

Relatório - Advanced Encryption Standard

Matheus Guaraci Lima Bouças Alves - 180046951

Aline Mitiko Otsubo - 170004601

Dep. Ciência da Computação – Universidade de Brasília (UnB)

1. Introdução

Vencedor do concurso proposto pelo National Institute of Standards and Technology (NIST) e substituto do DES, o algoritmo Rijndael (dos criptógrafos belga Vincent Rijmen e Joan Daemen) é uma cifra de blocos que se tornou o Advanced Encryption Standard (AES) [KATZ e LINDELL, 2014]. Nosso trabalho faz a implementação do algoritmo AES em python, utilizando uma chave de 128 bits (16 bytes) e o modo de operação contador (CTR).

2. Algoritmo AES

O AES funciona aplicando os mesmos passos definidos múltiplas vezes, além de também ser reversível (para decifrar, é só trocar a ordem dos passos de cifragem). O algoritmo trabalha com um bloco e altera o seu estado atual (state). O AES pode ser dividido em 4 operações: add round key; byte sub; shift row; mix column. O trecho de código a seguir mostra a implementação das funções de cifração e decifração.

```
def aes_encrypt(block, key, n_round):
    exp_key = key_expansion(key)

    block = add_round_key(exp_key[:16], block)
    del exp_key[:16]

    for i in range(n_round - 1):
        state = sub_bytes(block)
        state = shift_row(state)
        state = mix_columns(state)
        state = add_round_key(exp_key[:16], state)
        del exp_key[:16]

    # ultima rodada
    state = sub_bytes(block)
    state = shift_row(state)
    state = add_round_key(exp_key[:16], state)
    del exp_key[:16]

    return state
```

```
def aes_decrypt(block, key, n_round):
    exp_key = key_expansion(key)

    state = add_round_key(exp_key[-16:], block)
    del exp_key[-16:]

    for i in range(n_round - 1):
        state = shift_row_inv(state)
        state = sub_bytes_inv(state)
        state = add_round_key(exp_key[-16:], state)
        del exp_key[-16:]
        state = mix_columns_inv(state)

    state = shift_row_inv(state)
    state = sub_bytes_inv(state)
    state = add_round_key(exp_key[:16], state)
    del exp_key[-16:]

    return state
```

2.1. Add round key

Nesta etapa, o algoritmo pega a chave de 16 bytes e utiliza a sua expansão para fazer um XOR no estado do bloco a ser cifrado. Cada byte da chave expandida é utilizado uma única vez, nunca sendo reutilizado. Ou seja, os 16 bytes do estado do bloco fazem XOR com 16

bytes da chave, na próxima rodada, outros 16 bytes serão utilizados para fazer o XOR com o estado.

2.2. Byte sub

Nesta etapa, cada byte do estado é substituído por um byte de uma tabela de checagem fixa (a substitution box de Rijndael - S box). No processo de decifração, é utilizada uma tabela inversa.

2.3. Shift row

O algoritmo organiza o estado do bloco a ser cifrado em uma matriz 4x4. A primeira linha é inalterada; a segunda é deslocada à esquerda uma vez; a terceira linha é deslocada à esquerda duas vezes; a quarta é deslocada à esquerda três vezes. Para a decifração, é feito a mesma coisa, mas deslocando as linhas para direita.

2.4. Mix column

Novamente, o estado é organizado como uma matriz 4x4. Mas dessa vez, durante o processo de cifração, ocorre a multiplicação da matriz do estado com uma matriz específica. A multiplicação de matrizes desta etapa é diferente da que estamos imaginando, pois ela é feita no campo de Galois. Uma observação importante, é que a matriz a qual o estado é multiplicado é diferente nos casos de cifração e decifração (neste, utiliza-se a função inversa de mix column). A imagem a seguir mostra apenas a implementação da operação mix column.

```
def mix_columns(state):
    temp = copy(state)
    for i in range(0, 16, 4):
        state[i + 0] = galois_mult(temp[i + 0], 2) ^ galois_mult(temp[i + 3], 1) ^ galois_mult(temp[i + 2],
                                                                                                     1) ^ galois_mult(
                                                                                                     temp[i + 1], 3)
        state[i + 1] = galois_mult(temp[i + 1], 2) ^ galois_mult(temp[i + 0], 1) ^ galois_mult(temp[i + 3],
                                                                                                     1) ^ galois_mult(
                                                                                                     temp[i + 2], 3)
        state[i + 2] = galois_mult(temp[i + 2], 2) ^ galois_mult(temp[i + 1], 1) ^ galois_mult(temp[i + 0],
                                                                                                     1) ^ galois_mult(
                                                                                                     temp[i + 3], 3)
        state[i + 3] = galois_mult(temp[i + 3], 2) ^ galois_mult(temp[i + 2], 1) ^ galois_mult(temp[i + 1],
                                                                                                     1) ^ galois_mult(
                                                                                                     temp[i + 0], 3)
    return state
```

3. Expansão de chave

Como dito anteriormente, a chave utilizada pelo AES é expandida para que seja possível fazer a cifração de um bloco. A chave original, que possui 16 bytes, é expandida até alcançar 176 bytes. As etapas de expansão são: rot word; sub word; rcon; EK; K.

3.1. Rot word

Recebe 4 bytes e faz um shift circular. É semelhante à etapa do shift row.

