Bitcoin Transactions

Zichao Yang

Zhongnan University of Economics & Law

Date: October 24, 2024

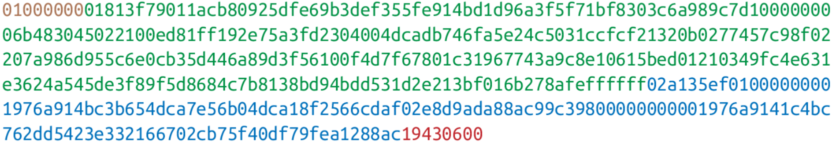
Transaction Components

A bitcoin transaction: explorer 1(concise), explorer 2(detailed)

At a high level, a transaction really only has four components:

- Version: indicates what additional features the transaction uses
- Inputs: define what bitcoins are being spent
- Outputs: define where the bitcoins are going
- Locktime: defines when this transaction starts being valid

Transaction Components



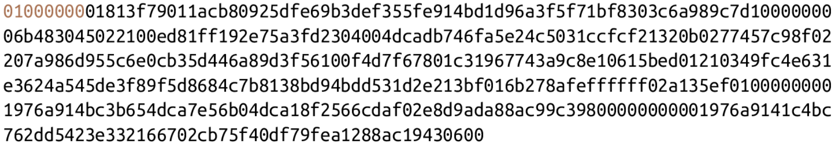
```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Figure 1: Transaction Components

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

The differently highlighted parts represent the version, inputs, outputs, and locktime, respectively.

Transaction Version



```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Figure 2: Version Component

In Bitcoin, the transaction version is generally 1, but there are cases where it can be 2. For example, transactions using an opcode called `OP_CHECKSEQUENCEVERIFY` as defined in BIP0112 require use of version > 1.

You may notice here that the actual value in hexadecimal is `01000000`, which does not look like 1. This is because the data is stored in the *little-endian* way.

Extra: Big-endian VS. Little-endian

1. Big-endian:
    - In this system, the most significant byte (MSB) is stored at the smallest memory address.
    - This is similar to how humans typically write numbers, with the most significant digits on the left.
2. Little-endian:
    - In this system, the least significant byte (LSB) is stored at the smallest memory address.
    - The bytes are stored in reverse order compared to big-endian.

Let us see how a 32-bit hexadecimal number `0x12345678` is stored in different systems.

Extra: Big-endian VS. Little-endian

Store a 32-bit hexadecimal number `0x12345678`...

(1) Big-endian encoding:

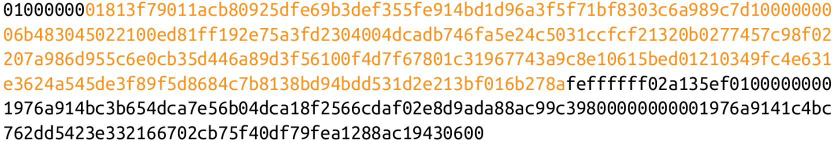
Memory address:	1000	1001	1002	1003
Data:	12	34	56	78

(2) Little-endian encoding:

Memory address:	1000	1001	1002	1003
Data:	78	56	34	12

In Bitcoin, both little-endian and big-endian encodings are used across different components, but there is no straightforward rule to determine where each encoding is applied.

Transaction Inputs



```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

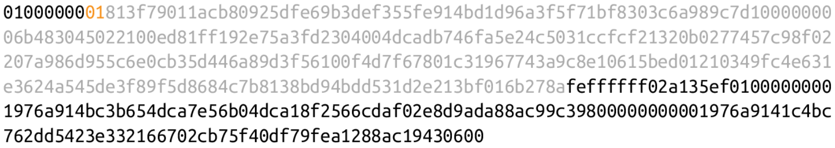
Figure 3: Inputs Component

Remember, Bitcoin's inputs are spending outputs of a previous transaction (i.e., UTXO).

Each input needs two things:

- A reference to bitcoins you received previously
- Proof that these are yours to spend

Transaction Inputs



```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Figure 4: Number of Inputs

The inputs field can contain more than one input. The number of inputs is the next part following the version component (in yellow).

In this case, the byte is `0x01`, which means 1 input in this transaction.

Bitcoin uses the **VarInt** (i.e., *variable integer*) encoding format to indicate how many inputs are included in a transaction.

