Trabalho de Implementação - Segurança Computacional 2024/1

CIC0201 - Prof. João Gondim

Daniel da Cunha Pereira Luz
211055540
Ciência da Computação
Universidade de Brasília

I. Introdução

Neste relatório será descrito as implementações de algoritmos de cifração e decifração desenvolvidas na matéria de Segurança Computacional da Universidade de Brasília no semestre de 2024.1 [6]. Apresentarei as implementações divididas em duas partes principais:

Na Parte I, foi desenvolvido a cifra de bloco AES [3] com chave e bloco de 128 bits, e aplicado o modo de operação CTR (Counter Mode). Além disso, realizou-se testes para validar as implementações, incluindo a cifração e decifração de arquivos e de uma imagem (selfie) com diferentes números de rodadas AES. Também foi explorado o modo de cifração autenticada GCM como uma atividade extra.

Na Parte II, foi implementado um sistema de geração e verificação de assinaturas digitais usando RSA [8]. Isso envolveu a geração de chaves, cálculo de hashes usando SHA-3, e verificação da assinatura para garantir a integridade dos dados.

Foi escolhido a linguagem Python para desenvolvimento, pela simplicidade e familiaridade que possuo com a ferramenta. O relatório inclui descrições detalhadas, resultados dos testes, e trechos do código fonte das implementações. Todos os scripts desenvolvidos, arquivos usados e especificações se encontram no repositório https://github.com/dancpluz/aes_rsa_python.

II. Parte 1 - Cifra de bloco e modo de operação CTR

A. Etapa 1: Implementação do AES

1) Introdução e Funções Auxiliares: Antes de começar, definiremos que vou implementar o AES-128 básico, tomei como referência o documento oficial [2], a página do wikipedia [3] e alguns vídeos de um youtuber que implementou o mesmo algoritmo [4].

Ao decorrer do projeto, criei algumas funções de utilidade para ajudar em conversões, representações, manusear arquivos e outras funcionalidades que serão aplicadas futuramente.

```python
# Transforma hexadecimal em bytes
def hex_to_bytes(hex_str):
    return bytes.fromhex(hex_str)
```

```python
# Transforma bytes em hexadecimal
def bytes_to_hex(byte_str):
    return byte_str.hex()
```

```python
# Converte o estado de 16 bytes em uma matriz 4x4
def state_to_matrix(state):
    return [[state[r + 4 * c] for c in range(4)] for r in range(4)]
```

```python
# Converte uma matriz 4x4 em um estado linear de 16 bytes
def matrix_to_state(matrix):
    return bytes(matrix[r][c] for c in range(4) for r in range(4))
```

```python
# Le um arquivo e retorna os dados binarios
def read_file(filename):
    with open(filename, 'rb') as file:
        return file.read()
```

```python
# Escreve dados binarios em um arquivo
def write_file(filename, data):
    with open(filename, 'wb') as file:
        file.write(data)
```

Agora em diante mostrarei as funções individuais desenvolvidas para cada etapa do AES [2].

2) SubBytes: Aqui utilizamos uma tabela de substituição fixa, conhecida como S-Box, para substituir cada byte do estado atual. Essa operação adiciona não-linearidade ao algoritmo, tornando mais difícil quebrar a cifra.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 1. Tabela S-Box

A S-Box é uma tabela de 256 bytes, onde cada entrada corresponde a uma substituição específica. A operação SubBytes substitui cada byte do estado pelo valor correspondente na S-Box.

```
def sub_bytes(state):
    return bytes(S_BOX[byte] for byte in state)
```

3) ShiftRows: Essa transformação realiza deslocamentos cíclicos nas linhas da matriz, o que ajuda a propagar a influência de cada byte pelo bloco de dados ao longo das rodadas, contribuindo para a difusão dos bits.

- **Primeira linha:** Não é alterada.
- **Segunda linha:** É deslocada para a esquerda por 1 posição.
- **Terceira linha:** É deslocada para a esquerda por 2 posições.
- **Quarta linha:** É deslocada para a esquerda por 3 posições.

Utilizamos a função `state_to_matrix` para transformar o estado de 16 bytes em uma matriz 4x4. E a função `matrix_to_state` para fazer a operação inversa.

```
def shift_rows(state):
    matrix = state_to_matrix(state)
    for i in range(4):
        matrix[i] = matrix[i][i:] + matrix[i][:i]
    return matrix_to_state(matrix)
```

4) MixColumns: Cada byte em uma coluna é substituído por uma combinação de todos os quatro bytes na coluna. Essa combinação é realizada usando operações de multiplicação e adição em um campo finito $(GF(2^8))$.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < Nb.$$

Figura 2. Multiplicação de matrizes usada em MixColumns

Começamos pela função `mul_gf` realiza a multiplicação de dois bytes a e b no campo finito $(GF(2^8))$. Aqui está o passo a passo do que a função faz:

(i) Realiza a multiplicação de 2 polinômios no campo finito $(GF(2^8))$
(ii) Verifica se o bit mais significativo `(0x80)` está definido. Se estiver, o resultado é reduzido usando o polinômio irredutível `(0x1b)`.
(iii) Retorna o resultado final limitado a 8 bits `(& 0xff)`.

```
def mul_gf(a, b):
    p = 0
    for _ in range(8):
        if b & 1:
            p ^= a
        carry = a & 0x80
        a <<= 1
        if carry:
            a ^= 0x1b
        b >>= 1
    return p & 0xFF
```

