Trabalho 2 (S-AES e AES/Modos de Operação)

Leandro Beloti Kornelius - 211020900
Lucca Magalhães Boselli Couto - 222011552

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
Segurança Computacional, 2025/1

1. Contextualização do AES e S-AES

O AES é um algoritmo de criptografia em que possui chave única, ou seja, é uma criptografia simétrica. Ademais, o texto é processado em blocos de tamanhos definidos por vez, sendo, por isso, uma cifra de bloco. Este algoritmo possui diversas aplicações atualmente, denotando a sua importância no ambiente acadêmico.

Entretanto, devido à complexidade de sua implementação, foi criado o algoritmo S-AES, o qual é uma versão simplificada do algoritmo AES para fins educacionais. Neste relatório abordaremos a implementação do S-AES e realizaremos a documentação de suas características e outras observações.

2. Implementação do AES

Iniciaremos com uma visão geral do algoritmo S-AES, o qual é uma cifra de bloco cujo tamanho é de 16 bits. Sua chave também possui o tamanho de 16 bits.

Com base na chave providenciada, são geradas outras duas chaves, as quais serão usadas nas rodadas do algoritmo. Iremos detalhar cada rodada e suas características particulares em breve.

Iniciaremos a implementação definindo o texto a ser cifrado assim como um exemplo de chave de 16 bits e o S-BOX o qual também terá uma explicação em breve. A declaração da S-BOX, key e do plaintext pode ser visualizada na Figura 1 presente no apêndice desse relatório.

É importante destacar que o código-fonte completo pode ser encontrado no arquivo **"Ex.ipynb"** localizado no seguinte repositório do github: https://github.com/LeandroKornelius/UnB-Advanced-Security/tree/main/Lista%20de%20Ex%202

2.1. Operação AddRoundKey

Essa é a primeira etapa do algoritmo. É aplicado um XOR na chave de 16 bits a um estado do mesmo comprimento. A utilização do XOR é utilizada pois permite que um novo estado diferente do anterior seja gerado e, no processo de descriptação, seja possível retornar ao estado inicial.

Antes das rodadas do algoritmo, é aplicada esta função à mensagem a ser cifrada. Tal mensagem pode ser uma string. Em função disso, é necessário converter essa string em um formato que o restante do algoritmo possa processar, uma matriz 2x2 de nibbles.

Para isso, foram também criadas duas funções auxiliares:

A função ”int_to_matrix” tem como objetivo transformar um número de 16 bits na matriz 2x2 de nibbles que será processada no algoritmo. O deslocamento de bits por 12, 8, 4 e sem deslocamento permite que o inteiro seja dividido em quatro nibbles para montar a matriz. Já a função ”matrix_to_int” tem o objetivo inverso e será usada no final do código para fácil visualização do texto cifrado.

Já a função ”plain_text_to_nibble_matrix” faz uso desta outra função para, a partir de uma string comum, conseguir gerar a matriz de nibbles desejada.

Os códigos para a função AddRoundKey e suas funções auxiliares podem ser visualizados na Figura 2.

2.2. Operação SubNibbles

A operação Substitute Nibbles busca aplicar a S-box fixa definida anteriormente ao estado de 16 bits. Para isso, a seguinte função exibida na Figura 3 foi implementada.

2.3. Operação ShiftRows

A função shift rows envolve trocar os dois últimos nibbles da matriz. Nesse sentido, o código mostrado na Figura 4 foi feito para realizar essa troca.

2.4. Operação MixColumns

Esta etapa é considerada complexa e difícil de compreender. O Campo Finito de Galois serve para reduzir a complexidade através da conversão de bytes para uma forma polinomial. Por termos matrizes de 4 bits, temos GF(2^4).

Dessa forma, será feita uma combinação dos elementos da matriz de estado usando multiplicações e somas neste campo infinito. Assim, é feita uma mistura linear das colunas da matriz.

As funções do Campo Finito de Galois e a de mix columns podem ser encontradas na figura 5 presente no apêndice.

2.5. KeyExpansion

Como teremos três chaves ao todo, será necessário expandir a chave inicial de 16 bits gerando 3 chaves de 16 bits. Para isso, é utilizada uma combinação de S-box e feitas rotações com constantes pré-definidas. Essa implementação pode ser vista na Figura 6.

