

TRABALHO DE SEGURANÇA COMPUTACIONAL

Autor: Gustavo Mello Tonnera

Professor: João José Costa Gondim

Disciplina: Segurança Computacional

Semestre: 2024.1

1. INTRODUÇÃO

Segurança computacional é o conjunto de práticas, técnicas e tecnologias utilizadas para proteger sistemas de informação contra acessos não autorizados, falhas, ataques cibernéticos e outras ameaças. Seu objetivo principal é garantir a confidencialidade, integridade e disponibilidade dos dados, bem como a autenticidade e não-repúdio das comunicações digitais. Para alcançar esses objetivos, a segurança computacional emprega uma variedade de métodos, como criptografia, controle de acesso, firewalls, detecção de intrusões e políticas de segurança, que juntos formam uma defesa robusta contra vulnerabilidades e riscos no ambiente digital.

A disciplina Segurança Computacional ofertada pelo professor João Gondim busca introduzir os estudantes de computação da UnB (Universidade de Brasília) aos conceitos dessa área. Durante as aulas, o professor explicou diversos conceitos relacionados a criptografia, como segredo perfeito, cifras de bloco, modos de operação e autenticação de mensagens. Como método avaliativo, o professor propôs um trabalho, o qual consiste na implementação da cifra de bloco AES no modo CTR (Counter) e na implementação de um gerador/verificador de assinaturas digitais RSA.

O trabalho desenvolvido pelo estudante Gustavo M. Tonnera foi desenvolvido utilizando a linguagem de programação Python e os resultados obtidos estão presentes nas próximas seções.

2. AES

A cifra de bloco AES (Advanced Encryption Standard) é um algoritmo de criptografia simétrica que opera sobre blocos de dados de 128 bits, utilizando chaves de 128, 192 ou 256 bits para realizar a criptografia. O AES transforma o texto claro em texto cifrado por meio de uma série de operações matemáticas, divididas em rodadas, cujo número varia conforme o tamanho da chave (10 rodadas para 128 bits, 12 para 192 bits e 14 para 256 bits). Cada rodada inclui substituições de bytes (SubBytes), embaralhamento de linhas (ShiftRows), mistura de colunas (MixColumns), e a adição de uma chave de rodada derivada da chave original (AddRoundKey). O processo é reversível, permitindo que o texto cifrado seja transformado de volta em texto claro durante a descryptografia, usando a mesma chave simétrica.

O modo de operação CTR (Counter) é uma técnica utilizada em cifras de bloco que transforma o algoritmo em uma cifra de fluxo, permitindo a criptografia de dados de

qualquer tamanho. Em vez de operar diretamente sobre blocos de dados, o modo CTR gera um fluxo de chave (keystream) ao encriptar um contador único e incrementado para cada bloco. Esse contador é combinado com um vetor de inicialização (IV) e, em seguida, criptografado usando a cifra de bloco, como o AES. O resultado é então combinado com o bloco de dados original através de uma operação XOR, produzindo o texto cifrado. O modo CTR permite paralelismo, pois cada bloco pode ser criptografado independentemente, e também facilita a descryptografia, seguindo o mesmo processo. Além disso, o CTR oferece flexibilidade para criptografar dados de tamanhos variáveis e é resistente a certos tipos de ataques, desde que o contador nunca seja reutilizado com a mesma chave.

2.1 IMPLEMENTAÇÃO DO MODO ECB

Para implementar o AES, foi criada uma classe chamada “AES”. Essa classe possui os métodos “encryptECBMode()” e “decryptECBMode()”, os quais cifram e decifram um arquivo utilizando AES no modo de operação ECB, respectivamente. Esses métodos foram desenvolvidos para verificar se os algoritmos de cifração e decifração estão funcionando corretamente.

```
def encryptECBMode(self, filename, key, rounds=10):
    encryptedData = b""
    # Extrair bytes do arquivo
    data = self.readTextFile(filename)
    newFilename = filename[:filename.index('.')] + '-AES_ECB_encrypted.txt'

    # Realizar a expansao da chave
    expandedKey = self.keyExpansion(key, rounds)

    # Criptografar
    for i in range(0, len(data), 16):
        state = data[i:i+16]
        if len(state) < 16:
            for _ in range(16-len(state)):
                state += b"\x00"

        encryptedData += self.encrypt(state, expandedKey, rounds)

    self.genEncryptedFile(newFilename, encryptedData)
```

