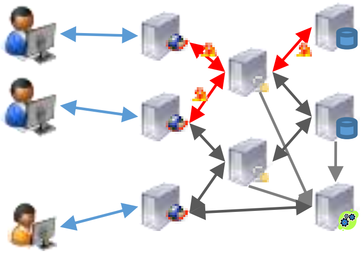


Verteilte Systeme/ Distributed Systems

Artur Andrzejak

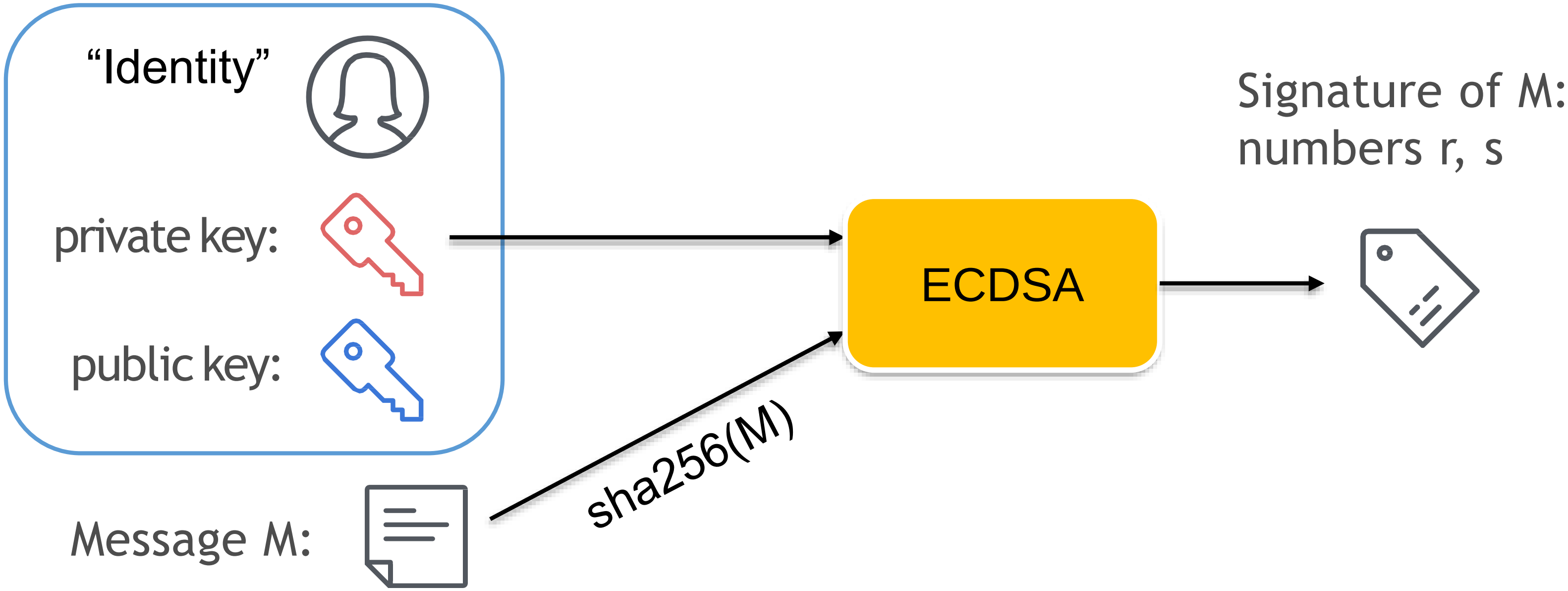
13

Signatures

Slides and content in part based on the course
BerkeleyX: CS198.1x “Bitcoin and Cryptocurrencies”
@ edX: https://courses.edx.org/courses/course-v1:BerkeleyX+CS198.1x+3T2018/course/

Signatures of Transactions in Bitcoin

- Bitcoin uses Elliptic Curve Digital Signature Algorithm (ECDSA, link) to sign a transaction/message
- To "sign" means to certify that a transaction content has been verified (or created) by a unique "identity"
- This "identity" is uniquely specified by a private/ public key

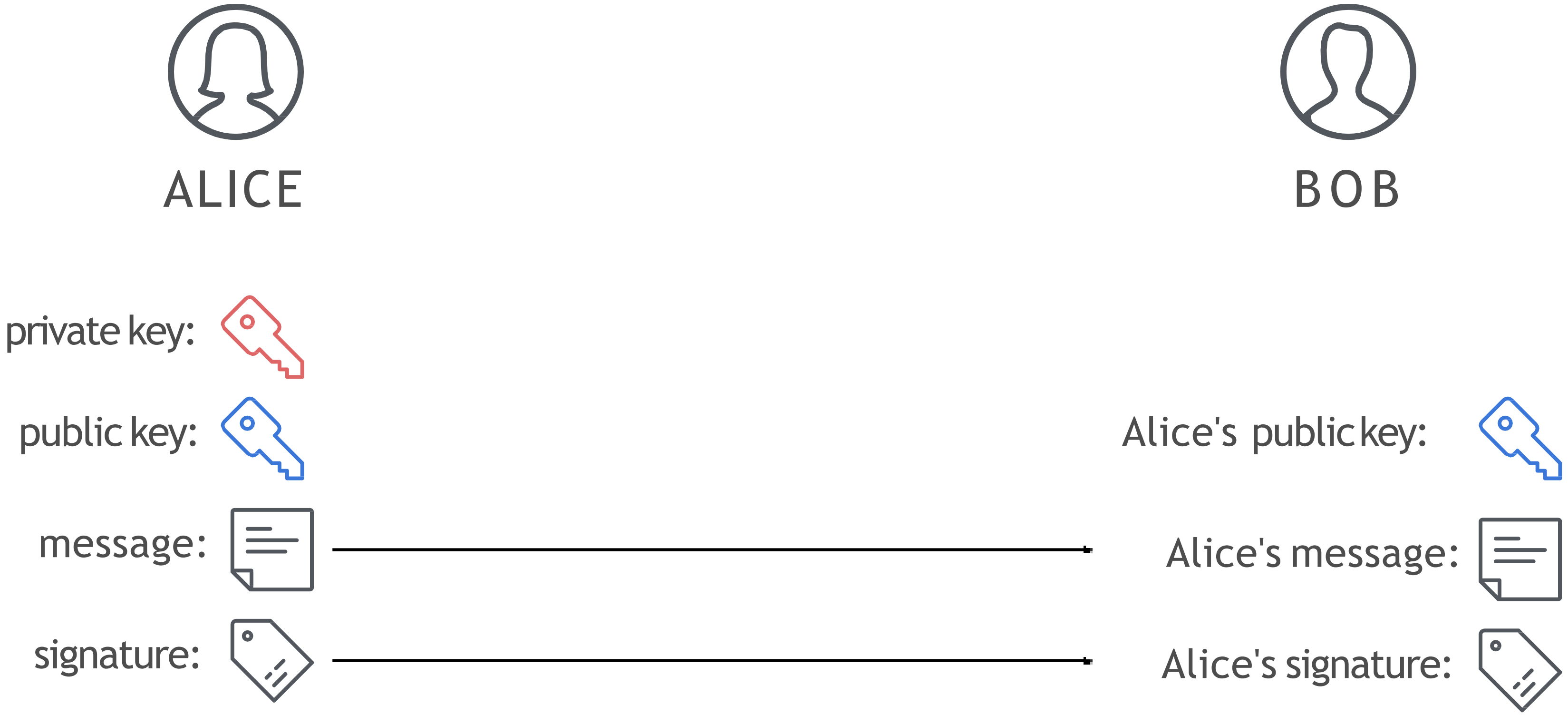


Signatures in General

- **Signature S**: a pair of two numbers r, s
- S is generated *from* a hash H of something to be signed (a message M), and from a **private key PRIV**
- With the **public key PUB** (corresponding to PRIV), an algorithm can be used on the signature S …
  - … to determine that it was originally produced from the hash H and the private key PRIV
  - This verification does not need to know the private key PRIV
- => If we know PUB, M, and S, we can verify that the signer knows PRIV, and has seen the same M as we see it now
- => Moreover, it is practically impossible to produce S without PRIV, so S can only come from owner of PRIV

Application Scenario

Alice sends message + signature



ALICE

BOB

private key:

public key:

message:

signature:

Alice's public key:

Alice's message:

Alice's signature:

