Trabalho de Implementação

Segurança Computacional – CIC/UnB

Nome: João Vitor Viana Chaves – 221018951

Professor: João José Costa Gondim

Github: https://github.com/JaumC/TrabalhoDeImplementa-o-SC

2024.1

Relatório

Introdução:

AES(Advanced Encryption Standard) também conhecido como Rijndael é um algoritmo de cifra de bloco simétrico que utiliza uma chave de **128, 192 ou 256 bits** para criptografar dados em blocos de 128 bits. Nesse relatório será explicitado os principais tópicos para o funcionamento desse modelo de criptografia bem como a implementação de um **modo contador** (CTR).

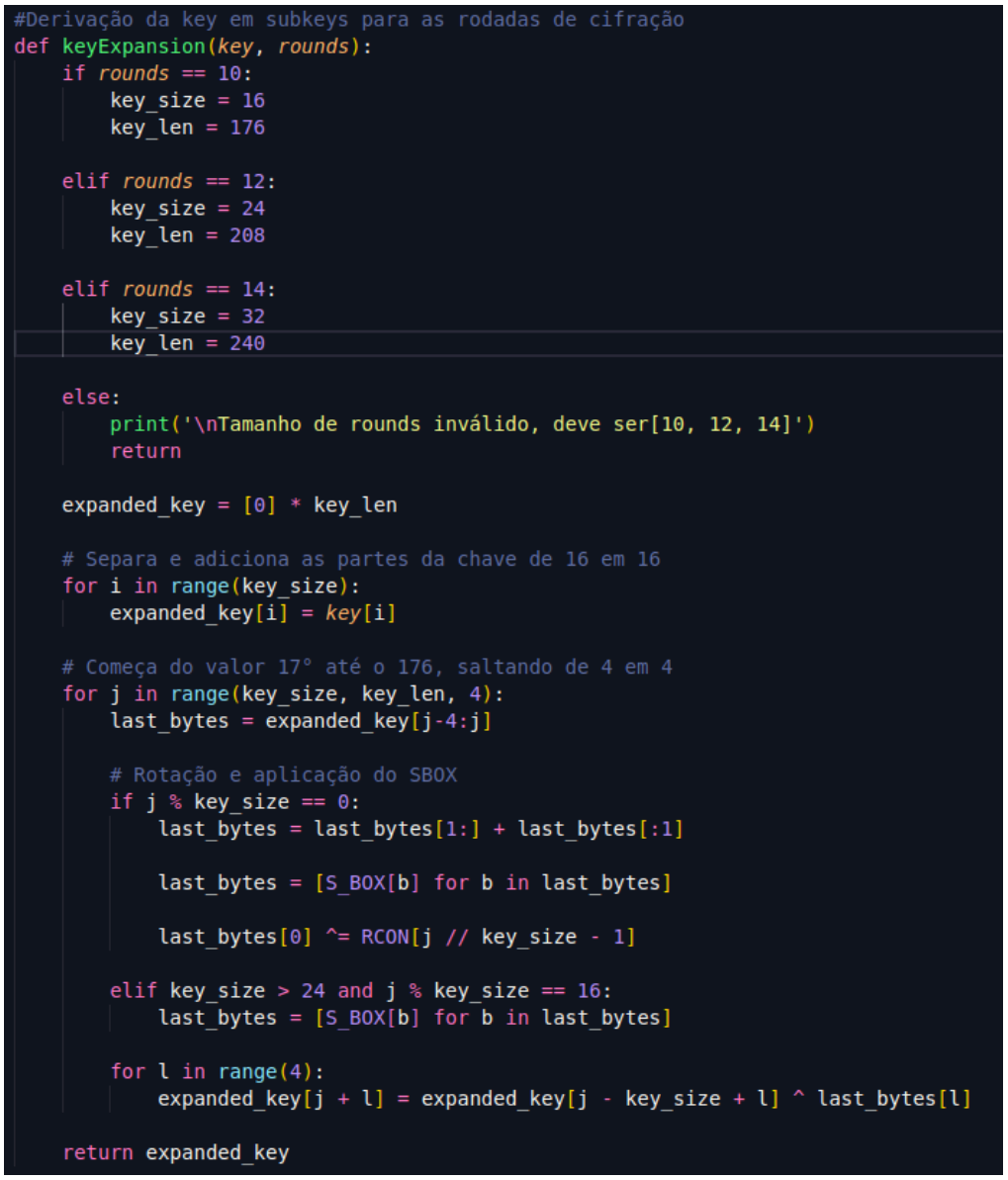
Também será tratado do RSA(Rivest-Shamir-Adleman), que é um dos algoritmos de criptografia assimétrica mais conhecidos e utilizados. Ele é amplamente usado para criptografia de dados e autenticação. Nessa implementação foi usado o **OAEP**(Optimal Asymmetric Encryption Padding) e o **Miller-Rabin**.