Figura 1- Método encryptRCBMode()

```

def decryptECBMode(self, filename, key, rounds=10):
    decryptedData = b""

    # Extrair bytes do arquivo
    newFilename = filename[:filename.index('.')] + '-AES_ECB_decrypted.txt'
    data = self.readTextFile(filename)

    # Realizar a expansao da chave
    expandedKey = self.keyExpansion(key, rounds)
    # Descriptografar
    for i in range(0, len(data), 16):
        state = data[i:i+16]
        if len(state) < 16:
            for _ in range(16-len(state)):
                state += b"\x00"

        decryptedData += self.decrypt(state, expandedKey, rounds)

    self.genDecryptedFile(newFilename, decryptedData)

```

Figura 2 - Método decryptECBMode()

Observa-se que esses métodos são usados, basicamente, para extrair os dados do arquivo e separar esses dados em blocos de 128 bits, os quais são passados como argumento das funções “encrypt()” e “decrypt()”. Esses métodos realizam, respectivamente, a cifração e a decifração do bloco passado como argumento. Além do bloco, a chave, após passar pelo processo de expansão de chave, e número de rodadas também são passados como argumentos desses métodos.

O processo de expansão de chaves é feito pelo método “keyExpansion()”, o qual recebe a chave e o número de rodadas como argumentos. O processo de expansão de chave no AES gera uma série de chaves de rodada a partir da chave original. Isso é feito dividindo a chave em palavras de 32 bits e, em cada etapa, aplicando uma rotação, substituição de bytes via S-box, e combinando com uma constante de rodada (Rcon) usando XOR. Esse processo continua até que todas as chaves necessárias sejam criadas, uma para cada rodada.

```

def keyExpansion(self, key, rounds):
    expandedKey = b""

    # Copiar a chave original
    for i in range(0, len(key), 4):
        expandedKey += key[i:i+4]

    # Expandir palavras restantes
    for i in range(4, 4*(rounds + 1)):
        temp = expandedKey[(i-1)*4:i*4]
        if i % 4 == 0:
            # Rotação da Word
            temp = temp[1:] + temp[:1]
            # Substituição da Word pelos valores da SBox
            temp = bytes([self.sbox[b] for b in temp])
            # XOR
            temp = bytes(temp[0] ^ self.rcon[i // 4 - 1]) + temp[1:]

        expandedKey += bytes([expandedKey[(i-4)*4+j] ^ temp[j] for j in range(4)])

    return expandedKey

```

Figura 3 - Método keyExpansion()

Os métodos “encrypt()” e “decrypt” implementam as rodadas do AES. As rodadas do AES são as etapas repetitivas em que o algoritmo transforma o texto claro em texto cifrado. Cada rodada consiste em quatro operações principais: **SubBytes**, onde cada byte do bloco é substituído por outro usando uma S-box pré-definida; **ShiftRows**, que realiza uma rotação das linhas da matriz de estado, deslocando os bytes para a esquerda; **MixColumns**, onde as colunas da matriz de estado são misturadas usando transformações matemáticas para aumentar a difusão dos dados; e **AddRoundKey**, que aplica uma operação XOR entre a matriz de estado e a chave de rodada expandida. A primeira e última rodada são ligeiramente diferentes, com a última omitindo a etapa MixColumns.

```

def encrypt(self, data, expandedKey, rounds):
    # Primeira rodada
    state = self.addRoundKey(data, expandedKey[:16])

    # Rodadas intermediárias
    for j in range(1, rounds):
        state = self.middleRound(state, expandedKey[j*16:(j+1)*16])

    # Rodada Final
    state = self.finalRound(state, expandedKey[-16:])

    return state

```

Figura 4 - Método encrypt()

```
def decrypt(self, data, expandedKey, rounds):
    # Primeira rodada
    state = self.addRoundKey(data, expandedKey[-16:])

    # Rodadas intermediárias
    for j in range(1, rounds):
        state = self.inverseMiddleRound(state, expandedKey[len(expandedKey)-(j+1)*16:len(expandedKey)-j*16])

    # Rodada Final
    state = self.inverseFinalRound(state, expandedKey[:16])

    return state
```

