Bitcoin Mechanisms

Kianoush Arshi **810198438**

Part 1-Address Generation

In this section, we generate a new address on the Bitcoin testnet.

Generating a Base58 (P2PKH) Bitcoin Testnet Address

To generate a Base58 Bitcoin testnet address, we need to first generate a private key and then use it to derive the corresponding public key and address. The private key is a random 256-bit number that is used to sign Bitcoin transactions, while the public key is derived from the private key and is used to verify the signature. The address is a Base58-encoded representation of the public key.

It is important to note that Bitcoin testnet addresses have a different prefix than mainnet addresses to prevent confusion between the two networks. Testnet addresses start with the letter "m" or "n," while mainnet addresses start with the number "1" or "3."

Differences Between Mainnet and Testnet Addresses

The main difference between mainnet and testnet addresses is that testnet addresses are used for testing and development purposes only. Transactions on the testnet do not involve real bitcoins and are not recorded on the mainnet blockchain. Testnet addresses use a simpler algorithm for generating private keys, which makes them easier to work with during testing and development. In contrast, mainnet addresses use a more complex algorithm to generate private keys, which provides a higher level of security against hacking and theft.

Method for Generating a Base58 Bitcoin Testnet Address using Python

To generate a Base58 Bitcoin testnet address in Python, we can use the following steps:

1. Generate a random 256-bit number to use as the private key.
2. Derive the public key from the private key using the Elliptic Curve Digital Signature Algorithm (ECDSA).
3. Apply SHA-256 and RIPEMD160 hashing algorithms to the public key to generate a hash.
4. Add a version byte prefix to the hash. For testnet addresses, this is 0x6f.
5. Apply a checksum to the version byte prefix and hash using the SHA-256 hashing algorithm.
6. Concatenate the version byte prefix, hash, and checksum to form the raw address.
7. Encode the raw address using Base58 encoding to obtain the final testnet address.

```python
def generate_address(_public_key):
    # P2PKH Address (6f for testnet and 00 for mainnet)
    version_byte_prefix = b'\x6f'

    # Hash the given public key
    public_key_bytes = bytes.fromhex(_public_key)
    sha256_1 = hashlib.sha256(public_key_bytes).digest()
    ripemd160 = hashlib.new('ripemd160')
    ripemd160.update(sha256_1)
    hash160 = ripemd160.digest()

    # Add version byte to the beginning of the RIPEMD-160 hash
    hash160_with_version = version_byte_prefix + hash160

    sha256_2 = hashlib.sha256(hashlib.sha256(hash160_with_version).digest()).digest()

    hash160_with_checksum = hash160_with_version + sha256_2[:4]  #(checksum)

    return base58.b58encode(hash160_with_checksum).decode()
```
Executed at 2023.05.26 12:53:04 in 200ms

```python
# Verify using: https://learnmeabitcoin.com/technical/address
address = generate_address(public_key)
print('Address: ', address)
compressed_address = generate_address(compressed_public_key)
print('Compressed Address: ', compressed_address)
```
Executed at 2023.05.26 12:53:04 in 248ms

```
Address:  mwvJj5pGoMZ7ZiKJxqjwWgPuDxFyi274go
Compressed Address:  msacutgG8zpyff8wb6E6qCQh8vVH4wJCbZ
```

Difference: On mainnet, we extend "0x80" to the left of the hashed public key and then generate the checksum to create the address(which must be converted to base58) but on testnet, we add "0x6f" to the left of the hashed public key

