

Trabalho 2 Segurança Computacional

1st Eduardo Ferreira Marques Cavalcante - T02 - 202006368

Departamento de ciência da computação

Universidade de Brasília

Brasília, Brasil

202006368@aluno.unb.br

2nd Pedro Rodrigues Diógenes Macedo - T01 - 211042739

Departamento de ciência da computação)

Universidade de Brasília

Brasília, Brasil

211042739@aluno.unb.br

I. INTRODUÇÃO

Neste trabalho, exploramos a cifra de bloco AES e o modo de operação CTR, dois componentes fundamentais em criptografia. A cifra AES é conhecida por sua robustez e versatilidade, sendo aplicada em diversos tipos de arquivos. Implementaremos tanto a cifração quanto a decifração, com a capacidade de ajustar o número de rodadas e a expansão da chave. Além disso, abordaremos o modo de operação CTR. Os testes incluem a cifração e decifração de um arquivo txt, validando nossa implementação e enfatizando a importância da criptografia na segurança da informação.

O AES (*Advanced Encryption Standard*) é um algoritmo amplamente usado para criptografar informações sensíveis, operando em blocos de 128 bits para as chaves. O modo de operação CTR (*Counter Mode*) é uma técnica que transforma o AES em uma cifra de fluxo, usando um contador para criar uma sequência de valores que são combinados com os dados a serem criptografados, oferecendo eficácia e segurança em criptografia.

O trabalho foi realizado utilizando a linguagem de programação Python com as bibliotecas padrões. A implementação pode ser encontrada em [1]

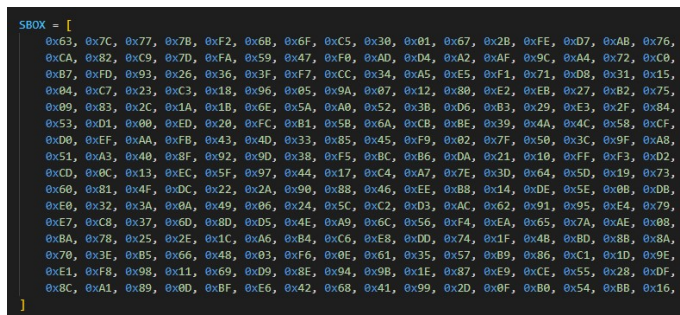
II. METODOLOGIA

A. Expansão de chave

Como dito anteriormente, a primeira parte do trabalho se caracteriza pela expansão da chave de 16 bytes(caracterizado como um bloco de 128 bits), assim, precisamos fazer operações em cima da chave, a qual definimos como um inteiro na base hexadecimal gerado de forma aleatória.

Para a expansão de chave, criamos as seguintes operações, definidas como funções em nosso código na seção `aes.key_expansion.py`.

- K: Retorna 4 bytes da chave original, dado um certo offset. Se for 0, retorna os bytes 0,1,2,3.
- EK: K: Retorna 4 bytes da chave expandida, dado um certo offset. Se for 0, retorna os bytes 0,1,2,3.
- RotWord: Shift circular em cima dos bytes. Ex: 1,2,3,4 se torna 2,3,4,1
- SubWord: Cada bytes é substituído pelo seu valor correspondente na tabela SBOX, assim HEX 19 se tornaria HEX D4:



```
SBOX = [
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x68, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF8, 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, 0x05, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0x8D, 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,
]
```

Figura 1. SBOX: tabela de substituição.

- Rcon: Baseado na rodada, a função retorna um valor baseado na seguinte tabela, com o cálculo do índice, ou seja, de qual posição pegar sendo a seguinte:

$$indice = ((R/(S/4)) - 1) \quad (1)$$

R: Número da rodada

S: Tamanho da chave, sempre de tamanho 16 bytes(128 bits)

Rcon(0)	=	01000000
Rcon(1)	=	02000000
Rcon(2)	=	04000000
Rcon(3)	=	08000000
Rcon(4)	=	10000000
Rcon(5)	=	20000000
Rcon(6)	=	40000000
Rcon(7)	=	80000000
Rcon(8)	=	1B000000
Rcon(9)	=	36000000
Rcon(10)	=	6C000000
Rcon(11)	=	D8000000
Rcon(12)	=	AB000000
Rcon(13)	=	4D000000
Rcon(14)	=	9A000000

Figura 2. Rcon: fonte: [2].

Assim se $R = 4$, índice = 0, com o valor retornado sendo 0x01000000.

Lembrando que sempre lidamos com chaves no formato hexadecimal, gerando um inteiro aleatório de tamanho 16 bytes ou recendo um input de usuário para um chave que deve ter 16 bytes de tamanho, a qual é posteriormente transformada para seu inteiro hexadecimal.

