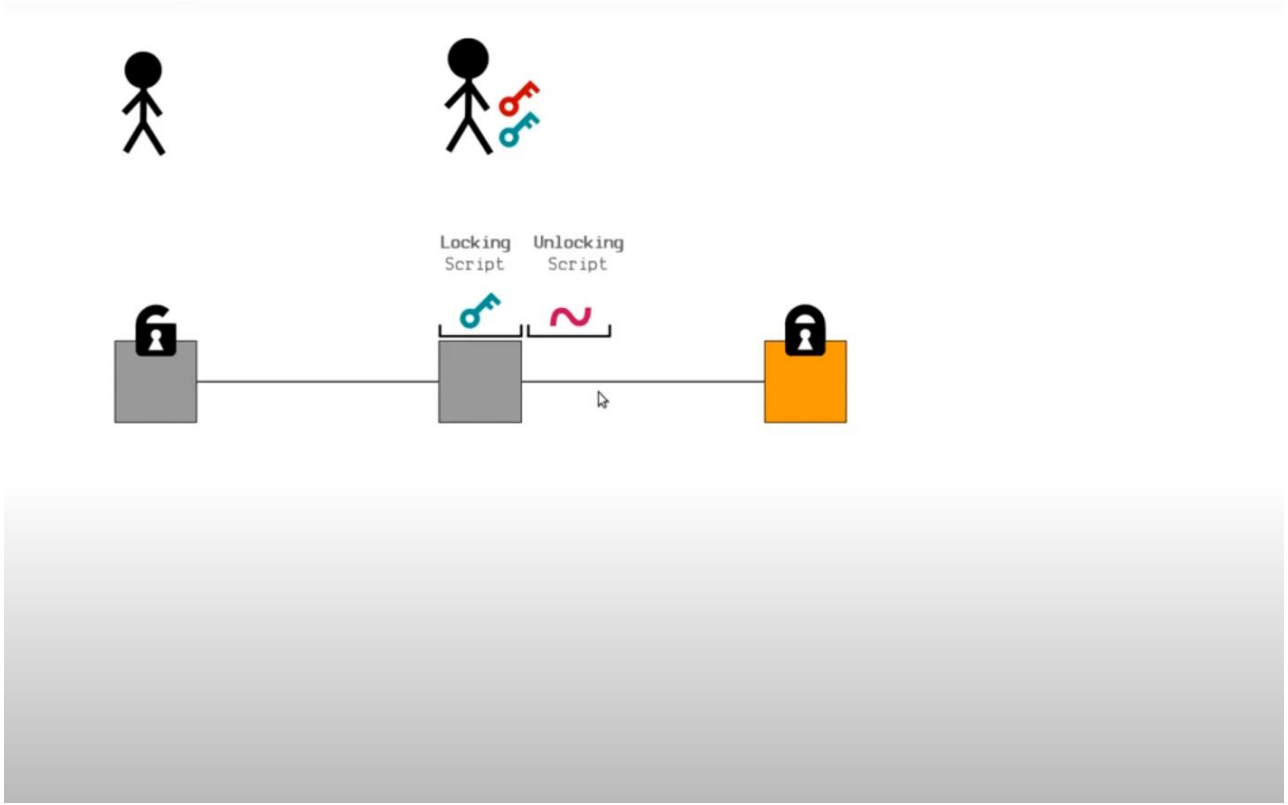
Script


Basics
Pay To Pubkey
Pay To Pubkey Hash
Pay To Multisig
Pay To Script Hash

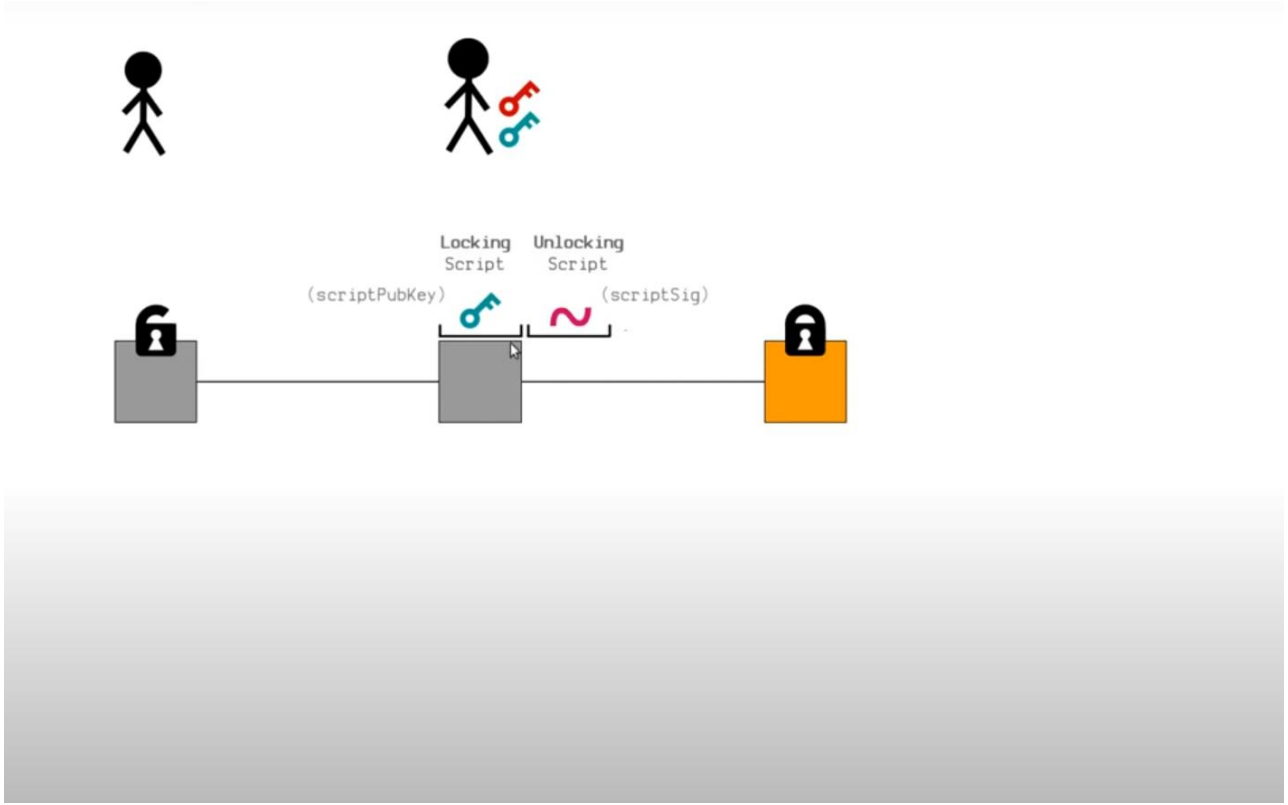
Script



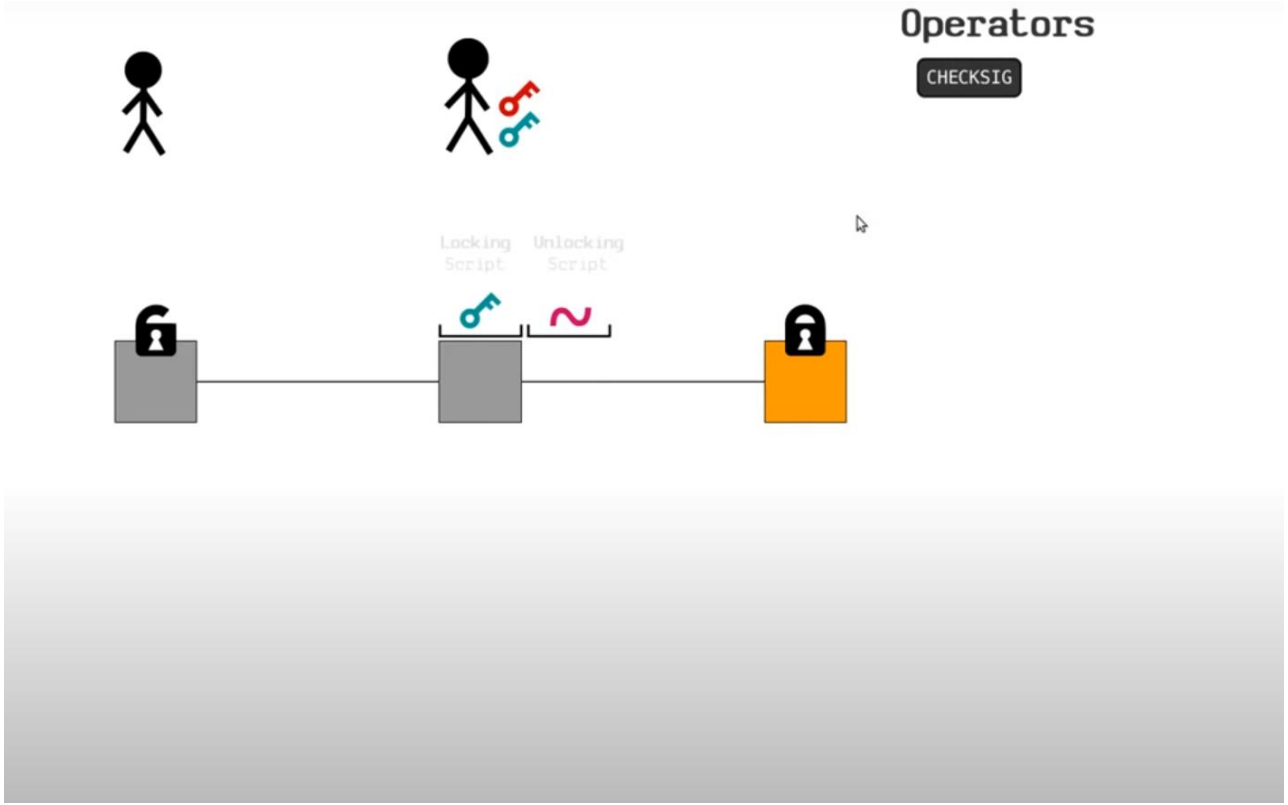
Script



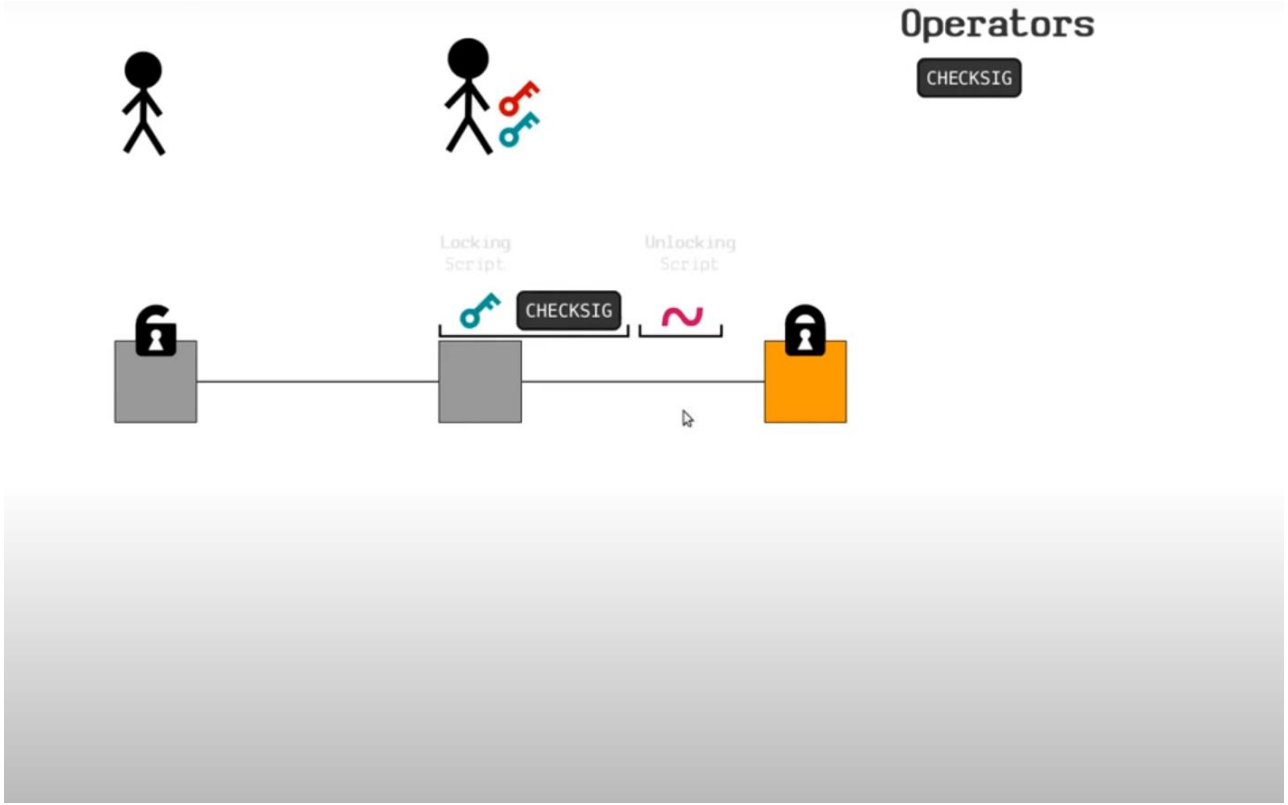
Script



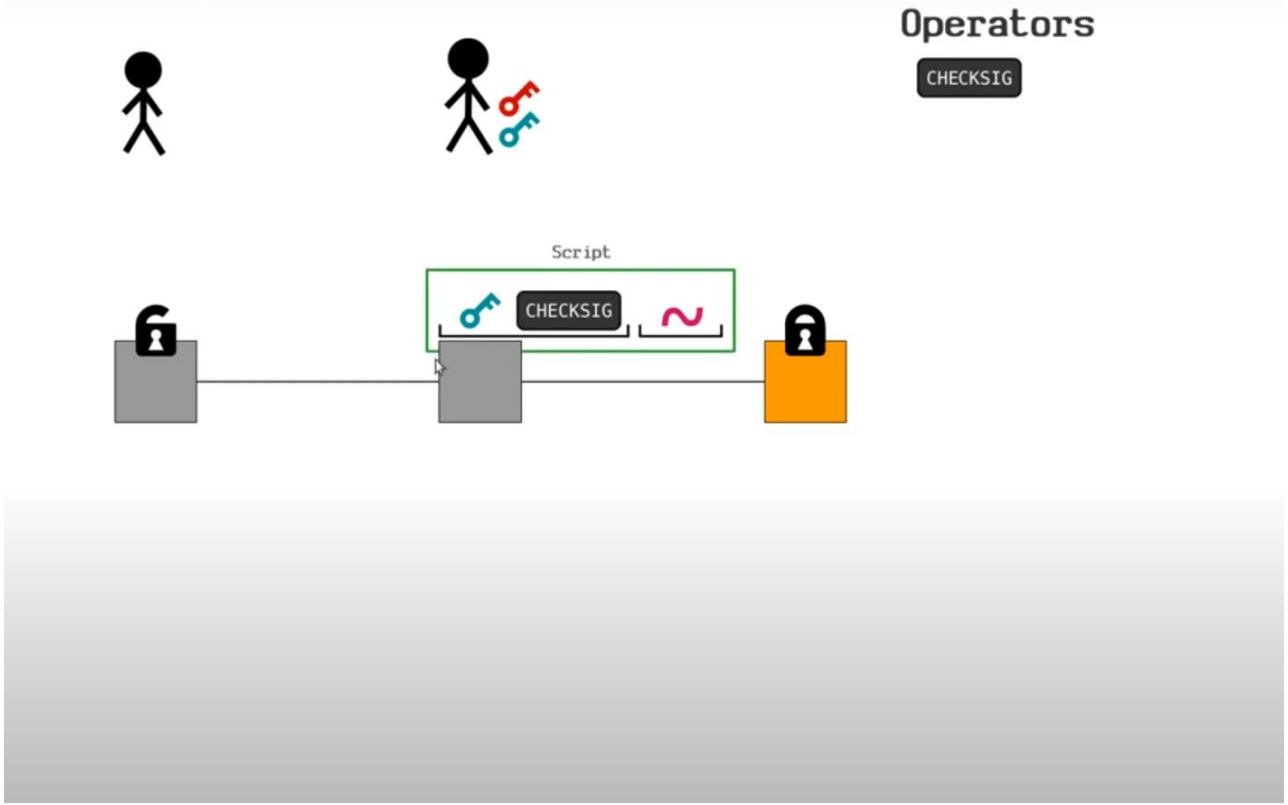
Script



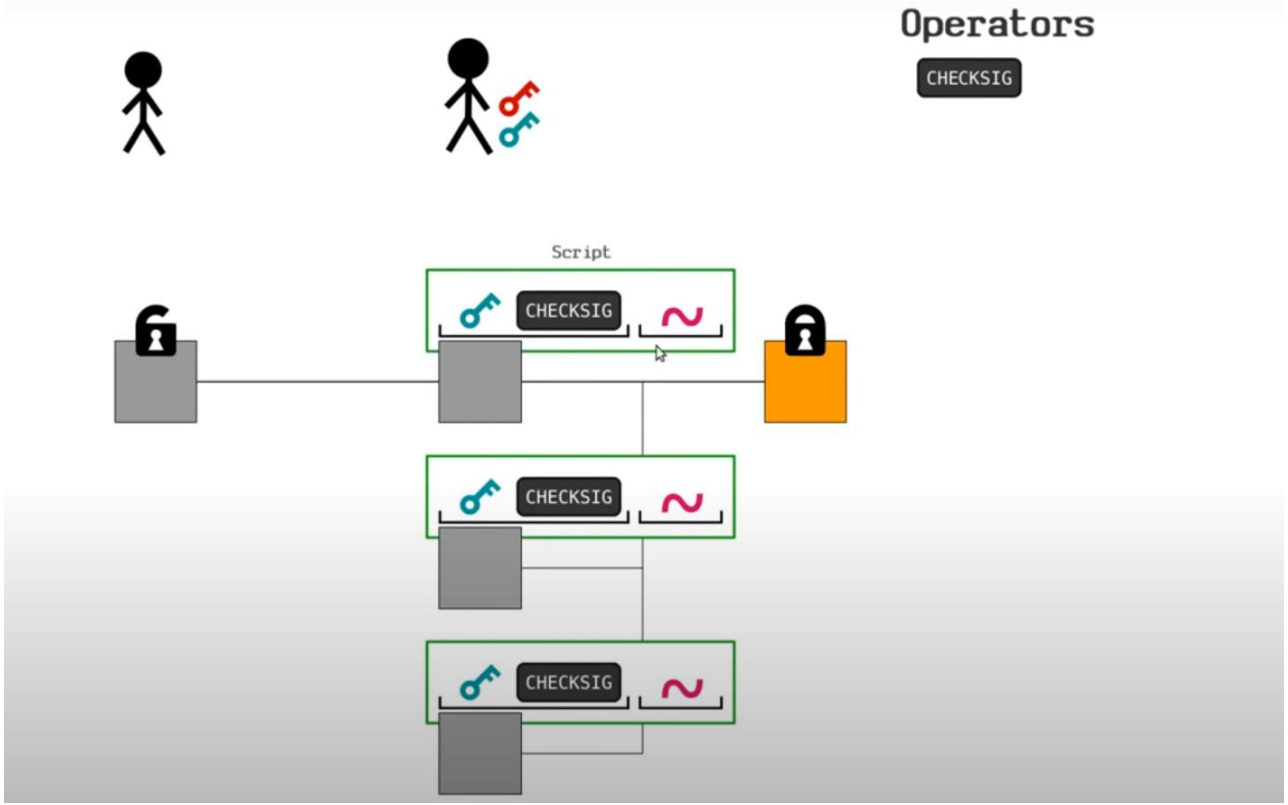
Script


Operators
CHECKSIG
Locking Script
Unlocking Script
CHECKSIG

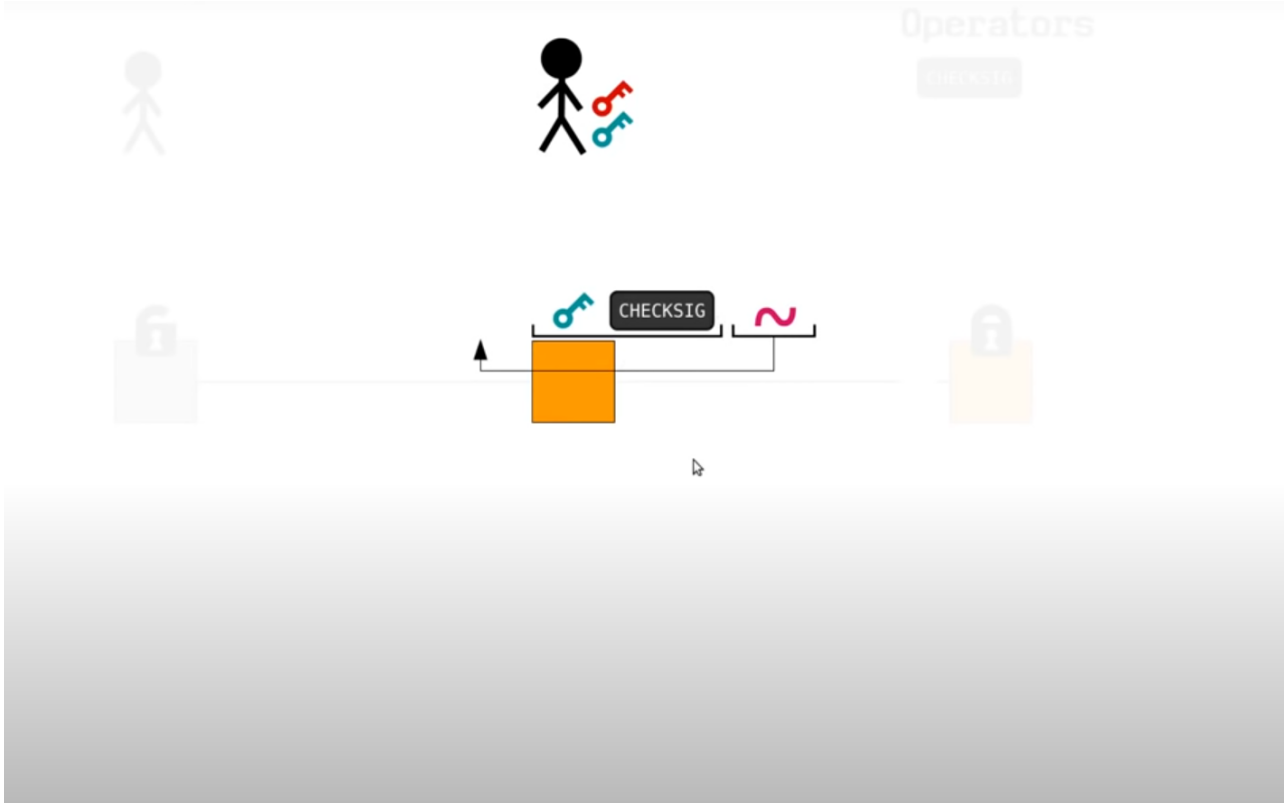
Script



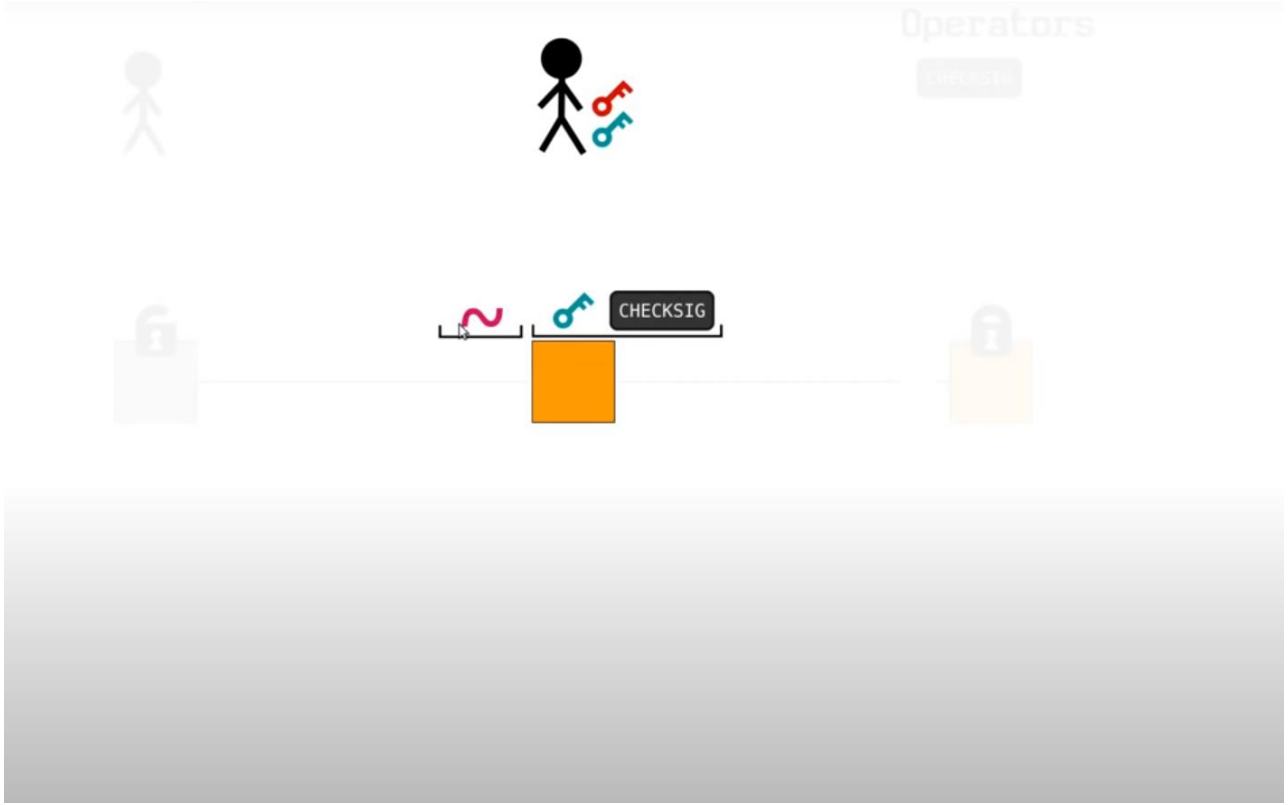
Script



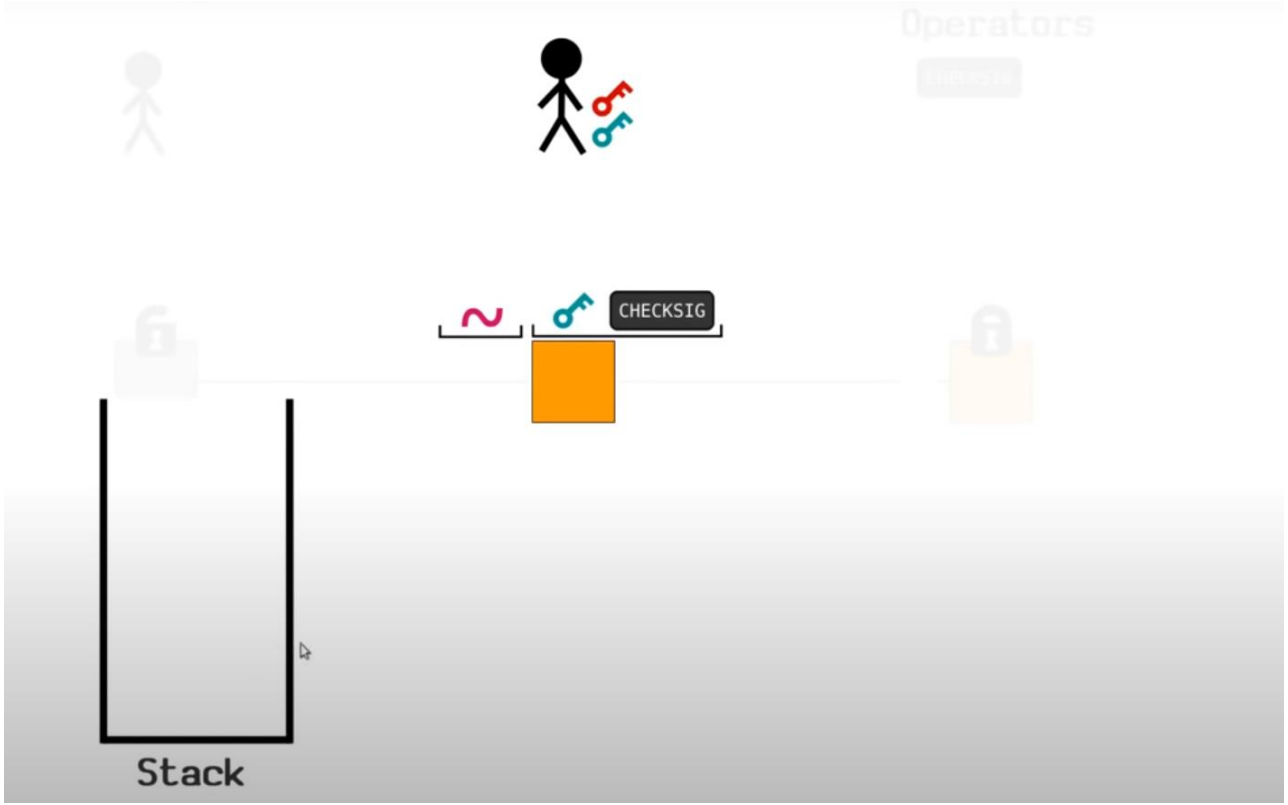
Script



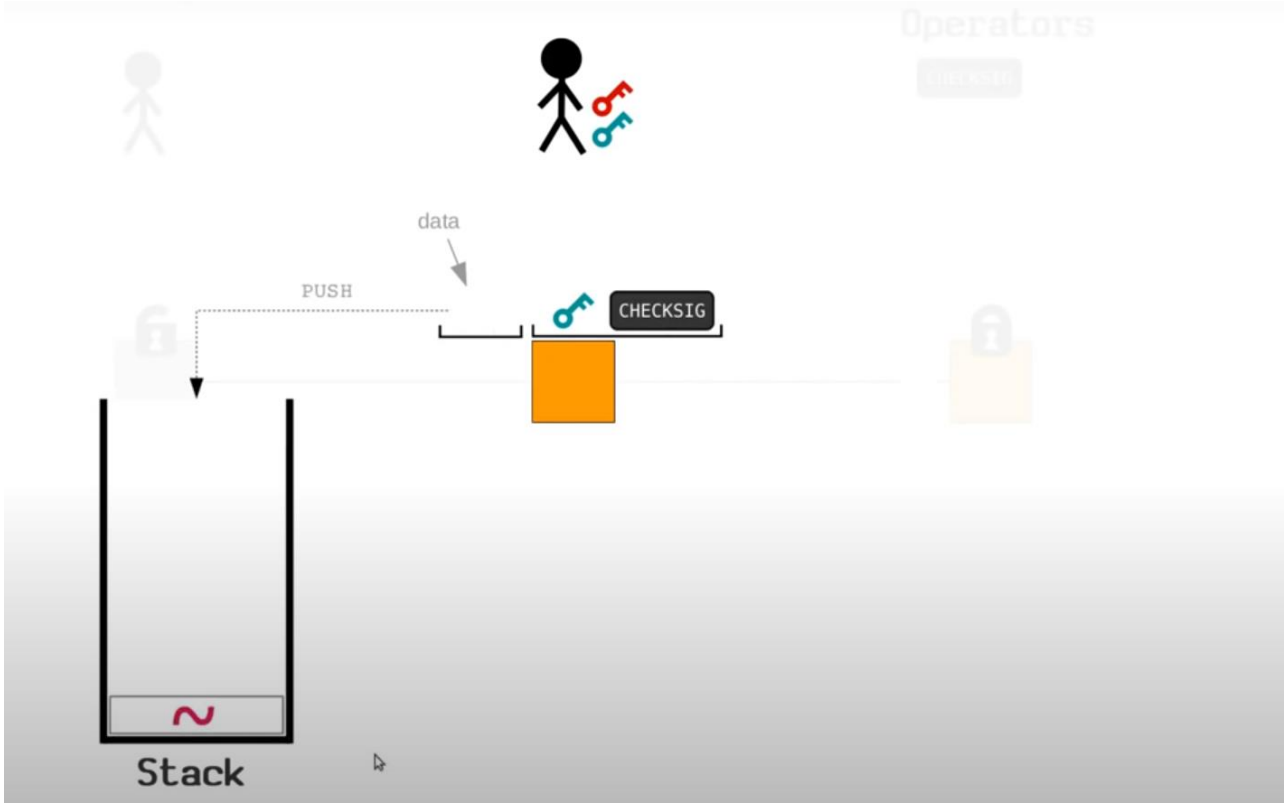
Script



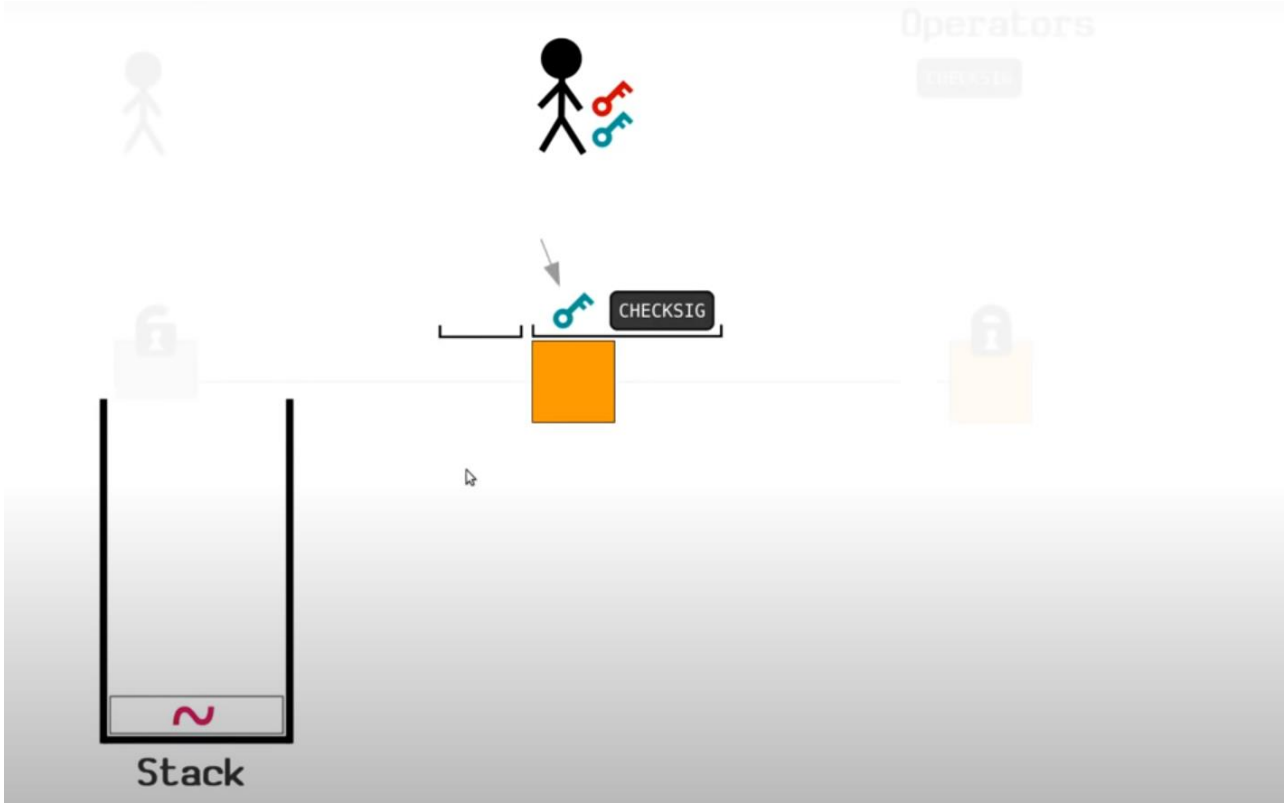
Script



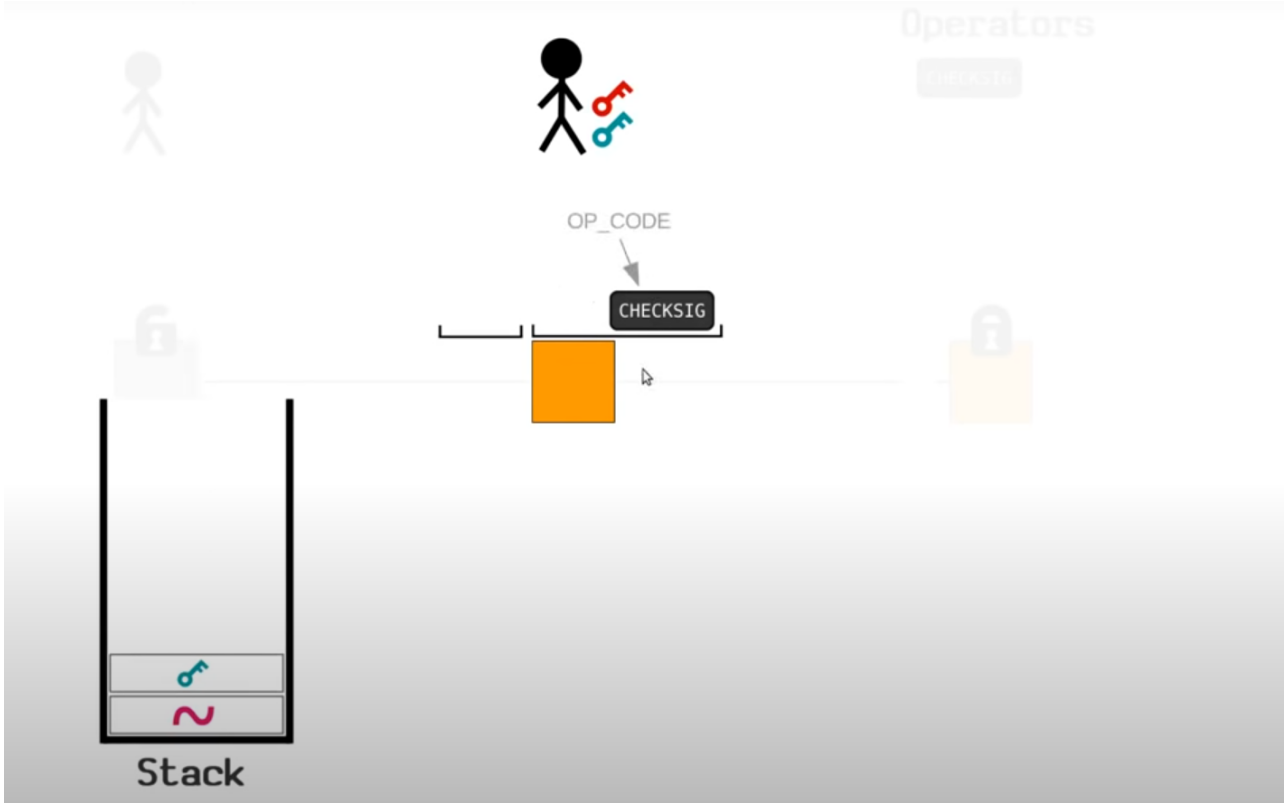
Script



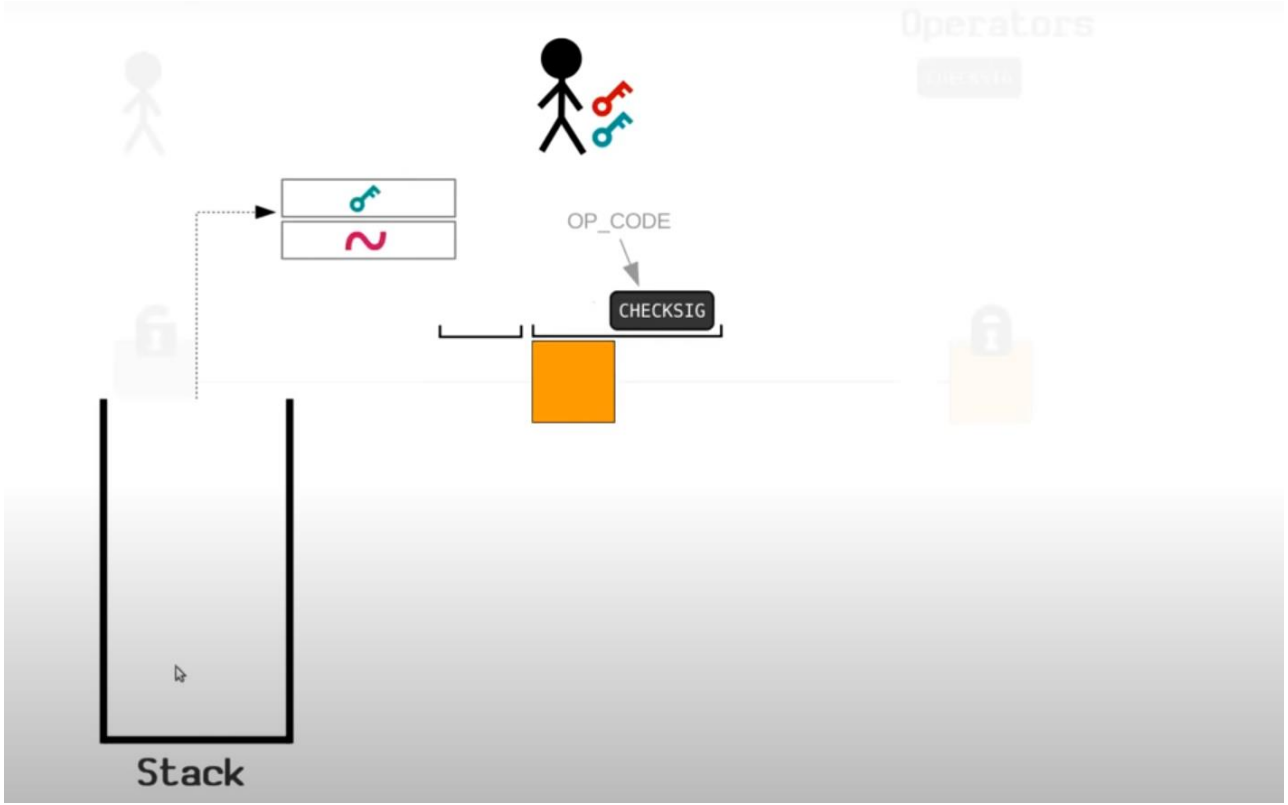
Script