Figura 5 - Método *decrypt()*

O método “middleRound()” realiza uma rodada intermediária e o método “finalRound()” realiza a rodada final do processo de criptografia. O método “inverseMiddleRound()” realiza uma rodada intermediária e o método “inverseFinalRound()” realiza a rodada final do processo de decifração. Esses quatro métodos utilizam os métodos “addRoundKey()”, “mixColumns()” e “inverseMixColumns()” para implementar as operações AddRoundKey e MixColumns da criptografia e da decifração.

```
def middleRound(self, state, roundKey):
    newState = b""

    # SubBytes
    newState = bytes([self.sbox[b] for b in state])

    # ShiftRows
    newState = newState[0:4] + newState[5:8] + int.to_bytes(newState[4], 4) + newState[10:12] + newState[8:10] + int.to_bytes(newState[15], 4) + newState[12:15]

    # MixColumns
    newColumns = b""
    for i in range(4):
        newColumns += self.mixColumns(bytes([newState[j*4 + i] for j in range(4)]))

    newState = int.to_bytes(newColumns[0], 4) + int.to_bytes(newColumns[4], 4) + int.to_bytes(newColumns[8], 4) + int.to_bytes(newColumns[12], 4) + int.to_bytes(newColumns[16], 4)

    # AddRoundKey
    newState = self.addRoundKey(newState, roundKey)

    return newState
```

Figura 6 - Método *middleRound()*

```
def finalRound(self, state, roundKey):
    newState = b""

    # SubBytes
    newState = bytes([self.sbox[b] for b in state])

    # ShiftRows
    newState = newState[0:4] + newState[5:8] + int.to_bytes(newState[4], 4) + newState[10:12] + newState[8:10] + int.to_bytes(newState[15], 4) + newState[12:15]

    # AddRoundKey
    newState = self.addRoundKey(newState, roundKey)

    return newState
```

Figura 7 - Método *finalRound()*

```
def inverseMiddleRound(self, state, roundKey):
    newState = b""

    # InverseShiftRows
    newState = state[0:4] + int.to_bytes(state[7], 4) + state[4:7] + state[10:12] + state[8:10] + state[13:16] + int.to_bytes(state[12], 4)

    # InverseSubBytes
    newState = bytes([self.inverseSbox[b] for b in newState])

    # AddRoundKey
    newState = self.addRoundKey(newState, roundKey)

    # InverseMixColumns
    newColumns = b""
    for i in range(4):
        newColumns += self.inverseMixColumns(bytes([newState[j*4 + i] for j in range(4)]))

    newState = int.to_bytes(newColumns[0], 4) + int.to_bytes(newColumns[4], 4) + int.to_bytes(newColumns[8], 4) + int.to_bytes(newColumns[12], 4) + int.to_bytes(newColumns[16], 4)

    return newState
```

Figura 8 - Método *inverseMiddleRound()*

```
def inverseFinalRound(self, state, roundKey):
    newState = b""

    # InverseShiftRows
    newState = state[0:4] + int.to_bytes(state[7]) + state[4:7] + state[10:12] + state[8:10] + state[13:16] + int.to_bytes(state[12])

    # InverseSubBytes
    newState = bytes([self.inverseSBox[b] for b in newState])

    # AddRoundKey
    newState = self.addRoundKey(newState, roundKey)

    return newState
```

Figura 9 - Método `inverseFinalRound()`

```
def addRoundKey(self, state, roundKey):
    return bytes([state[i] ^ roundKey[i] for i in range(len(state))])
```

Figura 10 - Método `addRoundKey()`

```
def mixColumns(self, column):
    newColumn = b""

    newColumn += int.to_bytes(self.tableMult2[column[0]] ^ self.tableMult3[column[1]] ^ column[2] ^ column[3])
    newColumn += int.to_bytes(column[0] ^ self.tableMult2[column[1]] ^ self.tableMult3[column[2]] ^ column[3])
    newColumn += int.to_bytes(column[0] ^ column[1] ^ self.tableMult2[column[2]] ^ self.tableMult3[column[3]])
    newColumn += int.to_bytes(self.tableMult3[column[0]] ^ column[1] ^ column[2] ^ self.tableMult2[column[3]])

    return newColumn
```