Application Scenario

Bob can easily verify if Alice signed

ALICE

BOB

private key: 🔑

public key: 🔑

message: 📄

signature: 🏷

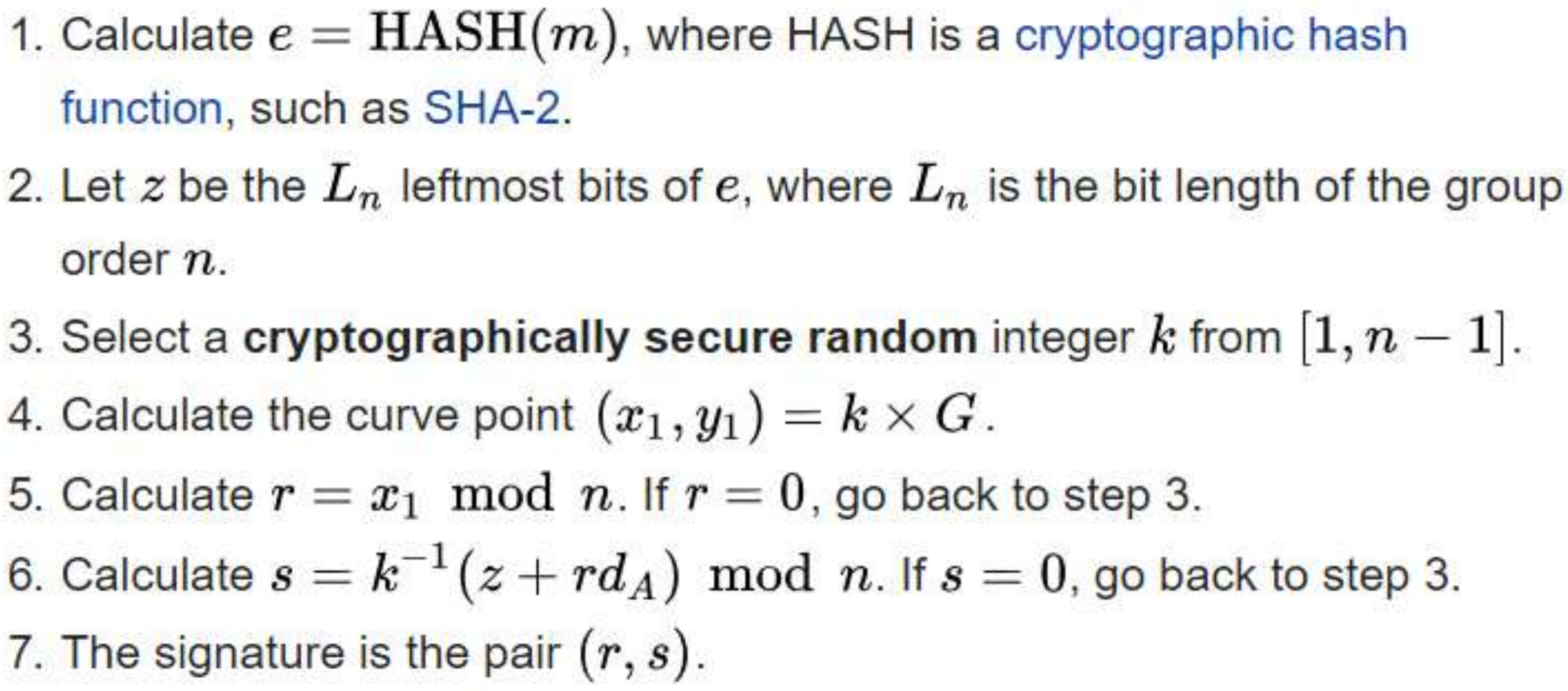
Alice's message: 📄

🔑 + 🏷 = ✓ or ✗

Alice's public key

Alice's signature

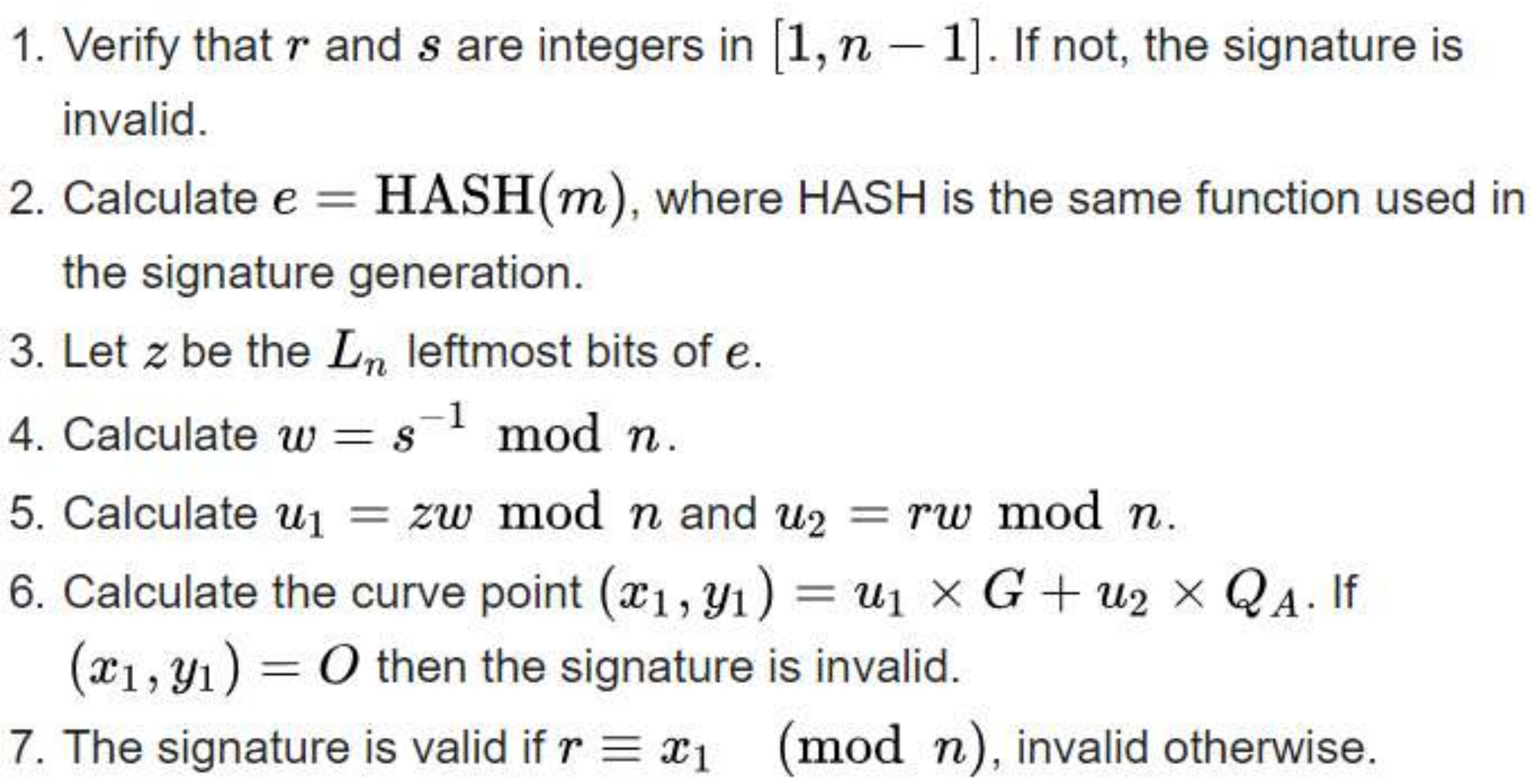
Signature Generation Algorithm (link)



1. Calculate $e = \mathrm{HASH}(m)$, where HASH is a cryptographic hash function, such as SHA-2.
2. Let z be the L_n leftmost bits of e, where L_n is the bit length of the group order n.
3. Select a **cryptographically secure random** integer k from $[1, n-1]$.
4. Calculate the curve point $(x_1, y_1) = k \times G$.
5. Calculate $r = x_1 \mod n$. If $r = 0$, go back to step 3.
6. Calculate $s = k^{-1}(z + rd_A) \mod n$. If $s = 0$, go back to step 3.
7. The signature is the pair (r, s).

rd_A: private key
Q_A: public key ($Q_A = d_A * G$)
G: elliptic curve base point (public)
n: integer order of G, must be prime

Signature Verification Algorithm (link)



1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $e = \mathrm{HASH}(m)$, where HASH is the same function used in the signature generation.
3. Let z be the L_n leftmost bits of e.
4. Calculate $w = s^{-1} \mod n$.
5. Calculate $u_1 = zw \mod n$ and $u_2 = rw \mod n$.
6. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
7. The signature is valid if $r \equiv x_1 \pmod{n}$, invalid otherwise.

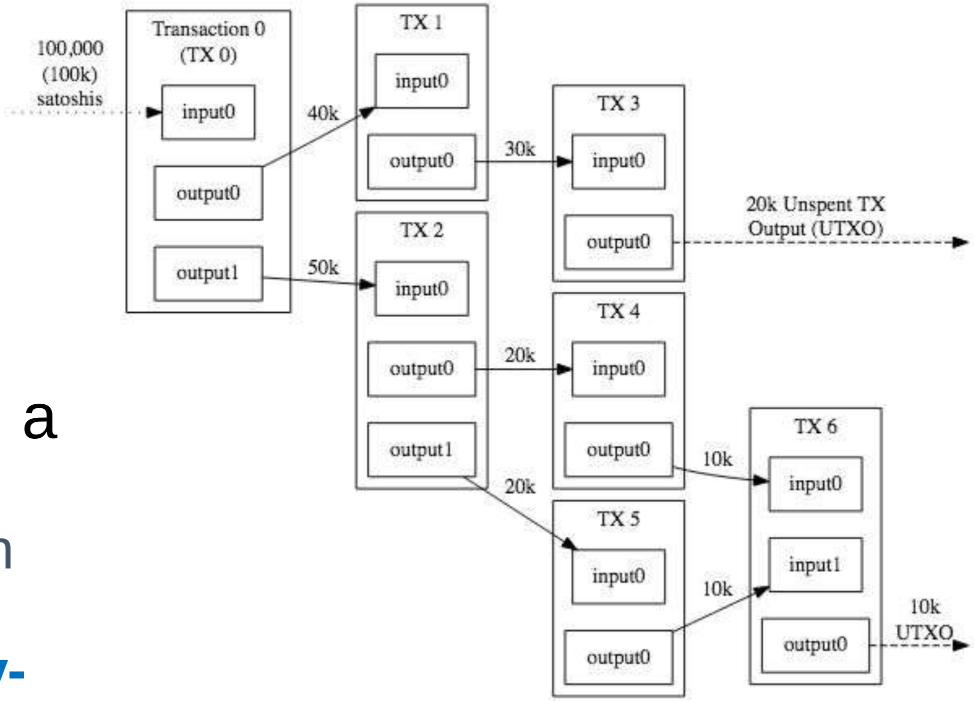
rd_A: private key
Q_A: public key ($Q_A = d_A * G$)
G: elliptic curve base point (public)
n: integer order of G, must be prime

Bitcoin Script

Slides and content in part based on the course
BerkeleyX: CS198.1x “Bitcoin and Cryptocurrencies”
@ edX: https://courses.edx.org/courses/course-v1:BerkeleyX+CS198.1x+3T2018/course/

