**Shenzhen University**

**Report of The Experiments**

**Course： Information Security and Blockchain**

**Topic： Bitcion Account and Transactions**

**Class： 22文华班**

**StudentID： 2022280297**

**Name： 陈应权**

**Date： 2024年11月16日星期六**

**Score：**

# 1. Experiment Content

1) To generate a bitcoin account: write a program that inputs a 256-bit random number *x*, outputs the public and private keys, and the bitcoin account address.

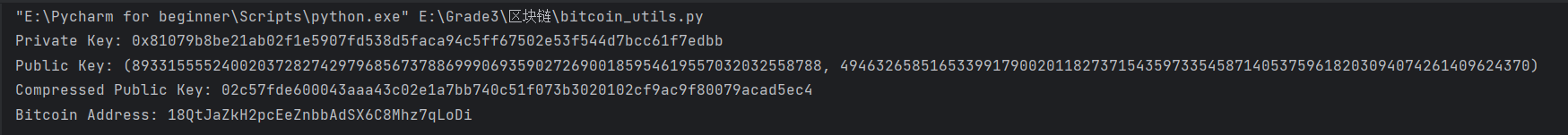
2) (Optional) Simulate making a transaction request to transfer 0.1 bitcoins from account A to account B. A signs the transaction message.

**2. The experimental code and results screenshots**

**1)To generate a bitcoin account:**

Code：

1. import hashlib
2. import os
3. *#下面代码实现了输入一个私钥，然后通过secp2561k1标准比特币椭圆曲线，生成公钥后再计算出P2PKH账号地址。*
4. *# 设定secp256k1参数*
5. p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
6. a = 0
7. b = 7
8. *# 生成点 G (secp256k1标准生成点)*
9. Gx = 0x79BE667EF9DCBBAC55A62B70F9B8C1C4C7B5B5F7B78A1B24788C6F1C9B4D5C9B
10. Gy = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199A1B8F94D6AC28A6
11. *# 函数：椭圆曲线加法*
12. def point\_add(P, Q):
13. """P和Q是椭圆曲线上的点，计算P + Q"""
14. if P == (0, 0):
15. return Q
16. if Q == (0, 0):
17. return P
18. x1, y1 = P
19. x2, y2 = Q
20. if x1 == x2 and y1 == y2:  *# P == Q*
21. *# 点加倍公式*
22. m = (3 \* x1 \*\* 2 + a) \* pow(2 \* y1, -1, p) % p
23. else:
24. *# 点加公式*
25. m = (y2 - y1) \* pow(x2 - x1, -1, p) % p
26. x3 = (m \*\* 2 - x1 - x2) % p
27. y3 = (m \* (x1 - x3) - y1) % p
28. return (x3, y3)
29. *# 函数：标量乘法*
30. def point\_multiply(k, P):
31. """计算 k \* P，k是标量，P是椭圆曲线上的点"""
32. result = (0, 0)  *# 点的零元素*
33. addend = P
34. while k:
35. if k & 1:
36. result = point\_add(result, addend)
37. addend = point\_add(addend, addend)  *# 点加倍*
38. k >>= 1  *# 右移1位*
39. return result
40. *# 计算公钥*
41. def private\_key\_to\_public\_key(private\_key):
42. """通过私钥计算公钥"""
43. G = (Gx, Gy)
44. public\_key = point\_multiply(private\_key, G)
45. return public\_key
46. *# 压缩公钥*
47. def compress\_public\_key(public\_key):
48. """将公钥压缩"""
49. x, y = public\_key
50. *# 判断y坐标的奇偶性，选择合适的压缩格式*
51. prefix = b'\x02' if y % 2 == 0 else b'\x03'
52. compressed\_key = prefix + x.to\_bytes(32, byteorder='big')
53. return compressed\_key
54. *# 哈希函数：生成比特币地址*
55. def public\_key\_to\_address(public\_key):
56. """通过公钥生成比特币地址"""
57. x, y = public\_key
58. pubkey\_bytes = x.to\_bytes(32, byteorder='big') + y.to\_bytes(32, byteorder='big')
59. *# RIPEMD160(SHA256(public\_key))*
60. sha256 = hashlib.sha256(pubkey\_bytes).digest()
61. ripemd160 = hashlib.new('ripemd160', sha256).digest()
62. *# 添加版本前缀 0x00 (用于P2PKH)*
63. version = b'\x00'
64. prefixed\_ripemd160 = version + ripemd160
65. *# 计算校验和*
66. checksum = hashlib.sha256(hashlib.sha256(prefixed\_ripemd160).digest()).digest()[:4]
67. *# 生成最终比特币地址*
68. address\_bytes = prefixed\_ripemd160 + checksum
69. address = base58\_encode(address\_bytes)
70. return address
71. *# Base58 编码*
72. def base58\_encode(data):
73. """将数据编码为Base58格式"""
74. alphabet = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'
75. n = int.from\_bytes(data, 'big')
76. result = ''
77. while n > 0:
78. n, mod = divmod(n, 58)
79. result = alphabet[mod] + result
80. for byte in data:
81. if byte == 0:
82. result = alphabet[0] + result
83. else:
84. break
85. return result
86. *# 生成256位的随机私钥，返回int类型*
87. private\_key = int.from\_bytes(os.urandom(32), byteorder='big')
88. *# 示例：生成私钥、公钥和比特币地址*
89. *# private\_key = 0x7f3c3b8d4f4baf6c8d7bdbf26f3f2f21c2a22755d37a9b508575539f42931e7  # 示例私钥（256位）*
90. *# 计算公钥*
91. public\_key = private\_key\_to\_public\_key(private\_key)
92. print(f"Private Key: {private\_key:#064x}")
93. print(f"Public Key: ({public\_key[0]}, {public\_key[1]})")
94. *# 生成压缩后的公钥*
95. compressed\_public\_key = compress\_public\_key(public\_key)
96. print(f"Compressed Public Key: {compressed\_public\_key.hex()}")
97. *# 生成比特币地址*
98. bitcoin\_address = public\_key\_to\_address(public\_key)
99. print(f"Bitcoin Address: {bitcoin\_address}")

