# Bitcoin Mechanics

Transactions

# Recap: Bitcoin Script

```
<sig>
<pubKey>
---------------
OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG
```

# Recap: a P2PKH script

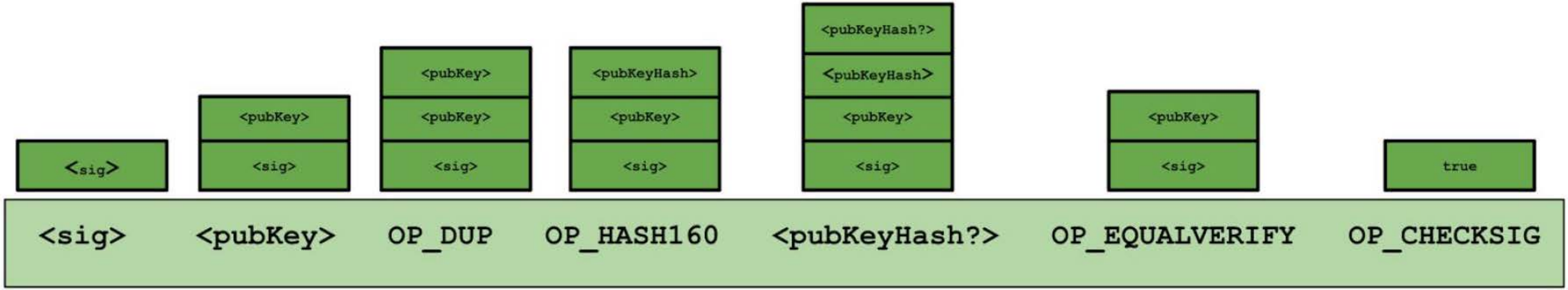
<sig> <pk> **DUP HASH256** <pkhash> **EQVERIFY CHECKSIG**

| | | |
|---|---|---|
| **stack**: | empty | init |
| | <sig> <pk> | push values |
| | <sig> <pk> <pk> | **DUP** |
| | <sig> <pk> <hash> | **HASH256** |
| | <sig> <pk> <hash> <pkhash> | push value |
| | <sig> <pk> | **EQVERIFY** |
| | 1 | **CHECKSIG** verify(pk, Tx, sig) |

⇒ successful termination

# Recap: stack operation

- The validation



# Recap: P2PKH

- Alice specifies recipient's pk in $UTXO_B$

- Recipient's pk is not revealed until UTXO is spent (the 1st usage) (some security against attacks on pk)

- Miner cannot change <$Addr_B$> and steal funds:

  invalidates the signature that created the $UTXO_B$

# Practice

- Before P2PKH, there used to be P2PK
  - That is, pay-to-public-key
  - What would the scriptpk and scriptsig be like in such transaction?

"For several weeks MtGox customers have been affected by bitcoin withdrawal issues that compounded on themselves. Publicly, MtGox declared that 'transaction malleability' caused the system to be subject to theft, and that something needed to be done by the core devs to fix it.

Gox's own workaround solution was criticized, and eventually a fix was provided by Blockchain.info. The truth, it turns out, is that the damage had already been done. At this point 744,408 BTC are missing due to malleability-related theft which went unnoticed for several years.

The cold storage has been wiped out due to a leak in the hot wallet. The reality is that MtGox can go bankrupt at any moment, and certainly deserves to as a company."

# Segregated Witness

**ECDSA malleability:**

- given (m, sig) anyone can create (m, sig') with sig ≠ sig'

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

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

⇒ leads to problems and attacks

**Segregated witness:** signature is moved to witness field in Tx

TxID = Hash(Tx without witnesses)

# Segregated Witness

- **Malleability**
  - **A gold coin got hammered, so it is not round any more; will this gold coin be used later?**
- Transaction Malleability
  - The signature of the transaction is modified a little; however, it is still a valid signature
  - Without accessing private key
  - Due to many reasons:
    - One example, OpenSSL verifies the signature not strictly
- The consequence
  - Txid will be changed

# Segregated Witness

- mtgox attack:
  - An attacker applies an account in an exchange center; and deposit bitcoins in it
  - The attacker then apply a withdraw; the exchange center will initiate a transaction
  - The transaction will be broadcast to the network; but before the transaction is confirmed in the network, the attacker received the transaction and slightly modifies the scriptsig, generate a new transaction(still valid); and broadcast to the network