Os códigos contam como uma pequena interação em loop, usando a linguagem de programação python, visando manter o dinamismo e fácil viagem de criptografia para testagem do código e suas funções. **As informações de como rodar o projeto estão descritas no *README.md* do projeto.**

Principais Tópicos do AES-CTR:

1. **Expansão da Chave (Key Expansion)**

- O processo de expansão da chave no AES envolve gerar uma série de subchaves a partir da chave original. Essas subchaves são usadas em cada etapa do processo de criptografia e são necessárias para transformar o texto claro em texto cifrado. O algoritmo de expansão de chave é projetado para garantir que a chave original não seja facilmente recuperável e que a criptografia seja segura. A verificação inicial do tamanho da chave é necessária para distinguir quantas subchaves serão geradas, usando o Bloco de Substituição(**S-BOX)** para garantir a não linearidade e a Constante de Rotação**(RCON)** para introduzir variabilidade nas chaves geradas.



```python
#Derivação da key em subkeys para as rodadas de cifração
def keyExpansion(key, rounds):
    if rounds == 10:
        key_size = 16
        key_len = 176

    elif rounds == 12:
        key_size = 24
        key_len = 208

    elif rounds == 14:
        key_size = 32
        key_len = 240

    else:
        print('\nTamanho de rounds inválido, deve ser[10, 12, 14]')
        return

    expanded_key = [0] * key_len

    # Separa e adiciona as partes da chave de 16 em 16
    for i in range(key_size):
        expanded_key[i] = key[i]

    # Começa do valor 17° até o 176, saltando de 4 em 4
    for j in range(key_size, key_len, 4):
        last_bytes = expanded_key[j-4:j]

        # Rotação e aplicação do SBOX
        if j % key_size == 0:
            last_bytes = last_bytes[1:] + last_bytes[:1]

            last_bytes = [S_BOX[b] for b in last_bytes]

            last_bytes[0] ^= RCON[j // key_size - 1]

        elif key_size > 24 and j % key_size == 16:
            last_bytes = [S_BOX[b] for b in last_bytes]

        for l in range(4):
            expanded_key[j + l] = expanded_key[j - key_size + l] ^ last_bytes[l]

    return expanded_key
```

2. **SubBytes**
   - O passo SubBytes é uma substituição não linear onde cada byte do bloco de dados é substituído por um byte de acordo com uma tabela chamada S-BOX. A S-BOX é uma matriz 16x16 que mapeia cada byte para outro byte. Esse

processo introduz não linearidade na cifra, aumentando a segurança do AES contra ataques de análise diferencial e linear.

```
#Percorre o state e substitui cada valor pelo correspondente em SBOX
def subBytes(state):
    for i in range(4):
        for j in range(4):
            state[i][j] = S_BOX[state[i][j]]
    return state
```

3. **ShiftRows**
    - No passo ShiftRows, as linhas do bloco de dados são rotacionadas (ou deslocadas) para a esquerda por diferentes quantidades de bytes. Este passo ajuda a dispersar os bytes ao longo do bloco e contribui para a difusão dos dados, garantindo que alterações em um byte afetam muitos outros bytes após várias rodadas.

```
#Alteração da posição dos bytes nas linhas 1, 2 e 3 da matriz
def shiftRows(state):
    state[1] = state[1][1:] + state[1][:1]
    state[2] = state[2][2:] + state[2][:2]
    state[3] = state[3][3:] + state[3][:3]

    return state
```