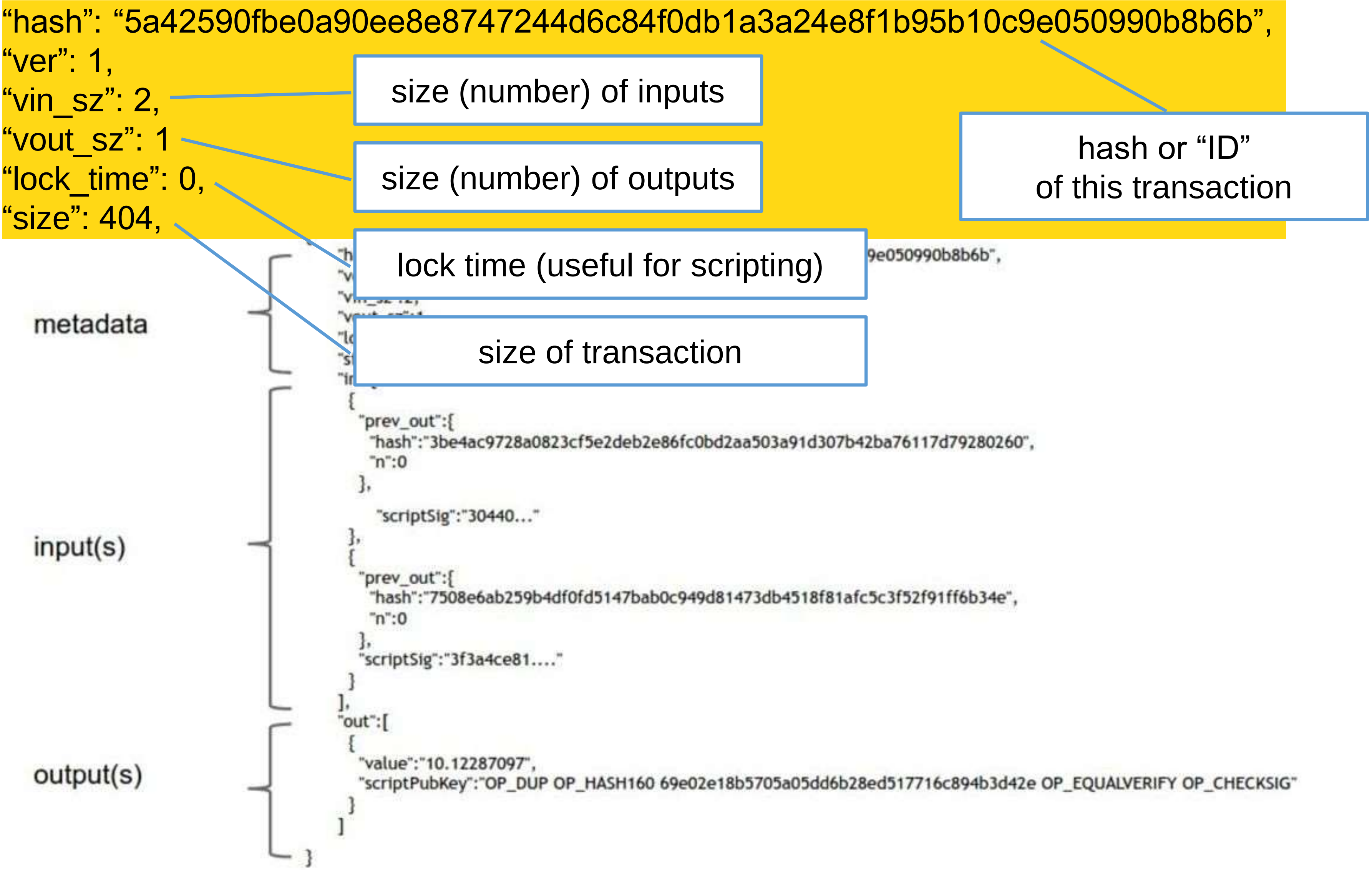
Reminder: the UTXO Model

- Bitcoin uses a UTXO model
- Transactions map inputs to outputs
- Transactions contain signature of owner of funds
- Spending Bitcoin is redeeming previous transaction outputs with a proof
  - Public Key + Signature in **Pay-to-Pub-Key-Hash**
  - Script + Signature in **Pay-to-Script-Hash**



Source: Bitcoin Developer Guide

Contents of a Transaction



“hash”: “5a42590fbe0a90ee8e8747244d6c84f0db1a3a24e8f1b95b10c9e050990b8b6b”,
“ver”: 1,
“vin_sz”: 2,
“vout_sz”: 1
“lock_time”: 0,
“size”: 404,

size (number) of inputs

size (number) of outputs

lock time (useful for scripting)

size of transaction

hash or “ID”
of this transaction

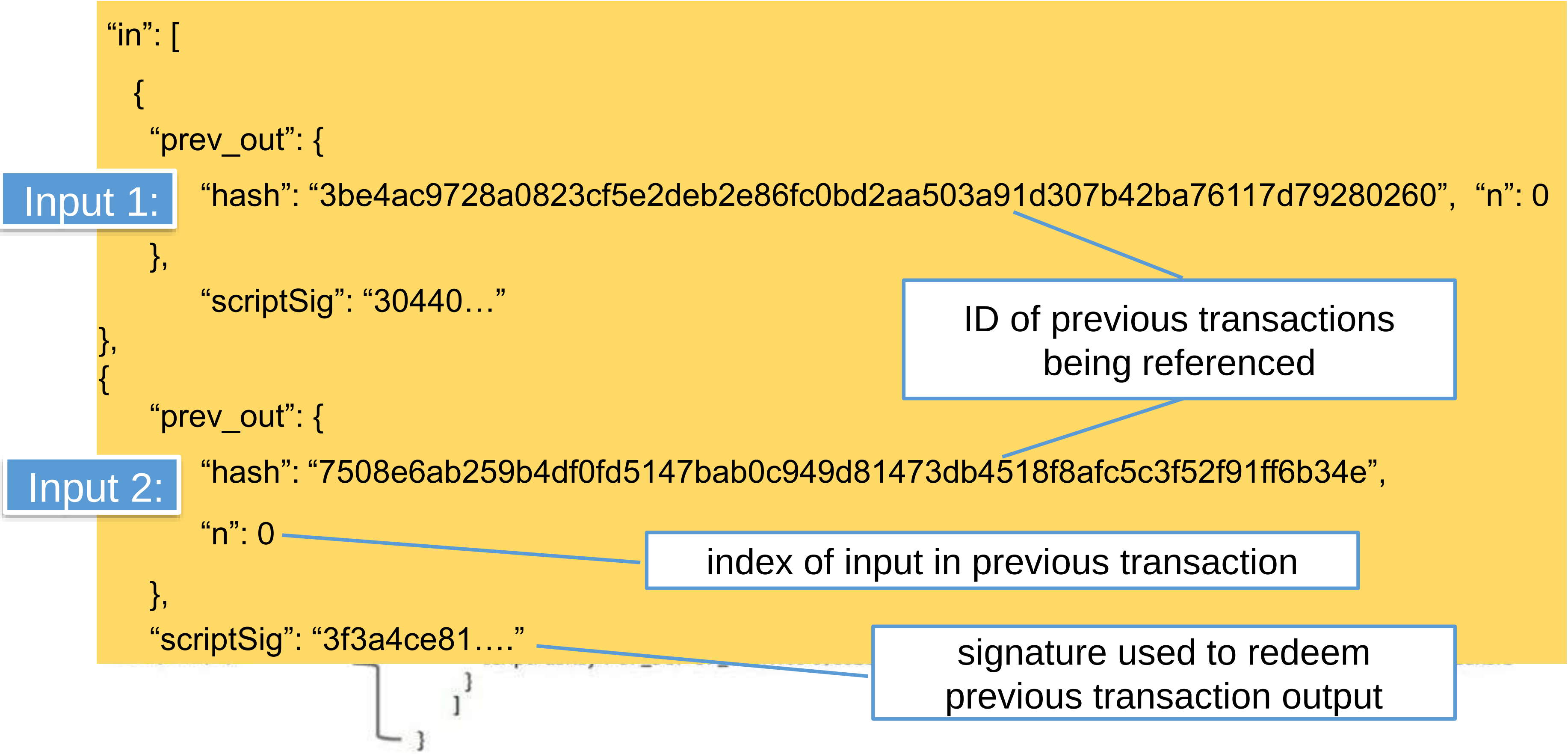
```
"h                                                   9e050990b8b6b",
"v
"vi
"v
"lo
"si
"in
  {
   "prev_out":{
    "hash":"3be4ac9728a0823cf5e2deb2e86fc0bd2aa503a91d307b42ba76117d79280260",
    "n":0
   },

     "scriptSig":"30440..."
  },
  {
   "prev_out":{
    "hash":"7508e6ab259b4df0fd5147bab0c949d81473db4518f81afc5c3f52f91ff6b34e",
    "n":0
   },
   "scriptSig":"3f3a4ce81...."
  }
],
"out":[
  {
   "value":"10.12287097",
   "scriptPubKey":"OP_DUP OP_HASH160 69e02e18b5705a05dd6b28ed517716c894b3d42e OP_EQUALVERIFY OP_CHECKSIG"
  }
]
}
```

Source: Bitcoin and Cryptocurrency Technologies by Arvind Narayanan et al., https://goo.gl/3dK3Cs

Contents of a Transaction: Inputs

```
“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”: “7508e6ab259b4df0fd5147bab0c949d81473db4518f8afc5c3f52f91ff6b34e”,
     “n”: 0
   },
   “scriptSig”: “3f3a4ce81….”
                  }
                 ]
            }
```

Input 1:

Input 2:

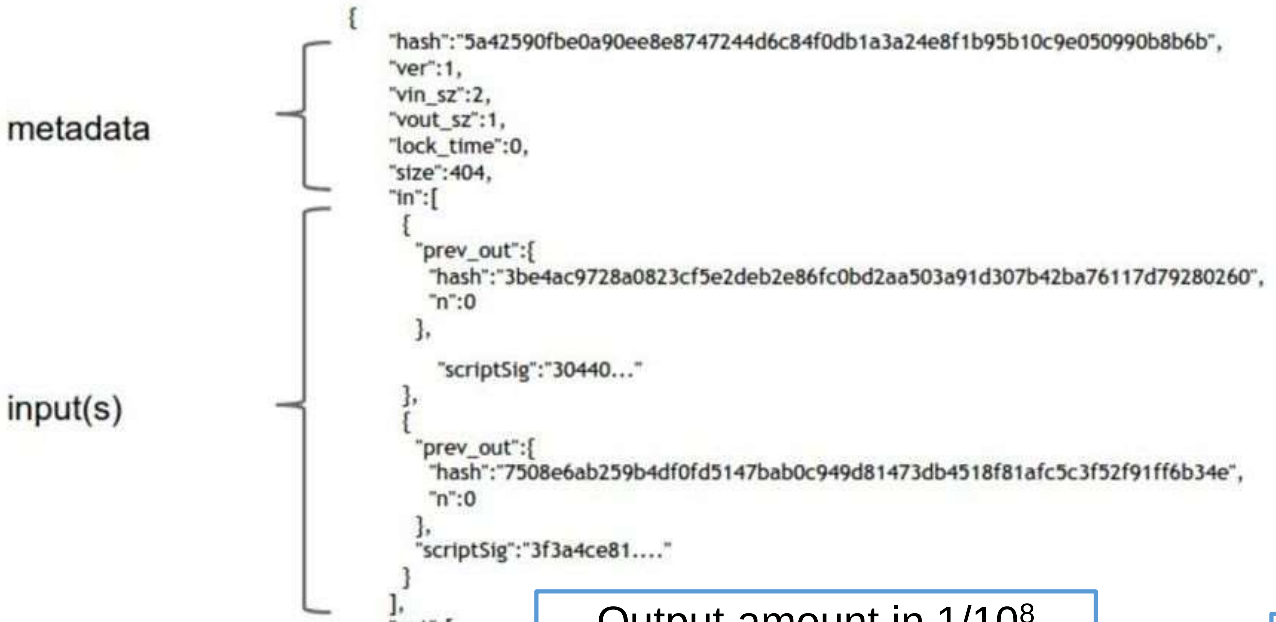
ID of previous transactions being referenced

index of input in previous transaction

signature used to redeem previous transaction output

Source: Bitcoin and Cryptocurrency Technologies by Arvind Narayanan et al., https://goo.gl/3dK3Cs

Contents of a Transaction: Outputs



metadata

input(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": 1012287097
    "scriptPubKey": "OP_DUP OP_HASH160 69e02e18b5705a05dd6b28ed517716c894b3d42e
                    OP_EQUALVERIFY OP_CHECKSIG
  },
}
```

Output amount in 1/10^8 Bitcoints (in "satoshi"s)

Output script

Type of script

Source: Bitcoin and Cryptocurrency Technologies by Arvind Narayanan et al., https://goo.gl/3dK3Cs

Bitcoin Scripts

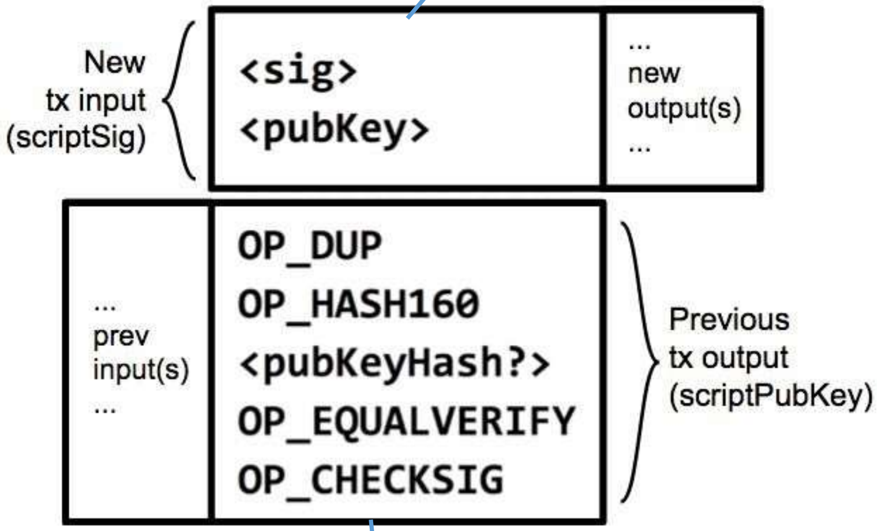
- Output specifications are scripts in a simple language
- Scripting allows for future extensibility of Bitcoin
- Script or “Bitcoin Scripting Language”
  - Max. 256 operations (incl. 75 reserved, 15 blocked)
  - No loops (=> not Turing-complete), but powerful ops
  - Stack based: inputs/outputs are put on stack, operations work on the top stack elements
- Conventions:
  - <data>: put “data” on top of stack
  - OP_...: perform operation OP_...

Pay-to-Pub-Key-Hash: Most Common Script

- "**scriptPubKey**": "OP_DUP OP_HASH160 69e02e18… OP_EQUALVERIFY OP_CHECKSIG
- Output part of a transfer/redeem script
  - Means: "This amount can be redeemed by the public key that hashes to address X, plus a signature from the owner of that public key"
- To redeem, we need **scriptSig** with two additional inputs (from a next, i.e. redeem-transaction):
  - <sig>: signature of the next (input) tx, signed by the receiver this tx
  - <pubKey>: public key of the receiver of this tx

Reedeming: scriptPubKey + scriptSig

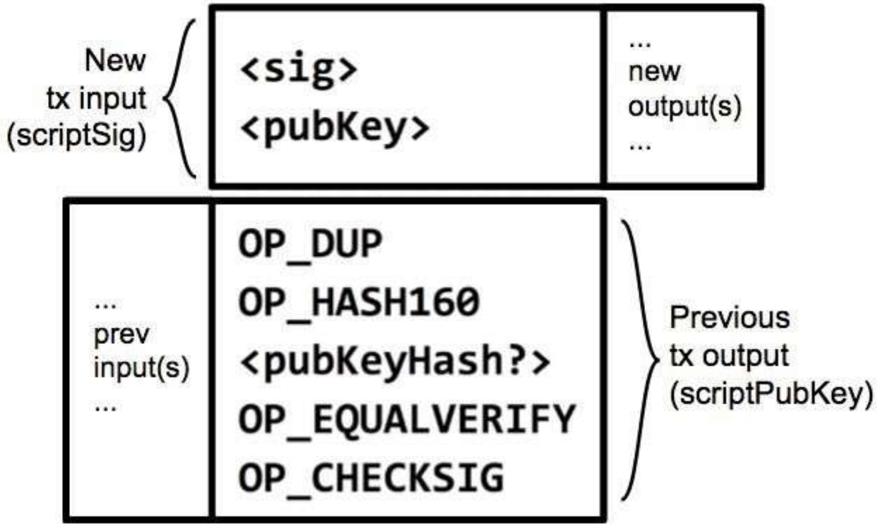
Input of a “next” tx:
P includes the previous output in this “next” tx, and confirms this by a signature



A: Output of a “previous” tx:
grants amounts to a pubKeyHash-owner P

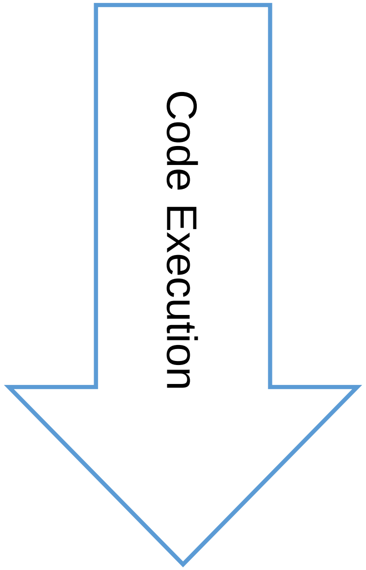
- A: locking script (**scriptPubKey**): found in previous transaction output, specifies requirements for redeeming transaction
- B: unlocking script (**scriptSig**): found in transaction input, provided by the spender to redeem the output of a previous transaction
- Bitcoin validating node will execute the locking and unlocking scripts in sequence

Execution Details