VarInt

VarInt works by these rules:

- If the number is below 253, encode the number as a single byte (`0x00` to `0xfc` or 0 to 252 in decimal).
- If the number is between 253 and $2^{16} - 1$, we encode the number in 3 bytes. The number starts with `0xfd` (i.e., 253), then the real value is encoded in 2 bytes in little-endian (e.g., `0xfdff00` is 255).
- If the number is between 2^{16} and 2^{32}–1, we encode the number in 5 bytes. The number starts with `0xfe` (i.e., 254), then the real value is encoded in 4 bytes in little-endian (e.g., `0xfe7f110100` is 70,015).
- If the number is between 2^{32} and 2^{64}–1, we encode the number in 9 bytes. The number starts with `0xff` (i.e., 255), then the real value is encoded in 8 bytes in little-endian (e.g., `0xff6dc7ed3e60100000` is 18,005,558,675,309).

VarInt

Try to answer the following questions:

Q1: Why do we need this VarInt encoding format in Bitcoin?

Q2: Why do we need to attach the prefix `0xfd`, `0xfe`, or `0xff` to the real number?

Q3: Can you figure out the reason why `0xfdff00` is 255? Shouldn't `0xff00` be 65,280 in decimal?

VarInt

A1: The idea behind Bitcoin's VarInt encoding is to optimize storage by using as few bytes as possible for smaller numbers.

A2: The prefix is used to instruct the system to locate the actual value in the next 2, 4, or 8 bytes, optimizing memory usage by avoiding the need to store small numbers inefficiently.

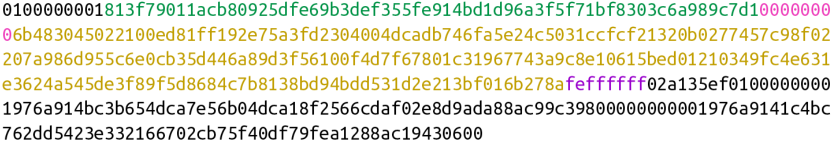
A3: The two bytes `0xff00` represent the value in little-endian order. In little-endian, the least significant byte comes first, meaning `0xff00` should be interpreted as `0x00ff` (i.e., 255 in decimal).

Fields of an Input

Now, let us move to the next part inside the input component: inputs.

Each input contains four fields. The first two fields point to the previous transaction output and the last two fields define how the previous transaction output can be spent:

- Previous transaction ID
- Previous transaction index, points to the UTXO
- ScriptSig
- Sequence



```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Figure 5: Fields of an Input

Fields of an Input

The **previous transaction ID** is the hash256 of the previous transaction's contents.

As each transaction may have multiple outputs, we rely on **previous transaction index** to locate the UTXO we will spend.

Previous transaction ID is 32 bytes and previous transaction index is 4 bytes, both in little-endian format.

The **ScriptSig** relates to Bitcoin's smart contract language, **Script**, UTXO owners use it to prove their ownership. And the ScriptSig field is a variable-length field, not a fixed-length field.

Flawed High-frequency Trades Idea

What about the last field: the sequence?

The **sequence**, combined with the **locktime** field, was originally intended as a way to do what Satoshi called “high-frequency trades”:

The trade transaction would start with sequence at 0 and with a far-away locktime (say, 500 blocks from now, so valid in 500 blocks).

After the first transaction, where Alice pays Bob x bitcoins, the sequence of each input would be 1. After the second transaction, where Bob pays Alice y bitcoins, the sequence of each input would be 2.

Using this method, we could have lots of payments compressed into a single on-chain transaction as long as they happened before the locktime became valid.

Flawed High-frequency Trades Idea

Q1: why does Satoshi think that Bitcoin needs a new method to process high-frequency trade?

Q2: unfortunately, it turns out that it's quite easy for a miner to cheat in these "high-frequency trades". Can you see why?

Flawed High-frequency Trades Idea

A1: because Bitcoin's transaction capacity is limited, we will go back to this point when we discuss blocks in Bitcoin.

A2: for example, if Bob is a miner, he could ignore the updated trade transaction with sequence number 2 and mine the trade transaction with sequence number 1, cheating Alice out of y bitcoins.

A much better design was created later with "payment channels", which is the basis for the **Lightning Network**.

Transaction Inputs

Two additional points worth mentioning here:

1. The amount of bitcoins spent in each input is not specified. We have no idea how much is being spent unless we look it up in the blockchain for the transaction(s) that we're spending.

2. We don't know if the transaction is unlocking the right UTXO without first knowing about the previous transaction. Every miner must independently verify that this transaction unlocks the right UTXO and that it doesn't spend nonexistent bitcoins.

Transaction Outputs

Outputs define where the bitcoins are going. Each transaction must have one or more outputs.

Like with inputs, output serialization starts with how many outputs there are as a **VarInt**:

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

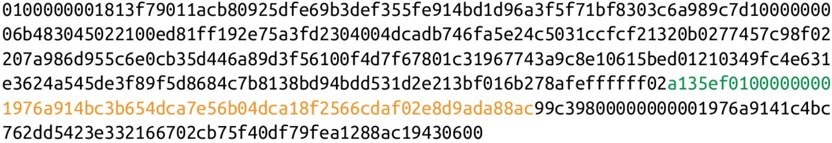
Figure 6: Number of Outputs

Fields of an Output

Each output has two fields: **amount** and **ScriptPubKey**.

The **amount** is the amount of bitcoins being assigned to this output and is specified in satoshis, or $\frac{1}{10^8}$ of a bitcoin.

The **ScriptPubKey**, like the **ScriptSig**, has to do with Bitcoin's smart contract language, **Script**. It's like a one-way safe that can receive deposits from anyone, but can only be opened by the owner of the safe.