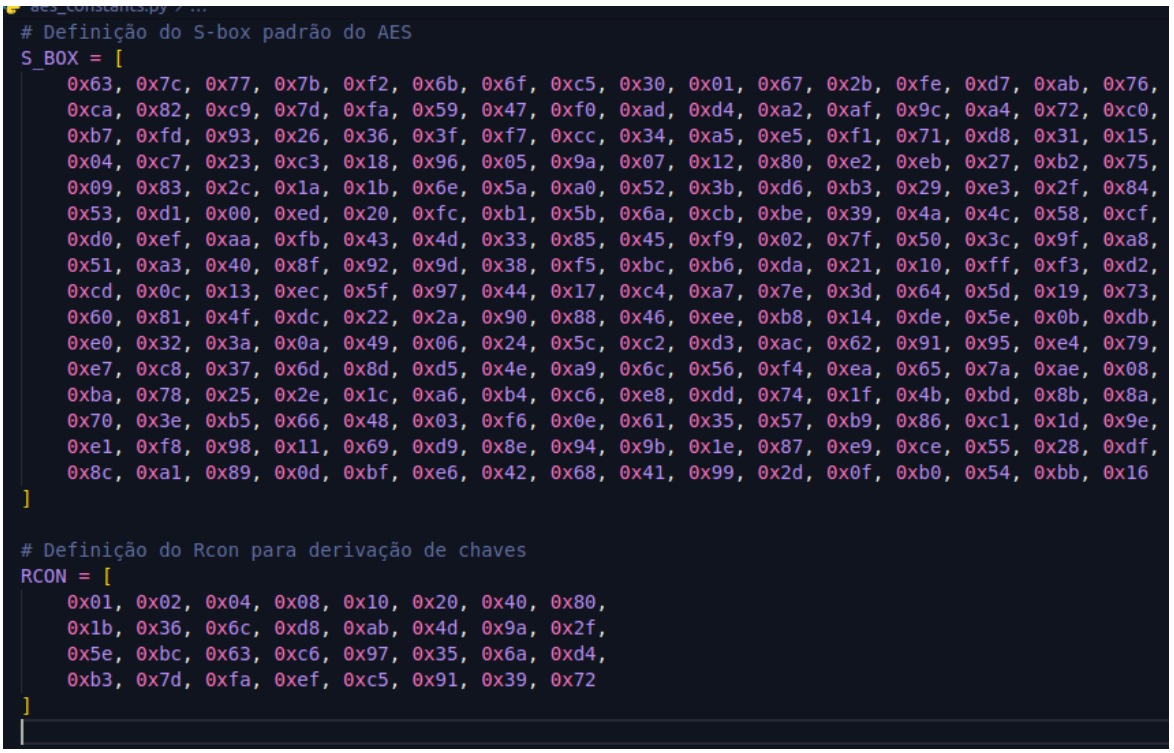
4. **MixColumns**
    - A função MixColumns é uma etapa de difusão onde cada coluna do bloco de dados é misturada usando uma transformação matricial. Este passo garante que a influência de cada byte se espalhe por todas as colunas do bloco, o que ajuda a aumentar a segurança contra ataques que tentam analisar padrões no texto cifrado. Ela faz uso de uma função auxiliar chamada de **gmul()** que tem função de realizar a multiplicação de dois números no campo finito de Galois **GF(2^8).** Ela é essencial para o cálculo das novas colunas na transformação MixColumns, esse processo garante que cada byte de um bloco de dados influencia os outros bytes, tornando a criptografia mais resistente a ataques.

```python
#Multiplicação das colunas da matriz no campo finito GF
def mixColumns(state):
    newState = [[0] * 4 for _ in range(4)]

    for c in range(4):
        newState[0][c] = gmul(0x02, state[0][c]) ^ gmul(0x03, state[1][c]) ^ state[2][c] ^ state[3][c]
        newState[1][c] = state[0][c] ^ gmul(0x02, state[1][c]) ^ gmul(0x03, state[2][c]) ^ state[3][c]
        newState[2][c] = state[0][c] ^ state[1][c] ^ gmul(0x02, state[2][c]) ^ gmul(0x03, state[3][c])
        newState[3][c] = gmul(0x03, state[0][c]) ^ state[1][c] ^ state[2][c] ^ gmul(0x02, state[3][c])

    for i in range(4):
        for j in range(4):
            state[i][j] = newState[i][j]

    return newState


#Multiplicação de dois elementos no campo finito GF
def gmul(a, b):
    p = 0

    for _ in range(8):
        if b & 1:
            p ^= a

        hi_bit_set = a & 0x80
        a <<= 1
        if hi_bit_set:
            a ^= 0x1B
        b >>= 1

    return p & 0xFF
```