Depois podemos criar a função de misturar as colunas, porém precisamos desenvolver uma função para misturar apenas uma. É isso que a função `mix_single_column` faz:

(i) Realiza a multiplicação da coluna pela matriz fixa.
(ii) Calcula o XOR de todos os elementos da coluna `(all_xor)` e atualiza cada elemento usando a função `mul_gf` para multiplicações.
(iii) Retorna a coluna transformada.

```
def mix_single_column(column):
    c_0 = column[0]
    c_1 = column[1]
    c_2 = column[2]
    c_3 = column[3]

    column[0] = mul_gf(c_0, 0x02) ^ mul_gf(c_1, 0x03) ^ c_2 ^ c_3
    column[1] = c_0 ^ mul_gf(c_1, 0x02) ^ mul_gf(c_2, 0x03) ^ c_3
    column[2] = c_0 ^ c_1 ^ mul_gf(c_2, 0x02) ^ mul_gf(c_3, 0x03)
    column[3] = mul_gf(c_0, 0x03) ^ c_1 ^ c_2 ^ mul_gf(c_3, 0x02)

    return column
```

Finalmente podemos aplicar em todas as colunas do estado:

```
def mix_columns(state):
    matrix = state_to_matrix(state)
    for i in range(4):
        column = [row[i] for row in matrix]
        mix_single_column(column)
        for j in range(4):
            matrix[j][i] = column[j]
    return matrix_to_state(matrix)
```

5) KeyExpansion: A expansão de chave no AES gera uma série de chaves de rodada a partir da chave original, que são utilizadas em cada uma das rodadas do processo de cifração. Para o AES-128, começamos com uma chave de 128 bits, se escolhermos 10 rodadas, geramos 11 chaves de 128 bits (uma para cada uma das 10 rodadas e uma chave adicional para a pré-rodada).

A chave de expansão é feita dividindo a chave original em palavras de 32 bits. Cada nova palavra é gerada a partir das palavras anteriores, com uma transformação adicional a cada múltiplo de 4 palavras. A transformação adicional envolve uma rotação de bytes, substituição de bytes usando a S-Box, e uma adição de um valor de constante de rodada (`rcon`).

Antes de implementarmos o KeyExpansion, precisamos definir algumas funções auxiliares:

- **RotWord:** Faz a rotação dos bytes de uma palavra (4 bytes).

```
def rot_word(word):
    return word[1:] + word[:1]
```

- **SubWord:** Aplica a substituição de bytes da S-Box a cada byte de uma palavra.

```
def sub_word(word):
    return bytes(S_BOX[b] for b in word)
```

- **Rcon:** Retorna o valor da constante de acordo com a tabela RCON.

```
RCON = bytes([
    0x01, 0x02, 0x04, 0x08, 0x10,
    0x20, 0x40, 0x80, 0x1B, 0x36,
    0x6C, 0xD8, 0xAB, 0x4D, 0x9A
])

def rcon(i):
    return bytes([RCON[i], 0x00, 0x00, 0x00])
```

Agora, podemos implementar a expansão de chave:

(i) Primeiro dividimos a chave de 128 bits em palavras de 32 bits e adicionamos à chave expandida.
(ii) Calculamos palavras adicionais necessárias para o número de rodadas especificado. A cada múltiplo de `nk`, aplicamos as funções auxiliares.
(iii) No final juntamos todas as palavras da chave expandida em uma única sequência de bytes.

```
def key_expansion(key, num_rounds):

    nk = 4
    nb = 4
    expanded_key = []

    for i in range(nk):
        expanded_key.append(key[4*i:4*(i+1)])

    for i in range(nk, nb * (num_rounds + 1)):
        temp = expanded_key[i-1]

        if i % nk == 0:
            temp = sub_word(rot_word(temp))
            temp = bytes(t ^ r for t, r in zip(temp, rcon(i // nk - 1)))

        expanded_key.append(bytes(t ^ p for t, p in zip(expanded_key[i - nk], temp)))

    return b''.join(expanded_key)
```

6) AddRoundKey: A função AddRoundKey é uma das operações principais do AES e é usada para combinar o estado atual com uma chave de rodada específica por meio de uma operação XOR. Os passos são o seguinte:

(i) Converter o Estado e a Chave em matrizes 4x4 para facilitar a operação XOR.
(ii) Aplicar a Operação XOR entre o estado e a chave.
(iii) Converter o Estado de volta para a representação linear de 16 bytes.

```
    def add_round_key(state, key):

    state_matrix = state_to_matrix(state)
    key_matrix = state_to_matrix(key)

    for r in range(4):
        for c in range(4):
            state_matrix[r][c] ^= key_matrix[r][c]

    return matrix_to_state(state_matrix)
```

7) Cifração: Agora vamos implementar a lógica completa do AES-128, utilizaremos um bloco de 128 bits, uma chave de 128 bits e será possível escolher o número de rodadas, como descrito na especificação. Segue as etapas necessárias:

(i) **KeyExpansion:** Expansão da chave para gerar todas as chaves de rodada necessárias.
(ii) **Round Inicial:**
- AddRoundKey

(iii) **Rounds Especificados:**
- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey

(iv) **Round Final:**

- SubBytes
- ShiftRows
- AddRoundKey

```
def aes_encrypt(plaintext, key, num_rounds=10):
    state = plaintext
    round_keys = key_expansion(key, num_rounds)

    state = add_round_key(state, round_keys[:16])

    for round in range(1, num_rounds):
        state = sub_bytes(state)
        state = shift_rows(state)
        state = mix_columns(state)
        state = add_round_key(state, round_keys[round * 16: (round + 1) * 16])

    state = sub_bytes(state)
    state = shift_rows(state)
    state = add_round_key(state, round_keys[num_rounds * 16: (num_rounds + 1) * 16])

    return state
```

8) Decifração: Para implementar a decifração, precisamos realizar o processo inverso da cifração AES-128. Isso inclui as operações inversas de SubBytes, ShiftRows, MixColumns, e AddRoundKey. A estrutura básica seguirá a mesma lógica, mas na ordem inversa.

- **InvSubBytes:** Substitui cada byte no estado pelo byte correspondente na S-Box inversa.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figura 3. Tabela S-Box Invertida

```
def inv_sub_bytes(state):
    return bytes(INV_S_BOX[byte] for byte in state)
```

- **InvShiftRows:** Desloca as linhas da matriz de estado na direção inversa.

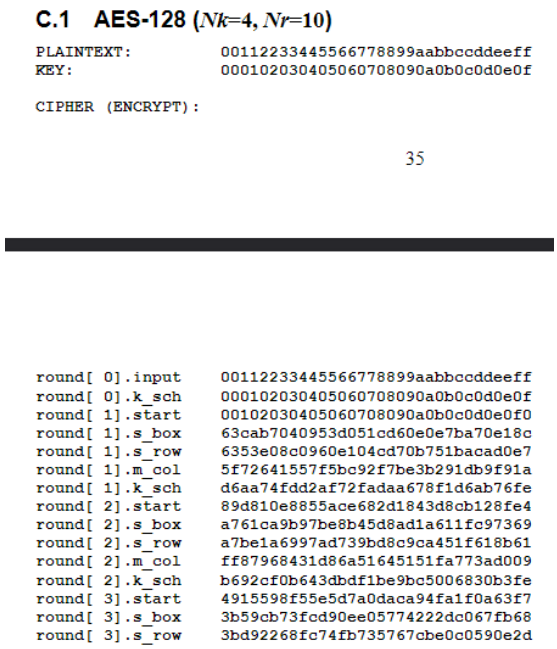
```
def inv_shift_rows(state):
    matrix = state_to_matrix(state)

    for i in range(1, 4):
        matrix[i] = matrix[i][-i:] + matrix[i][:-i]

    return matrix_to_state(matrix)
```

- **InvMixColumns:** Aplica uma transformação inversa no estado.

```
def inv_mix_columns(state):
    matrix = state_to_matrix(state)
    for i in range(4):
        column = [row[i] for row in matrix]
        column = inv_mix_single_column(column)
        for j in range(4):
            matrix[j][i] = column[j]
    return matrix_to_state(matrix)
```

- **AddRoundKey:** Continua sendo a operação XOR, mas agora usando as chaves na ordem inversa.

9) Testes e Conclusões: Para certificar que o algoritmo foi corretamente implementado, utilizei a documentação do AES [2] e segui os testes da página 41 em diante.



C.1 AES-128 (*Nk*=4, *Nr*=10)

PLAINTEXT: 00112233445566778899aabbccddeeff
KEY: 000102030405060708090a0b0c0d0e0f

CIPHER (ENCRYPT):

35

round[0].input 00112233445566778899aabbccddeeff
round[0].k_sch 000102030405060708090a0b0c0d0e0f
round[1].start 00102030405060708090a0b0c0d0e0f0
round[1].s_box 63cab7040953d051cd60e0e7ba70e18c
round[1].s_row 6353e08c0960e104cd70b751bacad0e7
round[1].m_col 5f72641557f5bc92f7be3b291db9f91a
round[1].k_sch d6aa74fdd2af72fadaa678f1d6ab76fe
round[2].start 89d810e8855ace682d1843d8cb128fe4
round[2].s_box a761ca9b97be8b45d8ad1a611fc97369
round[2].s_row a7be1a6997ad739bd8c9ca451f618b61
round[2].m_col ff87968431d86a51645151fa773ad009
round[2].k_sch b692cf0b643dbdf1be9bc5006830b3fe
round[3].start 4915598f55e5d7a0daca94fa1f0a63f7
round[3].s_box 3b59cb73fcd90ee05774222dc067fb68
round[3].s_row 3bd92268fc74fb735767cbe0c0590e2d

Figura 4. Testes para AES-128 com nk=4 e 10 Rodadas

O script a seguir utiliza as mesmas chaves e texto da documentação de teste, verifica a cifração e decifração do algoritmo. Foi observado que a implementação funciona corretamente com os padrões AES-128.

```
if __name__ == "__main__":
    plaintext = hex_to_bytes('00112233445566778899aabbccddeeff')
    key = hex_to_bytes('000102030405060708090a0b0c0d0e0f')
    expected_ciphertext = hex_to_bytes('69c4e0d86a7b0430d8cdb78070b4c55a')

    enciphered = aes_encipher(plaintext, key)
    deciphered = aes_decipher(enciphered, key)

    print("Texto cifrado:", bytes_to_hex(enciphered))
    print("Texto decifrado:", bytes_to_hex(deciphered))
    print('Igual ao original?', bytes_to_hex(plaintext) == bytes_to_hex(deciphered))


    print('Criptografia Correta?', bytes_to_hex(enciphered) == bytes_to_hex(expected_ciphertext))
```

B. Etapa 2: Implementação do modo de operação CTR

1) CTR: O modo CTR transforma uma cifra de bloco em uma cifra de fluxo. Ele opera gerando um fluxo de chaves que é então aplicado ao texto simples usando XOR. Juntaremos com a implementação do AES que desenvolvemos para gerar as chaves e cifrar textos e até arquivos.