Assim, a expansão é caracterizada por rodadas que se utilizam das funções acima para criar as sub-chaves e que consequentemente se tornaram as chaves de cifração. Como as chaves são de tamanho 16 bytes, 44 rodadas de expansão serão realizadas, de acordo com a seguinte tabela:

16 byte Key Expansion

Each round (except rounds 0, 1, 2 and 3) will take the result of the previous round and produce a 4 byte result for the current round. Notice the first 4 rounds simply copy the total of 16 bytes of the key

Round	Expanded Key Bytes	Function
0	0 1 2 3	$K(0)$
1	4 5 6 7	$K(4)$
2	8 9 10 11	$K(8)$
3	12 13 14 15	$K(12)$
4	16 17 18 19	$\text{Sub Word}(\text{Rot Word}(\text{EK}((4-1)*4))) \text{ XOR } \text{Rcon}((4/4)-1) \text{ XOR } \text{EK}((4-4)*4)$
5	20 21 22 23	$\text{EK}((5-1)*4) \text{ XOR } \text{EK}((5-4)*4)$
6	24 25 26 27	$\text{EK}((6-1)*4) \text{ XOR } \text{EK}((6-4)*4)$
7	28 29 30 31	$\text{EK}((7-1)*4) \text{ XOR } \text{EK}((7-4)*4)$
8	32 33 34 35	$\text{Sub Word}(\text{Rot Word}(\text{EK}((8-4)*4))) \text{ XOR } \text{Rcon}((8/4)-1) \text{ XOR } \text{EK}((8-4)*4)$
9	36 37 38 39	$\text{EK}((8-1)*4) \text{ XOR } \text{EK}((9-4)*4)$
10	40 41 42 43	$\text{EK}((10-1)*4) \text{ XOR } \text{EK}((10-4)*4)$
11	44 45 46 47	$\text{EK}((11-1)*4) \text{ XOR } \text{EK}((11-4)*4)$
12	48 49 50 51	$\text{Sub Word}(\text{Rot Word}(\text{EK}((12-4)*4))) \text{ XOR } \text{Rcon}((12/4)-1) \text{ XOR } \text{EK}((12-4)*4)$
13	52 53 54 55	$\text{EK}((13-1)*4) \text{ XOR } \text{EK}((13-4)*4)$
14	56 57 58 59	$\text{EK}((14-1)*4) \text{ XOR } \text{EK}((14-4)*4)$
15	60 61 62 63	$\text{EK}((15-1)*4) \text{ XOR } \text{EK}((15-4)*4)$
16	64 65 66 67	$\text{Sub Word}(\text{Rot Word}(\text{EK}((16-4)*4))) \text{ XOR } \text{Rcon}((16/4)-1) \text{ XOR } \text{EK}((16-4)*4)$
17	68 69 70 71	$\text{EK}((17-1)*4) \text{ XOR } \text{EK}((17-4)*4)$
18	72 73 74 75	$\text{EK}((18-1)*4) \text{ XOR } \text{EK}((18-4)*4)$
19	76 77 78 79	$\text{EK}((19-1)*4) \text{ XOR } \text{EK}((19-4)*4)$
20	80 81 82 83	$\text{Sub Word}(\text{Rot Word}(\text{EK}((20-4)*4))) \text{ XOR } \text{Rcon}((20/4)-1) \text{ XOR } \text{EK}((20-4)*4)$
21	84 85 86 87	$\text{EK}((21-1)*4) \text{ XOR } \text{EK}((21-4)*4)$
22	88 89 90 91	$\text{EK}((22-1)*4) \text{ XOR } \text{EK}((22-4)*4)$
23	92 93 94 95	$\text{EK}((23-1)*4) \text{ XOR } \text{EK}((23-4)*4)$
24	96 97 98 99	$\text{Sub Word}(\text{Rot Word}(\text{EK}((24-4)*4))) \text{ XOR } \text{Rcon}((24/4)-1) \text{ XOR } \text{EK}((24-4)*4)$
25	100 101 102 103	$\text{EK}((25-1)*4) \text{ XOR } \text{EK}((25-4)*4)$
26	104 105 106 107	$\text{EK}((26-1)*4) \text{ XOR } \text{EK}((26-4)*4)$
27	108 109 110 111	$\text{EK}((27-1)*4) \text{ XOR } \text{EK}((27-4)*4)$
28	112 113 114 115	$\text{Sub Word}(\text{Rot Word}(\text{EK}((28-4)*4))) \text{ XOR } \text{Rcon}((28/4)-1) \text{ XOR } \text{EK}((28-4)*4)$
29	116 117 118 119	$\text{EK}((29-1)*4) \text{ XOR } \text{EK}((29-4)*4)$
30	120 121 122 123	$\text{EK}((30-1)*4) \text{ XOR } \text{EK}((30-4)*4)$
31	124 125 126 127	$\text{EK}((31-1)*4) \text{ XOR } \text{EK}((31-4)*4)$
32	128 129 130 131	$\text{Sub Word}(\text{Rot Word}(\text{EK}((32-4)*4))) \text{ XOR } \text{Rcon}((32/4)-1) \text{ XOR } \text{EK}((32-4)*4)$
33	132 133 134 135	$\text{EK}((33-1)*4) \text{ XOR } \text{EK}((33-4)*4)$
34	136 137 138 139	$\text{EK}((34-1)*4) \text{ XOR } \text{EK}((34-4)*4)$
35	140 141 142 143	$\text{EK}((35-1)*4) \text{ XOR } \text{EK}((35-4)*4)$
36	144 145 146 147	$\text{Sub Word}(\text{Rot Word}(\text{EK}((36-4)*4))) \text{ XOR } \text{Rcon}((36/4)-1) \text{ XOR } \text{EK}((36-4)*4)$
37	148 149 150 151	$\text{EK}((37-1)*4) \text{ XOR } \text{EK}((37-4)*4)$
38	152 153 154 155	$\text{EK}((38-1)*4) \text{ XOR } \text{EK}((38-4)*4)$
39	156 157 158 159	$\text{EK}((39-1)*4) \text{ XOR } \text{EK}((39-4)*4)$
40	160 161 162 163	$\text{Sub Word}(\text{Rot Word}(\text{EK}((40-4)*4))) \text{ XOR } \text{Rcon}((40/4)-1) \text{ XOR } \text{EK}((40-4)*4)$
41	164 165 166 167	$\text{EK}((41-1)*4) \text{ XOR } \text{EK}((41-4)*4)$
42	168 169 170 171	$\text{EK}((42-1)*4) \text{ XOR } \text{EK}((42-4)*4)$
43	172 173 174 175	$\text{EK}((43-1)*4) \text{ XOR } \text{EK}((43-4)*4)$

