Bitcoin script - #06

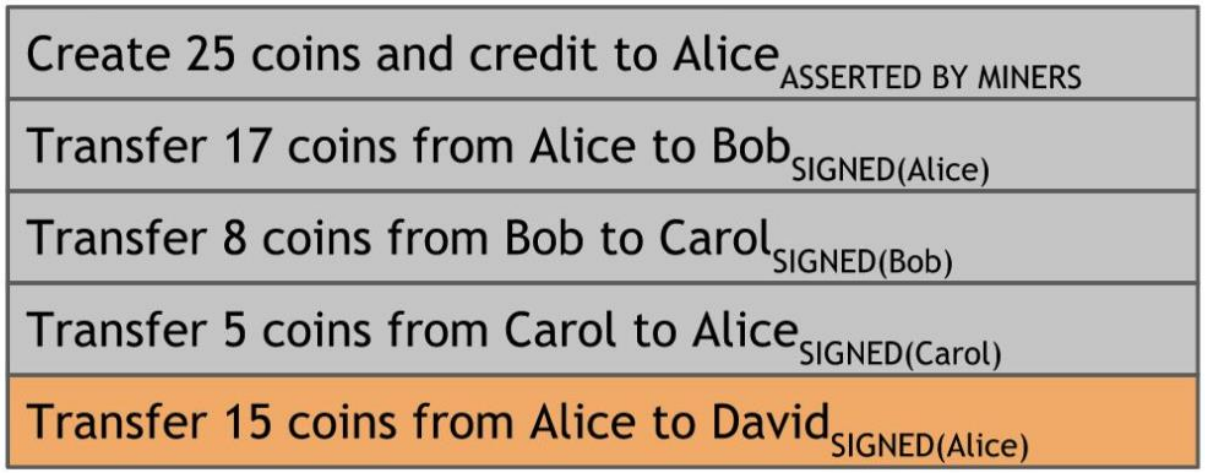
Blockchain and Distributed Ledger technologies

SAPIENZA
UNIVERSITÀ DI ROMA

Massimo La Morgia
massimo.lamorgia@uniroma1.it

Bitcoin transactions

Suppose individual transactions are added to the ledger one at a time.



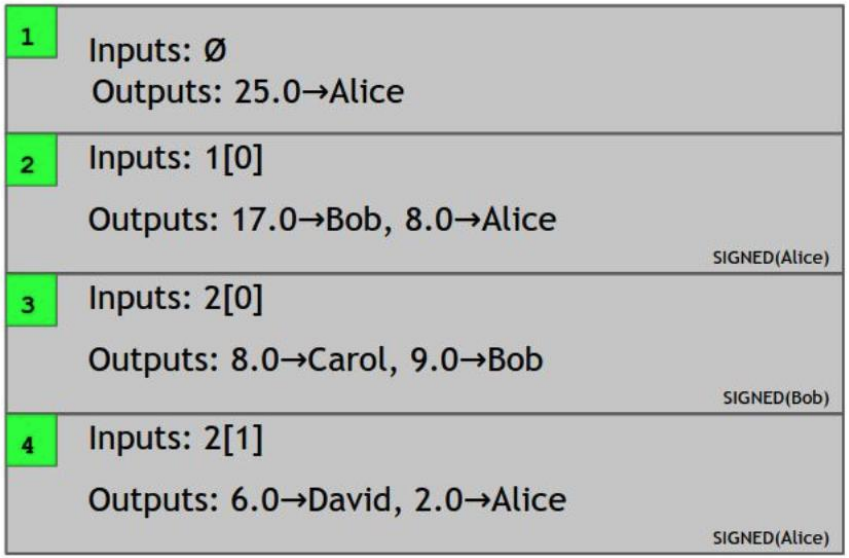
To determine if a transaction is valid, we need of an account based model and keep track of the balance of the accounts.

It is inefficient if we want to know if a user want to spent some money, because we have to look all the transactions that affected the account.

Otherwise, we need to use additional data structure that require a lot of extra housekeeping besides the ledger itself.

UTXO model

Every bitcoin transaction creates outputs that can be consumed as inputs in future transactions. UTXOs are simply the transaction outputs that have not been consumed yet and can still be used for spending.