(i) **Gerar o Nonce e o Counter:** O `nonce` (número arbitrário usado apenas uma vez) e o `counter` são combinados para formar o "bloco contador". Normalmente, o `nonce` é fixo para uma sessão de comunicação, enquanto o contador começa em 0 e é incrementado para cada bloco.
(ii) **Criptografar o Bloco Contador:** Cada bloco contador é criptografado com a chave AES, gerando o `keystream`.
(iii) **XOR com o Plaintext/Ciphertext:** Com o `keystream` é então aplicado um XOR com o bloco de `plaintext` para gerar o `ciphertext`.

```
def aes_ctr_encipher(plaintext, key, nonce, num_rounds=10):
    if isinstance(plaintext, str):
        plaintext = plaintext.encode('utf-8')

    block_size = 16
    ciphertext = b''

    for i in range(0, len(plaintext), block_size):
        counter = i // block_size
        counter_block = nonce + \
        counter.to_bytes(block_size - len(nonce), byteorder='big')

        keystream = aes_encipher(counter_block, key, num_rounds)

        block = plaintext[i:i + block_size]
        ciphertext_block = bytes(a ^ b for a, b in zip(block, keystream[:len(block)]))
        ciphertext += ciphertext_block

    return ciphertext
```

Para decifrar o CTR realiza-se a mesma operação da cifração:

```
def aes_ctr_decipher(ciphertext, key, nonce, num_rounds=10):
    return aes_ctr_encipher(ciphertext, key, nonce, num_rounds)
```

2) Testes e Conclusões: Foi desenvolvido alguns testes básicos para checar o algoritmo AES-CTR:

```
if __name__ == "__main__":
    key = hex_to_bytes('000102030405060708090a0b0c0d0e0f')

    nonce = hex_to_bytes('1234567890abcdef12345678')

    plaintext = "Este e um texto de teste para o modo CTR."

    ciphertext = aes_ctr_encipher(plaintext, key, nonce)

    decrypted_text = aes_ctr_decipher(ciphertext, key, nonce)

    print("Texto cifrado:", bytes_to_hex(ciphertext))
    print("Texto decifrado:", decrypted_text.decode('utf-8'))
```

Depois de testado, conclui-se que o algoritmo foi implementado com sucesso.

C. Extra 1: Modo de cifração autenticada GCM

GCM é um modo de operação que proporciona tanto confidencialidade quanto integridade/autenticidade dos dados. [5] Ele combina o modo CTR com uma autenticação baseada em Galois.

1) Galois: Primeiramente definimos o algoritmo da multiplicação no campo de Galois $GF(2^{128})$

- A multiplicação é realizada bit a bit. Se o bit menos significativo de `y` for 1, `x` é somado ao resultado parcial usando XOR.
- Se houver um `carry` no bit mais significativo de `x`, ele é manipulado através de um polinômio irreduzível específico (`0xE1000000000000000000000000000000`).
- O resultado é garantido para ter 128 bits.

$$x^{128} + x^7 + x^2 + x + 1$$

Figura 5. Polinómio do campo de Galois

```
def galois_multiply(x, y):
    result = 0
    for i in range(128):
        if y & 1:
            result ^= x
        carry = x & (1 << 127)
        x <<= 1
        if carry:
            x ^= 0xE1000000000000000000000000000000
        x &= (1 << 128) - 1
        y >>= 1
    return result & ((1 << 128) - 1)
```

2) GHASH: Agora podemos calcular aplicar a função anterior para gerar o hash de autenticação através do algoritmo `ghash`.

- Inicializa-se o valor `y` como 0.
- O dado de entrada `data` é processado em blocos de 16 bytes.
- Cada bloco é convertido para um inteiro grande, combinado com `y` via XOR, e então multiplicado no campo de Galois com `h` (o hash subkey).
- O resultado final, `y`, é convertido de volta para bytes e retorna como o GHASH.

```
def ghash(h, data):
    y = 0
    for i in range(0, len(data), 16):
        block = data[i:i+16]
        if len(block) < 16:
            block = block.ljust(16, b'\x00')
        y ^= int.from_bytes(block, byteorder='big')
        y = galois_multiply(y, int.from_bytes(h, byteorder='big'))
    return y.to_bytes(16, byteorder='big')
```

3) Cifração: A combinação da cifragem no modo CTR com a autenticação via GHASH torna o AES-GCM altamente seguro. Não só os dados são mantidos em segredo, mas também que qualquer tentativa de adulteração é detectada e bloqueada.