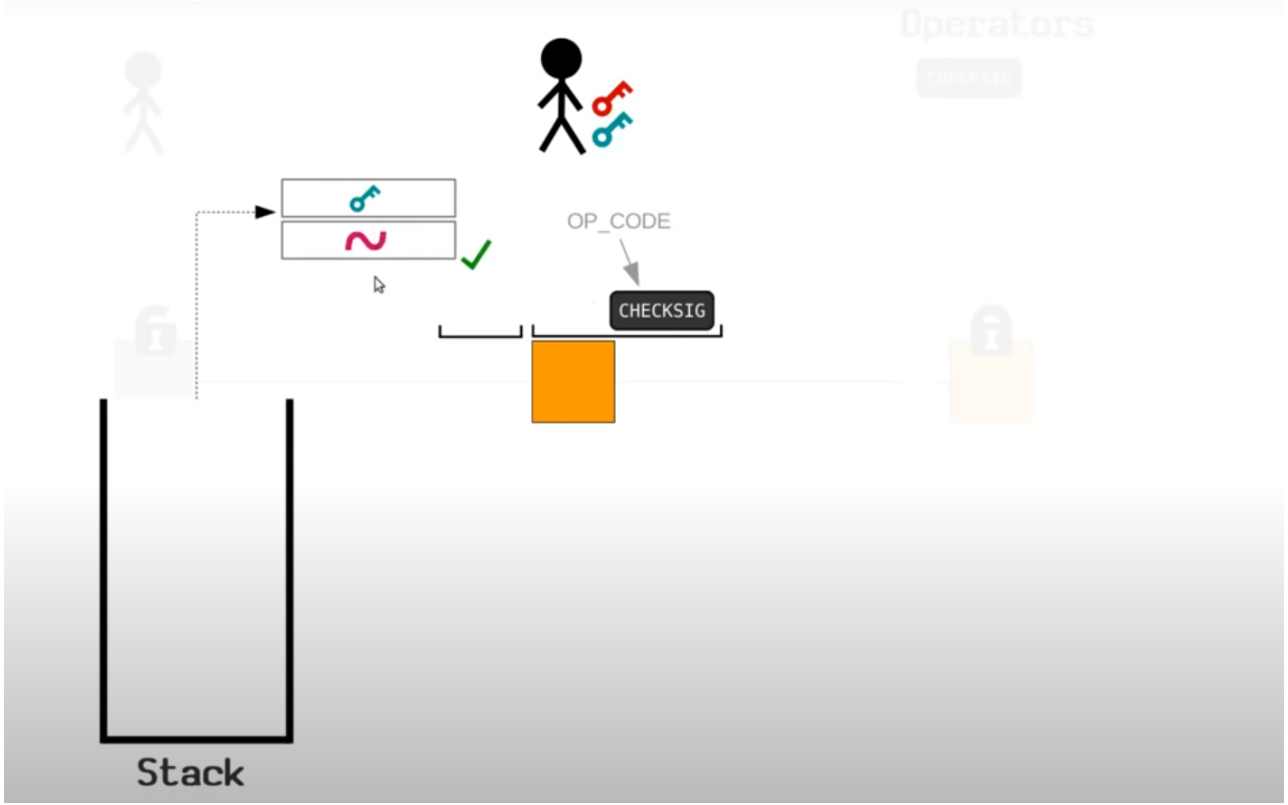
Script



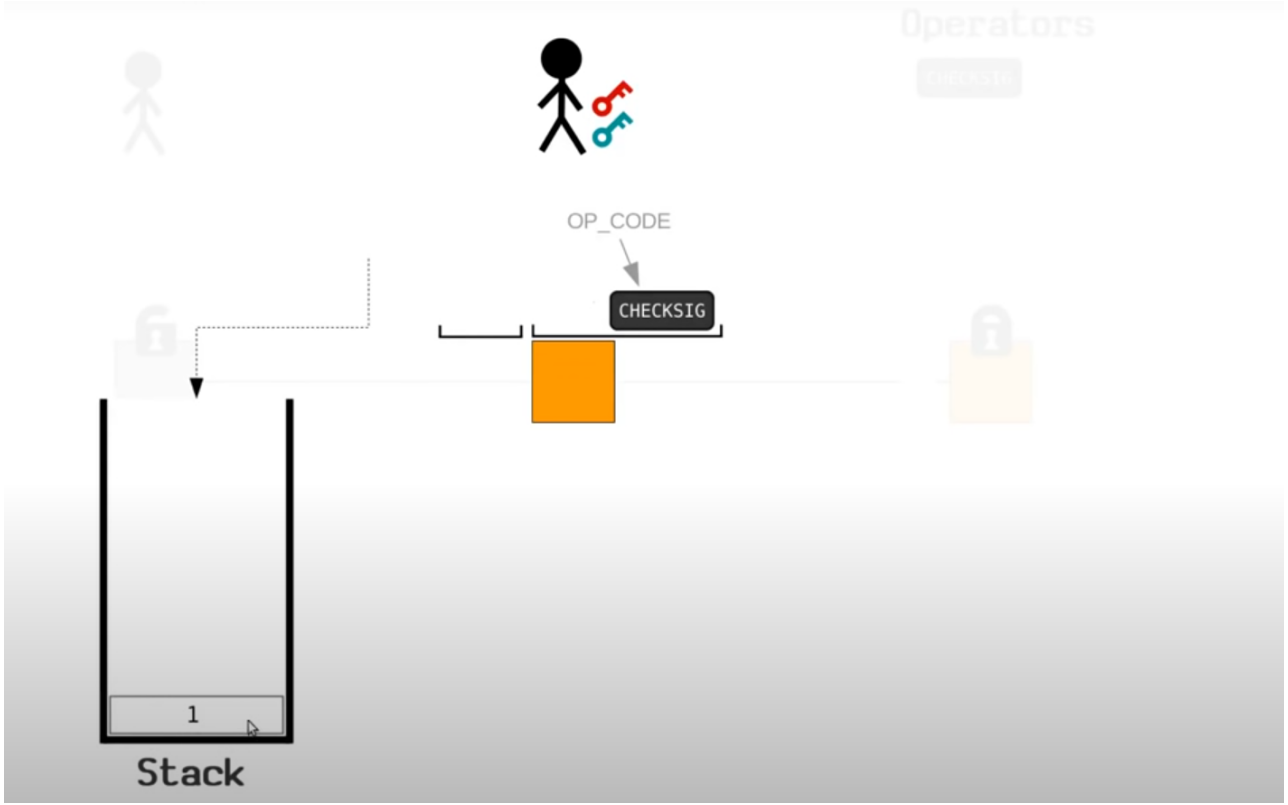
Script



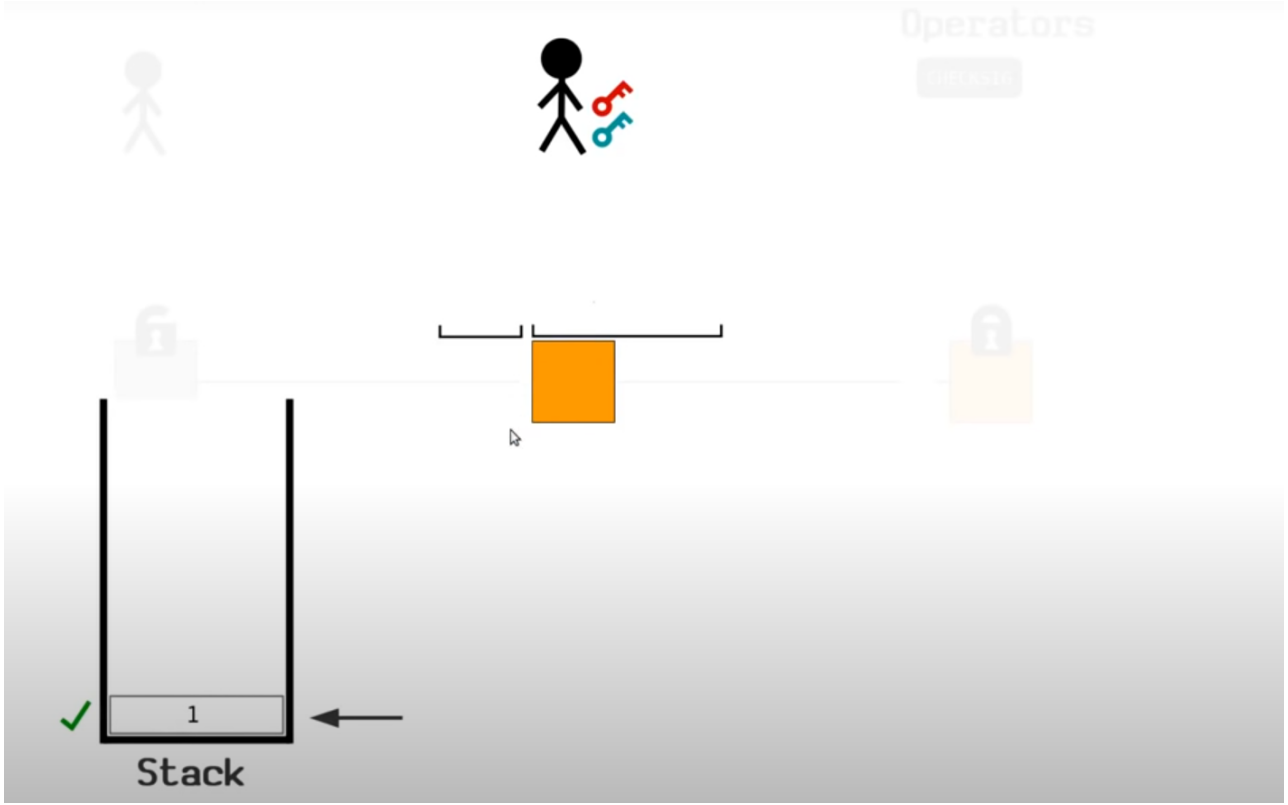
Script



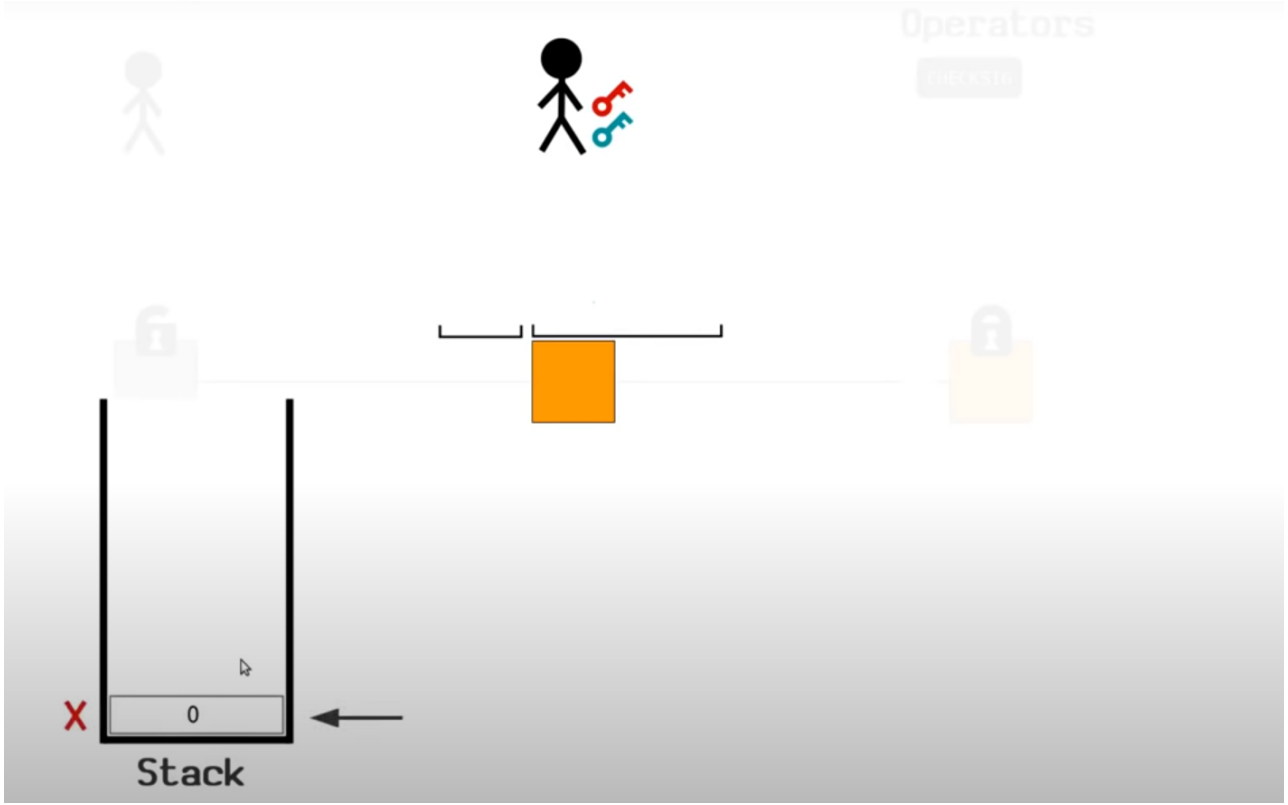
Script



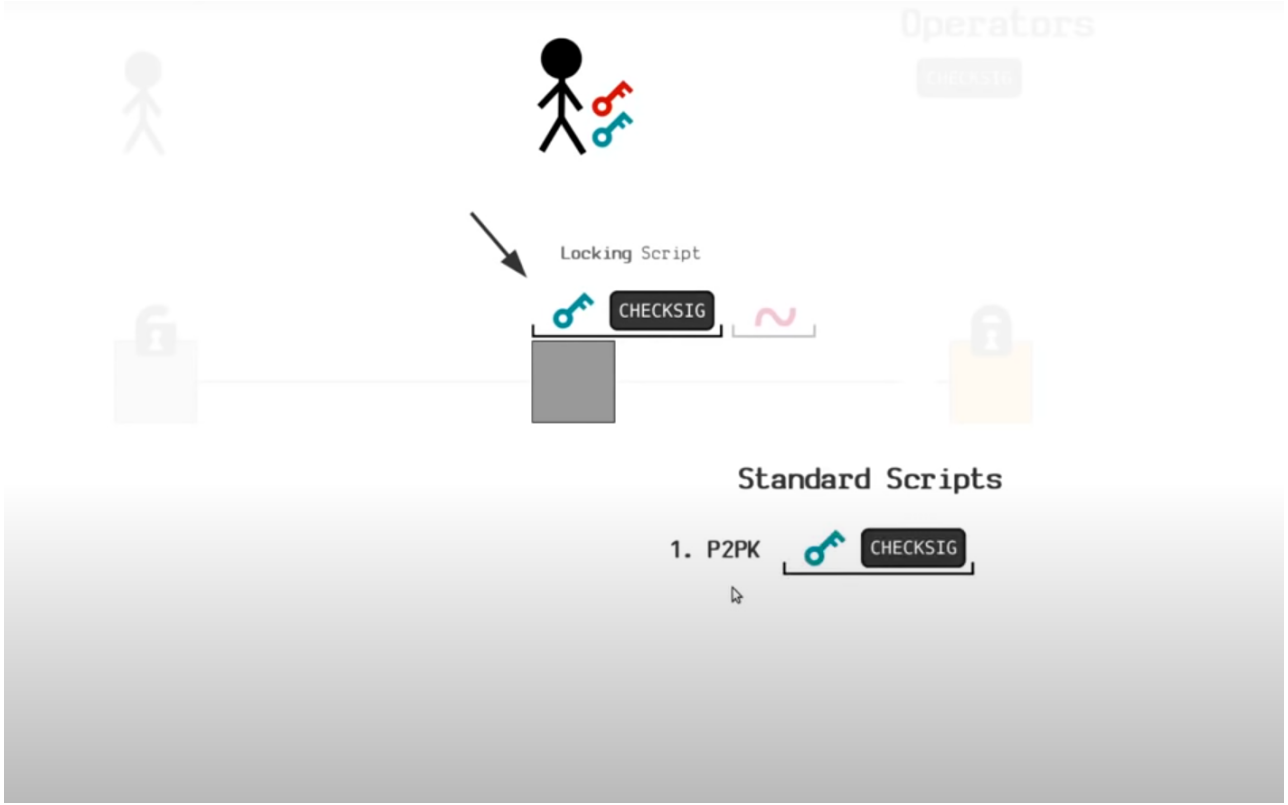
Script



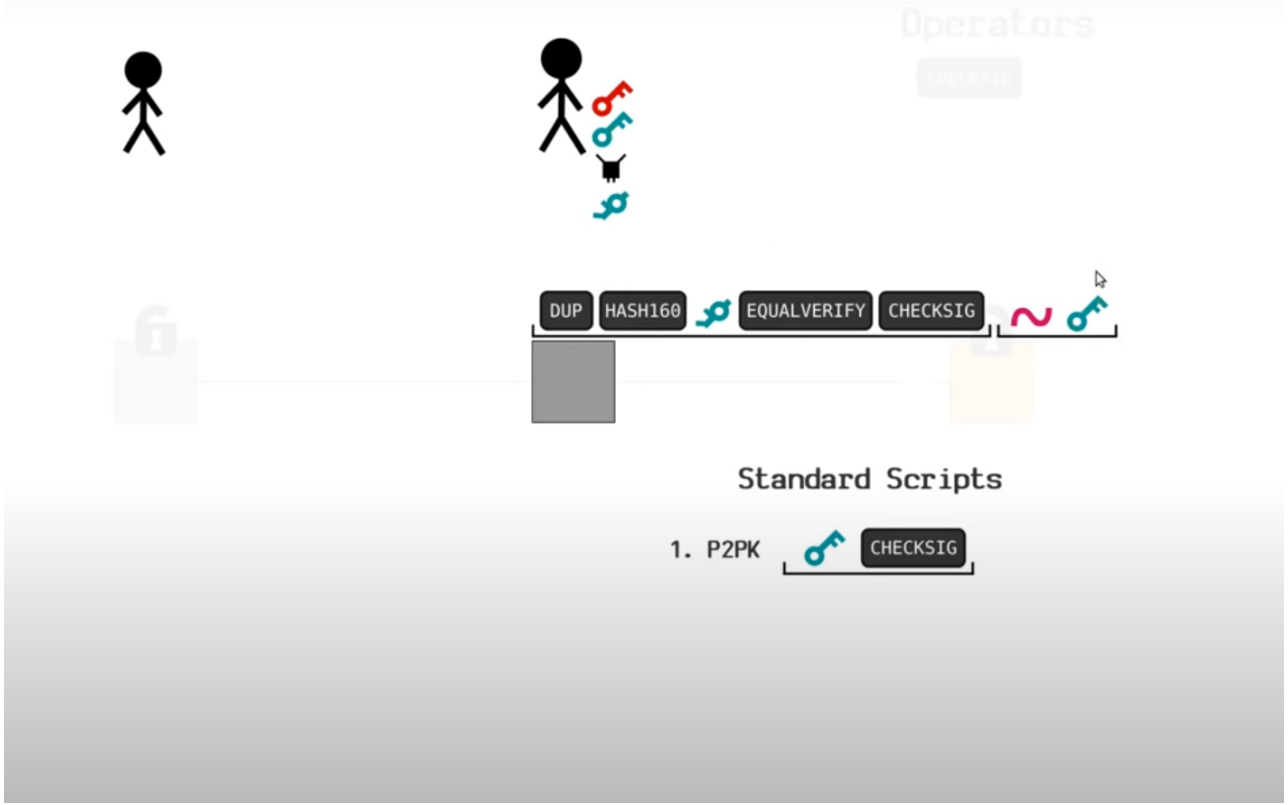
Script



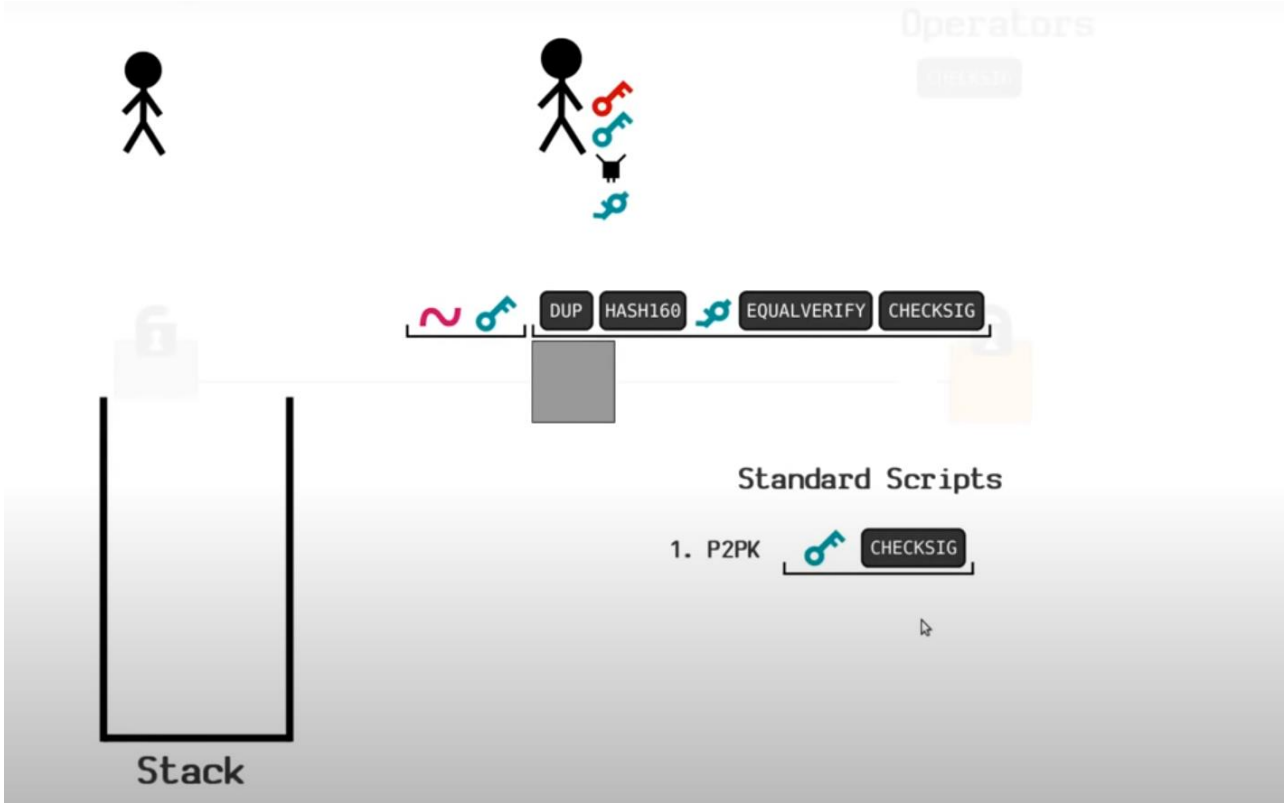
Pay to PublicKey (P2PK)



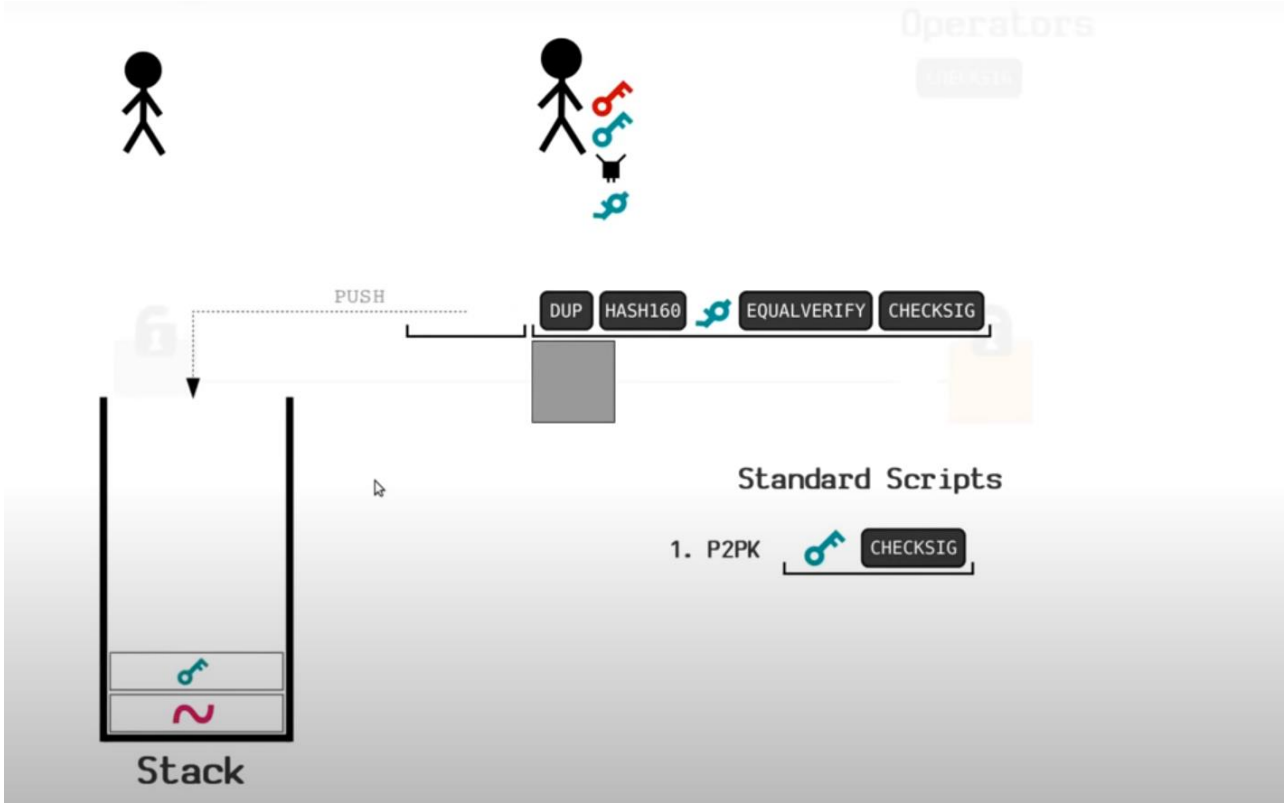
Pay to PublicKey Hash (P2PKH)



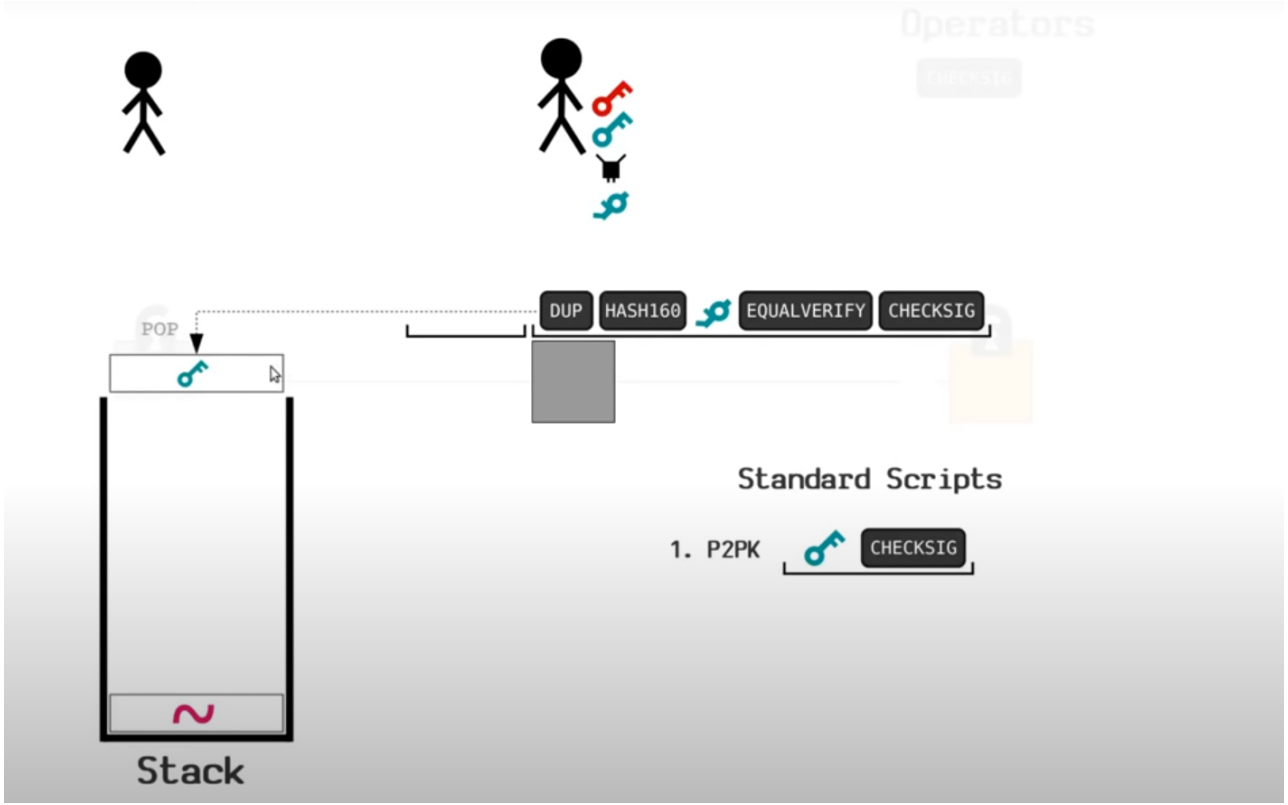
Pay to PublicKey Hash (P2PKH)



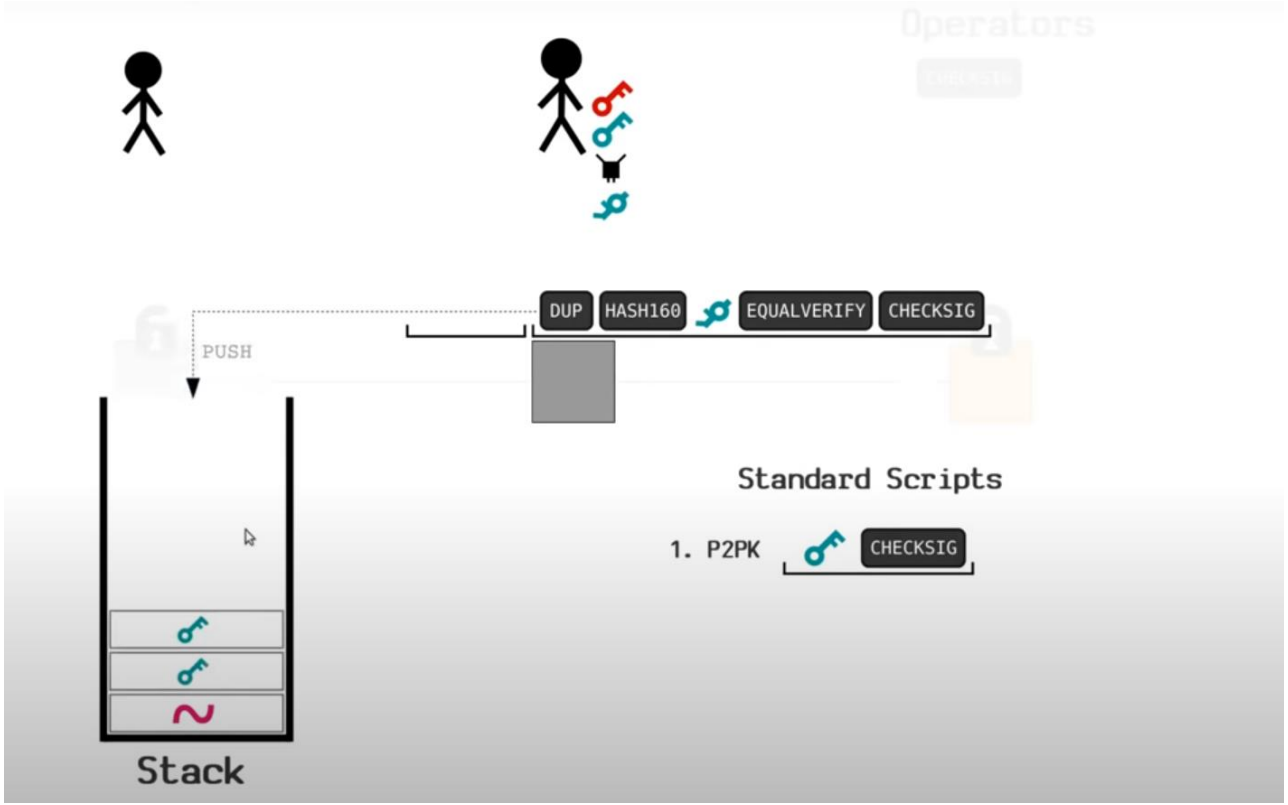
Pay to PublicKey Hash (P2PKH)



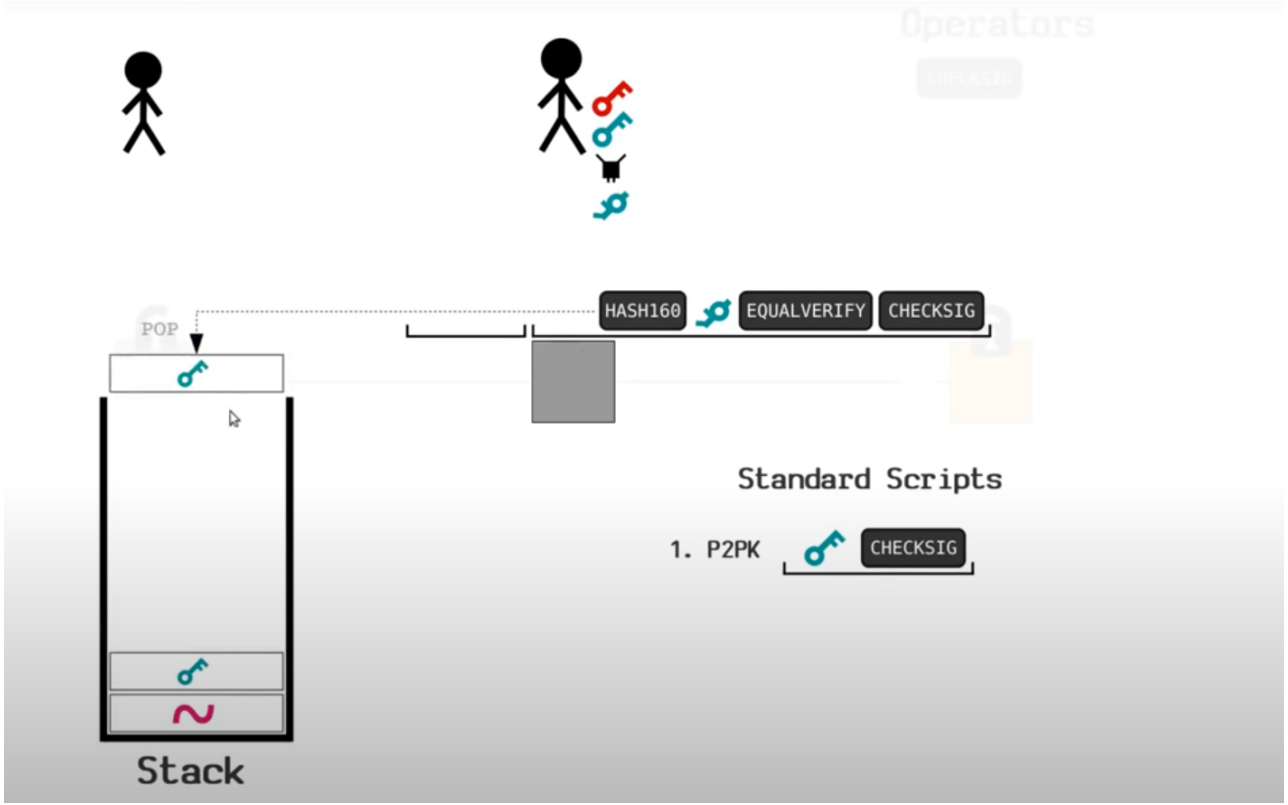
Pay to PublicKey Hash (P2PKH)



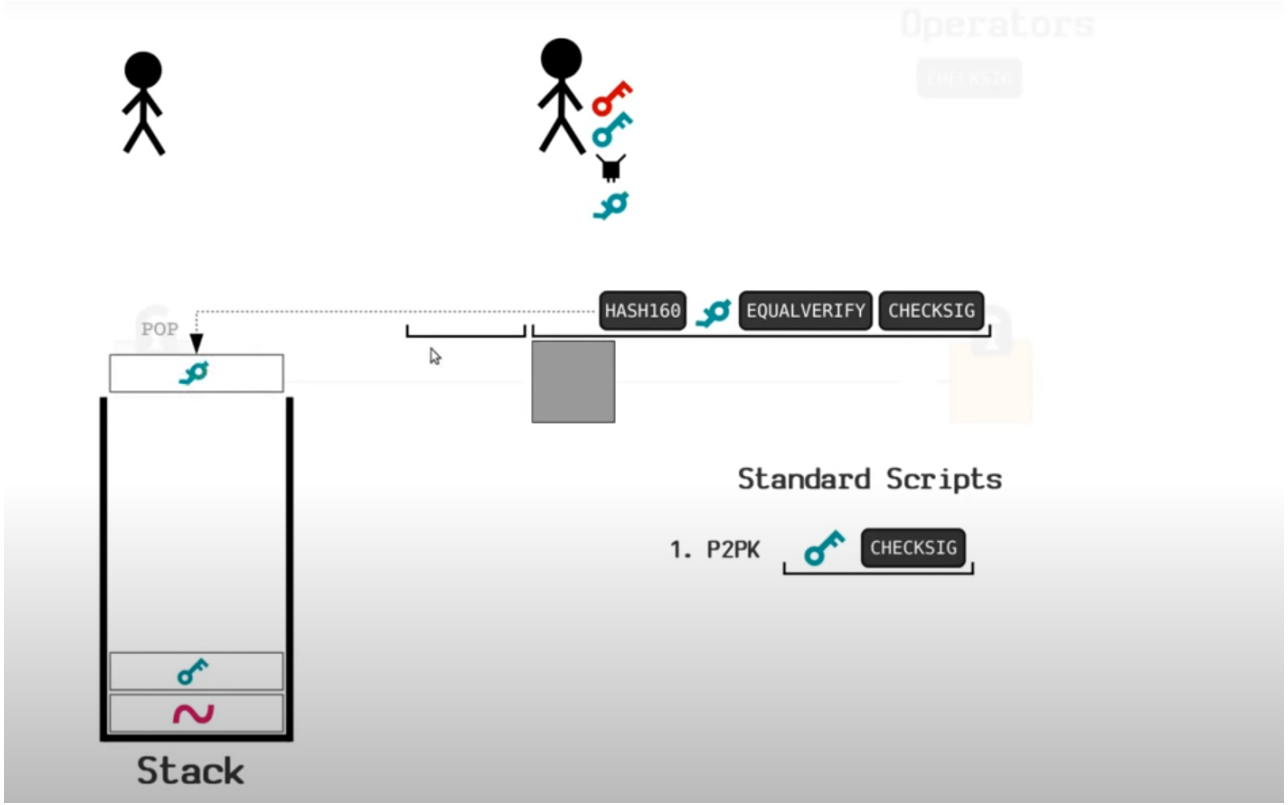
Pay to PublicKey Hash (P2PKH)



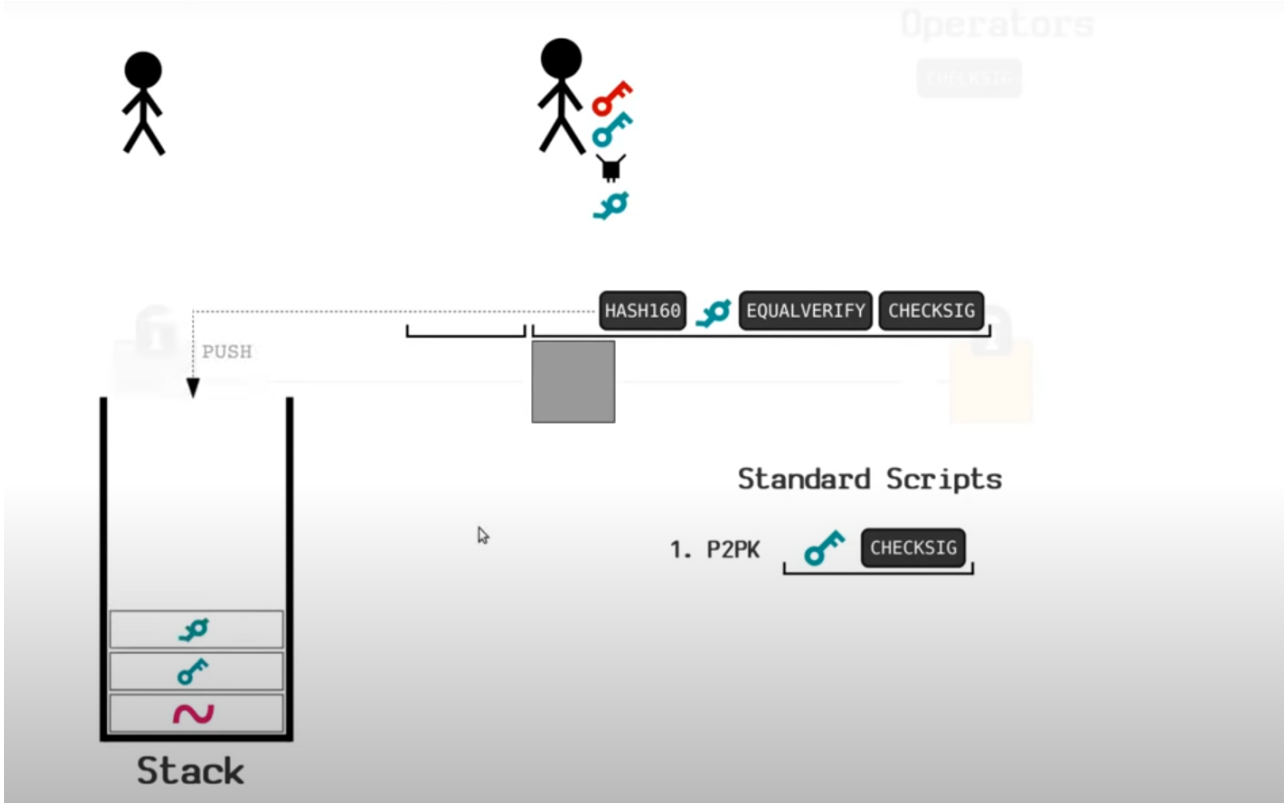
Pay to PublicKey Hash (P2PKH)