2.6. Encriptação utilizando S-AES

Com as funções definidas anteriormente, é feita a função principal do S-AES. Esta função segue as seguintes etapas e apresenta resultados intermediários das funções auxiliares:

Etapas antes das rodadas:

1. Conversão da mensagem de string para bits;
2. Aplicação de KeyExpansion para gerar 3 subchaves;
3. Adição da chave original com AddRoundKey.

Primeira rodada:

1. SubNibbles;

2. ShiftRows;
3. MixColumns;
4. AddRoundKey com a primeira chave gerada.

Segunda rodada:

1. SubNibbles;
2. ShiftRows;
3. AddRoundKey com a segunda chave gerada.

Podemos visualizar a implementação na Figura 7.

2.7. Testando com Etapas Intermediárias

Com isso, a seguinte função foi gerada para exibição em hexadecimal e em base64 para facilitar a visualização dos resultados. Por fim, foi chamada a função principal com a função de visualização definida para vermos o resultado final e as etapas intermediárias. Podemos conferir esse teste na Figura 8.

2.8. Comparações e Conclusão

Como mencionado anteriormente, o S-AES é uma simplificação do AES para fins educacionais.

Sob essa ótica, a versão simplificada do algoritmo apresenta blocos e chaves menores de 16 bits, já o AES possui mais opções para aplicações distintas, tendo blocos de 128 bits e chaves de 128, 192 ou 256 bits.

A quantidade de rodadas também é distinta. Enquanto o S-AES possui apenas 2 rodadas, o AES pode possuir 10, 12 ou 14 rodadas a depender do tamanho da chave utilizada no algoritmo.

Apesar das mesmas operações, a S-AES simplifica algumas delas, tornando o algoritmo menos seguro. Entretanto, isso traz simplicidade de entendimento, o que facilita o aprendizado e, por isso, seu uso.

3. Implementação do Modo de Operação ECB com o S-AES

O ECB é um modo de operação para algoritmos de cifra em bloco. Ele funciona dividindo o texto em blocos de tamanho definido pelo algoritmo de encriptação a ser utilizado. Após a divisão, cada bloco é cifrado de forma independente, mesmo que às vezes seja feito de forma paralela usando a mesma chave e o mesmo algoritmo.

Nesse sentido, o resultado deste modo de operação é a concatenação dos blocos cifrados.

Entretanto, apresenta um problema de segurança, pois blocos idênticos de textos simples, como é o caso do teste do código mostrado na Figura 9, geram blocos idênticos de texto cifrado. Com essas igualdades, é possível expor padrões, comprometendo a segurança da encriptação.

A função mostrada na Figura 9 tem o seguinte funcionamento:

1. Conversão de texto para binário;
2. Divisão em blocos de 16 bits e preenchimento caso não seja múltiplo;

3. Cifragem usando o algoritmo S-AES definido bloco a bloco;
4. Concatenação e codificação para base64.

Com o teste acima, também é possível visualizar como mensagens com um padrão apresentam blocos iguais, comprovando a fraqueza deste modo de operação. É importante ressaltar que, devido à grande extensão do output, ele não será anexado neste relatório, porém pode ser encontrado no código-fonte presente no repositório do github.

4. Simulação com AES Real usando Bibliotecas Criptográficas

Para esta parte, faremos o uso do AES Real com diversos modos de operação e compará-los para contrastar o tempo de execução para analisar a eficiência. Ademais, também será denotado o grau de aleatoriedade e segurança.

Primeiramente, vamos instalar os pacotes necessários para usar o AES real e não o simplificado usado apenas para fins educativos. É possível visualizar o pacote usado na Figura 10.

Nessa parte, iremos implementar a cifragem AES nos modos ECB, CBC, CFB, OFB, CTR.

Inicialmente, definimos uma chave ”aleatória” com 32 bytes e um plaintext suficientemente grande.Podemos ver essa implementação na Figura 11. Posteriormente, calcularemos o grau de aleatoriedade (entropia) do texto cifrado, de modo que seja possível avaliar se o texto criptografado possui uma distribuição estatística adequada para uma criptografia segura. É possível verificar o código na Figura 12.

