

Trabalho de Implementação 2 - Gerador e Verificador de Assinaturas Digitais RSA

H. de M. O. Lima – 211055281, L. P. Torres – 222011623, and M. N. Miyata – 180126890

Resumo—Este trabalho apresenta a implementação de criptografia assimétrica utilizando o algoritmo RSA. Foram codificados em python, a geração de chaves públicas e privadas, e a criação e verificação de assinaturas digitais. O trabalho também discute o uso de técnicas de otimização para diminuir o custo computacional.

Palavras-chave—Assinaturas Digitais, RSA, Criptografia Assimétrica

I. INTRODUÇÃO

O RSA (Rivest-Shamir-Adleman) é um algoritmo de criptografia assimétrica, em que são utilizadas duas chaves distintas: uma chave pública para criptografar mensagens e uma chave privada para descriptografá-las. O surgimento do RSA foi importante para resolver o problema de enviar uma mensagem criptografada sem que o remetente e o destinatário precisassem compartilhar uma chave secreta previamente [1].

O presente trabalho foi implementado em três partes: (1) geração de chaves públicas e privadas, (2) assinatura digital de mensagens e (3) verificação de assinaturas digitais e descriptografia de mensagens. A seguir, cada uma dessas partes é discutida em detalhes.

II. FUNDAMENTAÇÃO TEÓRICA

Para a execução deste trabalho, foram utilizados principalmente os conceitos de criptografia assimétrica, funções de hash criptográficas, cálculos probabilísticos e a teoria dos números, especialmente a geração de números primos grandes e o cálculo do inverso multiplicativo. Nas próximas seções, cada um desses conceitos será explorado em detalhes no contexto da implementação do RSA.

III. GERAÇÃO DE CHAVES

A geração de chaves se baseia no conceito de “trapdoor one-way function”, ou seja, uma função que é fácil de calcular em uma direção, mas quase impossível de inverter sem uma informação secreta (a “trapdoor”). No caso do RSA, a função é baseada na multiplicação de dois números primos grandes [2].

Neste trabalho, os números primos gerados são de 2048 bits, ou seja, possuem aproximadamente 617 dígitos decimais. Para gerar esses números, são combinados dois métodos: (1) o “Sieve of Sundaram” para eliminar rapidamente números divisíveis por primos pequenos, e (2) o “Miller-Rabin Primality Test” para verificar a primalidade dos números restantes. Como o algoritmo de Miller-Rabin é

mais custoso, ele é aplicado apenas a um subconjunto dos números gerados pelo Sieve of Sundaram.

Além disso, para complementar a geração de chaves, foi utilizada o “Optimal Asymmetric Encryption Padding” (OAEP) para garantir que a mensagem a ser assinada seja menor que o módulo n . A OAEP é uma técnica de padding que adiciona aleatoriedade à mensagem antes da assinatura, aumentando a segurança contra ataques de texto simples [2].

A. Sieve of Sundaram

O código implementado gera um número aleatório n de 2048 bits e utiliza o Sieve of Sundaram para gerar uma lista de números primos menores que n . O Sieve of Sundaram elimina números da forma $i + j + 2ij$, onde $1 \leq i \leq j$, resultando em uma lista de números que podem ser convertidos em primos utilizando a fórmula $2i + 1$. Abaixo está a implementação do Sieve of Sundaram:

```
1 def genPrimesList(nlimit):
2     new_nlimit = (nlimit-1) // 2
3
4     mark_table = [True for i in range(new_nlimit
5                                     +1)]
6     primes_list = []
7
8     for i in range(1, new_nlimit+1):
9         j = i
10        while (i + j + 2*i*j) <= new_nlimit:
11            mark_table[i + j + 2*i*j] = False
12            j += 1
13
14        if nlimit > 2:
15            primes_list.append(2)
16
17        for i in range(1, new_nlimit+1):
18            if mark_table[i]:
19                primes_list.append(2*i + 1)
20
21    return primes_list
```

B. Miller-Rabin Primality Test

Depois de filtrar os números com o Sieve of Sundaram, o código aplica o Miller-Rabin Primality Test para verificar a primalidade dos números restantes. O teste é um algoritmo probabilístico que determina se um número é composto ou provavelmente primo e segue os passos descritos abaixo:

1. **Escolha de Parâmetros:** Escolher um número ímpar $n > 3$ e um número de iterações k .
2. **Verificação Inicial:** Se n for par, o teste retorna FALSO (n é composto).

