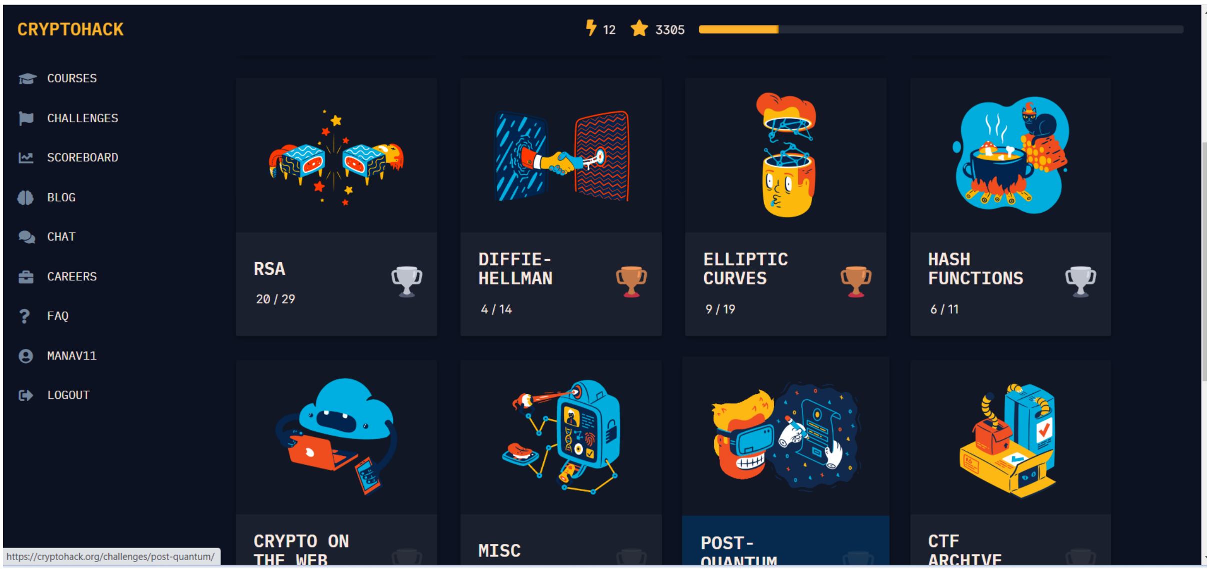
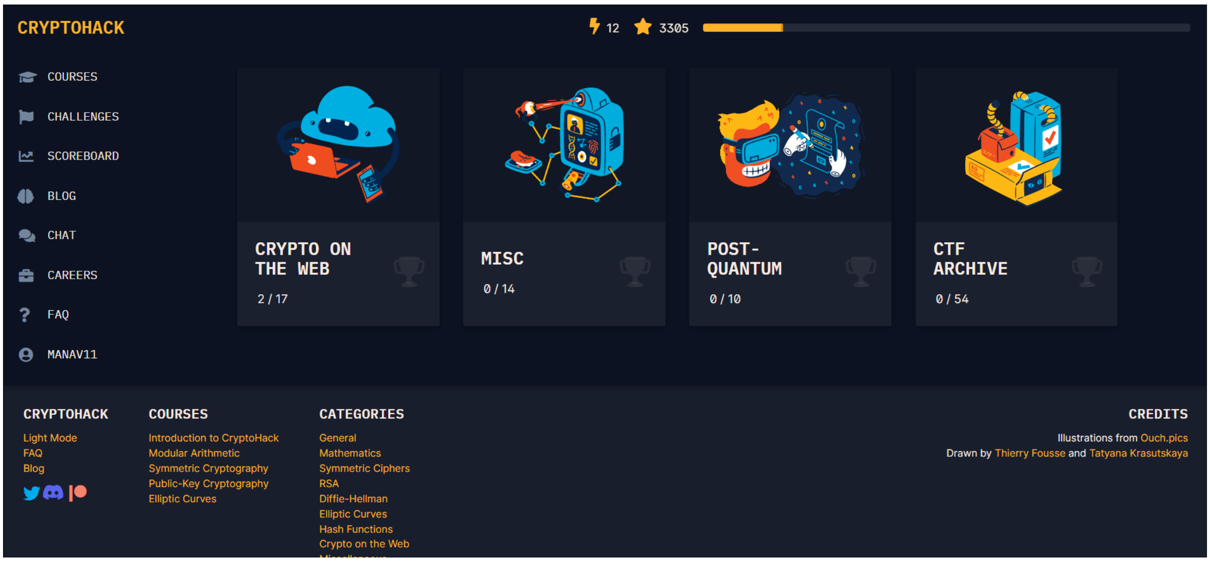
Crypto Hack Submission(Manav Makkar B00168193)





Codes and Implementations:

