Implementação de Assinatura Digital com RSA e OAEP

Giovanni Minari Zanetti, 202014280
Giulia Moura Ferreira, 200018795
Tiago Leão Buson, 200034162

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

Sumário

1. Introdução

A criptografia desempenha um papel fundamental na segurança da informação, sendo essencial para garantir a confidencialidade, autenticidade e integridade das comunicações digitais. Dentro deste contexto, o algoritmo RSA se destaca como um dos mais utilizados para a criptografia assimétrica, sendo amplamente empregado em diversas aplicações, desde a segurança em transmissão de dados até a implementação de assinaturas digitais. [6] [1]

Desenvolvido por Ron Rivest, Adi Shamir e Leonard Adleman em 1977, o **RSA** baseia-se na dificuldade computacional da fatorização de números primos grandes. [8] Para aumentar sua segurança e evitar vulnerabilidades conhecidas, técnicas como o padding **OAEP** (Optimal Asymmetric Encryption Padding) [2] e a utilização de funções hash criptográficas, como o **SHA-3** [7], são incorporadas.

Neste trabalho, é implementado um sistema de assinatura digital baseado em RSA, utilizando OAEP para reforçar a segurança e minimizar ataques. O projeto engloba a geração de chaves, a cifragem e decifragem de mensagens, bem como a criação e verificação de assinaturas digitais, garantindo a integridade e autenticidade dos dados.

O código-fonte completo do projeto está disponível no repositório do GitHub (link), permitindo acesso ao código.

1.1. Criptografia RSA

O **RSA** utiliza um par de chaves (pública e privada) para garantir a segurança da comunicação [5]. A geração das chaves ocorre conforme os seguintes passos:

- Escolha de dois números primos grandes (p e q)
- Cálculo de $n = p \cdot q$ (módulo)
- Cálculo da função totiente de Euler $\phi(n) = (p-1) * (q-1)$ (função totiente de Euler)
- Escolha de um expoente público e (geralmente 65537)
- Cálculo do expoente privado d (inverso multiplicativo modular de e módulo $\phi(n)$)

A segurança do **RSA** depende da dificuldade de fatorar o número , tornando inviável a obtenção da chave privada sem acesso direto aos fatores primos.

1.2. Teste de primalidade de Miller–Rabin

O teste de primalidade de **Miller-Rabin** detecta se um número é primo ou não. Este é um teste essencial para a criptografia assimétrica, uma vez que a necessidade de uma grande quantidade de números primos grandes é muito importante para a segurança dos algoritmos. Como este teste não dá indícios sobre a fatoração no número, ele confere uma maior segurança e é o mais utilizado para o teste de primalidade [4]. Ele funciona da seguinte forma:

- seja **n** um número primo, escolhe-se um número **a** aleatório, tal que $1 < a < n$
- seja **r** um expoente máximo tal que $n - 1$ é divisível por 2^r
- seja **s** um número tal que $\mathbf{s} = \frac{n-1}{2^r}$

Se **n** é um número primo e **a** não tiver um divisor em comum com **n**, então:

- $a^s \ mod \ n = 1$
- ou existe um **j** $\in$ **{0, 1, ... , r-1}**, tal que
- $a^{2^r d} \ mod \ n = -1$

Um número **a** que não satisfaz o teorema acima é denominado de testemunha contra a primalidade de **n**

1.3. OAEP (Optimal Asymmetric Encryption Padding)

O **OAEP** utiliza um esquema estruturado de padding, onde bytes **0x00** são adicionados antes da mensagem, seguidos por um byte delimitador **0x01**. A aleatoriedade é introduzida pelo uso de uma máscara gerada, garantindo segurança contra ataques. [3]

O **OAEP** utiliza:

- Uma função de hash criptográfica (como **SHA3-256**).
- Uma função de geração de máscara (**MGF**).
- Um **padding estruturado**, composto por uma sequência de bytes `0x00` seguidos por um delimitador `0x01`, garantindo que a mensagem possa ser corretamente identificada na decriptação.

Estas características tornam a cifragem probabilística, significando que a mesma mensagem cifrada múltiplas vezes produzirá diferentes cifrados.

1.4. Funções Hash Criptográficas - SHA-3

SHA-3 (Secure Hash Algorithm 3) é a mais recente família de funções hash criptográficas padronizada pelo NIST. Baseada no algoritmo Keccak, o **SHA-3** foi projetado como uma alternativa ao **SHA-2**, oferecendo:

- Resistência a colisões
- Resistência a pré-imagem
- Resistência a segunda pré-imagem

O **SHA-3** pode produzir hashes de diferentes tamanhos (224, 256, 384 e 512 bits), sendo o **SHA3-256** e **SHA3-512** os mais comumente utilizados.