Figura 11 - Método `mixColumns`

```
def inverseMixColumns(self, column):
    newColumn = b""

    newColumn += int.to_bytes(self.tableMult14[column[0]] ^ self.tableMult11[column[1]] ^ self.tableMult13[column[2]] ^ self.tableMult9[column[3]])
    newColumn += int.to_bytes(self.tableMult9[column[0]] ^ self.tableMult14[column[1]] ^ self.tableMult11[column[2]] ^ self.tableMult13[column[3]])
    newColumn += int.to_bytes(self.tableMult13[column[0]] ^ self.tableMult9[column[1]] ^ self.tableMult14[column[2]] ^ self.tableMult11[column[3]])
    newColumn += int.to_bytes(self.tableMult11[column[0]] ^ self.tableMult13[column[1]] ^ self.tableMult9[column[2]] ^ self.tableMult14[column[3]])

    return newColumn
```

Figura 12 - Método `inverseMixColumns()`

Para demonstrar o funcionamento do AES no modo ECB, foi criptografado o arquivo “teste.txt”, o qual contém o conteúdo ilustrado na figura 13, usando a chave presente na figura 14.

Em um tranquilo e pitoresco vilarejo, as árvores balançavam suavemente ao vento, enquanto o sol brilhava intensamente no céu azul. As crianças corriam pelo campo, rindo e brincando, e o som dos pássaros cantando preenchia o ar. O aroma das flores silvestres misturava-se com o cheiro do pão fresco saindo do forno da padaria local. As casas, com suas fachadas coloridas, pareciam contar histórias de tempos antigos e aventuras passadas. No centro da praça, um grupo de pessoas se reunia em torno de um pequeno quiosque, onde um artista local apresentava suas obras. Cada pincelada parecia capturar a essência do vilarejo e das pessoas que ali viviam. À medida que o dia avançava, a luz dourada do entardecer envolvia o lugar em uma aura mágica, e todos sabiam que estavam vivendo um momento especial, repleto de paz e alegria.

Figura 13 - Arquivo teste.txt

```
aesKey = bytes([0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0xf2, 0xfa, 0xc5, 0xe6, 0x7c, 0x82])
```

Figura 14 - Chave usada para criptografar o arquivo teste.txt

Ao utilizar o método “`encryptECBMode()`”, obtemos o arquivo “test-AES_ECB_encrypted.txt”, o qual contém o conteúdo do arquivo “teste.txt” cifrado ilustrado abaixo.

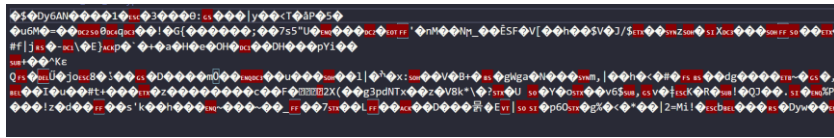


Figura 15 - Arquivo test-AES_ECB_encrypted.txt

Ao utilizar o método “decryptECBMode()”, obtemos o arquivo “test-AES_ECB_encrypted-AES_ECB_decrypted.txt”, o qual contém o conteúdo do arquivo “test-AES_ECB_encrypted.txt” decifrado ilustrado abaixo.

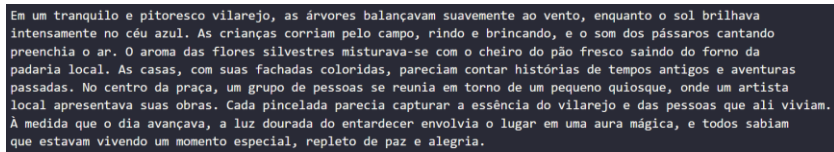


Figura 16 - Arquivo test-AES_ECB_encrypted-AES_ECB_decrypted.txt

Pode-se observar que o algoritmo está funcionando, uma vez que o conteúdo do arquivo “teste.txt” e “test-AES_ECB_encrypted-AES_ECB_decrypted.txt” são iguais.

2.2 IMPLEMENTAÇÃO DO MODO CTR

Para utilizar o modo CTR do AES, basta utilizar os métodos “encryptCTRMode()” e “decryptCTRMode()” para cifrar e decifrar um documento. As implementações desses métodos estão ilustradas nas figuras 17 e 18.

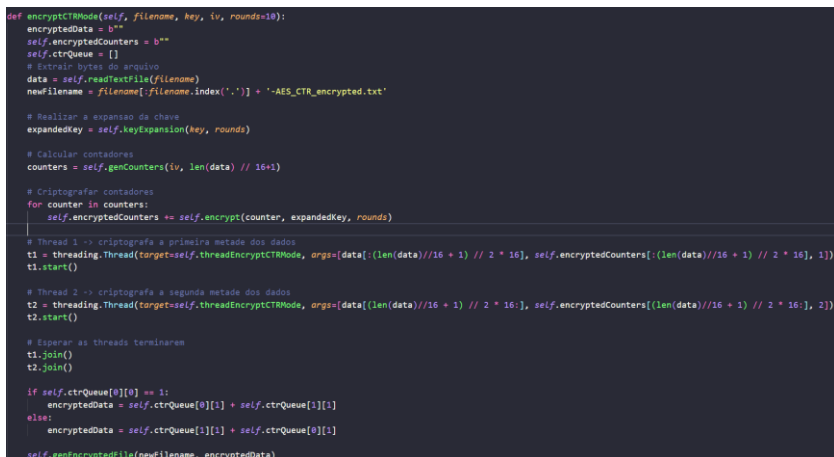


Figura 17 - Método encryptCTRMode()


```

def decryptCTRMMode(self, filename, key, iv, rounds=10):
    decryptedData = b""
    self.encryptedCounters = b""
    self.ctrQueue = []
    # Extrair bytes do arquivo
    newFilename = filename[filename.index('.'):] + '-AES_CTR_decrypted.txt'
    data = self.readTestFile(filename)

    # Realizar a expansao da chave
    expandedKey = self.keyExpansion(key, rounds)
    # Criar contadores
    counters = self.genCounters(iv, len(data) // 16)

    # Criptografar contadores
    for counter in counters:
        self.encryptedCounters += self.encrypt(counter, expandedKey, rounds)

    # Thread 1 -> criptografa a primeira metade dos dados
    t1 = threading.Thread(target=self.threadEncryptCTRMMode, args=[data[:len(data)//16 + 1] // 2 * 16, self.encryptedCounters[:len(data)//16 + 1] // 2 * 16, 1])
    t1.start()

    # Thread 2 -> criptografa a segunda metade dos dados
    t2 = threading.Thread(target=self.threadEncryptCTRMMode, args=[data[len(data)//16 + 1] // 2 * 16:, self.encryptedCounters[len(data)//16 + 1] // 2 * 16:, 2])
    t2.start()

    # Esperar as threads terminarem
    t1.join()
    t2.join()

    if self.ctrQueue[0][0] == 1:
        decryptedData = self.ctrQueue[0][1] + self.ctrQueue[1][1]
    else:
        decryptedData = self.ctrQueue[1][1] + self.ctrQueue[0][1]

    self.genDecryptedFile(newFilename, decryptedData)

```

Figura 18 - Método *decryptCTRMMode()*

Observa-se que esses métodos calculam os blocos de dados formados pelo contador utilizando o método “genCounters()”, os quais são cifrados ou decifrados com chave especificada por meio dos métodos “encrypt()” e “decrypt()”. Depois disso, duas threads são criadas para realizar a operação de XOR com o resultado da cifração/decifração com os dados do arquivo especificado. Essas threads são criadas para demonstrar um das principais vantagens do modo CTR: o paralelismo. As threads executam a função ilustrada na figura 19.

```

def threadEncryptCTRMMode(self, data, encryptedCounters, id):
    encryptedData = b""

    for i in range(0, len(data), 16):
        state = data[i:i+16]
        if len(state) < 16:
            for _ in range(16-len(state)):
                state += b"\x00"
        encryptedData += bytes([state[j] ^ encryptedCounters[i+j] for j in range(len(state))])

    with self.ctrQueueLock:
        self.ctrQueue.append((id, encryptedData))

```

Figura 19 - Método *threadEncryptCTRMMode()*

```
def genCounters(self, iv, numCounters):
    # Lista com os contadores
    lisCounters = []
    # Counter
    counter = int.from_bytes(bytes(iv))

    for _ in range(numCounters):
        lisCounters.append(counter.to_bytes(16))
        counter += 1

    return lisCounters
```

Figura 20 - Método genCounters()