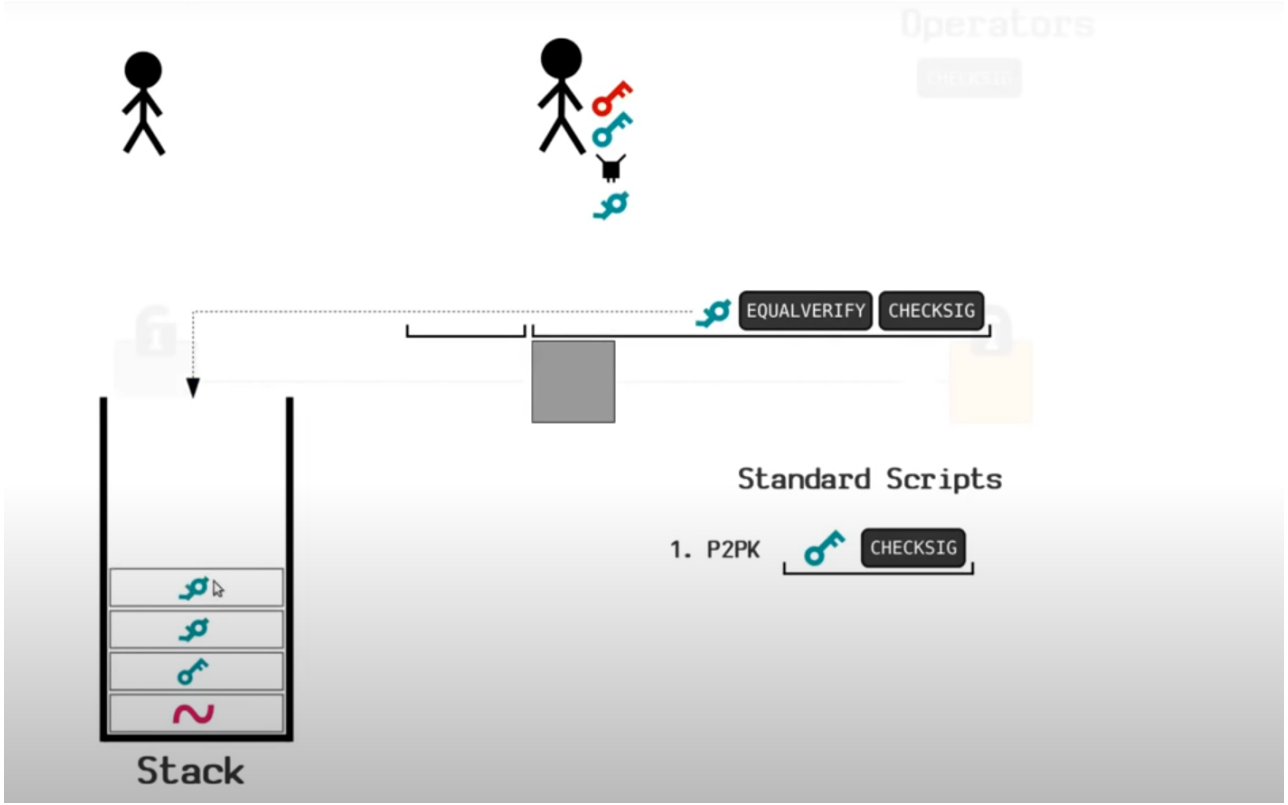
Pay to PublicKey Hash (P2PKH)



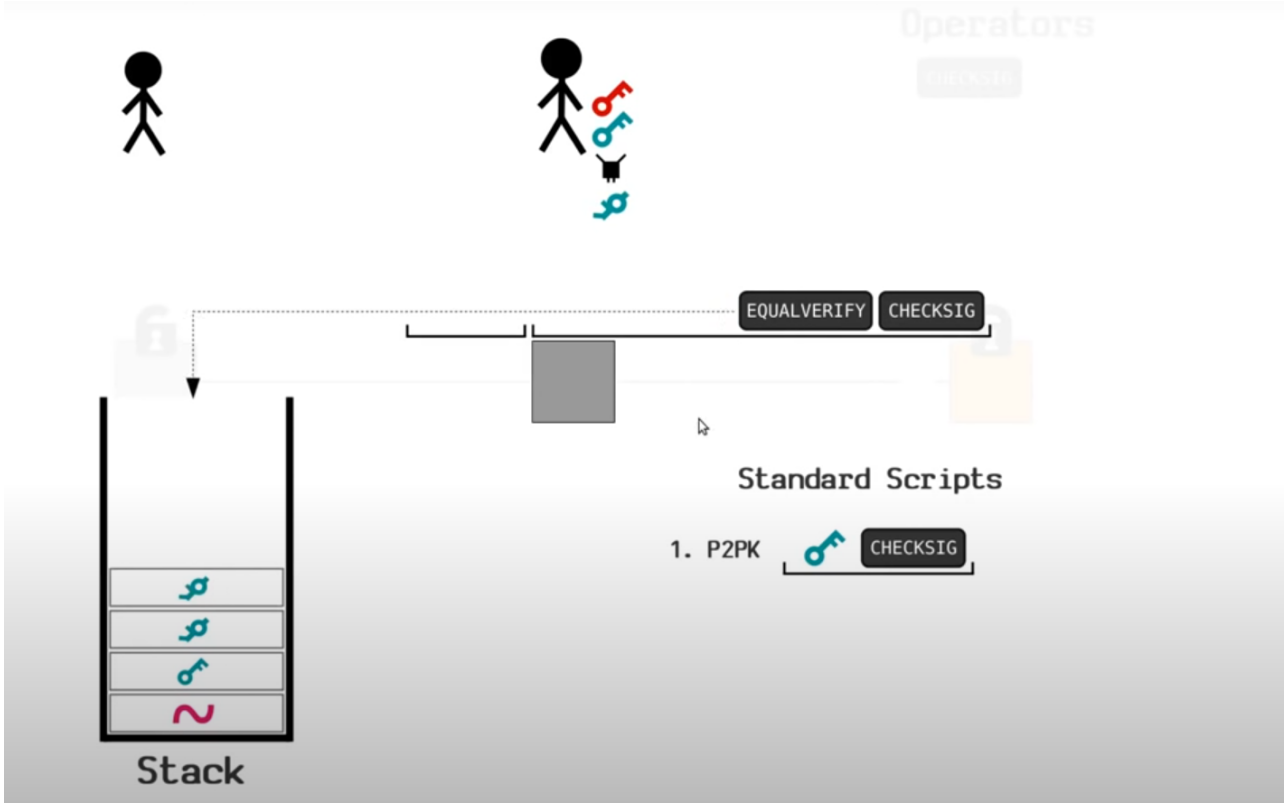
Pay to PublicKey Hash (P2PKH)



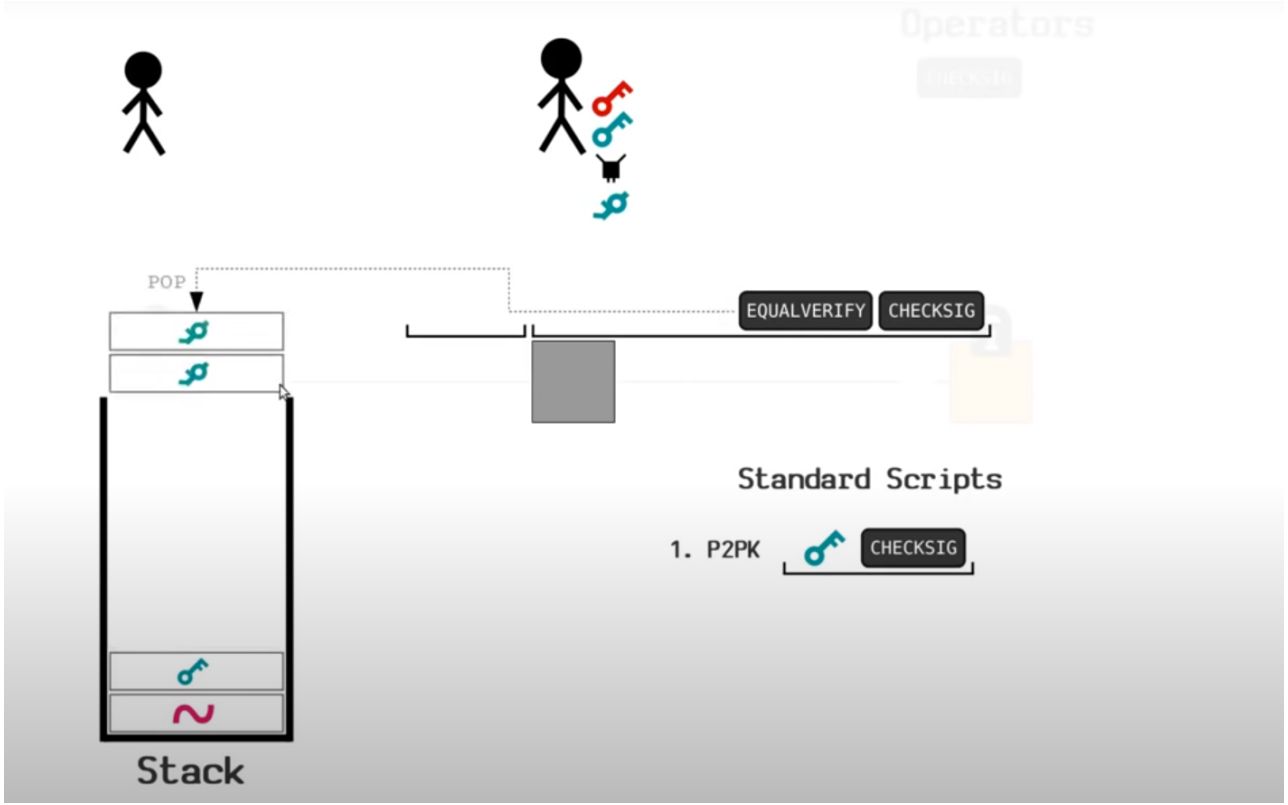
Pay to PublicKey Hash (P2PKH)



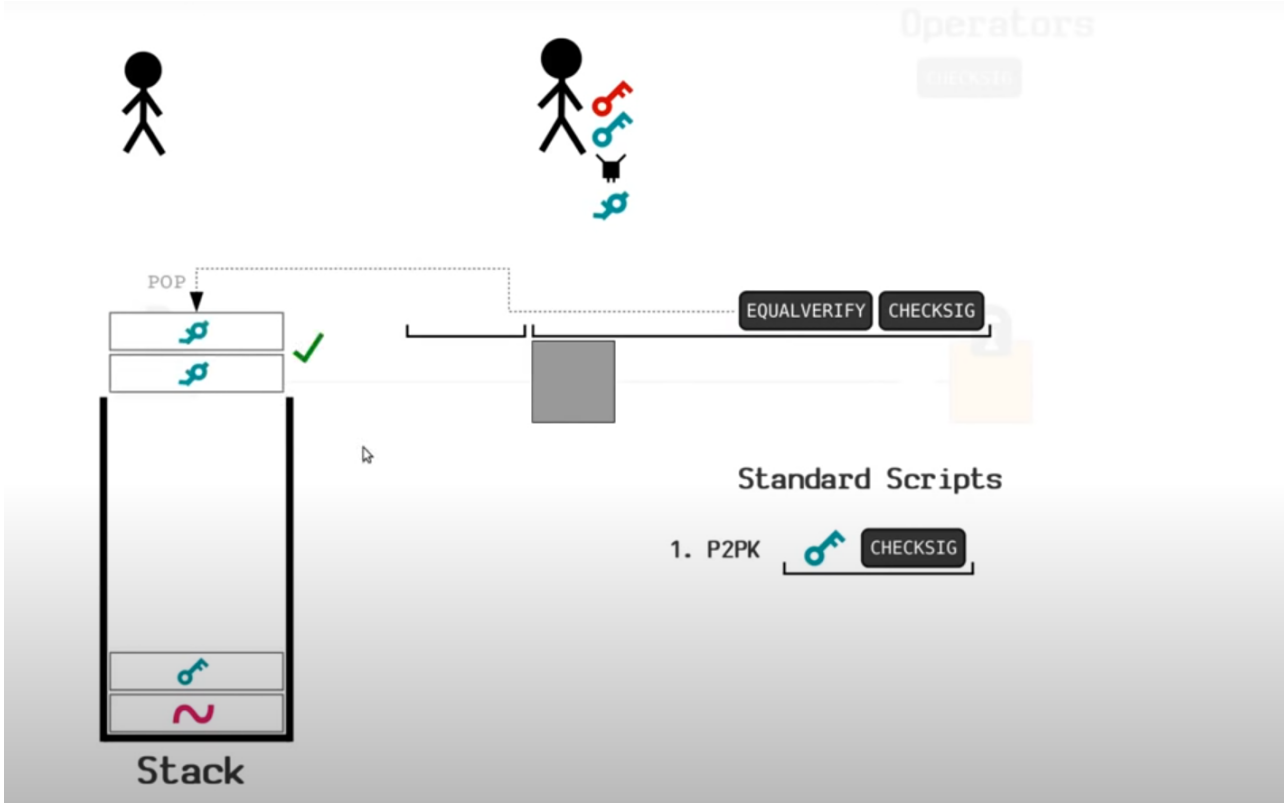
Pay to PublicKey Hash (P2PKH)



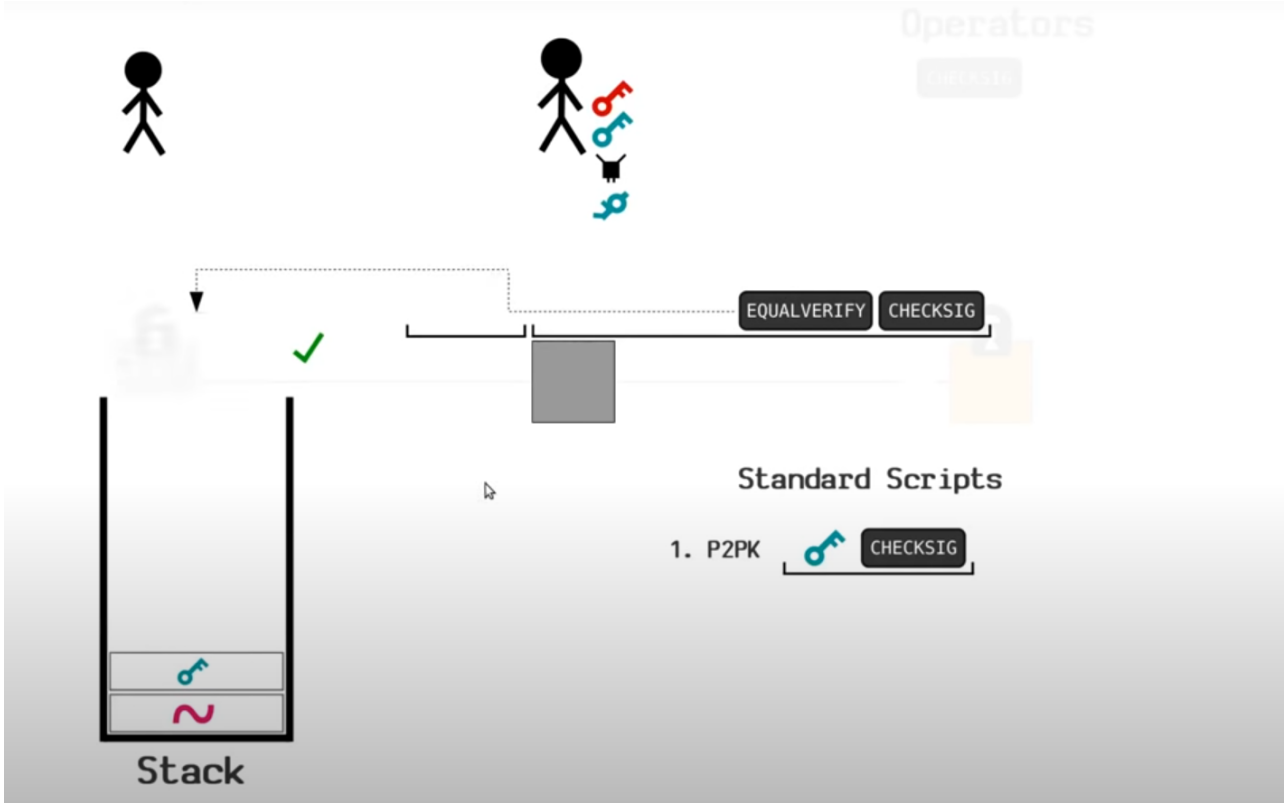
Pay to PublicKey Hash (P2PKH)



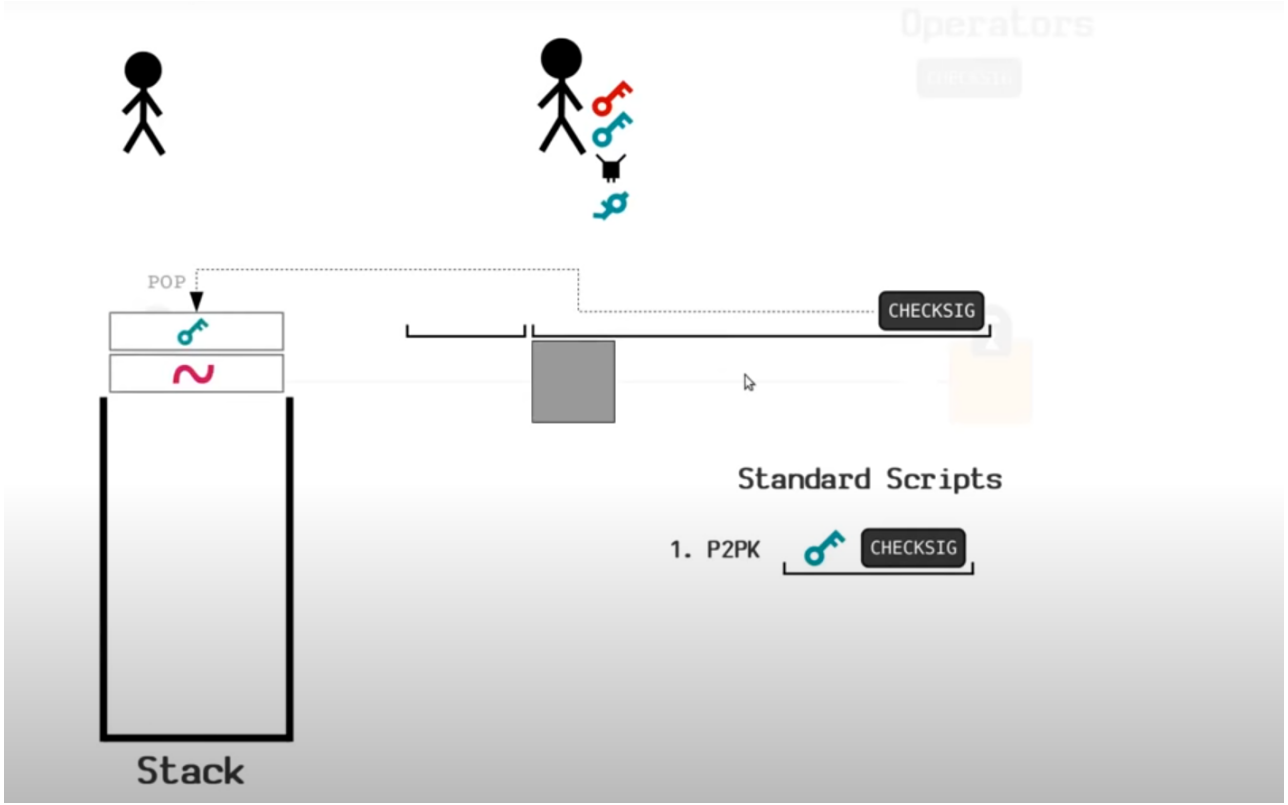
Pay to PublicKey Hash (P2PKH)



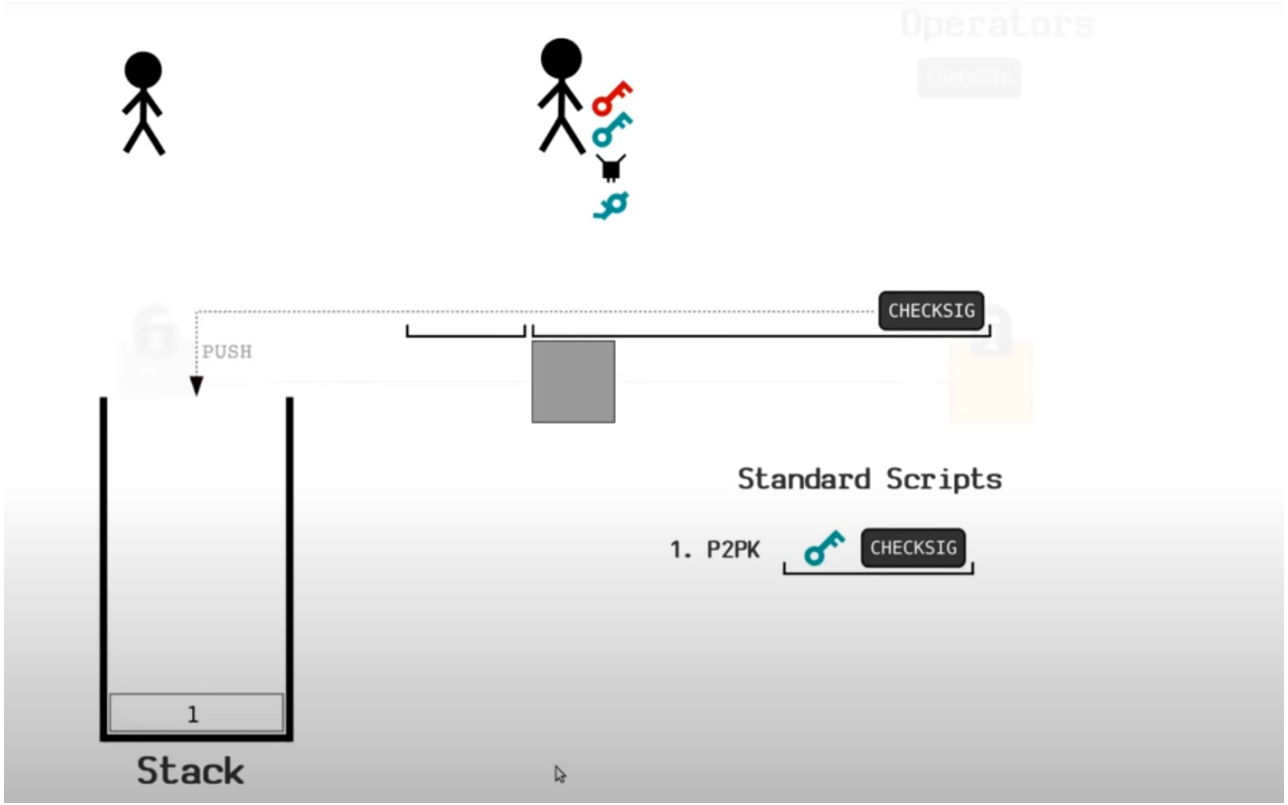
Pay to PublicKey Hash (P2PKH)



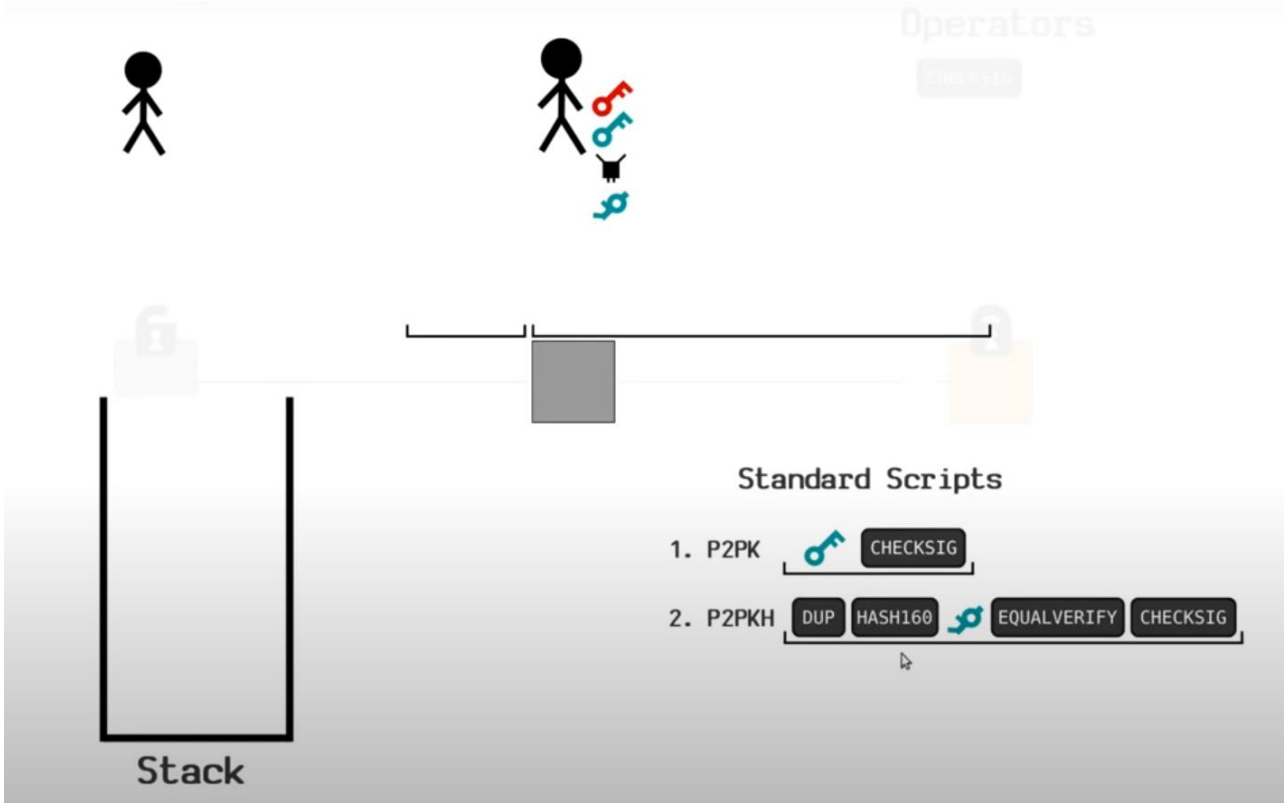
Pay to PublicKey Hash (P2PKH)



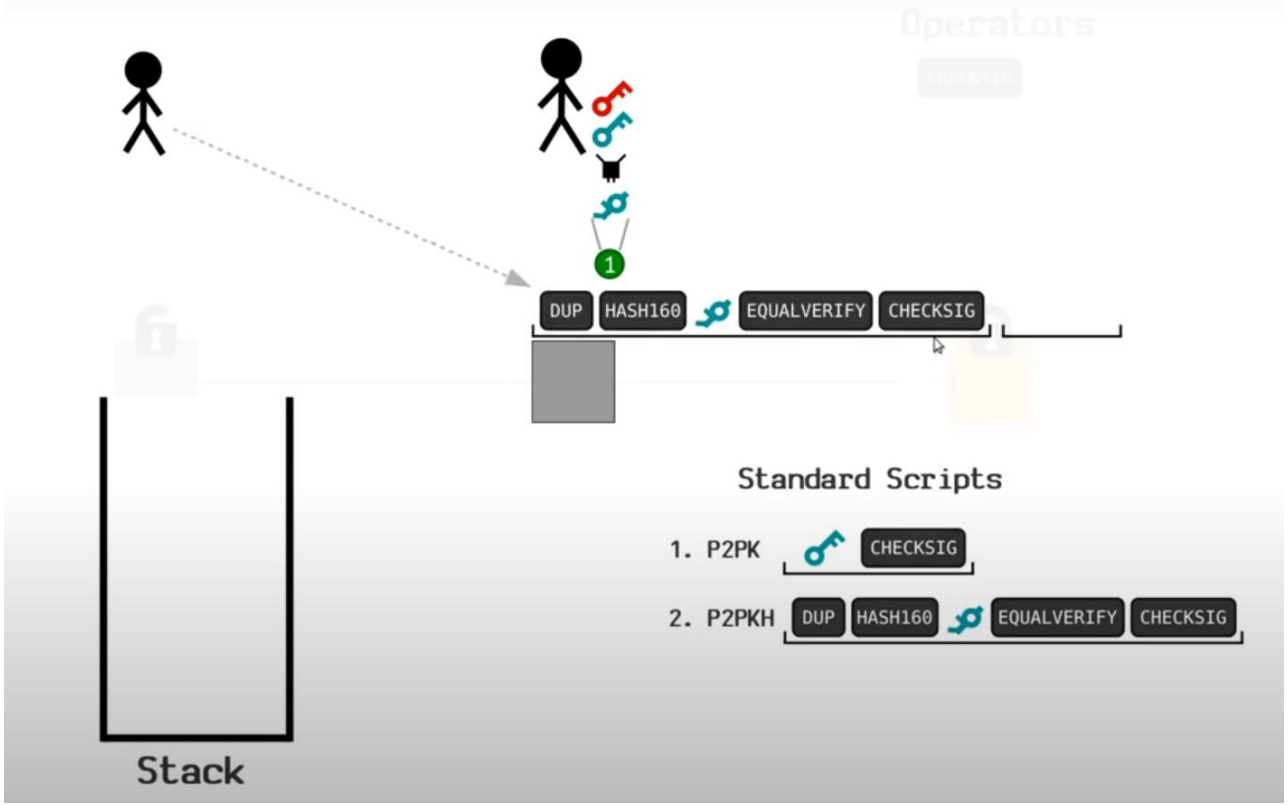
Pay to PublicKey Hash (P2PKH)



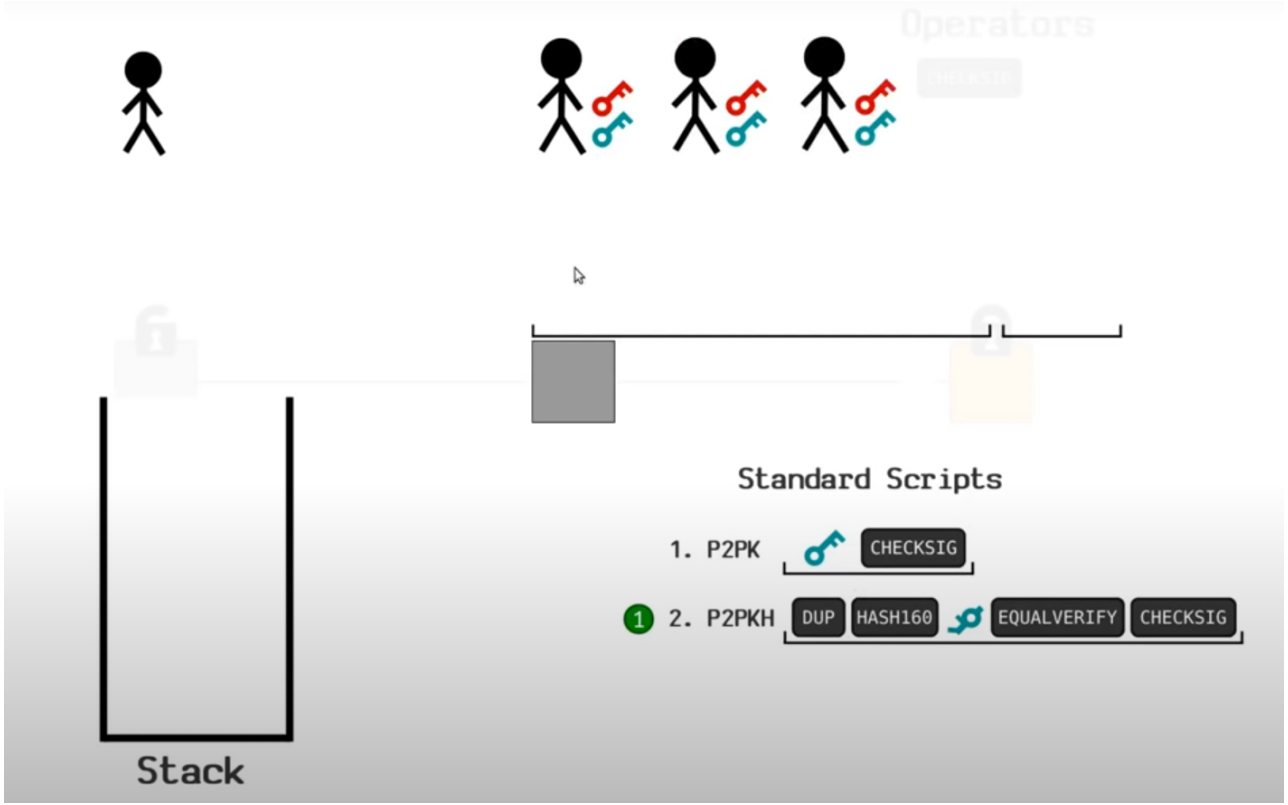
Pay to PublicKey Hash (P2PKH)



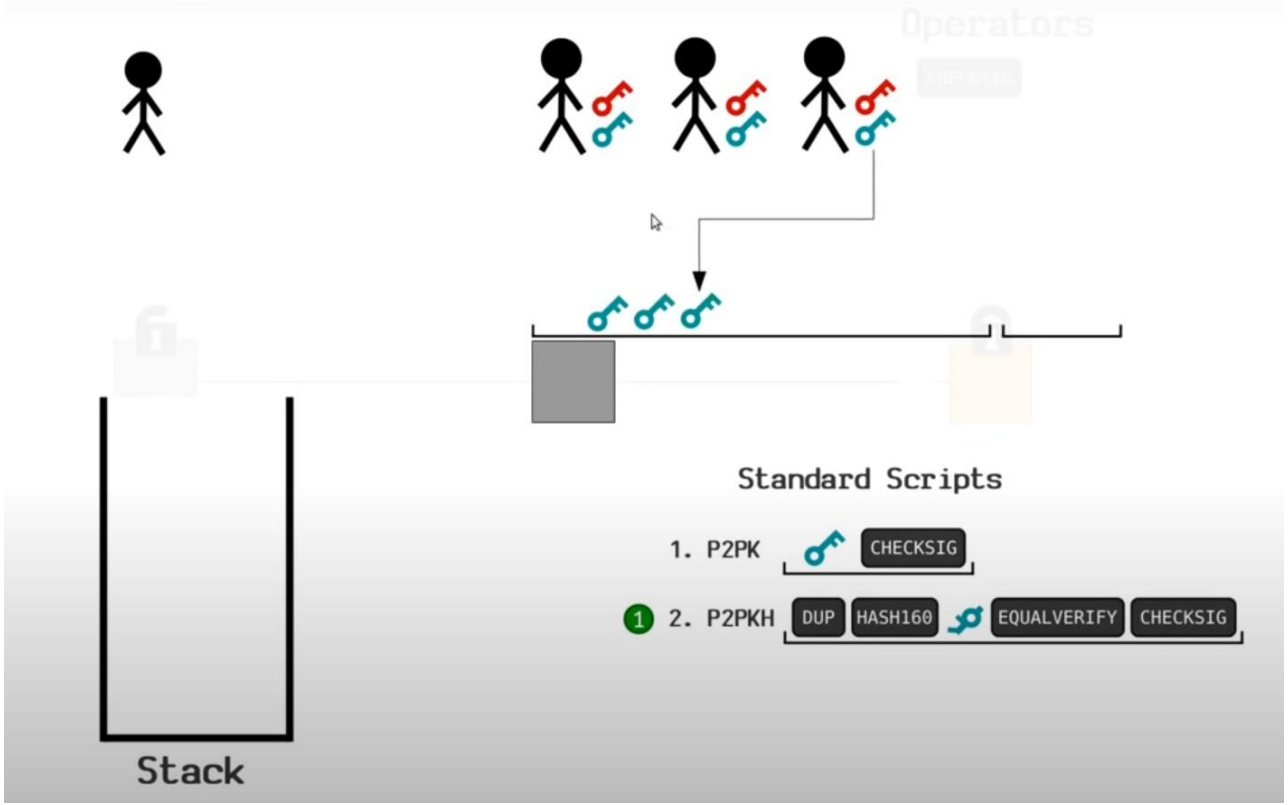
Pay to PublicKey Hash (P2PKH)



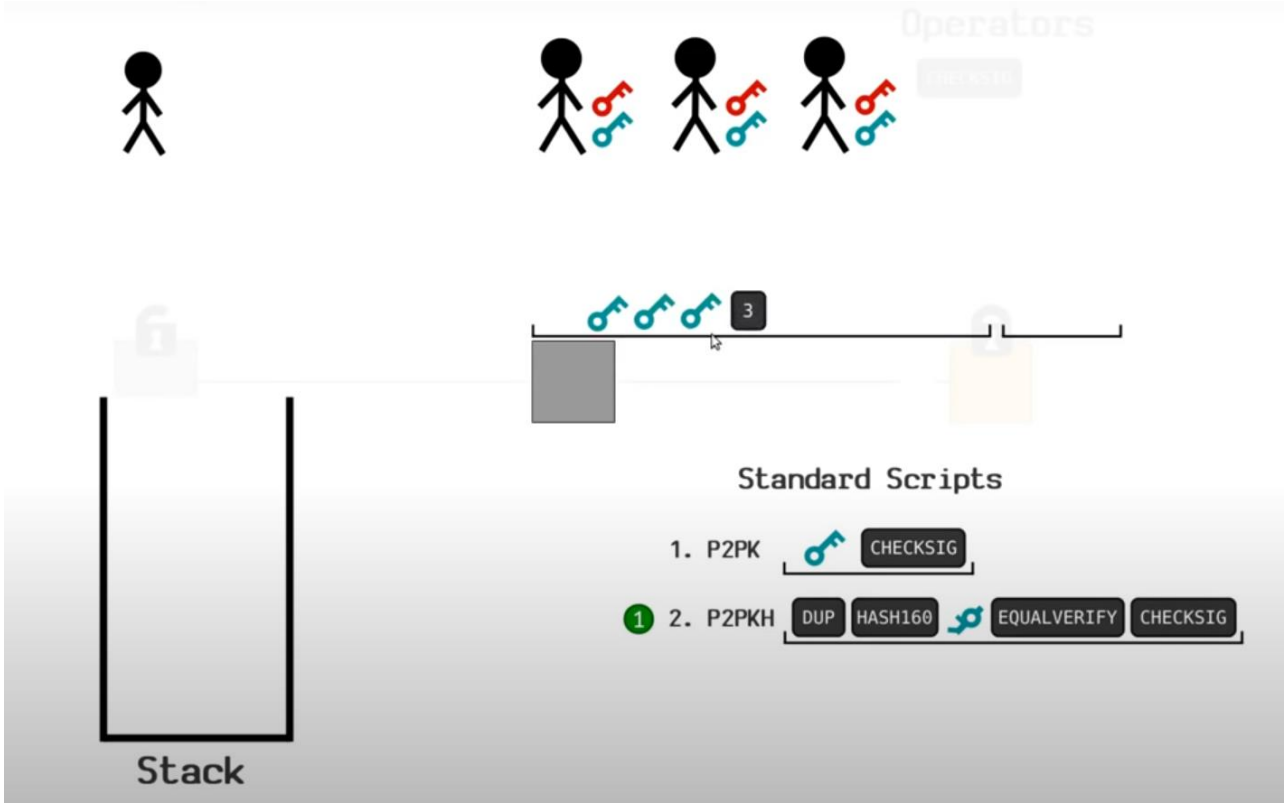
Pay to MultiSig (P2MS)



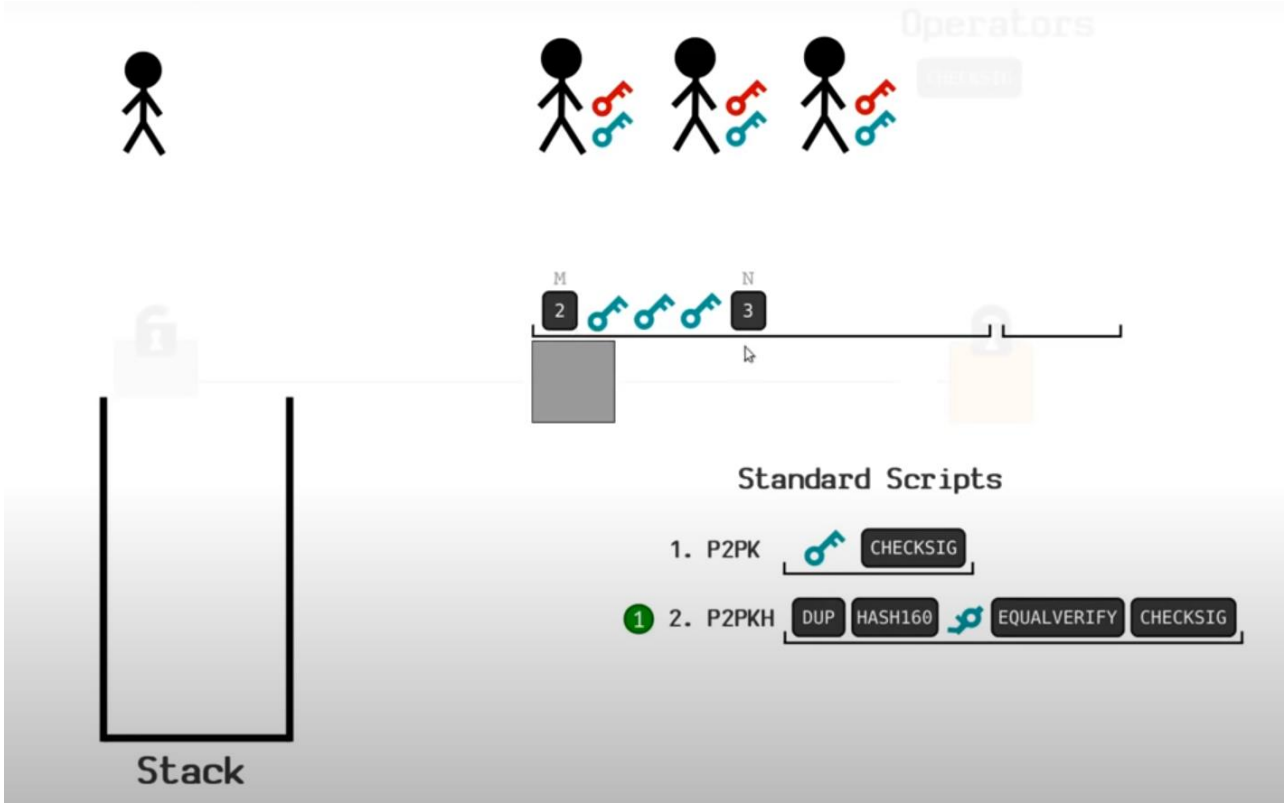
Pay to MultiSig (P2MS)