Fig. 1 The results of generate bitcoin account

**Private Key:** 0x81079b8be21ab02f1e5907fd538d5faca94c5ff67502e53f544d7bcc61f7edbb

**Public Key:** (89331555524002037282742979685673788699906935902726900185954619557032032558788, 49463265851653399179002011827371543597335458714053759618203094074261409624370)

**Compressed Public Key:** 02c57fde600043aaa43c02e1a7bb740c51f073b3020102cf9ac9f80079acad5ec4

**Bitcoin Address:** 18QtJaZkH2pcEeZnbbAdSX6C8Mhz7qLoDi

### Principle Analysis

The primary function of the code is to generate a Bitcoin P2PKH (Pay-to-Public-Key-Hash) address from a randomly generated private key using the secp256k1 elliptic curve. Below is a step-by-step explanation of the implementation's core principles.

### 1. Elliptic Curve Parameters

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F

a = 0

b = 7

* p: The prime modulus for secp256k1, defining the finite field over which the elliptic curve operates.
* a and b: The parameters of the elliptic curve equation:

### 2. Elliptic Curve Generator Point (GGG)

The generator point GGG is a predefined point on the secp256k1 curve, given by:

Where：

|  |  |
| --- | --- |
| Gx​ | =55066263022277343669578718895168534326250603453777594175500187360389116729240 |
| Gy​ | =32670510020758816978083085130507043184471273380659243275938904335757337482424 |

### 3. Elliptic Curve Point Addition

The addition of two points P=(x1​,y1​)and Q=(x2​,y2​) on the curve depends on whether the points are identical:

#### Case 1: Point Doubling (P=Q)

When P=Q, the slope (λ) is:

### 4. Elliptic Curve Scalar Multiplication

Scalar multiplication is implemented using the double-and-add method:

1. Decompose k into its binary representation.
2. For each bit of k:
   * If the bit is 1, add the current point to the result.
   * Always double the current point.
3. Continue until all bits of k are processed.

The result is:

### 5. Deriving Public Key from Private Key

The public key Q is computed as:

Q=k⋅G

### 6. Public Key Compression (compress\_public\_key)

The compressed public key is a more compact representation of the public key, consisting of:

* The xxx-coordinate.
* A prefix to indicate the parity of the y-coordinate

### 7. Generating Bitcoin Address (public\_key\_to\_address)

The steps to generate a Bitcoin P2PKH address:

1. Serialize the Public Key:
   * Combine xxx and y-coordinates into a byte sequence.
2. Double Hashing:
   * Compute the SHA256 hash of the serialized public key.
   * Compute the RIPEMD160 hash of the SHA-256 hash to produce a 160-bit hash.
3. Add Version Prefix:
   * Prepend the RIPEMD160 hash with a 0x00 byte to indicate a P2PKH address.
4. Checksum:
   * Perform double SHA256 hashing on the prefixed hash.
   * Extract the first 4 bytes as the checksum.
5. Base58 Encoding:
   * Concatenate the prefixed hash and checksum.
   * Encode the result into Base58 format to produce the final address.

### 8. Base58 Encoding (base58\_encode)

This function implements Base58 encoding, a format used by Bitcoin to avoid visually similar characters. The steps:

1. Convert the input byte sequence into an integer.
2. Iteratively divide by 58 to obtain the Base58 representation.
3. Add leading 1s to represent leading zero bytes.

