CRYPTOHACK: SYMMETRIC CRYPTOGRAPHY SOLUTIONS

CRYPTOGRAPHY

daniel.juarez@iteso.mx, ID: 752809

2025-03-04

Contents

1 CRYPTOHACK: SYMMETRIC CRYPTOGRAPHY SOLUTIONS **1**

1.1 KEYED PERMUTATIONS 1

1.2 RESISTING BRUTEFORCING 1

1.3 STRUCTURE OF AES 2

1.4 ROUND KEYS 3

1.5 CONFUSION TROUGH SUBSTITUTION 6

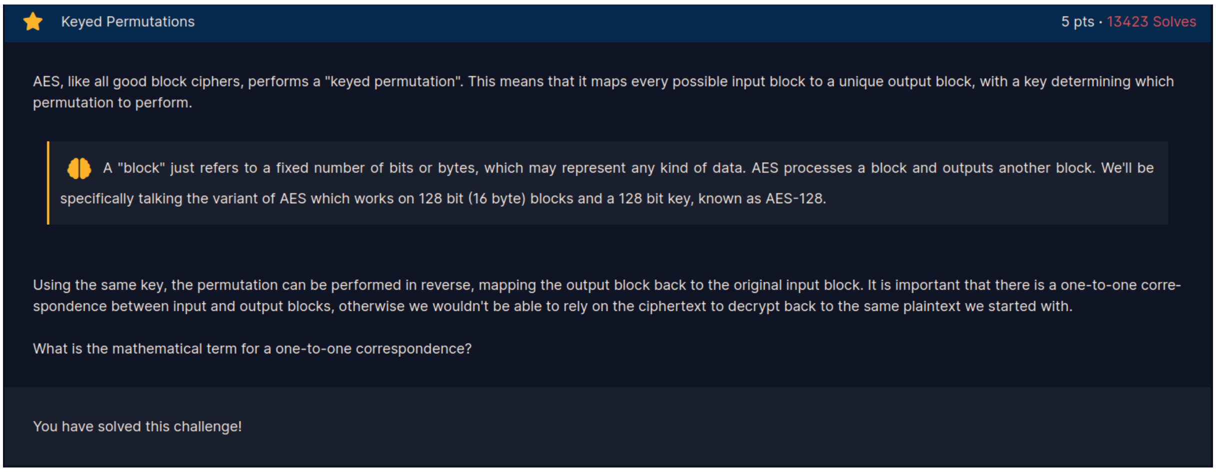
1.6 DIFFUSION TROUGH PERMUTATION 9

1 CRYPTOHACK: SYMMETRIC CRYPTOGRAPHY SOLUTIONS

1.1 KEYED PERMUTATIONS

TASK: What is the mathematical term for a one-to-one correspondence?

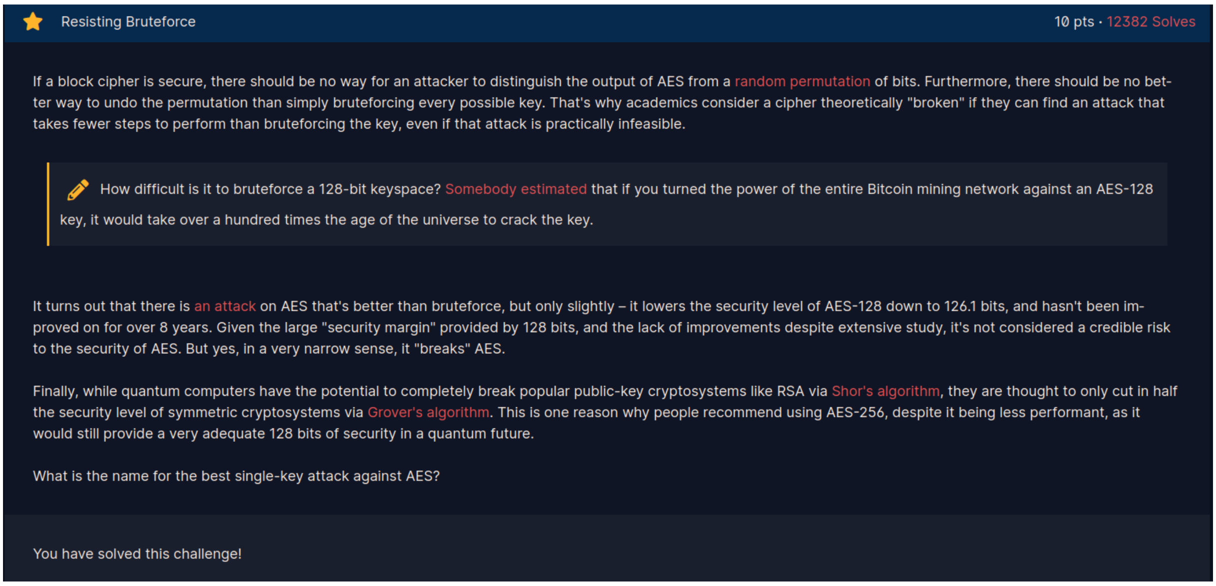
The solution for this challenge is the term **bijection**, which is a reference to how each block in the plaintext has a one-to-one correspondence to each block in the ciphertext.



1.2 RESISTING BRUTEFORCING

TASK: What is the name for the best single-key attack against AES?

After reading the section, finding the solution becomes obvious. The best alternative to attack AES is only via bruteforcing, which would be a **single-key** attack, however, during the reading we find out there is another and better alternative that reduces AES security, which corresponds to the **biclique attack**, being this the answer of the challenge.



Resisting Bruteforce 10 pts · 12382 Solves

If a block cipher is secure, there should be no way for an attacker to distinguish the output of AES from a random permutation of bits. Furthermore, there should be no better way to undo the permutation than simply bruteforcing every possible key. That's why academics consider a cipher theoretically "broken" if they can find an attack that takes fewer steps to perform than bruteforcing the key, even if that attack is practically infeasible.

> How difficult is it to bruteforce a 128-bit keyspace? Somebody estimated that if you turned the power of the entire Bitcoin mining network against an AES-128 key, it would take over a hundred times the age of the universe to crack the key.

It turns out that there is an attack on AES that's better than bruteforce, but only slightly – it lowers the security level of AES-128 down to 126.1 bits, and hasn't been improved on for over 8 years. Given the large "security margin" provided by 128 bits, and the lack of improvements despite extensive study, it's not considered a credible risk to the security of AES. But yes, in a very narrow sense, it "breaks" AES.

Finally, while quantum computers have the potential to completely break popular public-key cryptosystems like RSA via Shor's algorithm, they are thought to only cut in half the security level of symmetric cryptosystems via Grover's algorithm. This is one reason why people recommend using AES-256, despite it being less performant, as it would still provide a very adequate 128 bits of security in a quantum future.

What is the name for the best single-key attack against AES?

You have solved this challenge!

1.3 STRUCTURE OF AES

TASK: Included is a bytes2matrix function for converting our initial plaintext block into a state matrix. Write a matrix2bytes function to turn that matrix back into bytes, and submit the resulting plaintext as the flag.

The given code for this challenge is the following:

```python
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.  """
    ????

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

print(matrix2bytes(matrix))
```

The above will do the following process:

plaintext -> bytes2matrix = matrix

where bytes2matrix in each iteration creates an array of bytes from the text that takes blocks of 4 bytes from the plaintext.

therefore:

matrix2bytes will have to concatenate each matrix row to create the flag

The following solution will successfuly return the flag with the solution from the analysis we made above. The function **matrix2bytes** iterates over the matrix of bytes and appends all bytes converted to characters in the plaintext array:

```
def matrix2bytes(matrix):
    plaintext = []

    for i in range(4):
        for j in range(4):
            plaintext.append(chr(matrix[i][j]))

    return ''.join(plaintext)
```

Where the flag for this challenge was:

```
(env) user@localhost:~/Documents/Write-Ups/cryptography$ python3
↪  matrix_e1b463dddbee6d17959618cf370ff1a5.py
crypto{inmatrix}
```

1.4 ROUND KEYS

TASK: Complete the add_round_key function, then use the matrix2bytes function to get your next flag.

Where the given code is:

```
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):
    ???


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

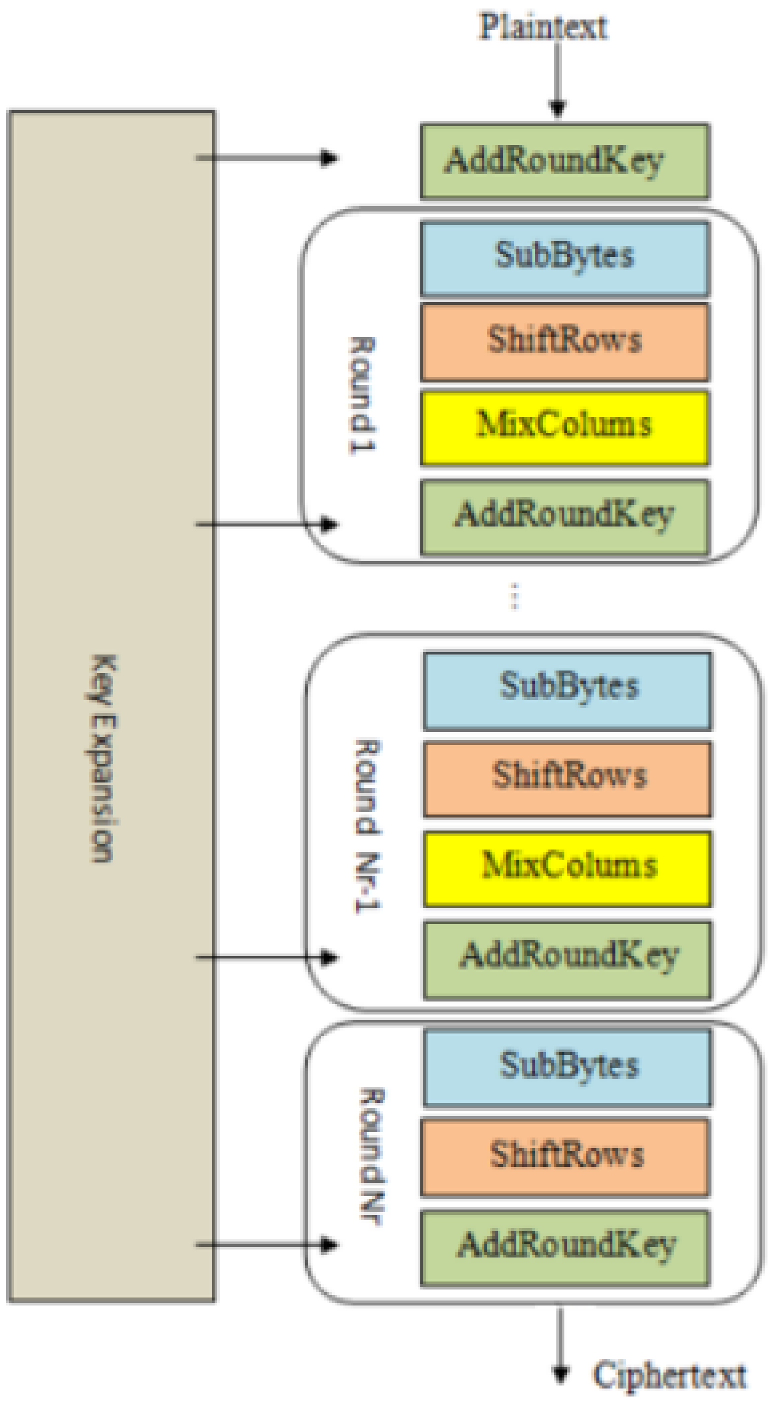


Figure 1.1: AES process

According to the AES process, the **add round key** process involves performing a XOR operation between the round key and the current state data. We assume in this challenge that the given **round key** corresponds to the **initial round key**, therefore a simple XOR operation between arrays would be enough to get the plaintext, therefore, the solution will be as follows:

```python
def matrix2bytes(matrix):
    plaintext = []

    for i in range(4):
        for j in range(4):
            plaintext.append(chr(matrix[i][j]))

    return ''.join(plaintext)


def add_round_key(s, k):
    plaintext = [[s[i][j] ^ k[i][j] for j in range(4)] for i in range(4)]
    return matrix2bytes(plaintext)
```

The **add_round_key** function uses list comprehensions to create the plaintext matrix by performing a XOR operation with each element of the state and the key, the function will return the accommodated plaintext at the end.

```
(env) user@localhost:~/Documents/Write-Ups/cryptography$ python3
↪   add_round_key_b67b9a529ae739156107a74b14adde98.py
crypto{r0undk3y}
```

1.5 CONFUSION TROUGH SUBSTITUTION

TASK: Implement sub_bytes, send the state matrix through the inverse S-box and then convert it to bytes to get the flag.

Where the given code is:

```python
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):
    ???


print(sub_bytes(state, sbox=inv_s_box))
```

The function **sub_bytes** aims to perform confusion by substituing each byte from the state with their corresponding byte in the **sbox**. This creates an almost perfect non-linear relationship between the plaintext and the key given that the substitution depends highly on the state, which from previous exercise we know the state depends from their state key, which ultimately depends on the AES key.

For a byte in the format **0xaf**, the subsitution will be as follows:

sbox[10][15] = s_byte

In the provided code, the **sbox** is given as a **tuple**, so we need to find the corresponding value in the matrix with the following formula:

sbox[a][b] where a and b are integers, sbox[a][b] = (a x sbox_columns) + b

Performing these operations in the **inv_s_box** should retrieve the plaintext, therefore, the solution for this exercise will be the following:

```
def sub_bytes(s, sbox=s_box):
    row = 0
    column = 0
    plaintext = []

    for i in range(4):
        for j in range(4):
            row = int(hex(s[i][j])[2], 16)
            column = int(hex(s[i][j])[3], 16)

            plaintext.append(chr(sbox[(row * 16) + column]))

    return ''.join(plaintext)
```

In the solution, the rows and columns values are extracted by converting the bytes of the state to hexadecimal and extracting the required part, then the desired portion is converted to an integer to finally implement the formula in the tuple and retrieve the desired value.

```
user@localhost:~/Documents/Write-Ups/cryptography/redone$ python3
↪  sbox_8fc04ffb95faf5a5e6959195d5e2d94e.py
crypto{l1n34rly}
```

1.6 DIFFUSION TROUGH PERMUTATION

TASK: We've provided code to perform MixColumns and the forward ShiftRows operation. After implementing inv_shift_rows, take the state, run inv_mix_columns on it, then inv_shift_rows, convert to bytes and you will have your flag.

Where the given code is:

```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):
    ???


# 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],
]
```

The above code performs difusion in the state by spreading part of the input to every part of the output, the **shift_rows** function performs the following operation in the state:

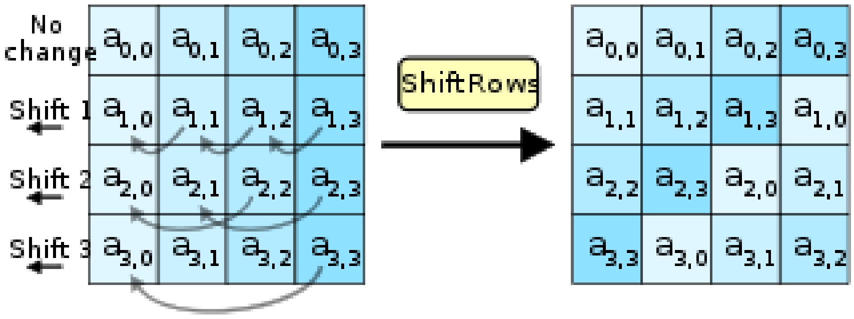


Figure 1.2: shift rows operation

If the original matrix:

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

was replaced for:

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

Then doing the opposite should reasamble the original plaintext, therefore, the solution will be:

```python
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]
```

After doing what the task says, we receive the following output:

```
user@localhost:~/Documents/Write-Ups/cryptography/redone$ python3
↪  diffusion_ee6215282094b4ae8cd1b20697477712.py
crypto{d1ffUs3R}
```