Generating Vanity Address

Our approach relies on brute force. We generate addresses using the same code as the previous question and check if the bytes in positions 2 to 4 match those in the input. We continue generating new addresses until we find one that meets our criteria (Here it's having Jee at the beginning of the address).

```
def generate_vanity_address(initial):
    result_arr = generate_G_coeffs()
    while True:
        _vanity_private_key = hex(random.getrandbits(256))
        _x, _y = secp256k1(int(_vanity_private_key[2:], 16), result_arr)

        # Zero-pad _x to a fixed width of 64 hexadecimal digits (32 bytes)
        _x_hex = '{:0>64x}'.format(_x)

        _vanity_public_key = '03' + _x_hex if int(hex(_y), 16) % 2 == 1 else '02' + _x_hex
        _vanity_address = generate_address(_vanity_public_key)
        print(_vanity_address)
        if _vanity_address[1:4] == initial:
            return _vanity_address, _vanity_public_key, _vanity_private_key
Executed at 2023.05.26 12:53:05 in 130ms

vanity_address, vanity_public_key, vanity_private_key = generate_vanity_address('Jee')
print('vanity address : %s' % vanity_address)
print(vanity_public_key)
wifVanity = to_wif(vanity_private_key[2:])
print('wif : %s' % wifVanity)
Executed at 2023.05.26 12:53:08 in 3s 713ms

mx42gYwMPFayfPvcRHcyY16VERYTp5VLFk
mkGihdm58E3oo1eCat9M7PQqJ1rQuB5fa9
mi5FrqJptCY8MeLMaLLHtXMyvDBodENYU4
mpxpiGBzjCE6fkjVc7JJb2VNnSwPzmxUWL
mu3tBVeJxzD4DLUyWMKtmz8jshnBJok4Ju
mwcRdhr4eSPdjCLT35YCgZT5UNuywFQzjC
mwWQfCsrvVVXop1KshYuEW57tcCcbw9Xo4
n1s4nff4AnRzGRBCFVsdzIvDrPp7UvgtP9
```

Part 2-Transaction Execution

In this part, we submit different transactions on the Bitcoin testnet.

This is a helpful stack visualizer: Bitcoin IDE

0-Faucets

Bitcoin faucets are websites or applications that give away small amounts of Bitcoin or other cryptocurrencies for free. These faucets are often used for testing purposes, such as testing transactions or blockchain applications, and are typically funded by donations or advertising revenue. I used coinfaucet.eu btc-testnet faucet and Bitcoin Testnet Faucet for funding my project.

1-Unspendable and Public TXOs

In this part, we create a transaction that has two outputs:

1. This output can't be used in any other transaction as an input.
2. This one can be used by everyone as input for their own transaction.

1. My Address: `mvw1uJQNoPAwwcEDZmdfax3F1GveG28Ewo`

2. My Private Key:

```
4a03b64a24c0eab2b212652f2b56ca4f5e2216f349bef3b6535ec591145b18a1
```

3. Transaction Hash:

```
53d7af7e0f7318b01f8dc5044a3318e803f7aaf2c24f18f43389e6854527cfde
```

4. Transaction Spending Hash:

```
f3f1395b51da572f1b6b5594e2c4206c67b98fa2ccb7879f8d775ec86b131eed
```

2-P2MS

In this part, we execute a MultiSig transaction.

Here's how OP_MULTISIG works:

The script pushes the required number of signatures (m) onto the stack, followed by the corresponding public keys (n keys in total).

The script then pushes the number of signatures provided (n') onto the stack, followed by the actual signatures (n' signatures in total).

The script checks that n' is less than or equal to n, and that m is less than or equal to n'.

The script then verifies that the provided signatures match the corresponding public keys, using the OP_CHECKSIG opcode for each signature and public key pair.

If the required number of signatures are valid, the script returns a boolean true value to allow the transaction to be spent. Otherwise, it returns a boolean false value.

1. **My Address:** `mjifgpTzeHEF3ieEQ7zBvPVsyMVnS2v7Kq`
2. **My Private Key(WIF):**
   ```
   92uLz2m5S89jQoFbX88fjudD4dZwjMdpBnut4ZbvX73c58nGPBm
   ```
3. **First Private Key(WIF):**
   ```
   92W78XPjA2fTR4qGcFQed5bAv19mJZkzDkXRLHyZob3DBRMgiPU
   ```
4. **Second Private Key(WIF):**
   ```
   92Y775cuvvvdjsoiRdu3ntEV3d22Zk1AR1HvqR2DDU8HCW1xkRo
   ```
5. **Third Private Key(WIF):**
   ```
   93TgTKfXUyTEcGXsWjuwptBs9XgoXrfb1vw6AJpi68a3eQbz3aa
   ```
6. **Transaction Hash:**

```
ed778d3dae7a992a94f754cd013fa3e36ffd91062624c0d8167a506e89ba773a
```

7. **Transaction Spending Hash:**

```
8. 71a184d310728808ad71b5587b889b07d575a28a4512c9f5fc7d7d4c86a82309
```

3-Custom Conditions on Transactions

In this part, we add some custom conditions to the transaction. Anyone that has the prime numbers can use the transactions UTXO.

