Note: I in no way own any of the knowledge I have written in this document. This is just my one stop reference when I need it. I have made this as a part of learning blockchain technology. I am following the blockchain at Berkeley's course as a base along with many other references wherever I get stuck. Thanks to all the people in the references who've helped me learn blockchain.

Cryptographic Hash Functions:

How do we ensure trust in communication in a trustless environment? By cryptographic hash functions.
Humans have fingerprints as unique identifiers. If we can design a way to generate fingerprints of our meaningful data, then we can ensure the integrity of our information. Fingerprints are like standardized randomness. You can't guess what someone's fingerprint will look like just by looking at them. In the same way, you shouldn't be able to guess the data that produced a digital fingerprint.
Cryptographic hash functions are one way functions that take some input and produce a pseudorandom output. We say pseudorandom because, while it appears random to us, it is actually always going to be the same output for some given input.
Input to cryptographic hash function is pre-image and output is image. We call these hash functions as cryptographic because they are built for security.

Three special properties of cryptographic hash functions

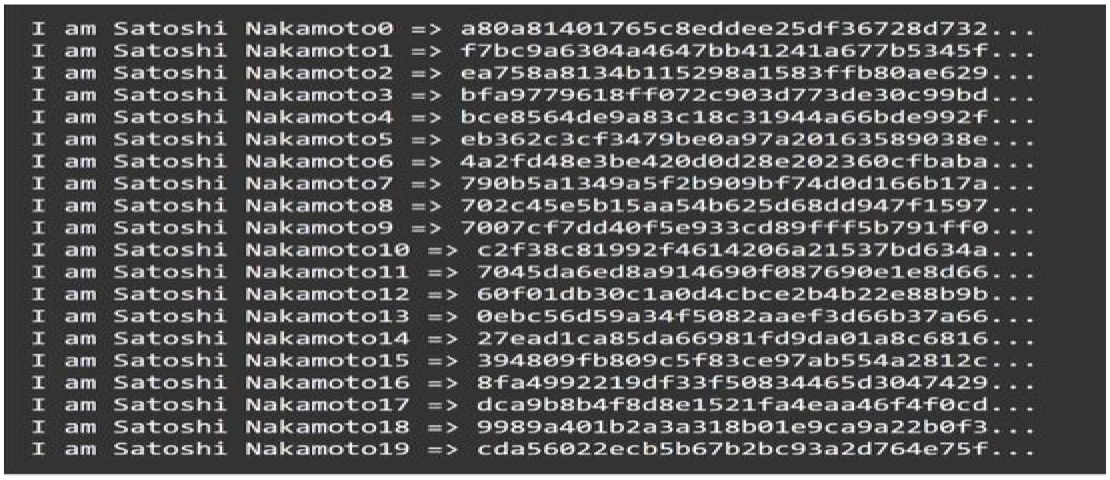
- Preimage resistance

  This refers to the difficulty of finding input given the output.
  If y = H(x), given y and H function, it's computationally difficult to find x.
  <u>Fingerprint analogy:</u>
  Whose fingerprint is that?
- Second preimage resistance

  Given x it is computationally difficult to find some y such that
  H(x) == H(y)
  <u>Fingerprint analogy:</u>
  Can you find someone with the same fingerprint as you?
- Collision resistance

  Expanding the concept of second preimage resistance, any two arbitrary input mappings to the same output is a bad thing.
  It is computationally difficult to find x and y such that
  H(x) == H(y)
  <u>Fingerprint analogy:</u>
  Can you find two random people with the same fingerprint?

Avalanche Effect : Consequences of above three properties is the avalanche effect. A small change in the input produces a pseudorandom change in the output. Prevents guessing of output based on inputs.

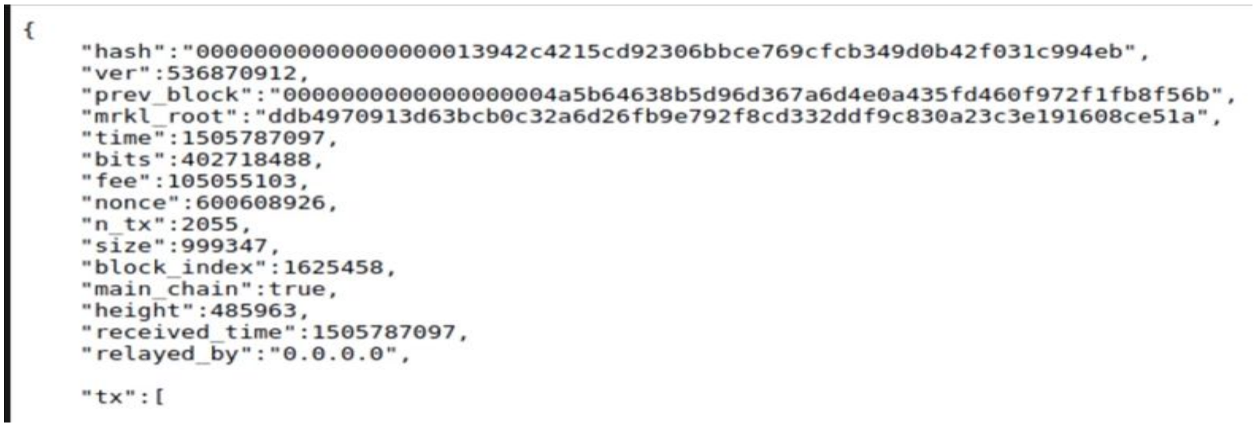


```
I am Satoshi Nakamoto0 => a80a81401765c8eddee25df36728d732...
I am Satoshi Nakamoto1 => f7bc9a6304a4647bb41241a677b5345f...
I am Satoshi Nakamoto2 => ea758a8134b115298a1583ffb80ae629...
I am Satoshi Nakamoto3 => bfa9779618ff072c903d773de30c99bd...
I am Satoshi Nakamoto4 => bce8564de9a83c18c31944a66bde992f...
I am Satoshi Nakamoto5 => eb362c3cf3479be0a97a20163589038e...
I am Satoshi Nakamoto6 => 4a2fd48e3be420d0d28e202360cfbaba...
I am Satoshi Nakamoto7 => 790b5a1349a5f2b909bf74d0d166b17a...
I am Satoshi Nakamoto8 => 702c45e5b15aa54b625d68dd947f1597...
I am Satoshi Nakamoto9 => 7007cf7dd40f5e933cd89fff5b791ff0...
I am Satoshi Nakamoto10 => c2f38c81992f4614206a21537bd634a...
I am Satoshi Nakamoto11 => 7045da6ed8a914690f087690e1e8d66...
I am Satoshi Nakamoto12 => 60f01db30c1a0d4cbce2b4b22e88b9b...
I am Satoshi Nakamoto13 => 0ebc56d59a34f5082aaef3d66b37a66...
I am Satoshi Nakamoto14 => 27ead1ca85da66981fd9da01a8c6816...
I am Satoshi Nakamoto15 => 394809fb809c5f83ce97ab554a2812c...
I am Satoshi Nakamoto16 => 8fa4992219df33f50834465d3047429...
I am Satoshi Nakamoto17 => dca9b8b4f8d8e1521fa4eaa46f4f0cd...
I am Satoshi Nakamoto18 => 9989a401b2a3a318b01e9ca9a22b0f3...
I am Satoshi Nakamoto19 => cda56022ecb5b67b2bc93a2d764e75f...
```

SHA-256 used by bitcoin in many scenarios, designed by NSA, member of SHA-2 (Secure Hash Algorithm) family of cryptographic hash functions
Input : Less than 2^64 bits. (can take in huge amounts as well, thus say arbitrary)
Output : 256 bit fixed size.

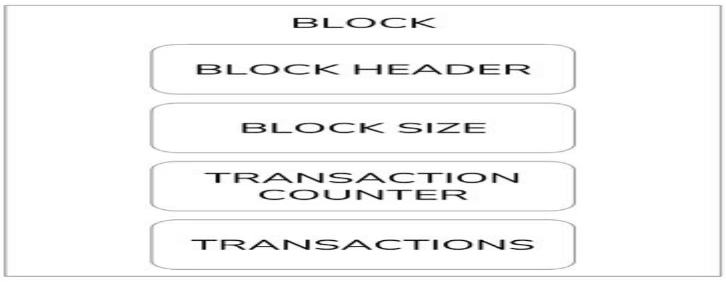
Bitcoin uses SHA-256^2 (squared), also called SHA-256d, which uses SHA-256 twice in a row.
SHA256(SHA256(x))

A Tamper-Evident Database:
Example of a block in JSON format:

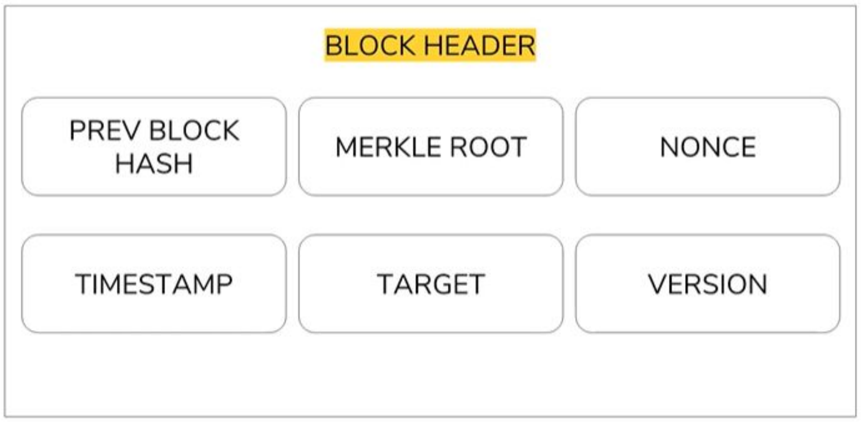


```
{
    "hash":"000000000000000000013942c4215cd92306bbce769cfcb349d0b42f031c994eb",
    "ver":536870912,
    "prev_block":"0000000000000000004a5b64638b5d96d367a6d4e0a435fd460f972f1fb8f56b",
    "mrkl_root":"ddb4970913d63bcb0c32a6d26fb9e792f8cd332ddf9c830a23c3e191608ce51a",
    "time":1505787097,
    "bits":402718488,
    "fee":105055103,
    "nonce":600608926,
    "n_tx":2055,
    "size":999347,
    "block_index":1625458,
    "main_chain":true,
    "height":485963,
    "received_time":1505787097,
    "relayed_by":"0.0.0.0",

    "tx":[
```



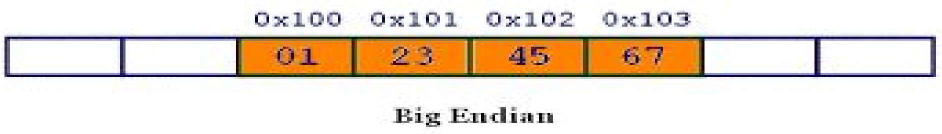
components of a block.

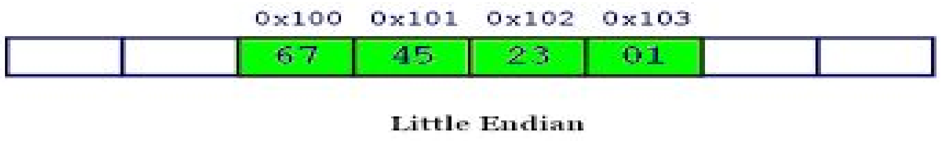


```
blockID = H(blockHeader) = H(prevBlockHash || merkleRoot || nonce...)
```

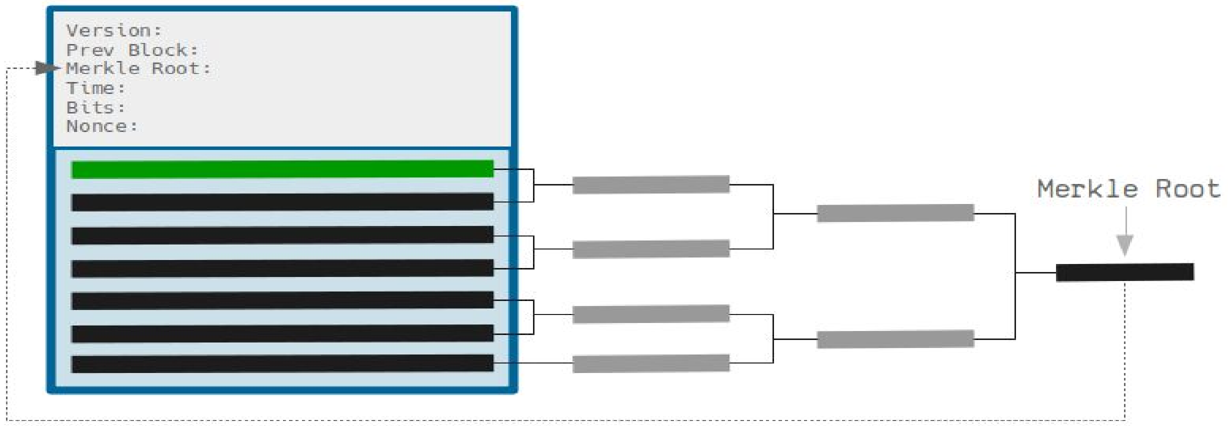
Block header represents the metadata associated with every block.
Merkle root represents summary of transactions, previous block hash represents chaining, nonce represents proof of work.
First bitcoin ->
https://www.blockchain.com/en/btc/block/00000000d1145790a8694403d4063f323d499e655c83426834d4ce2f8dd4a2ee
Merkle Root :
A merkle tree is a very specific version of a binary tree where the lowest level is made of the hashes of the information that we would like to summarize.
Watch this later : https://www.youtube.com/watch?v=gUwXCt1qkBU
https://learnmeabitcoin.com/guide/merkle-root