1.5. Assinaturas Digitais RSA

Uma assinatura digital **RSA** combina o **RSA** com funções hash para criar um mecanismo de autenticação e integridade. O processo envolve:

- Geração do hash da mensagem original
- Cifragem do hash com a chave privada do remetente
- Anexação da assinatura à mensagem

A verificação ocorre através de:

- Decifragem da assinatura com a chave pública do remetente
- Comparação do hash decifrado com o hash da mensagem recebida

1.5.1. Importância da Codificação BASE64

A codificação **BASE64** é utilizada para converter dados binários em texto ASCII, permitindo:

- Transmissão segura por canais que suportam apenas texto
- Representação consistente da assinatura
- Facilidade de armazenamento e manipulação

2. Desenvolvimento

2.1. Visão Geral do Projeto

O projeto implementa um Gerador/Verificador de Assinaturas **RSA** em arquivos, garantindo a autenticidade e integridade das mensagens. A assinatura digital baseada em **RSA** e **OAEP** foi desenvolvida utilizando conceitos fundamentais de criptografia assimétrica, combinando algoritmos robustos de hash e padding seguro para evitar vulnerabilidades comuns.

O funcionamento geral do sistema ocorre da seguinte forma:

Encriptação e Assinatura

- Geração das chaves **RSA** (pública e privada) com números primos grandes (mínimo de 1024 bits), testados pelo algoritmo de **Miller-Rabin**.
- Cálculo do hash da mensagem em claro, usando **SHA-3** (*sha3_256*).
- Aplicação do **OAEP** para proteção contra ataques de texto-cifrado.
- Assinatura digital da mensagem (cifração do hash) utilizando **RSA**.
- Codificação do resultado em **BASE64** para facilitar transmissão e armazenamento.

Decriptação e Verificação

- PDecodificação do **BASE64**e separação da mensagem e assinatura.
- Decifração **RSA** da assinatura (decifração do hash) com a chave pública.
- Comparação do hash obtido com o hash da mensagem recebida para verificar a integridade.

2.2. Geração de Chaves

```python
# Gerar par de chaves pública e privada para o RSA
def generate_keys(key_size=1024):  3 usages
    p = generate_large_prime(key_size)
    q = generate_large_prime(key_size)

    n = p * q
    phi = (p - 1) * (q - 1)

    e = 65537  # Expoente público comumente usado
    d = modular_inverse(e, phi)

    return {
        'public_key': (e, n),
        'private_key': (d, n),
        "primes": (p, q)
    }
```

Figure 1. Função de geração de chaves

A figura 1 mostra a função de geração de chaves do **RSA**. Essa função chama a função de geração de números primos, escolhe o valor do expoente público como sendo 65537, que é um valor comumente usado, e também chama a função de inverso modular, para calcular **d**.

```python
def generate_large_prime(bits=1024):  2 usages
    # Gera um número primo aleatório de 1024 bits e o testa pelo metodo de Miller-Rabin
    while True:
        candidate = random.getrandbits(bits)
        candidate |= (1 << bits - 1) | 1
        if miller_rabin_test(candidate):
            return candidate
```

Figure 2. Função de geração de números primos

A figura 2 mostra a função de geração de números primos, que chama a função de teste de **Miller-Rabin**

```python
def miller_rabin_test(n, k=20):  1 usage
    # Teste probabilístico de primalidade de um número, usando iterações (20 como default)
    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False

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

    for _ in range(k):
        a = random.randrange( start: 2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

Figure 3. Função Miller Rabin

A imagem 3 mostra a implementação do teste de primalidade **Miller-Rabin**. Inicialmente testa se o número é menor que 2 (não é primo), se é 2 ou 3 (é primo) ou se é divisível por 2 (nesse caso elimina-se metade das possibilidades). Ao aumentar o valor de **k**, aumenta-se a precisão do teste, e ao diminuir, reduz-se o custo computacional. Para manter um equilíbrio, foi escolhido o valor de 20 iterações para o teste.

```python
def modular_inverse(a, m):  1 usage
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        a, m = m, a % m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1
```

Figure 4. Função Inverso Modular

A figura 4 mostra a implementação da funcão de inverso modular, que usa o Algoritmo de Euclides Extendido para ser mais rápida e eficiente.

Esse processo garante que apenas a chave privada pode decifrar mensagens cifradas com a chave pública e vice-versa.

2.3. Encriptação

```python
def encrypt(message, public_key):  2 usages
    # Encripta a mensagem usando RSA-OAEP
    e, n = public_key

    # Converte mensagem para bytes se é uma string
    if isinstance(message, str):
        message = message.encode('utf-8')

    # Aplica padding à mensagem
    padded = oaep_pad(message, n)

    # Converte para inteiro e encripta
    m_int = int.from_bytes(padded, byteorder: 'big')
    c_int = pow(m_int, e, n)

    return c_int
```

Figure 5. Função de Encriptação RSA

A figura 5 mostra a função de encriptação **RSA**. Ela utiliza a chave pública para tal, e aplica um padding à mensagem por meio do **OAEP**.

2.4. OAEP

```python
def oaep_pad(message, n):  1 usage
    # Aplica padding OAEP à mensagem
    k = (n.bit_length() + 7) // 8  # Comprimento de n em bytes
    mLen = len(message)

    # Checagem de comprimento
    if mLen > k - 2 * HASH_LENGTH - 2:
        raise ValueError("Message too long")

    # Gera padding aleatório
    lHash = HASH_FUNCTION(b'')  # Hash da label vazia
    PS = b'\x00' * (k - mLen - 2 * HASH_LENGTH - 2)
    DB = lHash + PS + b'\x01' + message

    # Gera seed aleatória
    seed = os.urandom(HASH_LENGTH)

    # Mascara DB
    dbMask = mgf1(seed, k - HASH_LENGTH - 1)
    maskedDB = bytes(a ^ b for a, b in zip(DB, dbMask))

    # Mascara seed
    seedMask = mgf1(maskedDB, HASH_LENGTH)
    maskedSeed = bytes(a ^ b for a, b in zip(seed, seedMask))

    return b'\x00' + maskedSeed + maskedDB
```

Figure 6. Função de Padding OAEP

A figura 6 mostra a função de padding OAEP, e este é realizado da seguinte forma:

- Gera-se um hash de uma label opcional para ser associado à mensagem (**lHash**);
- Preenchimento (**PS**) de *0x00* com tamanho k - mLen - 2 * HASH_LENGTH - 2, onde **k** é o tamanho do módulo RSA **n** em bytes;
- Concatenação (**DB**) de **lHash**, **PS**, *0x01* e da mensagem;
- Geração de uma seed de mesmo tamanho da saída do hash para gerar uma máscara (**dbMask**) para **DB** de tamanho k - HASH_LENGTH - 1;
- Aplicação de uma operação XOR entre **DB** e **dbMask** para ofuscar **DB** (**maskedDB**);
- Geração de **seedMask** a partir de **maskedDB** com mesmo tamanho da saída do hash;
- Ofuscamento da **seed** (**maskedSeed**) aplicando XOR entre **seed** e **seedMask**;
- Concatenação de *0x00*, **maskedSeed** e **maskedDB**, resultando no bloco final OAEP que será encriptado com RSA.

```python
def mgf1(seed, length):  4 usages
    if length > (2**32 * HASH_LENGTH):
        raise ValueError("Mask too long")

    T = b''
    for counter in range(ceil(length / HASH_LENGTH)):
        C = counter.to_bytes( length: 4,  byteorder: 'big')
        T += HASH_FUNCTION(seed + C)
    return T[:length]
```

Figure 7. Função MGF1

A figura 7 mostra o funcionamento da função de geração de máscara, que pode possuir tamanho variado. A máscara é gerada pela concatenação de hashes de valores de um contador, extendidos para 4 bytes.

2.5. Decriptação

```python
def decrypt(ciphertext, private_key):  2 usages
    # Decripta a mensagem usando RSA-OAEP
    d, n = private_key

    # Decripta
    m_int = pow(ciphertext, d, n)

    # Converte para bytes
    em = m_int.to_bytes((n.bit_length() + 7) // 8,  byteorder: 'big')

    # Tira o padding
    message = oaep_unpad(em, n)

    # Tenta decodificar como UTF-8 se possível
    try:
        return message.decode('utf-8')
    except UnicodeDecodeError:
        return message
```

Figure 8. Função de Decriptação RSA

A figura 8 mostra a função de decriptação **RSA**, que usa chave privada. Após decriptar a mensagem, o padding **OAEP** é retirado e a mensagem original é retornada.

```python
def oaep_unpad(padded, n):  1 usage
    # Remove padding OAEP da mensagem
    k = (n.bit_length() + 7) // 8

    # Checagem básica de formato
    if len(padded) != k:
        raise ValueError("Erro de decodificação")
    if padded[0] != 0:
        raise ValueError("Erro de decodificação")

    # Separa a mensagem
    maskedSeed = padded[1:HASH_LENGTH + 1]
    maskedDB = padded[HASH_LENGTH + 1:]

    # Recupera a seed
    seedMask = mgf1(maskedDB, HASH_LENGTH)
    seed = bytes(a ^ b for a, b in zip(maskedSeed, seedMask))

    # Recupera DB
    dbMask = mgf1(seed, k - HASH_LENGTH - 1)
    DB = bytes(a ^ b for a, b in zip(maskedDB, dbMask))

    # Verifica o padding
    lHash = HASH_FUNCTION(b'')
    if not DB.startswith(lHash):
        raise ValueError("Erro de decodificação")

    # Acha a mensagem
    i = HASH_LENGTH
    while i < len(DB):
        if DB[i] == 1:
            return DB[i + 1:]
        if DB[i] != 0:
            raise ValueError("Erro de decodificação")
        i += 1
    raise ValueError("Erro de decodificação")
```

Figure 9. Função de Unpadding

A figura 9 mostra a função de unpadding. Nesta função, **maskedSeed** e **maskedDB** são separados, e a partir deles, são geradas máscaras com a função **MGF1** para obter a seed e **DB** originais.

Depois, itera em DB a partir do endereço seguinte a **lHash** (hash da label opcional), e verifica se *0x01* foi encontrado. Se sim, isso significa que a mensagem estará contida a partir do próximo endereço.

Esse processo impede ataques baseados na análise de padrões do texto cifrado

2.6. Assinatura RSA

```python
def sign_message(message, private_key):  2 usages
    # Calcula o hash da mensagem
    hash_msg = HASH_FUNCTION(message.encode('utf-8'))

    # Cifra o hash com a chave privada
    signature = encrypt(hash_msg, private_key)

    return signature
```

Figure 10. Função de Assinatura RSA

A figura 10 mostra a função que realiza a assinatura **RSA**. Os hashes da mensagem são calculados e são encriptados pela encriptação **RSA** usando a chave privada.

Esse processo assegura que apenas o detentor da chave privada pode gerar assinaturas válidas.

2.7. Verificação

```python
def parse_signed_document(signed_document):  1 usage
    try:
        print("Iniciando o parsing do documento assinado.")
        # Decodifica o documento Base64
        decoded_data = base64.b64decode(signed_document)
        print("Documento decodificado de Base64 com sucesso.")

        # Divide a mensagem e a assinatura
        message_length = int.from_bytes(decoded_data[:4], byteorder: 'big')
        print(f"Comprimento da mensagem extraído: {message_length} bytes.")

        message = decoded_data[4:4 + message_length].decode('utf-8')
        print(f"Mensagem extraída: {message}")

        signature = int.from_bytes(decoded_data[4 + message_length:], byteorder: 'big')
        print(f"Assinatura extraída: {signature}")

        return message, signature
    except Exception as e:
        raise ValueError(f"Erro ao fazer o parsing do documento assinado: {e}")
```

Figure 11. Função de Parsing do Documento Assinado

A figura x mostra a função que realiza o parse do documento assinado, primeiramente decodificando-o de **BASE64** para um formato de bytes. Finalmente, separa a mensagem da assinatura e retorna esses valores.

```python
def verify_signed_document(signed_document, public_key):  2 usages
    try:
        print("Iniciando a verificação do documento assinado...")
        # Parsing do documento assinado
        message, signature = parse_signed_document(signed_document)

        # Verifica a assinatura
        print("Verificando a assinatura...")
        is_valid = verify_signature(message, signature, public_key)
        print("Verificação da assinatura concluída.")
        return is_valid, message
    except ValueError as e:
        return False, str(e)
```

Figure 12. Função de Verificação do Documento Assinado

A figura 12 mostra a função de verificação de documento assinado, que inicialmente chama a função que faz o parse no documento, e depois chama a função de verificação de assinatura, finalmente, retorna um booleano que indica se a assinatura é válida ou não, e a mensagem original.

