# Question 1

## Part 1 (generate bitcoin private key)

For this part, we generate a random 256 bit(32 byte) number for private key.

random\_number = random.getrandbits(256)

private\_key = random\_number.to\_bytes(32, byteorder="big")

To generate the Wallet Import Format (WIF) of a private key, the following steps can be followed:

1. Start with the private key in its hexadecimal format.
2. Add the test network prefix (0xEF) to the beginning of the private key (0x80 for main network).
3. Perform a double hash operation on the extended private key using a cryptographic hash function, such as SHA-256.
4. Take the first 4 bytes of the resulting hash and append them to the extended private key.
5. Encode the extended private key, including the appended checksum, into a base58 encoding algorithm .
6. The resulting string is the WIF representation of the private key.[[1]](#footnote-1)

By following these steps, the private key can be converted into a WIF format that is commonly used for importing private keys into cryptocurrency wallets.

To generate a public key based on a given private key, the following steps can be followed:

1. Utilize the ECDSA (Elliptic Curve Digital Signature Algorithm) to generate a public key corresponding to the provided private key.
2. Add the public key prefix (0x04) to the beginning of the generated public key. This prefix distinguishes uncompressed public keys.
3. Apply a hash function, such as SHA-256, followed by RIPEMD160, to the public key. This process results in a hashed value.
4. Prepend the test network prefix (0x6f) to the hashed value obtained in the previous step. This prefix is used to identify the network or purpose for which the public key is intended.
5. Calculate the double hash of the extended key using SHA-256, obtaining a new hash result.
6. Append the first 4 bytes (checksum) of the double hash to the end of the extended key.
7. The resulting string is the public key derived from the given private key.

By following these steps, the public key can be obtained from a private key.

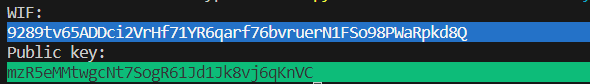


Figure test result

## Part 2 (generate bitcoin address)

In the following section, we focus on generating a vanity address based on a user-defined input string. A vanity address is an address that contains a specific subset of characters, starting from the second character and ending at the [2+n]th character of the Bitcoin public key address. To determine the number of attempts required to achieve the desired vanity address, we implement a code that iteratively searches for a matching address.

Please note that when the input string is longer than three characters, the calculation process can be time-consuming due to the increased complexity of finding a matching subset.

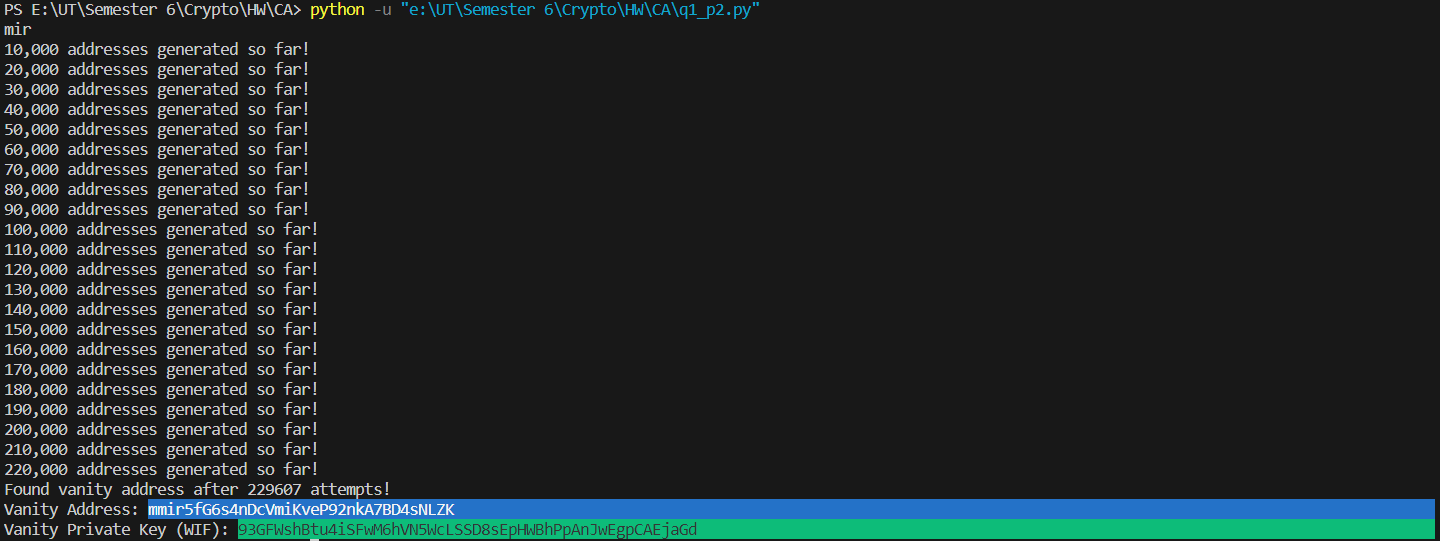


Figure . Vanity address generation result starting with “mir”

Found vanity address after 229607 attempts!

Vanity Address: mmir5fG6s4nDcVmiKveP92nkA7BD4sNLZK

Vanity Private Key (WIF): 93GFWshBtu4iSFwM6hVN5WcLSSD8sEpHWBhPpAnJwEgpCAEjaGd

# Question 2

To acquire a certain amount of Bitcoin on the test network, we can utilize a faucet website[[2]](#footnote-2).