5. **AddRoundKey**
    - AddRoundKey é uma operação fundamental no algoritmo AES que combina cada byte do bloco de dados com um byte correspondente de uma subchave usando uma operação de XOR. Essa etapa é essencial em cada rodada do processo de criptografia para garantir que a chave de criptografia afete o bloco de dados, contribuindo para a confidencialidade e segurança do processo de cifragem.

```python
#Aplicação das chaves de rodada para cada bloco do state
def addRoundKey(state, roundKey):
    if isinstance(roundKey, list) and all(isinstance(i, int) for i in roundKey):
        roundKey = [list(roundKey[i:i+4]) for i in range(0, len(roundKey), 4)]

    # Aplica a operação XOR em cada byte do estado
    for i in range(4):
        for j in range(4):
            state[i][j] ^= roundKey[i][j]
    return state
```

6. **S-BOX e RCON**
    - São duas matrizes de valores predefinidos usados para auxiliar etapas importantes da criptografia:



```python
# Definição do S-box padrão do AES
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
]

# Definição do Rcon para derivação de chaves
RCON = [
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
    0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
    0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4,
    0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72
]
```

7. **Counter (CTR Mode) e AES**
    - No modo CTR, um contador é usado para gerar uma série de valores exclusivos para cada bloco de dados. O contador é criptografado usando o AES para produzir uma sequência de chaves de fluxo. Cada bloco de texto claro é então “XORado” com a sequência de chaves de fluxo para produzir o texto cifrado. Uma característica importante do modo CTR é que ele permite a criptografia e a descriptografia paralelas, o que pode aumentar a eficiência.
    - Aqui, a função recebe o plaintext, a chave, o nonce para o contador e os rounds que serão aplicados as chaves. As operações auxiliares apresentadas acima são chamadas em ordem específicas para o melhor funcionamento. A criptografia e descriptografia são praticamente iguais:

```python
def encryptAES(text, key, nonce, rounds):
    plaintext = text

    counter = 0
    cipherBlocks = []
    expanded_key = keyExpansion(key, rounds)

    # Converte a chave expandida em uma lista de matrizes 4x4
    expanded_key_4x4 = [list(expanded_key[i:i+16]) for i in range(0, len(expanded_key), 16)]

    # Divide o texto em blocos de 16 bytes
    blocks = [plaintext[i:i+16] for i in range(0, len(plaintext), 16)]

    for block in blocks:
        # Cria o bloco de contador
        counter_block = nonce + counter.to_bytes(4, byteorder='big')
        state = bytes_to_state(counter_block)

        # Aplica a cifra AES
        state = addRoundKey(state, expanded_key_4x4[0])

        for round in range(1, rounds):
            state = subBytes(state)
            state = shiftRows(state)
            state = mixColumns(state)
            state = addRoundKey(state, expanded_key_4x4[round])


        state = subBytes(state)
        state = shiftRows(state)
        state = addRoundKey(state, expanded_key_4x4[rounds])

        # Criptografa o bloco de texto
        encrypted_counter = state_to_bytes(state)

        cipherBlock = bytes([b ^ c for b, c in zip(block, encrypted_counter)])
        cipherBlocks.append(cipherBlock)

        counter += 1

    ciphertext = b''.join(cipherBlocks)
```