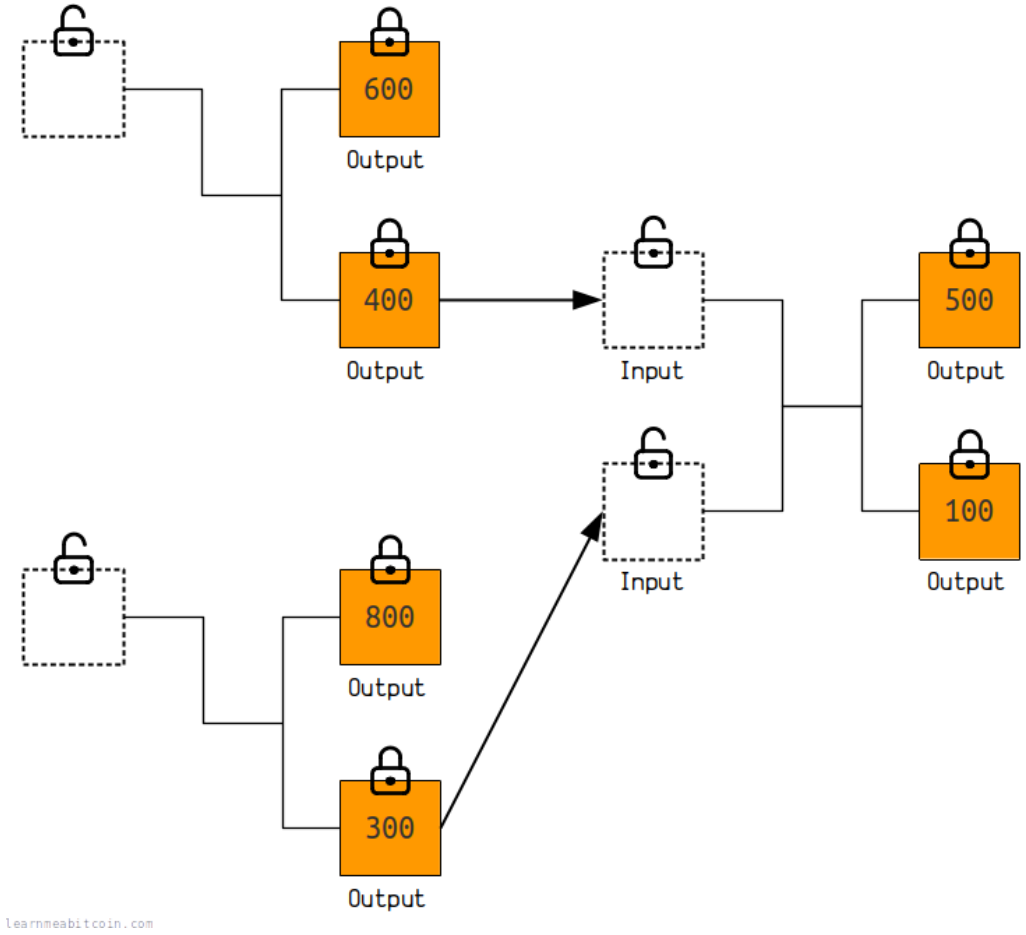


When your node receives a new transaction from the network, it needs to validate that all of its inputs are referencing outputs that have not already been spent.

If the transaction's inputs are all unspent outputs (UTXOs), then the transaction is valid.
if the transaction is trying to spend an output that has already been spent in a previous transaction, then the transaction is invalid and will be rejected.

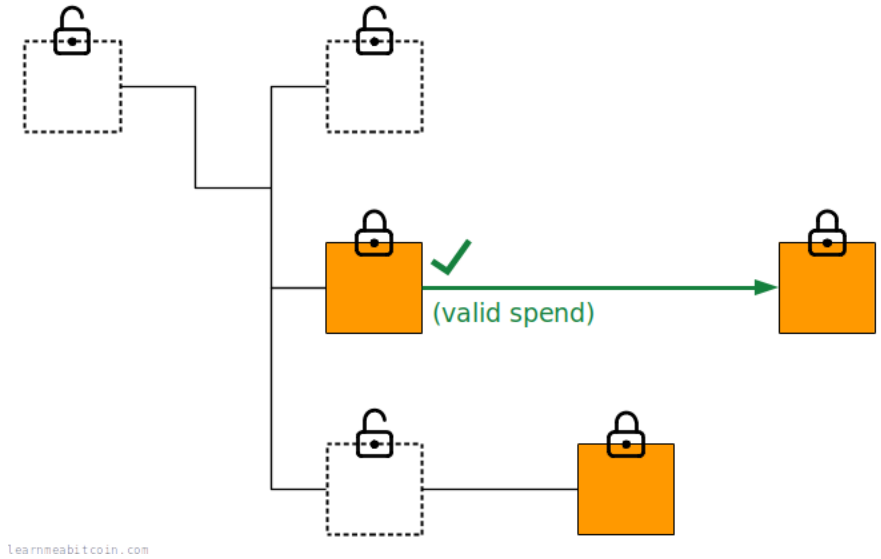
UTXO model

An input is an output that is being spent in a transaction. It's basically a reference to a previously unspent output, along with some unlocking code that unlocks the output so that it can be spent.



UTXO model

When a node receives a new transaction from the network, it needs to validate that all of its inputs are referencing outputs that have not already been spent.



The "balance" of an address is the sum of all the UTXOs locked to that address.

In Bitcoin Core, all of the UTXOs are stored in the chainstate database.

This is a separate database that gets stored in memory (RAM), which makes it faster to access than having to trawl through the raw blockchain files to check if an output has been spent or not.

Bitcoin transactions

A bitcoin transaction is just a bunch of data that unlocks and locks up batches of bitcoins.