# Segregated Witness

- After the hacker’s new transaction is in the blockchain (the hacker can use the bitcoin now and the original transaction will be regarded as a double-spending), he would file a complain to the exchange center, saying he hasn’t received the bitcoin yet
- The exchange center will check the blockchain with the original txid, which indeed is not included, so the exchange center will repay the hacker

# Segragated witness

- **Segregated Witness**, or **SegWit**, is the process by which the block size limit on a blockchain is increased by removing signature data from transactions that are included in each block.
  - Originally, there was no limit to the size of blocks. However, this allowed malicious actors to make up fake "block" data that was very long as a form of DoS attack
  - Block is constrained to a max size of one megabyte

# Segragated witness

- Digital signature accounts for 65% of the space in a given transaction
- Segwit ignores the signature, therefore increase the one MB limit for block sizes to a little under four MB

# Transaction Ops: multisig

- Why **OP_CHECKMULSIG?**
  - Added to bitcoin in 2011
  - requires specifying *N public keys, and a parameter M , for a threshold, M-of-N multi-signature (2-of-3; 3-of-5)*
  - M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG
  - Two-factor authentication wallet - One private key is on your primary computer, the other on your smartphone — the funds cannot be spent without a signature from both devices

# multisig

- Why **OP_CHECKMULSIG?**
  - 2-of-3: family expenditure, at least 2 of the family members agree on the expenditure
  - Escrow transaction:
    - Alice buys from bob; but they don’t trust each other
    - They both trust Carol
    - Initiate a 2-of-3 transaction

# multisig

- How does it work?
  - To redeem: OP_0 ...signatures...
  - For example: 2-of-3
  - Stack before OP_checkmultisig

3
(pubKey3)
(pubKey2)
(pubKey1)
2
(sig2)
(sig1)
0

# Multi-sig

1. Pop n off of the stack (number of public keys)
2. Pop n public keys off of the stack.
3. Pop m off of the stack (number of required signatures)
4. Pop m signatures off of the stack.
5. Pop one more element off of the stack, and ignore it. (This is a bug, but it can't be fixed because this is consensus-critical code.)
6. Loop through all of the public keys, starting with the keys at the top of the stack.
   ① For each public key, check a single signature.
   ② For the first public key checked, start with the signature closest to the top of the stack.
   ③ If it fails to verify, go to the next public key and check the same signature.
   ④ If it succeeds, go to the next public key with the next signature.
   ⑤ Note that the signatures need to be in the same order as the key's that they're signing for.
7. If all of the signatures succeeded with one of the keys, CHECKMULTISIG returns 1, otherwise 0.

# Transaction types: P2SH: pay to script hash 2012

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

Usage: payee publishes hash(redeem script) ⟵ Bitcoint addr.

payer sends funds to that address

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

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

payer can specify complex conditions for when UTXO can be spent

# Why P2SH?

- P2SH: pay-to-script-hash
  - P2SH means "pay to a script matching this hash, a script which will be presented later when this output is spent".

*Table 5-4. Complex Script without P2SH*

| | |
|---|---|
| Locking Script | 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG |
| Unlocking Script | Sig1 Sig2 |

*Table 5-5. Complex Script as P2SH*

| | |
|---|---|
| Redeem Script | 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG |
| Locking Script | OP_HASH160 <20-byte hash of redeem script> OP_EQUAL |
| Unlocking Script | Sig1 Sig2 redeem script |

# Why P2SH?

- Complex scripts are replaced by shorter fingerprint in the transaction output, making the transaction smaller
- P2SH shifts the burden of constructing the script to the recipient not the sender
- P2SH shifts the burden in data storage for the long script from the output (which is in the UTXO set and therefore impacts memory) to the input (only stored on the blockchain)
- P2SH shifts the burden in data storage for the long script from the present time (payment) to a future time (when it is spent)
- P2SH shifts the transaction fee cost of a long script from the sender to the recipient who has to include the long redeem script to spend it

# P2SH

Miner verifies:

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

(2) ScriptSig = true ⟵ script is satisfied

# Example P2SH: multisig

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

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

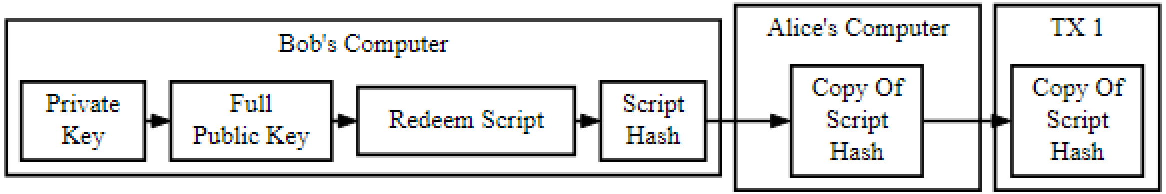
<2> <PK$_1$> <PK$_2$> <PK$_3$> <3> CHECKMULTISIG

hash gives P2SH address

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

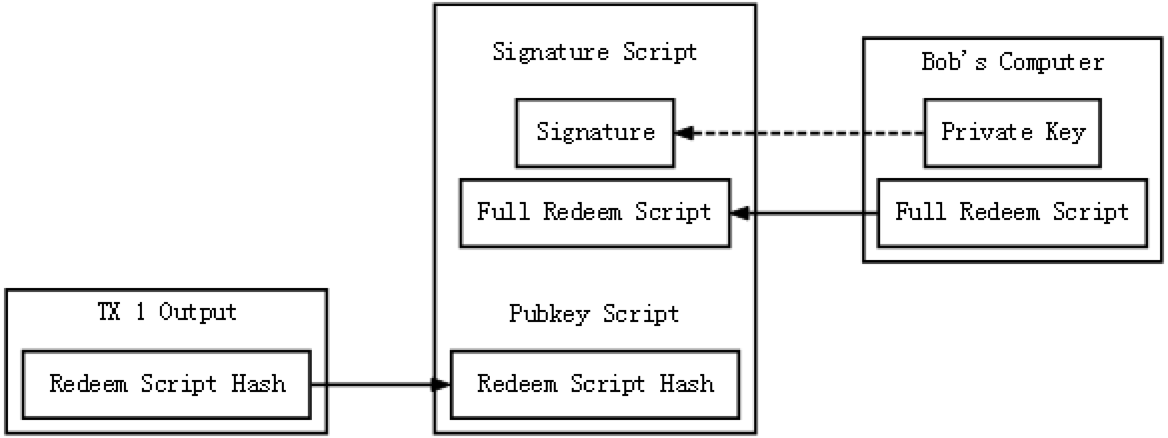
# P2SH

- Alice pays to Bob in P2SH (pay to P2SH)
  - scriptPubkey:
  - OP_HASH160 [20-byte-hash-value] OP_EQUAL



# P2SH

- Spend P2SH
  - ...signatures... {serialized script}




Spending A P2SH Output

# P2SH example

- The script is a 2-of-3 multisig script
  - go-bitcoin-multisig keys --count 3 –concise

```
KEY #1
Private key:
5JruagvxNLXTnkksyLMfgFgf3CagJ3Ekxu5oGxpTm5mPfTAPez3
Public key hex:
04a882d414e478039cd5b52a92ffb13dd5e6bd4515497439dffd691a0f12af9575fa349b5694ed3155b136f09e63975a17
Public Bitcoin address:
1JzVFZSN1kxGLTHG41EVvY5gHxLAX7q1Rh
--------------
--------------
KEY #2
Private key:
5JX3qAwDEEaapvLXRfbXRMSiyRgRSW9WjgxeyJQWwBugbudCwsk
Public key hex:
046ce31db9bdd543e72fe3039a1f1c047dab87037c36a669ff90e28da1848f640de68c2fe913d363a51154a0c62d7adea1
Public Bitcoin address:
14JfSvgEq8A8S7qcvxeaSCxhn1u1L71vo4
--------------
--------------
KEY #3
Private key:
```

# P2SH Example

- go-bitcoin-multisig address --m 2 --n 3 --public-keys 04a882d414e478039cd5b52a92ffb13dd5e6bd4515497439dffd691a0f12af9575fa349b5694ed3155b136f09e63975a1700c9f4d4df849323dac06cf3bd6458cd, 046ce31db9bdd543e72fe3039a1f1c047dab87037c36a669ff90e28da1848f640de68c2fe913d363a51154a0c62d7adea1b822d05035077418267b1a1379790187, 0411ffd36c70776538d079fbae117dc38effafb33304af83ce4894589747aee1ef992f63280567f52f5ba870678b4ab4ff6c8ea600bd217870a8b4f1f09f3a8e83

# P2SH example

- -------------------- Your *P2SH ADDRESS* is: 347N1Thc213QqfYCz3PZkjoJpNv5b14kBd

  Give this to sender funding multisig address with Bitcoin.