(i) **Geração do Nonce:** Semelhante ao modo CTR, utiliza-se um `nonce`.
(ii) **Criptografia no Modo CTR:** A cifra em si é feita usando o modo CTR.
(iii) **Cálculo da Tag de Autenticidade:** Uma tag de autenticidade é gerada para garantir que os dados não foram alterados. Isso é feito utilizando o algoritmo do GHASH.

```
def aes_gcm_encipher(plaintext, key, nonce, num_rounds=10):
    ciphertext = aes_ctr_encipher(plaintext, key, nonce, num_rounds)

    h = aes_encipher(b'\x00' * 16, key, num_rounds)
    len_c = len(ciphertext) * 8

    ghash_input = ciphertext + len_c.to_bytes(8, byteorder='big')
    auth_tag = ghash(h, ghash_input)

    tag = aes_ctr_encipher(auth_tag, key, nonce, num_rounds)

    return ciphertext, tag
```

4) Decifração: A decifragem no modo GCM reverte o processo de cifragem usando o modo CTR. O texto cifrado é transformado de volta ao texto original, usando a mesma sequência gerada durante a cifragem. A tag de autenticidade é recalculada e comparada com a original.

```
def aes_gcm_decipher(ciphertext, key, nonce, tag, num_rounds=10):
    plaintext = aes_ctr_decipher(ciphertext, key, nonce, num_rounds)

    h = aes_encipher(b'\x00' * 16, key, num_rounds)
    len_c = len(ciphertext) * 8

    ghash_input = ciphertext + len_c.to_bytes(8, byteorder='big')
    auth_tag = ghash(h, ghash_input)
    expected_tag = aes_ctr_encipher(auth_tag, key, nonce, num_rounds)

    if expected_tag != tag:
        raise ValueError('Tag de autenticidade nao confere, os dados podem ter sido corrompidos!')

    return plaintext
```

5) Teste Arquivo e Openssl: Foram usados dois arquivos para teste, um arquivo de texto (`texto.txt`) e um arquivo de imagem (`selfie.jpg`).

```
base_folder = "arquivos"

def test_aes_gcm_file(input_file, key, nonce, output_enciphered_file='enciphered', \
output_deciphered_name='deciphered', num_rounds=10):
    input_path = path.join(base_folder, input_file)
    output_enciphered_path = path.join(base_folder, output_enciphered_file)
    output_deciphered_path = path.join(base_folder, output_deciphered_name)

    plaintext = read_file(input_path)

    # Criptografar o arquivo
    ciphertext, tag = aes_gcm_encipher(plaintext, key, nonce, num_rounds)
    output_enciphered_name = path.splitext(output_enciphered_path)[0]

    write_file(output_enciphered_name + ".bin", ciphertext)
    write_file(output_enciphered_name + ".tag", tag)

    # Descriptografar o arquivo
    enciphered_data = read_file(output_enciphered_name + ".bin")
    saved_tag = read_file(output_enciphered_name + ".tag")

    output_deciphered_name = path.splitext(output_deciphered_path)[0]
    input_file_ext = path.splitext(input_path)[1]

    try:
        deciphered_plaintext = aes_gcm_decipher(enciphered_data, key, nonce, saved_tag, num_rounds)
        write_file(output_deciphered_name + input_file_ext, deciphered_plaintext)
        print("O modo GCM para arquivo esta funcionando!")
    except ValueError as e:
        print("Erro ao decifrar o arquivo: ", str(e))

if __name__ == "__main__":
    key = hex_to_bytes('000102030405060708090a0b0c0d0e0f')
    nonce = hex_to_bytes('000000000000000000000000')

    test_aes_gcm_file("texto.txt", key, nonce, "texto_cifrado_gcm", "texto_decifrado_gcm")

    test_aes_gcm_file("selfie.jpg", key, nonce, "selfie_cifrado_gcm", "selfie_decifrado_gcm")
```

Para checar a implementação foi utilizado `openssl` para comparar e verificar se a cifra está correta. Rodamos esses comandos de cifração do AES-128 com a mesma `key` e `nonce` do algoritmo.

```
    openssl enc -aes-128-ctr -in texto.txt -out texto_cifrado_openssl.bin \
    -K 000102030405060708090a0b0c0d0e0f -iv 00000000000000000000000

    openssl enc -aes-128-ctr -in selfie.jpg -out selfie_cifrado_openssl.bin \
    -K 000102030405060708090a0b0c0d0e0f -iv 00000000000000000000000
```

Ao cifrar os arquivos com a biblioteca, foi aplicado o algoritmo de decifração desenvolvido. Não foi possível testar o GHASH e a criação da tag com o `openssl`, então a tag usada no algoritmo foi criada anteriormente.

```
    key = hex_to_bytes('000102030405060708090a0b0c0d0e0f')
    nonce = hex_to_bytes('000000000000000000000000')

    enciphered_data = read_file(path.join(base_folder, "selfie_cifrado_openssl.bin"))
    saved_tag = read_file(path.join(base_folder, "selfie_cifrado_gcm.tag"))

    try:
        deciphered_plaintext = aes_gcm_decipher(enciphered_data, key, nonce, saved_tag)
        write_file(path.join(base_folder, 'selfie_decifrado_openssl_gcm.jpg'), deciphered_plaintext)
        print("A decifracao GCM para selfie cifrada em OpenSSL esta funcionando!")
    except ValueError as e:
        print("Erro ao decifrar arquivo OpenSSL: ", str(e))

    enciphered_data = read_file(path.join(base_folder, "texto_cifrado_openssl.bin"))
    saved_tag = read_file(path.join(base_folder, "texto_cifrado_gcm.tag"))

    try:
        deciphered_plaintext = aes_gcm_decipher(enciphered_data, key, nonce, saved_tag)
        write_file(path.join(base_folder, 'texto_decifrado_openssl_gcm.txt'), deciphered_plaintext)
```

```
        print("A decifracao GCM para texto cifrado em OpenSSL esta funcionando!")
    except ValueError as e:
        print("Erro ao decifrar arquivo OpenSSL: ", str(e))
```