You can have multiple inputs in a transaction. This is because you sometimes need to consume multiple outputs (as inputs) to be able to create the **total amount** you want to send in the transaction.

However, the more inputs you have, the bigger the transaction is going to be (in bytes), which will make it more expensive as transactions are selected for mining based on the size of the fee per the amount of space they take up in a block.

So when you make a bitcoin transaction, you want to strategically select the fewest inputs that will allow you to create the total amount you want to send.

Consolidating transactions: it is a self-transfer that aggregates multiple UTXOs into a single, larger output. It is useful because, each UTXO must be spent in full and included as a transaction input, increasing the size and cost of future transactions. By combining these fragments into one UTXO during periods of low network fees, users can minimize future transaction costs.

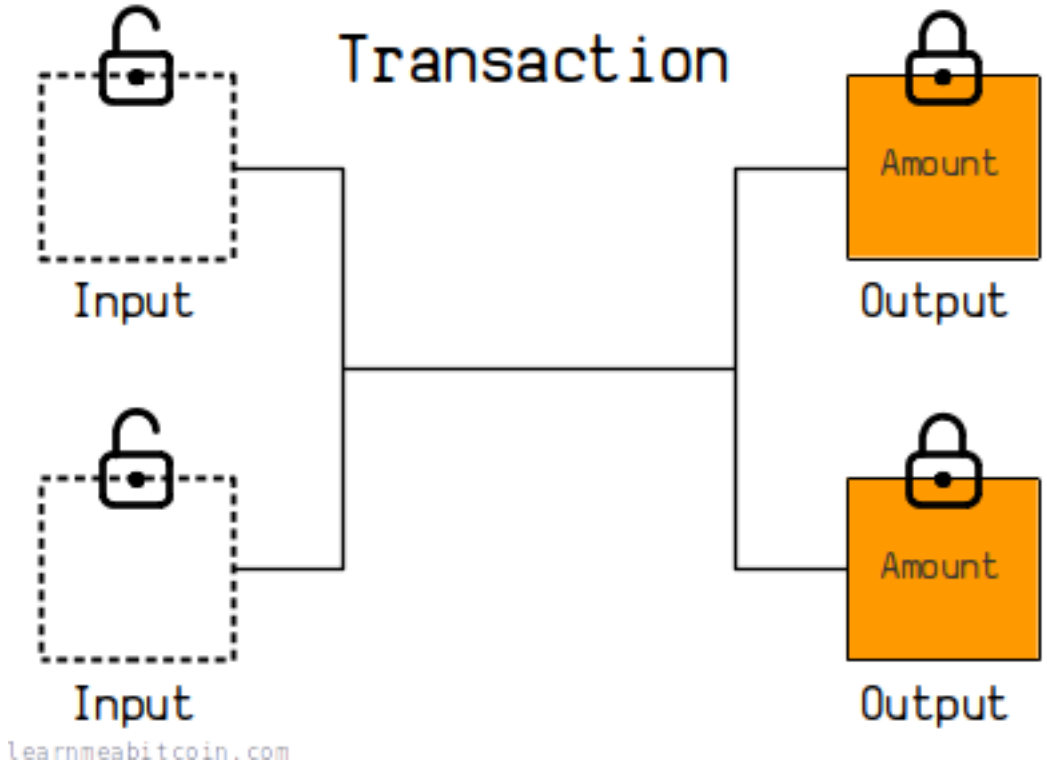
Bitcoin transactions

Change address: A change address is an output address, generated and controlled by the sender's wallet, used to return any remaining value when the total of the selected UTXOs exceeds the amount required for a transaction.

Joint payment: a joint payment is a cooperative Bitcoin transaction where multiple participants contribute inputs and collectively authorize a single transaction, improving efficiency and privacy.