```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

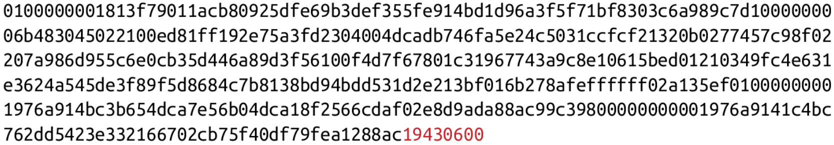
Figure 7: Fields of an Output

Locktime

Locktime was originally construed as a way to do high-frequency trades. A transaction with a locktime of 600,000 cannot go into the blockchain until block 600,001. Locktime & Sequence

If the locktime is greater than or equal to 500,000,000, it's a Unix timestamp. If it's less than 500,000,000, it's a block number.

BIP0065 introduced `OP_CHECKLOCKTIMEVERIFY`, which makes locktime more useful by making an output unspendable until a certain locktime.



```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Figure 8: Locktime

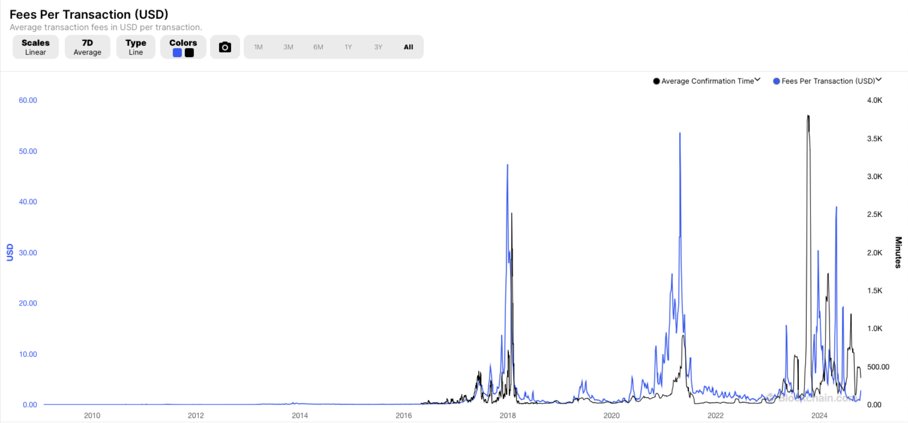
Transaction Fee

For any non-coinbase transactions, the sum of the inputs has to be greater than or equal to the sum of the outputs.

The **transaction fee** is simply the sum of the inputs minus the sum of the outputs. This difference is what the miner gets to keep.

Transaction fee serves as an incentive for miners to include transactions in blocks.

Transaction Fee



Fees Per Transaction (USD)

Average transaction fees in USD per transaction.

Scales Linear

7D Average

Type Line

Figure 9: Fees Per Transaction (USD)

Source: blockchain.com

Coinbase Transaction

Coinbase is the required first transaction in every Bitcoin block and is the only mechanism that pumps newly minted bitcoins into the system.

The coinbase transaction's outputs are kept by whomever successfully mined the block, and are called *coinbase reward*. Coinbase reward and transaction fees constitute the *block reward*.

Until today, the majority of the block reward has come from the coinbase reward.

The initial coinbase reward was 50 bitcoins. Coinbase reward halves after every 210,000 blocks are mined (also called **Bitcoin halving**). The latest Bitcoin halving happened on April 20, 2024 at the block height of 840,000, and the current reward is 3.125 bitcoins.

Coinbase Transaction

```
0100000001000000000000000000000000000000000000000000000000000000000000000000fffffff
f5e03d71b07254d696e656420627920416e74506f6f6c20626a31312f4542312f4144362f43205914
293101fabe6d6d678e2c8c34afc36896e7d9402824ed38e856676ee94bfdb0c6c4bcd8b2e5666a040
0000000000000c7270000a5e00e00ffffffff01faf20b58000000001976a914338c84849423992471
bffb1a54a8d9b1d69dc28a88ac00000000