  -------------------- Your *REDEEM SCRIPT* is:
  524104a882d414e478039cd5b52a92ffb13dd5e6bd4515497439dffd691a0
  f12af9575fa349b5694ed3155b136f09e63975a1700c9f4d4df849323dac06
  cf3bd6458cd41046ce31db9bdd543e72fe3039a1f1c047dab87037c36a669f
  f90e28da1848f640de68c2fe913d363a51154a0c62d7adea1b822d0503507
  7418267b1a13797901874104 11ffd36c70776538d079fbae117dc38effafb3
  3304af83ce4894589747aee1ef992f63280567f52f5ba870678b4ab4ff6c8ea
  600bd217870a8b4f1f09f3a8e8353ae
- Keep private and provide this to redeem multisig balance later. --------------------

| **Description** | **redeemScript bytes** |
| --- | --- |
| OP_2 | 52 |
| Push 65 bytes to stack | 41 |
| <pubKeyA> | 04a882d414e478039cd5b52a92ffb13dd5e6bd4515497439dffd691a0f12af9575fa349b5694ed3155b136f09e63975a1700c9f4d4df849323dac06cf3bd6458cd |
| Push 65 bytes to stack | 41 |
| <pubKeyB> | 046ce31db9bdd543e72fe3039a1f1c047dab87037c36a669ff90e28da1848f640de68c2fe913d363a51154a0c62d7adea1b822d05035077418267b1a1379790187 |
| Push 65 bytes to stack | 41 |
| <pubKeyC> | 0411ffd36c70776538d079fbae117dc38effafb33304af83ce4894589747aee1ef992f63280567f52f5ba870678b4ab4ff6c8ea600bd217870a8b4f1f09f3a8e83 |
| OP_3 | 53 |
| OP_CHECKMULTISIG | ae |

# P2SH

- P2SH address:
  - redeemScriptHash = RIPEMD160(SHA256(redeemScript))
  - P2SHAddress := base58check.Encode("05", redeemScriptHash)

- Alice’s Transaction

| Description | | Hex Bytes |
|---|---|---|
| Version byte | | 01000000 |
| Input count | | 01 |
| Previous tx hash (reversed) | | acc6fb9ec2c3884d3a12a89e7078c83853d9b7912281cefb14bac00a2737d33a |
| Output index | | 00000000 |
| scriptSig length of 138 bytes | | 8a |
| scriptSig | Push 71 bytes to stack | 47 |
| | <signature> | 304402204e63d034c6074f17e9c5f8766bc7b5468a0dce5b69578bd08554e8f21434c58e02<br>20763c6966f47c39068c8dcd3f3dbd8e2a4ea13ac9e9c899ca1fbc00e2558cbb8b01 |
| | Push 65 bytes to stack | 41 |
| | <pubKey> | 0431393af9984375830971ab5d3094c6a7d02db3568b2b06212a7090094549701bbb9e84d<br>9477451acc42638963635899ce91bacb451a1bb6da73ddfbcf596bddf |
| Sequence | | ffffffff |
| No. of outputs | | 01 |
| Amount of 65600 in LittleEndian | | 4000010000000000 |
| scriptPubKey length of 23 bytes | | 17 |
| scriptPubKey | OP_HASH160 | a9 |
| | Push 20 bytes to stack | 14 |
| | redeemScriptHash | 1a8b0026343166625c7475f01e48b5ede8c0252e |
| | OP_EQUAL | 87 |
| locktime | | 00000000 |