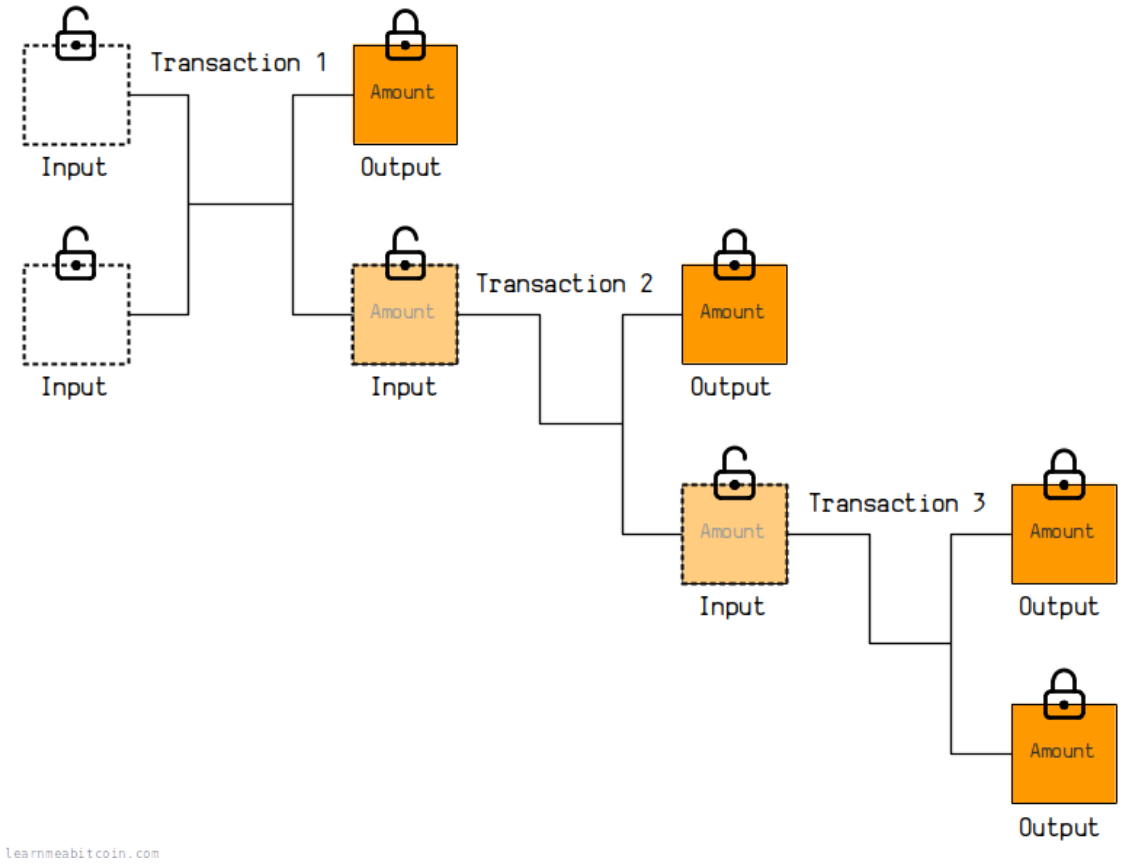
Bitcoin transactions

A transaction selects existing batches of bitcoins (**inputs**) and unlocks them.



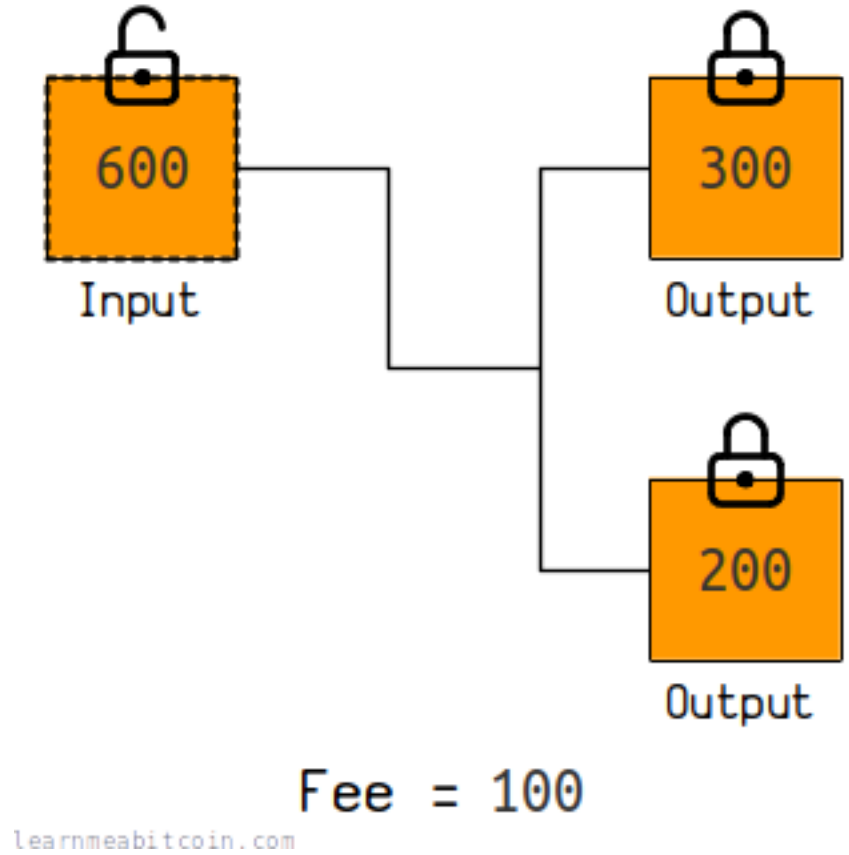
Bitcoin transactions

You can think of a transaction as being part of a chain of outputs; one transaction creates an output, and then a future transaction selects that output (as an input) and unlocks it to create new outputs.



UTXO model FEE

A transaction fee is the remainder of a transaction.

