**区块链与数字货币——作业1**

**刘轩铭 3180106071 软件工程**

## sha256在比特币中发挥的作用

sha256等哈希算法在区块链加密中起到了至关重要的作用。比如，对交易、区块数据进行哈希运算并将结果作为编号防止交易和区块数据被篡改；通过Merkle树根哈希值归纳区块的交易；通过哈希值将区块链链结成有序的、防篡改的链条；利用哈希算法构造工作量证明共识算法等。

之所以能有这样的作用，主要是因为sha256等哈希算法具有一下的这些特点，我将结合实验对这些特点进行说明（基于自行给出的sha256算法）：

* **正向容易**



Sha256正向计算非常容易。上面的例子中，不论输出是否为很大的数据，我们的都能在短时间内计算出结果。因为sha256中的运算过程大多是位运算，所以这样的计算是很简单很快速的。这一特性导致它能够被大量用在各种数据的加密过程中，即使被大量应用，它的计算单位时间也很短，不会造成很明显的耽搁。

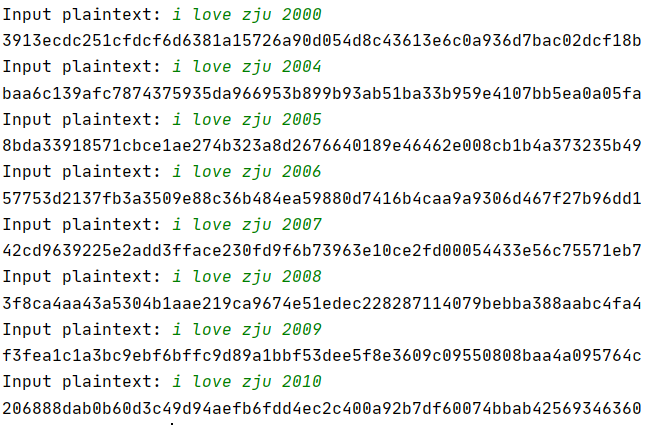
* **难以逆向推导：**



以上方实验结果为例，输入为123，而输出为如此长的一串哈希字符串，根本无法逆向推导。这使得sha256可以被用在比特币交易等场景中，用户给出自己密钥和地址的哈希值，对方仅能够对结果进行验证，而不能知道原来的密钥和地址明文。

* **雪崩效应：**

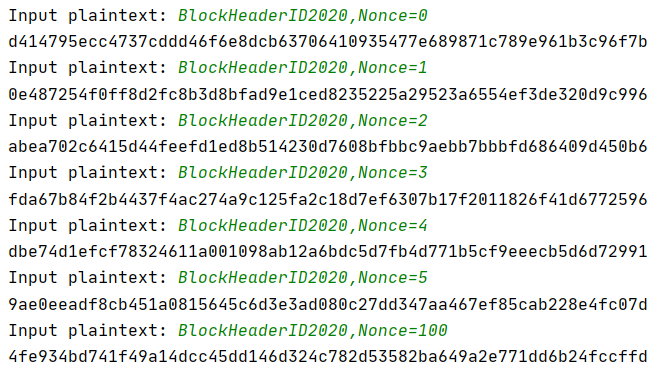
在sha256中，只要输入的数据发生微小改变，结果就会有很大的不同：



在上面的实验中可以看出，虽然我们的字符串只有很小的改变，但是结果却发生了很大的变化。这一特性可以被用在Merkle树中。如果在新的区块中，有用户篡改了某笔交易的数据，哪怕是轻微的改变，其他节点通过观察Merkle树根的巨大变化，就能发现该区块是不能被信任的。

* **输出空间远远小于输入空间：**

sha256的输出空间是256个Bits的字符串空间，这远远小于它可能的输入空间。



这里模拟了工作量证明的做法：节点确定区块头的Merkle树根等其他信息，通过改变Nonce的值来进行Hash运算，让得到的结果小于一定难度系数下的特定值。由于输入空间很大，这里的Nonce必须不断改变才能找到想要的值。这样的工作是很繁琐的，也就起到了工作量证明的效果。这就是为什么sha256可以被作为哈希算法用在比特币的工作量证明中。