Nossa função principal `encrypt_aes`, presente na Figura 13, realizará algumas etapas:

- Aplicará um *padding* para garantir que o texto tenha um tamanho múltiplo de 16 (tamanho do bloco AES);
- Executará 10 cifragens consecutivas para obter a média de tempos;
- Criará o modo de operação com a chave e, caso necessário, o *IV*;
- Medirá, em milissegundos, o tempo de cifragem utilizando o `perf_counter`;
- Mostrará o modo usado, o *ciphertext* codificado, o tempo médio e a entropia.

Chamaremos a função principal para cada modo de operação e podemos conferir essas chamadas na Figura 14. Além disso, percebe-se que, mesmo que a mensagem e a chave sejam as mesmas, os resultados de cada cifra estão diferentes. Isso acontece, e é desejável que seja desse modo, pois o vetor de inicialização muda a cada execução, já que ele é ”aleatório”. Os outputs dos modos ECB, CBC, CFB, OFB e CTR podem ser visualizados no código-fonte presente no github devido ao seu tamanho, o qual dificulta a visualização no relatório.

No que diz respeito ao tempo de execução, percebe-se que o modo CTR tende a ser mais rápido, pois cada bloco pode ser processado independentemente e em paralelo devido ao fato de que cada contador é único e independente. Outro modo de execução com baixo tempo de execução é o ECB, que é mais simples e não possui vetor de inicialização.

Em relação à entropia, vê-se que o OFB possui o maior grau de entropia. Se compararmos, por exemplo, com o ECB, é esperado que o OFB tenha maior entropia, pois utiliza vetor de inicialização e faz mix de blocos para que não haja padrões.

Abaixo é possível ver uma tabela comparativa dos tempos de execução e entropia de cada modo de execução:

Table 1. Comparação dos modos de operação de cifragem

Critério	ECB	CBC	CFB	OFB	CTR
Tempo (ms)	0.000002260	0.000004060	0.000005790	0.000004820	**0.000001690**
Entropia	7.9585	7.9563	7.9562	**7.9617**	7.9576

A. Anexo I: Declaração da S-BOX

```
import base64

plain_text = "oi"
key = 0x3A94

SBOX = {
    0x0: 0x9, 0x1: 0x4, 0x2: 0xA, 0x3: 0xB,
    0x4: 0xD, 0x5: 0x1, 0x6: 0x8, 0x7: 0x5,
    0x8: 0x6, 0x9: 0x2, 0xA: 0x0, 0xB: 0x3,
    0xC: 0xC, 0xD: 0xE, 0xE: 0xF, 0xF: 0x7,
}
```

Figure 1. Implementação da SBOX e declaração da chave

B. Anexo II: Operação AddRoundKey

```python
def matrix_to_int(matrix):
    return (
        (matrix[0][0] << 12) | (matrix[0][1] << 8) |
        (matrix[1][0] << 4) | matrix[1][1]
    )

def int_to_matrix(int_value):
    return [
        [(int_value >> 12) & 0xF, (int_value >> 8) & 0xF],
        [(int_value >> 4) & 0xF, int_value & 0xF]
    ]

def plain_text_to_nibble_matrix(plain_text):
    bits = int.from_bytes(plain_text.encode(), 'big')
    return  int_to_matrix(bits)

def add_round_key(state, round_key):
    return [
        [state[0][0] ^ round_key[0][0], state[0][1] ^ round_key[0][1]],
        [state[1][0] ^ round_key[1][0], state[1][1] ^ round_key[1][1]]
    ]
```

Figure 2. Implementação da função AddRoundKey

C. Anexo III: Operação SubNibbles

```python
def sub_nibbles(state):
    # state must be in nibble matrix form
    return [[SBOX[nibble] for nibble in row] for row in state]
```