3. **Decomposição:** Calcula-se $n - 1$ na forma $2^s \cdot d$, onde d é ímpar e s representa quantas vezes $n - 1$ pode ser dividido por 2.
4. **Testes com Bases Aleatórias:** Para cada iteração, escolhe-se uma base aleatória a no intervalo $[2, n-2]$ e calcula-se $x = a^d \pmod{n}$.
5. **Verificação de Condições:** Em cada iteração são verificadas as seguintes condições:
 - Se x é 1 ou $n-1$, o teste continua para a próxima iteração.
 - Caso contrário, repete-se o processo $s - 1$ vezes, calculando $x = x^2 \pmod{n}$ e verificando se x é $n - 1$.
 - Se nenhuma das condições for satisfeita, o teste retorna FALSO (n é composto).
6. **Conclusão:** Se todas as iterações forem concluídas sem retornar FALSO, o teste retorna VERDADEIRO (n é provavelmente primo).

As etapas do Miller-Rabin Primality Test são implementadas no seguinte código:

```

1 def millerRabin(prime_number_candidate,
2   iterations):
3     if prime_number_candidate % 2 == 0:
4       return False
5
6     d = prime_number_candidate - 1
7     s = 0
8
9     while (d % 2 == 0):
10      d //= 2
11      s += 1
12
13     for _ in range(iterations):
14       a = random.randrange(2,
15         prime_number_candidate - 1)
16       x = pow(a, d, prime_number_candidate)
17
18       if x == 1 or x == prime_number_candidate - 1:
19         continue
20
21       for _ in range(s - 1):
22         x = pow(x, 2, prime_number_candidate)
23
24         if x == prime_number_candidate - 1:
25           break
26       else:
27         return False
28
29     return True

```

C. Charmichael para Cálculo do Totiente

A combinação desses dois métodos permite a geração eficiente de números primos grandes p e q , cada um com 2048 bits. Esses números são então utilizados para calcular o módulo $n = p \cdot q$, onde n é o módulo RSA e é utilizado tanto na chave pública quanto na chave privada.

As chaves são calculadas pela função de “Carmichael” ($\lambda(n) = \text{lcm}(p-1, q-1)$), que é uma versão otimizada da função totiente de “Euler” ($\phi(n) = (p-1)(q-1)$). A diferença entre as duas funções é que $\lambda(n)$, é o menor valor

que satisfaz a condição de coprimos com n , enquanto $\phi(n)$ pode ser maior [2]. No código, a função de Carmichael é implementada da seguinte forma:

```

1 def charmichael(n, is_prime):
2     if is_prime:
3         return n-1
4
5     a_list = []
6     exponent = 1
7
8     for i in range(n-1):
9         if math.gcd(n, i+1) == 1:
10            a_list.append(i+1)
11
12     while not doesExponentHoldsForIntegerList(
13       a_list, n, exponent):
14       exponent += 1
15
16     return exponent

```

A função `charmichael` calcula o menor expoente, que é utilizado para calcula o totiente em `totient = math.lcm(p-1, q-1)`. Isso equivale a função de Carmichael definida por $\lambda(n) = \text{lcm}(p-1, q-1)$. Além disso satisfaz a $a^{\lambda(n)} \equiv 1 \pmod{n}$ para todo a coprimo com n [2].

Com o módulo n e o totiente $\lambda(n)$ calculados, a chave pública é formada pelo par (e, n) , onde e é o expoente público escolhido como 65537 por ser um número primo que equilibra segurança e eficiência. A chave privada é formada pelo par (d, n) , onde d é o inverso multiplicativo de e módulo $\lambda(n)$. Abaixo está a implementação da geração das chaves:

```

1 def generateKeys():
2     p = genprimes.genPrimeNumber(2048)
3     q = genprimes.genPrimeNumber(2048)
4
5     n = p*q
6
7     totient = math.lcm(charmichael(p, True),
8       charmichael(q, True))
9
10    e = generateE(totient)
11
12    d = getModularMultiplicativeInverse(e,
13      totient)
14
15    return {"e": e, "d": d, "n": n}

```

D. OAEP

O OAEP é um método de preenchimento usado junto com o RSA para garantir que a cifração seja segura e não determinística. Sem esse mecanismo, uma mesma mensagem cifrada duas vezes com a mesma chave resultaria sempre na mesma cifra, o que poderia facilitar ataques de análise de frequência ou ataques de texto escolhido, nos quais um atacante cifra textos conhecidos e compara os resultados para tentar inferir o conteúdo ou a chave.