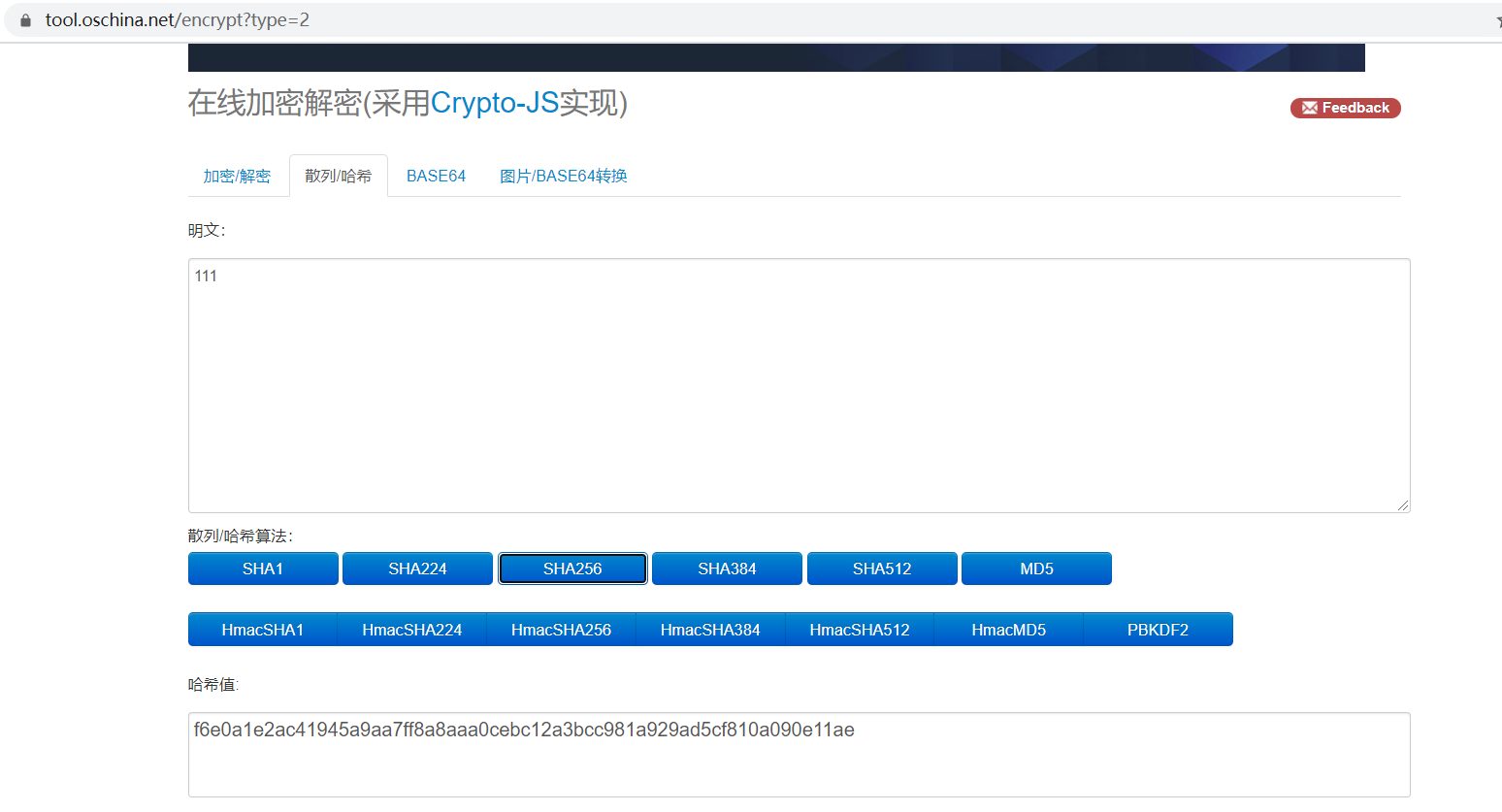
## 二、sha256源码实现

我使用Python完成了sha256算法的实现，源码请见main.py文件。下面是一些验证结果：

**我的结果：**



**其他工具结果：**



**我的结果：**



**其他工具结果：**



可以看出，我们实现的算法与网络上专业的sha256加密工具相比，也具有很高的准确性。

**附录：**

1. # const values
2. # 自然数中前64个质数(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...)的立方根的小数部分取前32bits
3. \_K = (0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
4. 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
5. 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
6. 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
7. 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
8. 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
9. 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
10. 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
11. 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
12. 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
13. 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
14. 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
15. 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
16. 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
17. 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
18. 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2)
20. # 自然数中前8个质数(2, 3, 5, 7, 11, 13, 17, 19)的平方根的小数部分取前32bits
21. \_H = (0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
22. 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19)
24. **class** Sha256:
26. **def** \_\_init\_\_(self):
27. # initialize
28. self.H = \_H
29. self.K = \_K
31. **def** hash(self, m):
32. **if** m:
33. **return** self.update(m)
34. **else**:
35. **return** ""
37. **def** update(self, m):
38. m = self.pad(m) # pre process: make m's length = 0mod512