Bringing it altogether:

```
N_ROUNDS = 10

key        = b'\xc3,\\\xa6\xb5\x80^\x0c\xdb\x8d\xa5z*\xb6\xfe\\'
ciphertext = b'\xd1O\x14j\xa4+O\xb6\xa1\xc4\x08B)\x8f\x12\xdd'

s_box = (
```

```
    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,
)

inv_s_box = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3,
0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE,
0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA,
0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B,
0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65,
0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7,
0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8,
0xB3, 0x45, 0x06,
```

```
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13,
0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4,
0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75,
0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18,
0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78,
0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80,
0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9,
0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83,
0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21,
0x0C, 0x7D,
)

def bytes2matrix(text):
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    out = []
    for r in matrix:
        for c in r:
            out.append(c.to_bytes(2,byteorder='little').decode())
    return ''.join(out)

def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][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[1][3], s[2][3], s[3][3], s[0][3]

def inv_sub_bytes(s, sbox=inv_s_box):
    for i in range(len(s)):
        for j in range(len(s[i])):
            s[i][j] = (sbox[s[i][j]])



def add_round_key(s, k):
    for i in range(len(s)):
        for j in range(len(s[i])):
            s[i][j] = (s[i][j] ^ k[i][j])

xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)

def mix_single_column(a):
    # 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(s):
    for i in range(4):
        mix_single_column(s[i])
def inv_mix_columns(s):
    # see Sec 4.1.3 in The Design of Rijndael
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)

def expand_key(master_key):
    """
    Expands and returns a list of key matrices for the given master_key.
    """

    # Round constants https://en.wikipedia.org/wiki/AES_key_schedule#Round_constants
    r_con = (
        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,
    )

    # Initialize round keys with raw key material.
    key_columns = bytes2matrix(master_key)
    iteration_size = len(master_key) // 4

    # Each iteration has exactly as many columns as the key material.
    i = 1
    while len(key_columns) < (N_ROUNDS + 1) * 4:
        # Copy previous word.
        word = list(key_columns[-1])

        # Perform schedule_core once every "row".
        if len(key_columns) % iteration_size == 0:
            # Circular shift.
            word.append(word.pop(0))
```

```
            # Map to S-BOX.
            word = [s_box[b] for b in word]
            # XOR with first byte of R-CON, since the others bytes of R-CON are 0.
            word[0] ^= r_con[i]
            i += 1
        elif len(master_key) == 32 and len(key_columns) % iteration_size == 4:
            # Run word through S-box in the fourth iteration when using a
            # 256-bit key.
            word = [s_box[b] for b in word]

        # XOR with equivalent word from previous iteration.
        word = bytes(i^j for i, j in zip(word, key_columns[-iteration_size]))
        key_columns.append(word)

    # Group key words in 4x4 byte matrices.
    return [key_columns[4*i : 4*(i+1)] for i in range(len(key_columns) // 4)]

def decrypt(key, ciphertext):
    round_keys = expand_key(key) # Remember to start from the last round key and work backwards through them when decrypting

    # Convert ciphertext to state matrix
    state = bytes2matrix(ciphertext)
    # Initial add round key step
    add_round_key(state,round_keys[-1])


    for i in range(N_ROUNDS - 1, 0, -1):
        inv_shift_rows(state)
        inv_sub_bytes(state, inv_s_box)
        add_round_key(state,round_keys[i])
        inv_mix_columns(state)

    # Run final round (skips the InvMixColumns step)
    inv_shift_rows(state)
    inv_sub_bytes(state, inv_s_box)
    add_round_key(state,round_keys[0])

    # Convert state matrix to plaintext
    plaintext = matrix2bytes(state)

    return plaintext

print(decrypt(key, ciphertext))
```

flag = crypto{MYAES128}

Mode of operation starter:

```
import requests
```

```python
# request encrypted flag
r = requests.get('http://aes.cryptohack.org/block_cipher_starter/encrypt_flag/')
res = r.json()['ciphertext']
# print(res)

# request plaintext/decrypting flag
endpointdec = 'http://aes.cryptohack.org/block_cipher_starter/decrypt/' + res
dec = requests.get(endpointdec)
res1 = dec.json()['plaintext']
# print(res1)

by = bytes.fromhex(res1)
finalres = by.decode()
print(finalres)
```

Crypto{bl0ck_c1ph3r5_4r3_f457_!}

Diffusion through permutation:

```python
def shift_rows(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 inv_shift_rows(s):

s[1][1], s[2][1], s[3][1], s[0][1] = s[0][1], s[1][1], s[2][1], s[3][1]

s[2][2], s[3][2], s[0][2], s[1][2] = s[0][2], s[1][2], s[2][2], s[3][2]

s[3][3], s[0][3], s[1][3], s[2][3] = s[0][3], s[1][3], s[2][3], s[3][3]

# learned from http://cs.ucsb.edu/~koc/cs178/projects/JT/aes.c
```

```
xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)

def mix_single_column(a):

# 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(s):

for i in range(4):

mix_single_column(s[i])

def inv_mix_columns(s):

# see Sec 4.1.3 in The Design of Rijndael

for i in range(4):
```

```
u = xtime(xtime(s[i][0] ^ s[i][2]))
v = xtime(xtime(s[i][1] ^ s[i][3]))
s[i][0] ^= u
s[i][1] ^= v
s[i][2] ^= u
s[i][3] ^= v
mix_columns(s)
state = [
[108, 106, 71, 86],
[96, 62, 38, 72],
[42, 184, 92, 209],
[94, 79, 8, 54],
]
inv_mix_columns(state)
inv_shift_rows(state)
```

```
print(bytes(sum(state, [])))
```

Solution: crypto{d1ffUs3R}

Confusion through substitution:

```
s_box = (
    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,
)

inv_s_box = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3,
0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE,
0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA,
0xC3, 0x4E,
```

```
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B,
0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65,
0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7,
0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8,
0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13,
0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4,
0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75,
0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18,
0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78,
0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80,
0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9,
0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83,
0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21,
0x0C, 0x7D,
)

state = [
    [251, 64, 182, 81],
    [146, 168, 33, 80],
    [199, 159, 195, 24],
    [64, 80, 182, 255],
]

def sub_bytes(s, sbox=s_box):
    for i in range(4):
        for j in range(4):
            print(chr(sbox[s[i][j]]), end="")
print(sub_bytes(state, sbox=inv_s_box))
```