8. **KeyGen e IVGen(Nonce)**
    - A geração de chaves é baseada na quantidade de rounds preferidas e o nonce – Valor único usado uma única vez em uma operação criptográfica para garantir que cada execução de criptografia com a mesma chave produza um resultado diferente. – é gerado aleatoriamente caso não seja informado, o mesmo se aplica a chave:

```python
def keyGen(rounds):
    if rounds == 10:
        key_size = 16
    elif rounds == 12:
        key_size = 24
    elif rounds == 14:
        key_size = 32
    else:
        print('\nTamanho de rounds inválido, deve ser[10, 12, 14]')
        return

    keyConfirm = input('\nVocê já tem uma chave pronta?[y/n]: ')

    if keyConfirm == 'y':
        key_str = input('\nInforme a chave em hexadecimal:\nR: ')
        try:
            key_bytes = bytes.fromhex(key_str)
        except ValueError:
            print("Chave inválida. Deve estar no formato hexadecimal.")
            return

        if len(key_bytes) > key_size:
            key = key_bytes[:key_size]
        elif len(key_bytes) < key_size:
            key = key_bytes.ljust(key_size, b'\0')
        else:
            key = key_bytes
    else:
        key = get_random_bytes(key_size)

    print(f'\nSua chave é: {key.hex()}')
    return key

def IVGen():
    IVConfirm = input('\nVocê já tem um nonce pronto?[y/n]: ')

    if IVConfirm == 'y':
        IV_str = input('\nInforme-o em hexadecimal:\nR: ')
        try:
            nonce = bytes.fromhex(IV_str)
        except ValueError:
            print("Nonce inválido. Deve estar no formato hexadecimal.")
            return

        if len(nonce) != 12:
            print("Nonce deve ter exatamente 12 bytes.")
            return
```

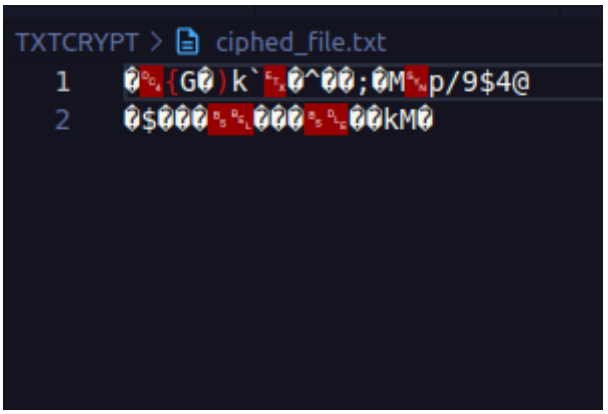
9. **Testes:**
    - Cifrar e decifrar textos foi concluído com todas as opções de rounds disponíveis:

```
Digite o texto a ser cifrado:
R: bom dia, isso é um teste

Digite a quantidade de rounds [10, 12 ou 14]: 12

Você já tem uma chave pronta?[y/n]:

Sua chave é: 9a58aec4716ca6b199c57c38d92361ea2750d0d318124cd3

Você já tem um nonce pronto?[y/n]:

Seu nonce é: 1032f96e08663501eaef2c6a

Plaintext: bom dia, isso é um teste

Selecione uma opção:
1)Encriptar
3)Sair
R: 1

Ciphertext: a28812079cbc21369cdabc68d231f780f3e61216cc4c7211bf

Selecione uma opção:
2)Decriptar
3)Sair
R: 2
Decrypt:
 bom dia, isso é um teste

Rounds: 12
PlainTxT: bom dia, isso é um teste
Encrypt: a28812079cbc21369cdabc68d231f780f3e61216cc4c7211bf
Decrypt: bom dia, isso é um teste
Nonce: 1032f96e08663501eaef2c6a
Key: 9a58aec4716ca6b199c57c38d92361ea2750d0d318124cd3

Sucesso! A fase de encriptação retornou: [ a28812079cbc21369cdabc68d231f780f3e61216cc4c7211bf ].
A decriptação retornou: [ bom dia, isso é um teste ].
Que é igual ao seu texto original: [ bom dia, isso é um teste ]
```