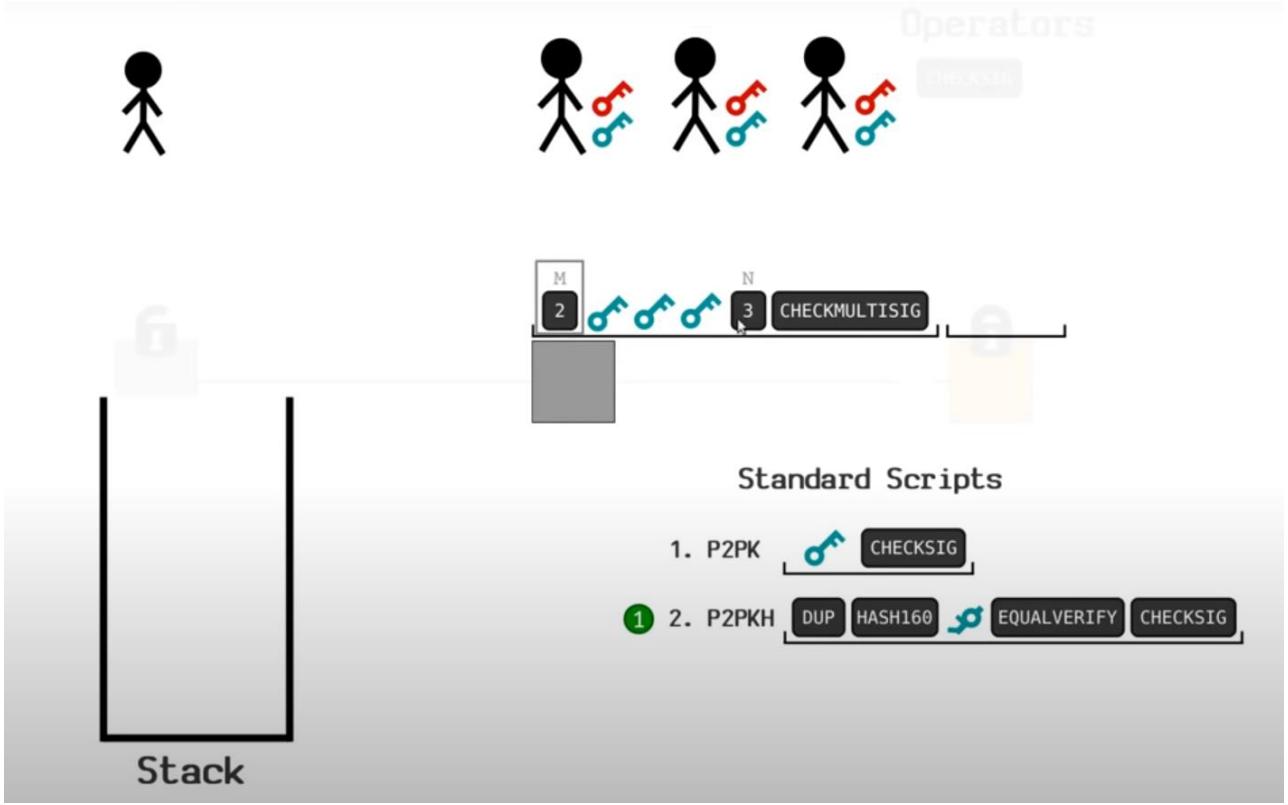
Pay to MultiSig (P2MS)



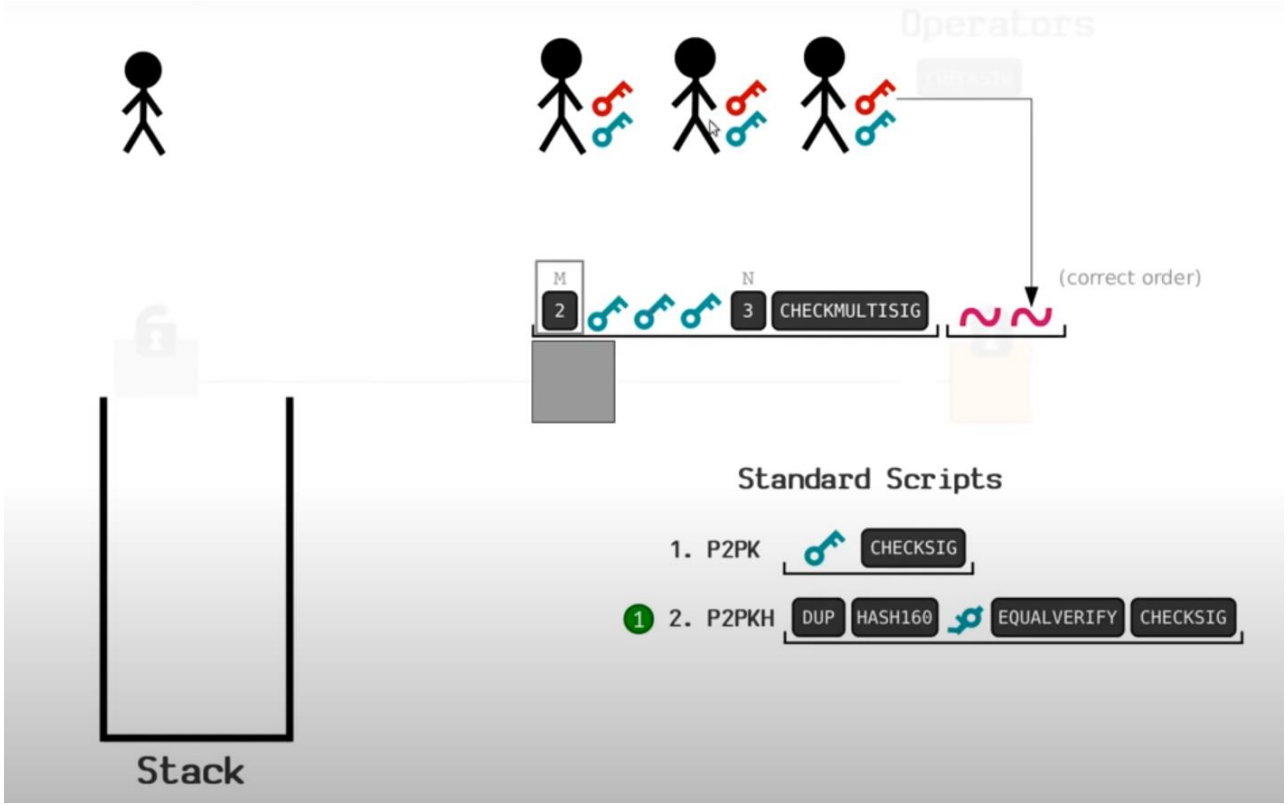
Pay to MultiSig (P2MS)



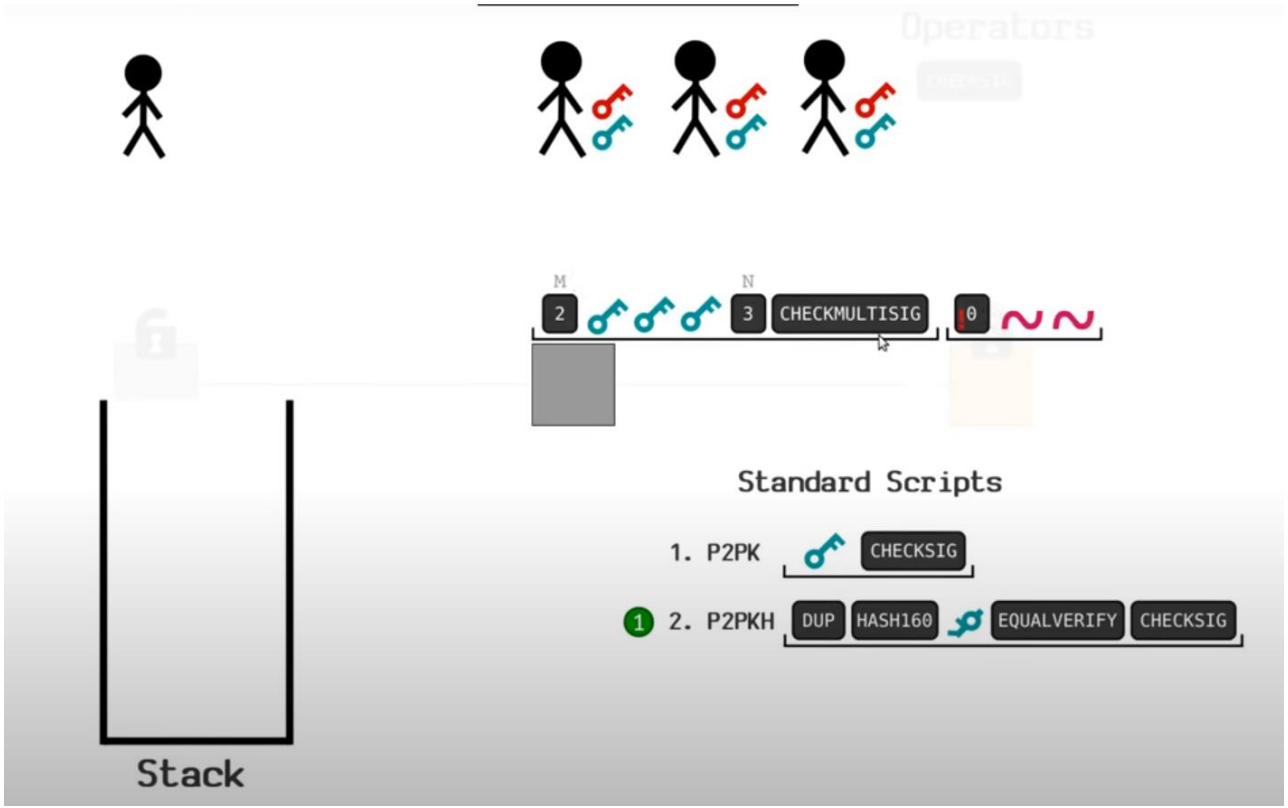
Pay to MultiSig (P2MS)



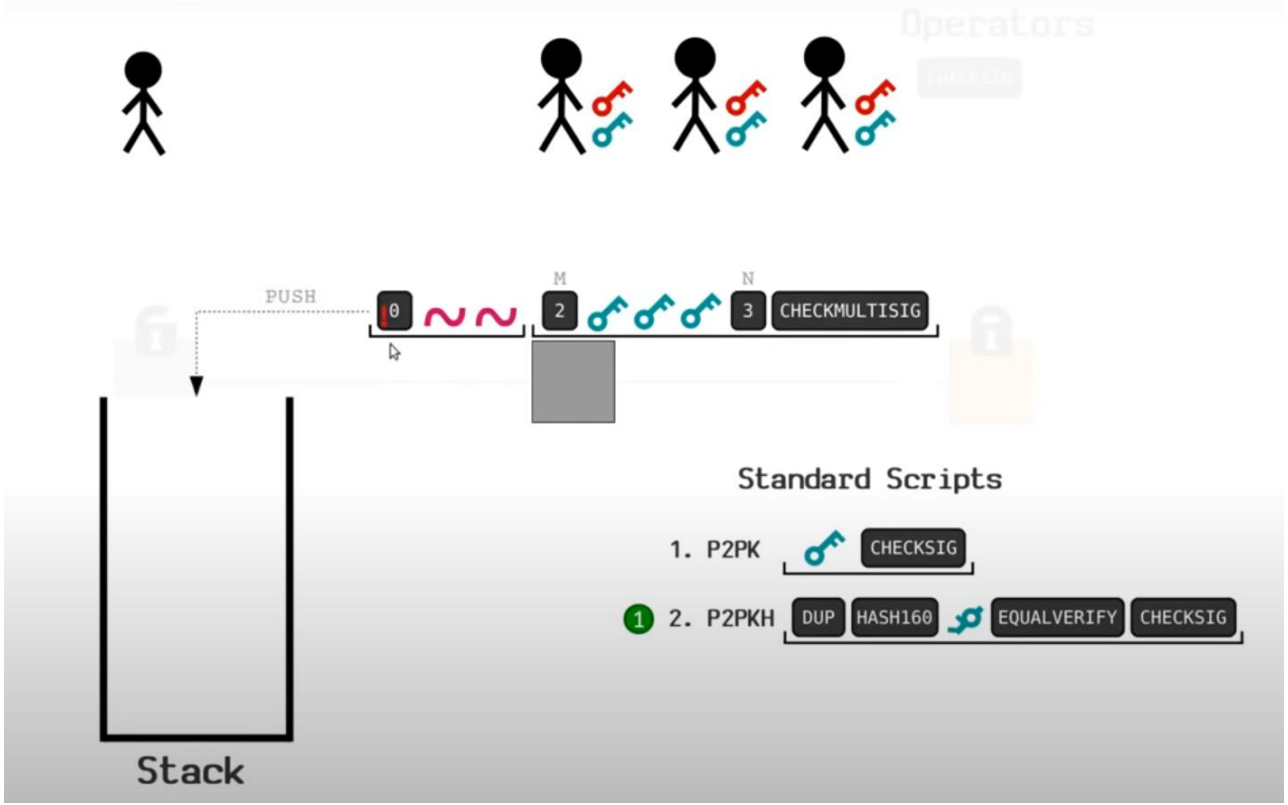
Pay to MultiSig (P2MS)



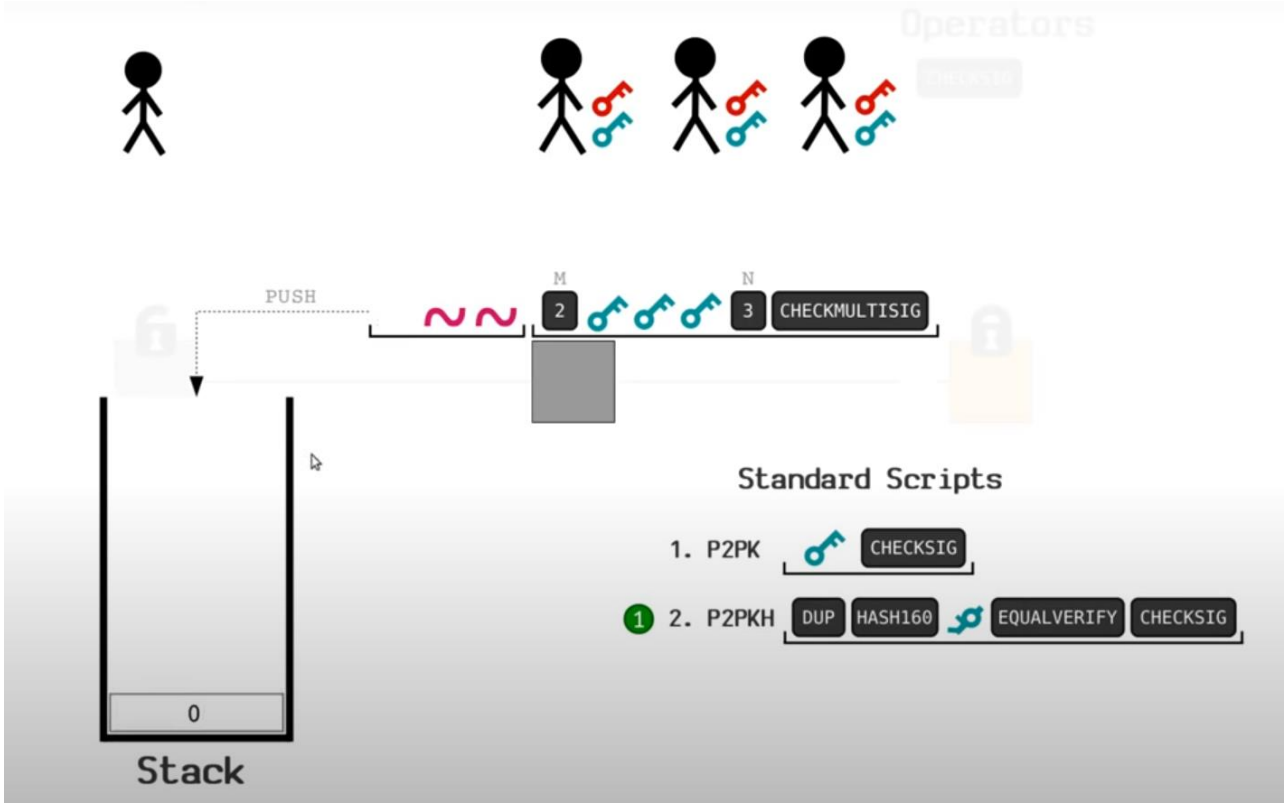
Pay to MultiSig (P2MS)



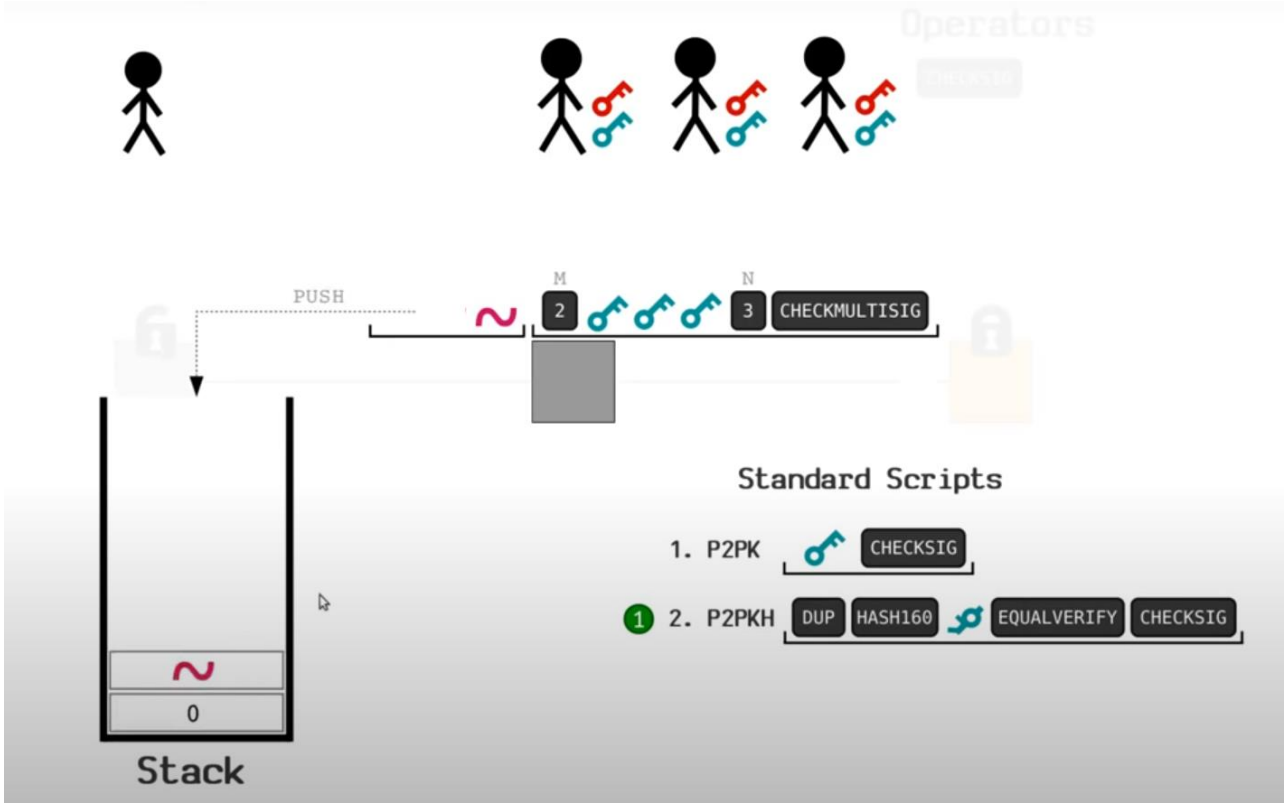
Pay to MultiSig (P2MS)



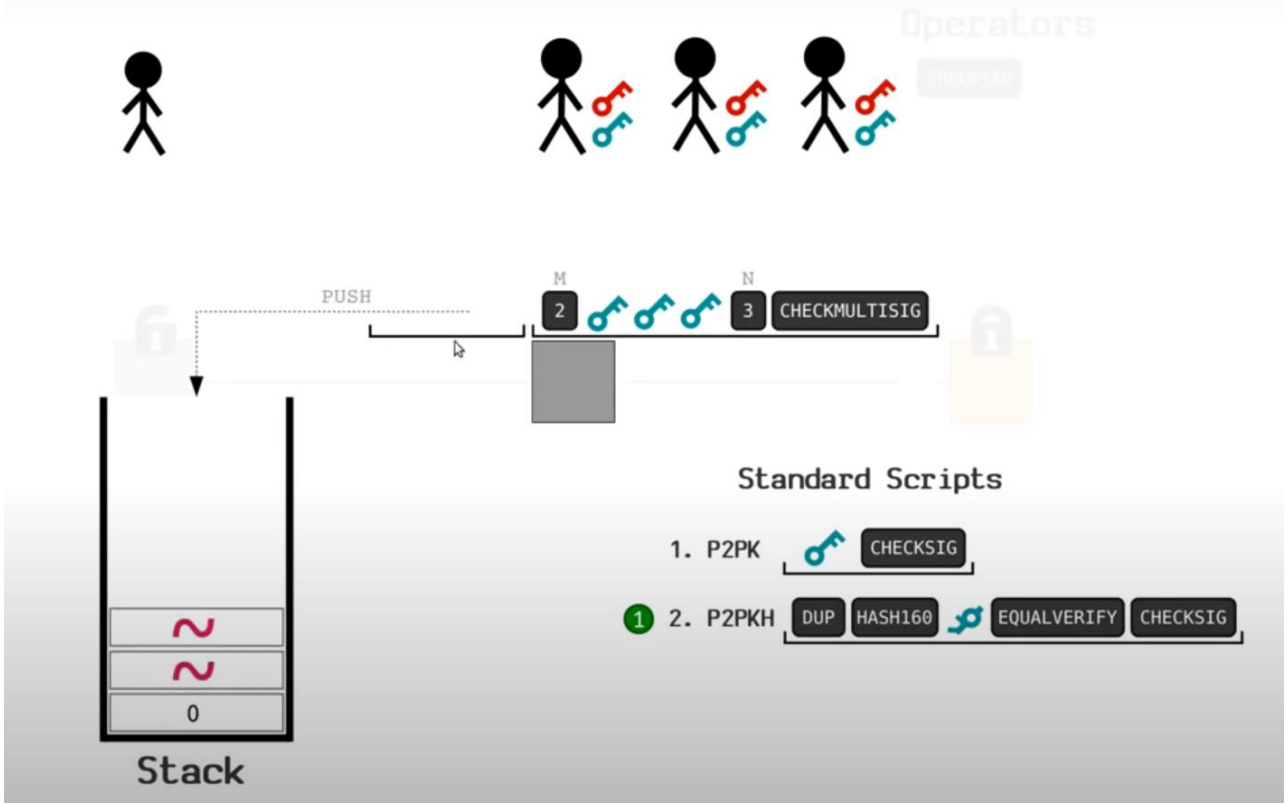
Pay to MultiSig (P2MS)



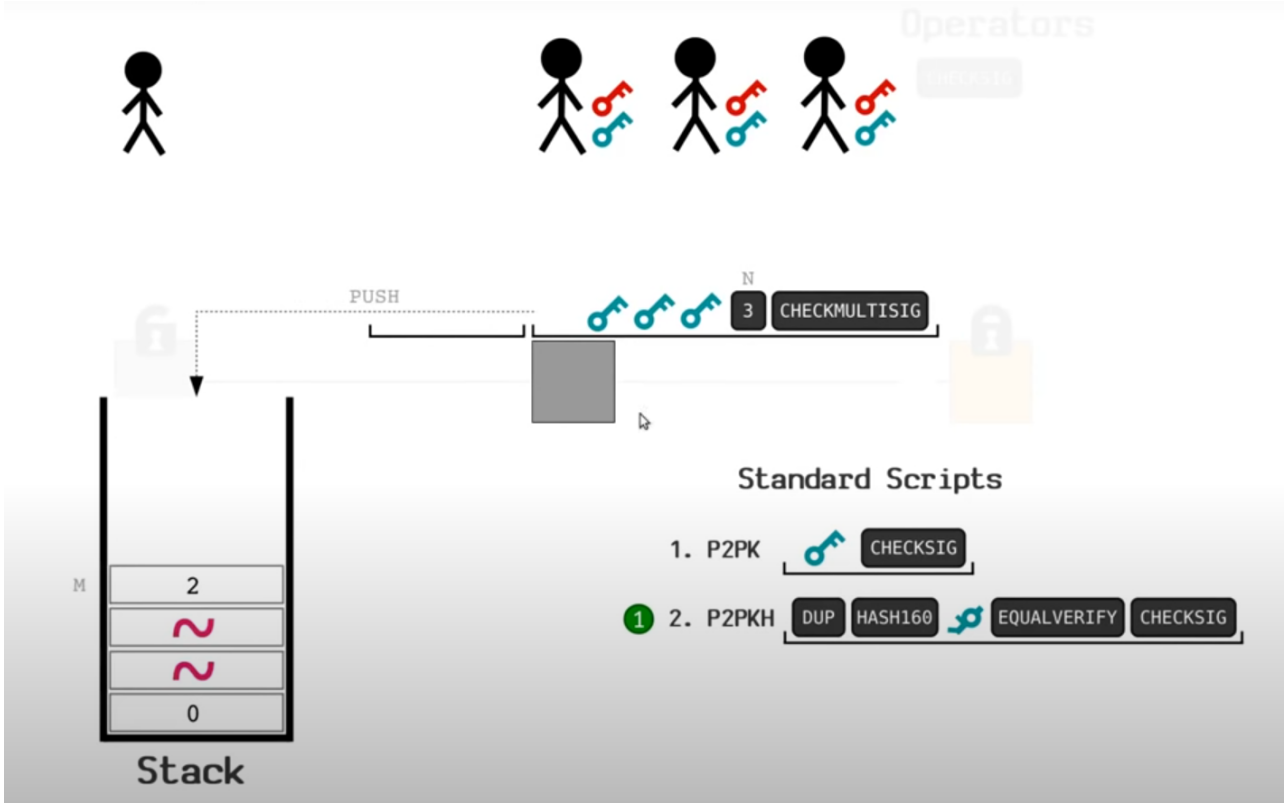
Pay to MultiSig (P2MS)



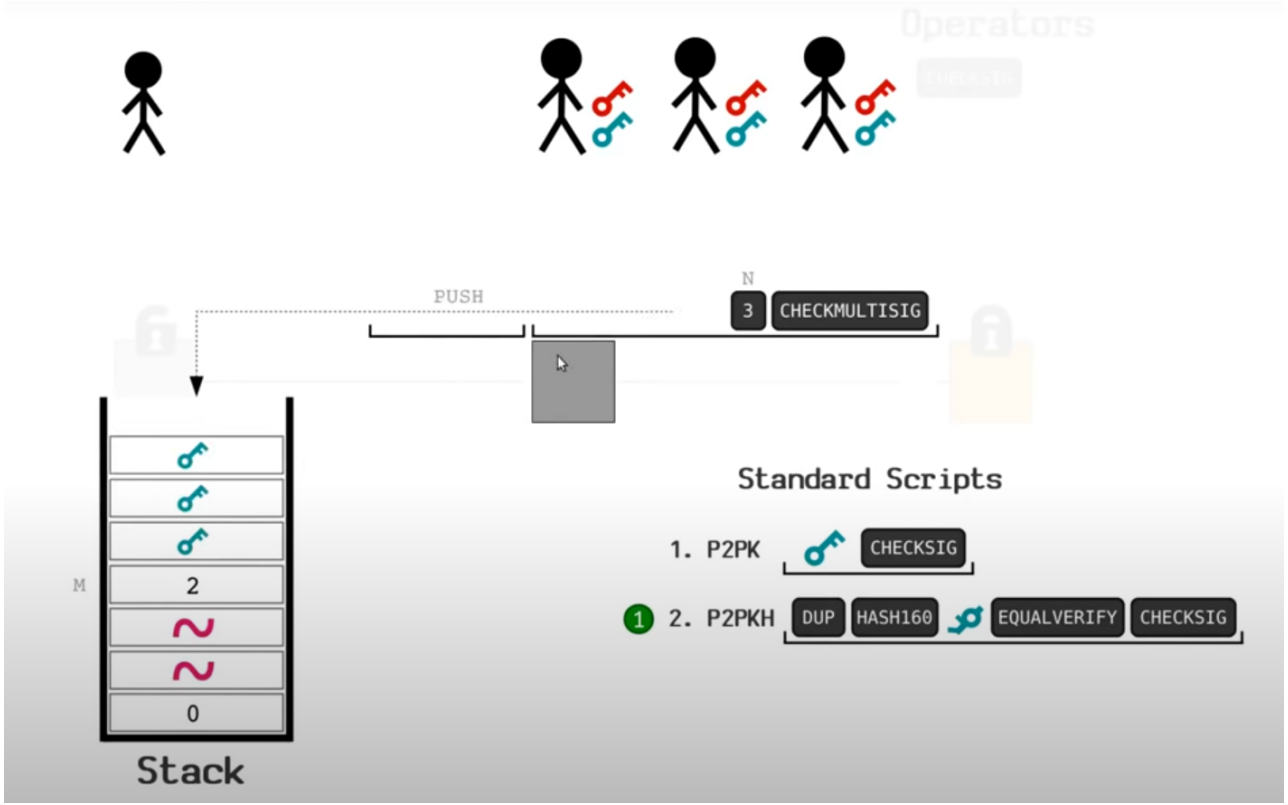
Pay to MultiSig (P2MS)



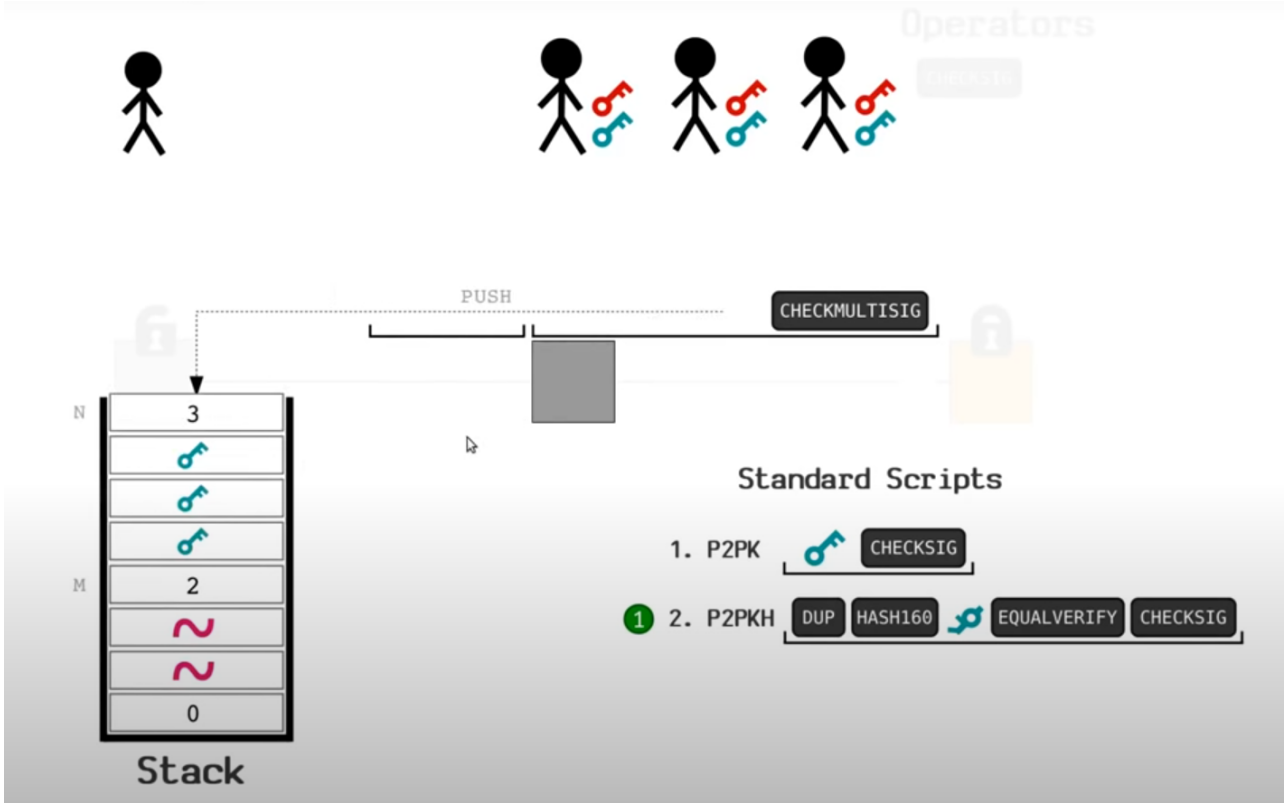
Pay to MultiSig (P2MS)



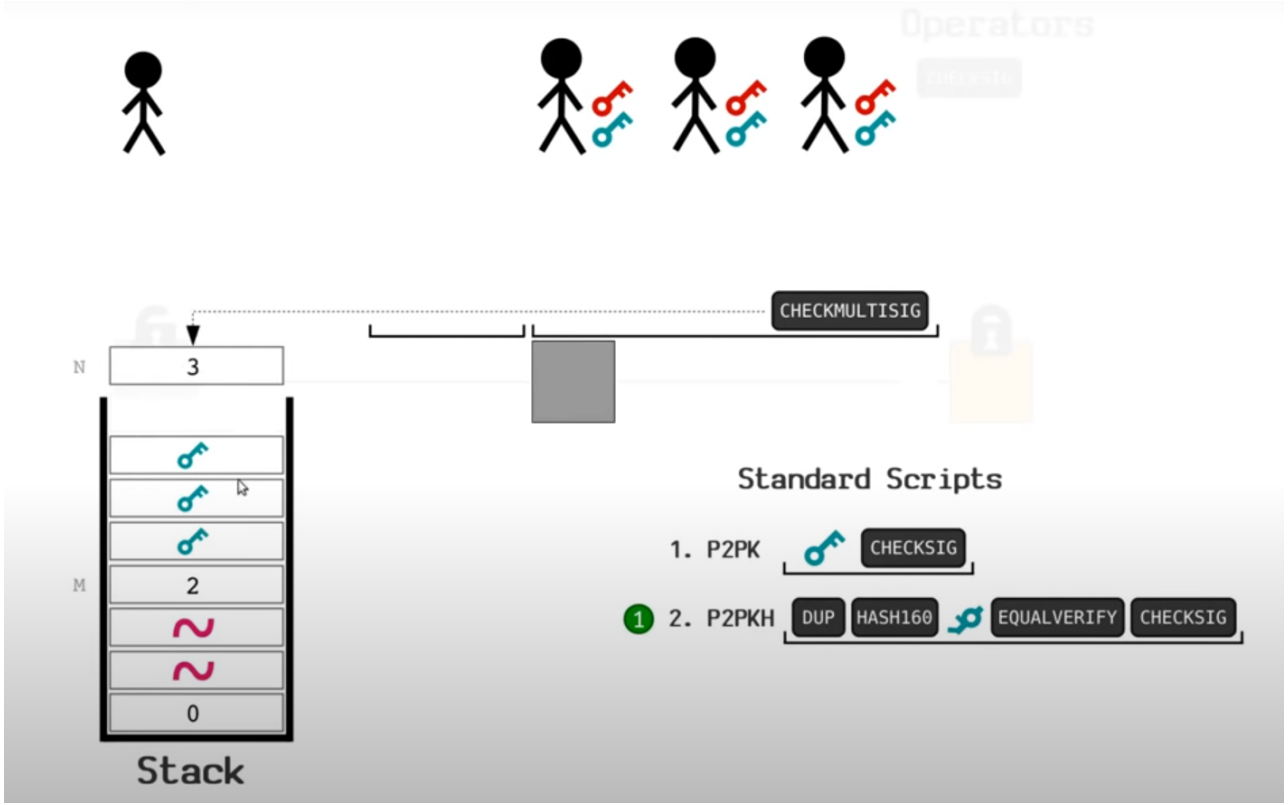
Pay to MultiSig (P2MS)



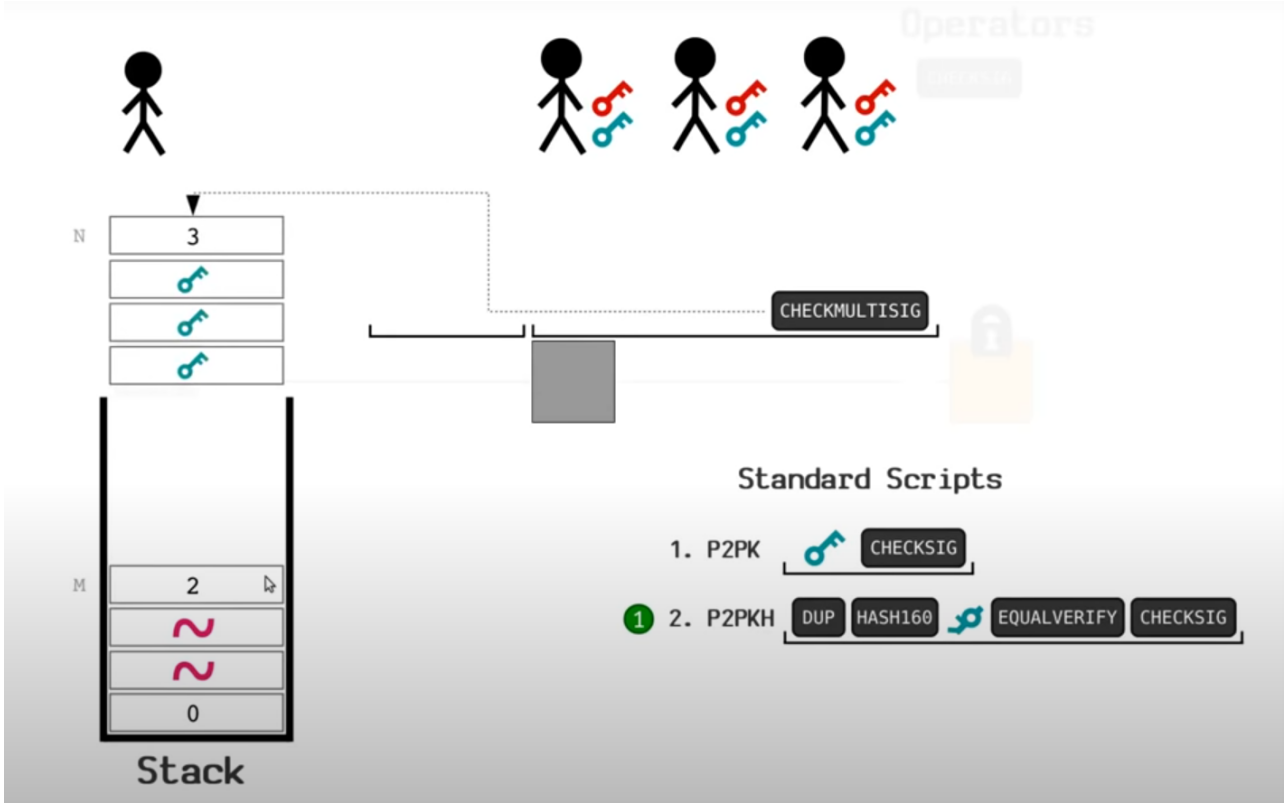
Pay to MultiSig (P2MS)



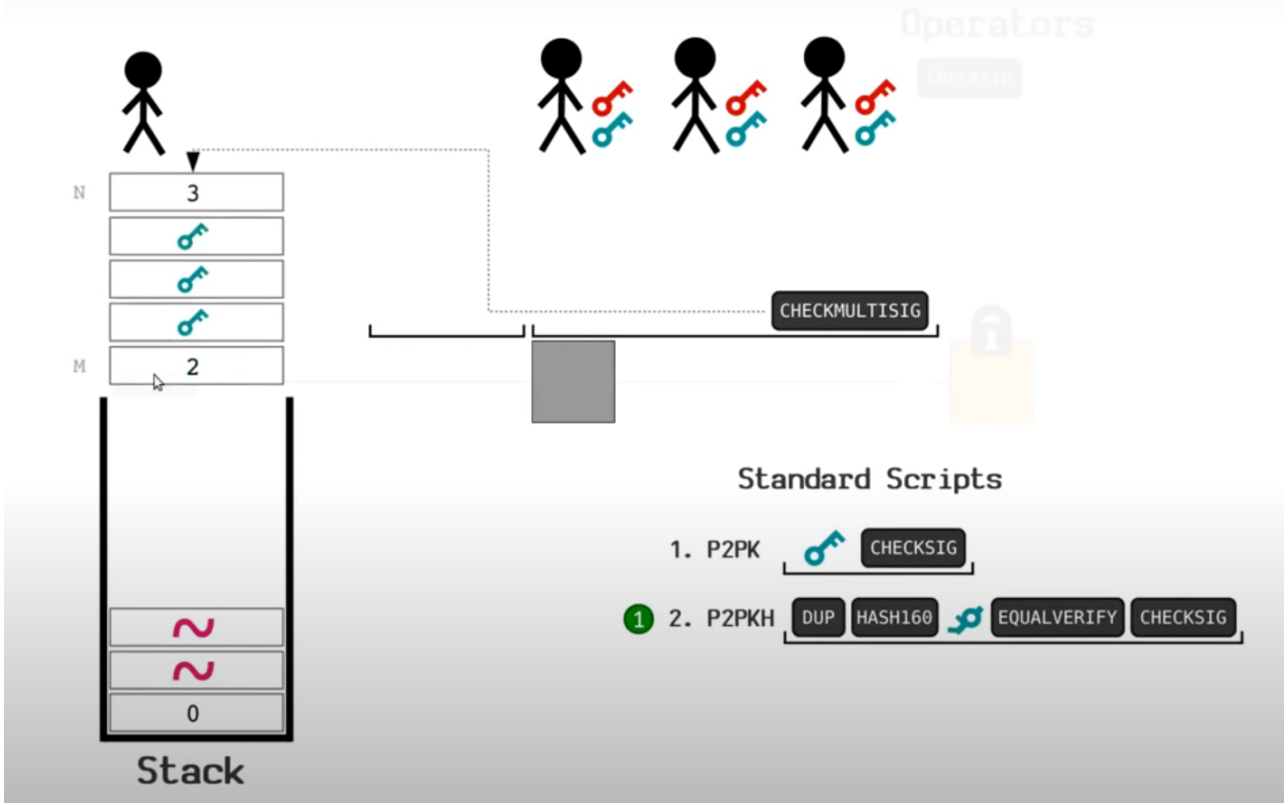
Pay to MultiSig (P2MS)



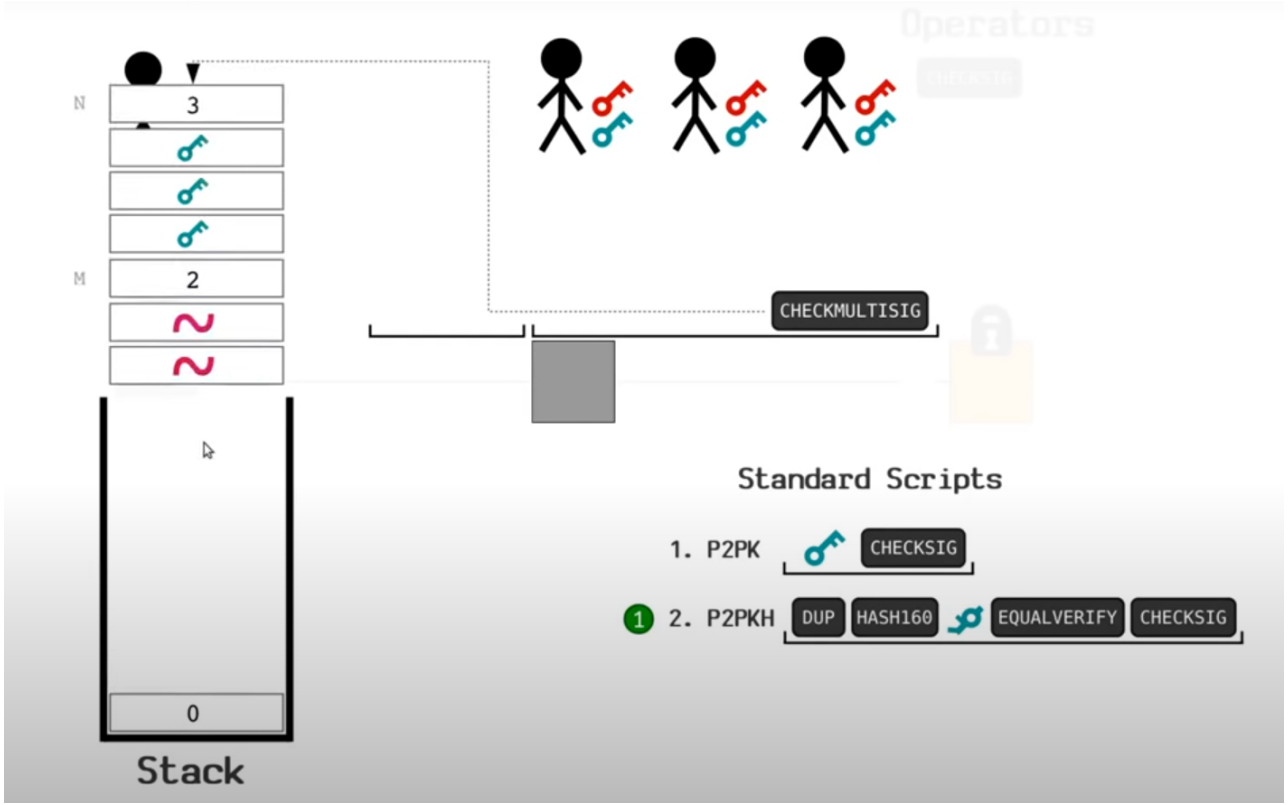
Pay to MultiSig (P2MS)



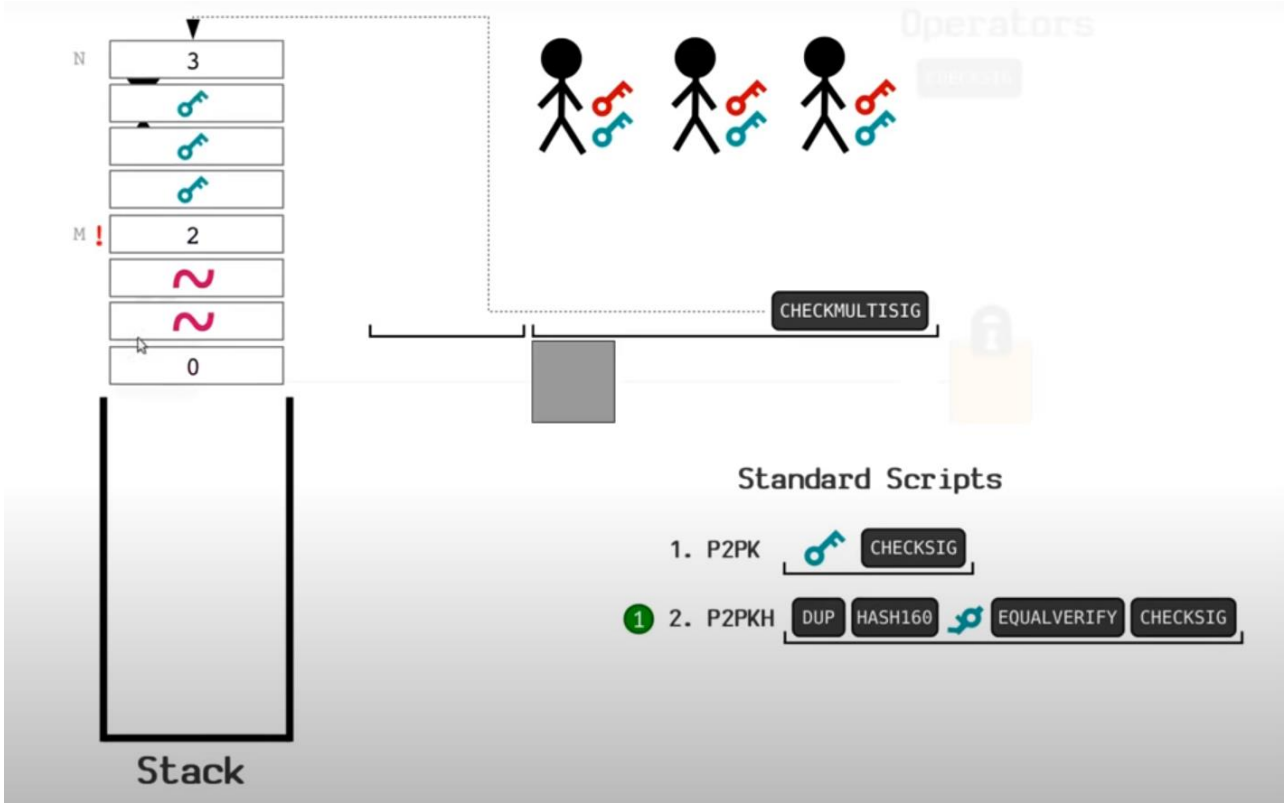
Pay to MultiSig (P2MS)



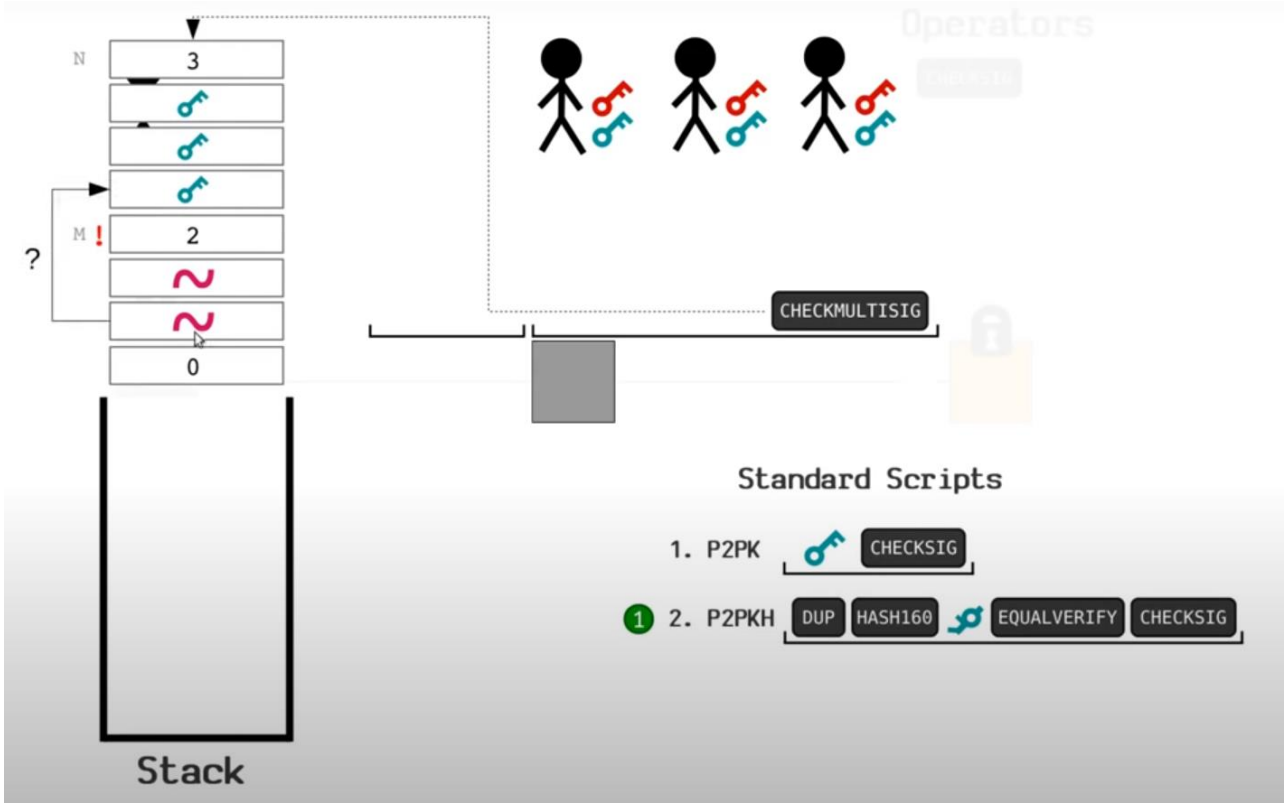
Pay to MultiSig (P2MS)



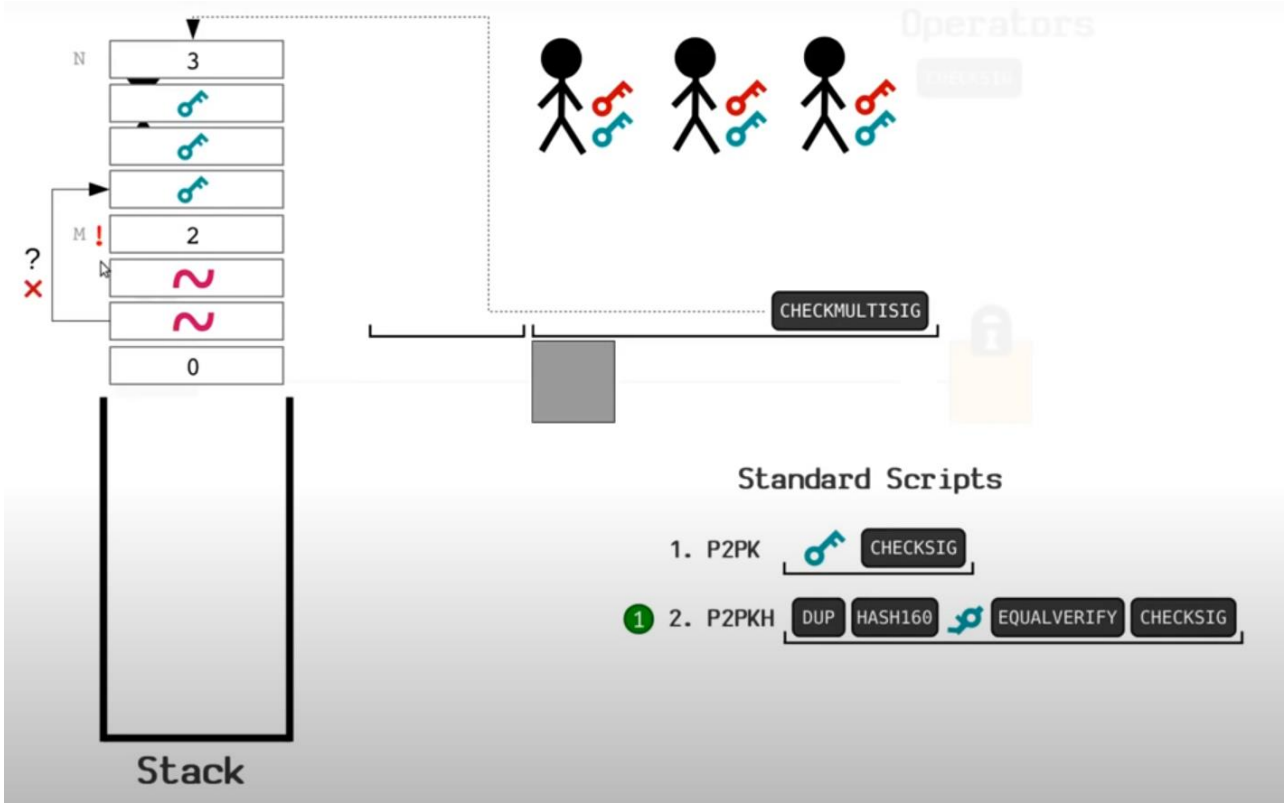
Pay to MultiSig (P2MS)



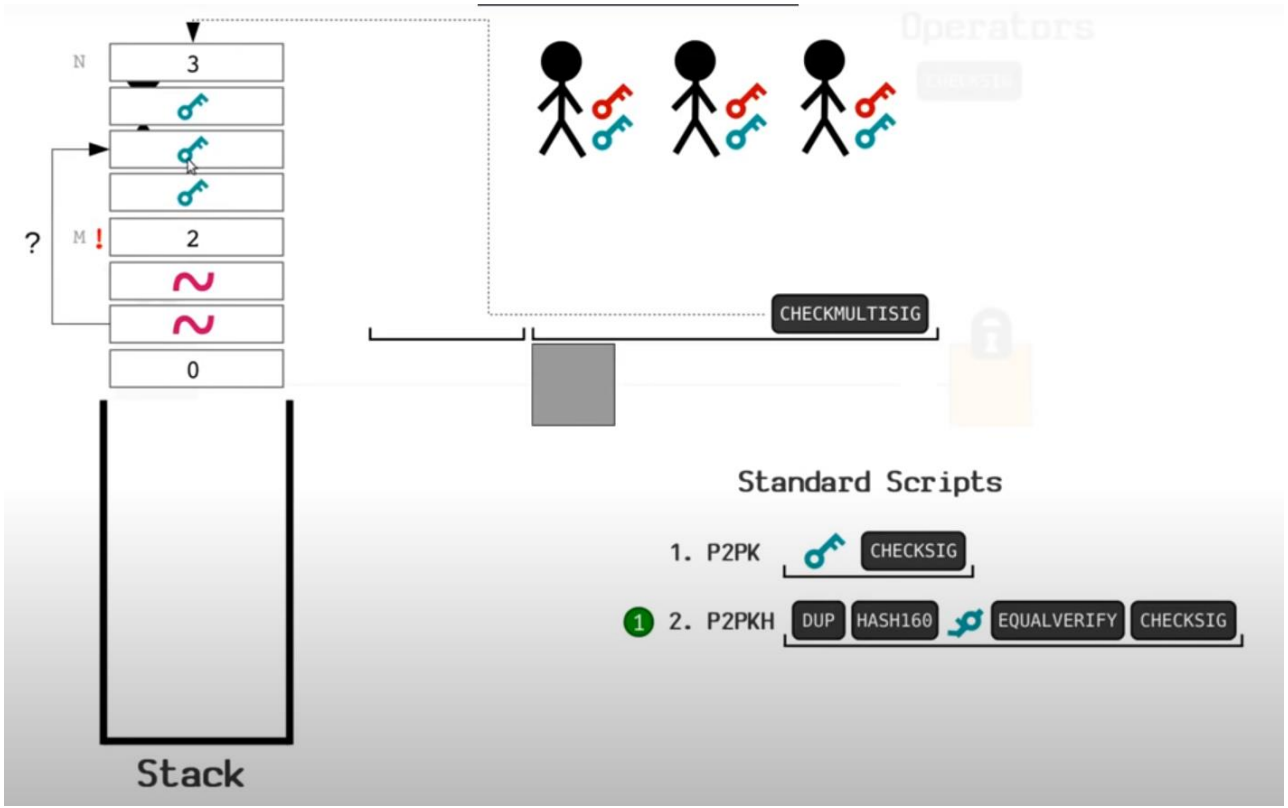
Pay to MultiSig (P2MS)



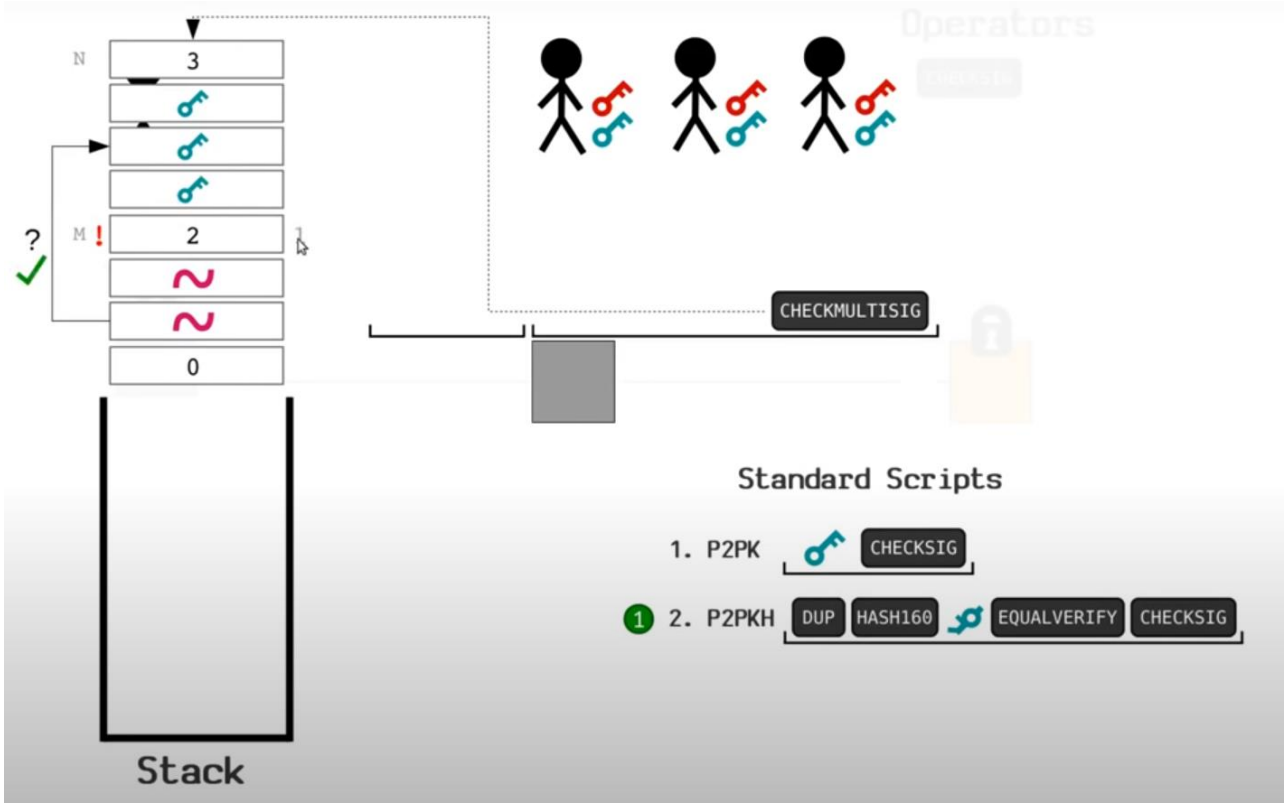
Pay to MultiSig (P2MS)



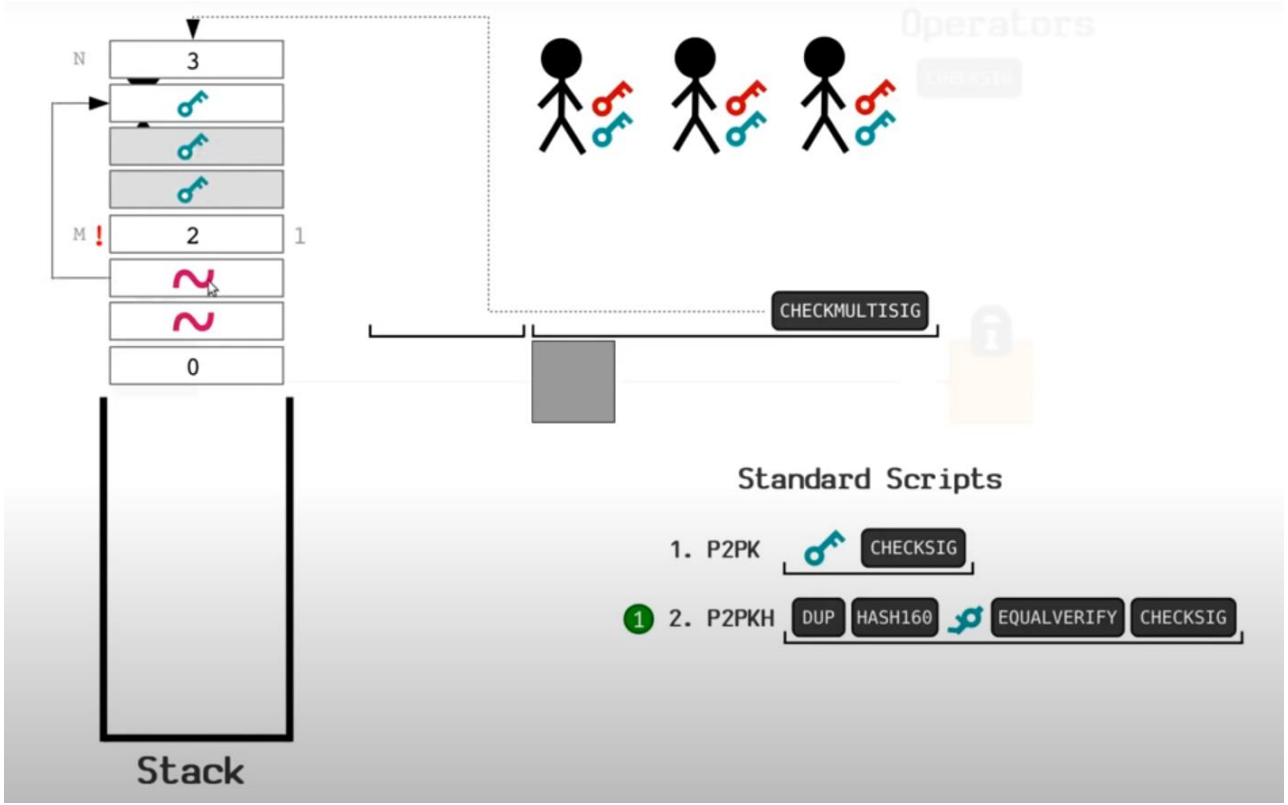
Pay to MultiSig (P2MS)



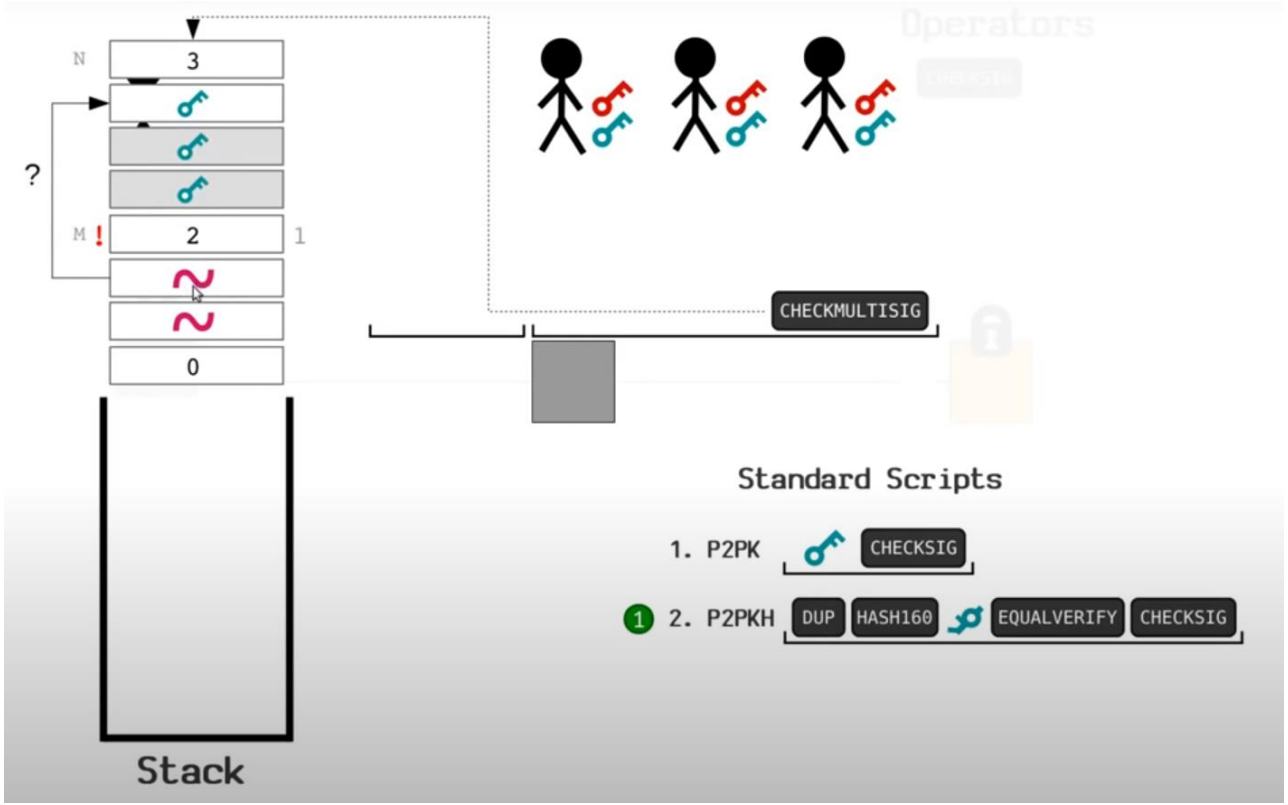
Pay to MultiSig (P2MS)



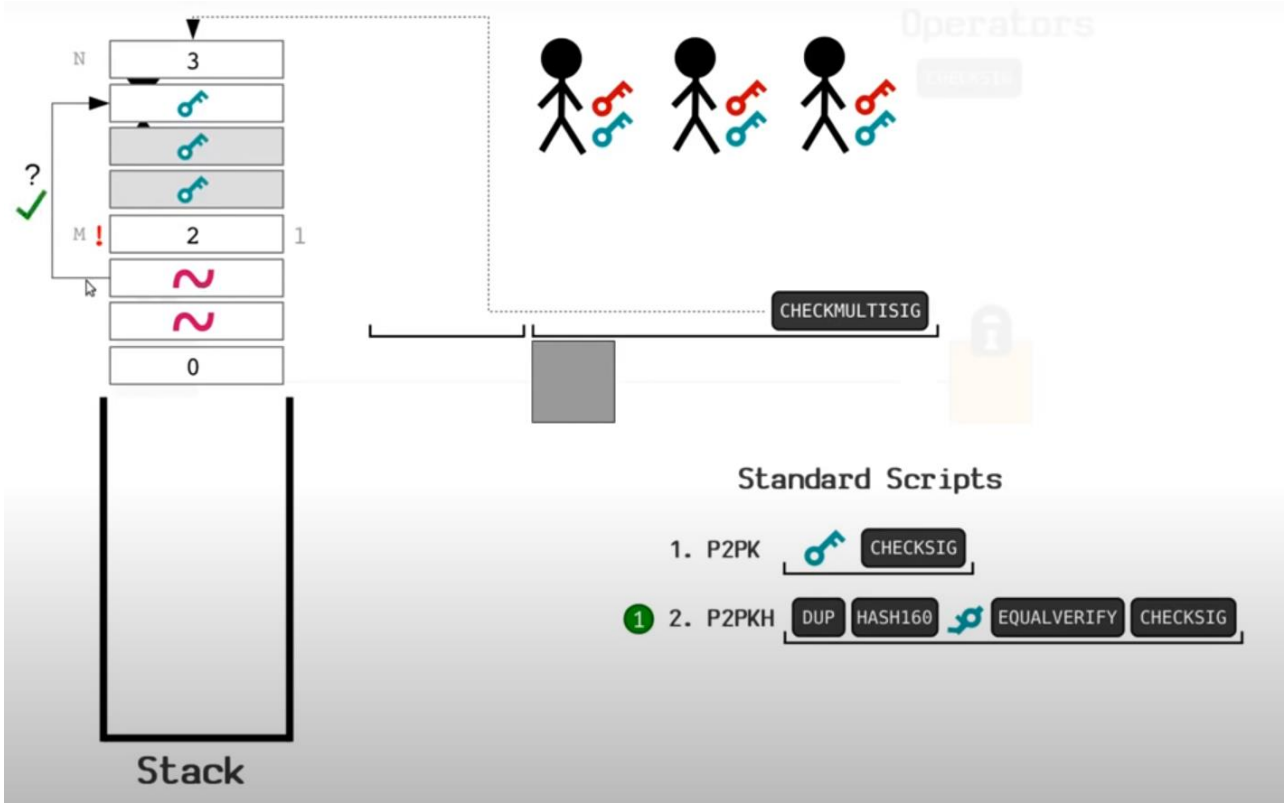
Pay to MultiSig (P2MS)



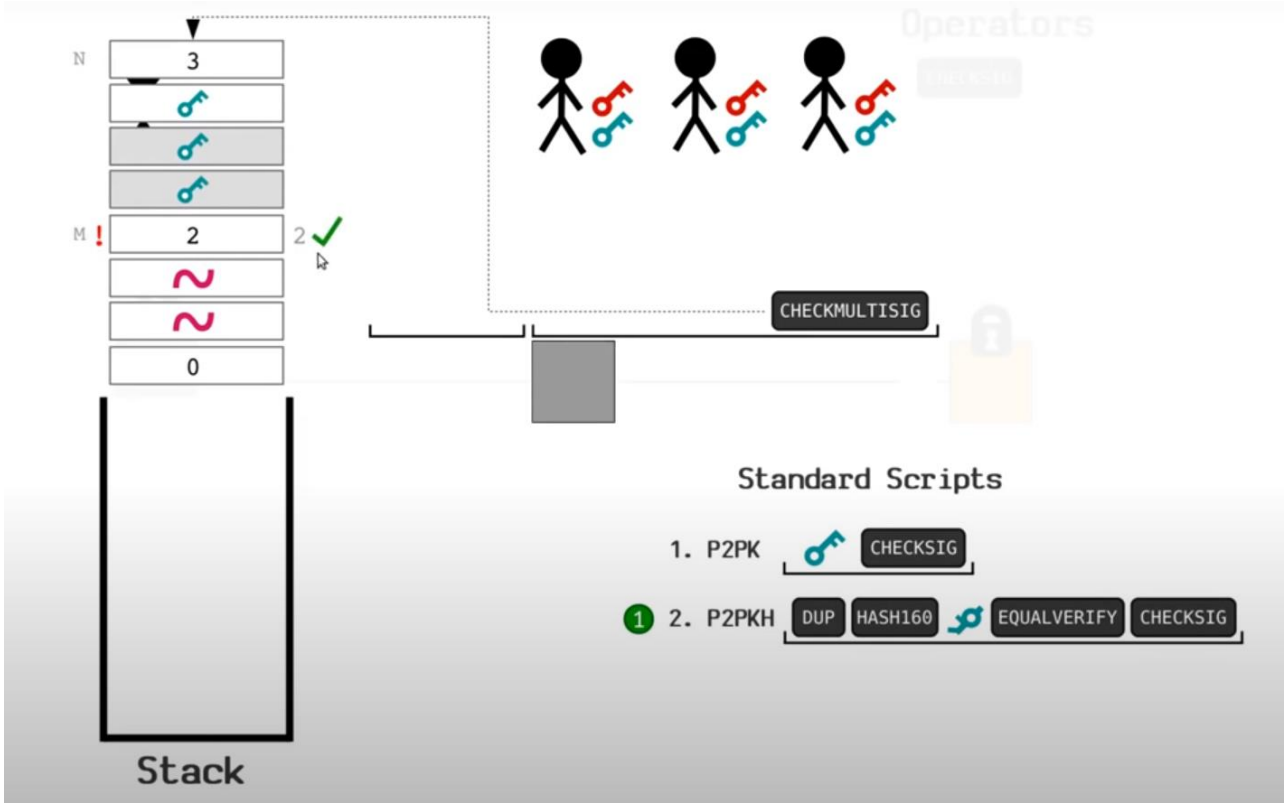
Pay to MultiSig (P2MS)



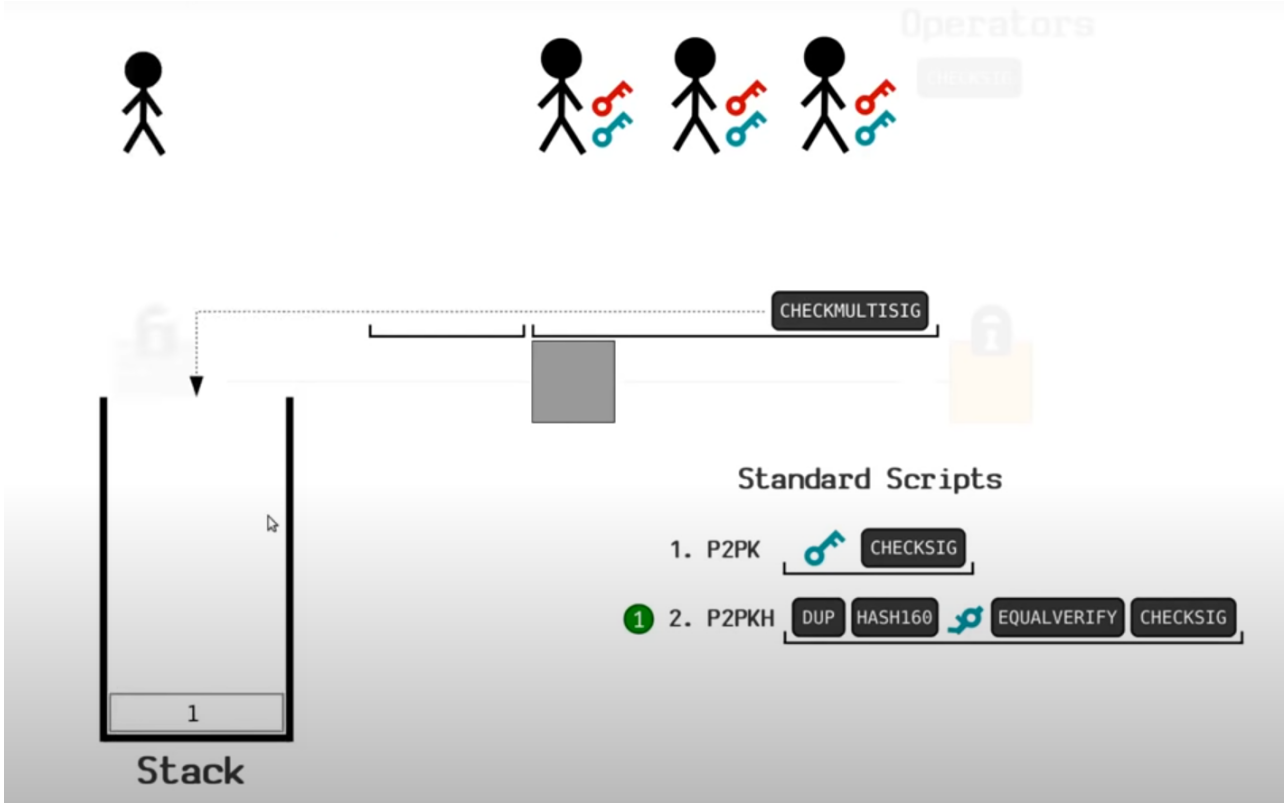
Pay to MultiSig (P2MS)



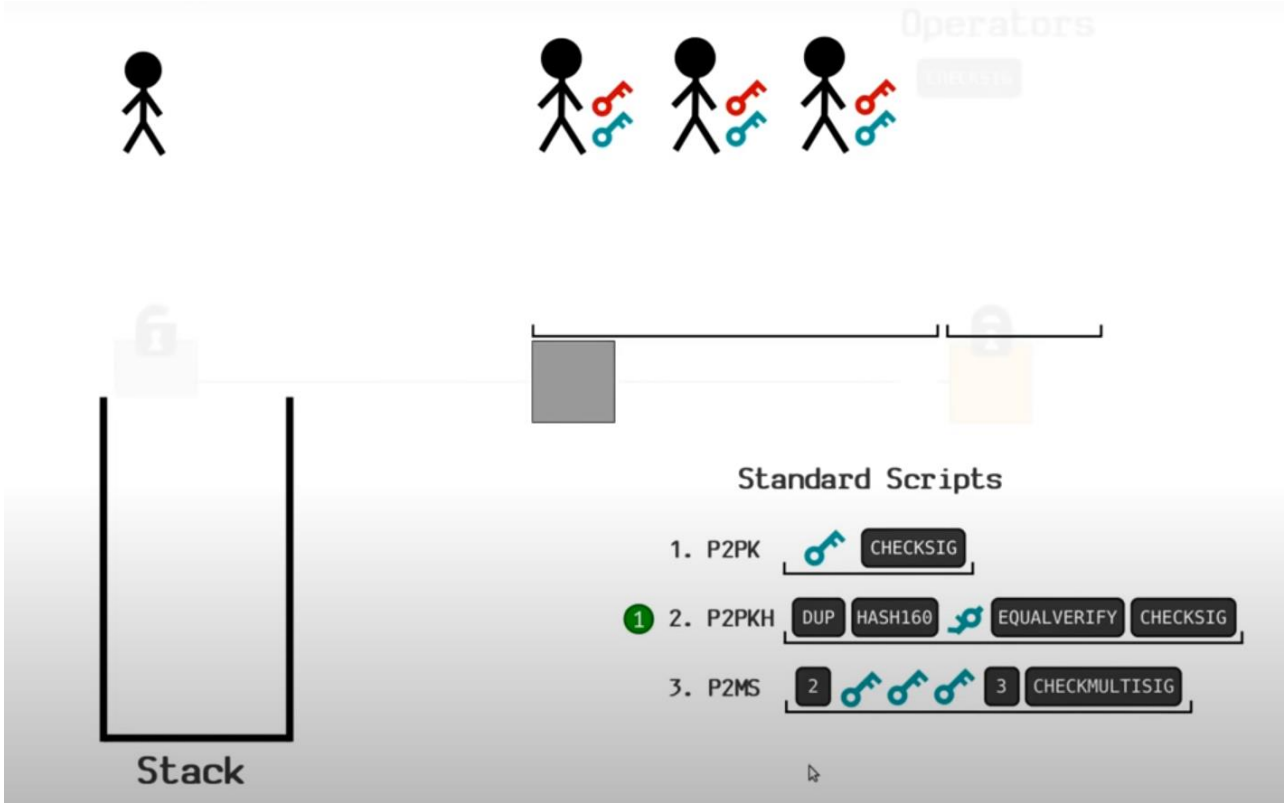
Pay to MultiSig (P2MS)