Figura 3. Rotina para cada rodada da expansão: fonte: [2].

Assim é criada uma chave expandida para o tamanho de 176 bytes, gerando 11 sub-chaves, com a primeira sempre sendo igual aos 16 bytes da primeira chave original. Cada sub-chave será utilizada posteriormente para a cifração da mensagem por blocos (16 bytes). Essa rotina se traduz para o seguinte snippet de código:

```

def expand_key_int(KeySize: int, key_original: int):

    expanded_key = []

    #K function returns 4 bytes of the Key after the specified offset. For example if offset is 0 then K will return
    # bytes 0,1,2,3 of the Expanded Key
    def K(offset: int): return key_original[ offset ]

    # EK function returns 4 bytes of the Expanded Key after the specified offset. For example if offset is 0 then
    # EK will return bytes 0,1,2,3 of the Expanded Key
    def EK(offset: int): return expanded_key[ offset//4 ]

    #returns a 4 byte value based on the RCON table
    def Rcon(round: int):

        i = int(( round/(KeySize/4) ) -1)
        return RCON[i]

    # circular shift on 4 bytes similar
    def RotWord(word: bytes): return ((word << 8) | ((word >> 24) & 0xFF)) & 0xFFFFFFFF

    # S-box value substitution
    def SubWord(word: int):

        aux = 0
        for j in range(4):
            aux = aux + (SBOX[(word & (0xFF << 8*j)) >> 8*j] << 8*j)

        return aux

    expanded_key.append(K(0))
    expanded_key.append(K(1))
    expanded_key.append(K(2))
    expanded_key.append(K(3))

    for index in range(4, 44):

        if index % 4 == 0:

            a = SubWord(RotWord(EK((index-1)*4)))
            b = Rcon(index)
            c = EK((index-4)*4)
            x = a ^ b
            expanded_key.append(x ^ c)

        else:

            expanded_key.append(((EK((index-1)*4) ^ EK((index-4)*4))))

    return expanded_key

```

Figura 4. Expansão de chave com chave hexadecimal de 16 bytes.