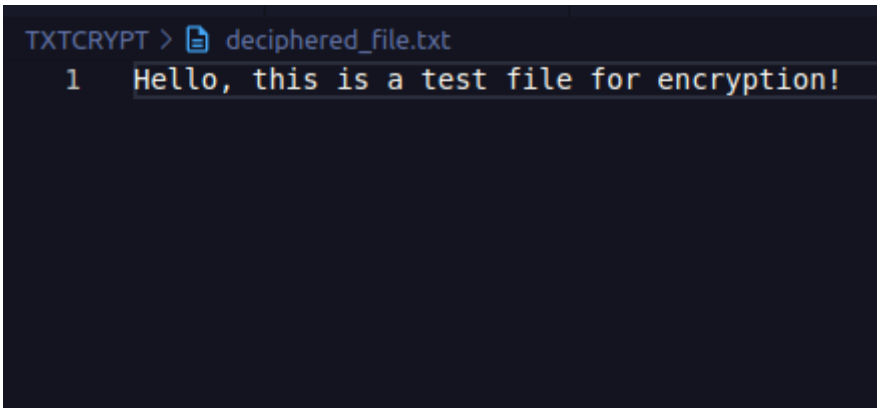
- Cifrar e decifrar arquivos foi concluído com todas as opções de rounds disponíveis:

```
Digite a quantidade de rounds [10, 12 ou 14]: 14

Você já tem uma chave pronta?[y/n]:

Sua chave é: 55be40b3a543293e22aa9a921ef77331beae0379d951e6ac98830fb23dbca09a

Você já tem um nonce pronto?[y/n]:

Seu nonce é: 96774420d99fe0309decc3fc

Selecione uma opção:
1)Cifrar Arquivo
2)Decifrar Arquivo
3)Sair
R: 1
Arquivo cifrado salvo em TXTCRYPT/

Selecione uma opção:
2)Decifrar Arquivo
3)Sair
R: 2
Arquivo decifrado salvo em TXTCRYPT/
```

Arquivo Cifrado:



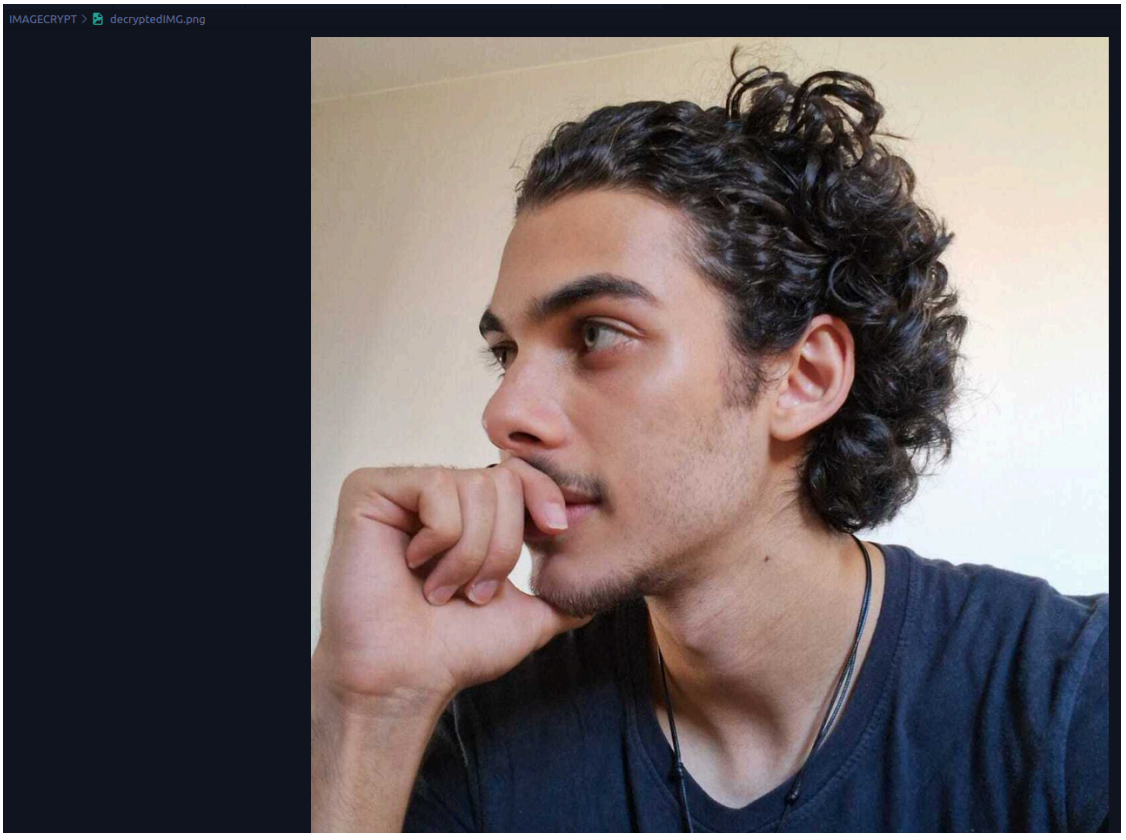
Arquivo Decifrado:



- Cifrar e decifrar imagens foi concluído com todas as opções de rounds disponíveis, **entretanto, não consegui capturar os rounds 1, 5, 9 e 13**:

Imagem criptografa em formato binário:

Imagem Decifrada:



OBS.: Os arquivos são adicionados nos diretórios TXTCRYPT e IMAGECRYPT do projeto.

10. **Não Concluído**
    - O modo de operação GCM não foi implementado neste projeto.
    - Rounds 1, 4, 9 e 13 das imagens não foram finalizados.

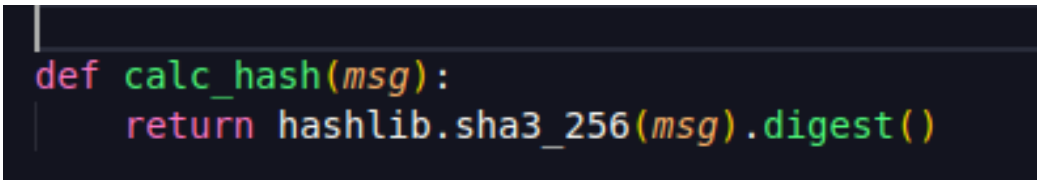
Principais Tópicos do RSA:

1. **RSA (Rivest-Shamir-Adleman)**
    - RSA é um sistema de criptografia de chave pública que usa dois pares de chaves, uma pública e uma privada. O algoritmo baseia-se na dificuldade de fatoração de grandes números primos. O processo RSA envolve a geração de uma chave pública e uma chave privada, criptografando dados com a chave pública e descriptografando com a chave privada.

```python
# Geração de um número primo grande
def largePrime(bits):
    while True:
        # Gera até encontrar um numero primo
        num = random.getrandbits(bits)
        num |= (1 << (bits-1)) | 1
        if miller_rabin(num):
            return num


# Criação da chave pública e privada
def genRSA(bits=1024):
    p = largePrime(bits)
    q = largePrime(bits)
    n = p * q
    phi = (p-1) * (q - 1)

    # Valor comum para RSA
    e = 65537

    d = mod_inverse(e, phi)

    pubKey = (e, n)
    privKey = (d, n)

    return pubKey, privKey
```

2. **SHA-3-256**

- SHA-3-256 é uma função de hash criptográfico que produz um resumo de 256 bits para qualquer entrada. Ele é projetado para ser seguro contra várias técnicas de ataque, incluindo colisões e pré-imagens. Em muitos sistemas criptográficos, SHA-3-256 é usado para gerar digests de mensagens ou para garantir a integridade dos dados.



```
def calc_hash(msg):
    return hashlib.sha3_256(msg).digest()
```