O OAEP adiciona aleatoriedade e estrutura à mensagem antes da cifração. Ele realiza uma sequência de transformações que envolvem gerar um número aleatório (seed), embaralhar a mensagem com funções de hash e, por fim,

cifrar o resultado com RSA. Isso garante que a mesma mensagem cifrada repetidas vezes produza cifras diferentes, reforçando a segurança contra ataques de análise de frequência por exemplo [2].

No código desenvolvido, a função `padMessage()` implementa o OAEP com base na especificação PKCS #1, utilizando SHA3-512 como função de hash. O processo ocorre nas seguintes etapas:

1. Montagem do bloco de dados (DB)

A mensagem é organizada no formato `DB = hash(L) || PS || 0x01 || mensagem` onde `hash(L)` é o hash de um rótulo opcional (no caso, vazio), `PS` é uma sequência de bytes zero usada como padding, `0x01` é um delimitador e `mensagem` é o conteúdo original.

2. Geração de uma sequência aleatória (seed)

Um seed aleatório de comprimento `hLen` é gerado usando a biblioteca `secrets`, que provê aleatoriedade adequada para uso criptográfico:

```
1 seed = secrets.token_bytes(self.oaep_hLen)
```

3. Geração das máscaras (MGF1)

A função **Mask Generation Function 1 (MGF1)** é utilizada para criar duas máscaras pseudoaleatórias, que estão implementadas em `DB_mask = MGF1(seed, k - hLen - 1)` e `seed_mask = MGF1(masked_DB, hLen)`. Além disso, a função MGF1 é responsável por expandir o seed em uma máscara de comprimento desejado usando a função de hash SHA3-512:

```
1 def MGF1(self, mgfSeed: bytearray, maskLen:
2     int) -> bytearray:
3     if maskLen > pow(2,32)*self.oaep_hLen:
4         raise ValueError('The mask length is
5             too long to be used on MGF1')
6
7     T_data = bytearray()
8     i = 0
9     while i < maskLen:
10         C_bstr = self.I2OSP(i, 4)
11         concat = bytearray(mgfSeed) +
12             bytearray(C_bstr)
13         seed_hash = bytearray(hashlib.
14             sha3_512(bytes(concat)).digest())
15         T_data.extend(seed_hash)
16         i += 1
17     return T_data[0:maskLen]
```

4. Embaralhamento da mensagem e do seed

As máscaras geradas são aplicadas por meio da operação XOR. O Data Block (DB) é embaralhado com `DB_mask`, e o seed é embaralhado com `seed_mask`.

5. Combinação final e conversão para inteiro

O bloco final codificado (EM) é formado pela concatenação dado por `EM = 0x00 || masked_seed || masked_DB`. Esse bloco tem o mesmo tamanho da chave RSA (`k` bytes) e é convertido para inteiro usando a primitiva OS2IP (Octet String to Integer Primitive) antes da operação modular de cifração:

```
1 m = self.OS2IP(EM)
2 crypt_m = pow(m, e, n)
```

Em conjunto, essas etapas implementam o esquema OAEP, garantindo que o RSA opere sobre dados embaralhados e aleatórios. Esse mecanismo aumenta a segurança da cifração, tornando-a resistente a ataques de texto escolhido e a padrões repetitivos de entrada, além de assegurar a integridade estrutural da mensagem cifrada.

IV. ASSINATURA DIGITAL

A assinatura digital é realizada utilizando a chave privada (d, n). Como sabemos que o custo computacional é alto, o código gera um hash SHA3-256 da mensagem original e assina esse hash. A assinatura é calculada como $s = h(m)^d \pmod{n}$, onde $h(m)$ é o hash da mensagem m .

O hash SHA3-256 gera um valor de 256 bits, que é significativamente menor que o módulo n de 4096 bits, tornando o processo de assinatura mais eficiente. Abaixo está a implementação da função de assinatura digital:

```
1 def calculate_sha3_hash(message):
2     if isinstance(message, str):
3         message = message.encode('utf-8')
4     sha3 = hashlib.sha3_256()
5     sha3.update(message)
6     return int.from_bytes(sha3.digest(), 'big')
```

O hash é então assinado com a chave privada:

```
1 def sign_message(message, d, n):
2     message_hash = calculate_sha3_hash(message)
3     signature = rsa_crypt.encrypt(message_hash, d,
4         n)
5     return signature
```

Por fim, o documento assinado é salvo em um texto estruturado com delimitadores e assinatura em base64 para facilitar o armazenamento e a transmissão.