6) Teste Selfie: Para este teste tirei essa selfie:

Figura 6. Selfie Autor

Apliquei o algoritmo AES-CTR com as rodadas 1, 5, 9 e 13. Não foi possível visualizar as imagens cifradas em `JPG`, porém utilizando a biblioteca `PIL`, foi possível renderizar as imagens em `bitmap`. Para isso, os bytes são agrupados em pixels de 3 bytes (`RGB`), e o tamanho da imagem é ajustado para formar uma matriz quadrada. Se necessário, os dados são preenchidos com zeros para completar os pixels.

```
def save_enciphered_image(ciphertext, num_rounds):
    num_bytes = len(ciphertext)
    num_pixels = int((num_bytes + 2) / 3)
    W = H = int(math.ceil(num_pixels ** 0.5))

    imagedata = ciphertext + b'\0' * (W * H * 3 - len(ciphertext))

    image = Image.frombytes('RGB', (W, H), imagedata)

    if not os.path.exists('selfies'):
        os.makedirs('selfies')

    image.save(f'selfies/selfie_{num_rounds}rodadas_cifrada.bmp')
    print(f'Imagem cifrada com {num_rounds} rodadas salva como "selfie_{num_rounds}rodadas_cifrada.bmp"')

def aes_ctr_encipher_image(image_file, key, nonce, num_rounds):
    plaintext = read_file(image_file)

    ciphertext = aes_ctr_encipher(plaintext, key, nonce, num_rounds)

    save_enciphered_image(ciphertext, num_rounds)

    return ciphertext

if __name__ == "__main__":
    key = hex_to_bytes('000102030405060708090a0b0c0d0e0f')  # 128-bit key em hex
    nonce = hex_to_bytes('1234567890abcdef12345678')  # 96-bit nonce em hex

    image_file = 'selfie.jpg'

    num_rounds_list = [1, 5, 9, 13]

    for num_rounds in num_rounds_list:
        aes_ctr_encipher_image(image_file, key, nonce, num_rounds)
```

Aqui estão os resultados obtidos, as imagens renderizadas em bitmap:

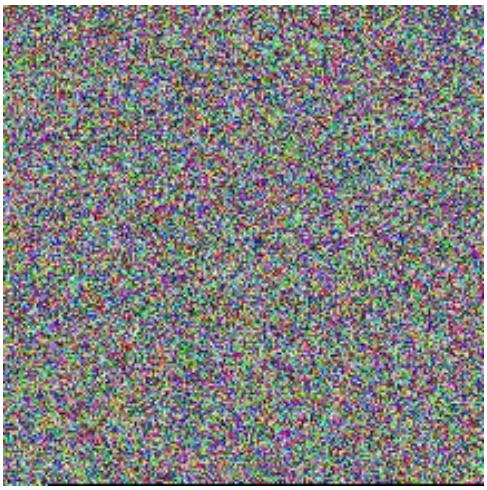


Figura 7. Selfie cifrada com AES-CTR de 1 Rodada

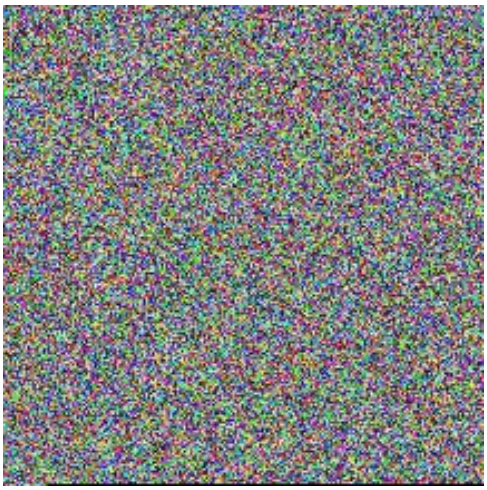


Figura 8. Selfie cifrada com AES-CTR de 5 Rodadas

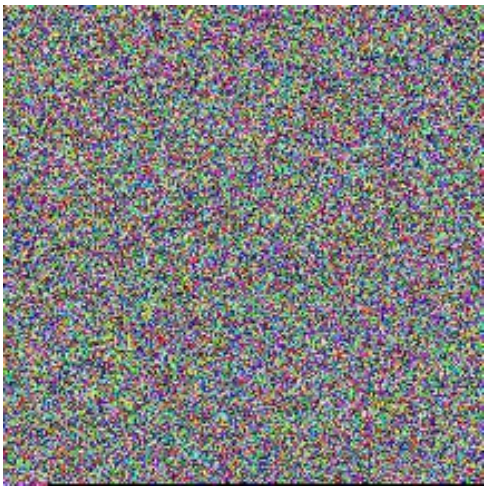


Figura 9. Selfie cifrada com AES-CTR de 9 Rodadas

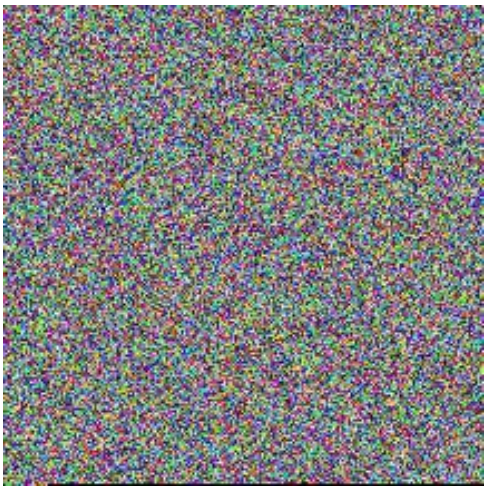


Figura 10. Selfie cifrada com AES-CTR de 13 Rodadas

Isso mostra como a criptografia altera a estrutura dos dados da imagem, tornando-a irreconhecível, e permite observar se há diferenças visuais perceptíveis com o aumento do número de rodadas de cifração.

III. Parte 2 - Gerador/Verificador de Assinaturas RSA

O RSA é um dos algoritmos criptográficos mais utilizados para garantir a segurança e autenticidade de informações em sistemas de comunicação [8]. Este algoritmo, baseado na dificuldade de fatoração de grandes números primos, permite a geração de chaves públicas e privadas para criptografia, além da criação de assinaturas digitais.

Uma assinatura digital RSA garante que um documento ou arquivo não tenha sido alterado após ser assinado, e que a assinatura tenha sido realmente gerada pela entidade detentora da chave privada correspondente à chave pública. Este processo envolve duas etapas principais: a geração de chaves e a assinatura/verificação.

Neste projeto, implementei um gerador e verificador de assinaturas RSA, aplicando-o a três tipos de arquivos: uma imagem (`selfie.jpg`), um documento de texto (`texto.txt`) e um documento PDF (`doc.pdf`). O sistema permite a criação de assinaturas digitais para esses arquivos e a verificação das mesmas para garantir a integridade e autenticidade dos arquivos. Não foi possível desenvolver o método OAEP como nas especificações, mas foi implementado uma versão simples e funcional do RSA. Tomei como referências a página do Wikipedia [7] e uma série de vídeos de um youtuber que implementa o mesmo algoritmo [8].

A. *Introdução e Funções Auxiliares*

Explicação de algumas funções simples que foram utilizadas nos testes e algoritmo principal.

```python
# Converte um numero inteiro para bytes.
def int_to_bytes(x):
    return x.to_bytes((x.bit_length() + 7) // 8, byteorder='big')
```


```python
# Converte um numero inteiro para bytes.
def int_to_bytes(x):
    return x.to_bytes((x.bit_length() + 7) // 8, byteorder='big')
```


```python
# Converte bytes para um numero inteiro.
def bytes_to_int(xbytes):
    return int.from_bytes(xbytes, byteorder='big')
```

```python
# Le um arquivo e retorna os dados binarios
def read_file(file_path):
    with open(os.path.join(base_folder,file_path), 'rb') as file:
        return file.read()
```

```python
# Salva a assinatura digital em um arquivo separado.
def save_signature(signature, file_path):
    signature_file = f"{file_path}.sig"
    with open(os.path.join(base_folder,signature_file), 'w') as file:
        file.write(signature)
    print(f"Assinatura salva em {signature_file}")
```

```python
# Salva o arquivo original com a assinatura digital incorporada.
def save_file_with_signature(file_data, signature, file_path):
    signed_file_path = f"{file_path}.signed"
    with open(os.path.join(base_folder,signed_file_path), 'wb') as file:
        file.write(file_data)
        file.write(b'\n---SIGNATURE---\n')
        file.write(signature.encode())
    print(f"Arquivo com assinatura salvo em {signed_file_path}")
```

B. *Geração de Chaves*

Aqui geramos o par de chaves RSA, pública e privada. [8]

(i) Dois números primos grandes, p e q, são gerados usando a função `generate_prime`.

```python
def generate_prime(bits):
    while True:
        p = random.getrandbits(bits)
        if is_prime(p):
            return p
```

(ii) Utiliza-se a função `is_prime` para fazer o teste de Miller-Rabin e verificar se o número é primo.

```python
def is_prime(n, k=5):
    if n <= 1 or n == 4:
        return False
    if n <= 3:
        return True

    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1

    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1:
```

```
                break
        else:
            return False
    return True
```

(iii) O módulo `n` é calculado como o produto dos dois números primos: `n = p * q`. Este valor faz parte tanto da chave pública quanto da chave privada.
(iv) A função de Euler `phi(n)` é calculada como: `phi = (p - 1) * (q - 1)`. Esse valor é usado para calcular a chave privada.
(v) Um valor padrão para o expoente público `e` é selecionado, geralmente 65537, que é um valor comum devido às suas propriedades de segurança e eficiência.
(vi) A chave privada `d` é calculada como o inverso modular de e em relação a `phi(n): d = pow(e, -1, phi)`. Esse valor é a parte secreta da chave privada e só deve ser conhecido pelo detentor da chave.
(vii) A função retorna duas tuplas: a chave pública e a chave privada.

```
def generate_keys(bits=1024):

    p = generate_prime(bits)
    q = generate_prime(bits)
    n = p * q
    phi = (p - 1) * (q - 1)

    e = 65537
    d = pow(e, -1, phi)

    return ((n, e), (n, d))
```

C. Assinatura de Mensagem

Aqui vamos gerar uma assinatura digital para uma mensagem usando a chave privada RSA. Foi utilizado a biblioteca pública `hashlib` para calcular o hash SHA3-256, e a `base64` para codificação e decodificação Base64, como foi permitido nas orientações [1].