        - 01000000 - version
        - 01 - # of inputs
        - 000...00 - previous tx hash
        - ffffffff - previous tx index
        - 5e0...00 - ScriptSig
        - ffffffff - sequence
        - 01 - # of outputs
        - faf20b58...00 - output amount
        - 1976...ac - p2pkh ScriptPubKey
        - 00000000 - locktime
```

Figure 10: Coinbase Transaction

The coinbase transaction structure is no different from that of other transactions with a few exceptions:

1. Coinbase transactions must have exactly one input.
2. The one input must have a previous transaction of 32 bytes of 00.
3. The one input must have a previous index of ffffffff.

ScriptSig in Coinbase

The bitcoins rewarded by coinbase transactions do not come from previous outputs, and miners do not need to prove that they are the owners of these bitcoins. Then what is in the ScriptSig?

It turns out that, aside from a few restrictions, the ScriptSig can be anything the miner chooses, as long as its evaluation on its own is valid:

- the ScriptSig has to be at least 2 bytes and no longer than 100 bytes.
- the ScriptSig should follow BIP0034, which regulates the first element of the ScriptSig.

Let's check out the genesis block's coinbase transaction created by Satoshi!

Transaction Data Structure

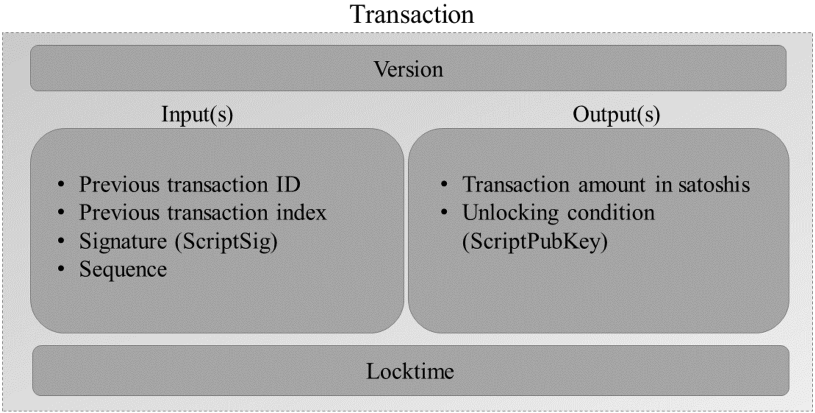

Transaction
Version
Input(s)
Output(s)
• Previous transaction ID
• Previous transaction index
• Signature (ScriptSig)
• Sequence
• Transaction amount in satoshis
• Unlocking condition (ScriptPubKey)
Locktime

Figure 11: Transaction Data Structure

Next, we explore two purposely overlooked fields: `ScriptSig` and `ScriptPubKey`, and discuss the scripting language used in Bitcoin.

Script

The conditions under which a UTXO can be used are written and verified in Script.

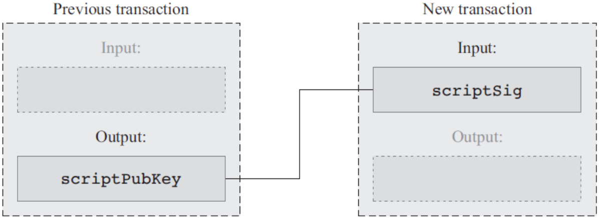


Figure 12: `ScriptSig` and `ScriptPubKey` in Transaction

Source: Schar, Fabian, and Aleksander Berentsen. *Bitcoin, Blockchain, and cryptoassets: a comprehensive introduction*. MIT press, 2020.

Script

Script is a scripting language that has a predefined list of operations: so-called *OP codes*.

Script is a stack-based language. When a function is evaluated, it places the corresponding value on top of the stack. If a function requires arguments, these in turn are taken from the topmost positions of the stack. (*last in, first out*)

Script is very simple and has been deliberately designed to not be Turing-complete, it does not support loops.

Turing Incomplete

Why is Bitcoin designed to be Turing incomplete?

- prevent vicious denial-of-service (Dos) attacks: a single Script program with an infinite loop could take down the whole Bitcoin network.
- simplicity for financial transactions: smart contracts with Turing completeness are very difficult to analyze, and are prone to create unintended behaviors and cause bugs.

Alternative approach: Ethereum and Solidity programming language

How Script Works

Scripts contain two types of commands: elements and operations.

Elements are data. A typical element can be a DER signature or a SEC pubkey.

Operations dictate how to process the data. They consume zero or more elements from the stack and push zero or more elements back to the stack. [List of OP codes]

After all the commands are evaluated, the top element of the stack must be nonzero for the script to resolve as **valid**.

Having no elements in the stack or the top element being 0 would resolve as **invalid**.

Example Operations

A typical operation is `OP_DUP`, which will duplicate the top element (consuming 0) and push the new element to the stack (pushing 1).

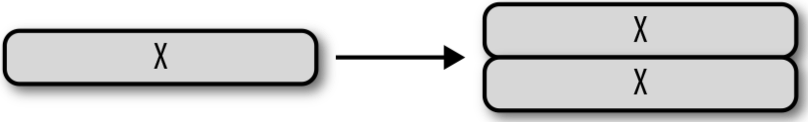


Figure 13: `OP_DUP` duplicates the top element

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Let's code it out in python!

Example Operations

One important operation is OP_HASH160, which does a SHA-256 followed by a RIPEMD-160 (recall: both are hash functions) to the top element of the stack (consuming 1) and pushes a new element to the stack (pushing 1).

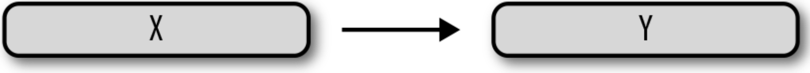


Figure 14: OP_HASH160 does a sha256 followed by ripemd160 to the top element

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Let's code it out in python!

Example Operations

Another important operation is `OP_CHECKSIG`.

It consumes two elements from the stack, the first being the pubkey and the second being a signature, and examines whether the signature is good for the given pubkey. If so, `OP_CHECKSIG` pushes a 1 to the stack; otherwise, it pushes a 0 to the stack.

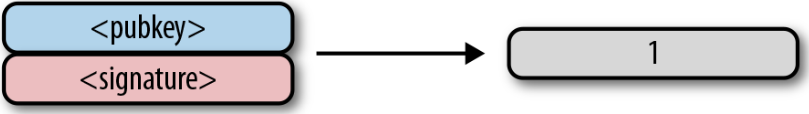


Figure 15: `OP_CHECKSIG` checks if the signature for the pubkey is valid or not
Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Let's check out the code in python!

Standard Scripts

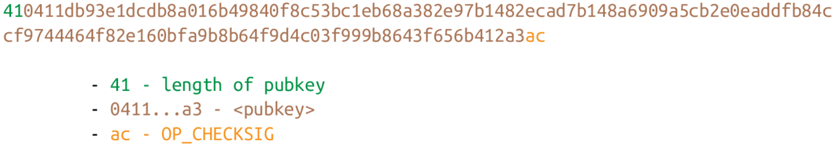
There are many types of standard scripts in Bitcoin, including the following:

1. **p2pk**: Pay-to-pubkey
2. **p2pkh**: Pay-to-pubkey-hash
3. **p2sh**: Pay-to-script-hash
4. **p2wpkh**: Pay-to-witness-pubkey-hash
5. **p2wsh**: Pay-to-witness-script-hash

p2pk: Pay-to-pubkey

Pay-to-pubkey (p2pk) was used largely during the early days of Bitcoin.

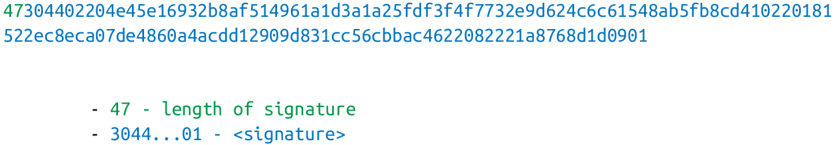
In p2pk, bitcoins are sent to a **public key**, and the owner of the private key can unlock or spend the bitcoins by creating a signature.



```
410411db93e1dcdb8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5cb2e0eaddfb84c
cf9744464f82e160bfa9b8b64f9d4c03f999b8643f656b412a3ac

        - 41 - length of pubkey
        - 0411...a3 - <pubkey>
        - ac - OP_CHECKSIG
```

Figure 16: Pay-to-pubkey (p2pk) ScriptPubKey



```
47304402204e45e16932b8af514961a1d3a1a25fdf3f4f7732e9d624c6c61548ab5fb8cd410220181
522ec8eca07de4860a4acdd12909d831cc56cbbac4622082221a8768d1d0901


        - 47 - length of signature
        - 3044...01 - <signature>
```

Figure 17: Pay-to-pubkey (p2pk) ScriptSig

How does p2pk work?

The ScriptPubKey and ScriptSig combine to make a command set that looks like the *p2pk combined* below.

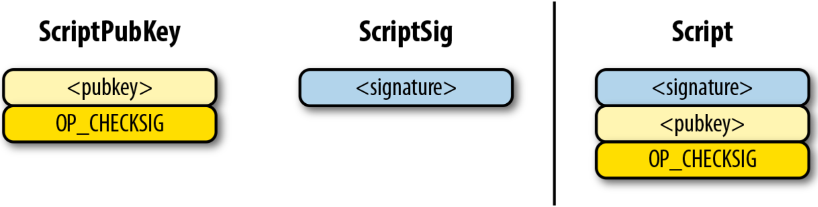


Figure 18: p2pk combined

How does p2pk work?

The two columns in the following *p2pk start* are **Script commands** and the **elements stack**.

At the end of the processing, the top element of the stack must be nonzero to be considered a valid ScriptSig. The Script commands are processed one at a time.

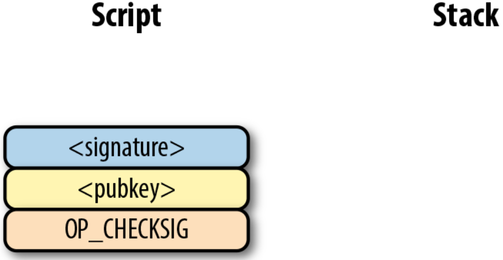


Figure 19: p2pk start

How does p2pk work?

Step 1: the first command is the signature, which is an element. This is data that is pushed to the stack.

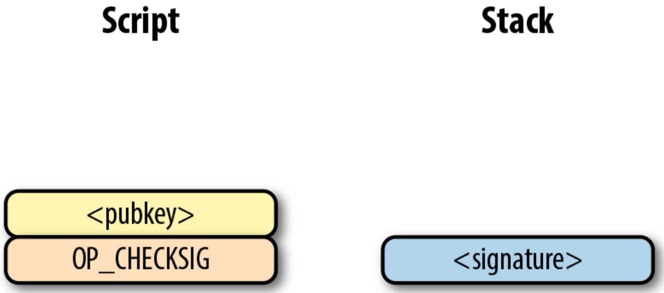


Figure 20: p2pk step 1

How does p2pk work?

Step 2: the second command is the pubkey, which is also an element. Again, this is data that is pushed to the stack.

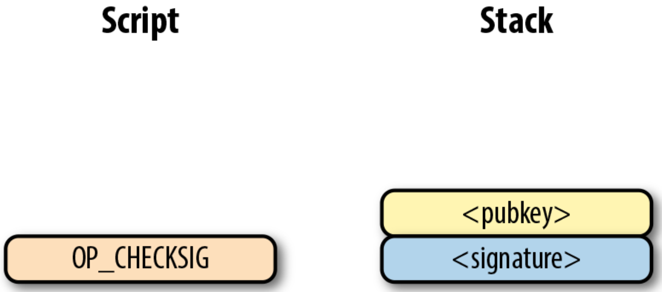


Figure 21: p2pk step 2

How does p2pk work?

Step 3: `OP_CHECKSIG` consumes two stack commands (pubkey and signature). `OP_CHECKSIG` will push a 1 to the stack if the signature is valid, and a 0 if not.

Now we end up with a single element on the stack. Since the top element is nonzero (1 is definitely not 0), this script is valid.

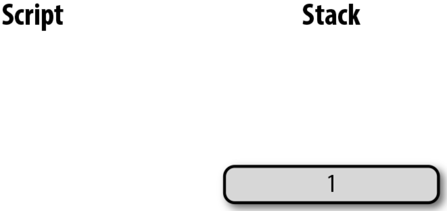


Figure 22: p2pk step 3

Problems with p2pk

1. The public keys are long. The secp256k1 public points are 33 bytes in compressed SEC and 65 bytes in uncompressed SEC format.
2. IP addresses were queried for the public keys in p2pk, which is not secure and prone to man-in-the-middle attacks.
3. The length of the public keys inflates the UTXO set, and requires more resources on the full nodes' end.
4. Public keys are stored in the ScriptPubKey field, and are exposed to the whole world. Should ECDSA someday be broken, these UTXOs could be stolen.

p2pkh: Pay-to-pubkey-hash

Pay-to-pubkey-hash (p2pkh) is an alternative script format that has two key advantages over p2pk:

1. The addresses are shorter.
2. It's additionally protected by sha256 and ripemd160.

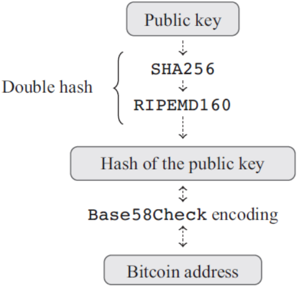
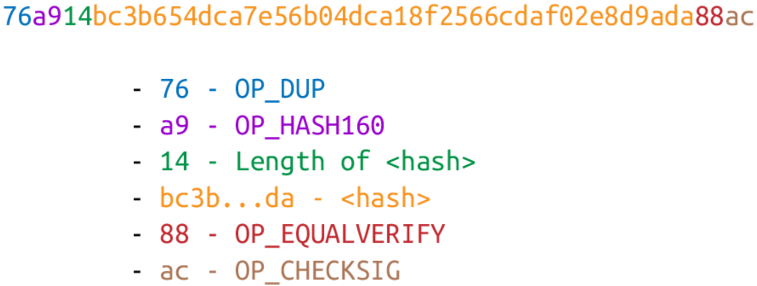


Figure 23: From the Public Key to the Bitcoin Address

p2pkh: ScriptPubKey



```
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac

- 76 - OP_DUP
- a9 - OP_HASH160
- 14 - Length of <hash>
- bc3b...da - <hash>
- 88 - OP_EQUALVERIFY
- ac - OP_CHECKSIG
```

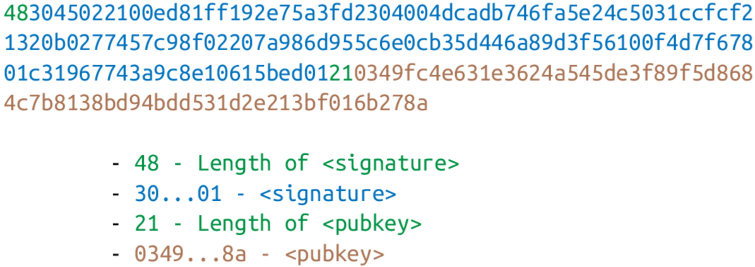
Figure 24: p2pkh: ScriptPubKey

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Like *p2pk*, `OP_CHECKSIG` is here and `OP_HASH160` makes an appearance.

Unlike *p2pk*, the SEC pubkey is not here, but a 20-byte hash is. There is also a new opcode here: `OP_EQUALVERIFY`.

p2pkh: ScriptSig



```
483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf2
1320b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f678
01c31967743a9c8e10615bed01210349fc4e631e3624a545de3f89f5d868
4c7b8138bd94bdd531d2e213bf016b278a