1. **My Address:** `mjifgpTzeHEF3ieEQ7zBvPVsyMVnS2v7Kq`
2. **My Private Key(WIF):**

```
3. 92uLz2m5S89jQoFbX88fjudD4dZwjMdpBnut4ZbvX73c58nGPBm
```

4. **Prime Numbers: 13 & 11**
5. **Transaction Hash:**

```
6. 74b76b48dfc9f4aac53fc1300881bcd6da17b9b6c0b065b234e04b27431dc769
```

7. **Transaction Spending Hash:**

```
8. bedc25abd69f87ad1ea15acb2dea44de29727ea227d0f6a338c52ceda01fdde5
```

4-Same as Transaction 1

Part 3-Forking Bitcoin

In this part, we create another block after block #8438 instead of the one on the longest chain. To put it simply, we mine another block #8439.

In Bitcoin mining, the miner performs a computationally-intensive process called proof-of-work, which involves finding a nonce (a random number) that, when combined with the block header, produces a hash that meets a specific difficulty target.

The block header is an 80-byte piece of data that is included in each block in the Bitcoin blockchain. It contains several pieces of information that are used to uniquely identify the block and link it to the previous block in the chain. Here's a breakdown of the different fields in the block header:

Version: A 4-byte field that specifies the version of the block.

Previous block hash: A 32-byte field that contains the hash of the previous block in the blockchain. This links the current block to the previous block and ensures that the blockchain is a continuous, unbroken chain of blocks.

Merkle root: A 32-byte field that contains the root hash of the Merkle tree of transactions in the block. This allows nodes on the network to efficiently verify that a specific transaction is included in the block without having to download and verify all of the other transactions in the block.

Timestamp: A 4-byte field that specifies the time at which the block was created.

Difficulty target(threshold): A 4-byte field that specifies the difficulty target for the block. This is used in the proof-of-work algorithm to ensure that the block takes a certain amount of computational effort to create.

Nonce: A 4-byte field that is used in the proof-of-work algorithm to vary the block header and produce a hash that meets the difficulty target.

My block

Previous Block number: 8438

Hash: 000000009505ec73031ca38a87a4cc884075f1f75096845e24b1b74f5d133e20

Coinbase hexadecimal data: 383130313938343333384b69616e6f7573684172736869

My address: 142KTHRxFNjDMFmGHz9cPRoBucZR94qiAD

block body:

010000000100000000000000000000000000000000000000000000000000000000000000000fffff

fff17163831303139383433384b69616e6f7573684172736869ffffffff0100f2052a010000001976a

9142128327e040a0aed617030577a812bfbfb210ffa88ac00000000

merkle_root: d560ae68ec6e269fea90c8b31ccc4256d77866017b6bf99f7622632b5d144202

Threshold: 0001000000000000000000000000000000000000000000000000000000000000

Nonce found: 29095

Block hash: 00009bbff8b5b272b5ed7ba9ddf4732faa60b4da71372bd26d7b4dda7fbb4e42

Block header:

02000000203e135d4fb7b1245e849650f7f1754088cca4878aa31c0373ec0595000000000242145

d2b6322769ff96b7b016678d75642cc1cb3c890ea9f266eec68ae60d5f2b770640000011fa771000

0

Block body:

01000000010000000000000000000000000000000000000000000000000000000000000000fffff

fff17163831303139383433384b69616e6f7573684172736869ffffffff0100f2052a010000001976a

9142128327e040a0aed617030577a812bfbfb210ffa88ac00000000

```
Block number:  8438
Hash:  000000009505ec73031ca38a87a4cc884075f1f75096845e24b1b74f5d133e20
Coinbase hexadecimal data:  383130313938343338 4b69616e6f7573684172736869
My address:  142KTHRxFNjDMFmGHz9cPRoBucZR94qiAD
block body:  010000000100000000000000000000000000000000000000000000000000000000000
merkle_root:  d560ae68ec6e269fea90c8b31ccc4256d77866017b6bf99f7622632b5d144202
Threshold:  0001000000000000000000000000000000000000000000000000000000000000
Nonce found :  29095
Block with hash  00009bbff8b5b272b5ed7ba9ddf4732faa60b4da71372bd26d7b4dda7fbb4e42
Block header:  02000000203e135d4fb7b1245e849650f7f1754088cca4878aa31c0373ec0595000
Block body:  010000000100000000000000000000000000000000000000000000000000000000000
```

Note:

Most of the transactions can be seen in this wallet: