Trabalho
Segurança Computacional

Harisson Freitas Magalhães, 202006466
Luiz Carlos da Silva Néto Vartuli, 200023314

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0201 - Segurança Computacional

202006466@aluno.unb.br, 200023314@aluno.unb.br

Abstract. *This project explores the implementation in Python language of the AES block cipher and the CTR mode of operation, as well as the RSA signature generator and verifier. The AES algorithm supports customizable rounds for flexible encryption and decryption, and the CTR mode was developed to enhance AES encryption. Tests included encrypting and decrypting files, with outputs verified against OpenSSL. Additionally, a selfie was encrypted with different AES rounds to demonstrate the algorithm's behavior. In Part II, RSA key generation and digital signature functionalities were implemented, utilizing SHA-3 hashing and RSA encryption with OAEP padding for secure message signing and verification.*

Resumo. *Este projeto implementa na linguagem Python a cifra de bloco AES e o modo de operação CTR, além de um gerador e verificador de assinaturas RSA. O AES permite personalizar o número de rodadas para criptografia e descriptografia flexíveis, enquanto o modo CTR foi desenvolvido para melhorar a cifragem com AES. Os testes incluíram a cifragem e decifragem de arquivos, com verificação usando OpenSSL. Uma imagem também foi cifrada usando diferentes números de rodadas do AES para demonstrar o comportamento do algoritmo. Na Parte II, foram implementadas funcionalidades de geração de chaves RSA e assinatura digital, utilizando hash SHA-3 e cifragem RSA com padding OAEP para assinatura e verificação seguras de mensagens.*

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

1.1. Descrição das Cifras

AES (Advanced Encryption Standard) é uma cifra de bloco Rijndael amplamente utilizada que cifra dados em blocos de 128 bits usando chaves de 128, 192 ou 256 bits. A implementação deste projeto foca na versão de 128 bits tanto para o bloco quanto para a chave. O AES consiste em uma série de etapas: SubBytes, ShiftRows, MixColumns, e AddRoundKey. Essas operações são repetidas em várias rodadas, dependendo do tamanho da chave. No nosso caso, implementamos a funcionalidade para ajustar o número de rodadas conforme necessário, oferecendo flexibilidade para explorar diferentes números de rodadas na criptografia e descriptografia.

1.1.1. Etapa SubBytes

SubBytes é uma operação de substituição não-linear usada em cada byte do bloco de dados de entrada. Esta operação é baseada em uma tabela de substituição fixa chamada S-Box (Substitution Box). A S-Box é projetada para fornecer resistência contra ataques de análise diferencial e linear, que são métodos comuns de criptoanálise.
Cada byte do bloco de entrada é substituído por um novo byte, que é encontrado na S-Box. O valor original do byte é usado como índice para a S-Box, e o valor correspondente na S-Box substitui o byte original. A S-Box é construída de forma que a operação inversa (usada na decifragem) seja possível usando uma tabela inversa correspondente. A Figura 1 mostra como a S-Box foi implementada. A Tabela inversa segue esse princípio.

```
sbox = bytes([
    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
])
```

Figura 1. Sbox representada em formato hexadecimal.

1.1.2. Etapa ShiftRows

ShiftRows opera nas linhas da matriz de estado, realizando um deslocamento cíclico dos bytes em cada linha com base em um deslocamento específico. No AES, a primeira linha permanece inalterada, enquanto os bytes da segunda linha são deslocados uma posição para a esquerda. Da mesma forma, os bytes da terceira e quarta linhas são deslocados duas e três posições para a esquerda, respectivamente. Essa técnica garante que cada coluna da matriz de saída da etapa ShiftRows seja composta por bytes de diferentes colunas da matriz de entrada, evitando que as colunas sejam cifradas de forma independente, o que, de outra forma, faria o AES funcionar como quatro cifras de bloco separadas.

1.1.3. Etapa MixColumns

Na etapa MixColumns do AES, cada coluna da matriz de estado é transformada usando uma transformação linear invertível. Essa função combina os quatro bytes de cada coluna e, ao fazer isso, garante que cada byte de entrada influencie todos os quatro bytes de saída. Esse processo proporciona difusão, combinando com a etapa ShiftRows para aumentar a segurança.

Durante a operação MixColumns, cada coluna é multiplicada por uma matriz fixa, resultando em novos valores para a coluna. A multiplicação é realizada no campo finito $GF(2^8)$, utilizando operações de adição (XOR) e multiplicação modulares. A transformação garante que as colunas da matriz sejam misturadas de forma complexa,

aumentando a resistência da cifra contra ataques.