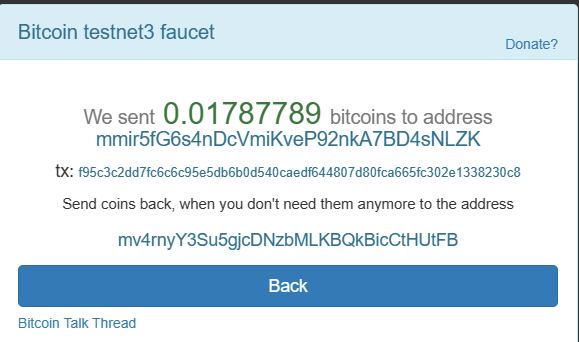


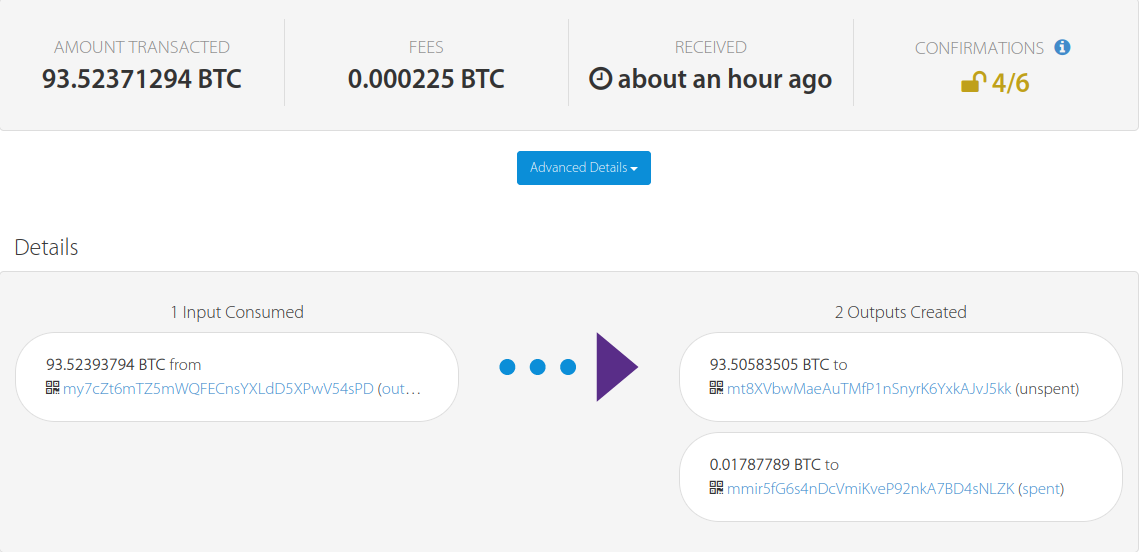
Figure . Bitcoin Received on the Provided Address

txid\_to\_spend = ('f95c3c2dd7fc6c6c95e5db6b0d540caedf644807d80fca665fc302e1338230c8')

After initiating a transaction, it is essential to wait for a certain number of blocks to be added to the blockchain to ensure the transaction is securely recorded and considered final. Generally, a common practice is to wait for at least six blocks, which provides a sufficient level of confirmation for the transaction.

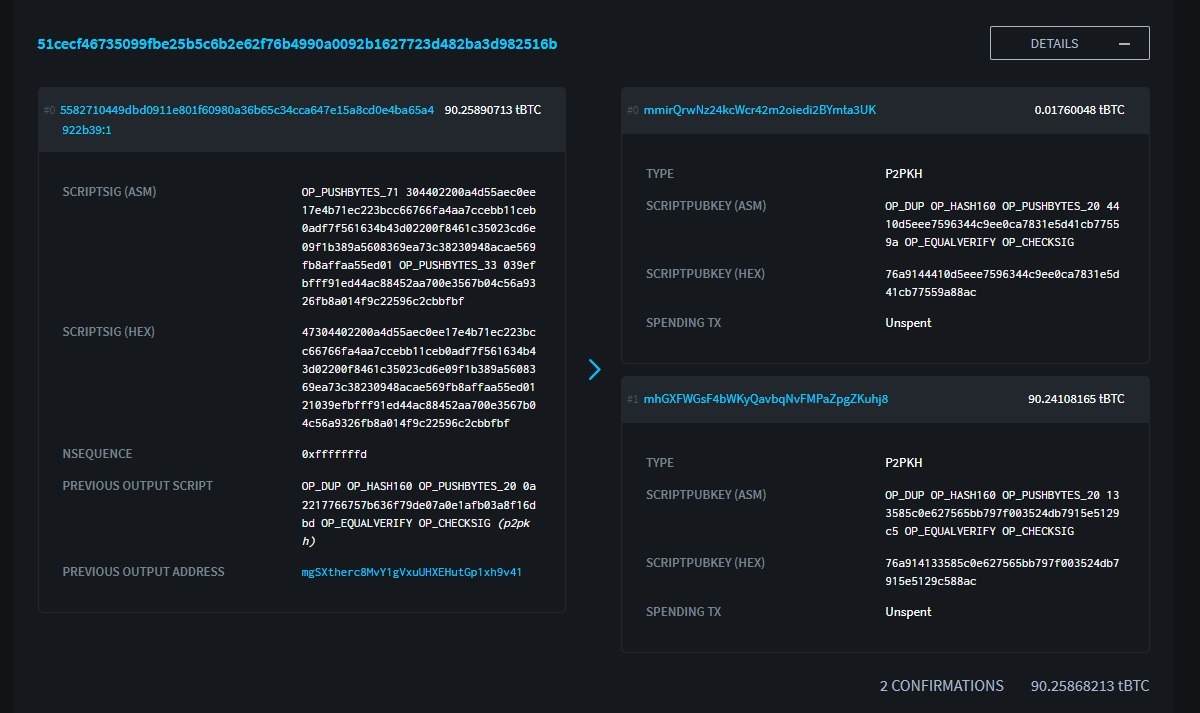
We initiated a transaction with one input and two outputs. Figure 4 illustrates the structure of this transaction. The purpose of the two outputs is as follows:

1. The first output, also shown in Figure 4, is spendable and can be used to transfer the specified Bitcoin amount to the provided address. This output allows the Bitcoin to be sent to the designated recipient.
2. The second output, as depicted in Figure 4, is not spendable and is utilized to return the Bitcoin amount to the faucet address. This ensures that any remaining funds or change from the transaction are returned to the original source.



Figure

To complete the transaction.py file, we utilized a specific [website](https://blockstream.info/testnet) to study our transaction and analyze the script used within it. This website provided valuable insights and information regarding the transaction structure and associated scripts as shown in figure 5.

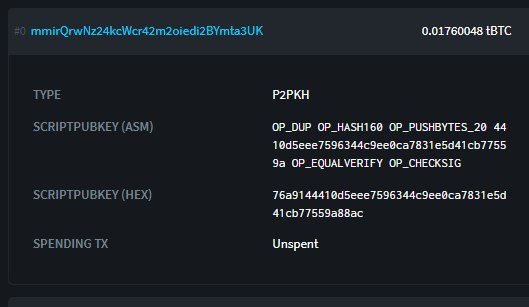


Figure

## Part 1

Utilizing the provided script, we implemented the necessary logic using script shown in figure 6 within the transaction.py file. This logic allows us to spend the unspent portion of the faucet transaction and generate a new transaction with one input and two outputs.

The first output is designated as spendable, allowing the specified Bitcoin amount to be sent to the desired address. The second output is configured as not spendable, serving the purpose of returning any remaining funds or change back to the faucet address.



Figure

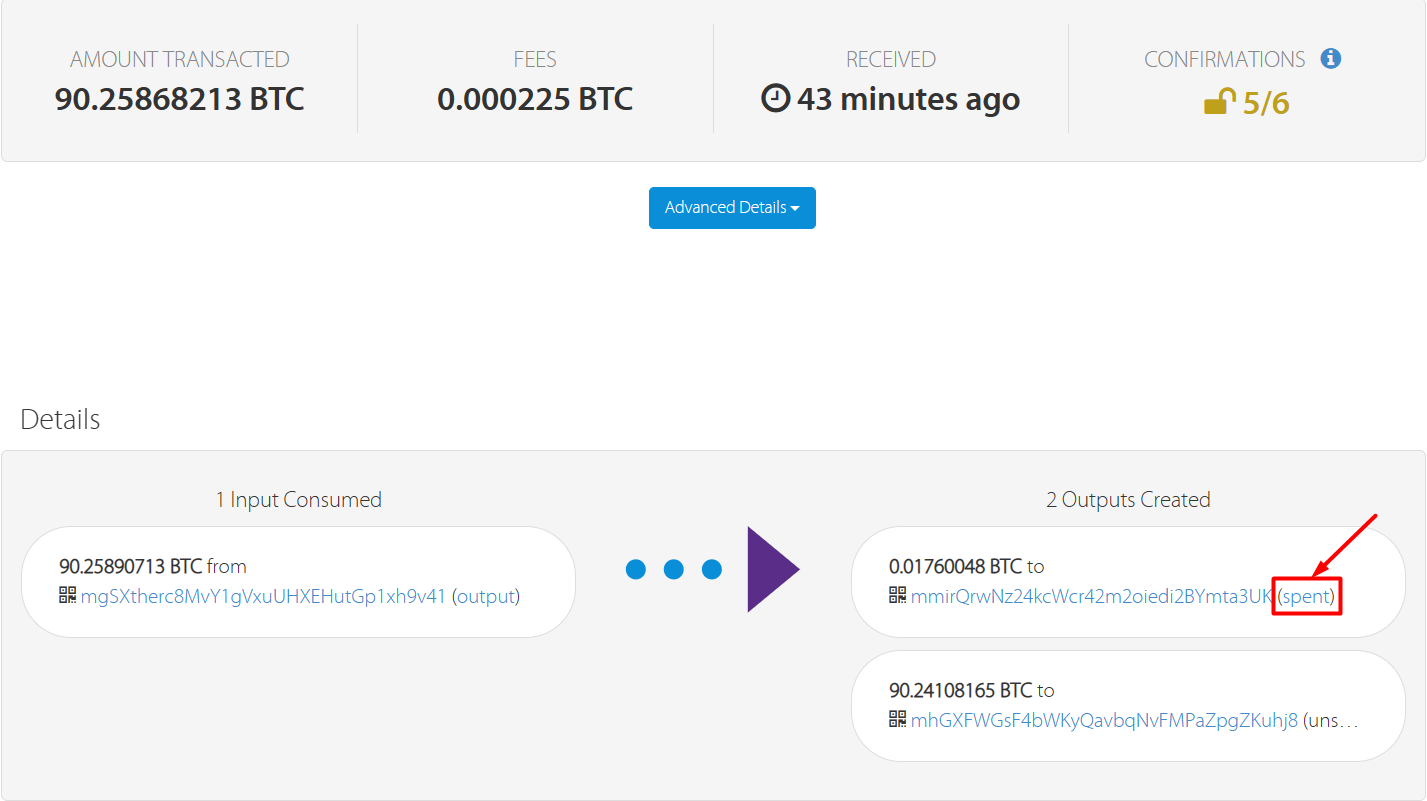
Upon executing the transaction.py code, the output is obtained, which is illustrated in Figure 7 of the report.

Figure

The transaction.py code execution yielded the following output:

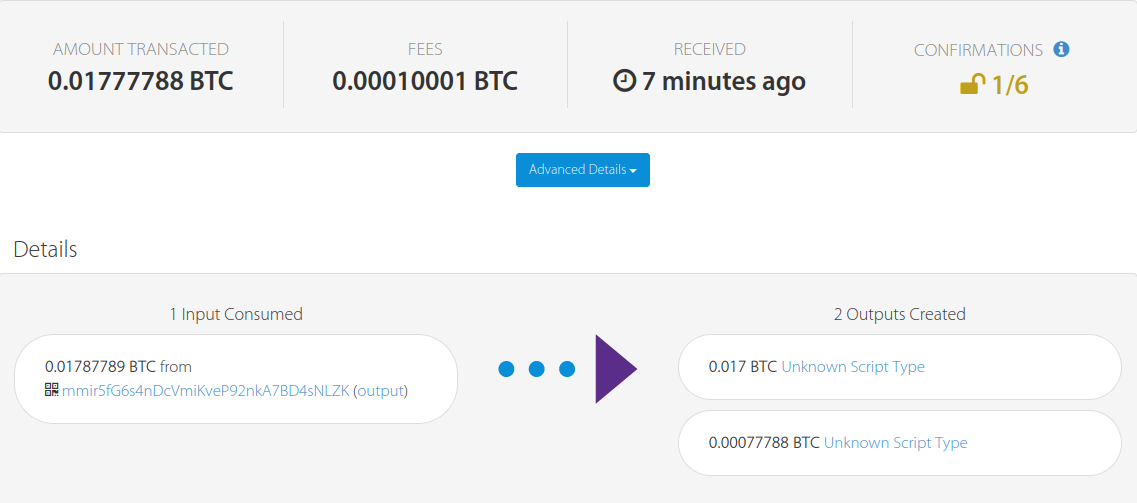
* "addresses": [ "mmir5fG6s4nDcVmiKveP92nkA7BD4sNLZK" ]
* "preference": "low"
* "total": 1500000
* "received": "2023-05-28T13:58:02.970264091Z"
* "double\_spend": false
* "script\_type": "pay-to-pubkey-hash"
* Txid = b9baa009551b44a091d4445be9bdf737afce282fa7c4620ffca5bcfbb4296974

By monitoring our transaction on the Bitcoin blockchain, it is evident that the funds have been successfully expended, as illustrated in Figure 8 on the faucet website.



Figure

The transaction information is as shown in figure 9.



Figure

In another transaction, we spend the spendable output of this transaction and return it to our original address as P2PKH output.(q2\_p1\_2.py file)

The address is : b9baa009551b44a091d4445be9bdf737afce282fa7c4620ffca5bcfbb4296974

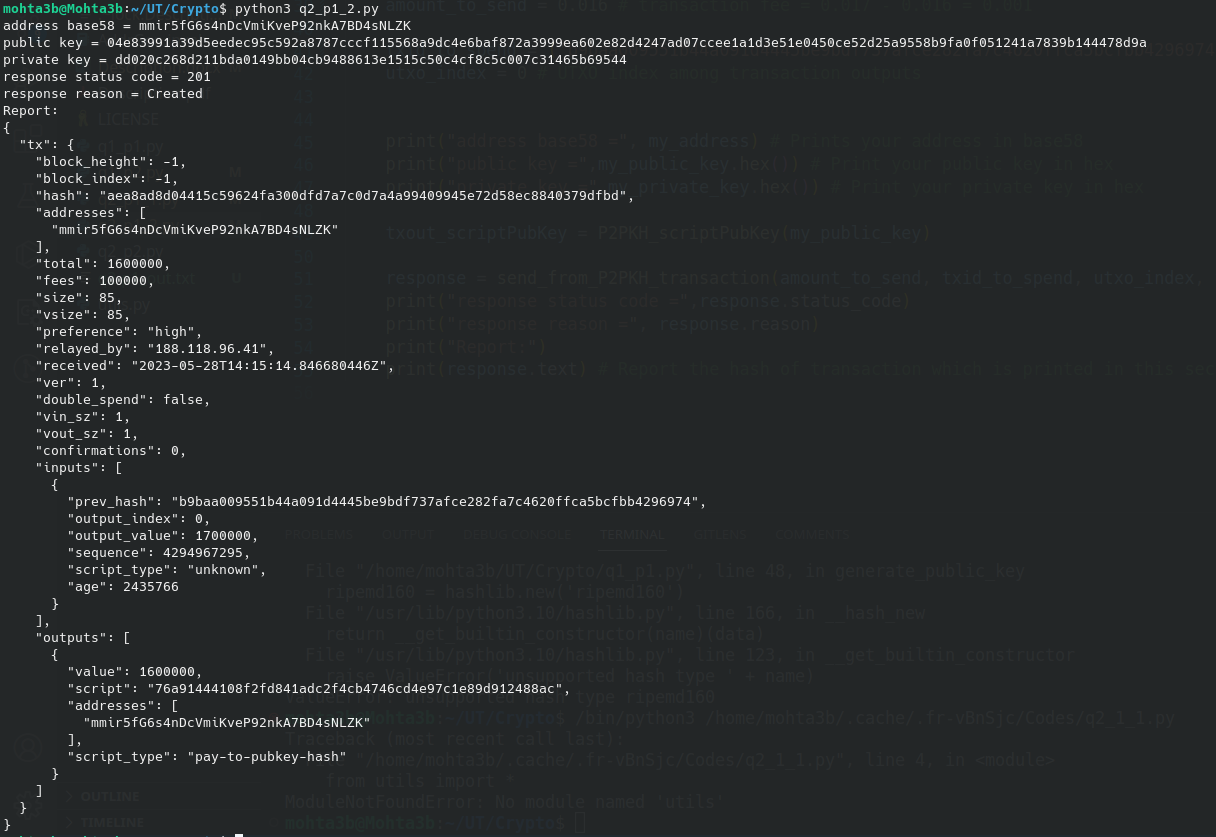


Figure : test code result

The transaction.py code execution yielded the following output:

* "addresses": [ "mmir5fG6s4nDcVmiKveP92nkA7BD4sNLZK" ]
* "preference": "high"
* "received": "2023-05-28T14:15:14.846680446Z"
* "double\_spend": false
* "script\_type": "unknown"
* Txid = aea8ad8d04415c59624fa300dfd7a7c0d7a4a99409945e72d58ec8840379dfbd

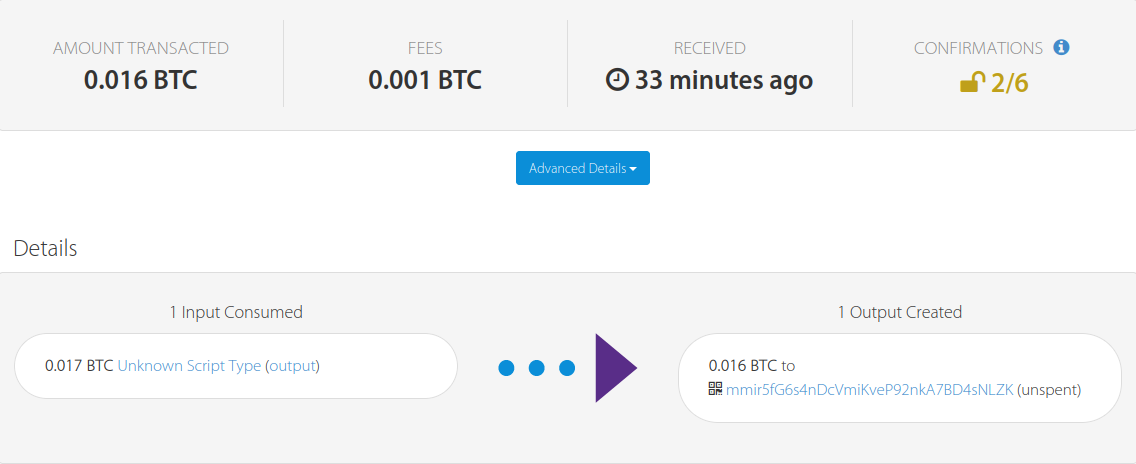
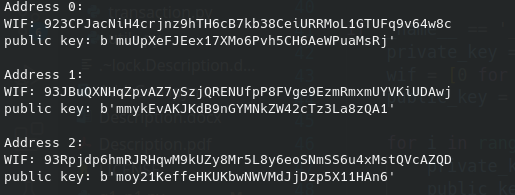


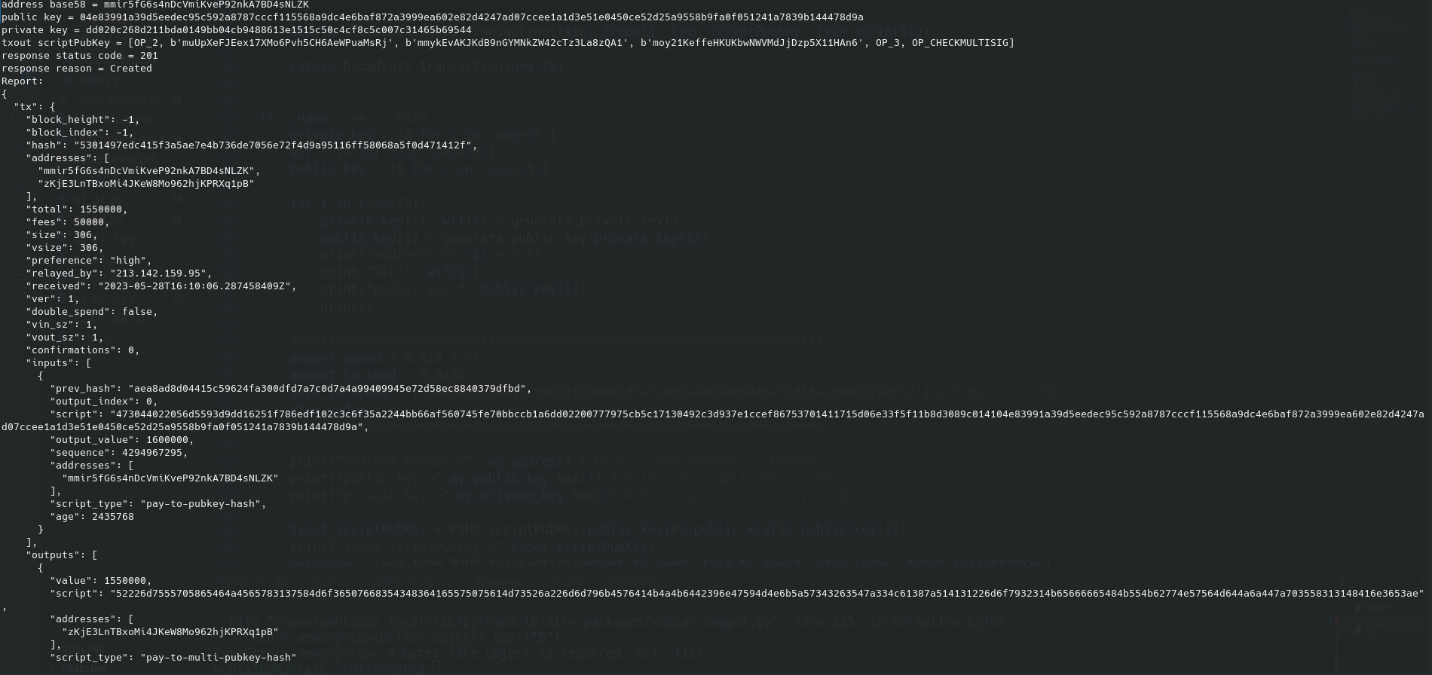
Figure : test result

## Part 2

 First we generate 3 bitcoin address on test network using the codes of the last part.

Figure

Address 1:   
WIF: 923CPJacNiH4crjnz9hTH6cB7kb38CeiURRMoL1GTUFq9v64w8c   
address: muUpXeFJEex17XMo6Pvh5CH6AeWPuaMsRj  
  
Address 2:   
WIF: 93JBuQXNHqZpvAZ7ySzjQRENUfpP8FVge9EzmRmxmUYVKiUDAwj   
address: mmykEvAKJKdB9nGYMNkZW42cTz3La8zQA1  
  
Address 3:   
WIF: 93Rpjdp6hmRJRHqwM9kUZy8Mr5L8y6eoSNmSS6u4xMstQVcAZQD   
address: moy21KeffeHKUKbwNWVMdJjDzp5X11HAn6

We create a transaction that has one input and one output, the output of which is MS2P or Multisig and can be spent by 2 people from these 3 addresses. The output is shown in figure 13.  


Figure

The transaction.py code execution yielded the following output:

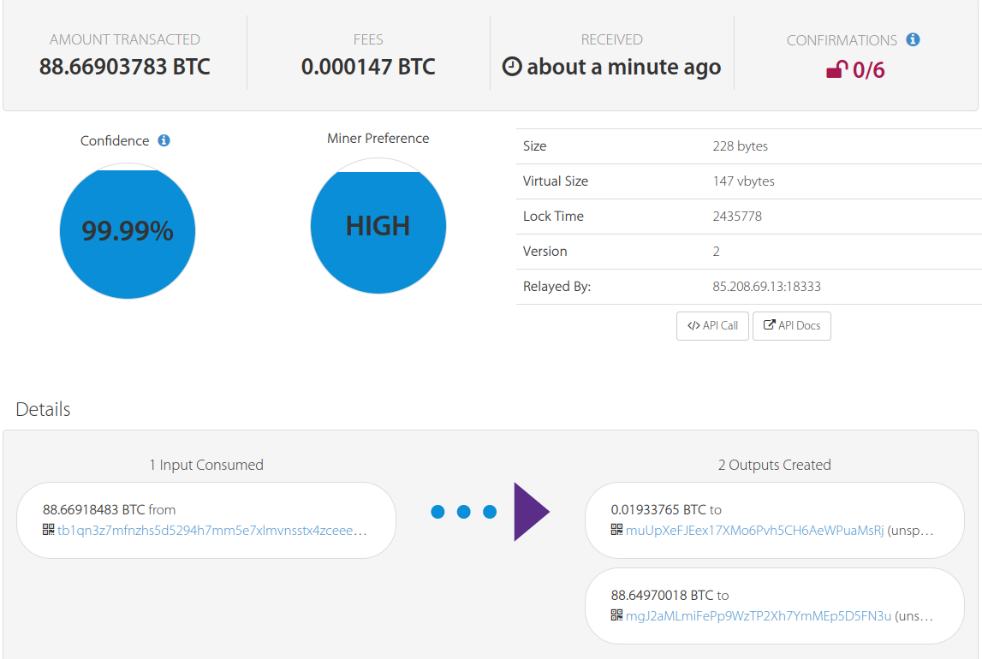
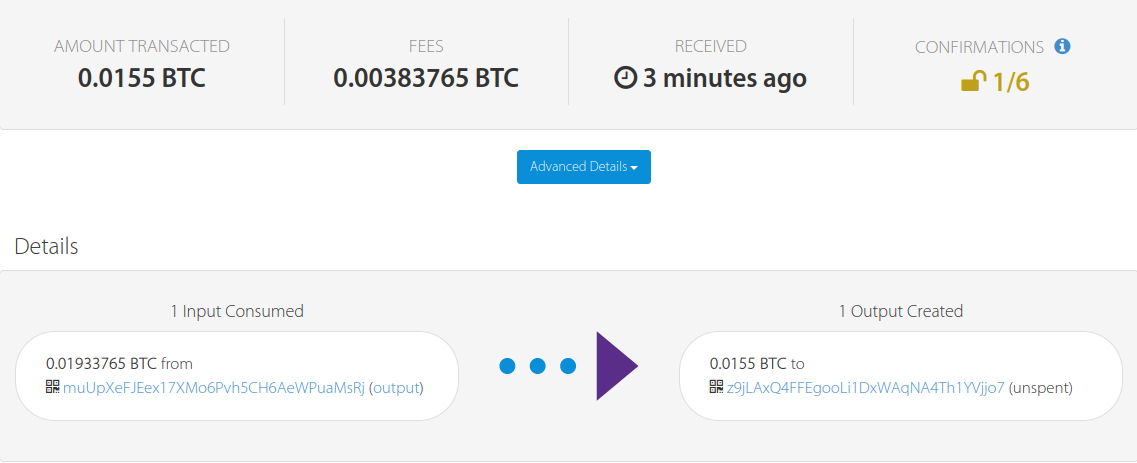
* "addresses": [ "mmir5fG6s4nDcVmiKveP92nkA7BD4sNLZK",   
   "zKjE3LnTBxoMi4JKeW8Mo962hjKPRXq1pB" ]
* "preference": "high"
* "received": "2023-05-28T16:10:06.287458409Z"
* "double\_spend": false
* "script\_type": "unknown"
* Txid = 5301497edc415f3a5ae7e4b736de7056e72f4d9a95116ff58068a5f0d471412f  
  

Figure test result

Then we send back this money to it’s original address in another transaction.



Figure

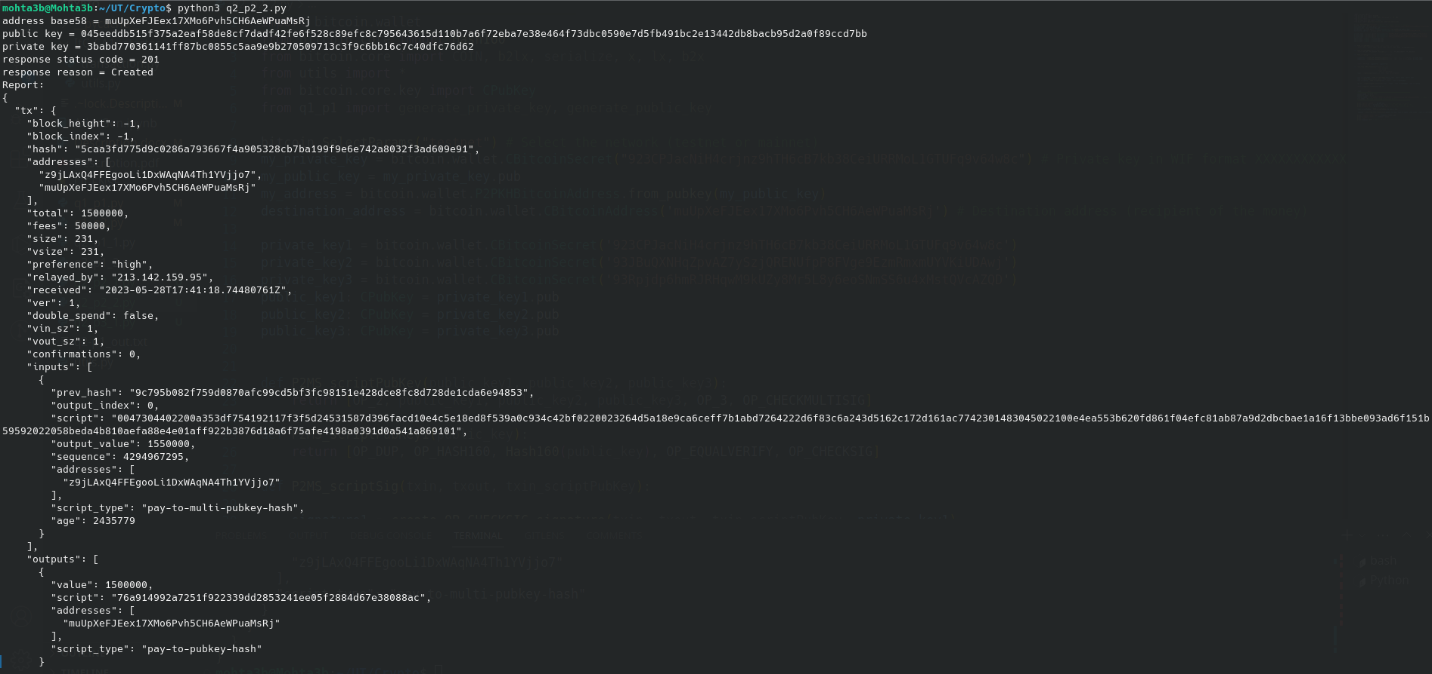


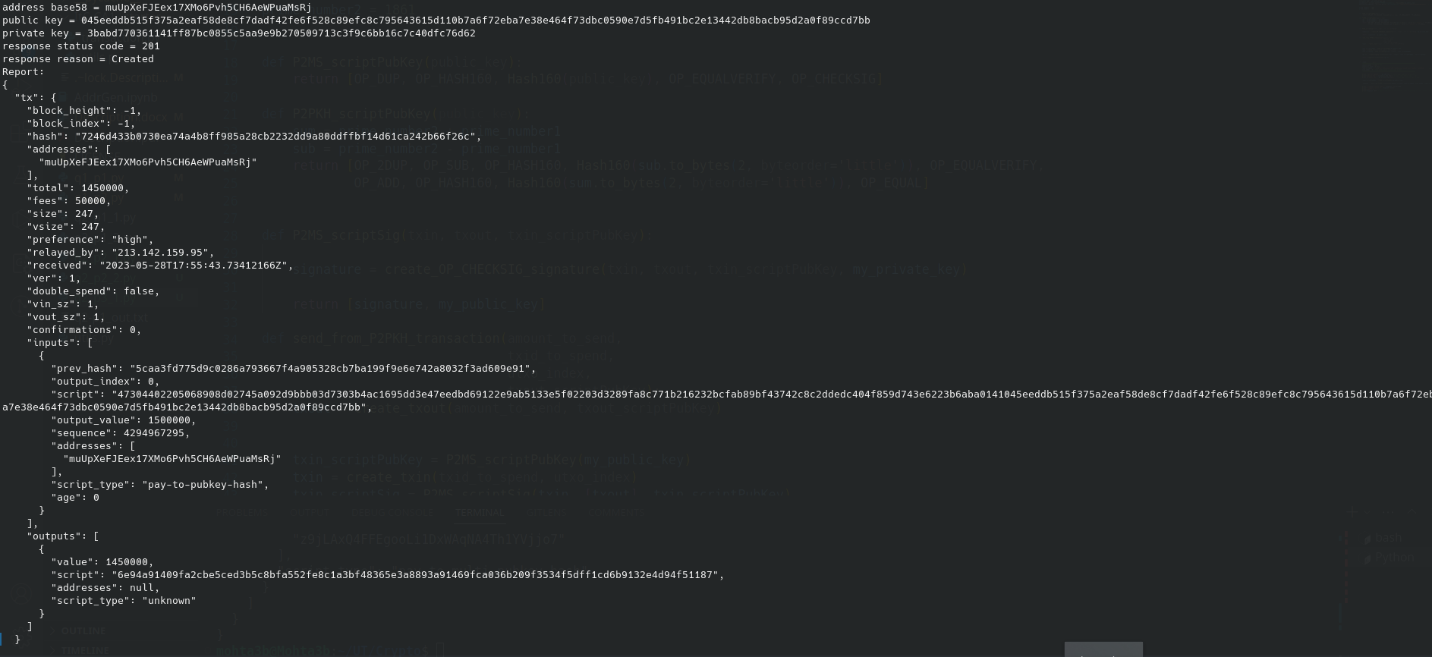
Figure test result

The transaction.py code execution yielded the following output:

* "addresses": [ "z9jLAxQ4FFEgooLi1DxWAqNA4Th1YVjjo7",   
       "muUpXeFJEex17XMo6Pvh5CH6AeWPuaMsRj" ]
* "preference": "high"
* "received": "2023-05-28T17:41:18.74480761Z"
* "double\_spend": false
* "script\_type": "pay-to-multi-pubkey-hash"
* Txid = 5caa3fd775d9c0286a793667f4a905328cb7ba199f9e6e742a8032f3ad609e91

# Part 3-1

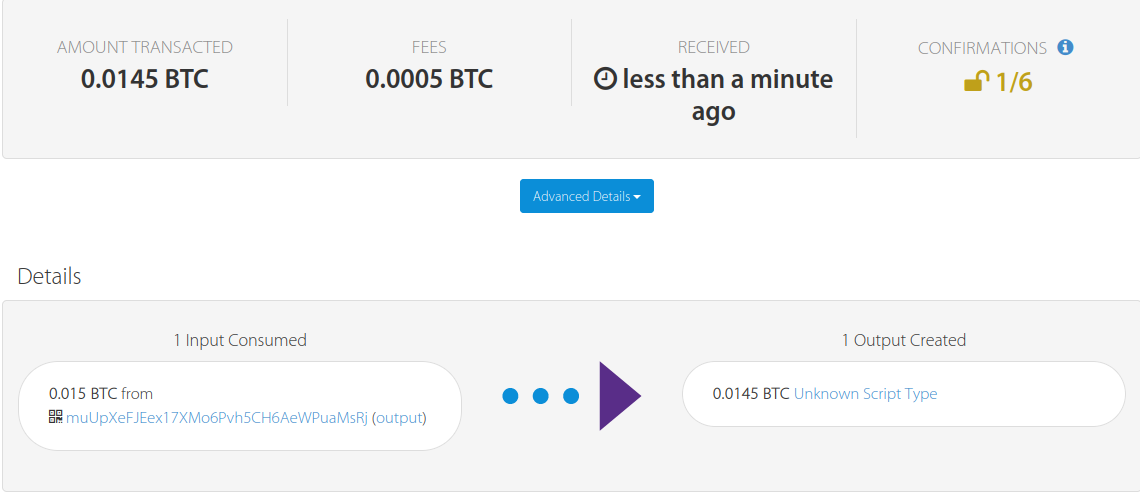
We anticipate that the first two numbers will be placed in the stack for spending. In this step, the first DUP2\_OP operation is performed to create copies of both numbers. The subtraction operation (SUB\_OP) is then applied, followed by the hashing of the resulting number. The hashed number is then concealed. The two hashed numbers are compared, and if they match, they are removed from the stack (EQUALVERIFY\_OP). Numbers we choose are 1009 and 1861. Test result is shown in figure 17.



Figure

The transaction.py code execution yielded the following output:

* "addresses": [ "muUpXeFJEex17XMo6Pvh5CH6AeWPuaMsRj“ ]
* "preference": "high"
* "received": "2023-05-28T17:55:43.73412166Z"
* "double\_spend": false
* "script\_type": "pay-to-pubkey-hash"
* Txid = 7246d433b0730ea74a4b8ff985a28cb2232dd9a80ddffbf14d61ca242b66f26c



Figure

# Part 3-2

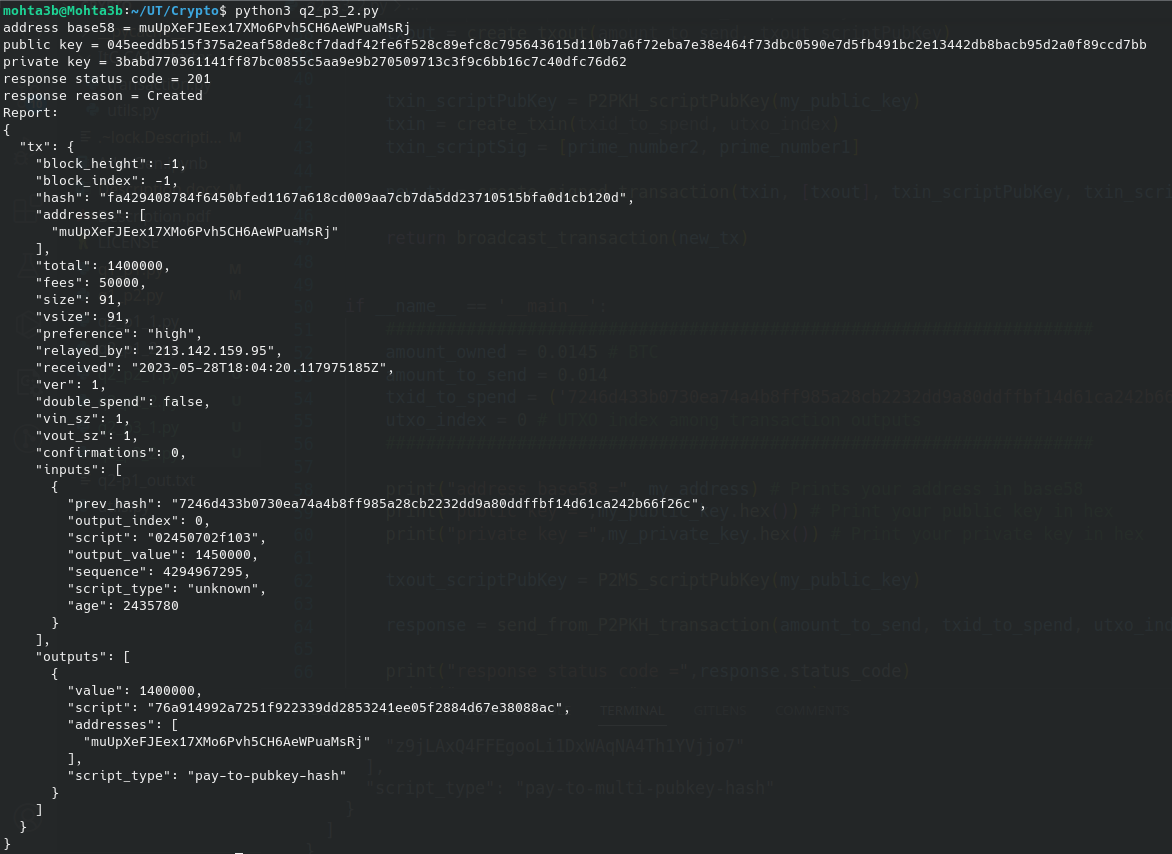


Figure test result

The transaction.py code execution yielded the following output:

* "addresses": [ "muUpXeFJEex17XMo6Pvh5CH6AeWPuaMsRj“ ]
* "preference": "high"
* "received": "2023-05-28T18:04:20.117975185Z"
* "double\_spend": false
* "script\_type": "pay-to-pubkey-hash"
* Txid = fa429408784f6450bfed1167a618cd009aa7cb7da5dd23710515bfa0d1cb120d