Figure 3. Implementação da função SubNibbles

D. Anexo IV: Operação ShiftRows

```python
def shift_rows(state):
    return [state[0], [state[1][1], state[1][0]]]
```

Figure 4. Implementação da função ShitfRows

E. Anexo V: Operação MixColumns e Campo Finito de Galois

```python
def galois_field_multiplication(a, b):
    p = 0
    for _ in range(4):
        if b & 1:
            p ^= a
        carry = a & 0b1000
        a <<= 1
        if carry:
            a ^= 0b10011
        b >>= 1
    return p & 0xF

def mix_columns(state):
    s00 = galois_field_multiplication(1, state[0][0]) ^ galois_field_multiplication(4, state[1][0])
    s10 = galois_field_multiplication(4, state[0][0]) ^ galois_field_multiplication(1, state[1][0])
    s01 = galois_field_multiplication(1, state[0][1]) ^ galois_field_multiplication(4, state[1][1])
    s11 = galois_field_multiplication(4, state[0][1]) ^ galois_field_multiplication(1, state[1][1])
    return [[s00, s01], [s10, s11]]
```

Figure 5. Implementação da função MixColumns

F. Anexo VI: Key Expansion

```python
def key_expansion(key):
    w = [(key >> 8) & 0xFF, key & 0xFF]
    RCON1, RCON2 = 0b10000000, 0b00110000

    def sub_rot(word):
        return ((SBOX[(word >> 4) & 0xF] << 4) | SBOX[word & 0xF])

    w.append(w[0] ^ RCON1 ^ sub_rot(w[1]))
    w.append(w[1] ^ w[2])
    w.append(w[2] ^ RCON2 ^ sub_rot(w[3]))
    w.append(w[3] ^ w[4])

    k0 = int_to_matrix((w[0] << 8) | w[1])
    k1 = int_to_matrix((w[2] << 8) | w[3])
    k2 = int_to_matrix((w[4] << 8) | w[5])

    return [k0, k1, k2]
```

Figure 6. Implementação da função KeyExpansion

G. Anexo VII: Encriptação usando S-AES

```python
def s_aes_encrypt_block(block_int, key):
    keys = key_expansion(key)
    state = int_to_matrix(block_int)
    print("Texto original:", state)

    state = add_round_key(state, keys[0])
    print("Após AddRoundKey (K0):", state)

    state = sub_nibbles(state)
    print("Após SubNibbles:", state)

    state = shift_rows(state)
    print("Após ShiftRows:", state)

    state = mix_columns(state)
    print("Após MixColumns:", state)

    state = add_round_key(state, keys[1])
    print("Após AddRoundKey (K1):", state)

    state = sub_nibbles(state)
    print("Após SubNibbles:", state)

    state = shift_rows(state)
    print("Após ShiftRows:", state)

    state = add_round_key(state, keys[2])
    print("Após AddRoundKey (K2):", state)

    return matrix_to_int(state)
```

Figure 7. Implementação da função de Encriptação com S-AES

H. Anexo VIII: Testando Etapas Intermediárias

```
[35]: def encrypt_string(plaintext, key):
          binary = int.from_bytes(plaintext.encode(), 'big')
          encrypted = s_aes_encrypt_block(binary, key)
          print("\nTexto cifrado (hex):", hex(encrypted))
          b64 = base64.b64encode(encrypted.to_bytes(2, 'big')).decode()
          print("Texto cifrado (base64):", b64)

      encrypt_string(plain_text, key)
```

```
Texto original: [[6, 15], [6, 9]]
,Após AddRoundKey (K0): [[5, 5], [15, 13]]
,Após SubNibbles: [[1, 1], [7, 14]]
,Após ShiftRows: [[1, 1], [14, 7]]
,Após MixColumns: [[12, 14], [10, 3]]
,Após AddRoundKey (K1): [[5, 9], [10, 0]]
,Após SubNibbles: [[1, 2], [0, 9]]
,Após ShiftRows: [[1, 2], [9, 0]]
,Após AddRoundKey (K2): [[2, 14], [10, 15]]
,
,Texto cifrado (hex): 0x2eaf
,Texto cifrado (base64): Lq8=
```