B. Bloco AES

O bloco AES foi implementado para cifrar blocos de 16 bytes, ou seja, a cada 16 bytes dos dados que serão cifrados/decifrados será realizado as operações que serão descritas a seguir. Para isso, foram implementadas as 4 funções básicas dessa cifração:

- add round key
- byte sub
- shift row
- mix column

Cada uma dessas funções são chamadas em uma ordem específica e que depende, como já foi visto, da quantidade de rodadas que será cifrado os dados. Essa ordem segue conforme a figura a seguir:

Round	Function
-	Add Round Key(State)
0	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
9	Add Round Key(Shift Row(Byte Sub(State)))

Figura 5. Ordem de chamada das funções na cifração

Round	Function
-	Add Round Key(State)
0	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
1	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
2	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))

3	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
4	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
5	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
6	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
7	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
8	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
9	Add Round Key(Byte Sub(Shift Row(State)))

Figura 6. Ordem de chamada das funções na decifração

Como pode ver, independente de quantas rodadas serão feitas, sempre será realizado um add round key com a primeira chave gerada da expansão. As demais seguem a mesma ordem, tirando a última, que não é feito o mix column.

A função AES mostra como foi implementado o bloco cifrador/decifrador AES:

```
def aes(block, sub_keys: list[int], rounds: int):
    #rodada 0
    state = add_round_key(block, sub_keys[0:4])

    for round in range(rounds):
        if round + 1 < rounds:
            state = enc_block_round(state, sub_keys[(4*round + 4): (4*round + 8)])
        else:
            #última rodada
            state = add_round_key(shift_row(byte_sub(state)),sub_keys[(4*round + 4): (4*round + 8)])

    return state
```

Figura 7. Cifrador AES

```

def dec_aes(block , sub_keys: list[int], rounds: int):

    dec_sub_keys = []

    for i in range(rounds + 1):
        dec_sub_keys = sub_keys[4*i: (4*i + 4)] + dec_sub_keys

    #rodada 0
    state = add_round_key(block, dec_sub_keys[0:4])

    for round in range(rounds):
        if round + 1 < rounds:
            state = dec_block_round(state, dec_sub_keys[(4*round + 4): (4*round + 8)])
        else:
            #última rodada
            state = add_round_key(dec_byte_sub(dec_shift_row(state)),dec_sub_keys[(4*round + 4): (4*round + 8)])

    return state

```

Figura 8. Decifrador AES

Na função add round key é feito um xor do bloco com uma das chaves.

Na função byte sub é feita a substituição de cada um dos bytes do bloco pelo byte respectivo na tabela S BOX (figura4).

Na função shift row os bytes do bloco são organizados em uma matriz 4x4, sendo preenchidas coluna por coluna. Logo após isso, as linhas da matriz são deslocadas.

Na função mix column os bytes do bloco são organizados novamente em uma matriz, conforme na função shift row. Após isso, há uma multiplicação dessa matriz por uma matriz que é constante. Essa multiplicação de matriz é similar a uma multiplicação de matriz normal, porém onde seria soma é feito a operação xor, e onde seria feito uma multiplicação, é feito uma multiplicação de campo de Galois.

Essa multiplicação é feita da seguinte forma, cada um dos elementos que estão sendo multiplicados são substituídos numa tabela constante L. Com os bytes substituídos, são então somados um com o outro. Caso fique mais que 0xFF, o resultado então é subtraído por 0xFF. Com esse resultado é então substituído em uma outra tabela constante E.

Todas essas funções foram implementadas das seguintes formas:

```

def add_round_key(state: list[int], round_key: list[int]):
    new_state = []
    for i in range(4):
        new_state.append(state[i] ^ round_key[i])

    return new_state

def byte_sub(state: list[int]):
    new_state = []
    for i in range(4):
        aux = 0
        for j in range(4):
            aux = aux + (SBOX[(state[i] & (0xFF << 8*j)) >> 8*j] << 8*j)
        new_state.append(aux)

    return new_state

def shift_row(state: list[int]):
    state_matrix = utils.gen_matriz(state)

    state_matrix[1] = state_matrix[1][1:] + [state_matrix[1][0]]
    state_matrix[2] = state_matrix[2][2:] + state_matrix[2][0:2]
    state_matrix[3] = [state_matrix[3][3]] + state_matrix[3][0:3]

    return utils.reverse_gen_matriz(state_matrix)