```
<sig>
<pubKey>
```

```
OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG
```



"scriptPubKey": "

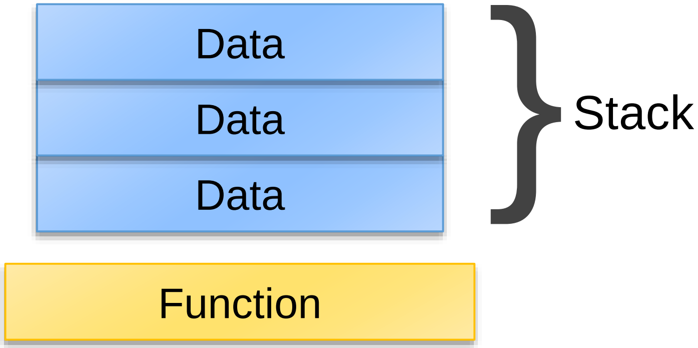
OP_DUP

OP_HASH160

69e02e18b5705a05d......

OP_EQUALVERIFY

OP_CHECKSIG"

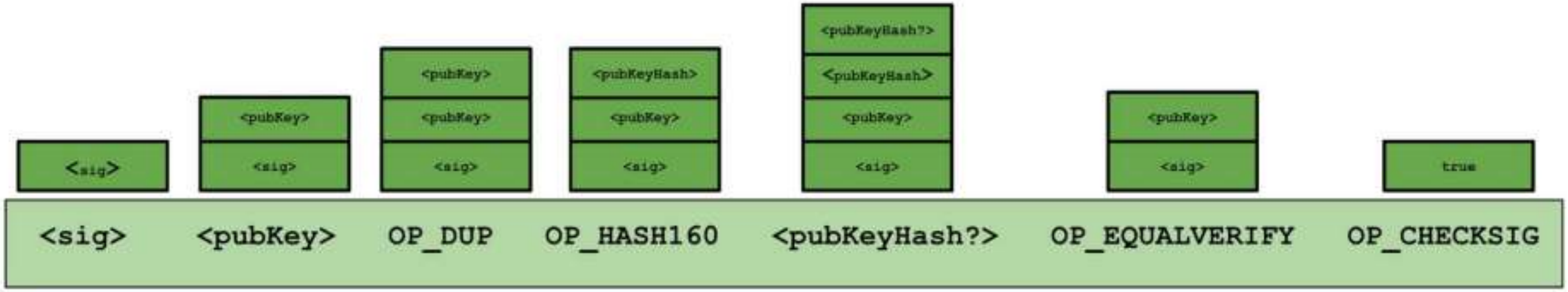


Conventions:

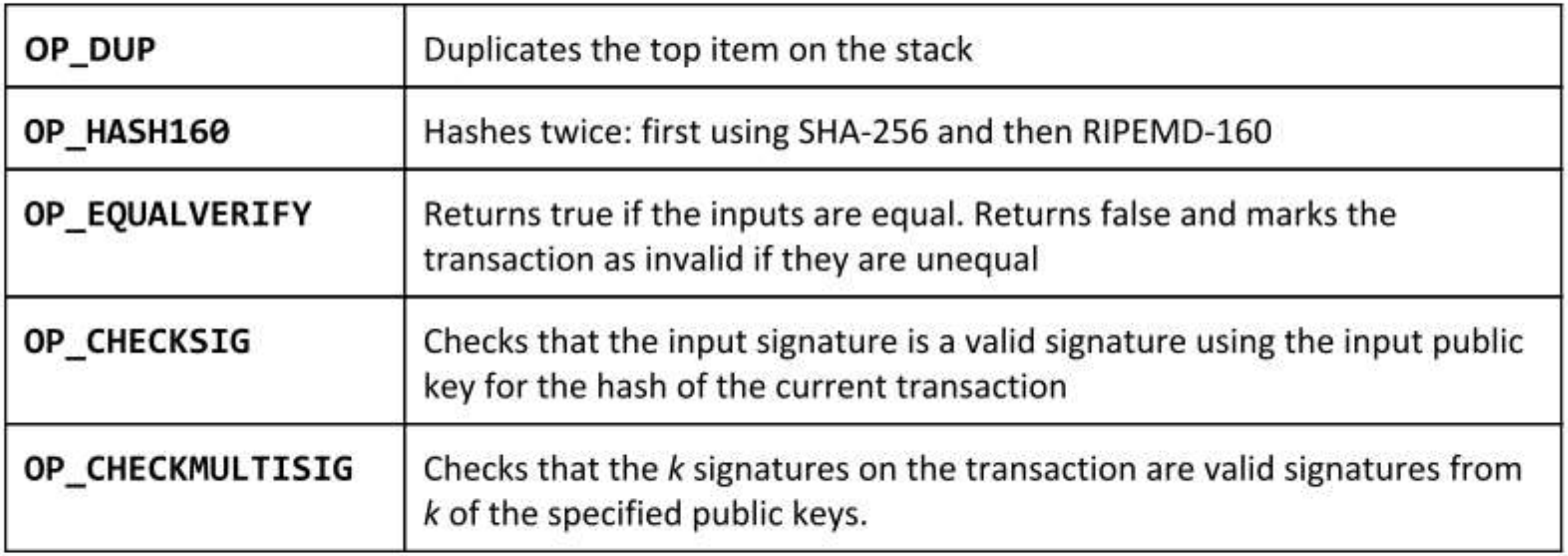
<data>: put "data" on top of stack

OP_....: perform operation OP_...

Operations Read and Write to Top of Stack

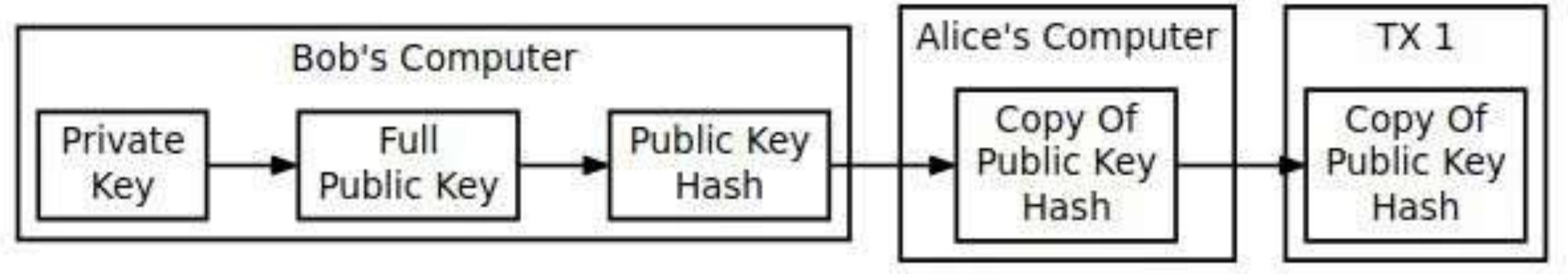


Some Common Script Operations

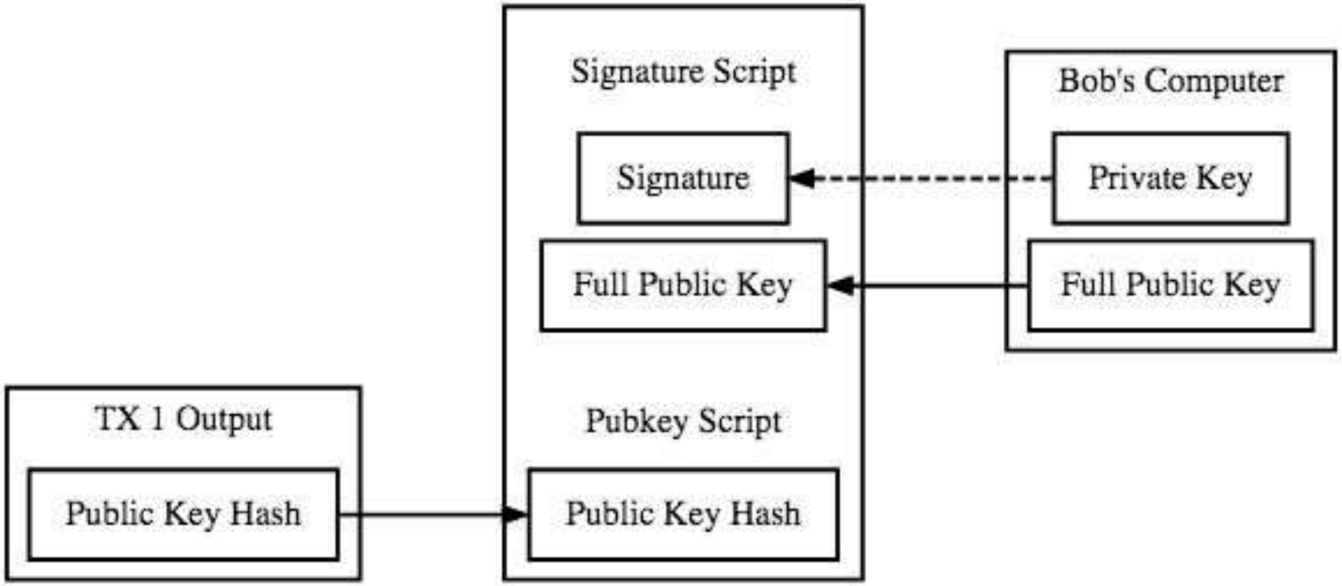


OP_DUP	Duplicates the top item on the stack
OP_HASH160	Hashes twice: first using SHA-256 and then RIPEMD-160
OP_EQUALVERIFY	Returns true if the inputs are equal. Returns false and marks the transaction as invalid if they are unequal
OP_CHECKSIG	Checks that the input signature is a valid signature using the input public key for the hash of the current transaction
OP_CHECKMULTISIG	Checks that the *k* signatures on the transaction are valid signatures from *k* of the specified public keys.

Pay-to-Pub-Key-Hash Example (A pays B)

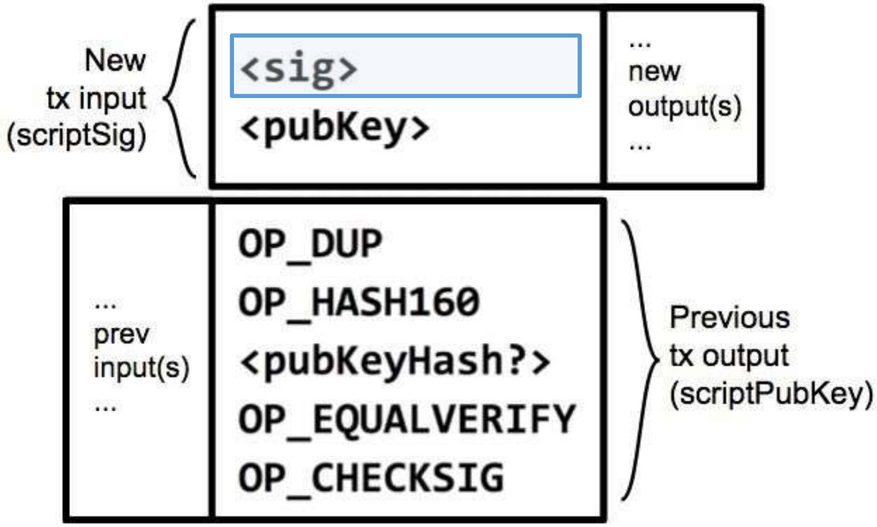


Creating A P2PKH Public Key Hash To Receive Payment



Spending A P2PKH Output

Pay-to-Pub-Key-Hash Example



New tx input (scriptSig)

<sig>
<pubKey>

… new output(s) …

… prev input(s) …

OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG

Previous tx output (scriptPubKey)

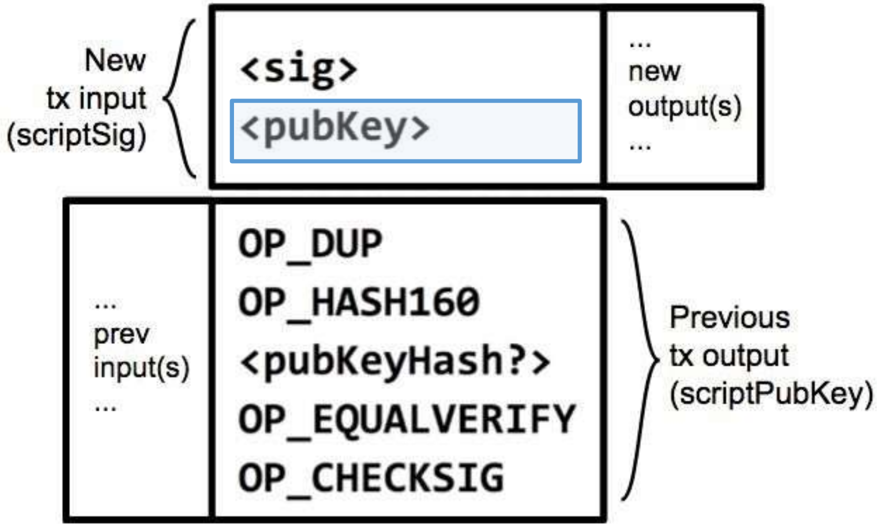
“next” tx

“previous” tx

<sig>

<sig>

Pay-to-Pub-Key-Hash Example



New tx input (scriptSig)

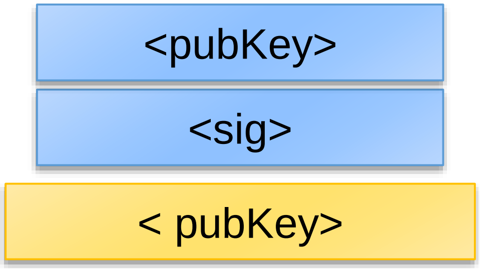
```
<sig>
<pubKey>
```

…
new
output(s)
…

…
prev
input(s)
…