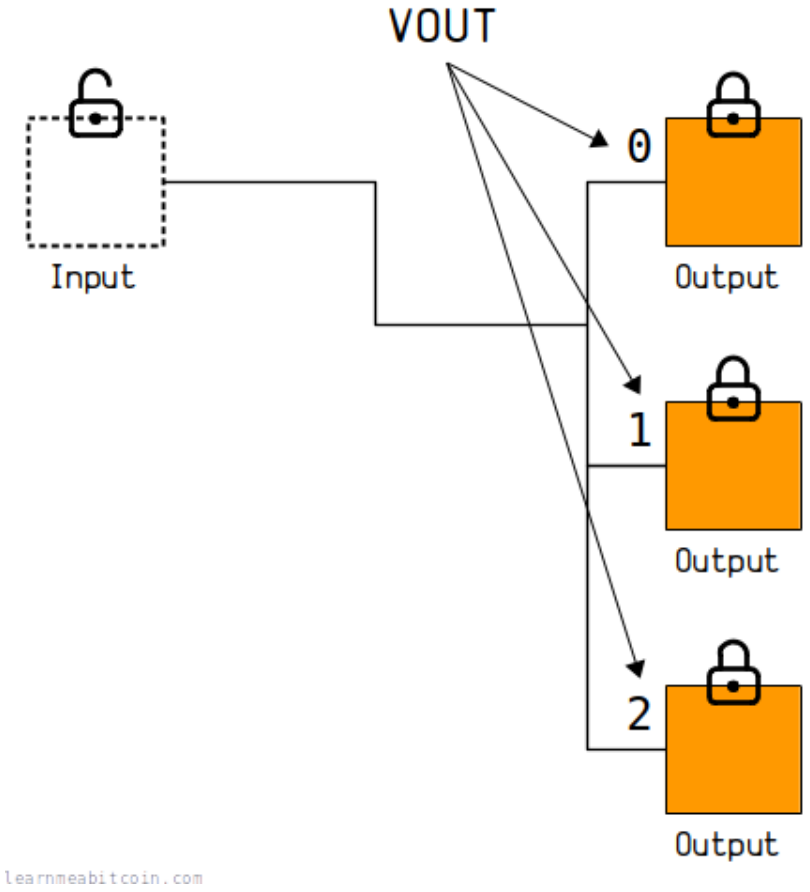


Bitcoin transactions – 2 input


```
{
  "version": "02000000",
  "inputcount": "02",
  "inputs": [
    {
      "txid": "e925fec58184fe99a4789993b62963b0813851793350b1aec99f3e75466225a5",
      "vout": "19000000",
      "scriptsigsize": "6a",
      "scriptsig":
"473044022046e23b8f6b749a15b0571848fe2b86bfbe7e158b23f37fb0335be0a71f48a5dd022076640b2d39659c26224c9e67101b34e
1394a38ff08a4bdd88d7503f63e54adc5012103b3e19c8169b81d15ec21f9d0f3ed4b2f18c7c9e1149ce5f7ddebed7732724b9f",
      "sequence": "ffffffff"
    },
    {
      "txid": "ca45253a9f908429e09986454e83dfaab6ea30502356078477c81f57967a5964",
      "vout": "1f000000",
      "scriptsigsize": "6b",
      "scriptsig":
"483045022100f39a8b1f91e5293448a05c642362e510a718d1fc04e781a3778a51a1abb968a8022031ce768c5ff01c4899f24f8600d67
777b3672c9e44ed5b1377672cfe89ef354b012103b3e19c8169b81d15ec21f9d0f3ed4b2f18c7c9e1149ce5f7ddebed7732724b9f",
      "sequence": "ffffffff"
    }
  ],
  "outputcount": "01",
  "outputs": [
    {
      "amount": "6742750000000000",
      "scriptpubkeysize": "17",
      "scriptpubkey": "a91484504805e56355c8cb8fc06032ce352920edfc0687"
    }
  ],
  "locktime": "00000000"
}
```

Bitcoin transactions (input)– VOUT

A VOUT (vector output) is an index number for a transaction output.