# P2SH

- Bob’s Transcation

| | | |
|---|---|---|
| scriptSig length of 349 bytes | | fd5d01 |
| scriptSig | OP_0 | 00 |
| | Push 71 bytes | 47 |
| | <sig A> | 30440220762ce7bca626942975bfd5b130ed3470b9f538eb2ac120c2043b445709369628022051d73c80328b543f744aa64b7e9ebefa7ade3e5c716eab4a09b408d2c307ccd701 |
| | Push 72 bytes | 48 |
| | <sig C> | 3045022100abf740b58d79cab000f8b0d328c2fff7eb88933971d1b63f8b99e89ca3f2dae602203354770db3cc2623349c87dea7a50cee1f78753141a5052b2d58aeb592bcf50f01 |
| | OP_PUSHDATA1 | 4c |
| | Push 201 bytes | c9 |
| | <redeemScript> | 524104a882d414e478039cd5b52a92ffb13dd5e6bd4515497439dffd691a0f12af9575fa349b5694ed3155b136f09e63975a1700c9f4d4df849323dac06cf3bd6458cd41046ce31db9bdd54 |

# P2SH

- Bob’s Transaction

| | | |
|---|---|---|
| No. of outputs | | 01 |
| Amount of 55600 in LittleEndian | | 30d9000000000000 |
| scriptPubKey length of 25 bytes | | 19 |
| scriptPubKey | OP_DUP | 76 |
| | OP_HASH160 | a9 |
| | Push 20 bytes | 14 |
| | <pubKeyHash> | 569076ba39fc4ff6a2291d9ea9196d8c08f9c7ab |
| | OP_EQUALVERIFY | 88 |
| | OP_CHECKSIG | ac |

# P2SH

- Combining Alice’s script pubkey & Bob Scirptsig
- <OP_0> <sig A> <sig C> <redeemScript> <OP_HASH160> <redeemScriptHash> <OP_EQUAL>

# P2SH

- Stepping through this script:
- *OP_0* and the two signatures are added to the stack, kept for later.
- The redeemScript is added to the stack.
- *OP_HASH160* hashes our redeemScript.
- redeemScriptHash is added to the stack.
- *OP_EQUAL* will compare OP_HASH160(redeemScript) and redeemScriptHash and check for equality. This confirms that our spending transaction is providing the correct redeemScript.
- Now the redeemScript can be evaluated:
- <OP_2> <A pubkey> <B pubkey> <C pubkey> <OP_3> <OP_CHECKMULTISIG>
- *OP_CHECKMULTISIG* will look at the 3 public keys and 2 signatures in the stack, and compare them one by one. As stated earlier, the order of signatures matters here and must match the order that the public keys were provided in.

# Base58

- Discrepancy:
  - In command line:
    - --destination 347N1Thc213QqfYCz3PZkjoJpNv5b14kBd
  - In Alice's transaction:
    - 1a8b0026343166625c7475f01e48b5ede8c0252e
  - 1a… is the hashed the result of redeem script
  - 347…. is the base58 encoded version of 1a8….
  - Why should the hash resulted be encoded again?
  - What is Base58?

# Base58

- With version 05, resulting in Base58Check encoded addresses that start with a "3"