Little and Big endian:
0x01234567

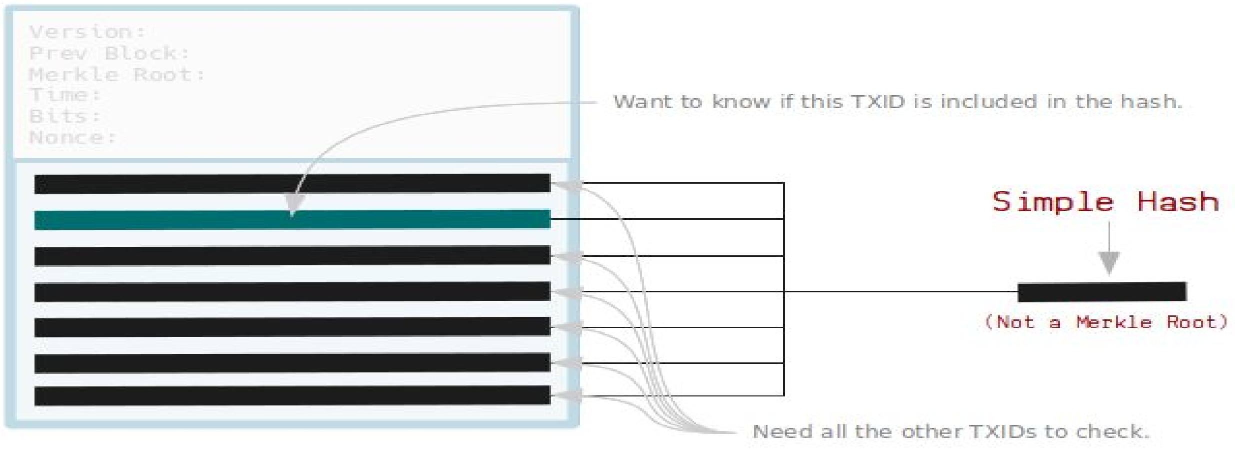




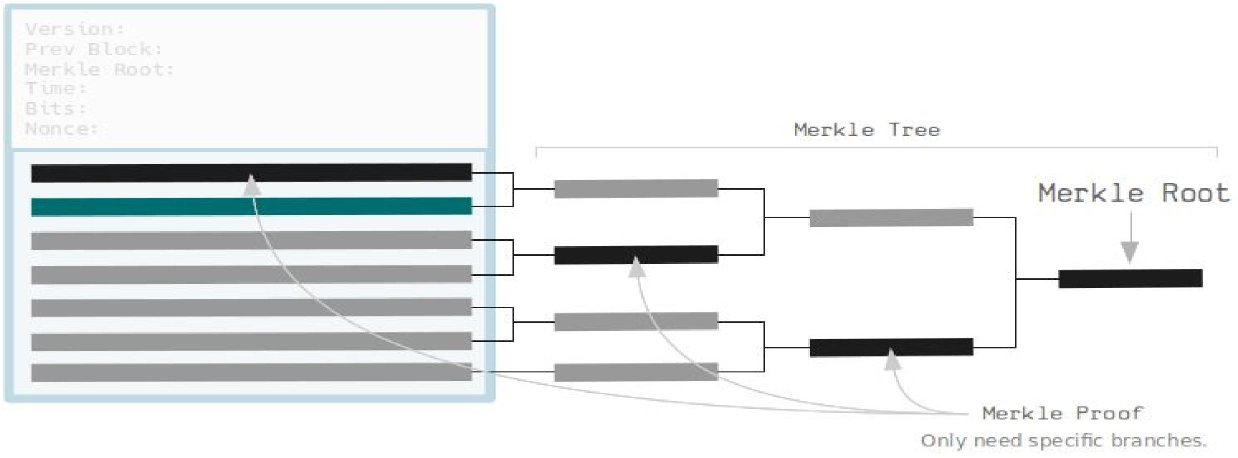
Merkle root is a fingerprint for all the transactions in a block. A merkle root is created by hashing together pairs of TXIDs, which gives you a short yet unique fingerprint for all the transactions in a block.



If we wanted to create a unique fingerprint for all the transactions in a block, we could just hash all the TXIDs together in one go. However, if we later wanted to check that a TXID was a part of that hash, we would need to know all the other TXIDs too.

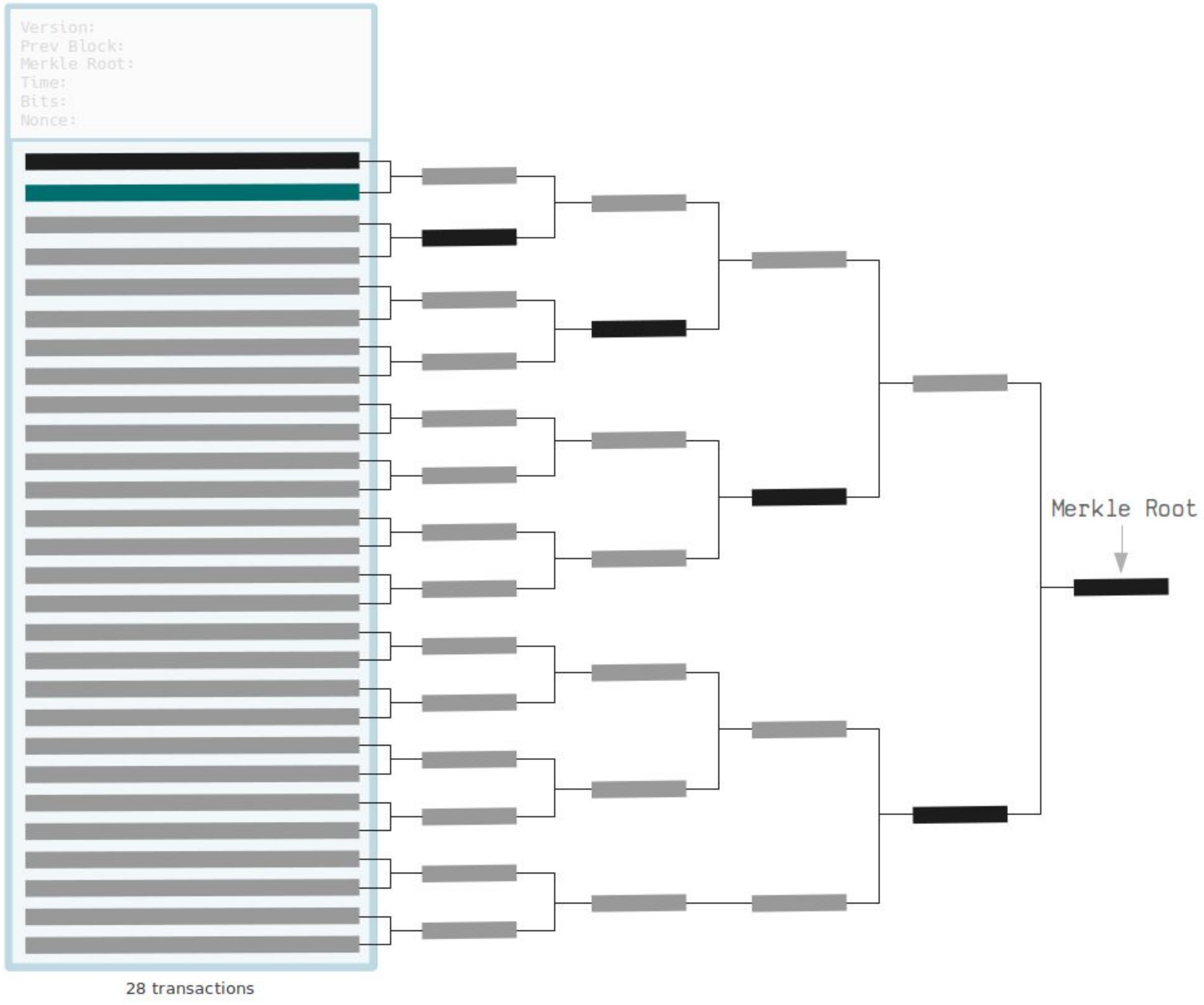

Version:
Prev Block:
Merkle Root:
Time:
Bits:
Nonce:
Want to know if this TXID is included in the hash.
Simple Hash
(Not a Merkle Root)
Need all the other TXIDs to check.

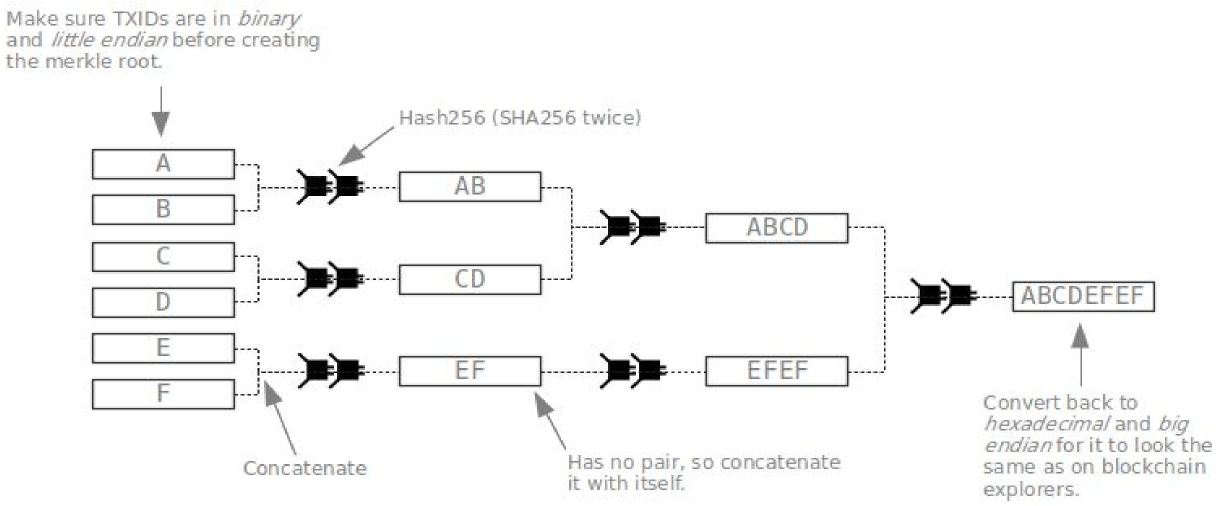
But with a merkle tree, if we want to check that a TXID is part of the merkle root, we would only need to know some of the hashes along the path of the tree



As a result, by using a merkle root as our fingerprint for the block header, we can later find out if a transaction exists in a block without having to know every other TXID in the block.

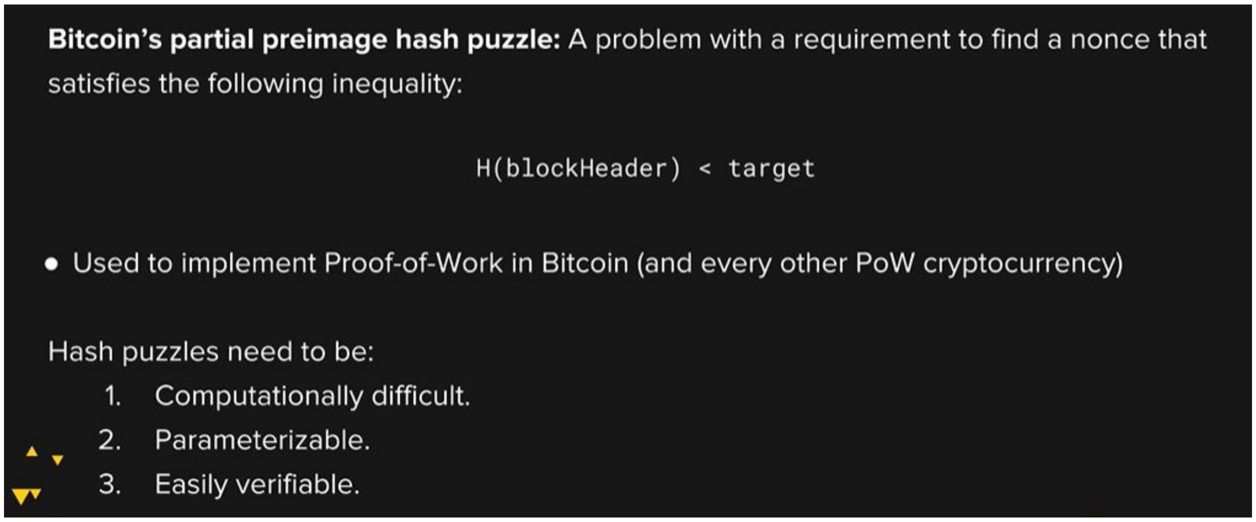
There doesn't appear to be much benefit to using merkle trees when you have a small number of transactions in a block, but you can really see the difference when you have a much larger number of starting "leaves" in the tree




Make sure TXIDs are in *binary* and *little endian* before creating the merkle root.
Hash256 (SHA256 twice)
A
B
C
D
E
F
AB
CD
EF
ABCD
EFEF
ABCDEFEF
Concatenate
Has no pair, so concatenate it with itself.
Convert back to *hexadecimal* and *big endian* for it to look the same as on blockchain explorers.