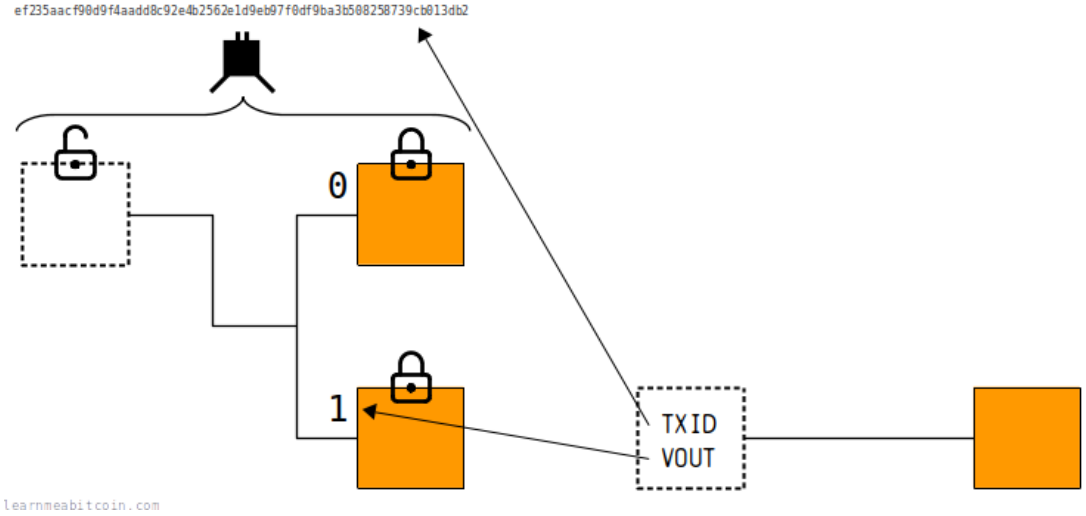
Bitcoin transactions (input) - TXID

A TXID (Transaction ID) is a unique reference for a bitcoin transaction.

A TXID is created by hashing the transaction data.

They are used in the following situations:

1. Searching for transactions, you typically use TXIDs to look up specific transactions on a blockchain explorer or from your own local node.
2. Referencing outputs from previous transactions for use as inputs when you create a bitcoin transaction.
3. TXIDs are used to create the merkle root for the block header.



Bitcoin transactions (input)- ScripSig

ScritpsigSize: The size in bytes of the upcoming ScriptSig.

A ScriptSig provides the unlocking code for a previous output.

Each output in a transaction has a locking code (**ScriptPubKey**) placed on it.
So when you come to select one as an input in a future transaction, you need to supply an unlocking code (**ScriptSig**) so that it can be spent. This locking/unlocking code uses a mini-programming language called **Script**.

Bitcoin transactions (input) - Sequence

The **sequence** field has control over the "finality" of a transaction, as in whether a transaction is in its "final" state before it gets mined into a block. If it's not in its "final" state, then it's possible for it to be replaced before it ends up in the blockchain.

These are the most common settings:

- =0xFFFFFFFF — Transaction is final.

- <=0xFFFFFFFE — Locktime.

This setting enables the transaction's locktime field to be used.

- <=0xFFFFFFFD — Replace By Fee (RBF).

This setting enables the RBF feature, which allows you to replace a transaction with a higher-fee one if it's still in the mempool.

- <=0xEFFFFFFF — Relative Locktime.

This setting allows you to set a locktime on the transaction relative to when the output being spent was mined.

0x00000000 to 0x0000FFFF — Blocks. Set the relative locktime as a number of blocks.
0x00400000 to 0x0040FFFF — Time. Set the relative locktime as a number of seconds.

Bitcoin transactions - Locktime

The locktime field allows you to prevent a transaction from being mined until after a specific block height or time.

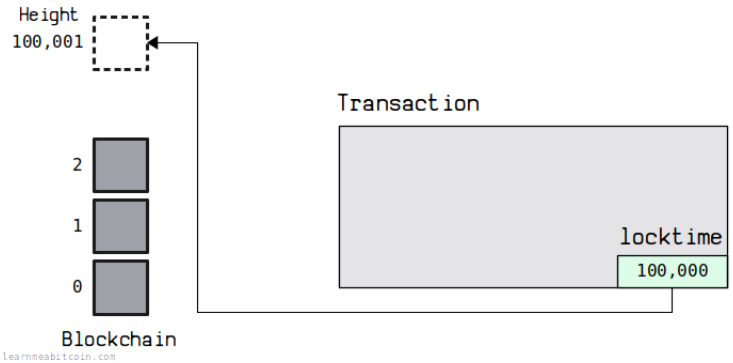
A transaction with a locktime in the future will not be accepted or relayed by nodes either, so you have to store it locally until the locktime you have set on the transaction has passed.

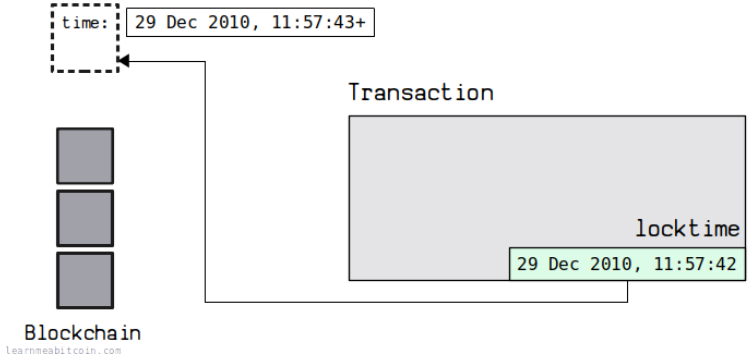
In other words, setting a locktime on a transaction is the equivalent of *post-dating a cheque*.

The locktime field is 4 bytes in size and can hold values between 0 (0x00000000) and 4294967295 (0xffffffff).

You can set a specific block height or time by using different ranges of values:

- <=499999999 Transaction cannot be mined until after a specific height.
- >=500000000 Transaction cannot be mined until after a specific time.