1.1.4. Etapa AddRoundKey

Na etapa AddRoundKey, a subchave é combinada com o estado. Para cada rodada, uma subchave é derivada da chave principal usando o planejamento de chave AES e cada subchave tem o mesmo tamanho que o estado. A subchave é adicionada combinando cada byte de estado com o byte correspondente da subchave usando o XOR bit a bit.

1.1.5. Cifragem e Decifragem AES.

- **Cifragem:** Para cifrar os dados, o AES realiza todas essas etapas que foram mostradas acima, começando com a adição de uma chave de rodada usando uma operação XOR. Em seguida, aplica uma substituição de bytes com uma tabela de substituição (S-box), desloca as linhas de bytes e mistura as colunas para aumentar a complexidade e a segurança. Depois de várias rodadas, o texto cifrado é gerado. (normalmente 10 rodadas para chaves de 128 bits). A implementação da cifragem pode ser visualizada na Figura 2

```python
def aes_encrypt(paintext, key, num_rounds):


    stages = key_expansion(key)

    stage = add_round_key(paintext, stages[0])

    for i in range(num_rounds):
        stage = sub_bytes(stage)
        stage = shift_rows(stage)
        stage = mix_columns(stage)
        stage = add_round_key(stage, stages[i+1])


    stage = sub_bytes(stage)

    stage = shift_rows(stage)

    stage = add_round_key(stage, stages[10])
    return stage
```

Figura 2. Cifragem AES em python.

- **Decifragem:**A decifragem do AES é o processo inverso, que reverte as operações aplicadas na cifragem. As operações incluem o deslocamento inverso das linhas, a substituição inversa dos bytes, a aplicação da chave de rodada com XOR e a mistura inversa das colunas. Essas operações restauram o texto claro original a partir do texto cifrado. Como pode ser visto na Figura 3.

```python
def aes_decrypt(paintext, key, num_rounds):

    stages = key_expansion(key)
    stage = add_round_key(paintext, stages[-1])
    stage = inv_shift_rows(stage)
    stage = inv_sub_bytes(stage)
    for i in range(num_rounds):
        stage = add_round_key(stage, stages[-i-2])
        stage = inv_mix_columns(stage)
        stage = inv_shift_rows(stage)
        stage = inv_sub_bytes(stage)
    stage = add_round_key(stage, stages[0])
    return stage
```

Figura 3. Decifragem AES em python.

1.2. Modo de Operação

O modo de operação CTR transforma uma cifra de bloco em uma cifra de fluxo. Em vez de simplesmente cifrar blocos de dados diretamente, o modo CTR utiliza um contador que é cifrado e combinado com os dados para produzir o texto cifrado. Cada bloco de texto cifrado é gerado cifrando um valor de contador e depois realizando uma operação XOR com o bloco de texto plano correspondente. O modo CTR permite a paralelização dos processos de cifragem e decifragem, sendo eficiente e seguro para operações em que a integridade e a confidencialidade são essenciais.

1.2.1. Cifragem e Decifragem no Modo CTR

O CTR (Counter) transforma o AES em uma cifra de fluxo, permitindo a cifragem e decifragem rápida e em paralelo. No modo CTR, um contador é combinado com um nonce fixo para garantir a unicidade de cada bloco. O contador é cifrado com o AES para gerar um fluxo de chave (keystream), que é combinado com o texto claro usando XOR para produzir o texto cifrado.

A decifragem no modo CTR é idêntica à cifragem. Usando o mesmo nonce e contador, o fluxo de chave é regenerado e combinado com o texto cifrado usando XOR, resultando na recuperação do texto claro original. O modo CTR é eficiente e permite operações paralelas, sendo ideal para aplicações que requerem alta performance. Essa função pode ser visualizada na Figura 4

```python
def aes_ctr_encrypt(plaintext: bytes, key: bytes, nonce: bytes, num_rounds: int) -> bytes:
    ciphertext = bytearray()
    counter = nonce

    for i in range(0, len(plaintext), 16):
        # Encripta o contador usando AES
        keystream = aes_ecb_encrypt(counter, key, num_rounds)
        block = plaintext[i:i+16]

        # Aplica XOR entre o bloco de texto plano e o keystream
        encrypted_block = bytes([b ^ k for b, k in zip(block, keystream)])
        ciphertext.extend(encrypted_block)


        counter = increment_counter(counter)

    return bytes(ciphertext)
```

Figura 4. Cifragem e decifragem do AES modo CTR em python.

2. Parte II - Gerador/Verificador de Assinaturas

Na Parte II, foi implementado um sistema de geração e verificação de assinaturas digitais usando o algoritmo RSA. Este sistema assegura a autenticidade e integridade dos arquivos ao criar uma assinatura digital que pode ser verificada posteriormente.