flag: crypto{l1n34rly}

Modular square root soluions:

```
def legendre_symbol(a, p):
    return pow(a, (p - 1) // 2, p)

def tonelli_shanks(n, p):
```

```python
    if legendre_symbol(n, p) != 1:
        return None  # No solution exists

    # Factorize p-1 as 2^s * q
    q = p - 1
    s = 0
    while q % 2 == 0:
        q //= 2
        s += 1

    # Find a non-residue (i.e., a number whose Legendre symbol is -1)
    z = 2
    while legendre_symbol(z, p) != p - 1:
        z += 1

    # Initialize variables
    m = s
    c = pow(z, q, p)
    t = pow(n, q, p)
    r = pow(n, (q + 1) // 2, p)

    while t != 1:
        # Find the smallest i such that t^(2^i) = 1
        i = 0
        tmp = t
        while tmp != 1:
            tmp = (tmp * tmp) % p
            i += 1

        # Update variables
        b = pow(c, 2 ** (m - i - 1), p)
        m = i
        c = (b * b) % p
        t = (t * b * b) % p
        r = (r * b) % p

    return r

def modular_square_root(a, p):
    sqrt_a_mod_p = tonelli_shanks(a, p)
    if sqrt_a_mod_p is not None:
        # There are two square roots modulo p, return the smaller one
        return min(sqrt_a_mod_p, p - sqrt_a_mod_p)
    else:
        return None
```

```
# Example usage
a =
847999465831677215194161651009712708755454127481243511200942577859549535970
024447040064240374705856680712781416539664021584419232790045411625797948743
201676932997076704673509124989867808806163479655955670495984642413182041604
843650138761721177012429279330807921415317997762444043861695857505836119397
568662004643987730833998929560453786749368387277884392177130730560277639878
697835386623166145337605677197206977639899901376958893619485934494126822318
419723136888706060921287550751893617206070220955712443047713742184713068260
166696869165144723691701863490240770479732850946185484243201500987801135402
2108661461024768

p =
305318518619943332526759351114879506944143327639090835141337698613509608950
765046872613698157357425494287891383008430820865500590828351414545266181606
341099691954863220157759430300604495570900648119401394317352091859964547391
635559107264935972226468555064456029536895274053622079269904423917050146047
770386858805275374898453591015524422928043984726423566093048106807315565420
023015478466351014559957325840713559030108567186807323373691284986552552770
036436690316945168513905059234167106012126184431098440415149424019696291589
754570790269063043287490399972629603012091581759200518906209470639363473072
38412281568760161

result = modular_square_root(a, p)
print(f"The modular square root of {a} modulo {p} is: {result}")
```

OUTPUT:

236233930768304863832777329858048929893213750552050038833827105205373474786
235177964731417681795335907187156004112528991924714607490715161276264086819
962118655952206833803260099131188222401602122267224313936218046123264673246
584884042545825793088785658337960096776173859678287785131848935567982281315
512304570528511209944814642675511016000251559241885043210364181581107154845
628426350780558944507365756538185052136796967569976075531078462357707644003
774768176030243492493211364006173877760119462224419275802418085391624442725
406544196255728257284916277274079898964794864520734973745744544040505715689
7508368531939120

Chinese reminder theorem code as well the output:

```
def chinese_remainder_theorem(moduli, remainders):
    """
    Chinese Remainder Theorem implementation
    :param moduli: List of pairwise coprime moduli
    :param remainders: List of remainders corresponding to the moduli
    :return: The solution modulo the product of moduli
    """
    def modinv(a, m):
```

```
    """
    Modular multiplicative inverse
    :param a: Integer
    :param m: Modulus
    :return: Modular inverse of a modulo m
    """
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

  product = 1
  for m in moduli:
      product *= m

  result = 0
  for mi, ai in zip(moduli, remainders):
      bi = product // mi
      result += ai * modinv(bi, mi) * bi

  return result % product

# Example usage
moduli = [5, 11, 17]
remainders = [2, 3, 5]

result = chinese_remainder_theorem(moduli, remainders)
print("The solution a is:", result)
```

The solution a is: 872

Adrien's signs:

```
from sympy.ntheory import legendre_symbol
p = 1007621497415251
c = [67594220461269, 501237540280788, 718316769824518, 296304224247167,
48290626940198, 30829701196032, 521453693392074, 840985324383794,
770420008897119, 745131486581197, 729163531979577, 334563813238599,
289746215495432, 538664937794468, 894085795317163, 983410189487558,
863330928724430, 996272871140947, 352175210511707, 306237700811584,
631393408838583, 589243747914057, 538776819034934, 365364592128161,
```

454970171810424, 986711310037393, 657756453404881, 388329936724352, 90991447679370, 714742162831112, 62293519842555, 653941126489711, 448552658212336, 970169071154259, 339472870407614, 406225588145372, 205721593331090, 926225022409823, 904451547059845, 789074084078342, 886420071481685, 796827329208633, 433047156347276, 21271315846750, 719248860593631, 534059295222748, 879864647580512, 918055794962142, 635545050939893, 319549343320339, 93008646178282, 926080110625306, 385476640825005, 483740420173050, 866208659796189, 883359067574584, 913405110264883, 898864873510337, 208598541987988, 23412800024088, 911541450703474, 57446699305445, 513296484586451, 180356843554043, 756391301483653, 823695939808936, 452898981558365, 383286682802447, 381394258915860, 385482809649632, 357950424436020, 212891024562585, 906036654538589, 706766032862393, 500658491083279, 134746243085697, 240386541491998, 850341345692155, 826490944132718, 329513332018620, 41046816597282, 396581286424992, 488863267297267, 92023040998362, 529684488438507, 925328511390026, 524897846090435, 413156582909097, 840524616502482, 325719016994120, 402494835113608, 145033960690364, 43932113323388, 683561775499473, 434510534220939, 92584300328516, 763767269974656, 289837041593468, 11468527450938, 628247946152943, 8844724571683, 813851806959975, 72001988637120, 875394575395153, 70667866716476, 75304931994100, 226809172374264, 767059176444181, 45462007920789, 472607315695803, 325973946551448, 64200767729194, 534886246409921, 950408390792175, 492288777130394, 226746605380806, 944479111810431, 776057001143579, 658971626589122, 231918349590349, 699710172246548, 122457405264610, 643115611310737, 999072890586878, 203230862786955, 348112034218733, 240143417330886, 927148962961842, 661569511006072, 190334725550806, 763365444730995, 516228913786395, 846501182194443, 741210200995504, 511935604454925, 687689993302203, 631038090127480, 961606522916414, 138550017953034, 932105540686829, 215285284639233, 772628158955819, 496858298527292, 730971468815108, 896733219370353, 967083685727881, 607660822695530, 650953466617730, 133773994258132, 623283311953090, 436380836970128, 237114930094468, 115451711811481, 674593269112948, 140400921371770, 659335660634071, 536749311958781, 854645598266824, 303305169095255, 91430489108219, 573739385205188, 400604977158702, 728593782212529, 807432219147040, 893541884126828, 183964371201281, 422680633277230, 218817645778789, 313025293025224, 657253930848472, 747562211812373, 83456701182914, 470417289614736, 641146659305859, 468130225316006, 46960547227850, 875638267674897, 662661765336441, 186533085001285, 743250648436106, 451414956181714, 527954145201673, 922589993405001, 242119479617901, 865476357142231, 988987578447349, 430198555146088, 477890180119931, 844464003254807, 503374203275928, 775374254241792, 346653210679737, 789242808338116, 48503976498612, 604300186163323, 475930096252359, 860836853339514, 994513691290102, 591343659366796, 944852018048514, 82396968629164, 152776642436549, 916070996204621, 305574094667054, 981194179562189, 126174175810273, 55636640522694, 44670495393401,

```
74724541586529, 988608465654705, 870533906709633, 374564052429787,
486493568142979, 469485372072295, 221153171135022, 289713227465073,
952450431038075, 107298466441025, 938262809228861, 253919870663003,
835790485199226, 655456538877798, 595464842927075, 191621819564547]
print(bytes.fromhex(hex(int(''.join(['1' if legendre_symbol(i,p)==1 else '0' for i in c]),
2))[2:]).decode())
```

Solution: crypto{p4tterns_1n_re5idu3s}

Modular Binomials:

```
from math import gcd

n =
14905562257842714057932724129575002825405393502650869767115942606408600343380327866258982402447992564988466588305174271674657844352454543958847568190372446723549627752274442789184236490768272313187410077124234699854724907039770193680822495470532218905083459730998003622926152590597710213127952141056029516116785229504645179830037937222022291571738973603920664929150436463632305664687903244972880062028301085749434688159905768052041207513149370212313943117665914802379158613359049957688563885391972151218676545972118494969247440489763431359679770422939441710783575668679693678435669541781490217731619224470152467768073
e1 =
12886657667389660800780796462970504910193928992888518978200029826975978624718627799215564700096007849924866627154987365059524315097631111242449314835868137
e2 =
12110586673991788415780355139635579057920926864887110308343229256046868242179445444897790171351302575188607117081580121488253540215781625598048021161675697
c1 =
14010729418703228234352465883041270611113735889838753433295478495763409056136734155612156934673988344882629541204985909650433819205298939877837314145082403528055884752079219150739849992921393509593620449489882380176216648401057401569934043087087362272303101549800941212057354903559653373299153430753882035233354304783275982332995766778499425529570008008029401325668301144188970480975565215953953985078281395545902102245755862663621187438677596628109967066418993851632543137353041712721919291521767262678140115188735994447949166616101182806820741928292882642234238450207472914232596747755261325098225968268926580993051
c2 =
14386997138637978860748278986945098648507142864584111124202580365103793165811666987664851210230009375267398957979494066880296418013345006977654742303441030008490816239306394492168516278328851513359596253775965916326353050138738183351643338294802012193721879700283088378587949
```

```
21991198231956871429805847767716137817313612304833733918657887480468724409753522369325138502059408241232155633806496752350562284794715321835226991147547651155287812485862794935695241612676255374480132722940682140395725089329445356434489384831036205387293760789976615210310436732813848937666608611803196199865435145094486231635966885932646519

q1 = pow(c1, e2, n)
q2 = pow(c2, e1, n)
d = pow(5, e1 * e2, n) * q1 - pow(2, e1 * e2, n) * q2
q = gcd(d, n)
p = n // q
print("crypto{%d,%d}" % (p,q))
```

Solutions:

crypto{112274000169258486390262064441991200608556376127408952701514962644340921899196091557519382763356534106376906489445103255177593594898966250176773605432765983897105047795619470659157057093771407309168345670541418772427807148039207489900810013783673957984006269120652134007689272484517805398390277308001719431273,132760587806365301971479157072031448380135765794466787456948786731168095877956875295282661565488242190731593282663694728914945967253173047324353981530949360031535707374701705328450856944598803228299967009004598984671293494375599408764139743217465012770376728876547958852025425539298410751132782632817947101601}

Keyed Permutations:

Solution: crypto{bijection}

Resisting Brute-force: crypto{biclique}

Structure of AES:

```
def bytes2matrix(text):
    """ Converts a 16-byte array into a 4x4 matrix.  """
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    """ Converts a 4x4 matrix into a 16-byte array.  """
    text = ''
    for i in range(len(matrix)):
        for j in range(4):
            text += chr(matrix[i][j])
    return text

matrix = [
    [99, 114, 121, 112],
    [116, 111, 123, 105],
    [110, 109, 97, 116],
    [114, 105, 120, 125],
]

print(matrix2bytes(matrix))
```

Solutions: crypto{inmatrix}

Round Keys:

```
state = [
    [206, 243, 61, 34],
    [171, 11, 93, 31],
    [16, 200, 91, 108],
    [150, 3, 194, 51],
]

round_key = [
    [173, 129, 68, 82],
    [223, 100, 38, 109],
    [32, 189, 53, 8],
    [253, 48, 187, 78],
]

def add_round_key(s, k):
    for i in range(4):
        for j in range(4):
            print(chr(s[i][j]^k[i][j]), end="")

print(add_round_key(state, round_key))
```

Solution: crypto{r0undk3y}

Lazy CBC:

```
def encrypt(plaintext):
    plaintext = bytes.fromhex(plaintext)
    if len(plaintext) % 16 != 0:
        return {"error": "Data length must be multiple of 16"}

    cipher = AES.new(KEY, AES.MODE_CBC, KEY)
    encrypted = cipher.encrypt(plaintext)

    return {"ciphertext": encrypted.hex()}


@chal.route('/lazy_cbc/get_flag/<key>/')
def get_flag(key):
    key = bytes.fromhex(key)

    if key == KEY:
        return {"plaintext": FLAG.encode().hex()}
    else:
        return {"error": "invalid key"}


@chal.route('/lazy_cbc/receive/<ciphertext>/')
def receive(ciphertext):
    ciphertext = bytes.fromhex(ciphertext)
    if len(ciphertext) % 16 != 0:
        return {"error": "Data length must be multiple of 16"}

    cipher = AES.new(KEY, AES.MODE_CBC, KEY)
    decrypted = cipher.decrypt(ciphertext)

    try:
        decrypted.decode() # ensure plaintext is valid ascii
    except UnicodeDecodeError:
        return {"error": "Invalid plaintext: " + decrypted.hex()}

    return {"success": "Your message has been received"}
```

Flag: crypto{50m3_p30pl3_d0n7_7h1nk_IV_15_1mp0r74n7_?}

Privacy enhanced mail:

```
from Crypto.PublicKey import RSA

f = open('privacy_enhanced_mail_1f696c053d76a78c2c531bb013a92d4a.pem','r')
a = RSA.importKey(f.read())
print(a.d)
```

output:
1568270028805633136478717104581997365499114994919795992986086122818002170731685192445620554366556581089267419005983133023143697091447477456271494562051914438978515890899418195134884601743250646416356496099378425415339540679910131476003344506519342959251234995202098293221852446234100210206343548931881331646451162173694393844071047069491233623768021974620459512895916180059521636623753829644733537581887195252002699310214832889708354718428649324119150595360166885894112979096690923694112785137020242113589709108676356988476009911229107205697063638041734901957976874805476010483879042470898826044392690667379597510468

salty

from Crypto.Util.number import inverse, long_to_bytes n = 110581795715958566206600392161360212579669637391437097703685154237017351570464767725324182051199901920318211290404777259728923614917211291562555864753005179326101890427669819834642007924406862482343614488768256951616086287044725034412802176312273081322195866046098595306261781788276570920467840172004530873767
e = 1 ct = 4498123071821218360427478592579314544265546502526455404602825131116449412748 5

print(long_to_bytes(ct))

flag: crypto{saltstack_fell_for_this!}

Flipping cookie:

Flag: crypto{4u7h3n71c4710n_15_3553n714l}

Confusion through substitution:

s_box = (

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,
)

inv_s_box = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
```

```
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)


state = [
    [251, 64, 182, 81],
    [146, 168, 33, 80],
    [199, 159, 195, 24],
    [64, 80, 182, 255],
]
def sub_bytes(s, sbox=s_box):
    #result = [[0 for j in range(4)] for i in range(4)]
    for i in range(4):
        for j in range(4):
```

```
            print(chr(sbox[s[i][j]]), end="")
print(sub_bytes(state, sbox=inv_s_box))
```

output: crypto{l1n34rly}

whats a lattice

```
import numpy as np
v1=[6, 2, -3]
v2=[5, 1, 4]
v3=[2, 7, 1]

numpy_array=np.asarray([v1,v2,v3])
det = np.linalg.det(numpy_array)
print(f"Volume: {abs(det)}")
```

```
Volume: 254.99999999999991
```

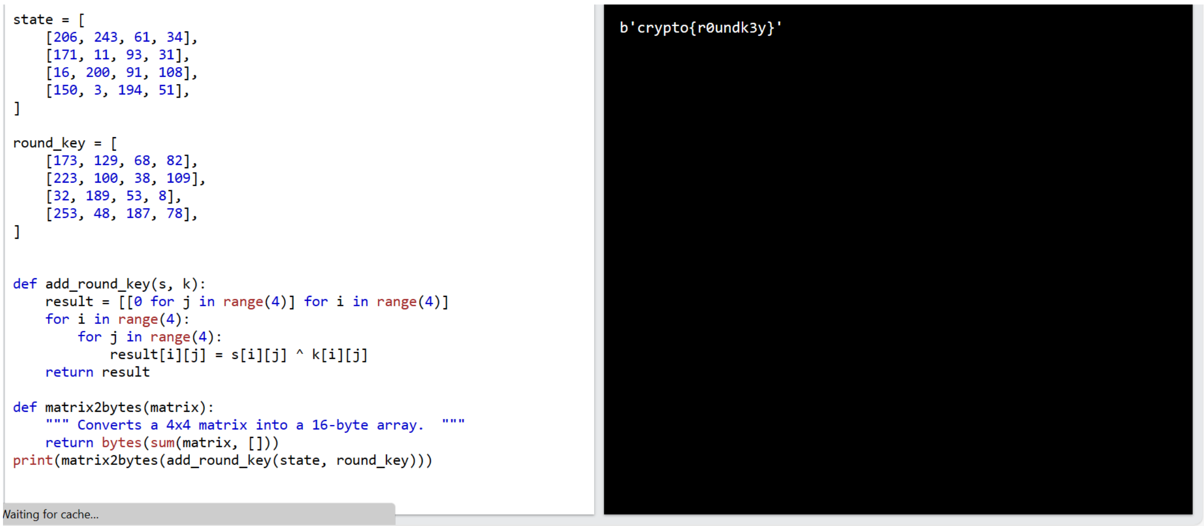
Structure of AES:

```
def bytes2matrix(text):
    """ Converts a 16-byte array into a 4x4 matrix.  """
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    """ Converts a 4x4 matrix into a 16-byte array.  """
    text = ''
    for i in range(len(matrix)):
        for j in range(4):
            text += chr(matrix[i][j])
    return text

matrix = [
    [99, 114, 121, 112],
    [116, 111, 123, 105],
    [110, 109, 97, 116],
    [114, 105, 120, 125],
]

print(matrix2bytes(matrix))
```

Solutions:

crypto{inmatrix}

Round keys:



```
state = [
    [206, 243, 61, 34],
    [171, 11, 93, 31],
    [16, 200, 91, 108],
    [150, 3, 194, 51],
]

round_key = [
    [173, 129, 68, 82],
    [223, 100, 38, 109],
    [32, 189, 53, 8],
    [253, 48, 187, 78],
]


def add_round_key(s, k):
    result = [[0 for j in range(4)] for i in range(4)]
    for i in range(4):
        for j in range(4):
            result[i][j] = s[i][j] ^ k[i][j]
    return result

def matrix2bytes(matrix):
    """ Converts a 4x4 matrix into a 16-byte array.  """
    return bytes(sum(matrix, []))
print(matrix2bytes(add_round_key(state, round_key)))
```


Bringing it altogether:

```
N_ROUNDS = 10

key        = b'\xc3,\\\xa6\xb5\x80^\x0c\xdb\x8d\xa5z*\xb6\xfe\\'
ciphertext = b'\xd1O\x14j\xa4+O\xb6\xa1\xc4\x08B)\x8f\x12\xdd'

s_box = (
    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,
)

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

```
def bytes2matrix(text):
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    out = []
    for r in matrix:
        for c in r:
            out.append(c.to_bytes(2,byteorder='little').decode())
    return ''.join(out)

def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][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[1][3], s[2][3], s[3][3], s[0][3]

def inv_sub_bytes(s, sbox=inv_s_box):
    for i in range(len(s)):
        for j in range(len(s[i])):
            s[i][j] = (sbox[s[i][j]])


def add_round_key(s, k):
    for i in range(len(s)):
        for j in range(len(s[i])):
            s[i][j] = (s[i][j] ^ k[i][j])

xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)

def mix_single_column(a):
    # 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(s):
    for i in range(4):
        mix_single_column(s[i])


def inv_mix_columns(s):
    # see Sec 4.1.3 in The Design of Rijndael
```

```
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)

def expand_key(master_key):
    """
    Expands and returns a list of key matrices for the given master_key.
    """

    # Round constants https://en.wikipedia.org/wiki/AES_key_schedule#Round_constants
    r_con = (
        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,
    )

    # Initialize round keys with raw key material.
    key_columns = bytes2matrix(master_key)
    iteration_size = len(master_key) // 4

    # Each iteration has exactly as many columns as the key material.
    i = 1
    while len(key_columns) < (N_ROUNDS + 1) * 4:
        # Copy previous word.
        word = list(key_columns[-1])

        # Perform schedule_core once every "row".
        if len(key_columns) % iteration_size == 0:
            # Circular shift.
            word.append(word.pop(0))
            # Map to S-BOX.
            word = [s_box[b] for b in word]
            # XOR with first byte of R-CON, since the others bytes of R-CON are 0.
            word[0] ^= r_con[i]
            i += 1
        elif len(master_key) == 32 and len(key_columns) % iteration_size == 4:
            # Run word through S-box in the fourth iteration when using a
            # 256-bit key.
            word = [s_box[b] for b in word]
```

```
        # XOR with equivalent word from previous iteration.
        word = bytes(i^j for i, j in zip(word, key_columns[-iteration_size]))
        key_columns.append(word)

    # Group key words in 4x4 byte matrices.
    return [key_columns[4*i : 4*(i+1)] for i in range(len(key_columns) // 4)]


def decrypt(key, ciphertext):
    round_keys = expand_key(key) # Remember to start from the last round key and work
backwards through them when decrypting

    # Convert ciphertext to state matrix
    state = bytes2matrix(ciphertext)
    # Initial add round key step
    add_round_key(state,round_keys[-1])


    for i in range(N_ROUNDS - 1, 0, -1):
        inv_shift_rows(state)
        inv_sub_bytes(state, inv_s_box)
        add_round_key(state,round_keys[i])
        inv_mix_columns(state)

    # Run final round (skips the InvMixColumns step)
    inv_shift_rows(state)
    inv_sub_bytes(state, inv_s_box)
    add_round_key(state,round_keys[0])

    # Convert state matrix to plaintext
    plaintext = matrix2bytes(state)

    return plaintext

print(decrypt(key, ciphertext))
```

flag = crypto{MYAES128}

Mode of operation starter:

```
import requests

# request encrypted flag
r =
requests.get('http://aes.cryptohack.org/block_cipher_starter/encrypt_fl
ag/')
res = r.json()['ciphertext']
# print(res)
```

```
# request plaintext/decrypting flag
endpointdec = 'http://aes.cryptohack.org/block_cipher_starter/decrypt/'
+ res
dec = requests.get(endpointdec)
res1 = dec.json()['plaintext']
# print(res1)

by = bytes.fromhex(res1)
finalres = by.decode()
print(finalres)
```

crypto{bl0ck_c1ph3r5_4r3_f457_!}

Extended GCD

```
a = 26513
b = 32321

if a < b:
    a,b = b,a  # Reversing the order of the given

r1,r2 = a,b
s1,s2 = 1,0
t1,t2 = 0,1

while r2 > 0:
    # The next line is just the computation for the GCD
    q,r = divmod(r1,r2)
    r1,r2 = r2,r

    # The next line is for the computation of the Bézout's identity
    s1,s2 = s2,s1 - q * s2
    t1,t2 = t2,t1 - q * t2

print(f"GCD:{r1}, u:{t1}, v:{s1}")
```

```
GCD:1, u:10245, v:-8404
```

Encoding challenge

```
from pwn import * # pip install pwntools
import json
from Crypto.Util.number import bytes_to_long, long_to_bytes
import base64
import codecs
import array


r = remote('socket.cryptohack.org', 13377, level = 'debug')

def json_recv():
        line = r.recvline()
        return json.loads(line.decode())

def json_send(hsh):
```

```
        request = json.dumps(hsh).encode()
        r.sendline(request)

for i in range(0,101):
        received = json_recv()

        if "flag" in received:
                print(received)
                break

        print("\n\n")
        print("Received type: ")
        print(received["type"])
        print("Received encoded value: ")
        print(received["encoded"])

        encoding = received["type"]
        word = received["encoded"]

        if encoding == "base64":#PASSED
                decoded = base64.b64decode(word).decode('utf-8')
        elif encoding == "hex": #PASSED
                decode_hex = codecs.getdecoder("hex_codec")
                decoded = decode_hex(word)[0].decode('utf-8')
        elif encoding == "rot13":#PASSED
                decoded = codecs.encode(word, 'rot_13')
        elif encoding == "bigint":
                # Spent way too long troubleshooting this
                # Its a string so to make it work you have
                # to convert it.
                decoded = long_to_bytes(int(word,16)).decode('utf-8')
        elif encoding == "utf-8": #PASSED
                decoded = array.array('b', word).tobytes().decode('utf-8')

        print("DECODED: "+decoded)

        to_send = {
                "decoded": decoded
        }
        json_send(to_send)
```

diffy hellman starter-2

```
p = 28151


def is_primitive_element(g):

    # Set of powers generated by g

    powers = set()


    # Calculate powers of g modulo p
```

```
    for i in range(1, p):
        power = pow(g, i, p)
        if power in powers:
            # If a power is repeated, g is not a primitive element
            return False
        powers.add(power)

    # If all elements in Fp are generated by g, it is a primitive element
    return len(powers) == p - 1

# Iterate over elements of Fp
for g in range(1, p):
    if is_primitive_element(g):
        # Found the smallest primitive element
        smallest_primitive_element = g
        break

# Print the smallest primitive element (the flag)
print("Smallest primitive element of Fp:", smallest_primitive_element)
```

output-7

diffey hellamn starter-1

```
p = 991  # Prime modulus
g = 209  # Element in the finite field Fp

# Calculate the modular multiplicative inverse of g modulo p
d = pow(g, -1, p)

print(d)
```

output-569

diffey hellman starter-3

g = 2

p =
2410312426921032588552076022197566074856950548502459942654116941958108831682612228890093858261341614673227141477904012196503648957050582631942730706805009223062734745341073406696246014589361659774041027169249453200378729434170325843778659198143763193776859869524088940195577346119843545301547043747207749969763750084308926339295559968882457872412993810129130294592999947926365264059284647209730384947211681434464714438488520940127459844288859336526896320919633919

a =
9721074438370337962458643162004582468469045984889816058567658904788530882468973454873284910377102192220389309433658486261941098303091793930182167633275721201247601400180386739998376433775904344138666111324039795471506590538973555933944925869784000443754656572960275929483495892164153637226683613286895889965413700975590903351376764115959493358573417971489261516942995759702928098053144314470434694474859576699499890902023202343378903232934018623049865998847 32815

Calculate g^a mod p to obtain the shared secret

shared_secret = pow(g, a, p)

Print the shared secret

print(shared_secret)

output-
180685769784072652332258672182091135848942012812924807867393365353393068167618175384941171571417360435232355655878375925266106118632027421488310488605016436812919171970740229157733048549951352236828939535952390140613802502252241242923897159127216051914467238953239367383226507005731948539979310118268217746536439627742471754343401766634380727697086447583039177640395755067836236831977656602511849206219694145126563805440017724857227134254861610396741 1990437357924

diffey hellman starter-4

A =
7024994321759546827855454126497548290928917435151613399449582140071062529184010196059572046267260420213349302324139391639462982952627264384735237153483986203041033148508748733180928553319502436928729321708341442409686692584583864184092319348082133205673559248373092105553222250560566166423618228522 95

04265881752580410194731633895345823963910901731715743835775619780738974844840425579683385344491015955892106904647602049559477279345982530488299847663103078045601

b =
12019233252903990344598522535774963020395770409445296724034378433497976840167805970589960962221948290951873387728102115996831454482299243226839490999713763440412177965861508773420532266484619126710566414914227560103715336696193210379850575047730388378348266180934946139100479831339835896583443691529372703954589071507717917136906770122077739814262298488662138085608736103418601750861698417340264213867753834679359191427098195887112064503104510489610448294420720

p =
2410312426921032588552076022197566074856950548502459942654116941958108831682612228890093858261341614673227141477904012196503648957050582631942730706805009223062734745341073406696246014589361659774041027169249453200378729434170325843778659198143763193776859869524088940195577346119843545301547043747207749969763750084308926339295559968882457872412993810129130294592999947926365264059284647209730384947211681434464714438488520940127459844288859336526896320919633919

shared_secret = pow(A, b, p)

print(shared_secret)

output-1174130740413820656533832746034841985877302086316388380165984436672307692443711310285014138545204369495478725102882673427892104539120952393788961051992901649694063179853598311473820341215879965343136351436410522850717408445802043003164658348006577408558693502220285700893404674592567626297571222027902631157072143330043118418467094237965591198440803970726604537807146703763571606861448354607502654664700390453794493176794678917352634029713320615865940720837909466

Gussian reduction

import math

def gaussian_lattice_reduction(v1, v2):

    while True:

        # Step (a): Swap vectors if ||v2|| < ||v1||

        if math.sqrt(v2[0]**2 + v2[1]**2) < math.sqrt(v1[0]**2 + v1[1]**2):

```
        v1, v2 = v2, v1

    # Step (b): Compute m = ⌊ v1·v2 / v1·v1 ⌉
    m = math.floor((v1[0]*v2[0] + v1[1]*v2[1]) / (v1[0]**2 + v1[1]**2))

    # Step (c): If m = 0, return v1, v2
    if m == 0:
        return v1, v2

    # Step (d): v2 = v2 - m*v1
    v2 = (v2[0] - m*v1[0], v2[1] - m*v1[1])

# Define the initial vectors
v = (846835985, 9834798552)
u = (87502093, 123094980)

# Apply Gaussian lattice reduction
v1, v2 = gaussian_lattice_reduction(v, u)

# Calculate the inner product of the new basis vectors
inner_product = v1[0]*v2[0] + v1[1]*v2[1]

# Print the inner product (the flag)
print("Inner product of the new basis vectors:", inner_product)
```

output- 7410790865146821

Size and basis

```
import math

# Define the vector
v = (4, 6, 2, 5)
```

```
# Calculate the size (norm) of the vector
size = math.sqrt(sum(component ** 2 for component in v))

# Print the size of the vector
print("The size of the vector is:", size)
```

output- 9

vectors

```
# Define the vectors v, w, and u
v = (2, 6, 3)
w = (1, 0, 0)
u = (7, 7, 2)

# Calculate the expression 3*(2*v - w) · 2*u

# Step 1: Calculate the vector 2*v - w
vector_1 = (2 * v[0] - w[0], 2 * v[1] - w[1], 2 * v[2] - w[2])

# Step 2: Multiply each component of vector_1 by 3
vector_2 = (3 * vector_1[0], 3 * vector_1[1], 3 * vector_1[2])

# Step 3: Multiply each component of vector_2 by 2*u and calculate the dot product
result = vector_2[0] * 2 * u[0] + vector_2[1] * 2 * u[1] + vector_2[2] * 2 * u[2]

# Print the result
print("The result of the expression is:", result)
```

output- 702

quadratic residue

```
p = 29
ints = [14, 6, 11]


def find_quadratic_residue(p, ints):
    quadratic_residue = None
    for a in range(1, p):
        if (a**2) % p in ints:
            quadratic_residue = (a**2) % p
            break
    return quadratic_residue


def calculate_square_root(p, quadratic_residue):
    a = 1
    while (a**2) % p != quadratic_residue:
        a += 1
    return a


quadratic_residue = find_quadratic_residue(p, ints)
square_root = calculate_square_root(p, quadratic_residue)


print(square_root)
```

output-8

legendry symbol

```
import math


# The prime number (p) and the list of integers (ints)
p =
101524035174539890485408575671085261788758965189060164484385690801466167356
667036677932998889725476582421738788500738738503134356158197247473850273565
349249573867251280253564698939768700489401960767007716413932851838937641880
157263936985954881657889497583485535527613578457628399173971810541670838543
309159139
```

ints =
[25081841204695904475894082974192007718642931811040324543182130088804239047149283334700530600468528298920930150221871666297194395061462592781551275161695411167049544771049769000895119729307495913024360169904315078028798025169985966732789207320203861858234048872508633514498384390497048416012928086480326832803,
45471765180330439060504647480621449634904192839383897212809808339619841633826534856109999027962620381874878086991125854247108359699799913776917227058286090426484548349388138935504299609200377899052716663351188664096302672712078508601311725863678223874157861163196340391008634419348573975841578359355931590555,
17364140182001694956465593533200623738590196990236340894554145562517924989208719245429557645254953527658049246737589538280332010533027062477684237933221198639948938784244510469138826808187365678322547992099715229218615475923754896960363138890331502811292427146595752813297603265829581292183917027983351121325,
14388109104985808487337749876058284426747816961971581447380608277949200244660381570568531129775053684256071819837294436069133592772543582735985855506250660938574234958754211349215293281645205354069970790155237033436065434572020652955666855773232074749487007626050323967496732359278657193580493324467258802863,
4379499308310772821004090447650785095356643590411706358119239166662089428685562719233435615196994728767593223519226235062647670077854687031681041462632566890129595506430188602238753450337691441293042716909901692570971955078924699306873191983953501093343423248482960643055943413031768521782634679536276233318,
85256449776780591202928235662805033201684571648990042997557084658000067050672130152734911919581661523957075992761662315262685030115255938352540032297113615687815976039390537716707854569980516690246592112936796917504034711418465442893323439490171095447109457355598873230115172636184525449905022174536414781771,
50576597458517451578431293746926099486388286246142012476814190030935689430726042810458344828563913001012415702876199708216875020997112089693759638454900092580746638631062117961876611545851157613835724635005253792316142379239047654392970415343694657580353333217547079551304961116837545648785312490665576832987,
96868738830341112368094632337476840272563704408573054404213766500407517251810212494515862176356916912627172280446141202661640191237336568731069327906100896178776245311689857997012187599140875912026589672629935267844696976980890380730867520071059572350667913710344648377601017758188404474812654737363275994871,
4881261656846638800623549662943393234361061827128610120046315649707078244180313661063004390750821317096754282796876479695558644108492317407662131441224257537276274962372021273583478509416358764706098471849536036184924640593888902859441388472856822541452041181244337124767666161645827145408781917658423571721,

```
182379367263675566641714275754755964607273693682462861388042847421242567003
671332500786085371298779682878854574179578685805533719994142274847376036889
926209532001436880610240926235564710530064641232051338946079238013719860274
582743437378603954962605386631831938775398151792467005258651521656009851052
57601565]


def calculate_legendre_symbol(a, p):
    """
    Calculate the Legendre symbol (a/p) for an integer 'a' modulo prime 'p'.
    Args:
        a (int): The integer 'a' for which to calculate the Legendre symbol.
        p (int): The prime number 'p' modulo which the Legendre symbol is calculated.
    Returns:
        int: The Legendre symbol (a/p). It can be 1, -1, or 0.

    """
    legendre_symbol = pow(a, (p-1)//2, p)
    return legendre_symbol



def find_quadratic_residue(p, ints):
    """
    Find the quadratic residue among the given integers modulo prime 'p'.

    Args:
        p (int): The prime number 'p' modulo which the quadratic residue is calculated.
        ints (list): List of integers among which to find the quadratic residue.

    Returns:
        int: The quadratic residue if found, otherwise None.

    """
```

```
    quadratic_residue = None
    for a in ints:
        legendre_symbol = calculate_legendre_symbol(a, p)
        if legendre_symbol == 1:
            quadratic_residue = a
            break
    return quadratic_residue

def calculate_square_root(p, quadratic_residue):
    """
    Calculate the square root of a quadratic residue modulo prime 'p'.

    Args:
        p (int): The prime number 'p' modulo which the square root is calculated.
        quadratic_residue (int): The quadratic residue for which to calculate the square root.

    Returns:
        int: The square root of the quadratic residue modulo 'p'.

    """
    square_root = pow(quadratic_residue, (p+1)//4, p)
    return square_root

# Find the quadratic residue
quadratic_residue = find_quadratic_residue(p, ints)

if quadratic_residue is not None:
    # Calculate the square root
    square_root = calculate_square_root(p, quadratic_residue)
    print("The square root of the quadratic residue is:", square_root)
```

```
else:
    print("No quadratic residue found in the given integers.")
```

output-
93291799125366706806545638475797430512104976066103610269938025709952247020061090804870186195285998727680200979853848718589126765742550855954805290253592144209552123062161458584575060939481368210688629862036958857604707468372384278049741369153506182660264876115428251983455344219194133033177700490981696141526

diffusion through permutation

```
def shift_rows(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]

# The inv_shift_rows function is the inverse operation of shift_rows.
# It reverses the shift performed in shift_rows, restoring the original state matrix.

def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][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[1][3], s[2][3], s[3][3], s[0][3]

# The mix_single_column function performs the MixColumns operation for a single column of the state matrix.
# It uses multiplication in the Rijndael's Galois field to ensure diffusion and non-linearity.
def mix_single_column(a):
    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)


# The mix_columns function applies the mix_single_column operation to each column of the state matrix.
def inv_mix_columns(s):
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)

# The inv_mix_columns function performs the inverse operation of mix_columns.
# It reverses the mixing by applying inverse transformations to each column of the state matrix.


def inv_mix_columns(s):
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)
state = [
```

```
    [108, 106, 71, 86],
    [96, 62, 38, 72],
    [42, 184, 92, 209],
    [94, 79, 8, 54],
]

inv_mix_columns(state)
inv_shift_rows(state)

result = []
for row in state:
    result.extend(row)

flag = bytes(result)
print(flag)
```