Bitcoin Scripting

Script is a programming language used as locking mechanism for output in bitcoin transactions.

- A locking script (ScriptPubKey) is placed on every transaction output.
- An unlocking script (ScriptSig) 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.
In other words if there's any error while the script is executing, the whole transaction will be invalid and shouldn't be accepted into the block chain.

Script is a **stack-based** programming language. It is not Turing-complete, with no loops.

The Bitcoin scripting language is very small. There's only room for **256** instructions, because each one is represented by one byte. Of those 256, 15 are currently disabled, and 75 are reserved.

Bitcoin Scripting

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.
OP_PUSHBYTES_XX	It will push the following XX bytes onto the stack. It's part of a group of opcodes that push a specific number of bytes onto the stack, from OP_PUSHBYTES_1 to OP_PUSHBYTES_75.

Others OP_CODE can be found at https://en.bitcoin.it/wiki/Script

Bitcoin transactions (output) -ScriptPubKey

Amount: The amount field is where you set the value of the output in satoshis (1 satoshi = 0.00000001 BTC). The total value of your outputs cannot exceed the total value of your inputs.

ScriptPubKey Size: This is the actual locking code that we're placing on the output.

There are a number of different locking code patterns you can use.
Some of the most common are:

P2PKH (Pay To Public Key Hash) - Lock the output to the hash of a public key. To unlock you need to provide the original public key and a valid signature.

P2SH (Pay To Script Hash) - Lock the output to the hash of a custom script. To unlock you need to provide the original script along with the script that satisfies it.

P2WPKH (Pay To Witness Public Key Hash) - Lock the output to the hash of a public key. Works the same as a P2PKH, but the unlocking code goes in the witness field instead of the scriptsig field.

P2WSH (Pay To Witness Script Hash) - Lock the output to the hash of a custom script. Works the same as a P2SH, but the unlocking code goes in the witness field instead of the scriptsig field.

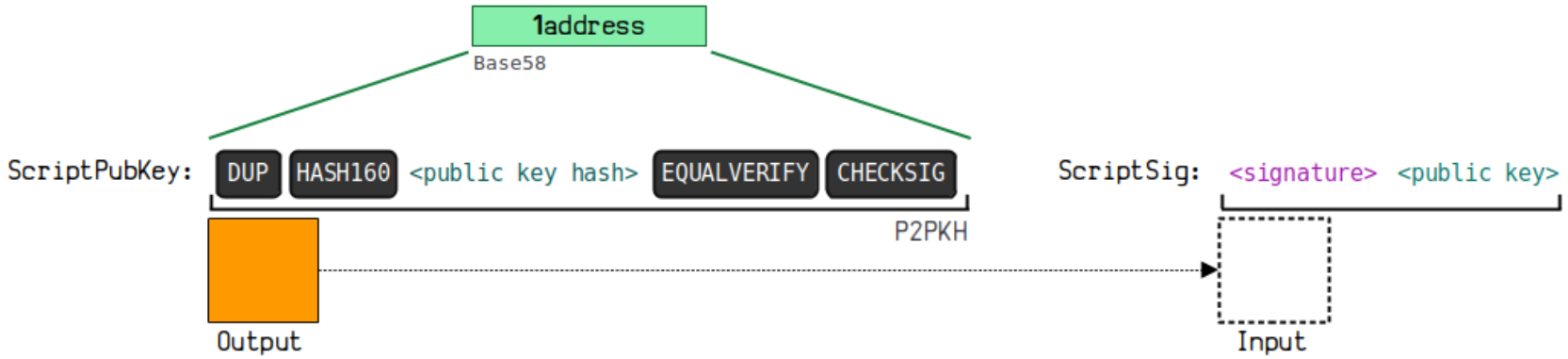
P2PKH

ScriptPubKey code

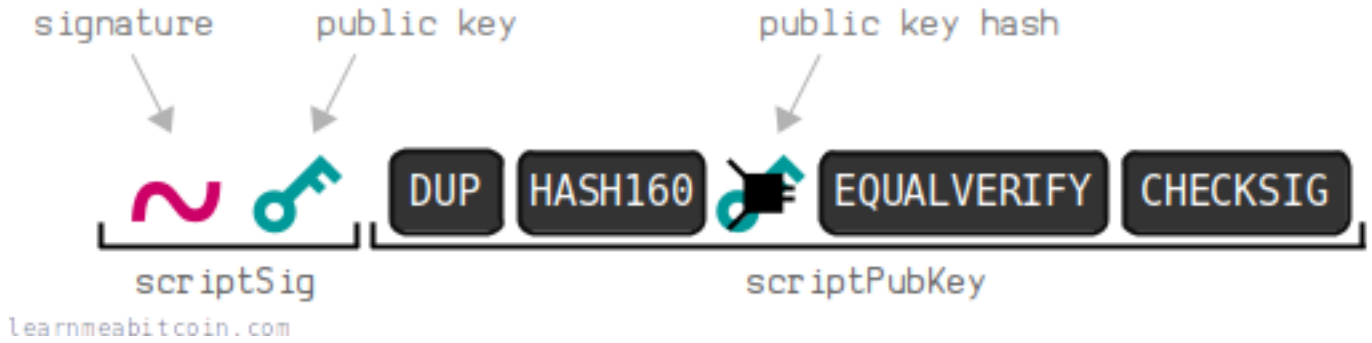
```
OP_DUP
OP_HASH160
OP_PUSHBYTES_20
55ae51684c43435da751ac8d2173b2652eb64105
OP_EQUALVERIFY
OP_CHECKSIG
```