```
OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG
```

Previous tx output (scriptPubKey)

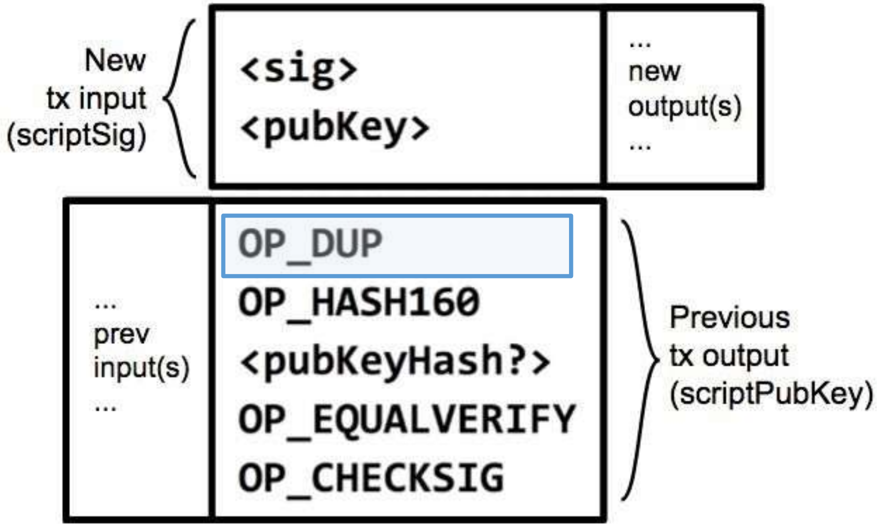


<pubKey>

<sig>

< pubKey>

Pay-to-Pub-Key-Hash Example



New tx input (scriptSig)

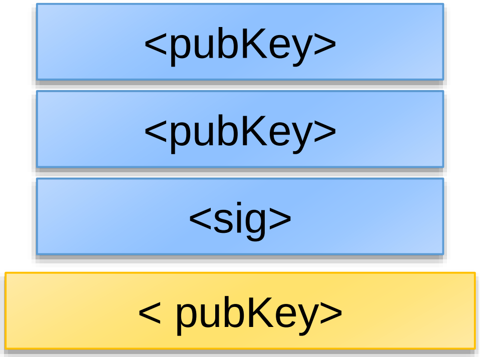
<sig>
<pubKey>

…
new output(s)
…

…
prev input(s)
…

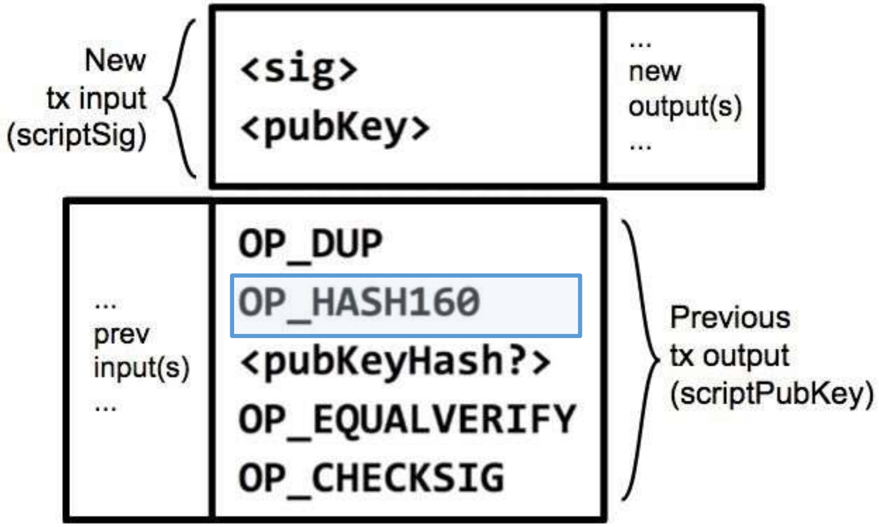
OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG

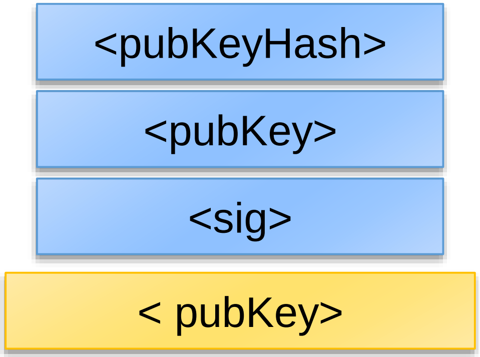
Previous tx output (scriptPubKey)



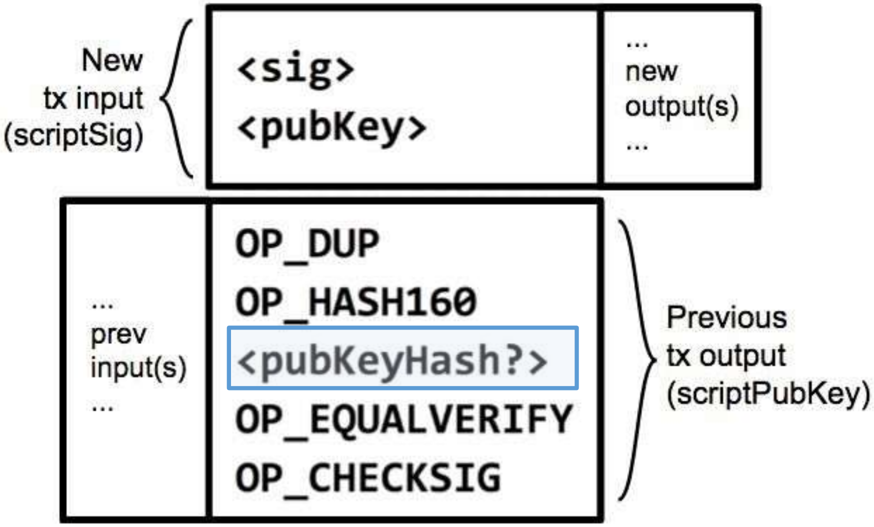
<pubKey>
<pubKey>
<sig>
< pubKey>

Pay-to-Pub-Key-Hash Example





Pay-to-Pub-Key-Hash Example



New tx input (scriptSig)

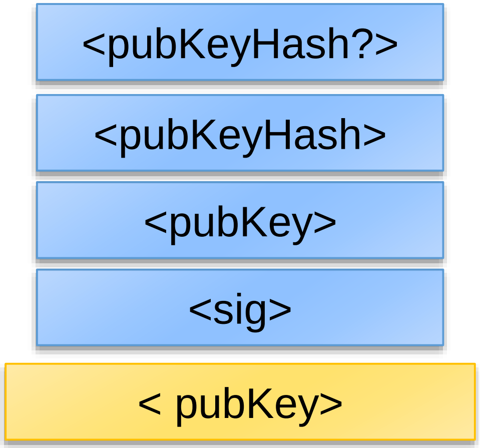
<sig>
<pubKey>

...
new
output(s)
...

...
prev
input(s)
...

OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG

Previous tx output (scriptPubKey)



<pubKeyHash?>

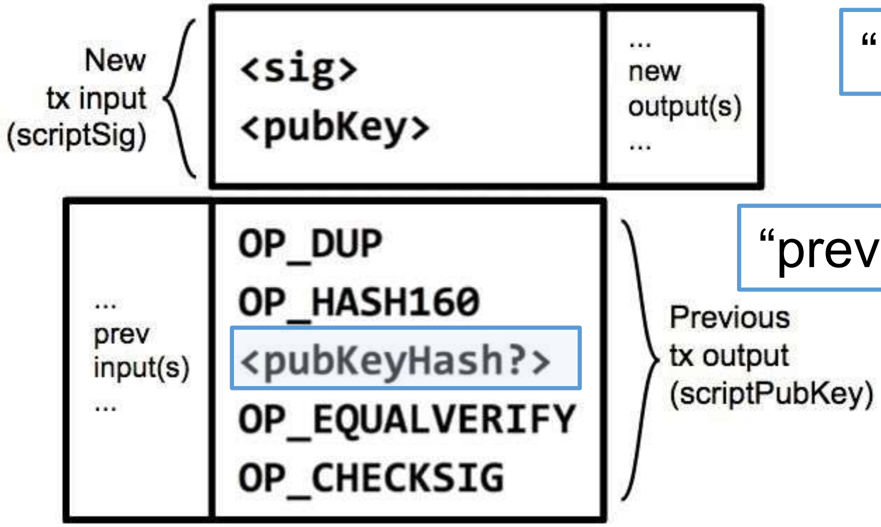
<pubKeyHash>

<pubKey>

<sig>

< pubKey>

Pay-to-Pub-Key-Hash Example



New tx input (scriptSig)