Previous Block Hash:
Since every block contains a hash of it's previous block, if one block is altered, all the blocks after that will also be altered. Changing any part of history will change entire future from that point.

Partial Preimage hash puzzle in proof of work:



Bitcoin's partial preimage hash puzzle: A problem with a requirement to find a nonce that satisfies the following inequality:

```
H(blockHeader) < target
```

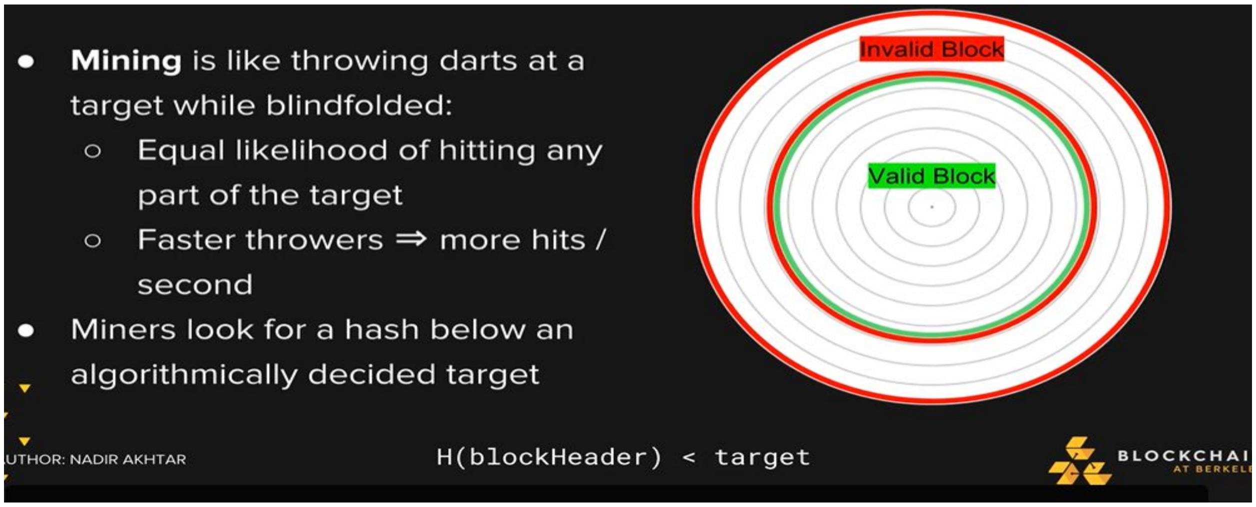
- Used to implement Proof-of-Work in Bitcoin (and every other PoW cryptocurrency)

Hash puzzles need to be:

1. Computationally difficult.
2. Parameterizable.
3. Easily verifiable.

Parameterizable means adjustable. We should be able to adjust the difficulty of the puzzle to be hard or easy. Easily verifiable means computers should not do much work to check if the answer is correct or not. It should just take one hash for example to prove that some nonce is correct, even if finding the nonce takes millions of tries.

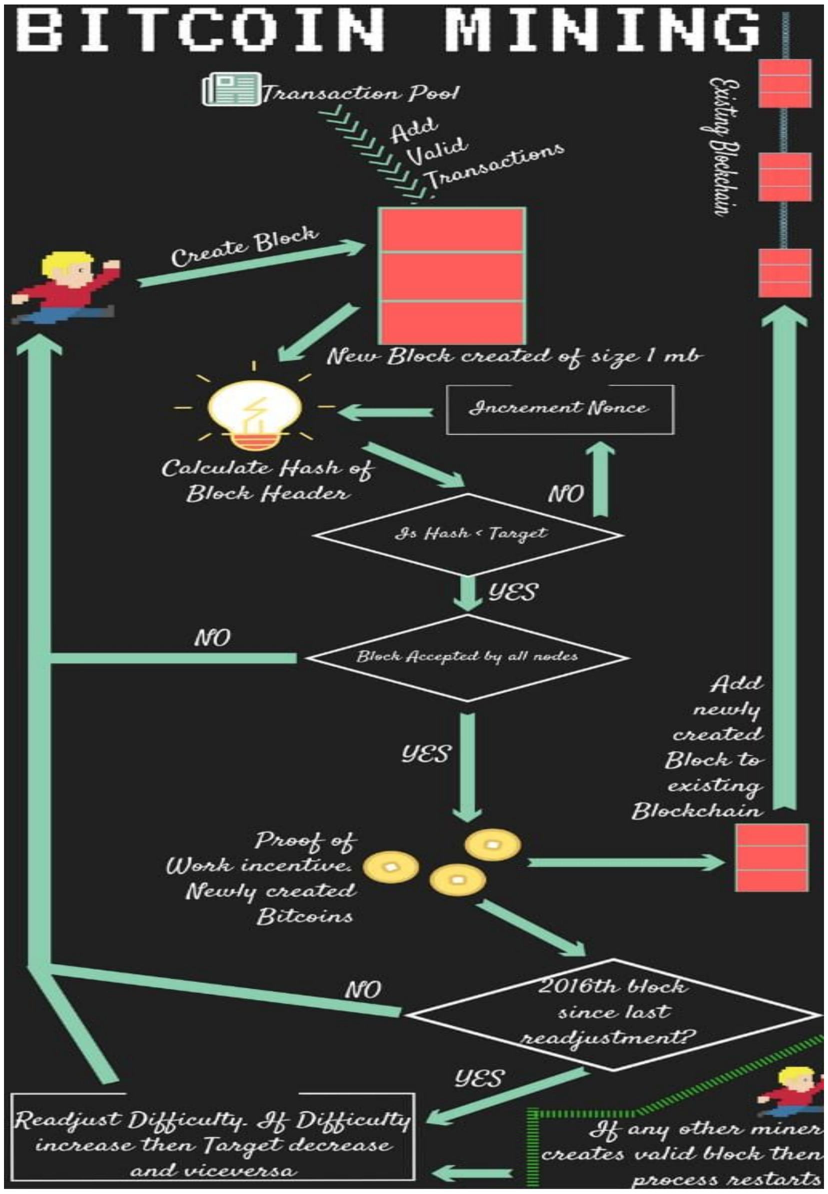
Mining :
Miners are looking for some hash output that is below some algorithmically decided target.



- **Mining** is like throwing darts at a target while blindfolded:
  - Equal likelihood of hitting any part of the target
  - Faster throwers ⇒ more hits / second
- Miners look for a hash below an algorithmically decided target

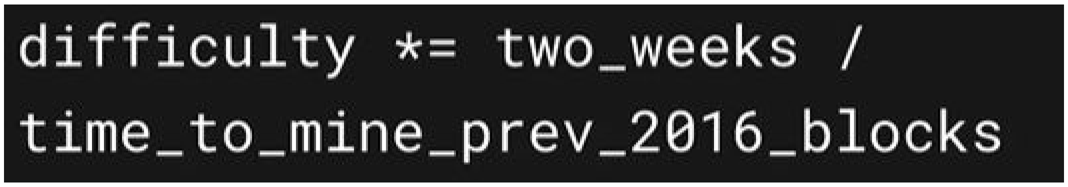
UTHOR: NADIR AKHTAR

```
H(blockHeader) < target
```


BITCOIN MINING
Transaction Pool
Add Valid Transactions
Existing Blockchain
Create Block
New Block created of size 1 mb
Increment Nonce
Calculate Hash of Block Header
NO
Is Hash < Target
YES
NO
Block Accepted by all nodes
YES
Add newly created Block to existing Blockchain
Proof of Work incentive. Newly created Bitcoins
NO
2016th block since last readjustment?
YES
Readjust Difficulty. If Difficulty increase then Target decrease and viceversa
If any other miner creates valid block then process restarts

Block Difficulty:

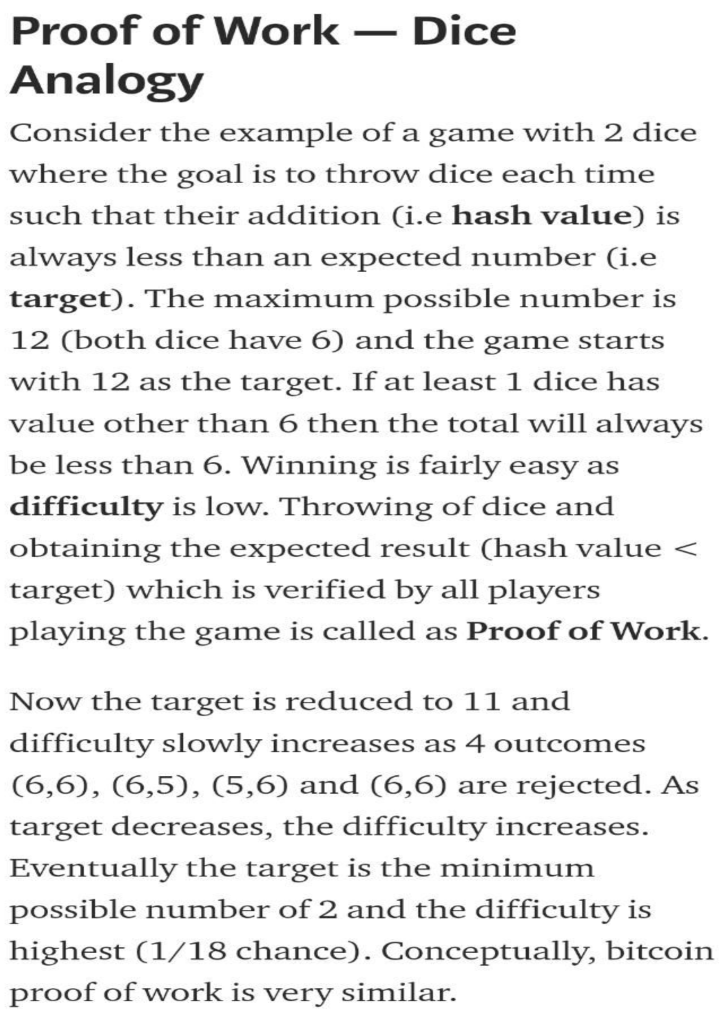
Difficulty is the representation of the number of expected computations required to find a block. Again, we can't predict how many computations anyone produces to solve the puzzle, but we can approximate based on how quickly the puzzle is solved on average. The difficulty is implemented as a requirement of a leading number of zeros on the block header hash. This is why the example block at the start of this section had many zeros at the start of its block hash. As the number of zeros increases, so will the difficulty, and vice versa.This difficulty adjusts with the global hashrate. We know that the amount of computing power in the network will always be changing, as miners join and leave the network, but we want to maintain a block time of ten minutes. For this reason, we have to raise and lower the difficulty alongside the hashpower growth and decay of the network.



```
difficulty *= two_weeks /
time_to_mine_prev_2016_blocks
```

Every two weeks, we check to see how long it took to calculate those 2016 blocks.
If every block took exactly 10 minutes, then it should have taken precisely two weeks to produce those 2016 blocks. If we took too long, it's because the puzzle was too hard to solve, and if we didn't take long enough, then the puzzle was too easy. We make adjustments on the puzzle difficulty accordingly going forward.

Let's say that the current difficulty is 10.
Then what's the new difficulty if the time to mine 2016 blocks is exactly two weeks?
Yes, it's still 10!
The puzzle was precisely as hard as we wanted, so the difficulty stays exactly the same.
What about when the time to mine those 2016 blocks is just one week?
Or a staggering 4 weeks?
If time to mine is one week, then the difficulty is 20!
We mined those blocks in half the expected time, meaning that the puzzle was half as
hard as necessary, so we make it twice as difficult.
If the time to mine is 4 weeks now, the difficulty is now 5!
We mined those blocks in twice the expected time, meaning that the puzzle was twice as
hard as necessary, so we make it half as difficult.
So, the difficulty is inversely proportional to the time to mine.



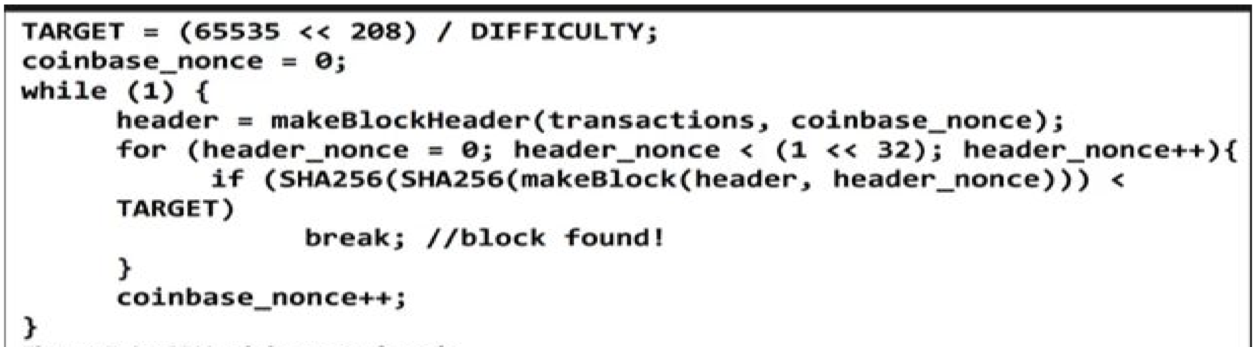
Proof of Work — Dice Analogy

Consider the example of a game with 2 dice where the goal is to throw dice each time such that their addition (i.e **hash value**) is always less than an expected number (i.e **target**). The maximum possible number is 12 (both dice have 6) and the game starts with 12 as the target. If at least 1 dice has value other than 6 then the total will always be less than 6. Winning is fairly easy as **difficulty** is low. Throwing of dice and obtaining the expected result (hash value < target) which is verified by all players playing the game is called as **Proof of Work**.

Now the target is reduced to 11 and difficulty slowly increases as 4 outcomes (6,6), (6,5), (5,6) and (6,6) are rejected. As target decreases, the difficulty increases. Eventually the target is the minimum possible number of 2 and the difficulty is highest (1/18 chance). Conceptually, bitcoin proof of work is very similar.

Coinbase Transaction:

The block reward to the miners goes into the coinbase transaction. Whenever a miner produces a block, they first make a coinbase transaction which is always the first transaction of the Merkle Tree. This coinbase transaction grants miners a reward of some bitcoins which can be spent at some later date. This is how new bitcoins are minted, or introduced, into the network.



```
TARGET = (65535 << 208) / DIFFICULTY;
coinbase_nonce = 0;
while (1) {
      header = makeBlockHeader(transactions, coinbase_nonce);
      for (header_nonce = 0; header_nonce < (1 << 32); header_nonce++){
            if (SHA256(SHA256(makeBlock(header, header_nonce))) <
      TARGET)
                  break; //block found!
      }
      coinbase_nonce++;
}
```

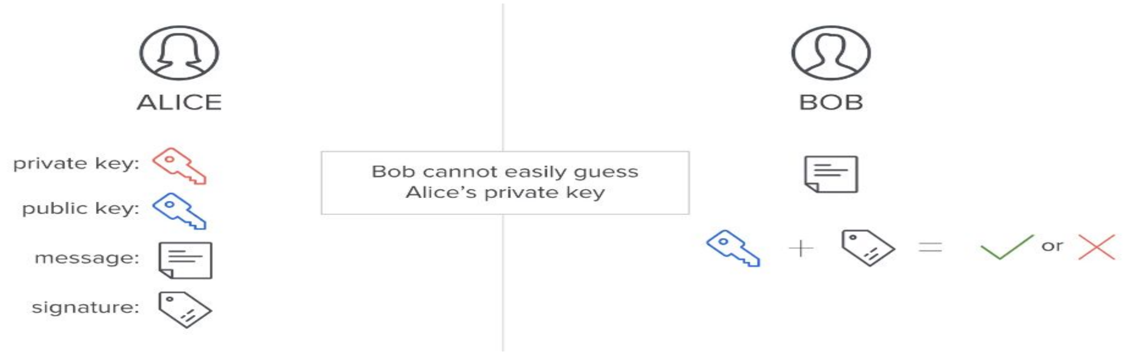
Figure 5.6 : CPU mining pseudocode.

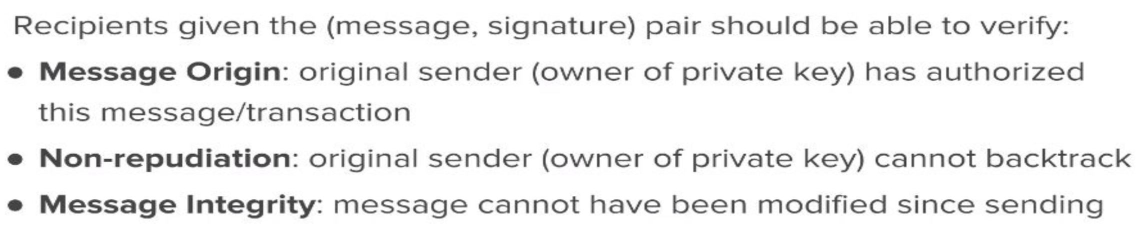
What is the difference between header and coinbase nonce?

Signatures and Authorizations:

Using digital signatures, we can ensure that the current transactions we send to the rest of the network are tamper evident and are sent by an authenticated person only, as that person's private key can only prove the ownership to that public key.

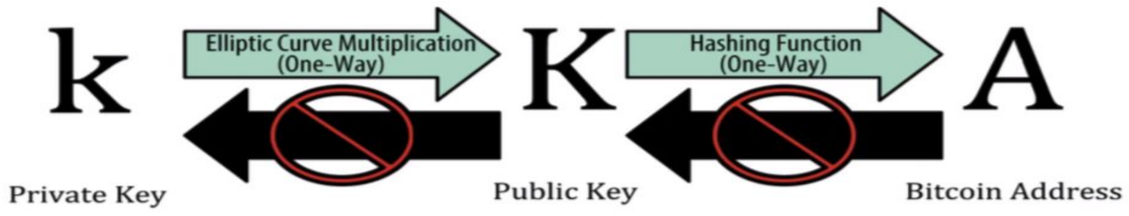
Digital Signature Schemes (DSS)
Private and public keys in bitcoin are generated through an algorithm called ECDSA (Elliptic Curve Digital Signature Algorithm).
Note here we are not trying to prove that the message is secret or something. Anyone with the sender's public key can definitely view the message. We are trying to prove that the message is sent by the valid user and it's not tampered anywhere in between if public keys verify the signature as correct, if the message was tampered, the signature would not have been valid.
The signature comprises of the message and the sender's private key (obviously in a way no one could know the sender's private key). Read the first answer and it's comments.
https://stackoverflow.com/questions/18257185/how-does-a-public-key-verify-a-signature



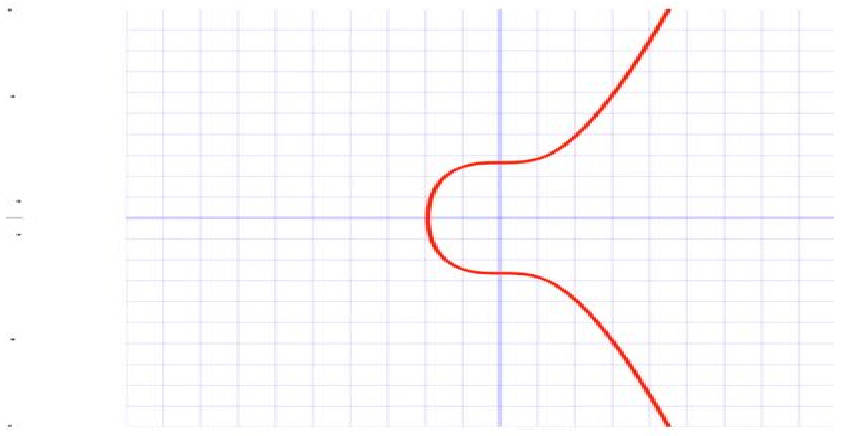

Recipients given the (message, signature) pair should be able to verify:

- **Message Origin**: original sender (owner of private key) has authorized this message/transaction
- **Non-repudiation**: original sender (owner of private key) cannot backtrack
- **Message Integrity**: message cannot have been modified since sending

Backtrack meaning sender should not be able to change to null or anything else. Once signed by her, she cannot take it back.


k
Elliptic Curve Multiplication (One-Way)
K
Hashing Function (One-Way)
A
Private Key
Public Key
Bitcoin Address

Elliptic Curve Cryptography:
An elliptic curve is a mathematical curve defined by the general form
y^2 = x^3 + ax + b
We take everything over a finite field because we want to encode every value possible in a constant amount of space.

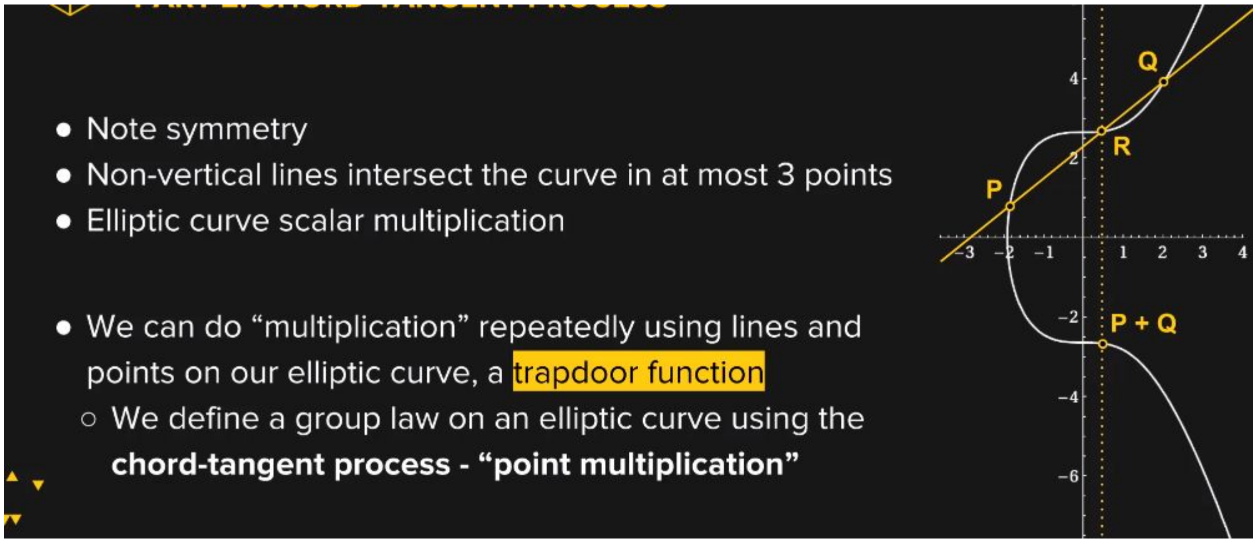


secp256k1 : $Y^2 = (X^3 + 7)$

The curve is specified with a few parameters:

- The actual curve formula (coefficients)
- Field
- Generator point

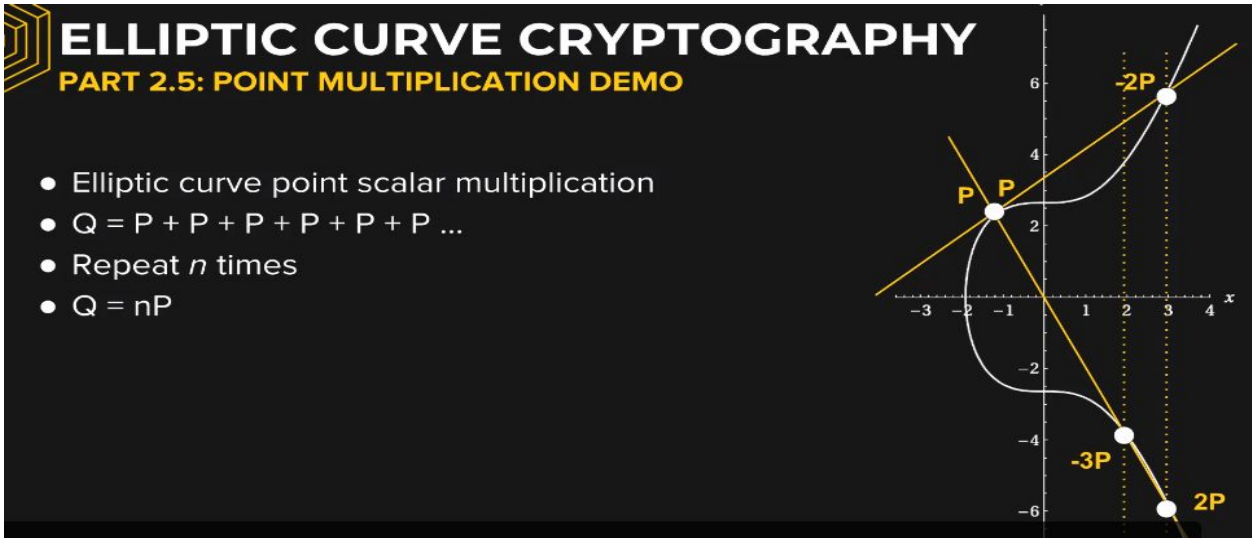
The symmetry of curve over the x-axis is preserved even when taking it over a finite field.
For later reading :
https://hackernoon.com/what-is-the-math-behind-elliptic-curve-cryptography-f61b25253da3



- Note symmetry
- Non-vertical lines intersect the curve in at most 3 points
- Elliptic curve scalar multiplication

- We can do “multiplication” repeatedly using lines and points on our elliptic curve, a trapdoor function
    - We define a group law on an elliptic curve using the **chord-tangent process - “point multiplication”**

This is a trapdoor or one way function because given a point k such that k = p+q, it is difficult to find points p and q.



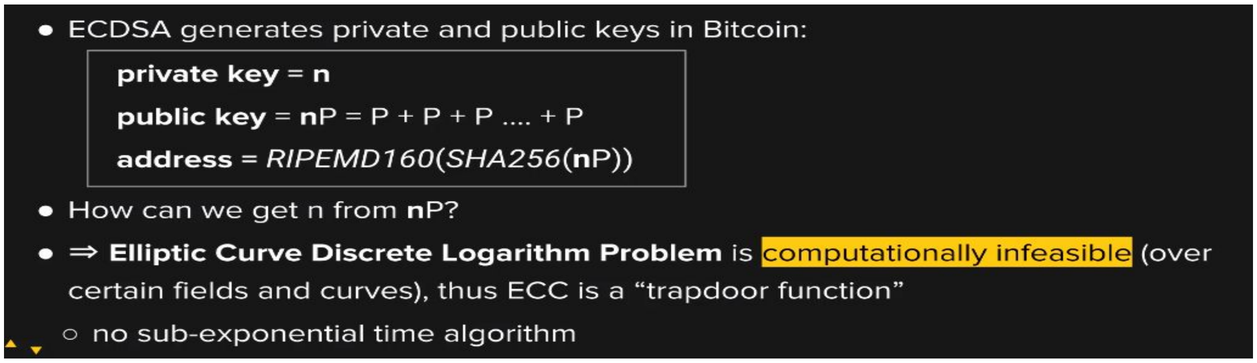
ELLIPTIC CURVE CRYPTOGRAPHY

PART 2.5: POINT MULTIPLICATION DEMO

- Elliptic curve point scalar multiplication
- Q = P + P + P + P + P + P + P ...
- Repeat *n* times
- Q = nP

We'll study all these concepts in detail in our next course with dev ++.

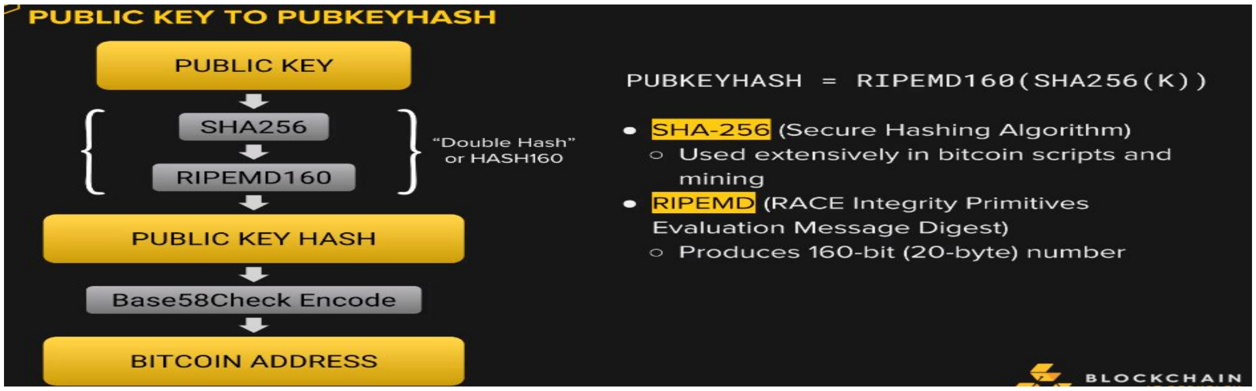
SHA-256 makes the address quantum resistant. Because quantum computers could be able to break elliptic curves and reverse the one way point scalar multiplication and get the private key out of the public key. But quantum computers cannot reverse hash functions.
RipeMD shortens the size from 256 to 160 bits.
Future read:
https://medium.com/coinmonks/reasons-why-quantum-supremacy-wont-threaten-bitcoin-d71f339fa626



- ECDSA generates private and public keys in Bitcoin:

  private key = **n**
  public key = **n**P = P + P + P .... + P
  address = *RIPEMD160*(*SHA256*(**n**P))

- How can we get n from **n**P?
- ⇒ **Elliptic Curve Discrete Logarithm Problem** is computationally infeasible (over certain fields and curves), thus ECC is a "trapdoor function"
  - no sub-exponential time algorithm

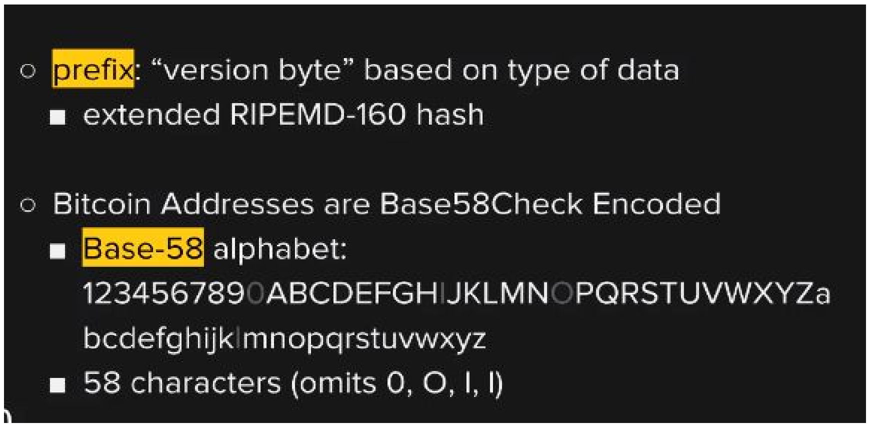


PUBLIC KEY TO PUBKEYHASH

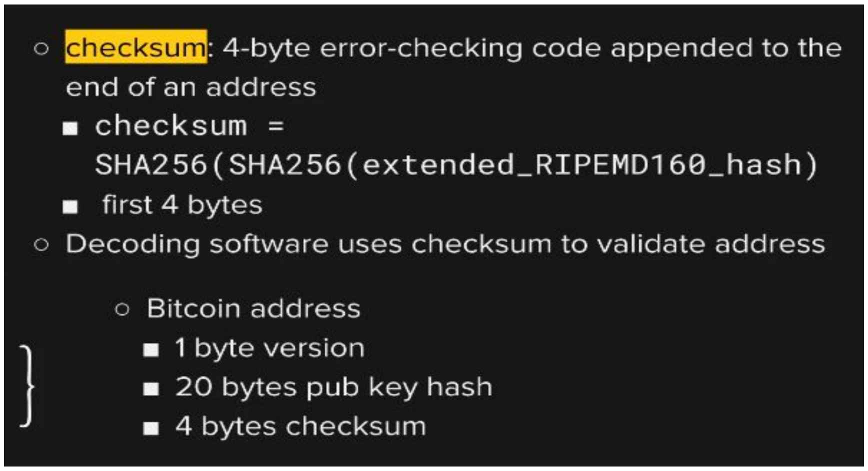
PUBKEYHASH = RIPEMD160(SHA256(K))

- SHA-256 (Secure Hashing Algorithm)
  - Used extensively in bitcoin scripts and mining
- RIPEMD (RACE Integrity Primitives Evaluation Message Digest)
  - Produces 160-bit (20-byte) number

We should make our public key hash a bit more human friendly.
In the version byte, we specify which network we're on: the main Bitcoin network, or a smaller test network.



- prefix: "version byte" based on type of data
  - extended RIPEMD-160 hash
- Bitcoin Addresses are Base58Check Encoded
  - Base-58 alphabet: 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
  - 58 characters (omits 0, O, I, l)

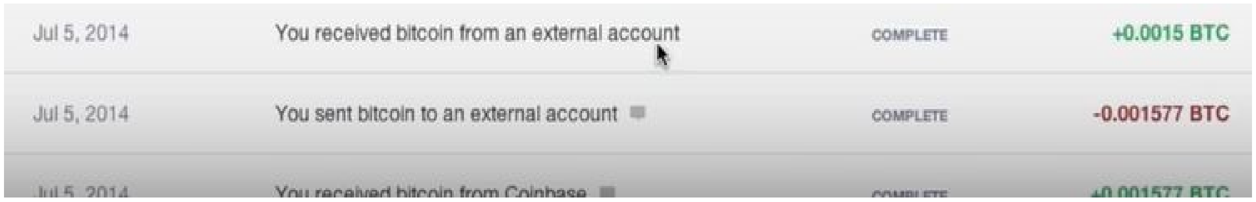


- checksum: 4-byte error-checking code appended to the end of an address
  - checksum = SHA256(SHA256(extended_RIPEMD160_hash)
  - first 4 bytes
- Decoding software uses checksum to validate address

  - Bitcoin address
    - 1 byte version
    - 20 bytes pub key hash
    - 4 bytes checksum

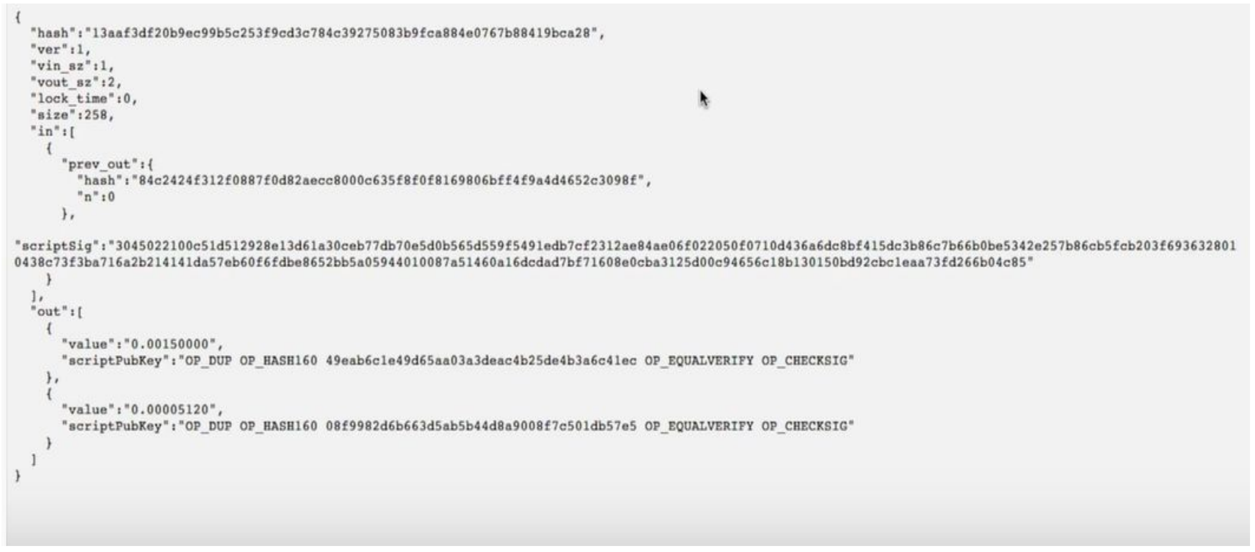
Contents of a Transaction:

https://www.youtube.com/watch?v=Shd9nXe1X-0
https://www.youtube.com/watch?v=f9nxuhLSyOg



Jul 5, 2014	You received bitcoin from an external account	COMPLETE	+0.0015 BTC
Jul 5, 2014	You sent bitcoin to an external account	COMPLETE	-0.001577 BTC
Jul 5, 2014	You received bitcoin from Coinbase	COMPLETE	+0.001577 BTC

- Hash: is the hash of all the information that's below it in the transaction.
- Version
- Number of inputs that we have in the transaction
- Number of outputs that we have in the transaction
- Lock
- Size of transaction



```
{
  "hash":"13aaf3df20b9ec99b5c253f9cd3c784c39275083b9fca884e0767b88419bca28",
  "ver":1,
  "vin_sz":1,
  "vout_sz":2,
  "lock_time":0,
  "size":258,
  "in":[
    {
      "prev_out":{
        "hash":"84c2424f312f0887f0d82aecc8000c635f8f0f8169806bff4f9a4d4652c3098f",
        "n":0
      },

"scriptSig":"3045022100c51d512928e13d61a30ceb77db70e5d0b565d559f5491edb7cf2312ae84ae06f022050f0710d436a6dc8bf415dc3b86c7b66b0be5342e257b86cb5fcb203f693632801
0438c73f3ba716a2b214141da57eb60f6fdbe8652bb5a05944010087a51460a16dcdad7bf71608e0cba3125d00c94656c18b130150bd92cbc1eaa73fd266b04c85"
    }
  ],
  "out":[
    {
      "value":"0.00150000",
      "scriptPubKey":"OP_DUP OP_HASH160 49eab6c1e49d65aa03a3deac4b25de4b3a6c41ec OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value":"0.00005120",
      "scriptPubKey":"OP_DUP OP_HASH160 08f9982d6b663d5ab5b44d8a9008f7c501db57e5 OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}
```

For input field:

prev_out is the transaction that passed the rights to us.

`eg: I have to give bob 5btc. This is the transaction. But the 5btc that I am passing on has been given to me by alice who once gave me 10btc. So this is that prev transaction that gave rights to me.`

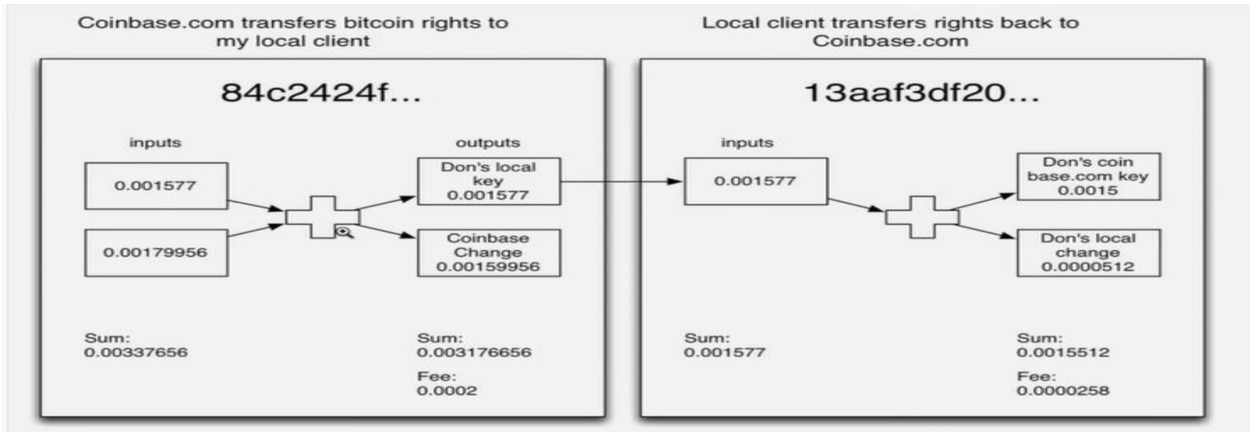
And n is output number in that transaction. If n is 0, it means we have output no. 1, etc.

Script signature is the proof that we have the rights to be able to use the previous transaction. We are not lying. Alice really gave us those 10 btc.

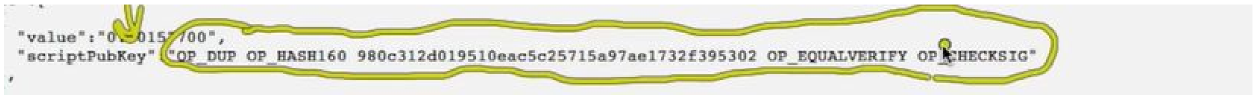
For output field:

The first output is the amount I am sending to bob(5). And the second is the change that I am going to receive back in my local client(4). I get 4 not 5, because 1 is the fee that was deducted.

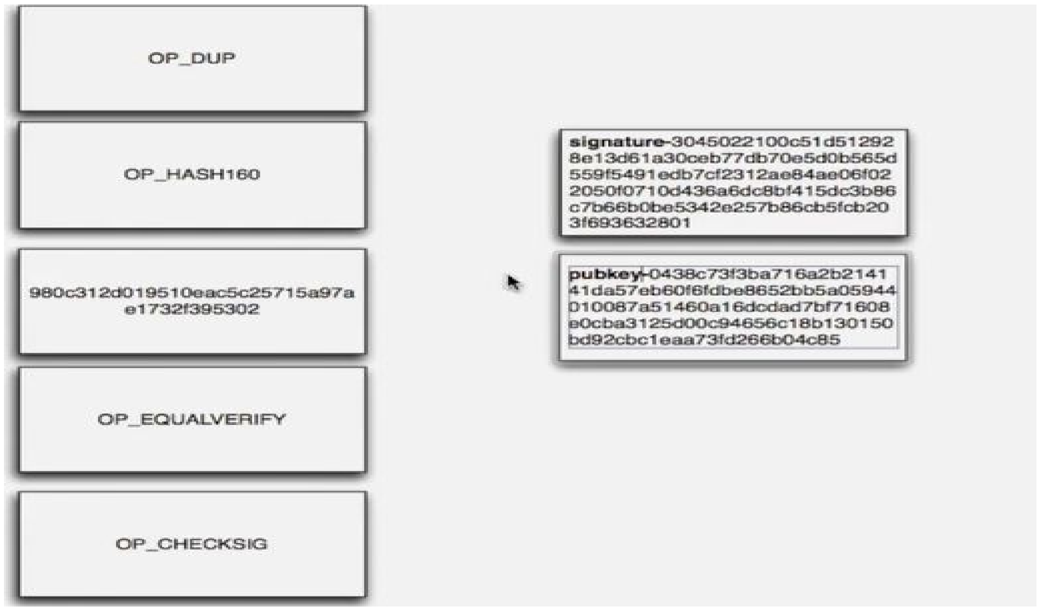
ScriptPubKey is the conditions of who can use this output that have to be met by the one using them as input in order for them to use those coins as a transaction. When I send the btc, I add the receiver's public key and in order to claim them, they need to sign using their private key.


Coinbase.com transfers bitcoin rights to my local client
84c2424f...
inputs
0.001577
0.00179956
outputs
Don's local key 0.001577
Coinbase Change 0.00159956
Sum: 0.00337656
Sum: 0.003176656
Fee: 0.0002
Local client transfers rights back to Coinbase.com
13aaf3df20...
inputs
0.001577
Don's coin base.com key 0.0015
Don's local change 0.0000512
Sum: 0.001577
Sum: 0.0015512
Fee: 0.0000258

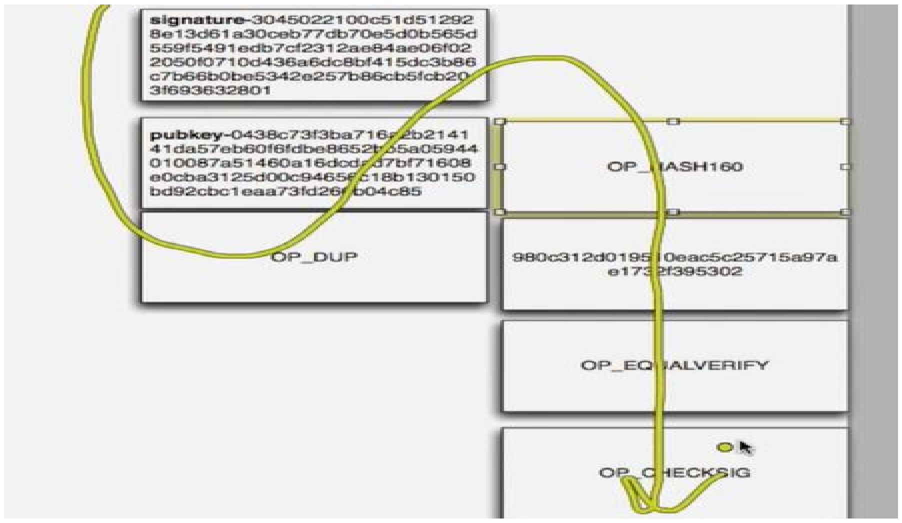
The miner who includes this block gets reward as well as the fee.
What are these conditions that we are talking about?



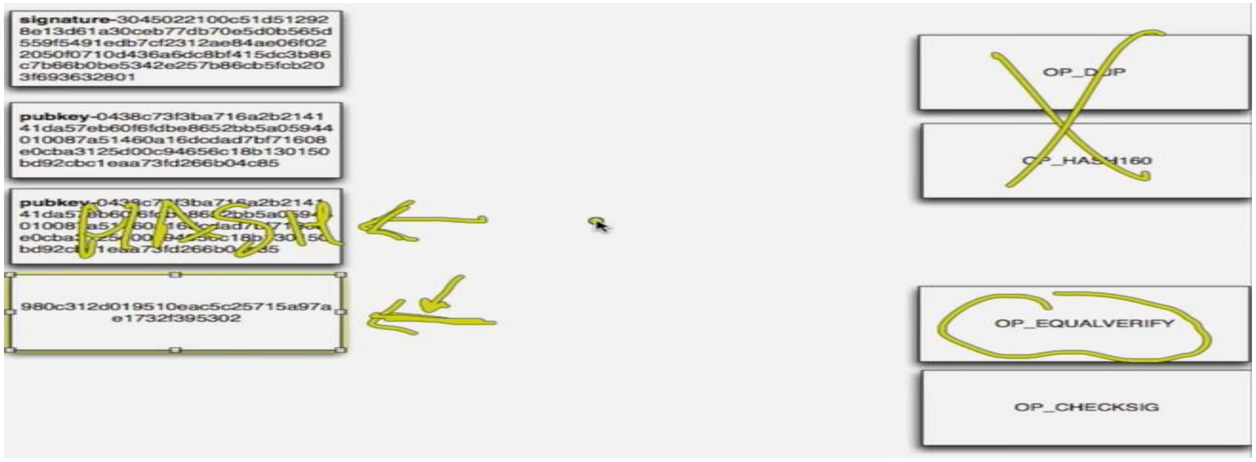
```
"value":"0[illegible]015[illegible]00",
"scriptPubKey" "OP_DUP OP_HASH160 980c312d019510eac5c25715a97ae1732f395302 OP_EQUALVERIFY OP_CHECKSIG"
,
```


OP_DUP
OP_HASH160
980c312d019510eac5c25715a97a e1732f395302
OP_EQUALVERIFY
OP_CHECKSIG
signature-3045022100c51d51292 8e13d61a30ceb77db70e5d0b565d 559f5491edb7cf2312ae84ae06f02 2050f0710d436a6dc8bf415dc3b86 c7b66b0be5342e257b86cb5fcb20 3f693632801
pubkey-0438c73f3ba716a2b2141 41da57eb60f6fdbe8652bb5a05944 010087a51460a16dcdad7bf71608 e0cba3125d00c94656c18b130150 bd92cbc1eaa73fd266b04c85

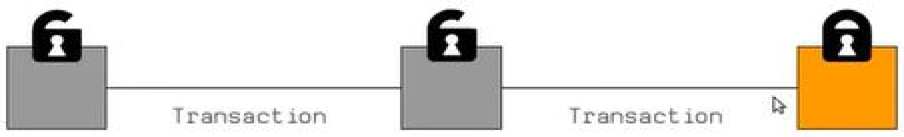
Lhs were in output section, a problem to be solved by the receiver to claim the ownership of transaction. Rhs is in input section, giving the proof of ownership.
Sequence of commands:


signature-3045022100c51d51292 8e13d61a30ceb77db70e5d0b565d 559f5491edb7cf2312ae84ae06f02 2050f0710d436a6dc8bf415dc3b86 c7b66b0be5342e257b86cb5fcb20 3f693632801
pubkey-0438c73f3ba716a2b2141 41da57eb60f6fdbe8652bb5a05944 010087a51460a16dcdad7bf71608 e0cba3125d00c94656c18b130150 bd92cbc1eaa73fd266b04c85
OP_DUP
OP_HASH160
980c312d019510eac5c25715a97a e1732f395302
OP_EQUALVERIFY
OP_CHECKSIG

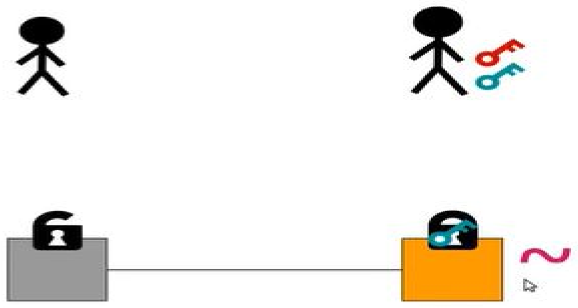
Take whatever is on the stack, take that and duplicate, or hash or etc.


signature-3045022100c51d512928e13d61a30ceb77db70e5d0b565d559f5491edb7cf23312ae84ae06f022050f0710d436a6dc8bf415dc3b86c7b66b0be5342e257b86cb5fcb203f693632801
pubkey-0438c73f3ba716a2b214141da57eb60f6fdbe8652bb5a05944010087a51460a16dcdad7bf71608e0cba3125d00c94656c18b130150bd92cbc1eaa73fd266b04c85
pubkey-0438c73f3ba716a2b2141
HASH
980c312d019510eac5c25715a97ae1732f395302
OP_DUP
OP_HASH160
OP_EQUALVERIFY
OP_CHECKSIG

This is a small computer program and you can vary it according to your events.
Bitcoin connects input and output through scripts rather than public and private keys in order to allow more complex transactions.



Consider batch of bitcoins as a lock. When you make a transaction, you unlock a batch of coins, create a new batch and put a new lock to it.



When bob sends a bitcoin batch to alice, he locks all of them up with Alice's public key. That lock can only be unlocked by Alice's signature which contains her private key.

Bitcoin Script:

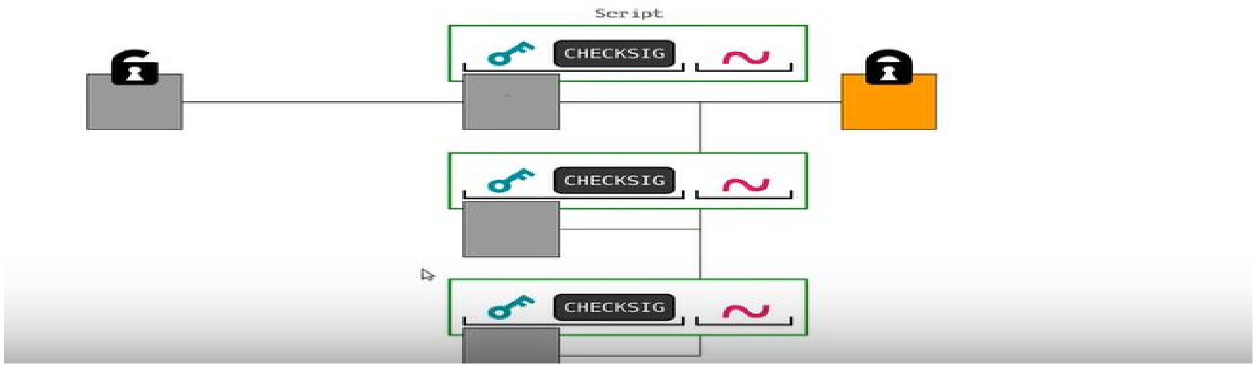
Bitcoin script is a language that was designed to be used for Bitcoin to process a variety of transactions, from payments between two people, to more complex multi-signature transactions. Let's look at how locking and unlocking work with these transactions.

Based on how you write the program, there are different types of locks

- Pay to Pubkey (simple) (35 bytes)

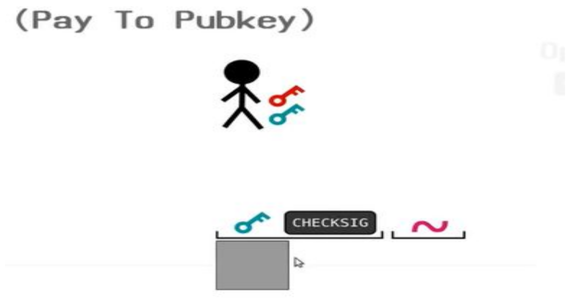
- Pay to Pubkey Hash (a little complex than above one and used commonly) (25 bytes)
- Pay to Multisig (105 bytes -- smallest)
- Pay to Script Hash

The code written to lock the bitcoin batch in the output section is called locking script, often called as the scriptPubKey and the code written to unlock the batch in the scriptSeg part of input section is called unlocking script, often called as scriptSig. Every script consists of some data and operators.

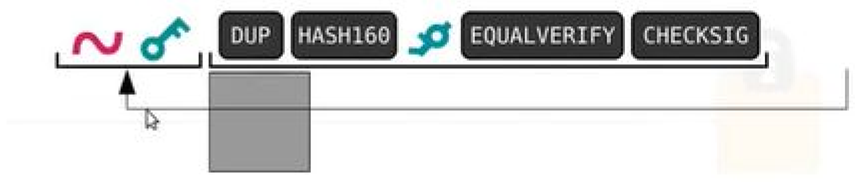


The new transaction (one in yellow) will have it's own locking script.
Any data that is found is pushed inside the stack. An op_code pops one or more data, operates on it, and pushes the result on the stack. If after running the entire script, a one is present on the top of the stack, that means the script is valid and the batch of bitcoins can be spent. If the stack is empty or a zero is present on the top, it's an invalid script. First unlocking script, locking script and then OP_checkSig is evaluated.

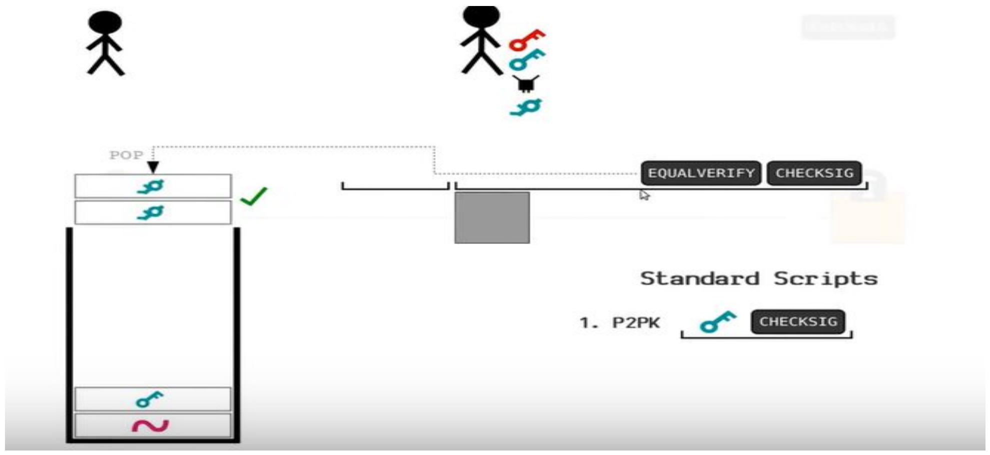
- Pay To Pubkey (P2PK)



- Pay To Pubkey Hash(P2PKH)

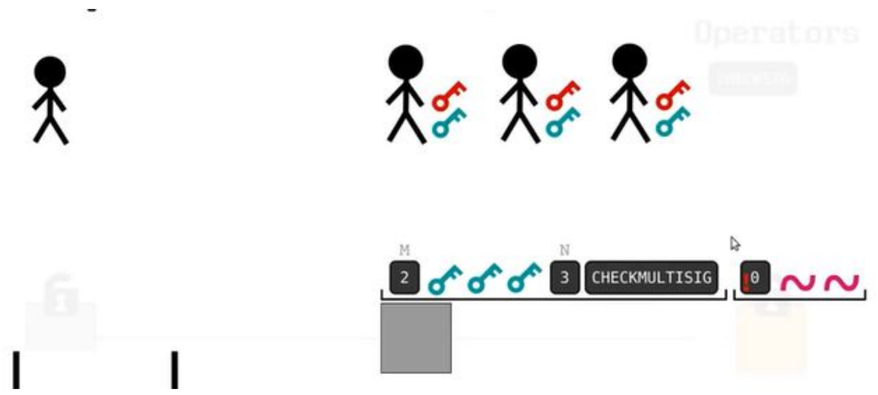


We use hash of the public key along with other operators.

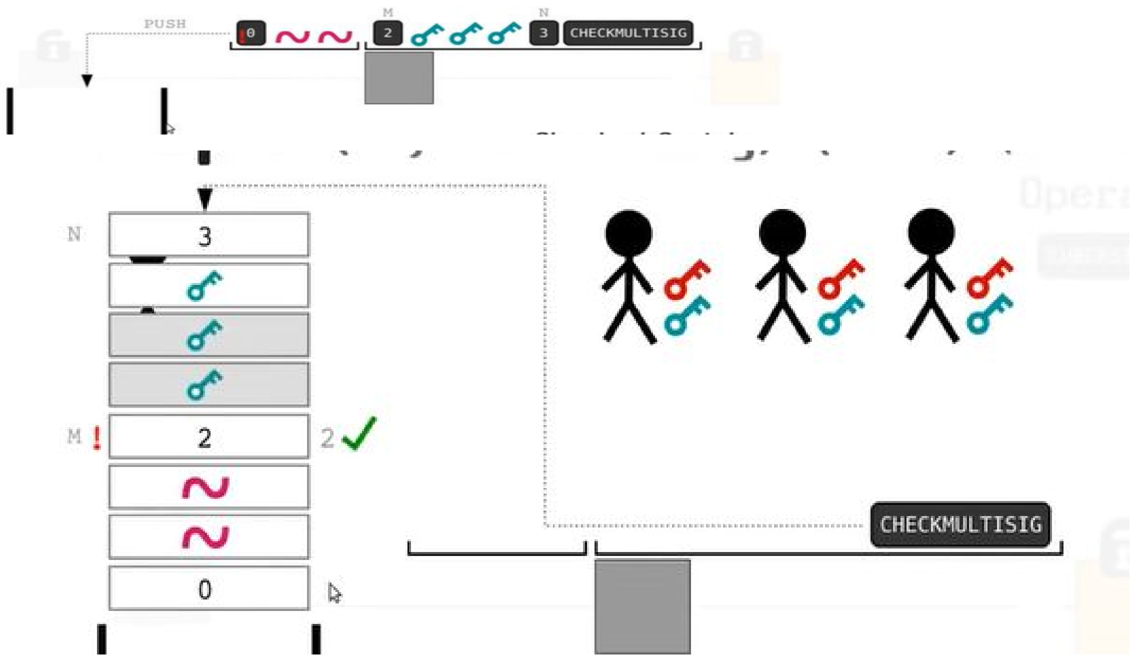


Extra checks if the public key you provided matches with the one originally provided. (hash). These are for the bitcoin addresses.

- Pay To Multisig (P2MS):
  If we want to send a transaction to a company or an organisation, then this is used.

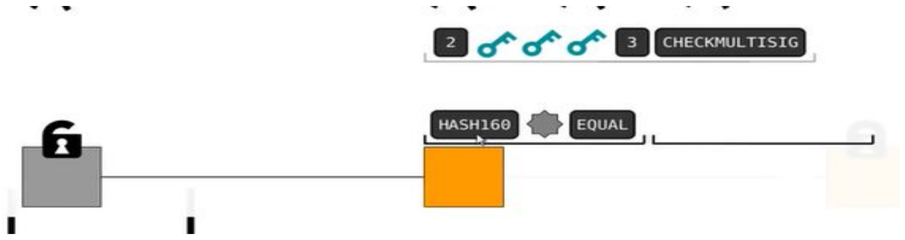


N is the number of people in the organisation, whose public keys we put in. M is the number of digital signatures required in the unlocking script (called as quorum). However there is a little bug in the checkMultisig implementation, that it pops M+1 items from the stack, so to not get the stack empty exception, we append a 0 to the signatures. Also order of signatures matter. When checkMultiSig gets a match for one public key and signature, then for the next signature, it will start checking from the next public key, it won't check the repeated signatures. Thus order matters.

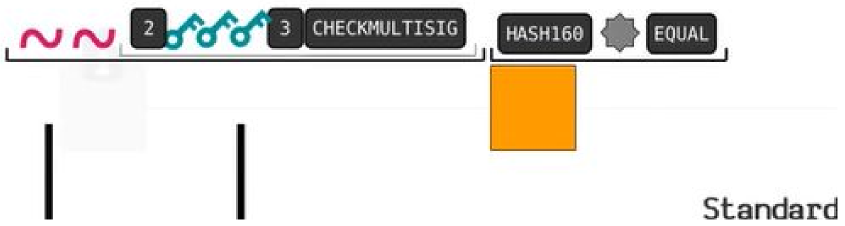


Problem with multisig is that the size of locking scripts can get really big. With 3 public keys the size is 105 bytes, but with every public key added, it gets bigger and bigger, because public keys are 33 bytes in size. And to whomsoever we give the locking script, they might have to pay due to locking data as a part of transaction fees. Thus pay to script hash is created to solve this

- Pay To Script Hash:
  Compress the locking script and hash it (20 bytes), give the hash to someone, they can lock that hash with some set of opcodes(23 bytes).