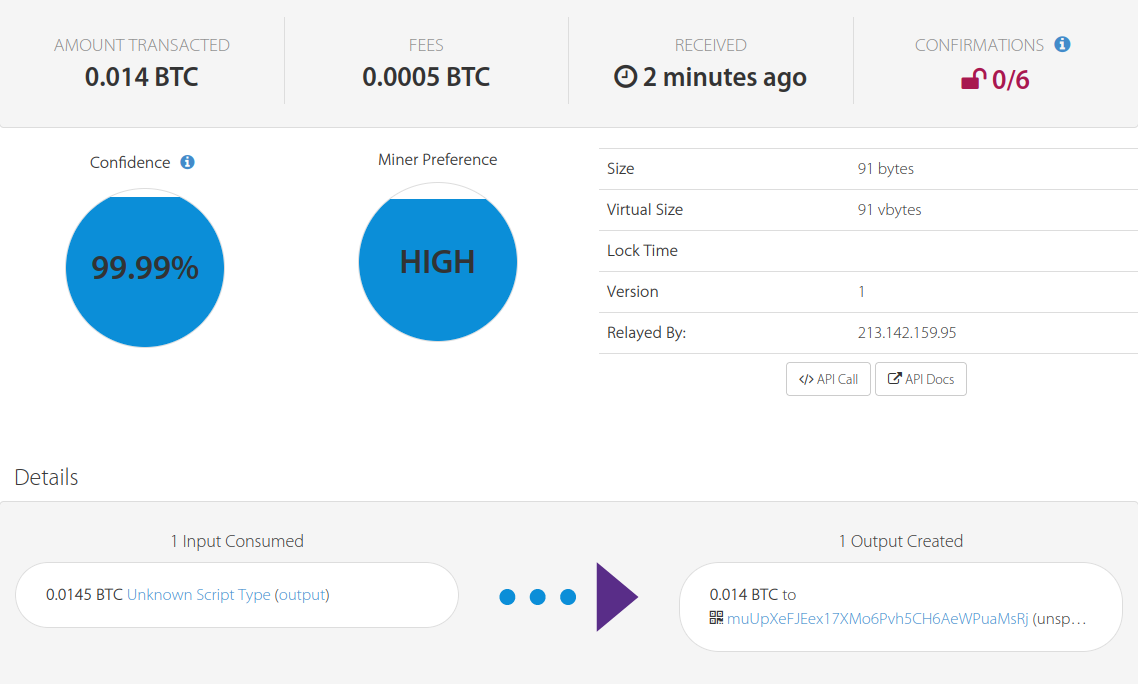
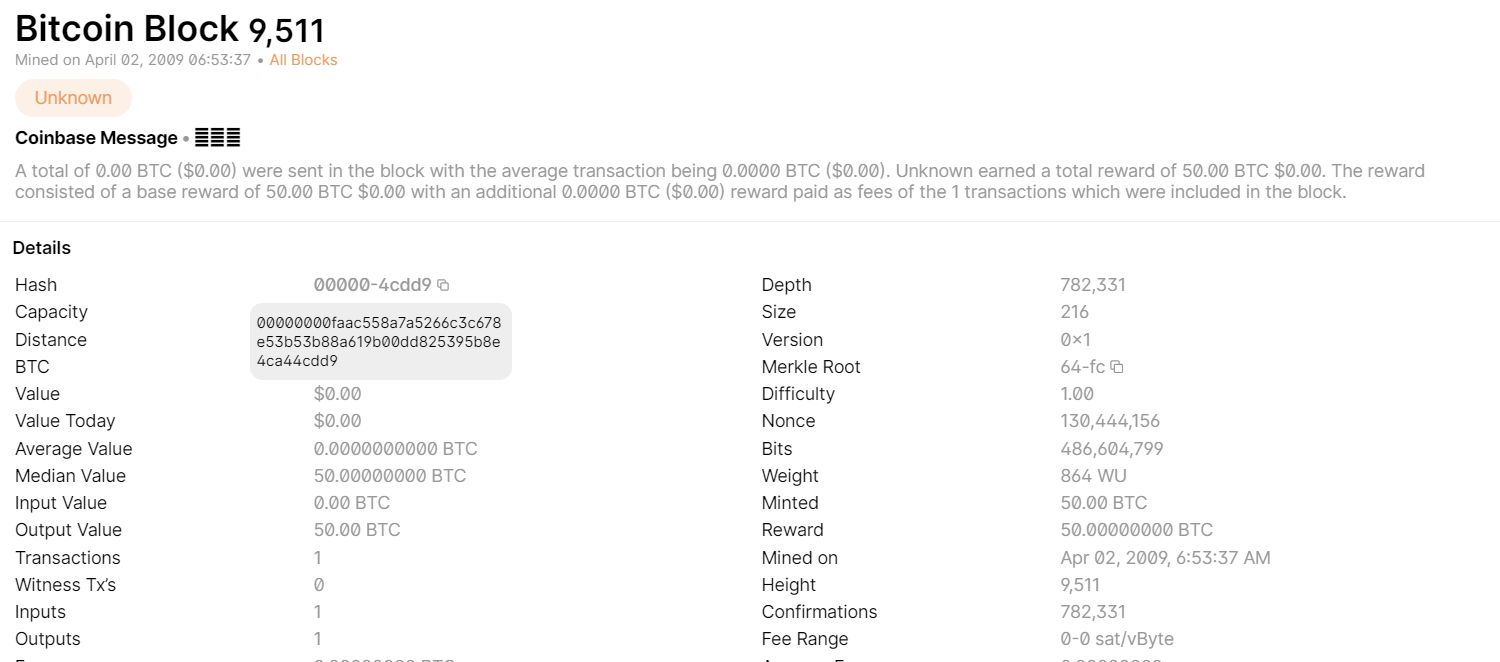


Figure test result

# Question 3

We utilize a specific [website](https://www.blockchain.com/explorer/blocks/btc/9511) to identify a desired block hash for mining a block on Bitcoin's main network.

Figure

The structure of a Bitcoin block is as follows:

* Magic Number
* Blocksize
* Blockheader
* Transaction Count
* Transactions

The blockheader section consists of six fields:

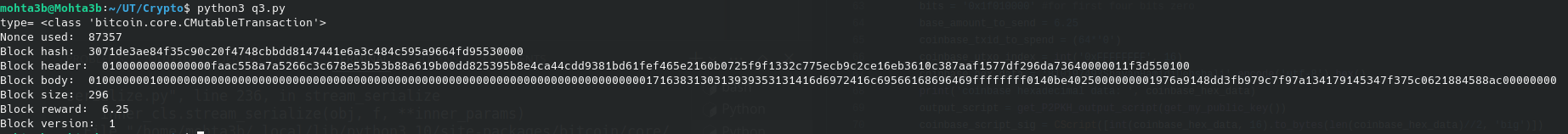
Version - Previous Block Hash - Merkle Root Hash – Timestamp – Bits - Nonce

In this case, if we consider the value of the Bits field as 0x1f010000, we can extract the exponent as "1f" and the coefficient as "010000". By combining these values, we can derive the Target value. The Target represents a specific level of difficulty for mining a block, where the leftmost 15 bits are zero, followed by a single "1", and the remaining bits are zero. Consequently, a valid hashed block must be numerically lower than this Target value, requiring it to have 16 leading zeros.

As of now, the current Bitcoin block reward is 6.25 BTC.

In our testing process, we iterate through several nonce values until we find a nonce for which the computed hash of the block is numerically smaller than the Target value. This successful outcome indicates that the block satisfies the difficulty requirements and can be added to the blockchain.

Result of test code is shown in figure 22.



Figure

Nonce used:  87357   
Block hash:  3071de3ae84f35c90c20f4748cbbdd8147441e6a3c484c595a9664fd95530000   
Block header:

0100000000000000faac558a7a5266c3c678e53b53b88a619b00dd825395b8e4ca44cdd9381bd61fef465e2160b0725f9f1332c775ecb9c2ce16eb3610c387aaf1577df296da73640000011f3d550100

Blockbody:

01000000010000000000000000000000000000000000000000000000000000000000000000000000001716383130313939353131416d6972416c69566168696469ffffffff0140be4025000000001976a9148dd3fb979c7f97a134179145347f375c0621884588ac00000000

Block size:  296   
Block reward:  6.25   
Block version:  1

1. For more info visit <https://en.bitcoin.it/wiki/Wallet_import_format> [↑](#footnote-ref-1)
2. <https://coinfaucet.eu/en/btc-testnet> [↑](#footnote-ref-2)