```
<sig>
<pubKey>
```

… new output(s) …

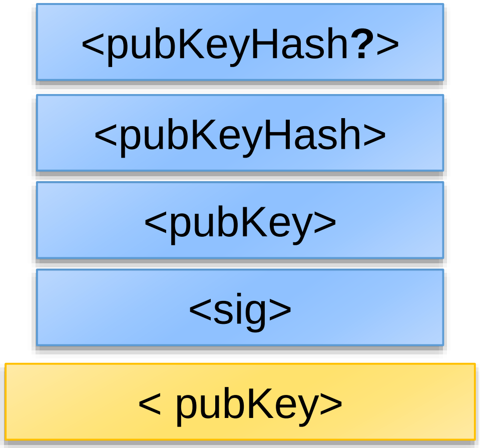
… prev input(s) …

```
OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG
```

Previous tx output (scriptPubKey)

“next” tx

“previous” tx



<pubKeyHash**?**>

<pubKeyHash>

<pubKey>

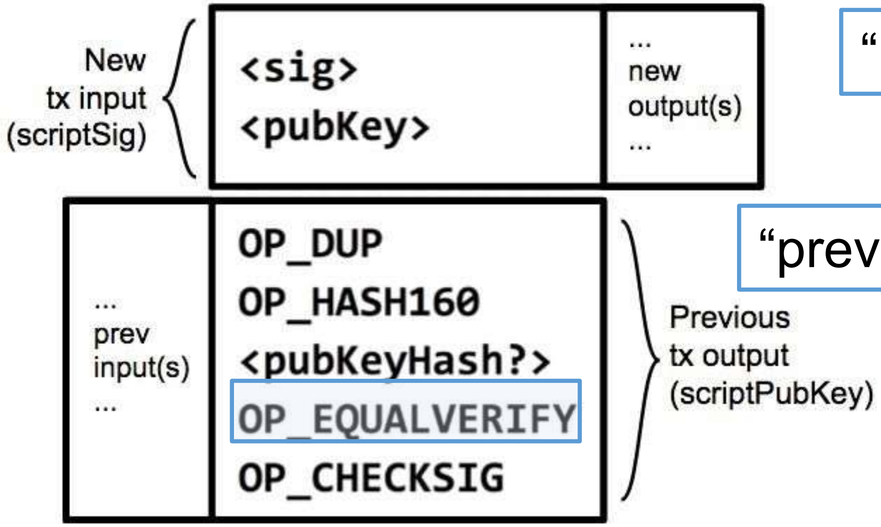
<sig>

< pubKey>

<pubKeyHash?> is the hash of a public key (of the BTC recipient R) that the sender S specified in the “previous” tx

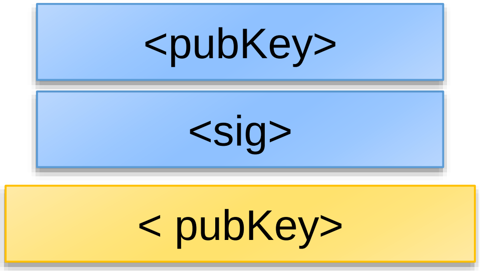
- the corresponding private key of R must be used to generate the signature to redeem these coins in the “next” tx

Pay-to-Pub-Key-Hash Example



“next” tx

“previous” tx


<pubKey>

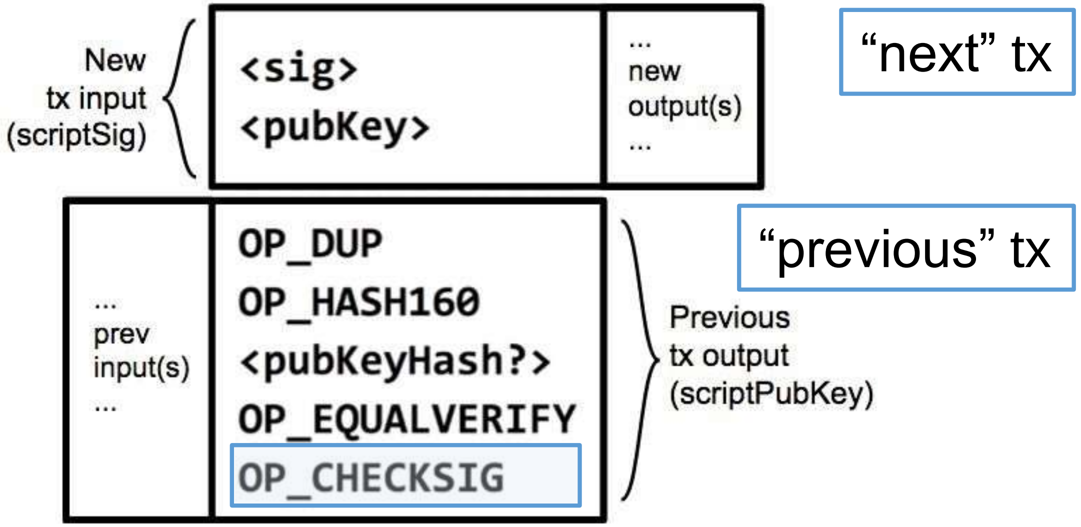
<sig>

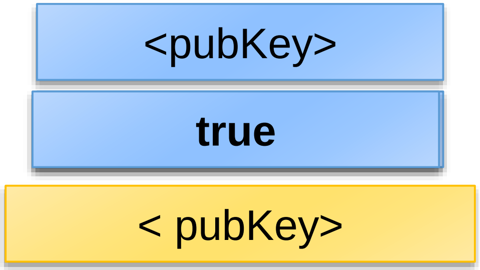
< pubKey>

EQUALVERIFY command checks that the two values at the top of the stack are equal

- If they aren’t, an error will be thrown, and the script will stop executing

Pay-to-Pub-Key-Hash Example

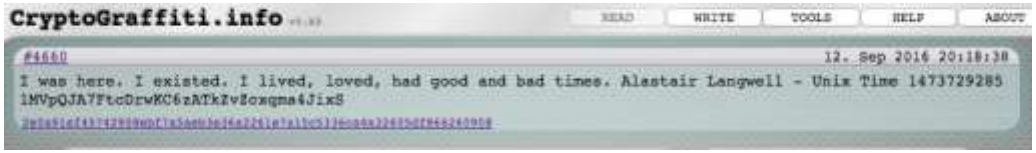




OP_CHECKSIG command checks if the signature (of the "next" tx) is valid

- It pops those two values off of the stack, and does the entire signature verification in one go

Proof of Burn



#4660 12. Sep 2016 20:18:38

I was here. I existed. I lived, loved, had good and bad times. Alastair Langwell - Unix Time 1473729285
1MVpQJA7FtcDrwKC6zATkZvZoxqma4JixS

[illegible]

Output script: OP_RETURN

<arbitrary data>

- How to write arbitrary data into the Bitcoin BC?
- Proof of Burn
  - OP_RETURN throws an error if reached
  - Output script can't be spent - you prove that you destroyed some currency
  - Anything after OP_RETURN is not processed, so arbitrary data can be entered
- Use cases
  - Prove existence of something at a particular point in time
  - Ex. A word you coined, hash of a document/music/creative works
  - Bootstrap altcoin by requiring that you destroy some Bitcoin to get altcoin

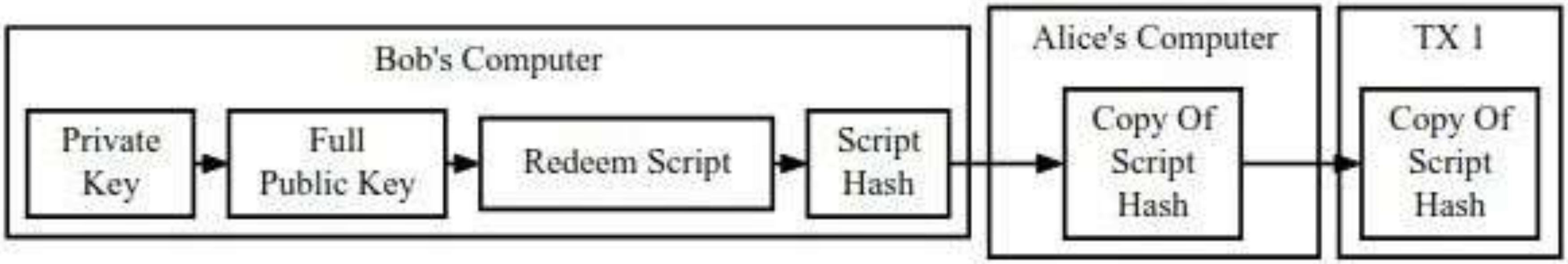
P2SH

Slides and content in part based on the course
BerkeleyX: CS198.1x “Bitcoin and Cryptocurrencies”
@ edX: https://courses.edx.org/courses/course-v1:BerkeleyX+CS198.1x+3T2018/course/

P2PKH vs. P2SH

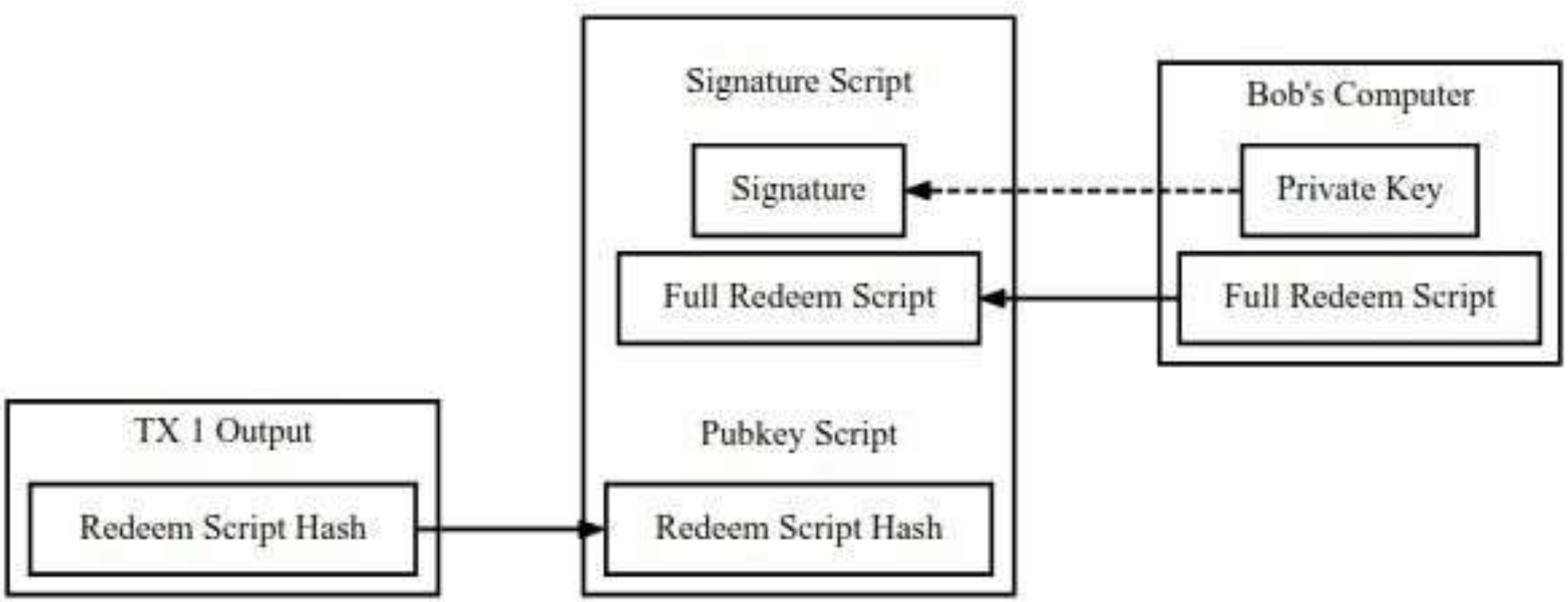
- In Bitcoin, senders specify a locking script, recipients provide an unlocking script
  - Pay-to-Pub-Key-Hash (**P2PKH**): Vendor (recipient of transaction) says “Send your coins to the hash of this Public Key.”
  - Simplest case, by far the most common case