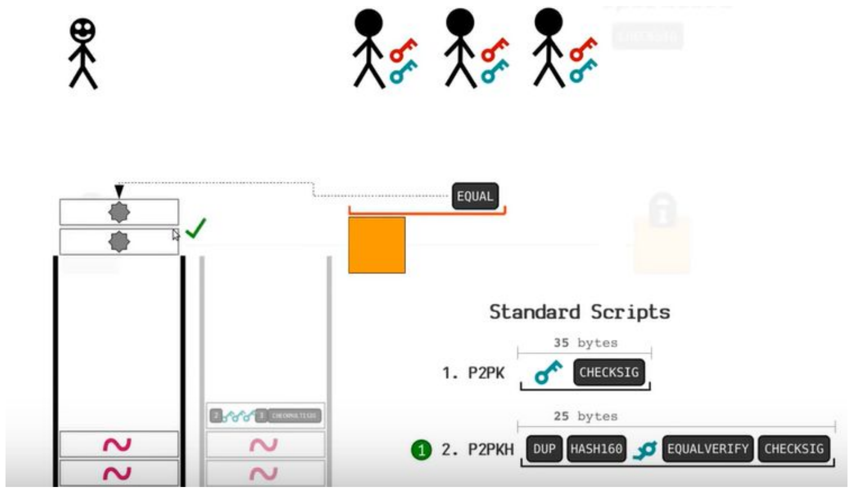


  To unlock it:



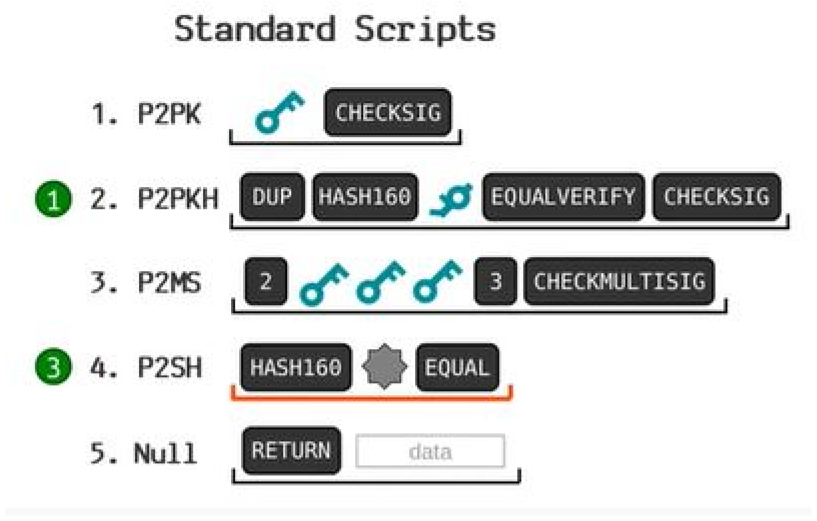
  Here, we have taken the data together, have serialized it, called as serialized locking script or redeemed script.

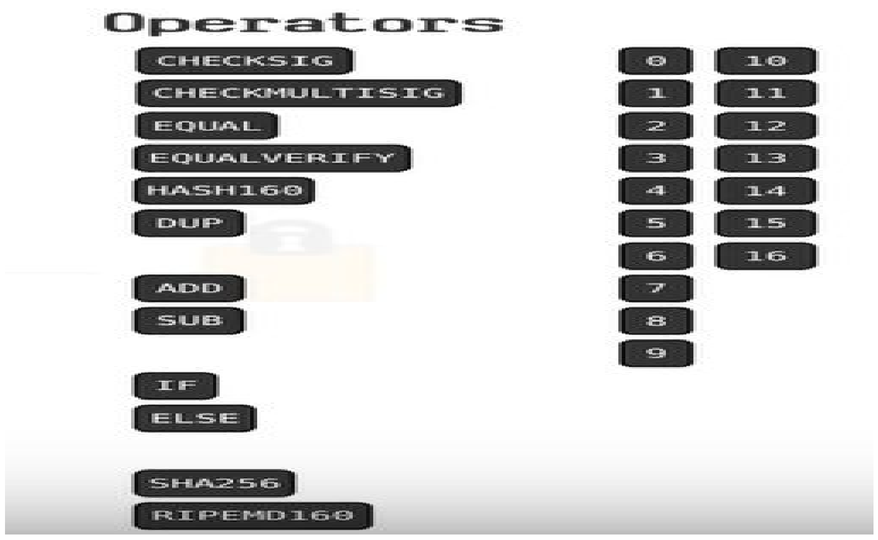
Checking if the hash of serialized locking script matches with the hash that we have provided



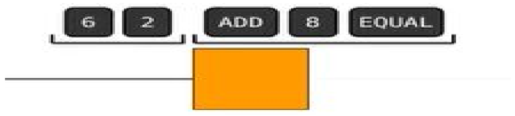
Now we take the stack we had copied and pop and we deserialize the redeemed script and run the stack again like before.

- Null data:
  Used in bitcoin
  Op_Return will always invalidate the script no matter what the batch of bitcoins or digital signature(unlocking script) is. This batch of bitcoins can never be spent. There are no bitcoins that are returned back to us, we add some transaction data in hexadecimal form in it. So basically this is the easiest way to add any arbitrary data in the blockchain.

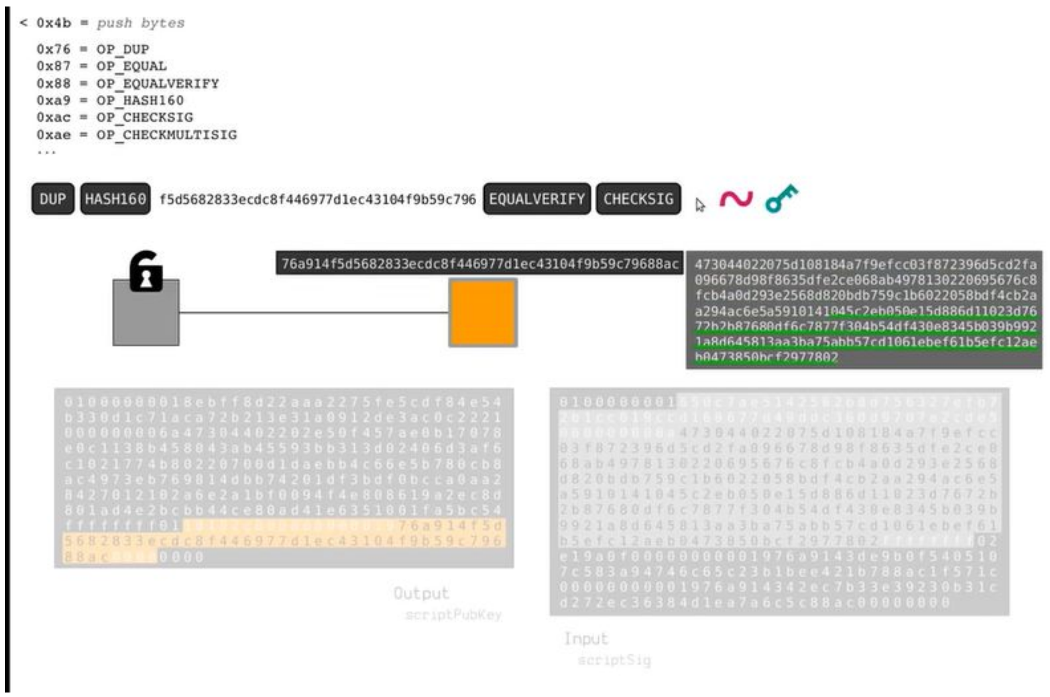




Eg :



Standard scripts are known to run quickly and verified. Non standard opcodes are not thoroughly tested. Bitcoin core rejects any non standard script to prevent any one slowing down the network by adding such time consuming scripts to it. However if using non standard script a block is blocked in to the blockchain it is considered valid(since it’s just a block, not much difference rather than a complete memory pool like bitcoin). However it’s hard to get them mined on the blockchain. Each byte in locking script correspond to a opcode. If anything is below 4b hexadecimal, it indicates, these many bytes further are a part of some data. Push these many bytes on stack.



```
< 0x4b = push bytes
  0x76 = OP_DUP
  0x87 = OP_EQUAL
  0x88 = OP_EQUALVERIFY
  0xa9 = OP_HASH160
  0xac = OP_CHECKSIG
  0xae = OP_CHECKMULTISIG
  ...

DUP HASH160 f5d5682833ecdc8f446977d1ec43104f9b59c796 EQUALVERIFY CHECKSIG

76a914f5d5682833ecdc8f446977d1ec43104f9b59c79688ac

473044022075d108184a7f9efcc03f872396d5cd2fa
096678d98f8635dfe2ce068ab4978130220695676c8
fcb4a0d293e2568d820bdb759c1b6022058bdf4cb2a
a294ac6e5a591014104 5c2eb050e15d886d11023d76
72b2b87680df6c7877f304b54df430e8345b039b992
1a8d645813aa3ba75abb57cd1061ebef61b5efc12ae
b0473850bcf2977802
```

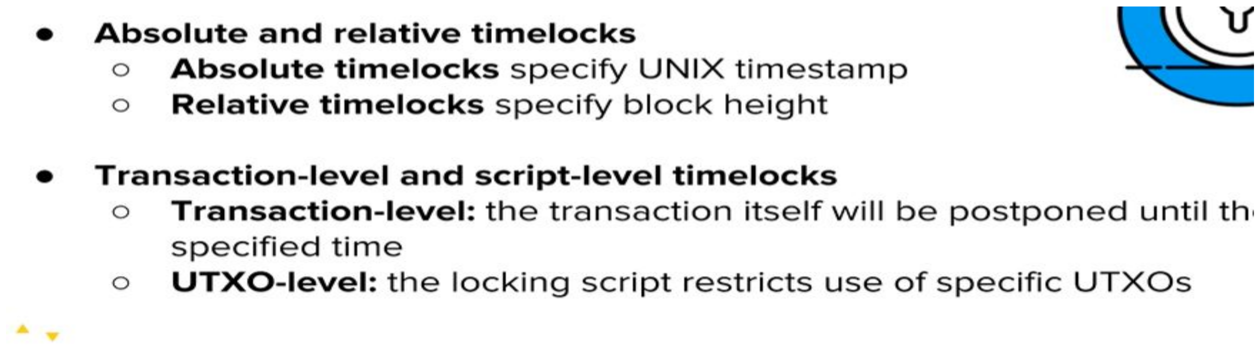
```
0100000001 8ebff8d22aaa2275fe5cdf84e54
b330d1c71aca72b213e31a0912de3ac0c2221
000000006a47304402202e50f457ae0b17078
e0c1138b45 8043ab45593bb313d02406d3af6
c1021774b80220700d1daebb4c66e5b780cb8
ac4973eb769814dbb74201df3bdf0bcca0aa2
842701210 2a6e2a1bf0094f4e808619a2ec8d
801ad4e2bcbb44ce80ad41e6351001fa5bc54
ffffffff01 [illegible] 76a914f5d
5682833ecdc8f446977d1ec43104f9b59c796
88ac [illegible] 0000
```

Output
scriptPubKey

```
0100000001 [illegible]
[illegible]
[illegible] 473044022075d108184a7f9efcc
03f872396d5cd2fa096678d98f8635dfe2ce0
68ab4978130220695676c8fcb4a0d293e2568
d820bdb759c1b6022058bdf4cb2aa294ac6e5
a591014104 5c2eb050e15d886d11023d7672b
2b87680df6c7877f304b54df430e8345b039b
9921a8d645813aa3ba75abb57cd1061ebef61
b5efc12aeb0473850bcf2977802 [illegible] 02
e19a0f00000000001976a9143de9b0f540510
7c583a94746c65c23b1bee421b788ac1f571c
00000000001976a914342ec7b33e39230b31c
d272ec36384d1ea7a6c5c88ac00000000
```

Input
scriptSig

Transaction Lock Time:
Restricts spending of funds until later time or some block height.



- **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 th
    specified time
  - **UTXO-level:** the locking script restricts use of specific UTXOs

Future reads:
https://3583bytesready.net/2016/09/06/hash-puzzes-proofs-work-bitcoin/
https://bitcoin.stackexchange.com/questions/8443/where-is-double-hashing-performed-in-bitcoin

Bitcoin operates under certain assumptions, such as an honest majority of computational power. The assumption we discussed in this last lecture was that cryptographic hashes are unique, given the properties of preimage, second preimage, and collision resistance.

Let’s say that you manage to break computer science as we know it and devise a way to make any input look like your preferred output. Formally, you can always construct an input y such that you can control the value of H(y) to be in your favor. You can create collisions at will. How can you then manipulate the Bitcoin protocol in your favor? There are several different answers, but you only need to provide one example.