### 9. Private Key Generation

A private key is a random 256-bit integer generated using a cryptographic random number generator (os.urandom).

Code:

private\_key = int.from\_bytes(os.urandom(32), byteorder='big')

### 10. Output Results

* Private Key: A randomly generated 256-bit number.
* Public Key: A point (x,y) on the secp256k1 curve derived from the private key.
* Compressed Public Key: The compact representation of the public key.
* Bitcoin Address: A Base58-encoded string representing the hash of the public key.

### Summary

This code successfully demonstrates the process of generating a Bitcoin address from a private key:

1. **Elliptic Curve Cryptography** is used to derive the public key from the private key.
2. **Public Key Compression** optimizes storage and transmission.
3. **Double Hashing and Base58 Encoding** generate a unique and secure Bitcoin address.

**2) (Optional) Simulate making a transaction**

**Code:**

1. import hashlib
2. import os
3. import sys
4. *# Suppress output from `bitcoin\_utils.py` during import*
5. sys\_stdout\_backup = sys.stdout  *# Backup the current stdout*
6. sys.stdout = open(os.devnull, 'w')  *# Redirect stdout to null*
7. from bitcoin\_utils import (
8. private\_key\_to\_public\_key,
9. compress\_public\_key,
10. public\_key\_to\_address
11. )
12. sys.stdout.close()  *# Close the null redirection*
13. sys.stdout = sys\_stdout\_backup  *# Restore stdout*
14. from ecdsa import SigningKey, VerifyingKey, SECP256k1, BadSignatureError
15. *# Generate Bitcoin account*
16. def generate\_bitcoin\_account():
17. private\_key = os.urandom(32)
18. sk = SigningKey.from\_string(private\_key, curve=SECP256k1)  *# Create a signing key*
19. public\_key = sk.verifying\_key  *# Get the corresponding verifying key (public key)*
20. compressed\_public\_key = public\_key.to\_string("compressed")  *# Generate compressed public key*
21. bitcoin\_address = public\_key\_to\_address((sk.verifying\_key.pubkey.point.x(), sk.verifying\_key.pubkey.point.y()))
22. return private\_key, public\_key, compressed\_public\_key, bitcoin\_address
23. *# Hash transaction message*
24. def hash\_transaction(transaction\_message):
25. return hashlib.sha256(transaction\_message.encode('utf-8')).digest()
26. *# ECDSA signature generation using library*
27. def ecdsa\_sign\_library(transaction\_message, private\_key):
28. sk = SigningKey.from\_string(private\_key, curve=SECP256k1)
29. msg\_hash = hash\_transaction(transaction\_message)
30. signature = sk.sign\_digest(msg\_hash)  *# Sign the hash of the transaction message*
31. return signature
32. *# ECDSA signature verification using library*
33. def verify\_signature\_library(transaction\_message, signature, compressed\_public\_key):
34. vk = VerifyingKey.from\_string(compressed\_public\_key, curve=SECP256k1)  *# Create verifying key*
35. msg\_hash = hash\_transaction(transaction\_message)  *# Hash the transaction message*
36. try:
37. vk.verify\_digest(signature, msg\_hash)  *# Verify the signature*
38. return True
39. except BadSignatureError:
40. return False
41. *# Simulate a transaction*
42. def simulate\_transaction():
43. *# Generate accounts for A and B*
44. A\_private\_key, A\_public\_key, A\_compressed\_pub\_key, A\_bitcoin\_address = generate\_bitcoin\_account()
45. B\_private\_key, B\_public\_key, B\_compressed\_pub\_key, B\_bitcoin\_address = generate\_bitcoin\_account()
46. *# Print A and B's keys and addresses*
47. print("A Private Key:", A\_private\_key.hex())
48. print("A Public Key:", A\_compressed\_pub\_key.hex())
49. print("A Bitcoin Address:", A\_bitcoin\_address)
50. print("B Private Key:", B\_private\_key.hex())
51. print("B Public Key:", B\_compressed\_pub\_key.hex())
52. print("B Bitcoin Address:", B\_bitcoin\_address)
53. *# Create a transaction message*
54. transaction\_message = f"From: {A\_compressed\_pub\_key.hex()}\nTo: {B\_compressed\_pub\_key.hex()}\nAmount: 0.1 BTC"
55. print("\nTransaction Message:")
56. print(transaction\_message)
57. *# A signs the transaction*
58. signature = ecdsa\_sign\_library(transaction\_message, A\_private\_key)
59. print(f"\nSignature: {signature.hex()}")
60. *# Verify the signature using the library*
61. signature\_valid = verify\_signature\_library(transaction\_message, signature, A\_compressed\_pub\_key)
62. print(f"\nSignature Valid: {signature\_valid}")
63. if \_\_name\_\_ == "\_\_main\_\_":
64. simulate\_transaction()

**Results：**