        - 48 - Length of <signature>
        - 30...01 - <signature>
        - 21 - Length of <pubkey>
        - 0349...8a - <pubkey>
```

Figure 25: p2pkh: ScriptSig

Source: Song, Jimmy. *Programming bitcoin: Learn how to program bitcoin from scratch.* O'Reilly Media, 2019.

Like *p2pk*, the ScriptSig has the DER signature. Unlike *p2pk*, the ScriptSig also has the SEC pubkey.

The main difference between *p2pk* and *p2pkh* ScriptSigs is that the SEC pubkey has moved from the ScriptPubKey to the ScriptSig.

How does p2pk work?

The ScriptPubKey and ScriptSig combine to form a list of commands that looks like the following *p2pkh combined*.

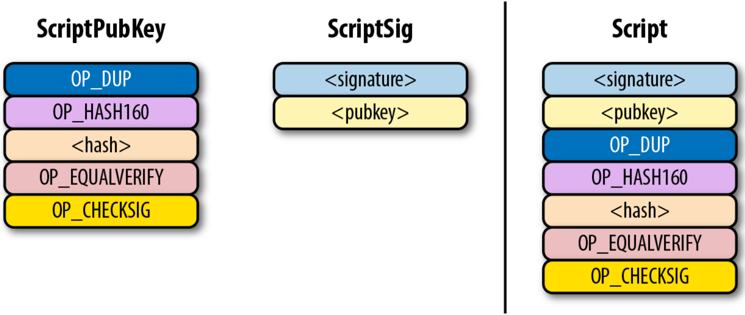


Figure 26: p2pkh combined

How does p2pk work?

We start with the commands as combined in the following plot.

Step 1: the first two commands are elements, so they are pushed to the stack.

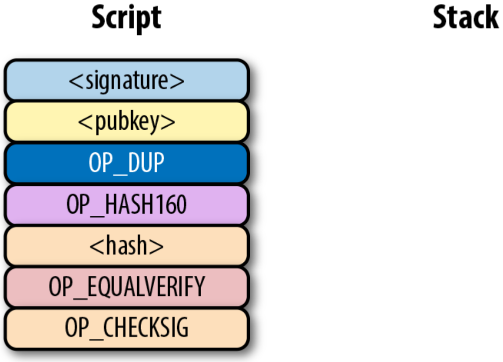


Figure 27: p2pkh step 1

How does p2pk work?

Step 2: `OP_DUP` duplicates the top element, so the pubkey gets duplicated.

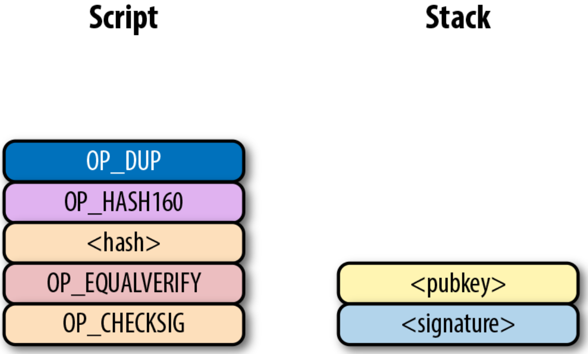


Figure 28: p2pkh step 2

How does p2pk work?

Step 3: `OP_HASH160` takes the top element and performs the hash160 operation on it (sha256 followed by ripemd160), creating a 20-byte hash.

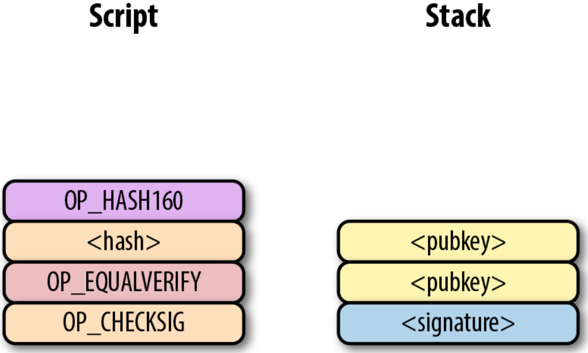


Figure 29: p2pkh step 3

How does p2pk work?

Step 4: The 20-byte hash is an element and is pushed to the stack.

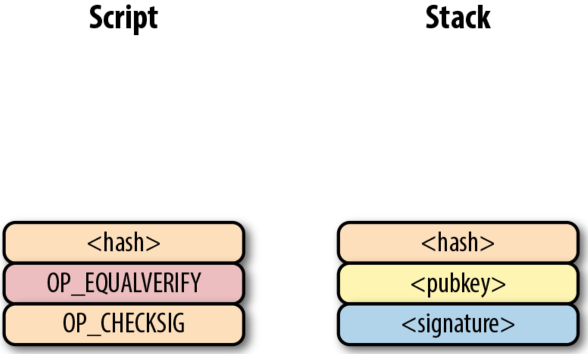


Figure 30: p2pkh step 4

How does p2pk work?

Step 5: `OP_EQUALVERIFY` consumes the top two elements and checks if they're equal. If they are equal, the script continues execution. If they are not equal, the script stops immediately and fails.

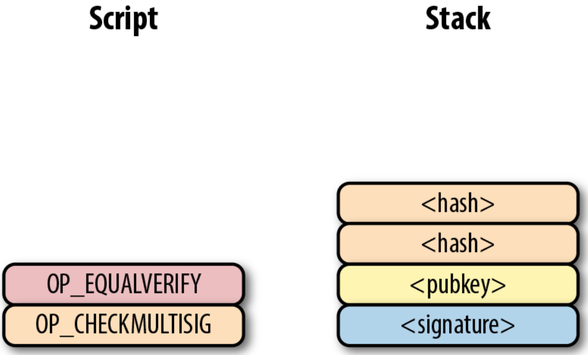


Figure 31: p2pkh step 5

How does p2pk work?

Step 6: `OP_CHECKSIG` consumes two stack commands (pubkey and signature). `OP_CHECKSIG` will push a 1 to the stack if the signature is valid, and a 0 if not.

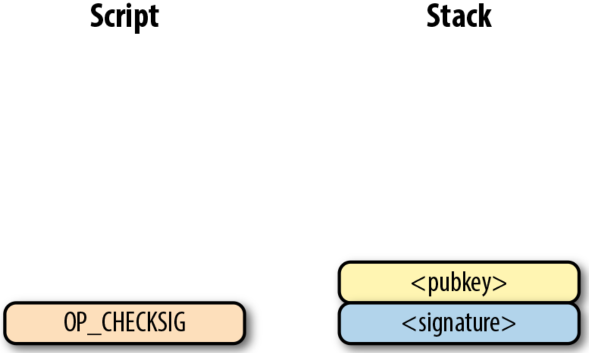


Figure 32: p2pkh step 6

How does p2pk work?

Step 7: we assume that the signature is valid, so eventually the stack is left with 1.

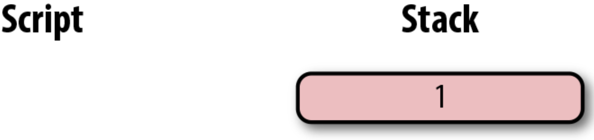


Figure 33: p2pkh step 7

p2pkh: ScriptPubKey

There are two ways the *p2pkh* script can fail:

1. If the ScriptSig provides a public key that does not hash160 to the 20-byte hash in the ScriptPubKey, the script will fail at `OP_EQUALVERIFY` (step 5).

2. If the ScriptSig has a valid public key, but has an invalid signature. `OP_CHECKSIG` will push a 0, ending in failure (step 6).

The major advantages of *p2pkh* are:

1. The ScriptPubKey is shorter (just 25 bytes)

2. A thief would not only have to solve the discrete log problem in ECDSA, but also figure out a way to find preimages of both ripemd160 and sha256.