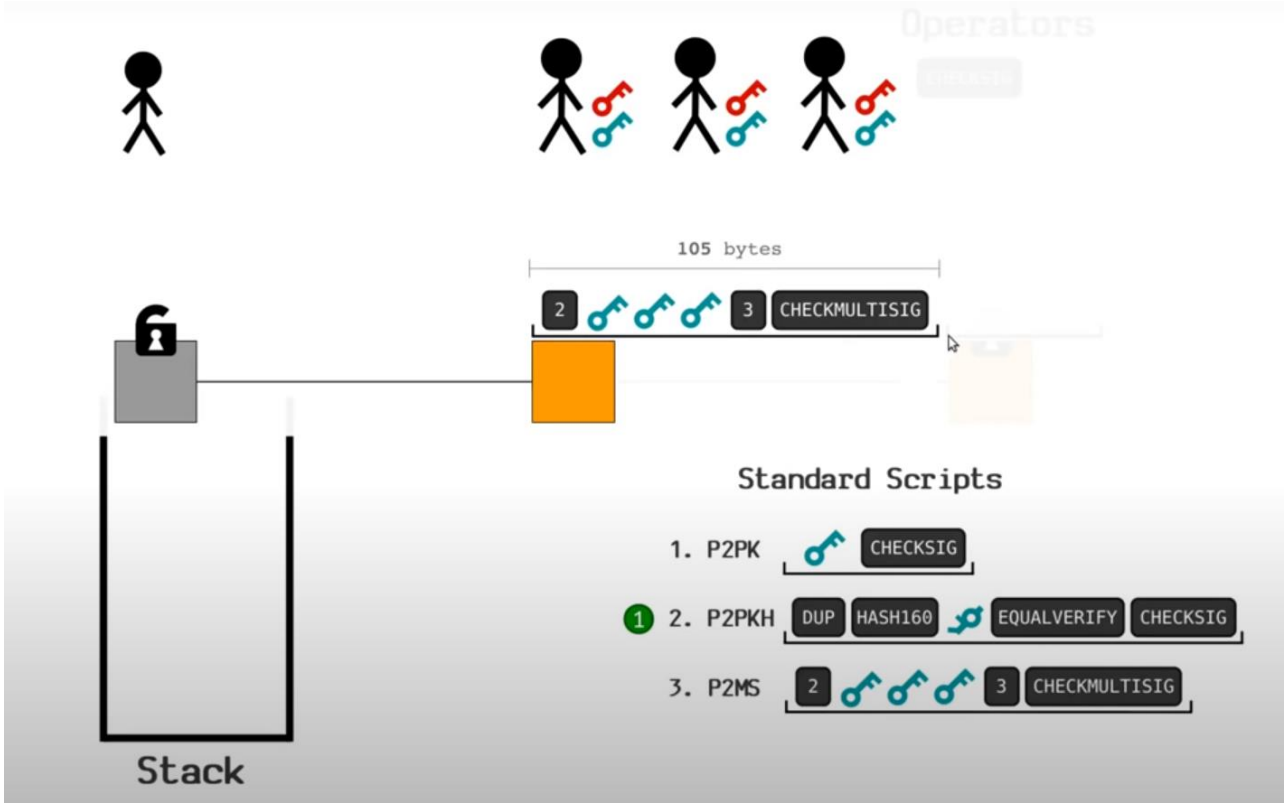
Pay to MultiSig (P2MS)



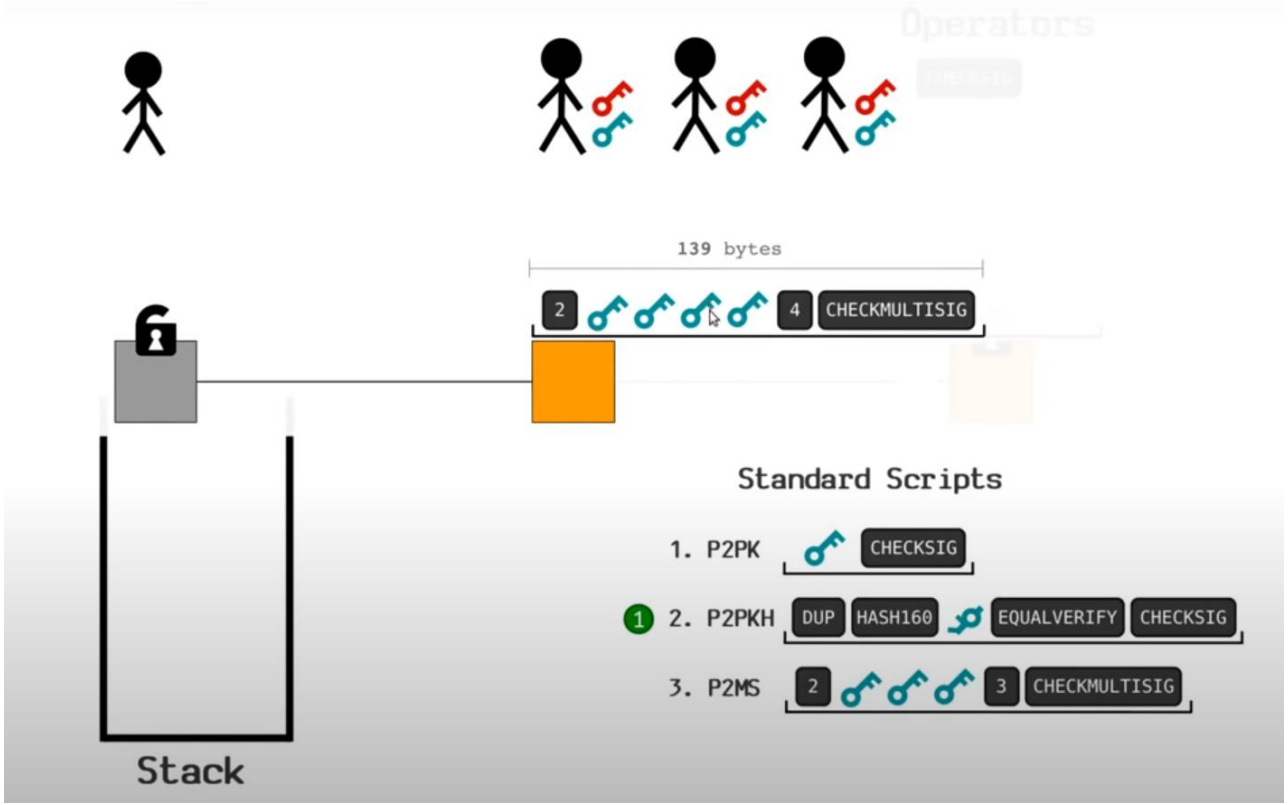
Pay to MultiSig (P2MS)



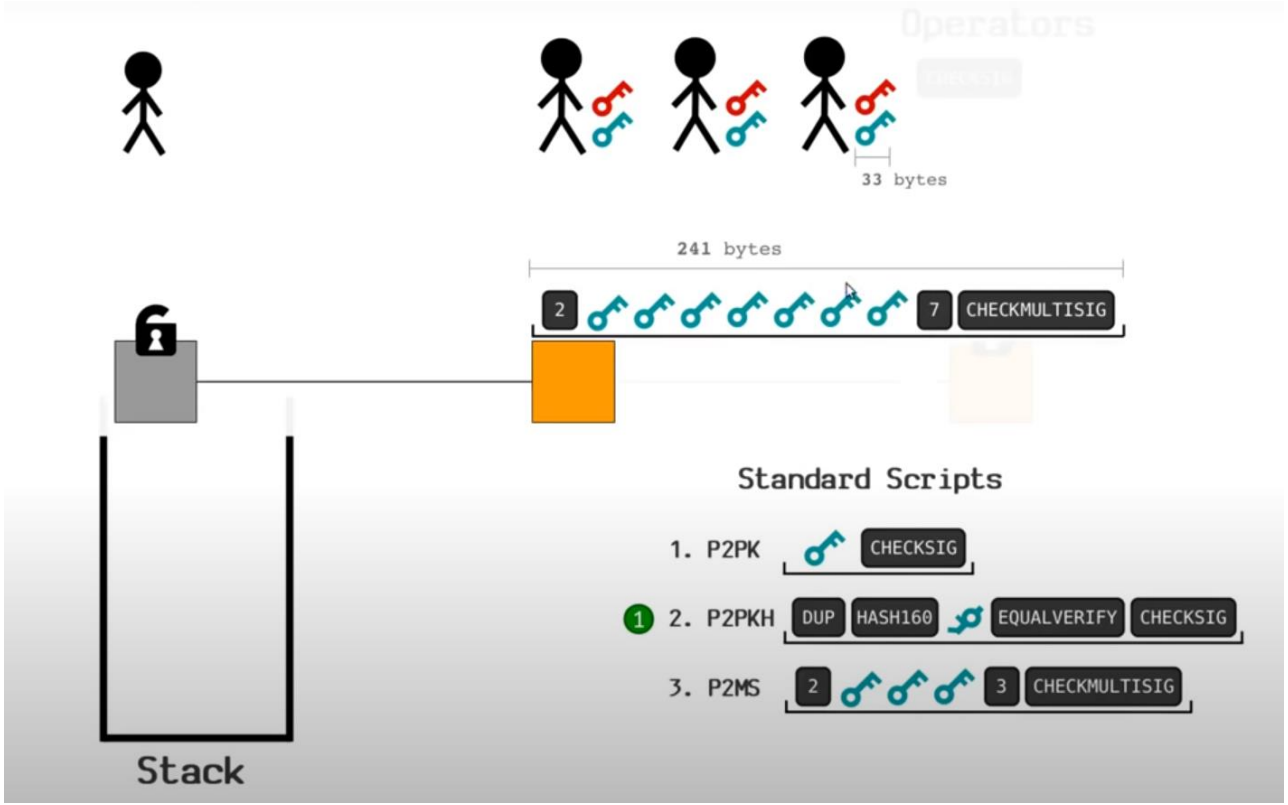
Problem!


105 bytes
2
3
CHECKMULTISIG
Standard Scripts
1. P2PK
CHECKSIG
1
2. P2PKH
DUP
HASH160
EQUALVERIFY
CHECKSIG
3. P2MS
2
3
CHECKMULTISIG
Stack

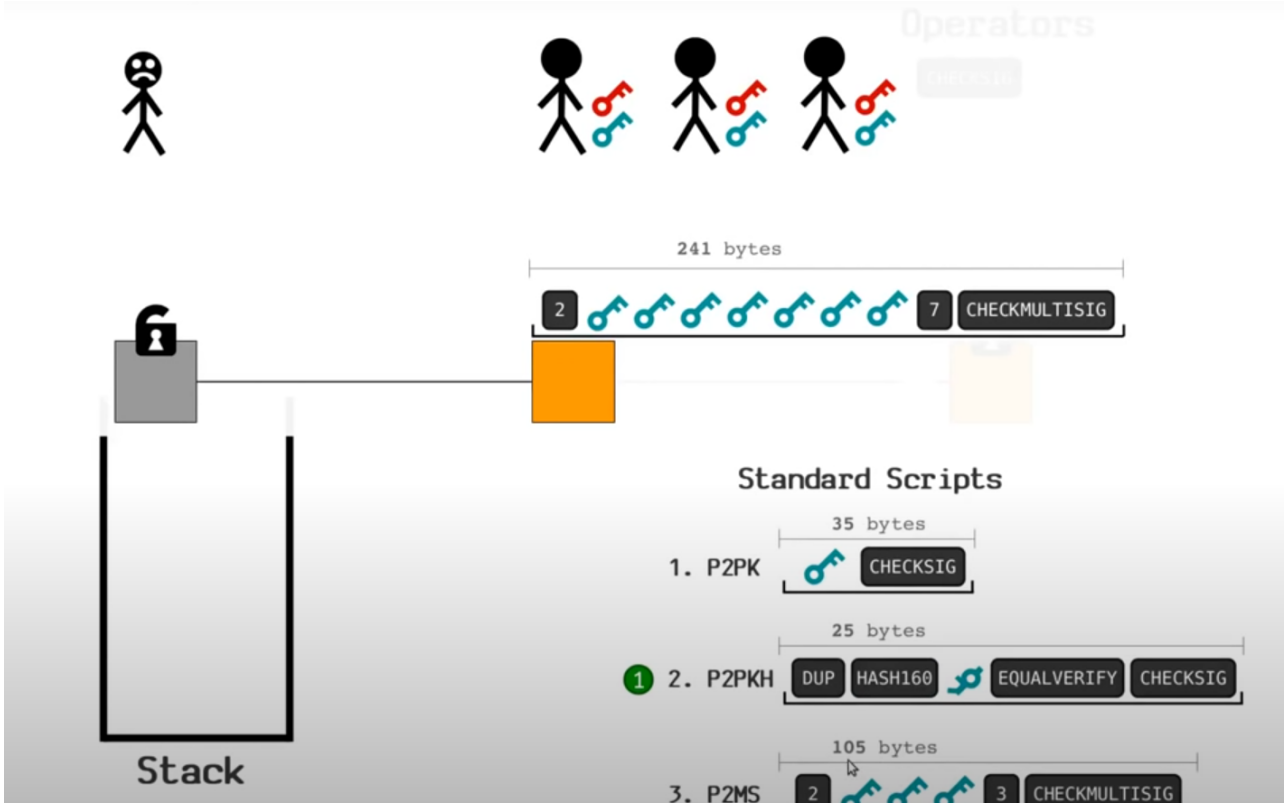
Problem!


139 bytes
2
4
CHECKMULTISIG
Standard Scripts
1. P2PK
CHECKSIG
1
2. P2PKH
DUP
HASH160
EQUALVERIFY
CHECKSIG
3. P2MS
2
3
CHECKMULTISIG
Stack

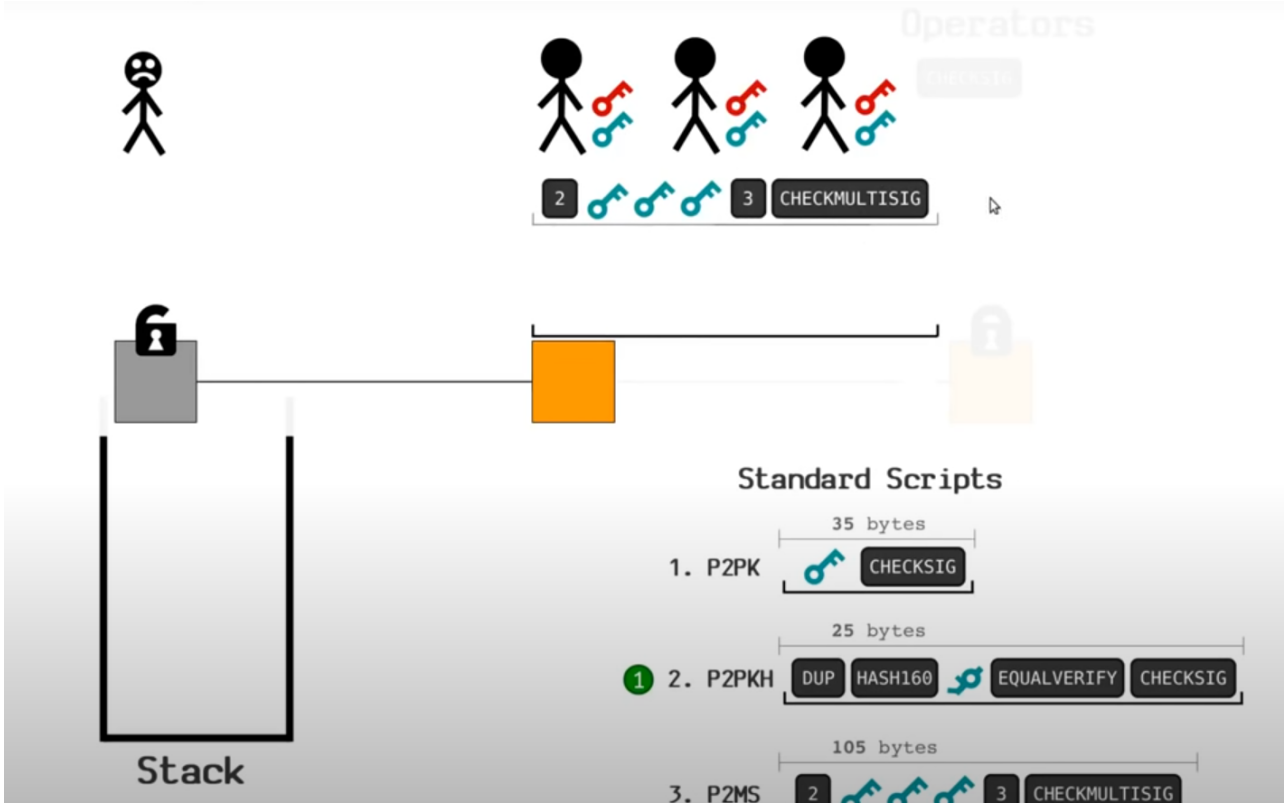
Problem!


33 bytes
241 bytes
2
7
CHECKMULTISIG
Standard Scripts
1. P2PK
CHECKSIG
1
2. P2PKH
DUP
HASH160
EQUALVERIFY
CHECKSIG
3. P2MS
2
3
CHECKMULTISIG
Stack

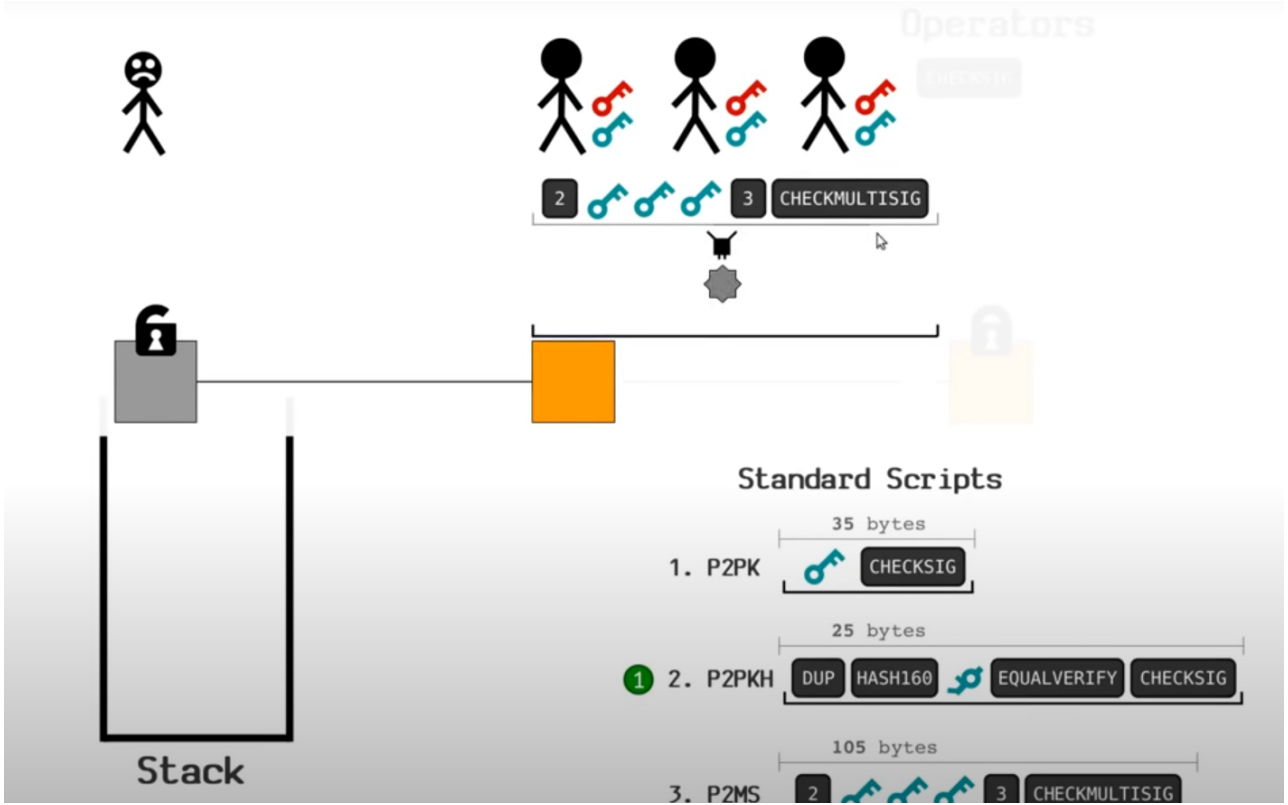
Problem!


241 bytes
2
7
CHECKMULTISIG
Stack
Standard Scripts
35 bytes
1. P2PK
CHECKSIG
25 bytes
1
2. P2PKH
DUP
HASH160
EQUALVERIFY
CHECKSIG
105 bytes
3. P2MS
2
3
CHECKMULTISIG

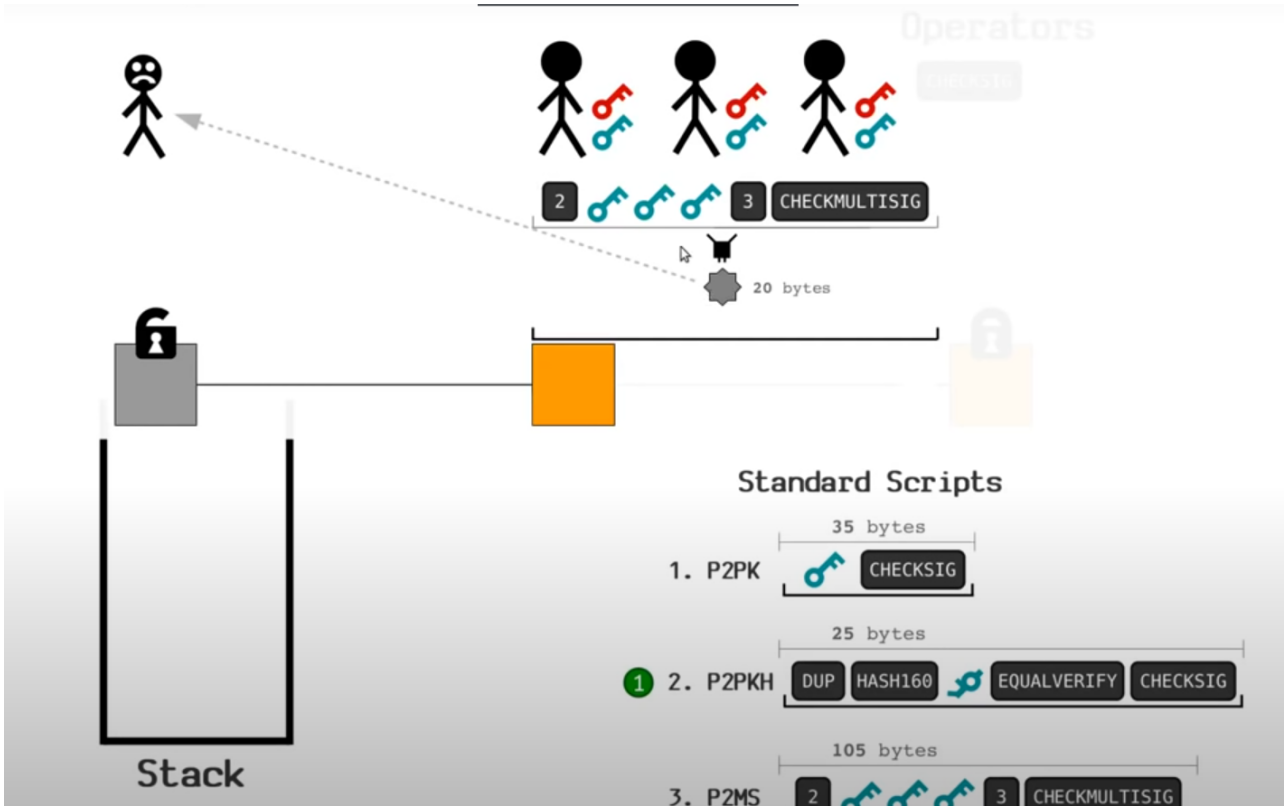
Pay to Script Hash (P2SH)



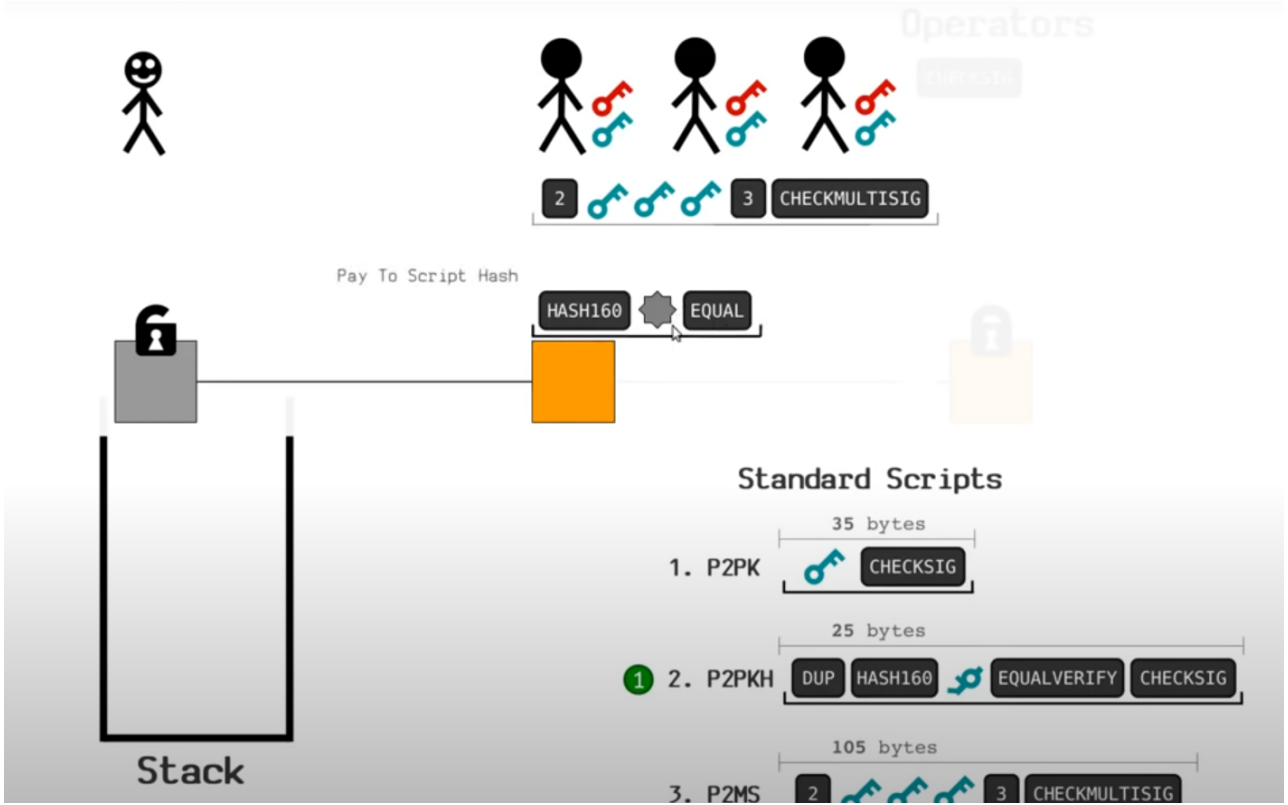
Pay to Script Hash (P2SH)



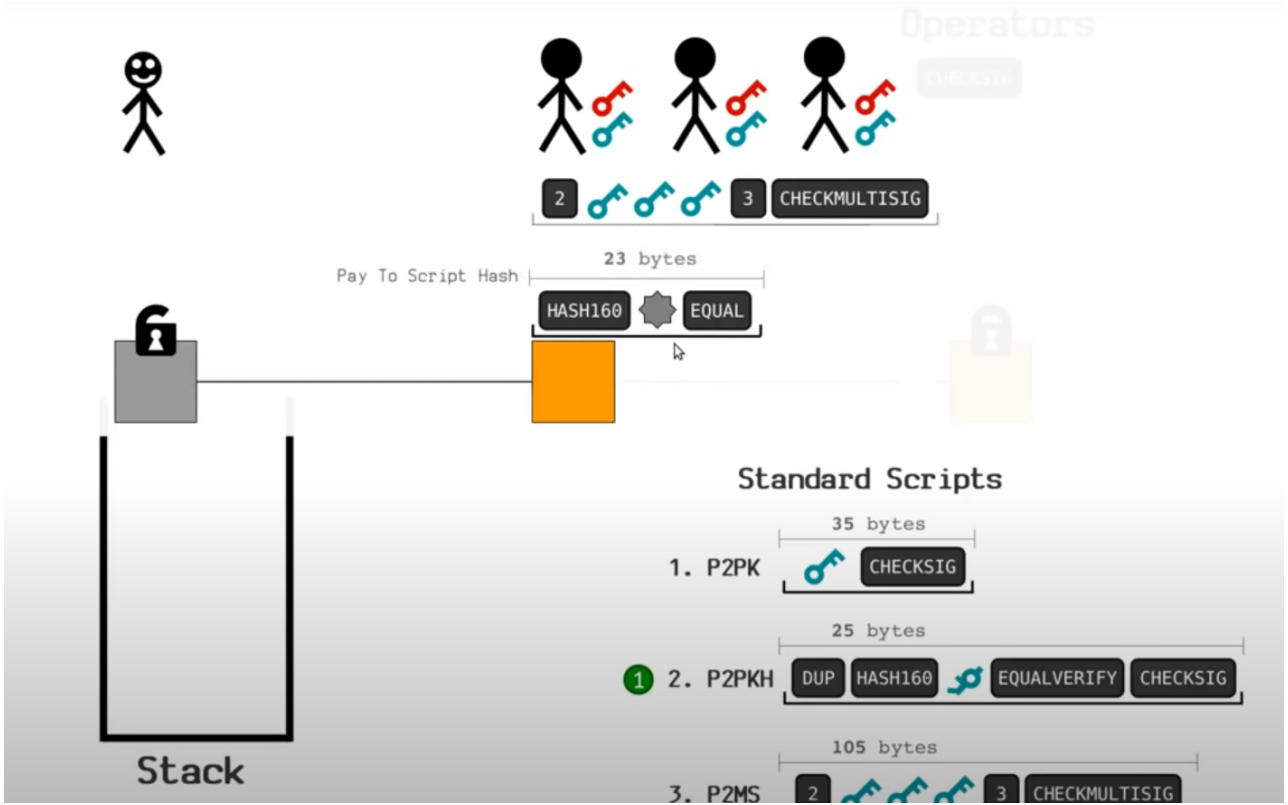
Pay to Script Hash (P2SH)



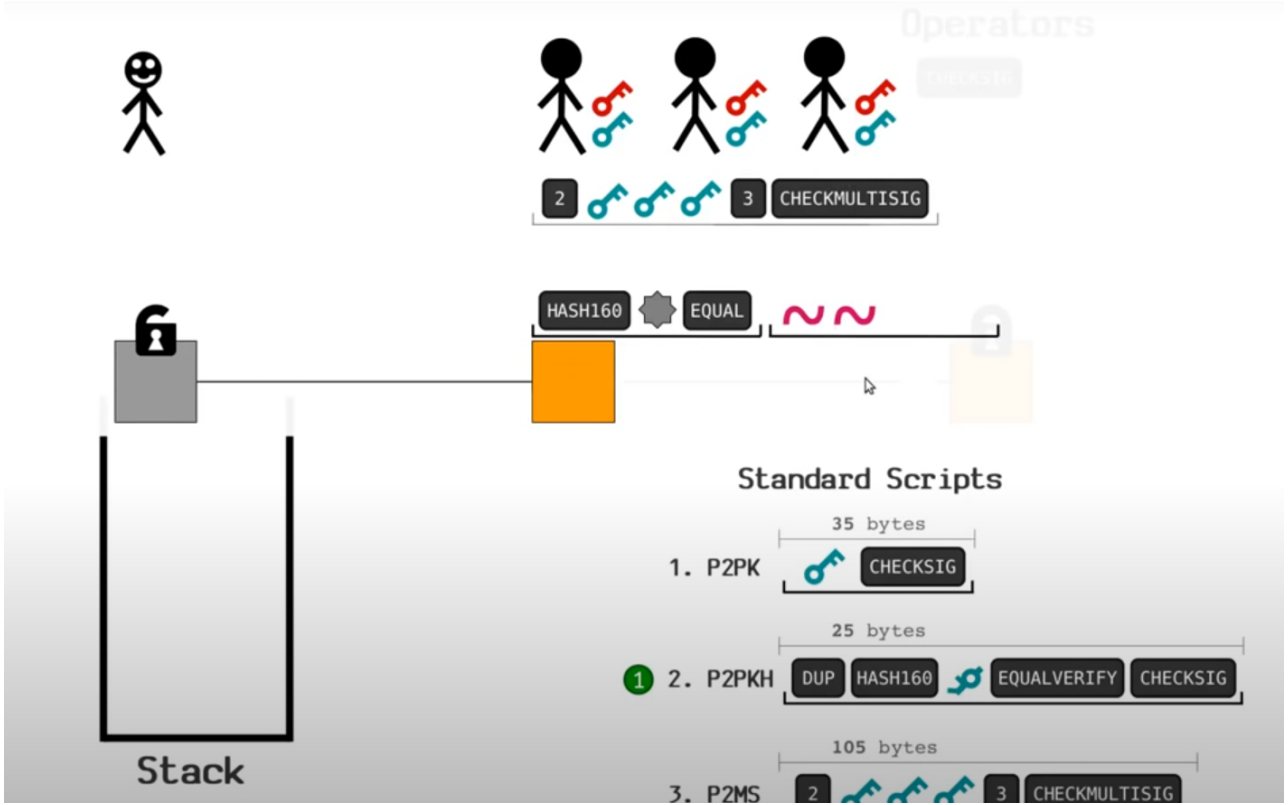
Pay to Script Hash (P2SH)



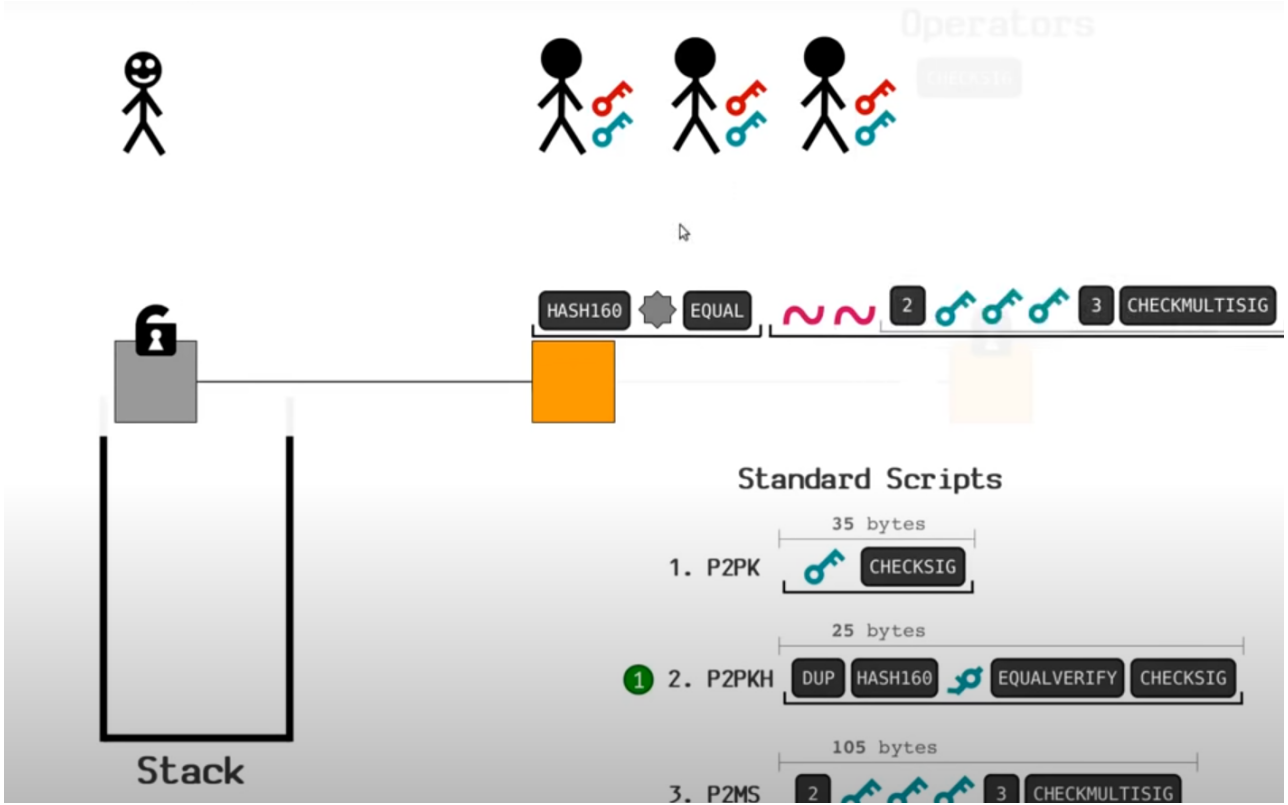
Pay to Script Hash (P2SH)



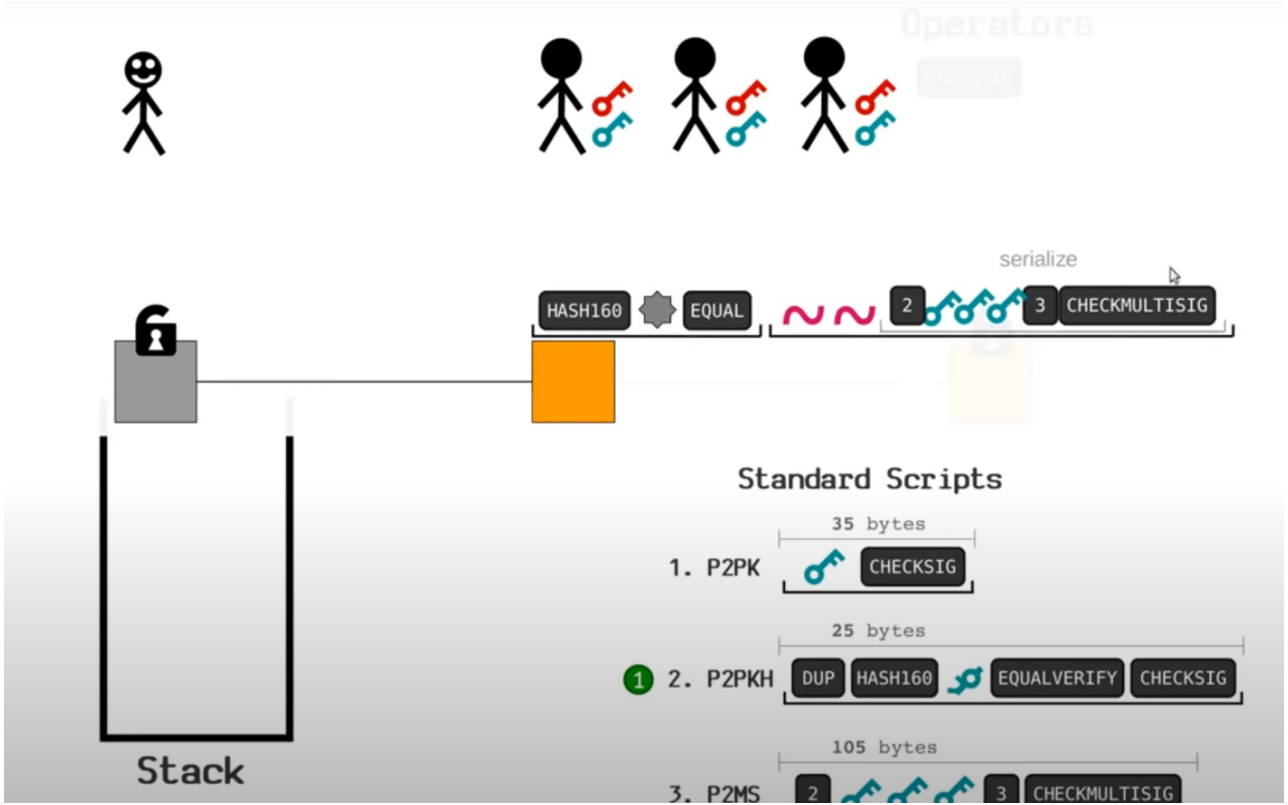
Pay to Script Hash (P2SH)



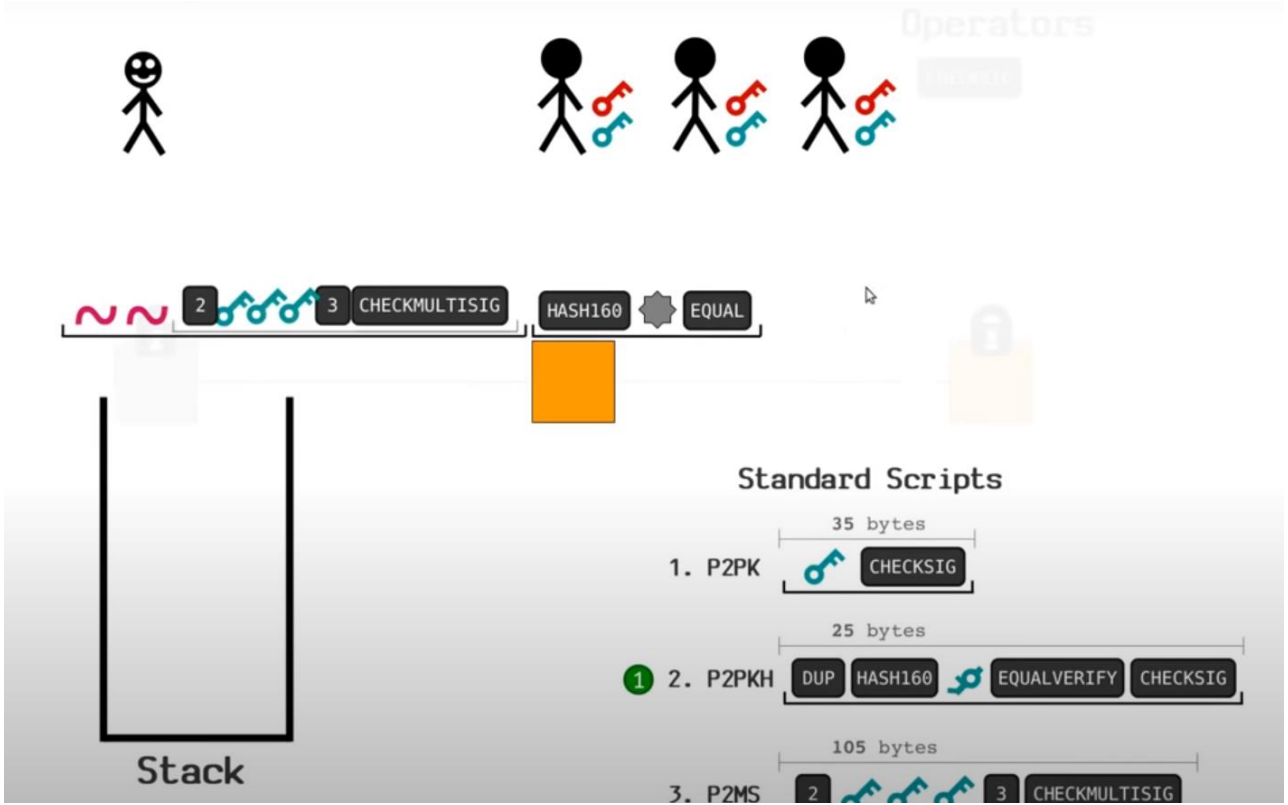
Pay to Script Hash (P2SH)