- Pay-to-Script-Hash (**P2SH**): Vendor says “Send your coins to the hash of this Script; I will provide the script and the data to make the script evaluate to true when I redeem the coins.”
  - A vendor cannot say, “To pay me, write a complicated output script that will allow me to spend using multiple signatures.”

Creating a P2SH Redeem Script Hash



Creating A P2SH Redeem Script Hash To Receive Payment

Spending a P2SH Output

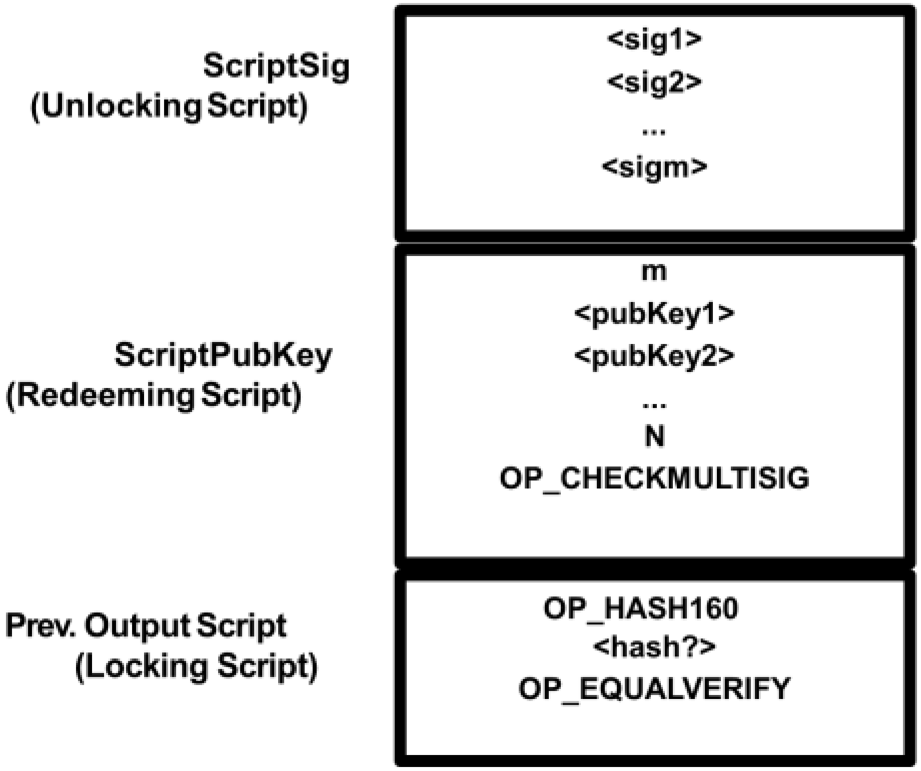


Spending A P2SH Output

Why P2SH?

- Offloads complicated script writing to recipients
- Makes more sense from a vendor-customer standpoint
  - Vendor (rather than customer) is responsible for writing correct and secure script
  - Customer doesn't care what the script actually is
- P2SH is the most important improvement to Bitcoin since inception
- Example: MultiSig
  - M of N specified signatures can redeem and spend the output of this transaction

Multisig Example



ScriptSig (Unlocking Script)

<sig1>
<sig2>
...
<sigm>

ScriptPubKey (Redeeming Script)

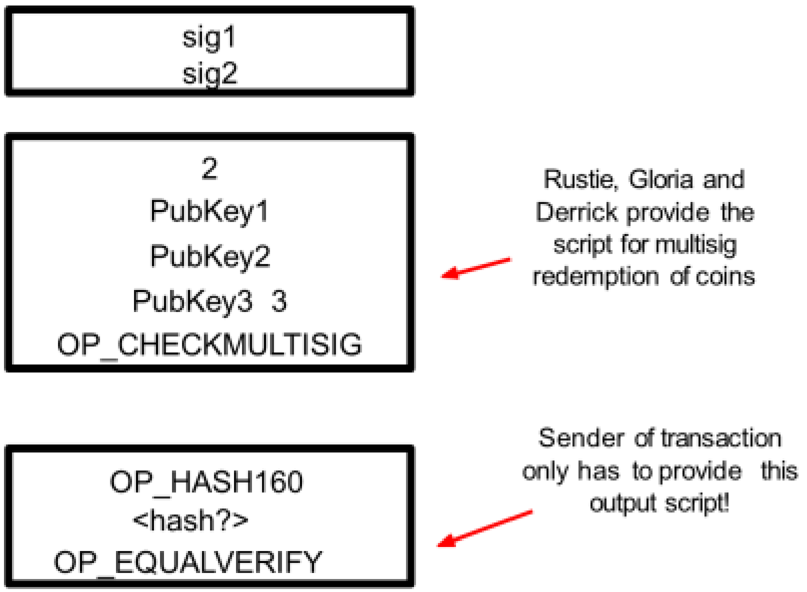
m
<pubKey1>
<pubKey2>
...
N
OP_CHECKMULTISIG

Prev. Output Script (Locking Script)

OP_HASH160
<hash?>
OP_EQUALVERIFY

e.g. Rustie, Gloria and Derrick are in charge of a joint account and make all spending decisions together:

2 of 3 MultiSig



sig1
sig2

2
PubKey1
PubKey2
PubKey3 3
OP_CHECKMULTISIG

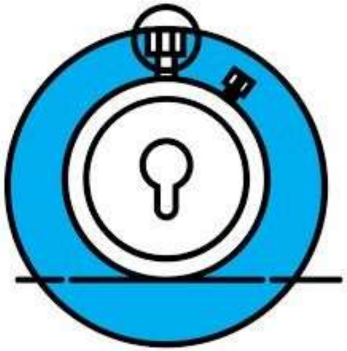
Rustie, Gloria and Derrick provide the script for multisig redemption of coins

OP_HASH160
<hash?>
OP_EQUALVERIFY

Sender of transaction only has to provide this output script!

Timelocks

- Extend bitcoin scripting into the dimension of time
- Absolute and relative timelocks
  - Absolute timelocks specify UNIX timestamp
  - Relative timelocks specify block height



- Transaction-level and script-level timelocks
  - Transaction-level: the transaction itself will be postponed until the specified time
  - UTXO-level: the locking script restricts use of specific UTXOs

Further Resources

- [Princeton Book] *Bitcoin and Cryptocurrency Technologies* by Arvind Narayanan et al., https://goo.gl/3dK3Cs
- *Mastering Bitcoin* by Andreas M. Antonopoulos
  - In Uni UB, free online from university domain
  - https://bitcoinbook.info/
  - As ASCII plus code examples
    - https://github.com/bitcoinbook/bitcoinbook/blob/develop/book.asciidoc or https://github.com/bitcoinbook
- The edX course: BerkeleyX, CS198.1x *Bitcoin and Cryptocurrencies*
  - https://courses.edx.org/courses/course-v1:BerkeleyX+CS198.1x+2T2018/course/

Thank you.

Additional Slides