```

Figura 9. Funções add round, byte sub e shift row


```
def mix_column(state: list[int]):
    mult_matrix = [
        [0x2, 0x3, 0x1, 0x1],
        [0x1, 0x2, 0x3, 0x1],
        [0x1, 0x1, 0x2, 0x3],
        [0x3, 0x1, 0x1, 0x2] ]

    state_matrix = utils.gen_matriz(state)

    result_matrix = [[], [], [], []]

    for coluna in range(4):
        for linha in range(4):
            result_matrix[linha].append(
                mul_galois(state_matrix[linha - linha][coluna], mult_matrix[linha][coluna - coluna]) ^
                mul_galois(state_matrix[linha + 1 - linha][coluna], mult_matrix[linha][coluna + 1 - coluna]) ^
                mul_galois(state_matrix[linha + 2 - linha][coluna], mult_matrix[linha][coluna + 2 - coluna]) ^
                mul_galois(state_matrix[linha + 3 - linha][coluna], mult_matrix[linha][coluna + 3 - coluna])
            )
    result_matrix = utils.reverse_gen_matriz(result_matrix)
```

Figura 10. Função mix column

```
def mul_galois(h: int, i: int):
    if(i == 1):
        return h
    if(h == 1):
        return i
    if(h == 0 or i == 0):
        return 0
    else:
        aux = L_TABLE[h] + L_TABLE[i]
        if aux > 0xFF:
            aux = aux - 0xFF
        return E_TABLE[aux]
```

Figura 11. Função multiplicação de campo de Galois

C. Modo CTR

Esse modo, diferentemente dos outros modos de operação não cifra os dados que serão cifrados pelo bloco AES. Ao invés disso, é passado um vetor de 16 bytes pelo qual será cifrado no bloco cifrador com as chaves da expansão. O resultado dessa cifração é então combinada com o bloco que será cifrado por meio da operação xor. No próximo bloco de 16 bytes que será cifrado, esse vetor é então incrementado e é refeito os passos anteriores.

Isso pode ser visto melhor na figura a seguir:

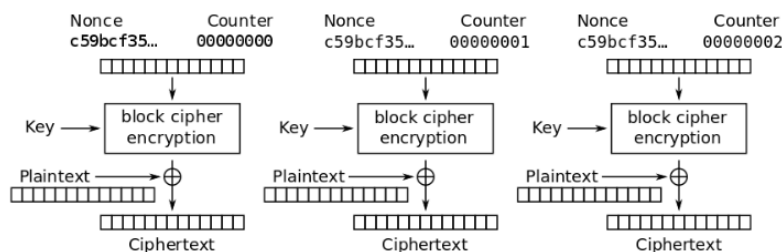


Figura 12. Esquema do modo de operação CTR

Esse modo foi implementado da seguinte forma:

```
def ctr(nomeArquivo, key, rounds, iv, mode):
    bytes_result = b''
    counter = int.from_bytes(iv, 'big')
    sub_keys = expansion.expand_key_int(16, key)
    blocks_16bytes = utils.prepare_message_ctr(nomeArquivo, mode)

    for i in range(0, len(blocks_16bytes)):
        block_counter = utils.ajusta_block_counter(counter.to_bytes(16, 'big'))

        aux = aes(block_counter, sub_keys, rounds)
        key_stream = b''
        for j in range(4):
            key_stream = key_stream + aux[j].to_bytes(4, 'big')

        bytes_result = bytes_result + (int.from_bytes(blocks_16bytes[i], 'big') ^
                                       int.from_bytes(key_stream, 'big')).to_bytes(16, 'big')
        counter += 1

    if mode == "cif":
        with open('cifra_ctr.txt', 'wb') as a:
            a.write(bytes_result)
            a.close()
    else:
        bytes_added = len(bytes_result) - 1
        bytes_result = bytes_result[0:len(bytes_result) - bytes_added]
        print(bytes_added)

        with open('decifrado_ctr.txt', 'wb') as a:
            a.write(bytes_result)
            a.close()
    return bytes_result
```

Figura 13. Função CTR

III. CONCLUSÕES

Nesse trabalho podemos aprender mais a fundo sobre a implementação do bloco cifrador e decifrador AES como também o modo de operação CTR. Percebemos como que essas estratégias são consideradas seguras e porque que são muito utilizadas atualmente. Além disso, é válido ressaltar as dificuldades que tivemos em fazer essa implementação, como a manipulação de tipos entre string e bytes e também a cifração de dados de arquivo, que infelizmente acabamos não conseguindo cifrar um arquivo de imagem para ter uma visualização melhor do impacto da cifra.

REFERÊNCIAS

- [1] E. Ferreira, R. Pedro, "Projeto 2 de Segurança Computacional 2023.2," GitHub, Oct. 30, 2023. <https://github.com/EduardoFMC/SC-AES/tree/main>
- [2] A. Berent, "AES (Advanced Encryption Standard) Simplified", <https://www.ime.usp.br/~rt/cranalysis/AESSimplified>