Para testar o funcionamento do modo CTR, o arquivo “teste.txt” foi cifrado com a mesma chave usada no modo ECB e com o vetor inicial presente na figura 21, por meio do método “encryptCTRMode()”. O resultado obtido foi o arquivo “test-AES_CTR_encrypted.txt”, cujo conteúdo está ilustrado na figura 22.

```
aesIV = [0xc4, 0xd9, 0xb8, 0xab, 0xf2, 0x9f, 0xa3, 0xfc, 0x77, 0x9b, 0x2a, 0xb4, 0xa3, 0x2c, 0x5f, 0xfa]
```

Figura 21- Vetor inicial



Figura 22 - Arquivo test-AES_CTR_encrypted.txt

Utilizando o método “decryptCTRMode()” para decifrar o arquivo “test-AES_CTR_encrypted.txt”, obtemos o arquivo “test-AES_CTR_encrypted-AES_CTR_decrypted.txt”, cujo conteúdo é idêntico ao conteúdo do arquivo “teste.txt”, o que significa que o algoritmo foi corretamente implementado.

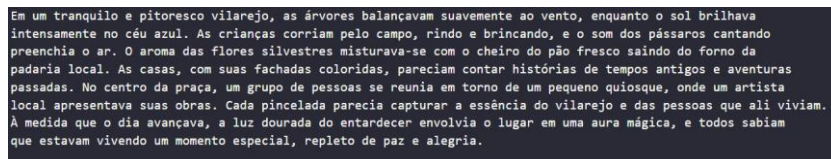


Figura 23 - Arquivo test-AES_CTR_encrypted-AES_CTR_decrypted.txt

3. RSA

O RSA é um algoritmo de criptografia assimétrica que se baseia na dificuldade de fatorar grandes números primos. Ele funciona gerando um par de chaves: uma pública, que pode ser compartilhada livremente, e uma privada, que deve ser mantida em segredo. O processo começa com a escolha de dois números primos grandes, cuja multiplicação resulta em um número chamado módulo. A chave pública é composta por esse módulo e um expoente público, enquanto a chave privada usa o mesmo módulo e um expoente privado, derivado do expoente público e dos primos escolhidos. Para criptografar uma mensagem, o emissor usa a chave pública do destinatário, elevando a mensagem ao expoente público e tomando o módulo do resultado. Para descriptografar, o destinatário aplica a chave privada, revertendo a operação. Além de criptografia, o RSA é amplamente usado para gerar e verificar assinaturas digitais, garantindo a autenticidade e integridade das mensagens. A segurança do RSA depende do tamanho das chaves, sendo altamente resistente a ataques, desde que as chaves sejam suficientemente longas.

3.1 IMPLEMENTAÇÃO DO GERADOR DE CHAVES

O processo de geração das chaves pública e privada do RSA começa com a escolha de dois números primos grandes, p e q . A multiplicação desses primos resulta em um número n , que será usado como o módulo para ambas as chaves. Em seguida, calcula-se o valor de $\phi(n)$, que é o produto de $(p-1)$ e $(q-1)$. O próximo passo é escolher um número e , que será o expoente público, tal que e seja relativamente primo a $\phi(n)$. O par (e, n) forma a chave pública. Para gerar a chave privada, calcula-se o valor de d , que é o inverso multiplicativo de e módulo $\phi(n)$. O par (d, n) constitui a chave privada. Esse processo garante que a chave pública possa ser usada para criptografar dados ou verificar assinaturas, enquanto a chave privada é necessária para descriptografar os dados ou gerar assinaturas.

Para implementar a geração de chaves do RSA, foi implementada a classe “RSAKeyGenerator”, a qual possui o método “genKeys()”, o qual gera um par de chaves (uma chave pública e uma chave privada) e salva essas chaves no arquivo “keys.json”.

```

def genKeys(self):
    # numeros primos de 512 bits
    p = self.genStrongPrimeNumber()
    q = self.genStrongPrimeNumber()
    # n = pq
    n = p*q
    # funcao totiente de euler em n:  $\phi(n) = (p-1)(q-1)$ 
    phi_n = (p-1)*(q-1)
    # calcular um numero relativamente primo a  $\phi(n)$  tal que  $1 < e < \phi(n)$ 
    e = 65537

    gcd, d, _ = self.inverse(e, phi_n)

    data = {
        "publicKey": {
            "n": n,
            "e": e
        },
        "privateKey": {
            "n": n,
            "d": d
        }
    }

    with open("src/files/keys.json", 'w', encoding='utf-8') as file:
        json.dump(data, file)