(i) A mensagem é convertida em um valor hash utilizando o algoritmo `SHA3-256`. O hash é então convertido para um número inteiro.
(ii) A assinatura digital é gerada elevando o valor do hash à potência `d` (chave privada) e tomando o resultado módulo `n` (parte comum da chave): `pow(hash_value, d, n)`.
(iii) O valor numérico da assinatura é convertido para bytes usando a função `int_to_bytes`.
(iv) Os bytes da assinatura são codificados em Base64.

```
def sign_message(message, private_key):
    n, d = private_key
    hash_value = int.from_bytes(sha3_256(message).digest(), byteorder='big')
    signature = pow(hash_value, d, n)
    signature_bytes = int_to_bytes(signature)
    return base64.b64encode(signature_bytes).decode()
```

D. Verificar Assinatura

(i) A assinatura digital em Base64 é decodificada de volta para bytes.
(ii) Os bytes da assinatura são convertidos de volta para um número inteiro usando a função `bytes_to_int`.
(iii) O hash criptografado (a assinatura) é descriptografado usando a chave pública, elevando-o a potência `e` e tomando o módulo `n`.
(iv) O hash da mensagem original é gerado novamente usando SHA3-256 para comparação.
(v) A função verifica se o hash obtido a partir da assinatura corresponde ao hash gerado da mensagem original.

```
def verify_signature(message, signature, public_key):
    n, e = public_key
    signature_bytes = base64.b64decode(signature)
    signature_int = bytes_to_int(signature_bytes)
    hash_value = pow(signature_int, e, n)

    calculated_hash = int.from_bytes(sha3_256(message).digest(), byteorder='big')
    return hash_value == calculated_hash
```

E. Testes e Conclusões

Para os testes do algoritmo optei por testar em três arquivos `selfie.jpg`, `texto.txt` e `doc.pdf`. Os três arquivos são lidos e assinados digitalmente usando as funções previamente desenvolvidas, então a assinatura é salva em `.sig` e o arquivo original é salvo com a assinatura incorporado em formato `.signed`. Depois verifica-se o conteúdo original usando a chave pública e repetido com outra chave pública diferente.

```
def test_file_signature(file_path, private_key, public_key):
    print(f"Testando arquivo: {file_path}")
    file_data = read_file(file_path)

    signature = sign_message(file_data, private_key)
    print("Assinatura:")
    print(signature)

    save_signature(signature, file_path)
    save_file_with_signature(file_data, signature, file_path)

    is_valid = verify_signature(file_data, signature, public_key)
    print(f"Assinatura valida: {is_valid}")

    different_public_key, _ = generate_keys()
    is_valid_with_different_key = verify_signature(file_data, signature, different_public_key)
    print(f"Assinatura valida (chave publica diferente): {is_valid_with_different_key}")
    print("\n")

if __name__ == "__main__":
    public_key, private_key = generate_keys()

    test_file_signature("selfie.jpg", private_key, public_key)
    test_file_signature("texto.txt", private_key, public_key)
    test_file_signature("doc.pdf", private_key, public_key)
```

O algoritmo desenvolvido implementa com sucesso uma forma de gerar e verificar assinaturas digitais RSA em arquivos. As funções são modulares e permitem fácil integração e adaptação em diferentes contextos. Todos os scripts desenvolvidos, arquivos usados e especificações se encontram no repositório https://github.com/dancpluz/aes_rsa_python.

REFERÊNCIAS

[1] Especificações do Trabalho CIC0201 - Segurança Computacional – 2024/1, Prof. João Gondim.
[2] PUB, NIST FIPS. 197: Advanced encryption standard (AES). Federal information processing standards publication, v. 197, n. 441, p. 0311, 2001.
[3] Wikipedia contributors. (2024, August 27). Advanced Encryption Standard. In Wikipedia, The Free Encyclopedia. Retrieved 15:17, August 30, 2024, from Advanced Encryption Standard
[4] Cyrill Gössi - Cryptography with Python 7: Implementing AES in Python #1 https://www.youtube.com/watch?v=1gCD1pZKc04
[5] Wikipedia contributors. (2024, June 12). Galois/Counter Mode. In Wikipedia, The Free Encyclopedia. Retrieved 15:17, August 30, 2024, from Galois/Counter Mode
[6] Slides da matéria CIC0201 - Segurança Computacional, Prof. João Gondim.
[7] Cyrill Gössi - Cryptography with Python 16: Implementing RSA in Python #1 https://www.youtube.com/watch?v=58LLuy1B8dk
[8] Wikipedia contributors. (2024, August 2). RSA (cryptosystem). In Wikipedia, The Free Encyclopedia. Retrieved 17:27, August 31, 2024, from RSA (cryptosystem)