3. **OAEP (Optimal Asymmetric Encryption Padding)**
   - OAEP é um esquema de preenchimento para RSA que adiciona uma camada de segurança à criptografia RSA. Ele garante que a criptografia RSA seja semântica segura. O OAEP combina a mensagem original com um valor de preenchimento, gerando um bloco de dados mais complexo antes da criptografia. Isso ajuda a proteger contra ataques de recuperação de texto claro e outras formas de ataque.

```python
def oaep_pad(msg, n, hash_func=hashlib.sha3_256):
    l = hash_func(b'').digest()
    hlen = len(l)

    # Padding string com bytes \x00
    ps = b'\x00' * (k - len(l) - len(msg) - 2 * hlen - 2)
    db = l + ps + b'\x01' + msg

    seed = os.urandom(hlen)

    # Gerando seed
    db_mask = mgf1(seed, len(db), hash_func)
    masked_db = bytes(x ^ y for x, y in zip(db, db_mask))

    seed_mask = mgf1(masked_db, hlen, hash_func)
    masked_seed = bytes(x ^ y for x, y in zip(seed, seed_mask))

    return masked_seed + masked_db


# Remoção do padding OAEP
def oaep_unpad(padded, n, hash_func=hashlib.sha3_256):
    k = (n.bit_length() + 7) // 8
    l = hash_func(b'').digest()
    hlen = len(l)

    # Extração da seed
    masked_seed = padded[:hlen]
    masked_db = padded[hlen:]

    # Revertendo a máscara
    seed_mask = mgf1(masked_db, hlen, hash_func)
    seed = bytes(x ^ y for x, y in zip(masked_seed, seed_mask))

    db_mask = mgf1(seed, len(masked_db), hash_func)
    db = bytes(x ^ y for x, y in zip(masked_db, db_mask))

    l_hash = db[:hlen]
    if l_hash != l:
        raise ValueError('Padding inválido: hash mismatch')

    db = db[hlen:]
    pos = db.find(b'\x01')
    if pos == -1:
        raise ValueError("Padding inválido: no delimiter")

    return db[pos + 1:]
```

4. **Primalidade Miller-Rabin**

- O teste Miller-Rabin é um método probabilístico para verificar se um número é primo. Ele é utilizado na geração de chaves RSA para garantir que os números primos usados para a geração das chaves sejam realmente primos. O teste é baseado em propriedades matemáticas e fornece uma maneira eficiente de verificar a primalidade, mesmo para números muito grandes.

```python
# Baseado em https://gist.github.com/KaiSmith/5886940
def miller_rabin(n, k=100):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

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

    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        possibly_prime = False
        for _ in range(s - 1):
            x = (x * x) % n
            if x == n - 1:
                possibly_prime = True
                break
        if not possibly_prime:
            return False
    return True


# Geração de um número primo grande
def largePrime(bits):
    while True:
        # Gera até encontrar um numero primo
        num = random.getrandbits(bits)
        num |= (1 << (bits-1)) | 1
        if miller_rabin(num):
            return num
```

5. **Teste de Assinaturas válidas**

Conclusão:

RSA OAEP: RSA (Rivest-Shamir-Adleman) é um algoritmo de criptografia assimétrica que usa uma chave pública para criptografar e uma chave privada para descriptografar dados. OAEP (Optimal Asymmetric Encryption Padding) é um esquema de preenchimento que melhora a segurança do RSA, protegendo contra ataques de recuperação de texto cifrado e garantindo que o texto original seja ocultado de forma robusta.

AES-CTR: AES (Advanced Encryption Standard) é um algoritmo de criptografia simétrica amplamente utilizado para garantir a confidencialidade dos dados. No modo CTR (Counter), AES criptografa um contador e combina o resultado com o texto original usando XOR para obter o texto cifrado. Esse modo é eficiente e permite processamento paralelo, além de garantir que a mesma chave e nonce não sejam reutilizados para manter a segurança.