

# 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 um gerador e verificador de assinaturas digitais utilizando o algoritmo RSA. O objetivo é garantir a autenticidade e integridade das mensagens trocadas entre as partes envolvidas. A implementação inclui a geração de chaves públicas e privadas, a assinatura digital de mensagens e a verificação dessas assinaturas. O trabalho também discute os desafios enfrentados durante o desenvolvimento e as soluções adotadas para superá-los.

## 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 precisem 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. 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.

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

- [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.