Figure 8. Testes das etapas intermediárias e resultados

I. Anexo IX: Implementação do ECB como o S-AES

```python
def encrypt_saes_ecb(plain_text, key):

    # Plaintext to binary split into 16-bit blocks
    bits = "".join(format(ord(c), '08b') for c in plain_text)

    # Padding with zeros
    if len(bits) % 16 != 0:
        bits += '0' * (16 - (len(bits) % 16))
    blocks = [bits[i:i+16] for i in range(0, len(bits), 16)]

    print(f"Texto original em binário: {bits}")
    print(f"\nBlocos de 16 bits:")
    for i, block in enumerate(blocks, start=1):
        print(f"Bloco {i}: {block}")

    encrypted_blocks = []
    print(f"\nBloco cifrado em hexadecimal:")
    for i, block in enumerate(blocks, start=1):
        plain_value = int(block, 2)
        encrypted_value = s_aes_encrypt_block(plain_value, key)
        encrypted_blocks.append(encrypted_value)
        print(f"Bloco {i} cifrado: {encrypted_value:04x}")

    combined_bits = ''.join(format(b, '016b') for b in encrypted_blocks)
    encrypted_bytes = int(combined_bits, 2).to_bytes(len(combined_bits)//8, 'big')
    base64_encrypted_text = base64.b64encode(encrypted_bytes).decode('utf-8')

    print(f"\nTexto cifrado em Base64: {base64_encrypted_text}")
    return base64_encrypted_text

encrypt_saes_ecb("ABABABABABAB", 0x3A94)
```

Figure 9. Implementação do ECB como o S-AES

J. Anexo X: Pacote utilizado para o AES real

```
!pip install cryptography
```

Figure 10. Instalação do pacote utilizado para o AES real

K. Anexo XI: Definição do plaintext e criação da chave

```
from math import log2
from collections import Counter
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import os
import base64
from time import perf_counter

plaintext = (
    b"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut "
    b"labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris "
    b"nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit "
    b"esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt "
    b"in culpa qui officia deserunt mollit anim id est laborum."
) * 10

key = os.urandom(32) # AES key of 256 bits (32 bytes)
```

Figure 11. Definição do plaintext e criação da chave

L. Anexo XII: Função do Cálculo de Entropia

```
def byte_entropy(data):
    counter = Counter(data)
    probs = [v / len(data) for v in counter.values()]
    entropy = -sum(p * log2(p) for p in probs)
    return entropy
```

Figure 12. Cálculo do Grau de Aleatoriedade

M. Anexo XIII: Função Principal da Encriptação do AES

```python
# AES encryption base function to encrypt and measure time
def encrypt_aes(mode_name, mode_constructor, iv_or_nonce=None):

    # Padding
    pad_len = 16 - (len(plaintext) % 16)
    padded = plaintext + bytes([pad_len] * pad_len)

    times = []
    last_ciphertext = None

    for _ in range(10):
        if iv_or_nonce:
            mode = mode_constructor(iv_or_nonce)
        else:
            mode = mode_constructor()

        cipher = Cipher(algorithms.AES(key), mode, backend=default_backend())
        encryptor = cipher.encryptor()

        start = perf_counter()
        ciphertext = encryptor.update(padded) + encryptor.finalize()
        end = perf_counter()

        times.append(end - start)
        last_ciphertext = ciphertext  # Keeps last cipher to be printed

    mean = sum(times) / len(times)

    print(f'\n - Modo: {mode_name}')
    print(f'Base64: {base64.b64encode(last_ciphertext).decode()}')
    print(f'Tempo médio de execução: {mean:.9f} milissegundos')
    print(f'Entropia (aleatoriedade): {byte_entropy(last_ciphertext):.4f} bits')
```

Figure 13. Encriptação do AES

N. Anexo XIV: Chamada de cada Modo de Operação

```
# ECB - doesn't have iv
encrypt_aes("ECB", modes.ECB)

# CBC
iv_cbc = os.urandom(16)
encrypt_aes("CBC", modes.CBC, iv_cbc)

# CFB
iv_cfb = os.urandom(16)
encrypt_aes("CFB", modes.CFB, iv_cfb)

# OFB
iv_ofb = os.urandom(16)
encrypt_aes("OFB", modes.OFB, iv_ofb)

# CTR
nonce_ctr = os.urandom(16)
encrypt_aes("CTR", modes.CTR, nonce_ctr)
```

Figure 14. Chamadas dos Modos de Operação