3.2. Sub word

Aplica a substituição de 4 bytes recebidos por 4 bytes definidos na S box.

3.3. Rcon

Retorna um valor de 4 bytes específico de acordo com a sua tabela.

3.4. EK

Retorna 4 bytes da chave expandida para de acordo com um offset fornecido para a função.

3.5. K

Faz a mesma coisa que EK, mas retorna 4 bytes da chave original.

O código abaixo mostra a implementação da expansão de chave

```
def key_expansion(key):
    w = [[]] * 44

    for i in range(4):
        w[i] = [key[4 * i], key[4 * i + 1], key[4 * i + 2], key[4 * i + 3]]

    for i in range(4, 44):
        temp = w[i - 1]
        word = w[i - 4]

        if i % 4 == 0:
            x = rotate(temp, 1)
            y = sub_bytes(x) # vai conter uma lista de inteiros
            rcon = r_con[int(i / 4)]

            temp = hexor(dec2hex(y), hex(rcon)[2:])

        xord = hexor(dec2hex(word), dec2hex(temp))

        w[i] = [hex2dec(xord[:2]),
                hex2dec(xord[2:4]),
                hex2dec(xord[4:6]),
                hex2dec(xord[6:8])]

    exp_key = []
    for list in w:
        for val in list:
            exp_key.append(val)

    return exp_key # retorna uma lista de inteiros

def add_round_key(key, state):
    for i in range(len(state)):
        state[i] = state[i] ^ key[i]
    return state
```

4. Modo de operação CTR

Com o modo contador, podemos utilizar o AES como uma cifra de fluxo. Utilizamos um nonce de 16 bytes gerado aleatoriamente, somamos ele com um contador que inicializa em 0. Ciframos a soma e fazemos um XOR com o plaintext, gerando o ciphertext de um bloco. Ao final teremos um fluxo de ciphertext de diversos blocos. A implementação está na imagem

abaixo.


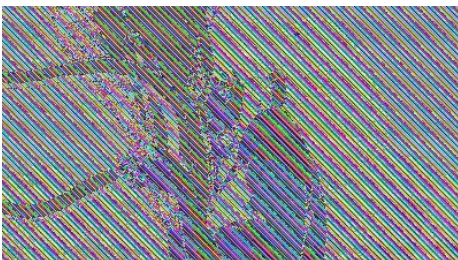
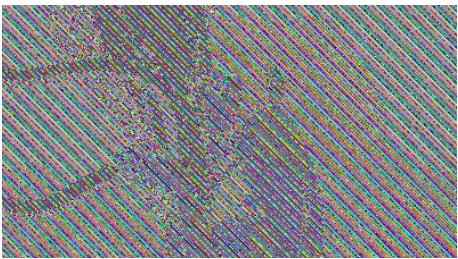
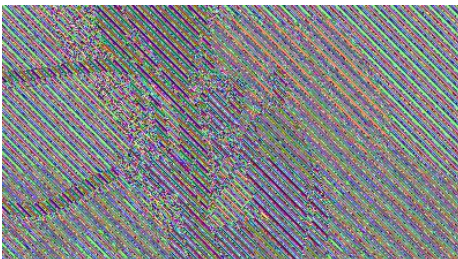
```
# modo ctr
def aes_ctr_encrypt(plaintext, key, nonce, n_round):
    ciphertext = []

    for i in range(0, len(plaintext), 16):
        ###counter = nonce.to_bytes(16, byteorder='big') # transforma nonce em uma sequencia de 16 bytes
        counter = i
        counter = counter.to_bytes(16, byteorder='big')
        count_list = list(counter) # cria uma lista para os 16 bytes, cada elemento equivale a 1 byte
        keystream = aes_encrypt(count_list, key, n_round)

        # XOR da keystream com o bloco de texto original
        ciphertext_block = [p ^ k for p, k in zip(plaintext[i:i + 16], keystream)]
        ciphertext.extend(ciphertext_block)
        # print(ciphertext_block, i)
        nonce = nonce + counter # Incrementa o nonce para o próximo bloco
        ###nonce+=1

    return ciphertext
```

Conforme o código acima, a cifração possui o número de rodadas a ser executada como parâmetro. Isto torna possível visualizar o resultado final da cifração numa rodada específica. As imagens a seguir mostram o resultado de cifrar uma imagem com 1, 5 e 9 rodadas.

	
Imagem original	rodada: 1
	
rodada: 5	rodada: 9

5. Conclusão

Apesar da implementação não ser das mais complexas, o AES é um algoritmo longo. Nos detalhes omitidos nas fontes é onde está a sua verdadeira complexidade. O livro do Katz e

Lindell apresenta a estrutura do algoritmo apenas para uma discussão numa abstração de alto nível. O livro até aconselha a não utilizar a sua descrição para implementação do algoritmo.

No documento do Adam, ele deixa de explicar as operações feitas no campo de Galois. Diversas tabelas do campo de Galois são apresentadas (como a própria S box), mas não é mencionado o que elas estão mapeando e como são feitas.

6. Referências

BERENT, Adam – “AES (Advanced Encryption Standard) Simplified”. Versão 1.1
Disponível em: <https://www.ime.usp.br/~rt/cranalysis/AESSimplified>. Acesso em:
25/10/2023

KATZ, Jonathan; LINDELL, Yehuda. Introduction to modern cryptography. 2.ed. Boca Raton: Chapman & Hall/CRC, 2014.