ScriptSig code

```
OP_PUSHBYTES_72
3045022100c233c3a8a510e03ad18b0a24694ef00c78101bfd5ac075b8c1037952ce26
e91e02205aa5f8f88f29bb4ad5808ebc12abfd26bd791256f367b04c6d955f01f28a772
401
OP_PUSHBYTES_33
03f0609c81a45f8cab67fc2d050c21b1acd3d37c7acfd54041be6601ab4cef4f31
```



P2PKH



When this script runs:

1. The original public key is **OP_DUP**licated and then **OP_HASH160**'ed.
2. This hashed value is compared with the public key hash in the ScriptPubKey to make sure it is **OP_EQUALVERIFY**.
3. If it matches, the script continues and the **OP_CHECKSIG** checks the signature against the public key, and puts an OP_1 on the stack if it's valid.

Bitcoin transactions – P2SH

Sometimes, we may need complex spending conditions such as multi-signature.
These conditions can make transactions cumbersome, especially for the sender.

To simplify how users send Bitcoin while still allowing complex spending rules, we can use P2SH (Pay-to-Script-Hash).
With P2SH, instead of sending coins to the hash of a public key, the sender sends them to the hash of a script.

The receiver gives the sender this script hash, and later, when spending the coins, the receiver must reveal the full script. They must also provide the data that makes the script evaluate to true. If the revealed script matches the stored hash and runs correctly, the transaction is valid.

This approach moves the complexity from the sender to the receiver.
The sender just sends to a simple address, without needing to know how the coins will be spent.

For example, Alice can send to Bob's P2SH address without knowing that it uses multi-signature.
Later, Bob provides the correct script and signatures to redeem the coins.

P2SH addresses begin with 3 instead of 1.

Bitcoin transactions – Escrow

Scenario: Imagine Alice and Bob want to do business together. Alice wants to pay Bob in Bitcoin for some physical goods.

The problem: Alice doesn't want to pay before receiving the goods, and Bob doesn't want to send the goods before getting paid.

Solution: Use an escrow transaction with a multisignature (multisig) setup and the help of a third party.

Protocol: Alice creates a 2-of-3 multisig transaction that includes three participants:
Alice, Bob, and a third person named Judy, who acts as an arbitrator.

Once the transaction is written in blockchain, the coins are held in escrow. Now Bob knows that the money is locked safely and that no one can cheat. He can send the goods to Alice with confidence.

If both Alice and Bob act honestly.
After Alice receives the goods, she and Bob both sign a transaction to release the coins to Bob. The 2-of-3 condition is met, and the payment is completed.

Otherwise Judy have to solve the dispute.

Bitcoin transactions – Micropayment

Scenario: Alice wants to pay Bob small amounts of Bitcoin for a service. For example, Bob provides Alice with wireless service and charges her a small fee for every minute she talks on the phone.

Problem: Creating one Bitcoin transaction per minute would not work. There would be too many transactions, and the fees would become too high. If each payment is close to the transaction fee, it becomes too expensive.

Goal: We want to combine many small payments into one final payment, while keeping both sides safe.

Solution: Alice and Bob can use a multisignature (multisig) setup to create a payment channel.

Bitcoin transactions – Micropayment

Protocol:

1. Alice starts by creating a 2-of-2 multisig transaction that locks up the maximum amount she might spend. The coins can only be released if both Alice and Bob sign.

2. After each minute of service, Alice signs a new transaction that sends a little more Bitcoin to Bob and returns the rest to herself. These transactions are only signed by Alice and are not yet published on the blockchain.

3. Bob receives these signed transactions but doesn't publish them right away. He keeps the latest one as proof of how much Alice owes him.

4. When Alice finishes using the service, she tells Bob, "I'm done." Bob then signs the last transaction (the one with the highest payment) and publishes it to the blockchain.

Result: The final transaction pays Bob the full amount for the service and refunds the rest to Alice. All previous transactions stay off the blockchain and are discarded.
This saves transaction fees and keeps the blockchain efficient.

Bitcoin transactions – Micropayment and Locktime

Possible problem: Bob might never sign the last transaction. If that happens, Alice's coins would stay locked forever.

To prevent this, they prepare a refund transaction before starting.
This refund sends all coins back to Alice but is locked until a future time (t) using the locktime parameter.