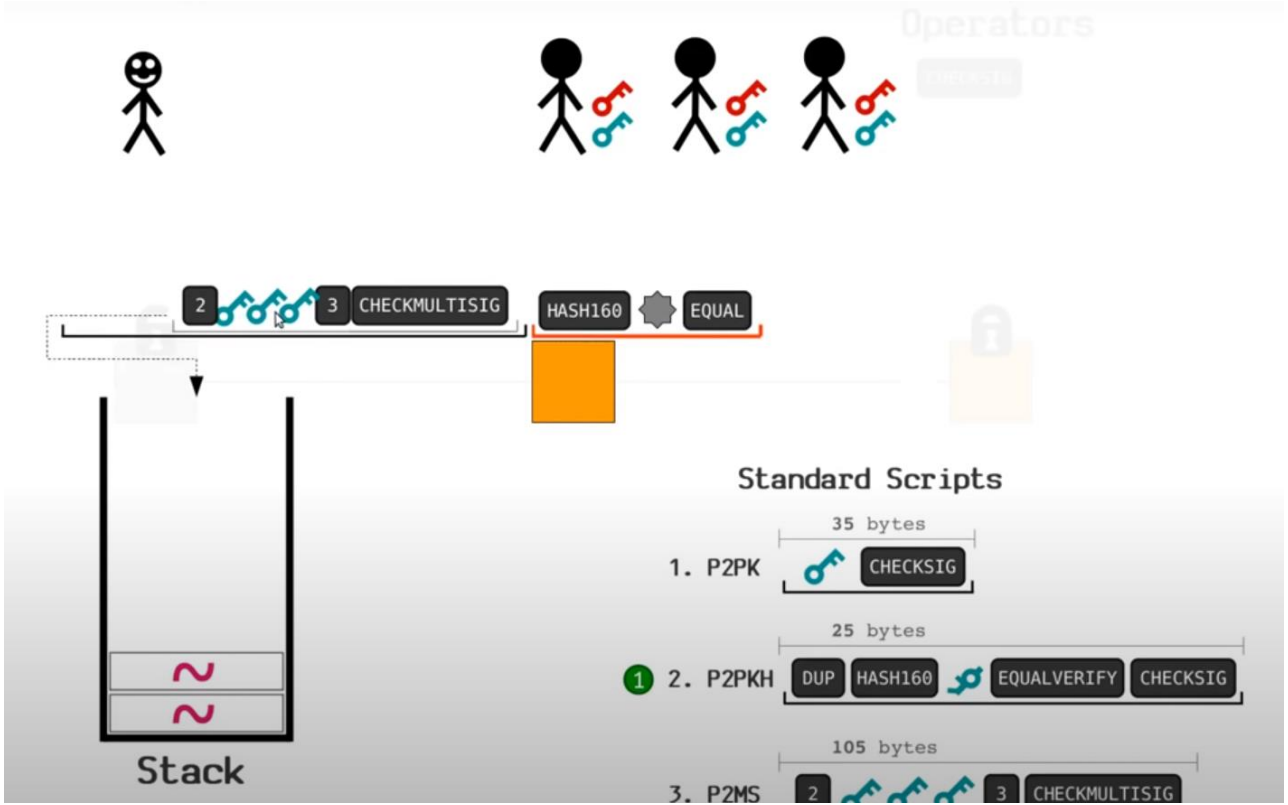
Pay to Script Hash (P2SH)



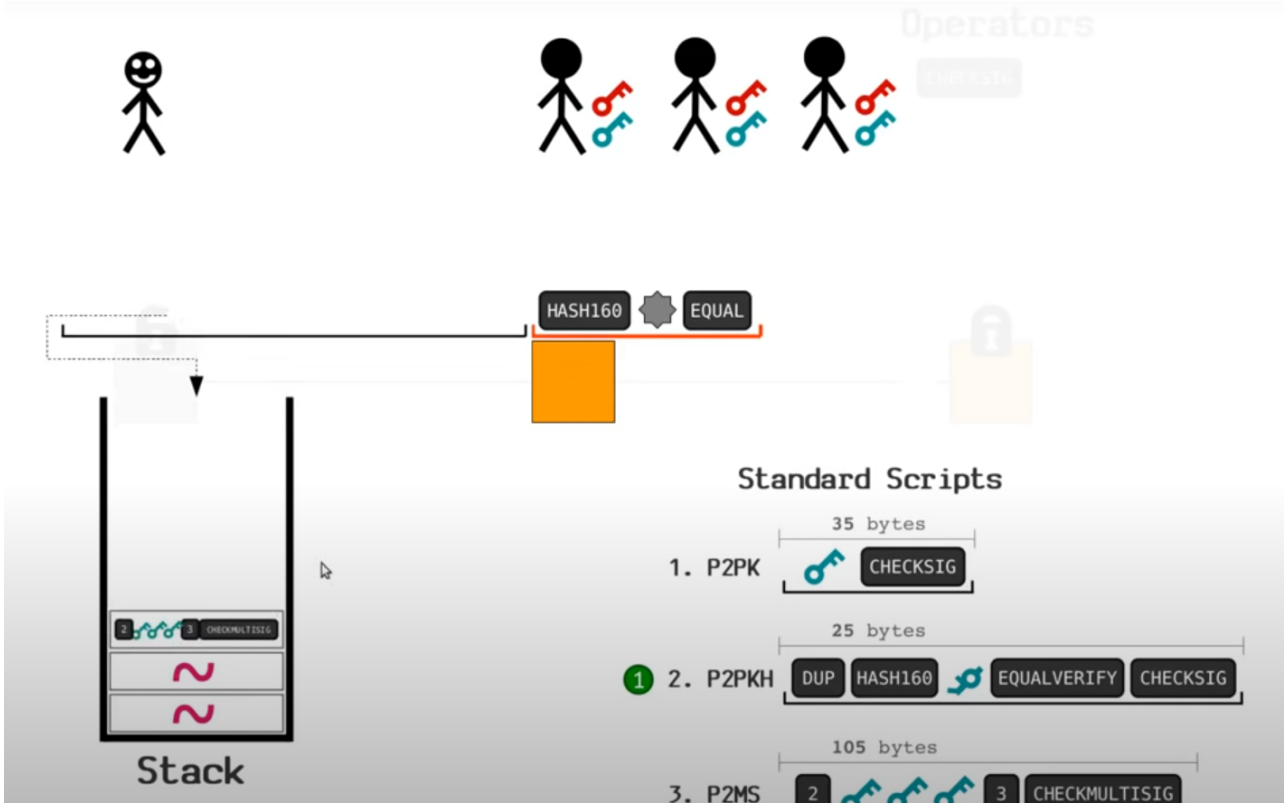
Pay to Script Hash (P2SH)



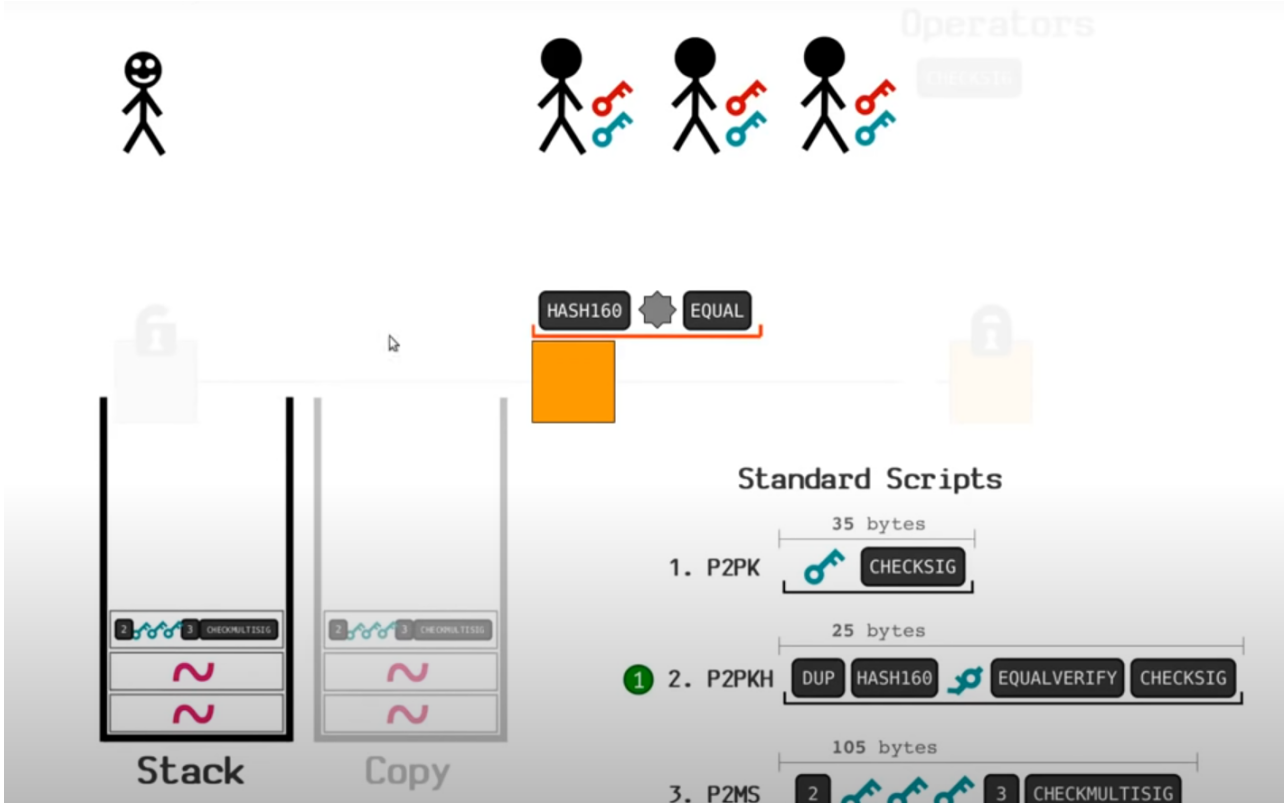
Pay to Script Hash (P2SH)



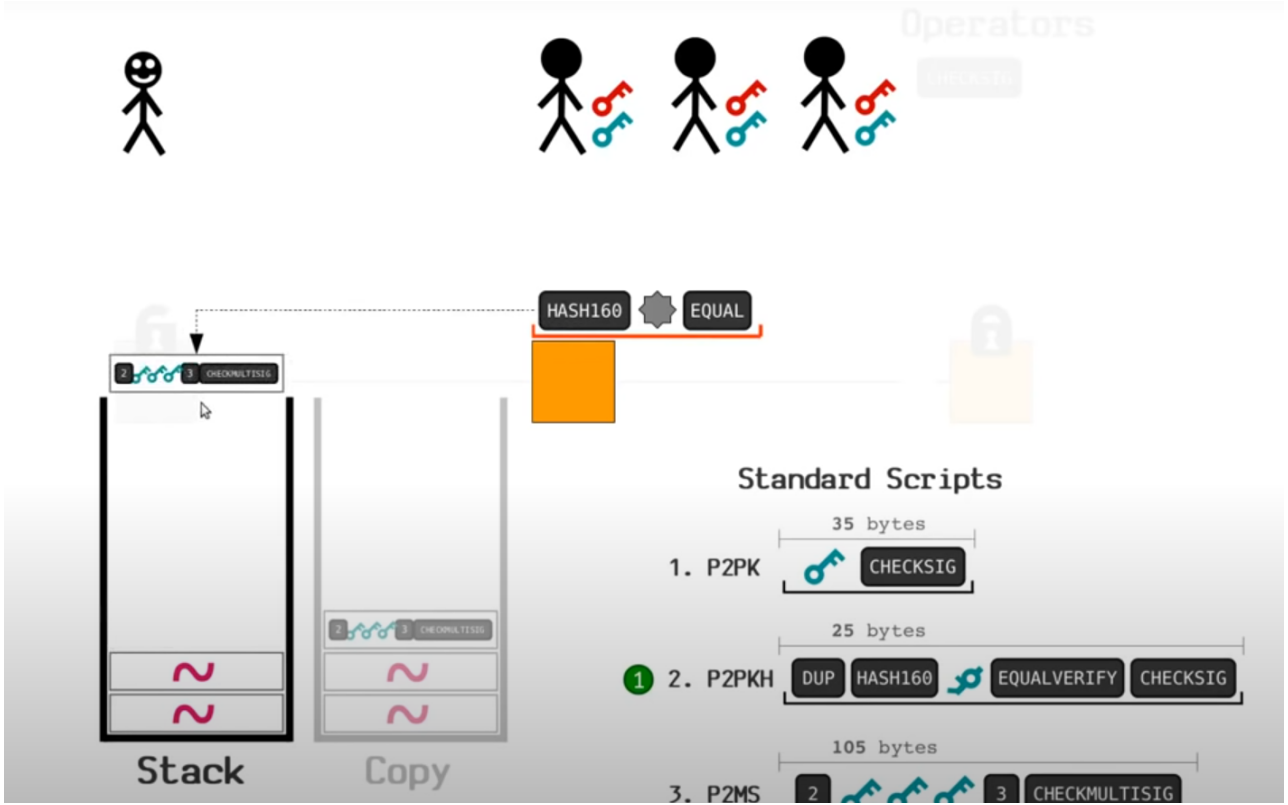
Pay to Script Hash (P2SH)



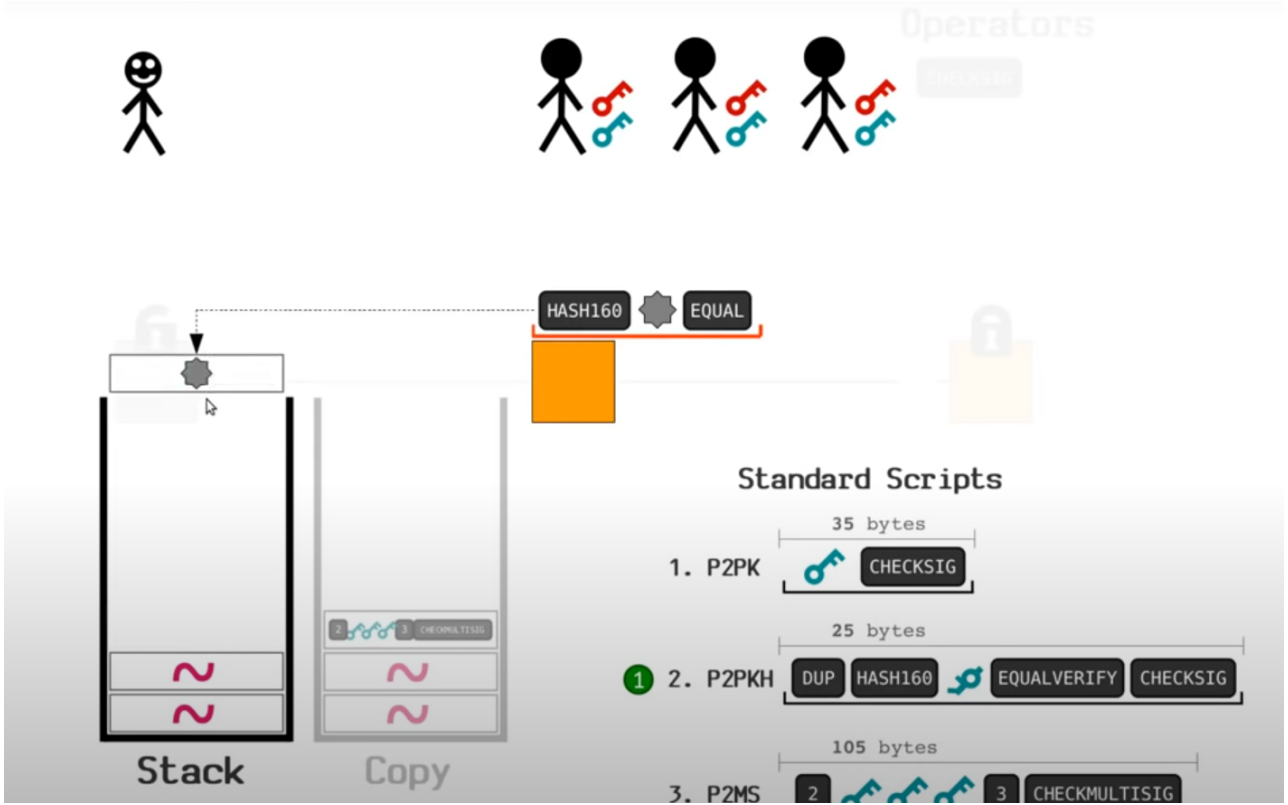
Pay to Script Hash (P2SH)



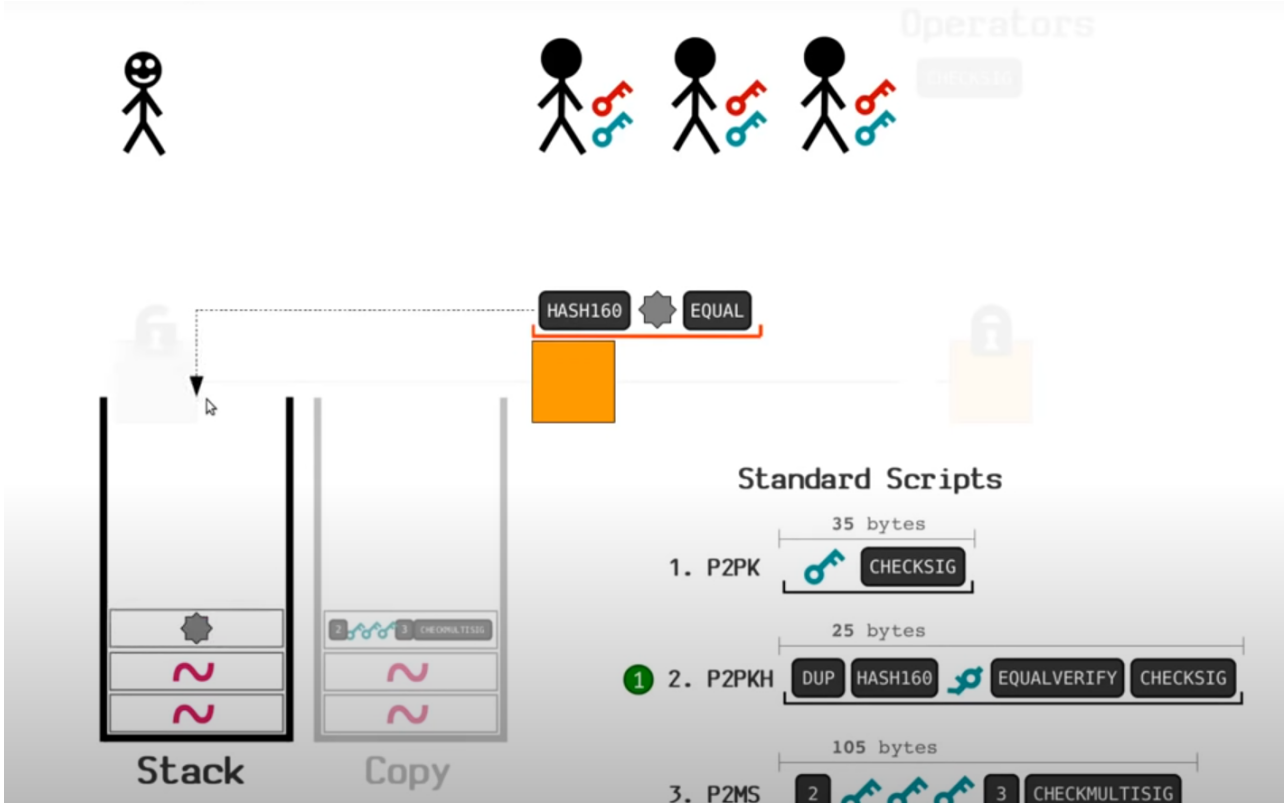
Pay to Script Hash (P2SH)



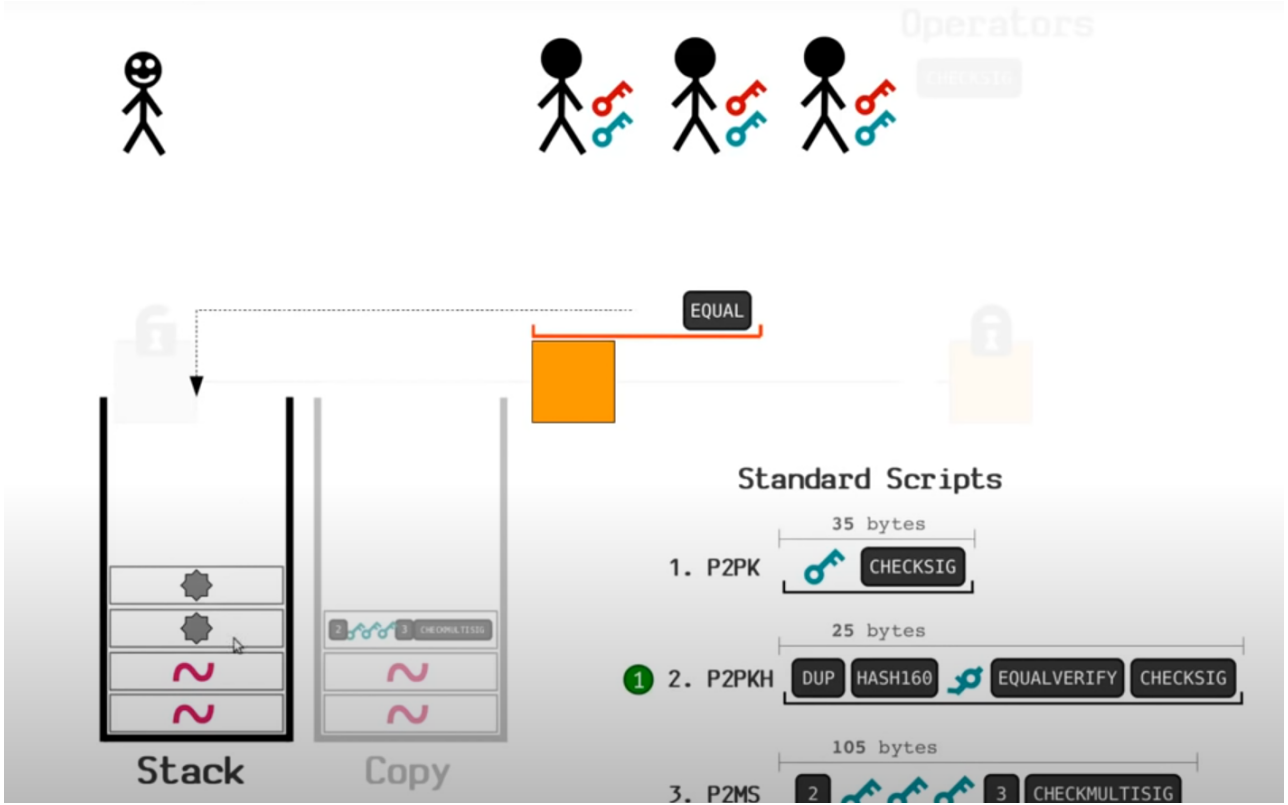
Pay to Script Hash (P2SH)



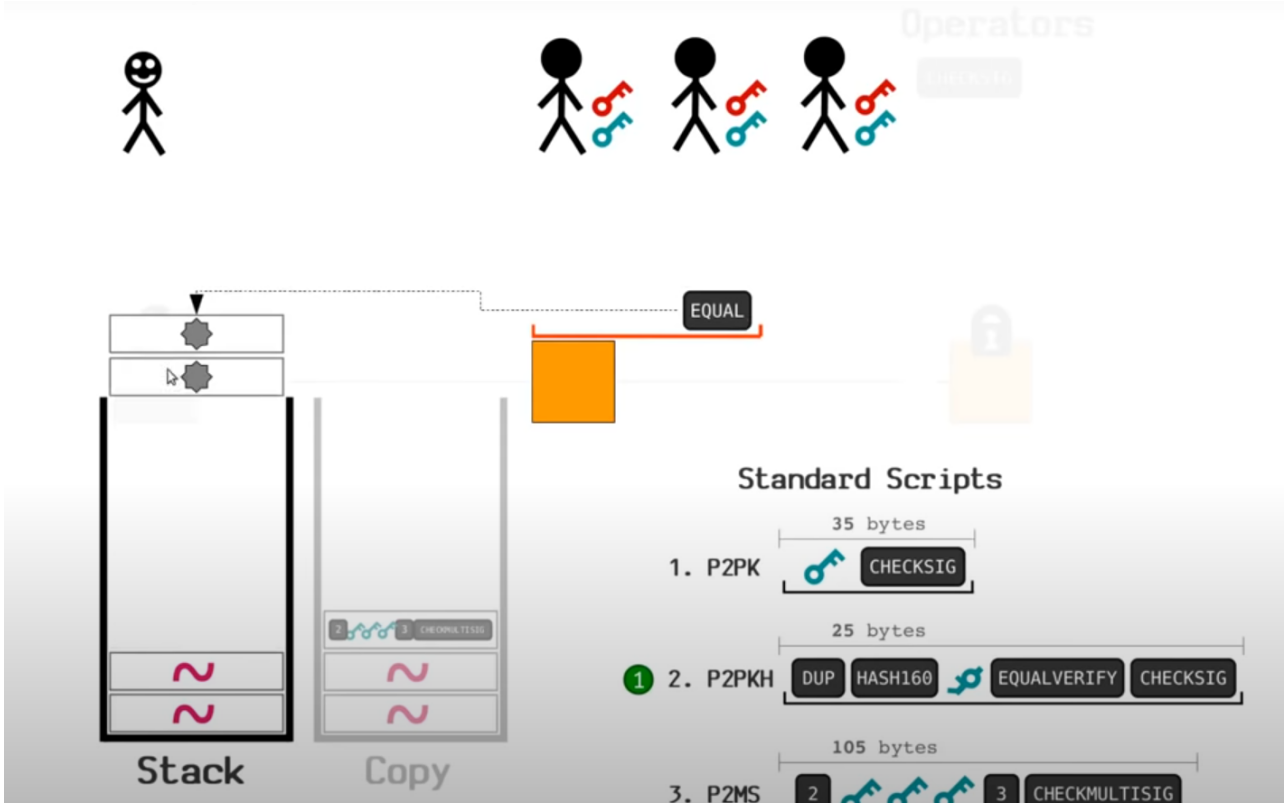
Pay to Script Hash (P2SH)



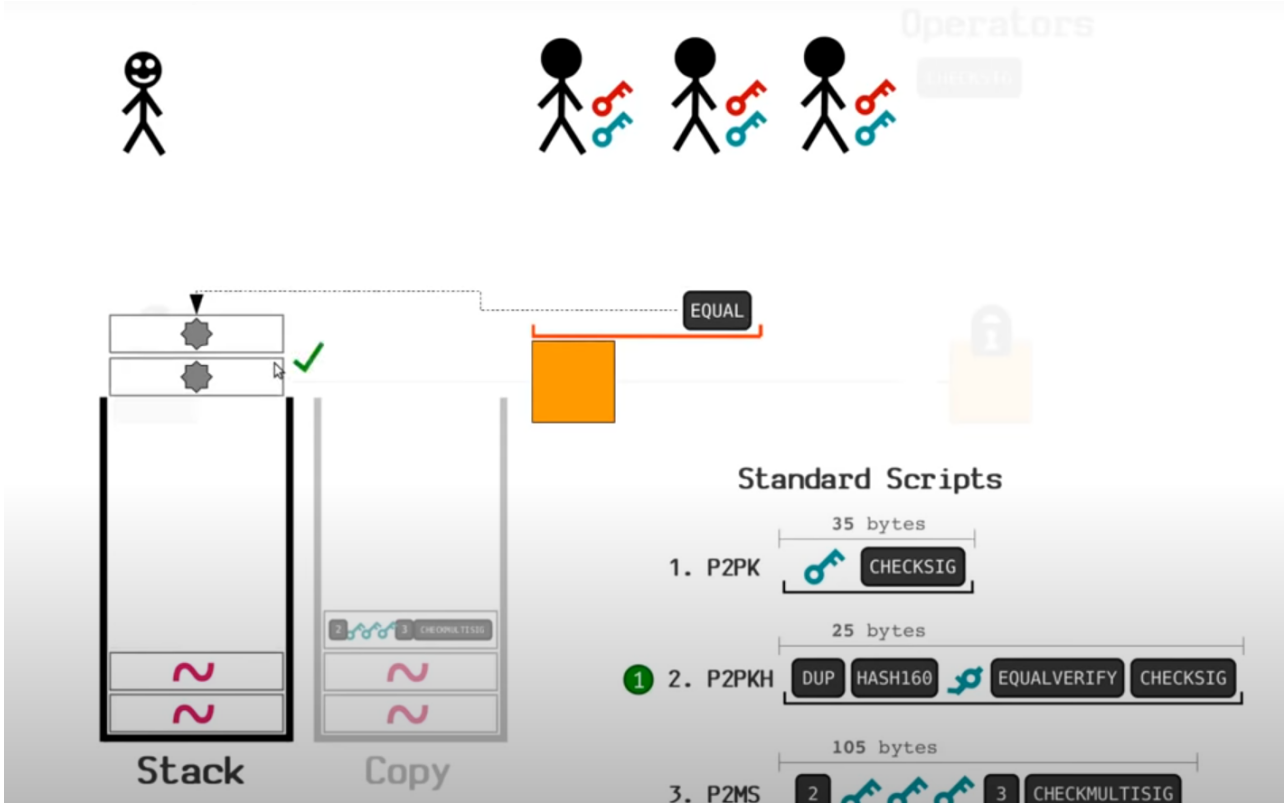
Pay to Script Hash (P2SH)



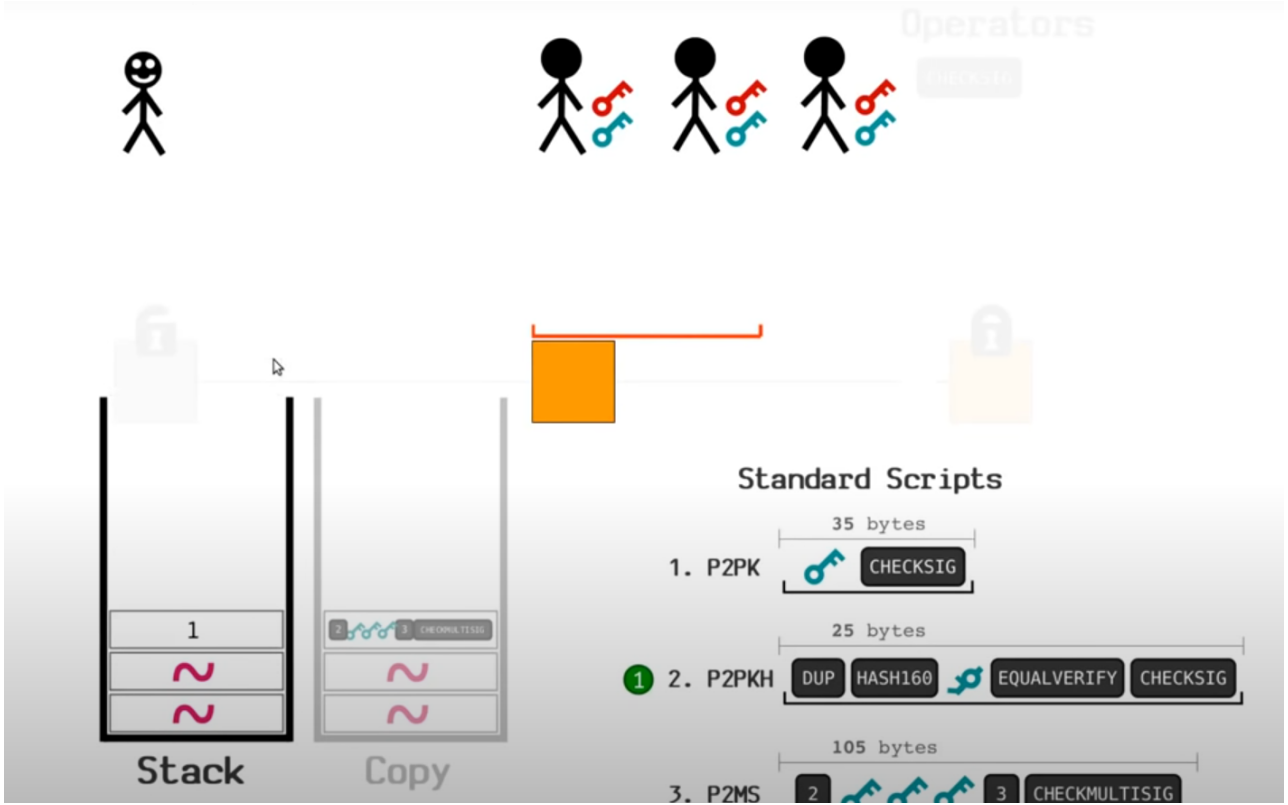
Pay to Script Hash (P2SH)



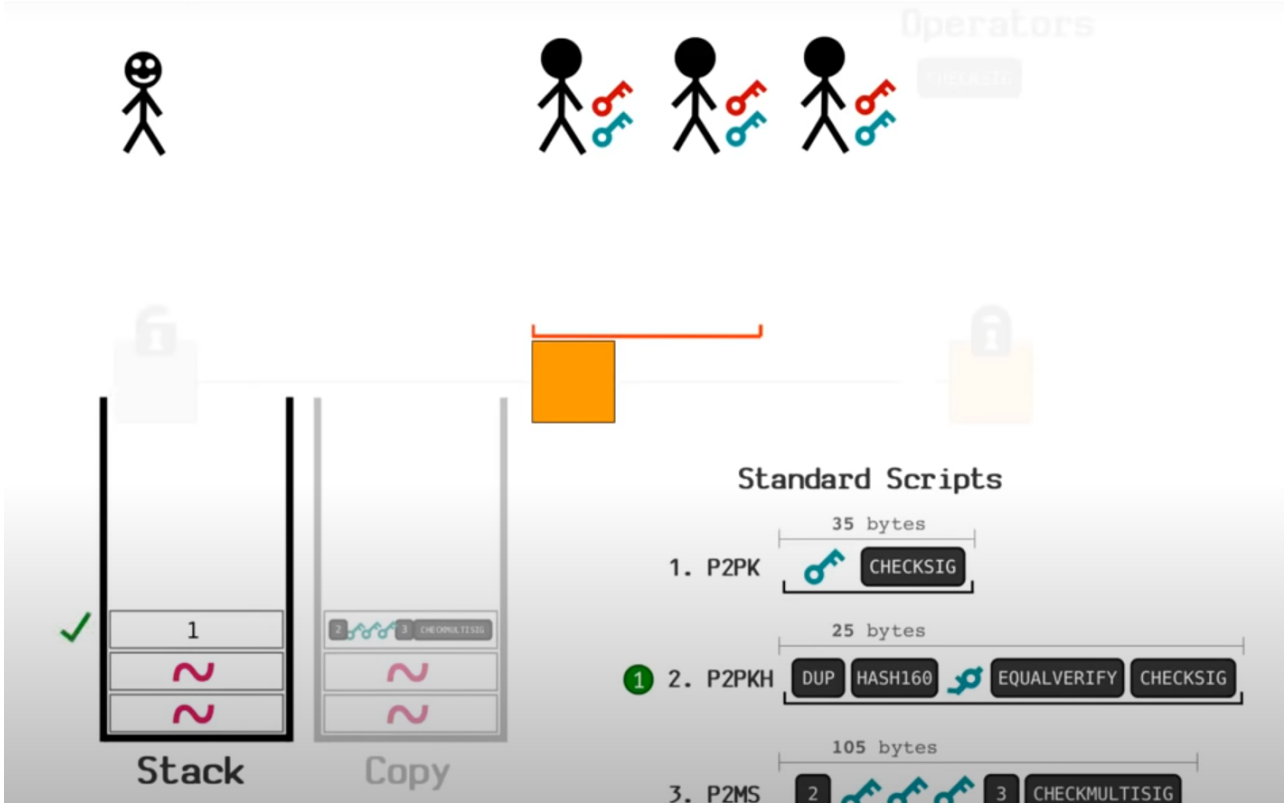
Pay to Script Hash (P2SH)



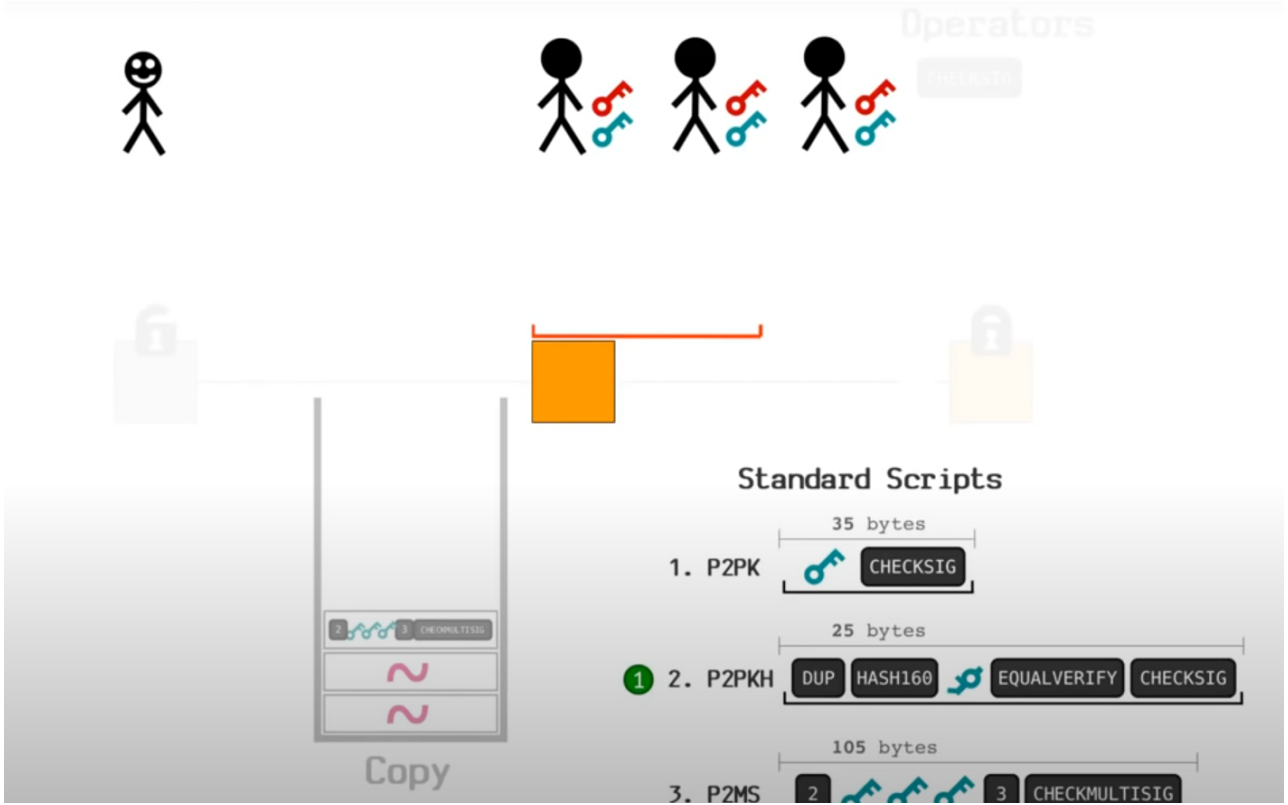
Pay to Script Hash (P2SH)



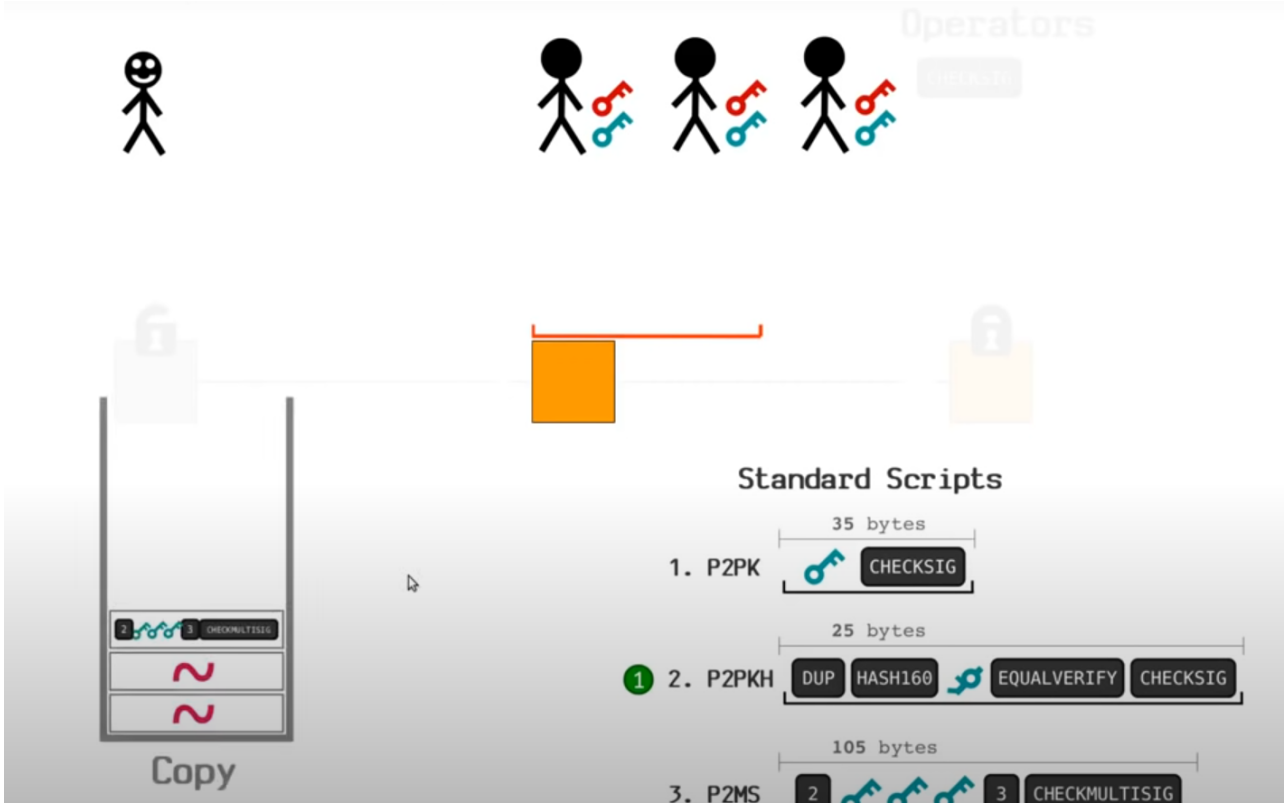
Pay to Script Hash (P2SH)