2.1. Descrição do RSA e OAEP

O RSA é um algoritmo de criptografia assimétrica que utiliza um par de chaves, uma pública e uma privada. A chave pública é usada para cifrar dados, enquanto a chave privada é utilizada para decifrar. Na nossa implementação, o RSA é usado tanto para cifração/decifração de dados como para a assinatura digital.

As chaves RSA são geradas usando dois números primos grandes, p e q, para calcular o módulo n e a chave pública e. A chave privada é derivada usando o inverso modular de e em relação a $(p-1)(q-1)$. A implementação da geração dessa chave pode ser vista fa Figura 5.

```python
def generateKey():
    #garante que p e q primos e p != q
    p = primeGenerator(1024)
    q = primeGenerator(1024)
    while(p == q):
        q = primeGenerator(1024)
    #gera chave do RSA, utilizando inverso modular
    n = p * q
    n2 = (p-1) * (q-1)
    e = random.randrange(2, n2)
    while(math.gcd(e, n2) != 1):
        e = random.randrange(2, n2)
    d = modularInverse(e, n2)
    priv_key = (d, n)
    publ_key = (e, n)
    return priv_key, publ_key
```

Figura 5. Geração de chaves RSA.

A assinatura digital com RSA é realizada ao criar um hash da mensagem utilizando a função de hash SHA-3 (SHA-3-256 neste caso). Esse hash é cifrado com a chave privada do emissor, formando a assinatura digital. Para verificar a assinatura, o receptor decifra o hash com a chave pública do emissor e o compara com um hash gerado da mensagem recebida.

Figura 6. Assinatura RSA.

Para aumentar a segurança do RSA, o esquema de preenchimento OAEP (Optimal Asymmetric Encryption Padding) é utilizado. OAEP adiciona redundância ao texto claro antes da cifragem RSA, tornando o texto resultante mais resistente a ataques. Na implementação, a função oaepEncr realiza a cifragem de uma mensagem com RSA usando OAEP, enquanto oaepDecr reverte o processo.

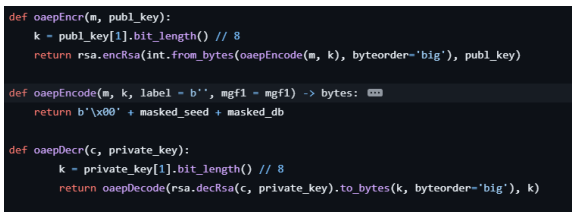

Figura 7. oaepEncr e oaepDecr.

2.2. Resultados

Em ambas as partes 1 e 2, tivemos os resultados esperado, de acorodo com a teoria vista em sala e em outros materiais de estudo. Tanto a implementação do AES quanto do seu modo de operação CTR, foram testados e comparados com o resultado obtido no Openssl, no qual tivemos os mesmos dados. O mesmo vale para o RSA juntamente com o OAEP

Sobre os extras, fizemos apenas a parte da selfie, que está nos arquivos, mas está com erro para 13 rodadas. O outro, que era a implementação do modo de cifração autenticada GCM - contador de Galois, não chegamos a iniciá-lo.

Para dar credibilidade aos autores e ajudar na aprendizagem de todos,o repositório completo com todos os arquivos que foram implementado para esse trabalho se encontra neste link

3. Conclusão

Neste trabalho, implementamos duas importantes técnicas de criptografia: o Advanced Encryption Standard (AES) e o RSA, com o esquema de preenchimento OAEP. A primeira parte focou na implementação do AES e do modo de operação CTR, oferecendo uma análise prática de cifragem e decifragem usando este algoritmo. O AES foi implementado com suporte para um número variável de rodadas, e o modo CTR foi aplicado para transformar o texto cifrado, garantindo segurança adicional.

Na segunda parte, abordamos a criptografia assimétrica com RSA e o preenchimento OAEP. A implementação de RSA permitiu a geração de chaves, cifragem e decifragem de mensagens, bem como a criação e verificação de assinaturas digitais. O uso de OAEP ajudou a melhorar a segurança da cifragem RSA, oferecendo uma proteção adicional contra ataques de criptografia.

Os testes realizados, incluindo a cifragem e decifragem de arquivos e a verificação de assinaturas digitais, confirmaram a eficácia das implementações. A comparação com ferramentas padrão como OpenSSL também validou a corretude das nossas abordagens.

Com isso, o trabalho não só demonstrou a aplicação prática de conceitos avançados de criptografia, mas também proporcionou uma base sólida para futuras explorações em segurança digital e criptografia.

Referências

Aulas de Segurança Computacional com o Prof. João Gondim e o Monitor Paulo Lopes. (2024). *Notas e Materiais de Aula*. Universidade de Brasília.

ANDERSON, R., Security Engineering, 2a ed., J. Willey, 2008.

KATZ, Jonathan; LINDELL, Yehuda. Introduction to modern cryptography. CRC press, 2014.