**HW 2**

Question 1:

S(a)⊕S(b) ≠ S(a⊕b).

x1 = 000000, x2 = 000001 x1 = 111111, x2 = 100000

let x1=a

row = 3, column = 15

s(a) = 8

let x2=b

row = 2, column = 0

s(b) =4

convert to binary

001000⊕000100 = 001100 =12

Let x1=a, x2=b

111111⊕100000 = 0111111

Row =1 , column =15

S(a⊕b) = 08

Then 12 ≠08

let x1=a

row = 0, column = 0

s(a) = 14

let x2=b

row = 1, column = 0

s(b) =0

convert to binary

001110⊕000000 = 001110 = 14

Let x1=a, x2=b

000000⊕000001 = 000001

Row =2 , column =0

S(a⊕b) = 0

Then 0 ≠14

x1 = 101010, x2 = 010101

let x1=a

row = 2, column = 5

s(a) = 6

let x2=b

row = 1, column = 10

s(b) =12

convert to binary

000110⊕001100 = 001010 =10

Let x1=a, x2=b

101010⊕010101 = 111111

Row =3 , column =15

S(a⊕b) = 13

Then 13 ≠10

Question 2:

Assuming that no two keys can produce the same ciphertext for the same plaintext (possibly not a realistic assumption) then the worst case is the one where you have to check all 256256 possible keys. I.e. the correct key is the very last possible key that you check.

In the average case you will find the key exactly halfway through your search, meaning 256÷2=255256÷2=255 key checks.

Question 3:

#!/usr/bin/env python

|  |
| --- |
|  |
| Sbox = ( |
|  | 0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76, |
|  | 0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0, |
|  | 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15, |
|  | 0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75, |
|  | 0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84, |
|  | 0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF, |
|  | 0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8, |
|  | 0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2, |
|  | 0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73, |
|  | 0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB, |
|  | 0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79, |
|  | 0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08, |
|  | 0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A, |
|  | 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E, |
|  | 0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF, |
|  | 0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16, |
|  | ) |
|  |  |

|  |
| --- |
| xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1) |
|  |  | |
|  |  | |
|  | Rcon = ( | |
|  | 0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, | |
|  | 0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A, | |
|  | 0x2F, 0x5E, 0xBC, 0x63, 0xC6, 0x97, 0x35, 0x6A, | |
|  | 0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5, 0x91, 0x39, | |
|  | ) | |
|  |  | |
|  |  | |
|  | def text2matrix(text): | |
|  | matrix = [] | |
|  | for i in range(16): | |
|  | byte = (text >> (8 \* (15 - i))) & 0xFF | |
|  | if i % 4 == 0: | |
|  | matrix.append([byte]) | |
|  | else: | |
|  | matrix[i / 4].append(byte) | |
|  | return matrix | |
|  |  | |
|  |  | |
|  | def matrix2text(matrix): | |
|  | text = 0 | |
|  | for i in range(4): | |
|  | for j in range(4): | |
|  | text |= (matrix[i][j] << (120 - 8 \* (4 \* i + j))) | |
|  | return text | |
|  |  | |
|  |  | |
|  | class AES: | |
|  | def \_\_init\_\_(self, master\_key): | |
|  | self.change\_key(master\_key) | |
|  |  | |
|  | def change\_key(self, master\_key): | |
|  | self.round\_keys = text2matrix(master\_key) | |
|  | # print self.round\_keys | |
|  |  | |
|  | for i in range(4, 4 \* 11): | |
|  | self.round\_keys.append([]) | |
|  | if i % 4 == 0: | |
|  | byte = self.round\_keys[i - 4][0] \ | |
|  | ^ Sbox[self.round\_keys[i - 1][1]] \ | |
|  | ^ Rcon[i / 4] | |
|  | self.round\_keys[i].append(byte) | |
|  |  | |
|  | for j in range(1, 4): | |
|  | byte = self.round\_keys[i - 4][j] \ | |
|  | ^ Sbox[self.round\_keys[i - 1][(j + 1) % 4]] | |
|  | self.round\_keys[i].append(byte) | |
|  | else: | |
|  | for j in range(4): | |
|  | byte = self.round\_keys[i - 4][j] \ | |
|  | ^ self.round\_keys[i - 1][j] | |
|  | self.round\_keys[i].append(byte) | |
|  |  | |
|  | # print self.round\_keys | |
| def encrypt(self, plaintext): | |
|  | | self.plain\_state = text2matrix(plaintext) | |
|  | |  | |
|  | | self.\_\_add\_round\_key(self.plain\_state, self.round\_keys[:4]) | |
|  | |  | |
|  | | for i in range(1, 10): | |
|  | | self.\_\_round\_encrypt(self.plain\_state, self.round\_keys[4 \* i : 4 \* (i + 1)]) | |
|  | |  | |
|  | | self.\_\_sub\_bytes(self.plain\_state) | |
|  | | self.\_\_shift\_rows(self.plain\_state) | |
|  | | self.\_\_add\_round\_key(self.plain\_state, self.round\_keys[40:]) | |
|  | |  | |
|  | | return matrix2text(self.plain\_state) | |

|  |  |
| --- | --- |
|  | |
| def \_\_add\_round\_key(self, s, k): | |
|  | | for i in range(4): | |
|  | | for j in range(4): | |
|  | | s[i][j] ^= k[i][j] | |
|  | |  | |
|  | |  | |
|  | | def \_\_round\_encrypt(self, state\_matrix, key\_matrix): | |
|  | | self.\_\_sub\_bytes(state\_matrix) | |
|  | | self.\_\_shift\_rows(state\_matrix) | |
|  | | self.\_\_mix\_columns(state\_matrix) | |
|  | | self.\_\_add\_round\_key(state\_matrix, key\_matrix) | |
|  | |  | |
| def \_\_sub\_bytes(self, s): |
|  | for i in range(4): | |
|  | for j in range(4): | |
|  | s[i][j] = Sbox[s[i][j]] | |

|  |
| --- |
| def \_\_shift\_rows(self, s): |
|  | s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1] |
|  | s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2] |
|  | s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3] |

|  |
| --- |
|  |
| def \_\_mix\_single\_column(self, a): |
|  | # please see Sec 4.1.2 in The Design of Rijndael |
|  | t = a[0] ^ a[1] ^ a[2] ^ a[3] |
|  | u = a[0] |
|  | a[0] ^= t ^ xtime(a[0] ^ a[1]) |
|  | a[1] ^= t ^ xtime(a[1] ^ a[2]) |
|  | a[2] ^= t ^ xtime(a[2] ^ a[3]) |
|  | a[3] ^= t ^ xtime(a[3] ^ u) |
|  |  |
|  |  |
|  | def \_\_mix\_columns(self, s): |
|  | for i in range(4): |
|  | self.\_\_mix\_single\_column(s[i]) |
|  |  |

|  |
| --- |
| import unittest |
|  | from aes import AES |
|  |  |
|  | class AES\_TEST(unittest.TestCase): |
|  | def setUp(self): |
|  | master\_key = 0x2b7e151628aed2a6abf7158809cf4f3c |
|  | self.AES = AES(master\_key) |
|  |  |
|  | def test\_encryption(self): |
|  | plaintext = 0x3243f6a8885a308d313198a2e0370734 |
|  | encrypted = self.AES.encrypt(plaintext) |
|  |  |
|  | self.assertEqual(encrypted, 0x3925841d02dc09fbdc118597196a0b32) |

|  |
| --- |
| if \_\_name\_\_ == '\_\_main\_\_': |
|  | unittest.main() |