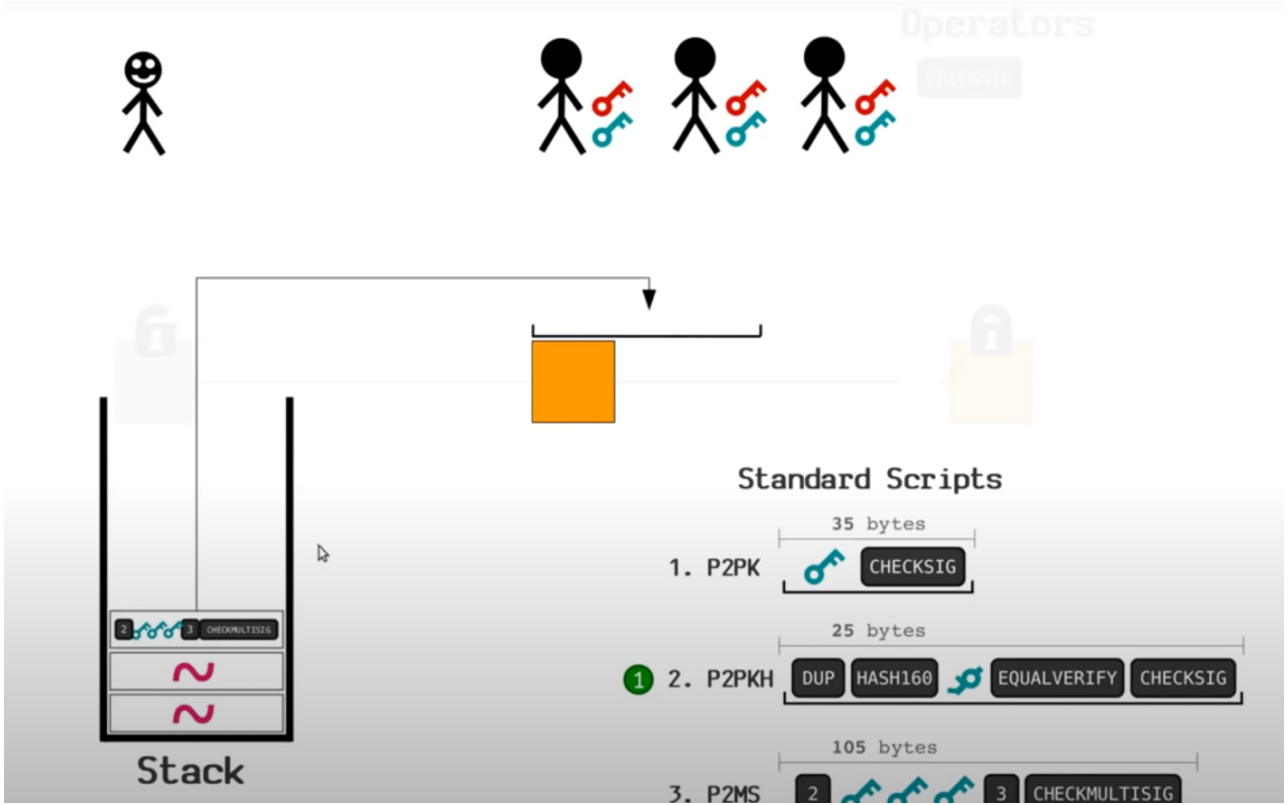
Pay to Script Hash (P2SH)



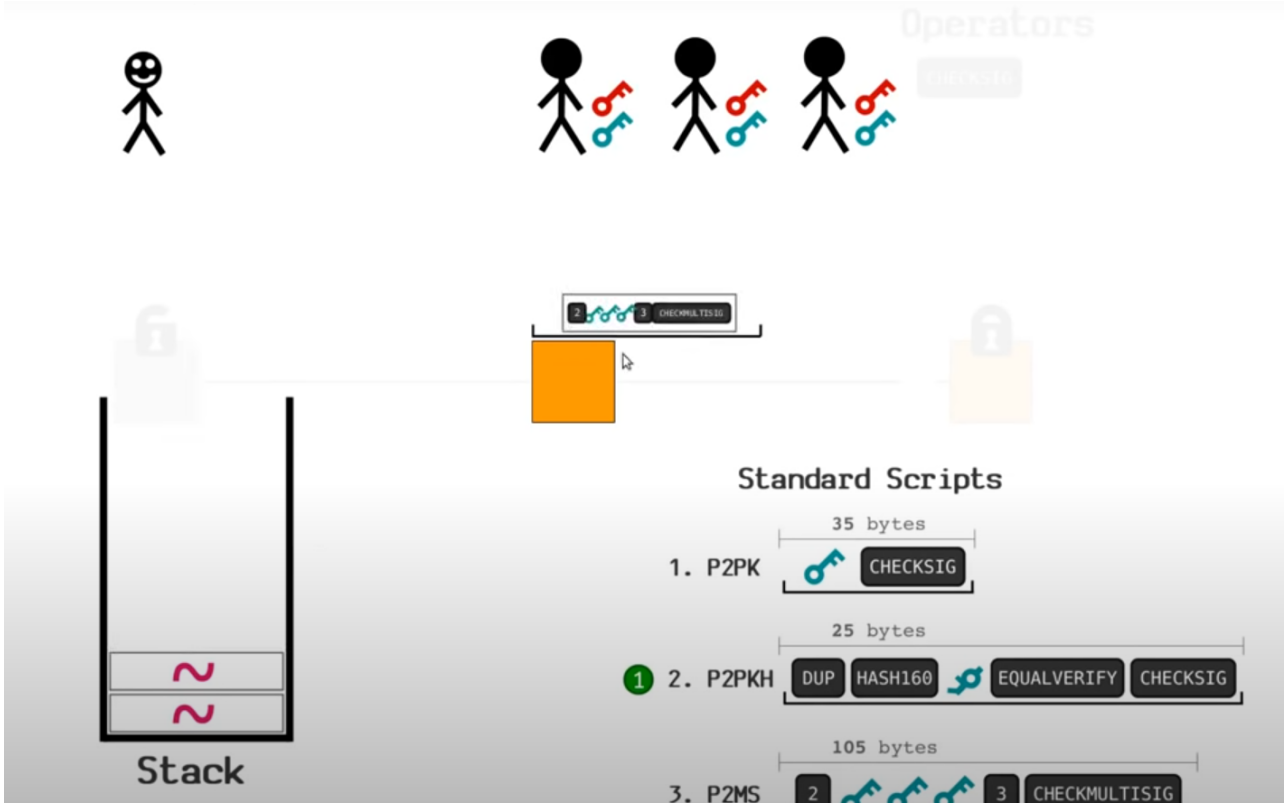
Pay to Script Hash (P2SH)



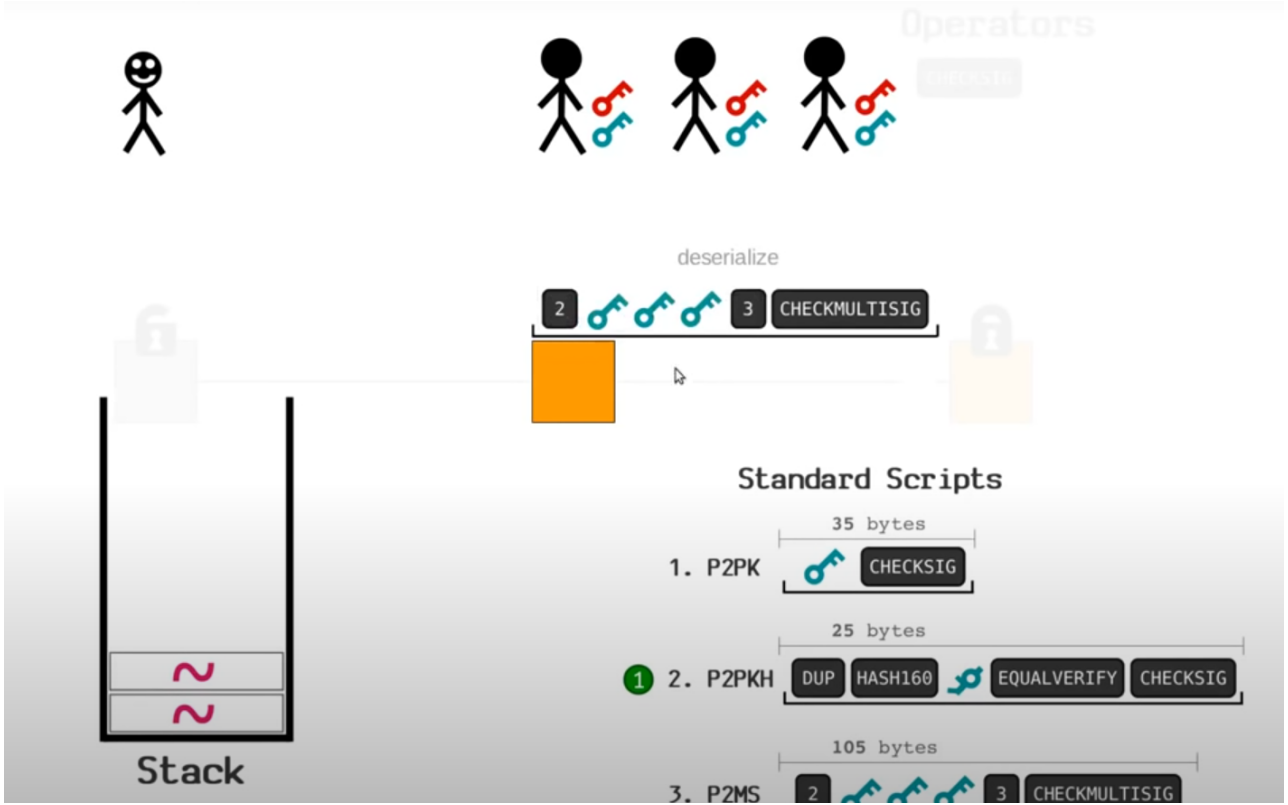
Pay to Script Hash (P2SH)



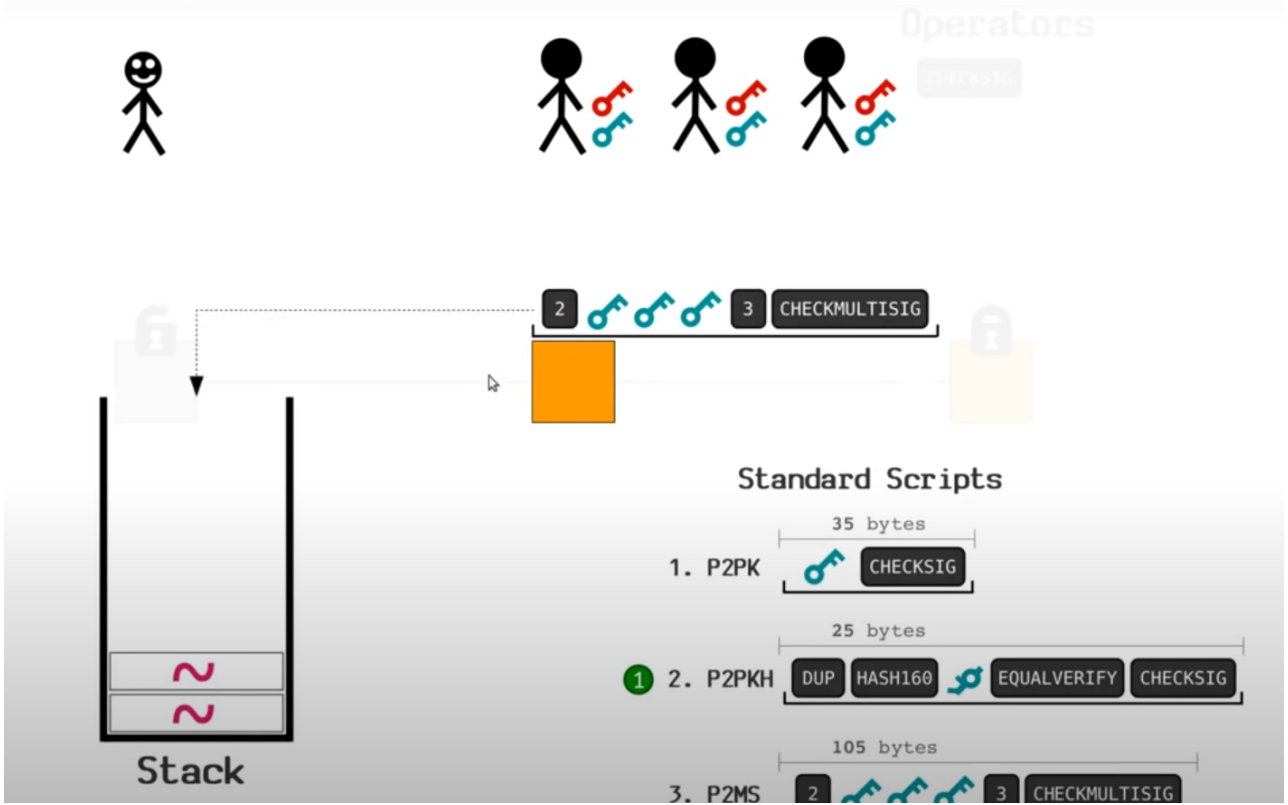
Pay to Script Hash (P2SH)



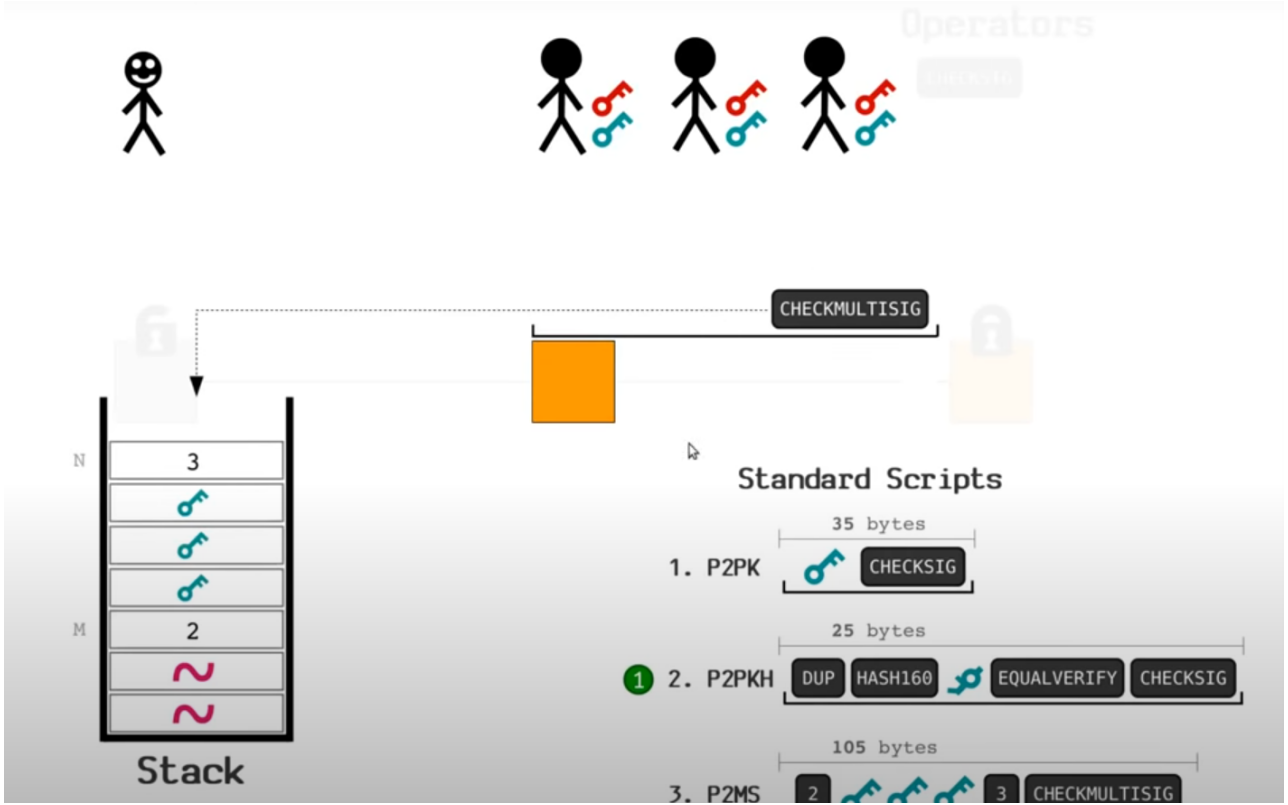
Pay to Script Hash (P2SH)



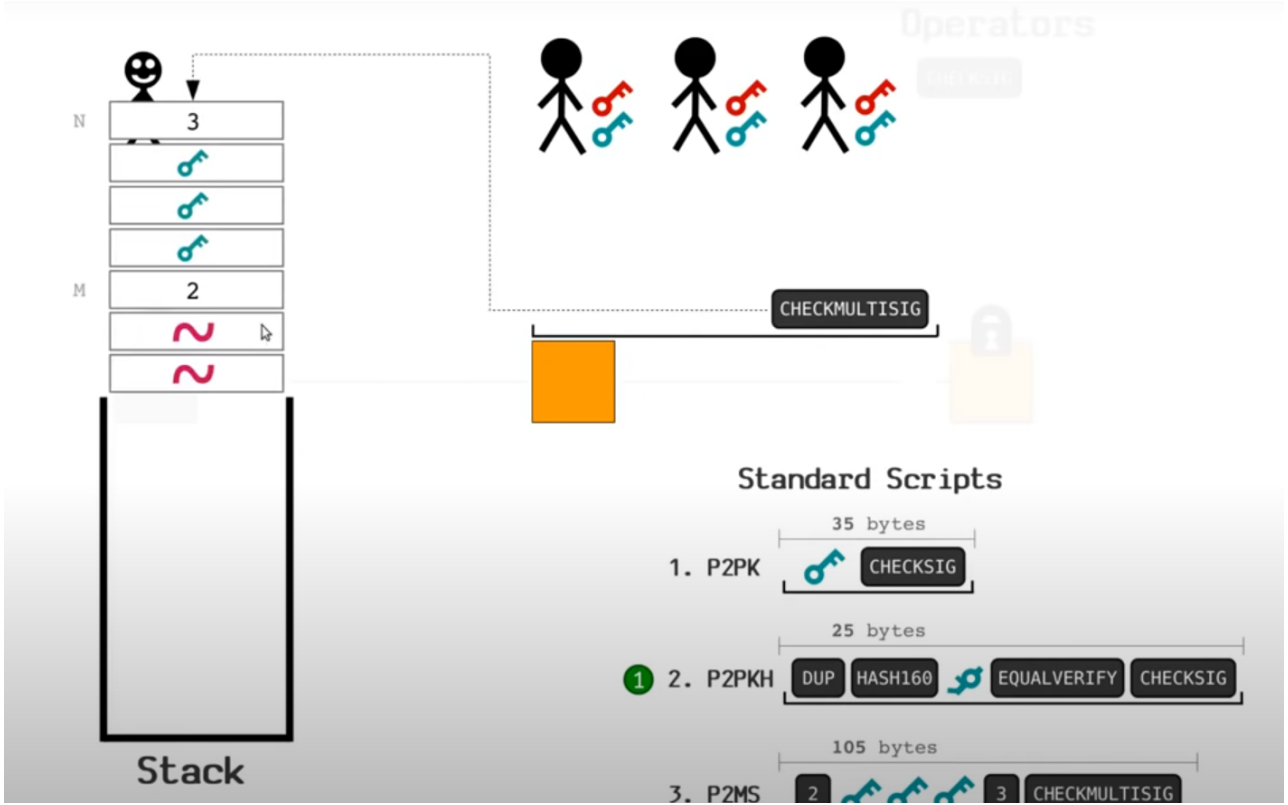
Pay to Script Hash (P2SH)



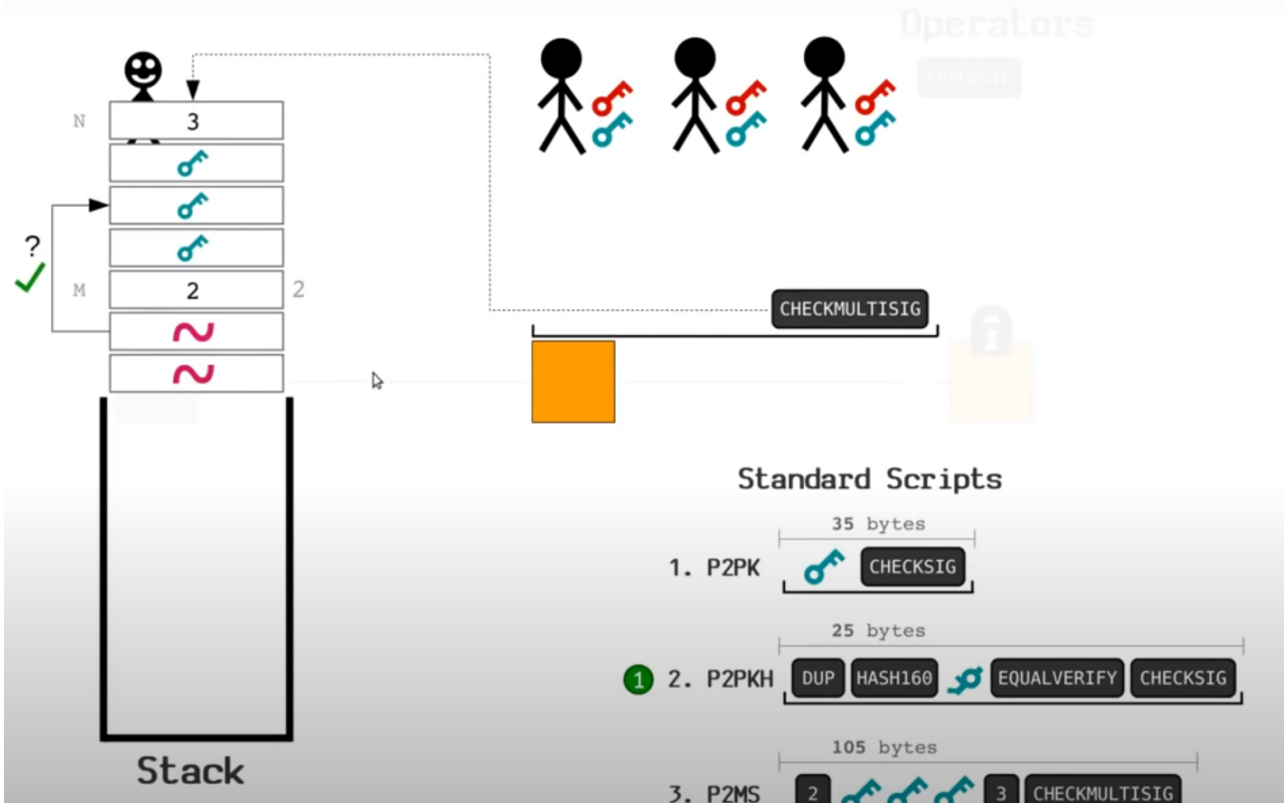
Pay to Script Hash (P2SH)



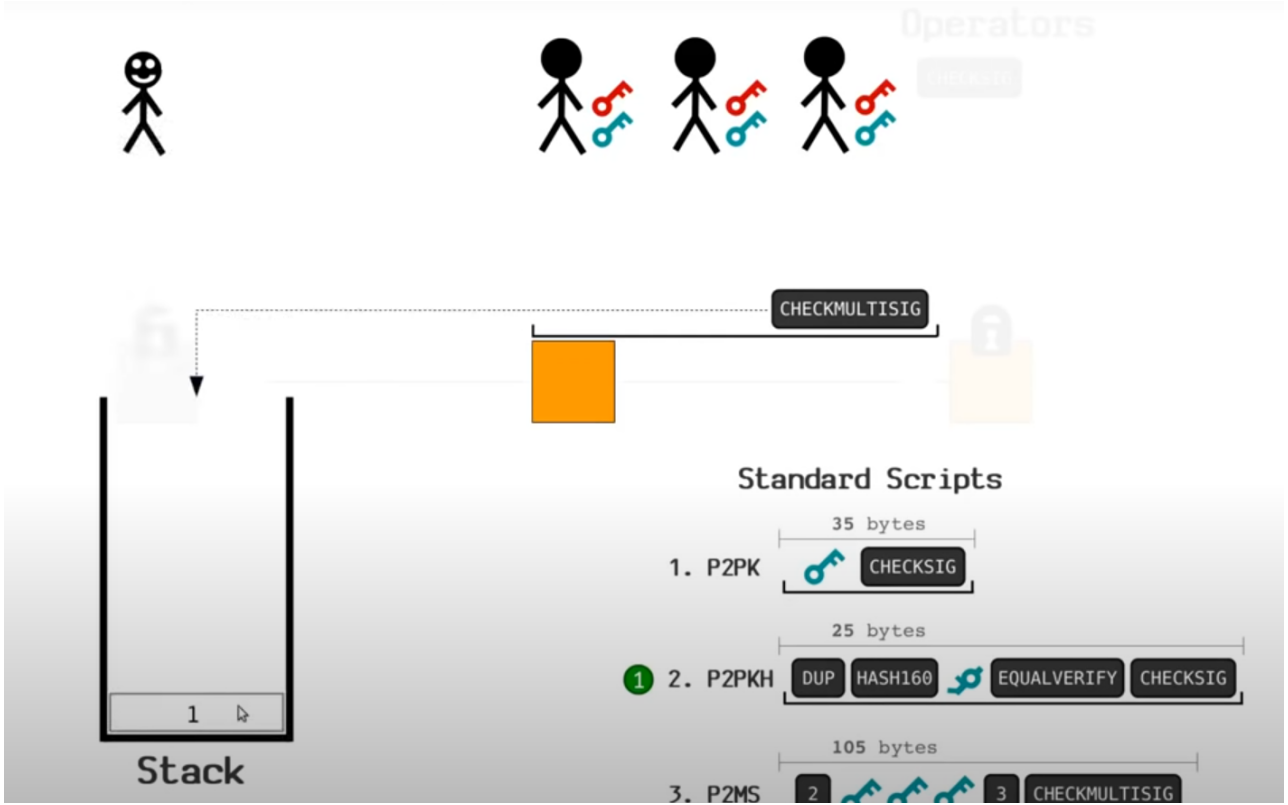
Pay to Script Hash (P2SH)



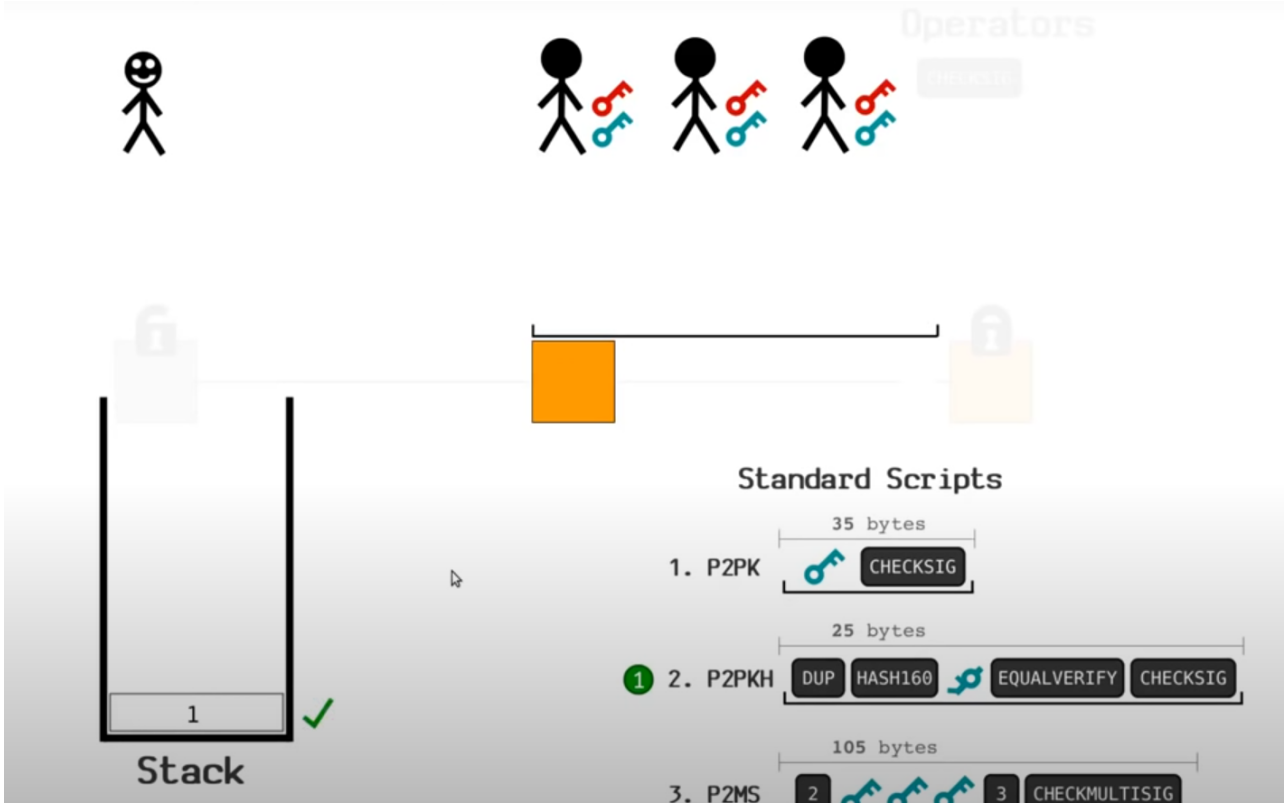
Pay to Script Hash (P2SH)



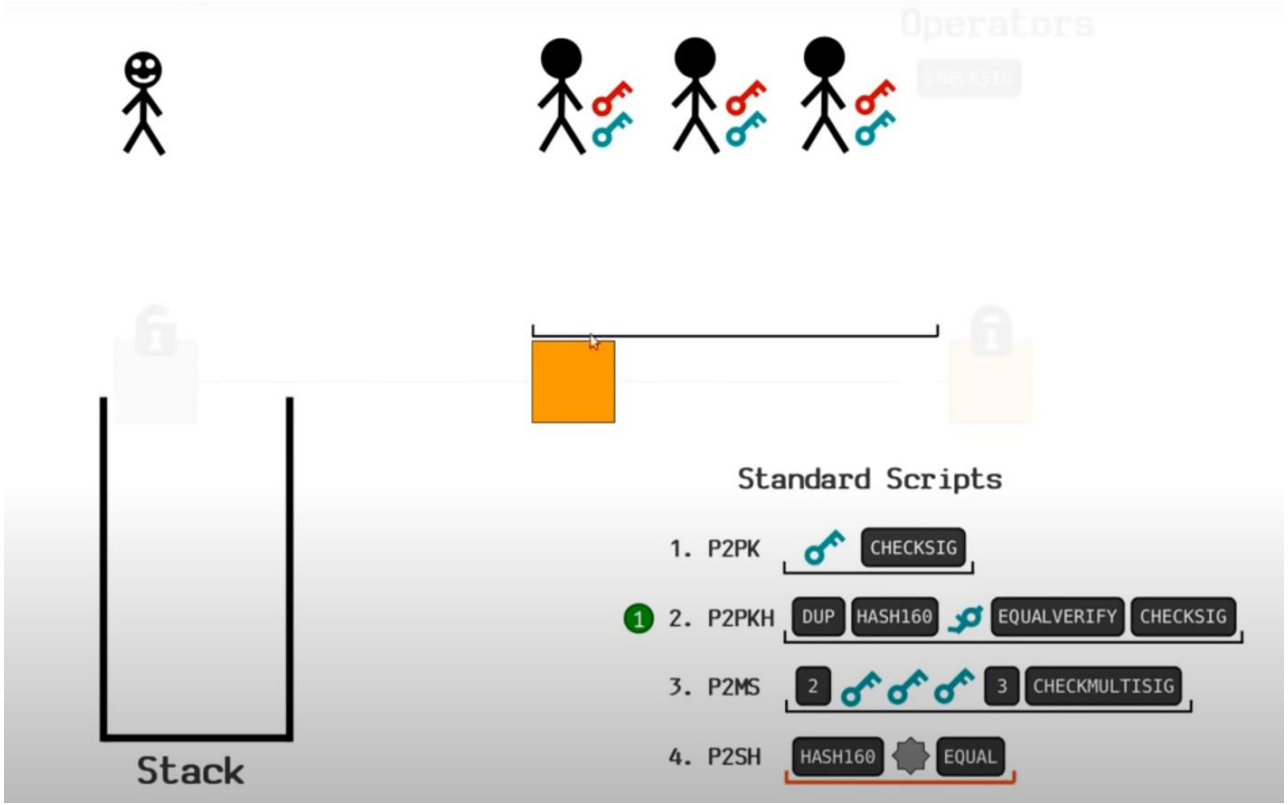
Pay to Script Hash (P2SH)



Pay to Script Hash (P2SH)



Pay to Script Hash (P2SH)



Example: a common script

<sig> <pk>	**DUP HASH256** <pkhash> **EQVERIFY CHECKSIG**

stack:	
empty	init
<sig> <pk>	push values
<sig> <pk> <pk>	**DUP**
<sig> <pk> <hash>	**HASH256**
<sig> <pk> <hash> <pkhash>	push value
<sig> <pk>	**EQVERIFY**
1	**CHECKSIG**

verify(pk, Tx, sig)

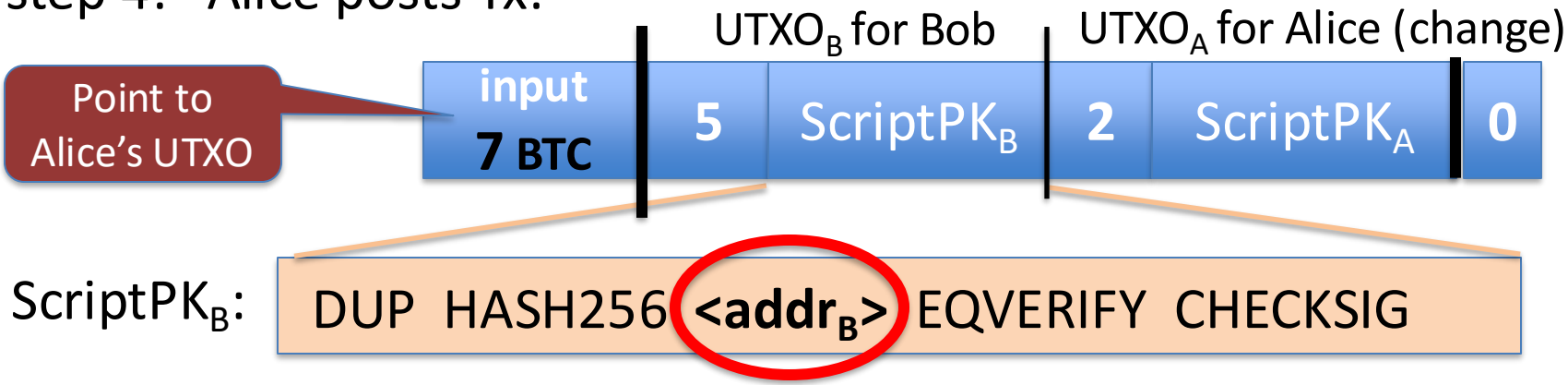
⇒ successful termination

Transaction types: (1) P2PKH

pay to public key hash

Alice want to pay Bob 5 BTC:

- step 1: Bob generates sig key pair $(pk_B, sk_B) \leftarrow Gen()$
- step 2: Bob computes his Bitcoin address as $addr_B \leftarrow H(pk_B)$
- step 3: Bob sends $addr_B$ to Alice
- step 4: Alice posts Tx:



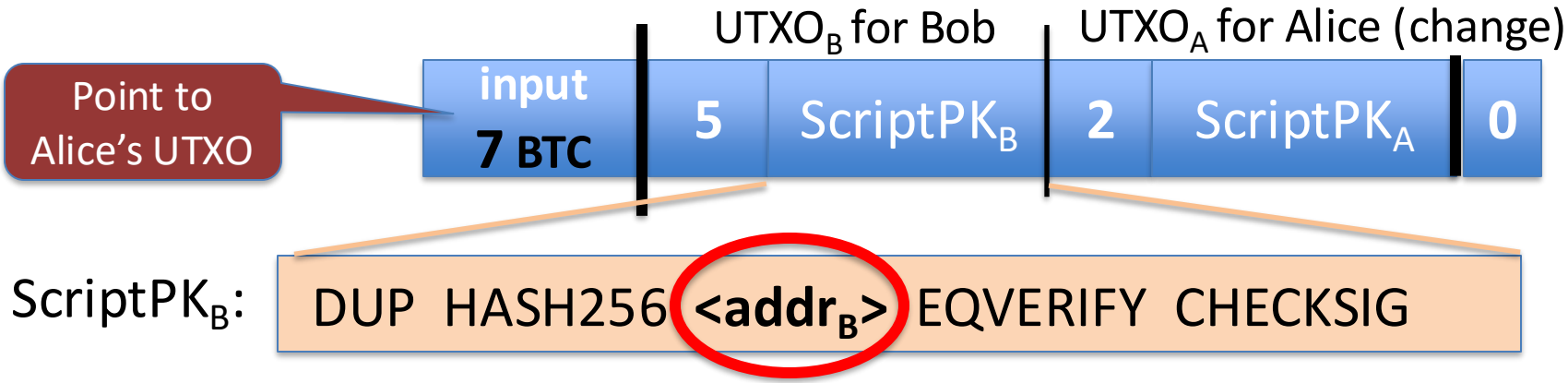
Transaction types: (1) P2PKH

pay to public key hash

"input" contains ScriptSig that authorizes spending Alice's UTXO

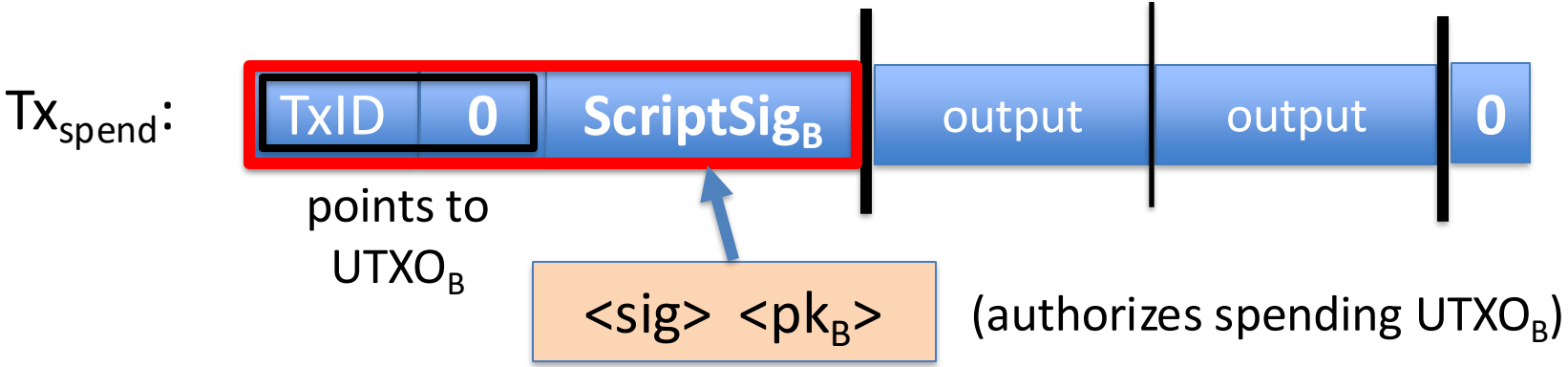
- example: ScriptSig contains Alice's signature on Tx

  $\Longrightarrow$ miners cannot change $ScriptPK_B$ (will invalidate Alice's signature)



Transaction types: (1) P2PKH

Later, when Bob wants to spend his UTXO: create a Tx_{spend}



<sig> = $Sign(sk_B, Tx)$ where $Tx = (Tx_{spend}$ excluding all ScriptSigs) (SIGHASH_ALL)

Miners validate that $ScriptSig_B$ | $ScriptPK_B$ returns true

P2PKH: comments

- Alice specifies recipient's pk in $UTXO_B$
- Recipient's pk is not revealed until UTXO is spent
  (some security against attacks on pk)
- Miner cannot change $<Addr_B>$ and steal funds:
  invalidates Alice's signature that created $UTXO_B$

Segregated Witness

ECDSA malleability:

Given (m, sig) anyone can create (m, sig') with sig ≠ sig'

⇒ miner can change sig in Tx and change TxID = SHA256(Tx)

⇒ Tx issuer cannot tell what TxID is, until Tx is posted

⇒ leads to problems and attacks

Segregated witness: signature is moved to witness field in Tx

TxID = Hash(Tx without witnesses)

Transaction types: (2) P2SH: pay to script hash

(pre SegWit in 2017)

Let's payer specify a redeem script (instead of just pkhash)

Usage: payee publishes hash(redeem script) ⟵ Bitcoint addr.
payer sends funds to that address

ScriptPK in UTXO: HASH160 <H(redeem script)> EQUAL

ScriptSig to spend: $<sig_1>$ $<sig_2>$ … $<sig_n>$ <redeem script>

payer can specify complex conditions for when UTXO can be spent

P2SH

Miner verifies:

(1) <ScriptSig> ScriptPK = true ⟵ payee gave correct script

(2) ScriptSig = true ⟵ script is satisfied

Example P2SH: multisig

Goal: spending a UTXO requires t-out-of-n signatures

Redeem script for 2-out-of-3: (set by payer)

<2> <PK_1> <PK_2> <PK_3> <3> CHECKMULTISIG

hash gives P2SH address

ScriptSig to spend: (by payee) <0> <sig1> <sig3> <redeem script>

Resources

- ECE/COS 470, Pramod Viswanath, Princeton 2024
- CS251, Dan Boneh, Stanford 2023
- Bitcoin Lesson – Transactions by learnmebitcoin