39. digest = list(self.H)
41. # Iteration times
42. update\_time = int(len(m) \* 8 / 512)
43. **for** i **in** range(update\_time):
44. # generate 64 words.
45. S = m[i: i + 64]
46. w = [int.from\_bytes(S[e: e + 4], "big") **for** e **in** range(0, 64, 4)] + ([0] \* 48)
47. **for** j **in** range(16, 64):
48. s0 = self.rotr(w[j - 15], 7) ^ self.rotr(w[j - 15], 18) ^ (w[j - 15] >> 3)
49. s1 = self.rotr(w[j - 2], 17) ^ self.rotr(w[j - 2], 19) ^ (w[j - 2] >> 10)
50. w[j] = (w[j - 16] + s0 + w[j - 7] + s1) & 0xFFFFFFFF
52. # initial digest
53. a, b, c, d, e, f, g, h = digest
55. # 64 times encryption
56. **for** j **in** range(64):
57. s0 = self.rotr(a, 2) ^ self.rotr(a, 13) ^ self.rotr(a, 22)
58. maj = (a & b) ^ (a & c) ^ (b & c)
59. t2 = s0 + maj
60. s1 = self.rotr(e, 6) ^ self.rotr(e, 11) ^ self.rotr(e, 25)
61. ch = (e & f) ^ ((~e) & g)
62. t1 = h + s1 + ch + self.K[j] + w[j]
64. h = g
65. g = f
66. f = e
67. e = (d + t1) & 0xFFFFFFFF
68. d = c
69. c = b
70. b = a
71. a = (t1 + t2) & 0xFFFFFFFF
73. # return he string
74. **return** "".join(format(t, "02x") for t in b"".join(
75. z.to\_bytes(4, "big") **for** z **in**
76. [(x + y) & (0xFFFFFFFF) **for** x, y **in** zip(digest, (a, b, c, d, e, f, g, h))]))
78. **def** pad(self, m):
79. **return** bytes(m, "ascii") + b"\x80" + (b"\x00" \* ((55 **if** (len(m) % 64) < 56 **else** 119) - (len(m) % 64))) + (
80. (len(m) << 3).to\_bytes(8, "big"))
82. **def** rotr(self, x, y):
83. **return** ((x >> y) | (x << (32 - y))) & 0xFFFFFFFF
85. **if** \_\_name\_\_ == '\_\_main\_\_':
86. encoder = Sha256()
87. **while**(True):
88. m = input("Input plaintext: ")
89. **if** m == "":
90. **break**
91. **print**(encoder.hash(m))