V. VERIFICAÇÃO DE ASSINATURA DIGITAL

A verificação da assinatura digital é realizada utilizando a chave pública (e, n) e foi implementada em duas etapas principais: (1) parsing do documento assinado para extrair a mensagem original e a assinatura, e (2) verificação da assinatura comparando o hash da mensagem original com o hash recuperado da assinatura.

A função `parse_signed_document` extrai a mensagem e a assinatura do documento assinado:

```
1 def parse_signed_document(signed_document):
2     try:
3         # Primeiro: separar linhas e remover
4             espaços em branco
5         # Segundo: encontrar os índices dos
6             delimitadores
7         # Terceiro: extrair mensagem e assinatura
8         # Quarto: decodificar assinatura de
9             base64 para inteiro
10        return message, signature
11    except Exception as e:
```

```

9      # Lançar erro se o parsing falhar
10     raise ValueError(f"Erro ao fazer parsing
        do documento assinado: {str(e)}")

```

A função `verify_signature` realiza a verificação da assinatura e recebe quatro parâmetros: a mensagem original, a assinatura, o expoente público `e` e o módulo `n`. A verificação é feita da seguinte forma:

```

1 def verify_signature(message, signature, e, n):
2     message_hash = calculate_sha3_hash(message)
3     decrypted_hash = rsa_crypt.decrypt(signature,
4                                     e, n)
5     return message_hash == decrypted_hash

```

VI. DESAFIOS E SOLUÇÕES

Durante a implementação do gerador e verificador de assinaturas digitais RSA, alguns desafios foram enfrentados e soluções foram adotadas para superá-los como serão discutidos a seguir.

A. Geração de Números Primos Grandes

A geração de números primos grandes é um desafio significativo devido à complexidade computacional envolvida. A combinação do Sieve of Sundaram com o Miller-Rabin Primality Test foi adotada para equilibrar eficiência e precisão na geração de números primos de 2048 bits.

O algoritmo de Miller-Rabin possui a complexidade de $O(k \cdot \log^3 n)$, onde k é o número de iterações e n é o número a ser testado. A escolha do número de iterações k é crucial para garantir um equilíbrio entre segurança e desempenho. No código, foi escolhido $k = 40$, o que proporciona uma alta probabilidade de que um número identificado como primo seja realmente primo.

Já o Sieve of Sundaram possui a complexidade de $O(n \log n)$, onde n é o limite superior para a geração de números primos. A combinação desses dois métodos permite a geração eficiente de números primos grandes, minimizando o tempo gasto na verificação de primalidade.

B. Cálculo do Totiente

O cálculo do totiente utilizando a função de Carmichael foi uma escolha estratégica para otimizar o desempenho. A função de Carmichael é mais eficiente que a função totiente de Euler, especialmente para números grandes, pois reduz o tamanho dos valores envolvidos no cálculo do inverso multiplicativo.

Em outras palavras, a função de Carmichael fornece um valor menor que ainda satisfaz as propriedades necessárias para a geração das chaves RSA, resultando em um processo de geração de chaves mais rápido e eficiente.

C. Assinatura e Verificação

A assinatura digital de hashes em vez de mensagens completas foi uma solução adotada para reduzir o custo computacional e ainda assim garantir a autenticidade e

integridade das mensagens. O uso do hash SHA3-256, que gera um valor de 256 bits, ou seja, 1/16 do tamanho do módulo n de 4096 bits, torna o processo de assinatura e verificação significativamente mais eficiente. Uma vez que a complexidade da exponenciação modular é $O(\log^3 n)$, cada bit a menos no valor a ser assinado ou verificado resulta em uma redução substancial no tempo de computação.

VII. CONCLUSÃO

Este trabalho apresentou a implementação de um gerador e verificador de assinaturas digitais utilizando o algoritmo RSA. A combinação do Sieve of Sundaram com o Miller-Rabin Primality Test permitiu a geração eficiente de números primos grandes, essenciais para a segurança do RSA. A utilização da função de Carmichael otimizou o cálculo do totiente, e a assinatura de hashes em vez de mensagens completas reduziu o custo computacional das operações de assinatura e verificação. A implementação foi realizada em Python, utilizando bibliotecas padrão para manipulação de números grandes e funções hash.

REFERÊNCIAS

- [1] Veritas, “O que é a criptografia RSA?” <https://www.veritas.com/pt/br/information-center/rsa-encryption>, 2025.
- [2] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2014.