```

Figura 24 - Método `genKeys()`

O método “`genKeys()`” chama o método “`genStringPrimeNumber()`” o qual retorna um número fortemente primo verificado pelo algoritmo de Miller Rabin.

```

def genStrongPrimeNumber(self):
    while True:
        rand_num = self.getRandomNumber512Bits()
        if rand_num % 2 == 0:
            continue
        if self.millerRabin(rand_num, 50):
            return rand_num

```

Figura 25 - Método `genStrongPrimeNumber()`

3.2 IMPLEMENTAÇÃO DO RSA COM OAEP

O OAEP (Optimal Asymmetric Encryption Padding) é um esquema de preenchimento (padding) usado em conjunto com o RSA para aumentar a segurança da criptografia. O principal objetivo do OAEP é evitar ataques determinísticos e ataques de texto simples escolhidos, que são possíveis quando o RSA é utilizado sem um esquema de preenchimento adequado. O OAEP adiciona redundância à mensagem original e a embaralha antes de ser criptografada. Isso é feito em duas etapas principais: primeiro, a mensagem é concatenada com um bloco de preenchimento e, em seguida, esse resultado

é misturado usando duas funções de hash e uma máscara gerada aleatoriamente. Esse processo garante que cada vez que a mesma mensagem é criptografada, o texto cifrado resultante será diferente, tornando muito mais difícil para um atacante inferir informações sobre a mensagem original ou realizar ataques baseados em repetição.

Para implementar o RSA com OAEP, foi criada uma classe chamada “RSA_OAEP” com métodos para cifrar e decifrar um arquivo de texto, gerar assinaturas, assinar documentos de texto e verificar assinaturas. Os métodos “OAEPEncode()” e “OAEPDecode()” realizam a codificação e decodificação do OAEP, respectivamente.

```
def OAEPEncode(self, data):
    PS = b"\x00" * (self.mLenDefault-len(data))

    DB = self.lHash + PS + b"\x01" + data
    seed = os.urandom(self.hLen)

    dbMask = self.mgf1(seed, self.k - self.hLen - 1)
    maskedDB = bytes([DB[i] ^ dbMask[i] for i in range(len(DB))])

    seedMask = self.mgf1(maskedDB, self.hLen)

    maskedSeed = bytes([seed[i] ^ seedMask[i] for i in range(len(seed))])

    EM = b"\x00" + maskedSeed + maskedDB

    return EM
```

Figura 26 - Método OAEPEncode()

```
def OAEPDecode(self, data):
    PS = b"\x00" * (self.mLenDefault-len(data))

    DB = self.lHash + PS + b"\x01" + data
    seed = os.urandom(self.hLen)

    dbMask = self.mgf1(seed, self.k - self.hLen - 1)
    maskedDB = bytes([DB[i] ^ dbMask[i] for i in range(len(DB))])

    seedMask = self.mgf1(maskedDB, self.hLen)

    maskedSeed = bytes([seed[i] ^ seedMask[i] for i in range(len(seed))])

    EM = b"\x00" + maskedSeed + maskedDB

    return EM
```

Figura 27 - Método OAEPDecode()

Os métodos “encrypt()” e “decrypt()” realizam, respectivamente, a cifração e decifração do RSA.

```
def encrypt(self, data, key):
    encrypted = pow(int.from_bytes(data), key[1], key[0])
    return encrypted.to_bytes((encrypted.bit_length()+ 7) // 8)

def decrypt(self, data, key):
    decrypted = pow(int.from_bytes(data), key[1], key[0])
    return decrypted.to_bytes((decrypted.bit_length()+ 7) // 8)
```

Figura 28 - Métodos encrypt() e decrypt()

3.3 IMPLEMENTAÇÃO DO GERADOR DE ASSINATURAS DIGITAIS

Assinaturas digitais são mecanismos criptográficos que garantem a autenticidade, integridade e não repúdio de um documento ou mensagem eletrônica. Elas funcionam através da geração de um hash (uma espécie de "impressão digital" do conteúdo) que é criptografado com a chave privada do remetente. Quando o destinatário recebe o documento, ele pode utilizar a chave pública correspondente para verificar a assinatura, garantindo que o conteúdo não foi alterado e que foi realmente assinado pelo titular da chave privada. Assim, as assinaturas digitais são amplamente utilizadas em transações eletrônicas, contratos digitais e comunicação segura.