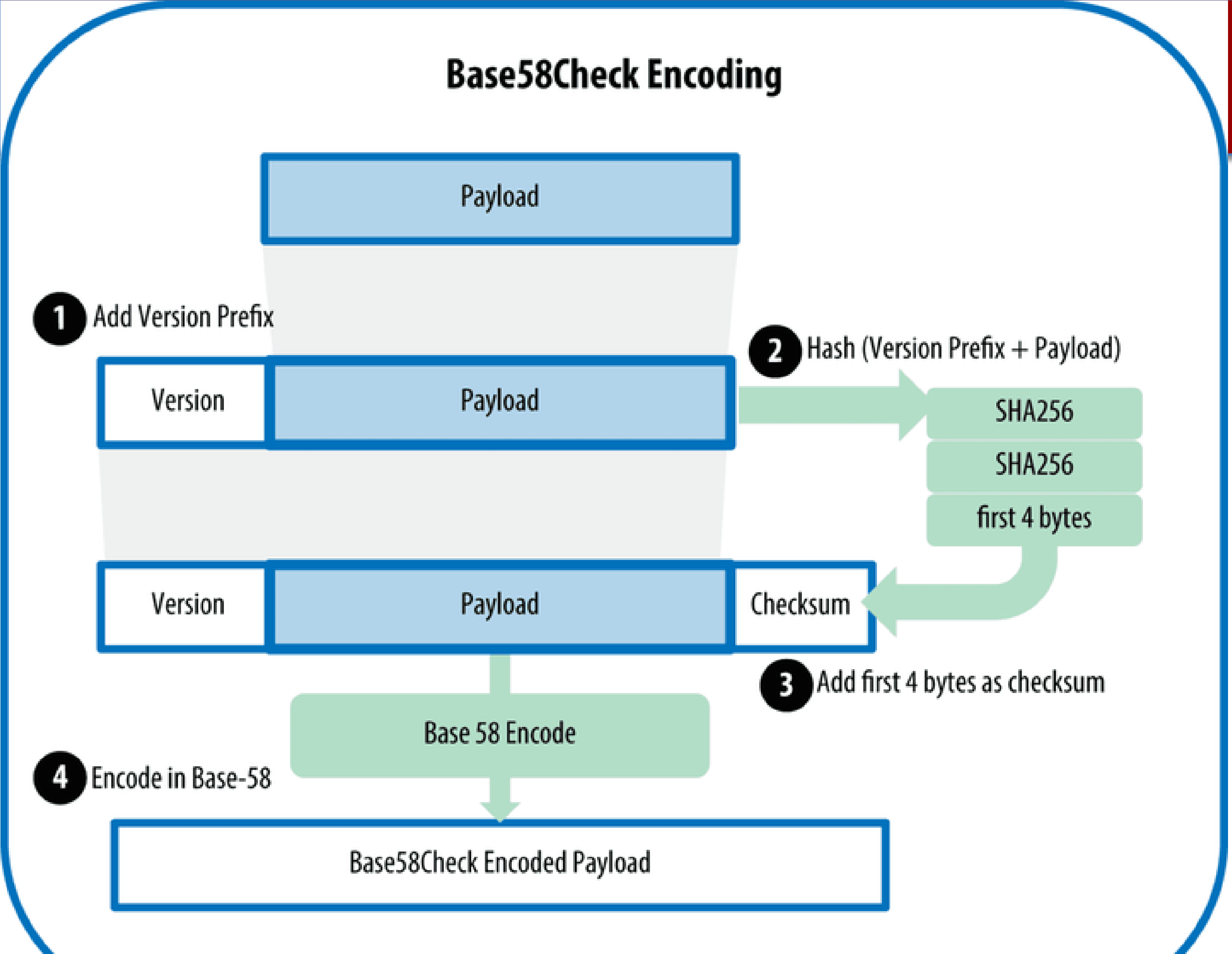
```
[redacted]@ubuntu:~/Desktop$ ./bx base58check-decode 347N1Thc213QqfYCz3PZkjoJpNv5b14kBd
wrapper
{
    checksum 2891525823
    payload 1a8b0026343166625c7475f01e48b5ede8c0252e
    version 5
}
[redacted]@ubuntu:~/Desktop$ ./bx base58check-encode 1a8b0026343166625c7475f01e48b5ede8c0252e
13RM5vDAU6j2kVqmrwiyL7SNfrdMxSEegV
[redacted]@ubuntu:~/Desktop$ ./bx base58check-encode 1a8b0026343166625c7475f01e48b5ede8c0252e --version 05
347N1Thc213QqfYCz3PZkjoJpNv5b14kBd
```

# Base58

- Base58 alphabet consists of the following characters:
- 123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

Base58 Mapping table

| Byte | Character | Byte | Character | Byte | Character | Byte | Character |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 |
| 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 |
| 8 | 9 | 9 | A | 10 | B | 11 | C |
| 12 | D | 13 | E | 14 | F | 15 | G |
| 16 | H | 17 | J | 18 | K | 19 | L |
| 20 | M | 21 | N | 22 | P | 23 | Q |
| 24 | R | 25 | S | 26 | T | 27 | U |
| 28 | V | 29 | W | 30 | X | 31 | Y |
| 32 | Z | 33 | a | 34 | b | 35 | c |
| 36 | d | 37 | e | 38 | f | 39 | g |
| 40 | h | 41 | i | 42 | j | 43 | k |
| 44 | m | 45 | n | 46 | o | 47 | p |
| 48 | q | 49 | r | 50 | s | 51 | t |
| 52 | u | 53 | v | 54 | w | 55 | x |
| 56 | y | 57 | z | | | | |


Base58Check Encoding
Payload
1 Add Version Prefix
Version
Payload
2 Hash (Version Prefix + Payload)
SHA256
SHA256
first 4 bytes
Version
Payload
Checksum
3 Add first 4 bytes as checksum
Base 58 Encode
4 Encode in Base-58
Base58Check Encoded Payload

END OF LECTURE