Para gerar uma assinatura digital de um documento, pode-se usar o método “genSignature()”, o qual utiliza a função de hash SHA-3 para gerar o hash do arquivo especificado e os métodos “OAEP encoding()” e “encrypt()” cifrar o hash.

```
def genSignature(self, filename, privateKey):
    data = self.readTextFile(filename)
    print("file = ", data)
    hashed_data = hashlib.sha3_256(data).digest()
    print("Hashed File = ", hashed_data)
    signature = self.encrypt(data=self.OAEPEncode(hashed_data), key=privateKey)
    return signature
```

Figura 29 - Método genSignature()

Para assinar um documento, pode-se usar o método “signTextDocument()”, o qual assinará o documento especificado no seguinte formato: pulará 3 linhas, escreverá 20 “#”, pulará outra linha e escreverá a assinatura especificada.

```
def signTextDocument(self, filename, signature):
    data = self.readTextFile(filename)

    newFilename = filename[:filename.index('.')] + '-signed.txt'
    with open(newFilename, 'w', encoding='latin-1', newline="\n") as file:
        file.write(f'{data.decode("latin-1")}\n\n\n{"#"*20}\n{signature.decode("latin-1")}' )
```

Figura 30 - Método signTextDocument()

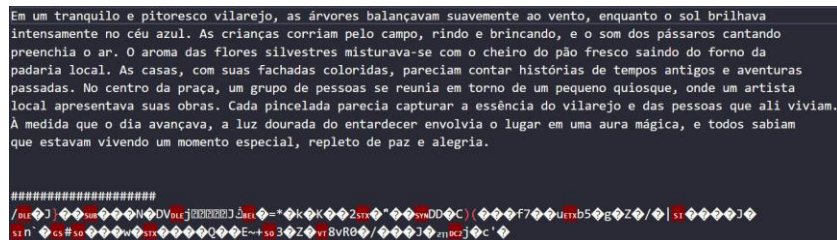


Figura 31- Arquivo assinado

Para verificar a assinatura de um arquivo, pode-se utilizar o método “checkSignature()”, o qual calculará o hash do arquivo especificado e realiza a comparação desse hash com o hash extraído da assinatura.

```
def checkSignature(self, filename, publicKey):
    data = self.readTextFile(filename).decode("latin-1")
    signature = self.extractSignatureFromTextFile(filename)
    i = data.index("\n\n\n"+"#"*20)

    data = data[:i].encode("latin-1")
    signature = self.OAEPDecode(self.decrypt(signature, publicKey))
    hashed_data = hashlib.sha3_256(data).digest()
    return signature == hashed_data
```

Figura 32 - Método checkSignature()

4. CONCLUSÃO

Neste trabalho, foi implementado o algoritmo de criptografia simétrica AES e um gerador/verificador de assinaturas digitais utilizando RSA com OAEP, abordando conceitos fundamentais de segurança computacional. A implementação do AES demonstrou a eficácia da criptografia simétrica na proteção de dados, destacando a importância do uso de chaves seguras e da escolha adequada dos modos de operação. A implementação do RSA com OAEP para assinaturas digitais reforçou a relevância das técnicas de criptografia assimétrica na garantia de autenticidade, integridade e não repúdio de documentos eletrônicos. Os resultados obtidos evidenciam a robustez e a aplicabilidade dessas técnicas em cenários práticos, especialmente em um contexto em que a segurança da informação é primordial. Através da experimentação e análise, foi possível compreender melhor as vantagens e desafios associados a essas abordagens criptográficas, consolidando o conhecimento teórico adquirido ao longo da disciplina e sua aplicação prática em soluções de segurança.

5. REFERÊNCIAS

[https://pt.wikipedia.org/wiki/RSA_\(sistema_criptogr%C3%A1fico\)](https://pt.wikipedia.org/wiki/RSA_(sistema_criptogr%C3%A1fico))

[https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

<https://www.youtube.com/watch?v=s22eJ1eVLTU>

https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test

https://en.wikipedia.org/wiki/Euclidean_algorithm

[https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

<https://en.wikipedia.org/wiki/SHA-3>

https://en.wikipedia.org/wiki/Mask_generation_function