```python
def verify_signature(message, signature, public_key):  2 usages
    # Calcula o hash da mensagem
    hash_msg = HASH_FUNCTION(message.encode('utf-8'))
    hash_int_original = int.from_bytes(hash_msg,  byteorder: 'big')

    # Decifra a assinatura usando a chave pública
    hash_signed = decrypt(signature, public_key)
    hash_int_signed = int.from_bytes(hash_signed,  byteorder: 'big')

    # Verifica se o hash calculado é igual ao hash da assinatura
    return hash_int_original == hash_int_signed
```

Figure 13. Função de Verificação da Assinatura

A figura 13 mostra a função de verificação da assinatura, que calcula o hash da mensagem original recebida separadamente, decifra a assinatura, obtendo o hash recebido, e compara os dois, para validar sua procedência.

2.8. Execução do Programa

```python
if __name__ == "__main__":

    print("Gerando chaves RSA...")
    keys = generate_keys(key_size=1024)
    print("Chaves geradas com sucesso.")

    # Lendo a mensagem do arquivo
    try:
        with open("documento.txt", "r", encoding="utf-8") as file:
            original_message = file.read().strip()
        print(f"Mensagem lida do arquivo: {original_message}")
    except FileNotFoundError:
        print("Erro: O arquivo 'documento.txt' não foi encontrado.")
        exit(1)

    print("Gerando a assinatura da mensagem...")
    signature = sign_message(original_message, keys['private_key'])
    print(f"Assinatura gerada: {signature}")

    print("Formatando o documento assinado em Base64...")
    message_bytes = original_message.encode('utf-8')
    message_length_bytes = len(message_bytes).to_bytes( length: 4,  byteorder: 'big')
    signature_bytes = signature.to_bytes((keys['public_key'][1].bit_length() + 7) // 8, 'big')

    signed_document = base64.b64encode(message_length_bytes + message_bytes + signature_bytes).decode('utf-8')
    print(f"Documento assinado formatado: {signed_document}")

    # Verifica o documento assinado
    is_valid, result = verify_signed_document(signed_document, keys['public_key'])

    if is_valid:
        print(f"A assinatura é válida. Mensagem original: {result}")
    else:
        print(f"A assinatura é inválida. Erro: {result}")
```

Figure 14. Main.py

O programa principal (figura 14) integra todas as partes do sistema, realizando a geração de chaves, leitura da mensagem, assinatura digital, formatação do documento assinado e verificação da autenticidade. O fluxo de execução é:

1. Geração das chaves RSA;
2. Leitura da mensagem de um arquivo *documento.txt*;
3. Geração da assinatura digital;
4. Formatação do documento assinado usando **Base64**;
5. Verificação da autenticidade da assinatura.

O programa imprime no console cada etapa do processo, facilitando a depuração e compreensão da execução.

3. Conclusão

Os resultados da execução do programa (figura 15) demonstram o correto funcionamento do sistema de assinatura digital baseado em RSA e OAEP. As chaves foram geradas com sucesso, a mensagem foi corretamente lida do arquivo, assinada, e codificada em Base64 para transmissão segura. A assinatura digital foi validada com sucesso, garantindo a autenticidade e integridade da mensagem original.

```
Gerando chaves RSA...
Chaves geradas com sucesso.
Mensagem lida do arquivo: Mensagem de teste de assinatura e verificação!
Gerando a assinatura da mensagem...
Assinatura gerada: 569380564539652949241734436449496740751537178124694021455310061210766
Formatando o documento assinado em Base64...
Documento assinado formatado: AAAAME1lbnNhZ2VtIGRlIHRlc3RlIGRlIGFzc2luYXR1cmEgZSB2ZXJpZml
Iniciando a verificação do documento assinado...
Iniciando o parsing do documento assinado.
Documento decodificado de Base64 com sucesso.
Comprimento da mensagem extraído: 48 bytes.
Mensagem extraída: Mensagem de teste de assinatura e verificação!
Assinatura extraída: 569380564539652949241734436449496740751537178124694021455310061210766
Verificando a assinatura...
Verificação da assinatura concluída.
A assinatura é válida. Mensagem original: Mensagem de teste de assinatura e verificação!
```

Figure 15. Resultados

3.1. Explicação dos resultados

1. **Geração das chaves RSA**: O sistema gera um par de chaves (pública e privada) com sucesso.
2. **Leitura da mensagem**: A mensagem original é lida de um arquivo.
3. **Codificação do documento assinado**: A assinatura digital é formatada em Base64, permitindo um armazenamento e transmissão mais seguros.
4. **Verificação da assinatura digital**:
   - O sistema realiza o parsing do documento assinado, extraindo a mensagem original e a assinatura.
   - A assinatura é decifrada usando a chave pública e comparada ao hash da mensagem original.
   - Como os valores coincidem, a assinatura é validada com sucesso.

3.2. Conclusão Final

Este experimento evidencia a eficiência do uso de **criptografia RSA** associada ao **padding OAEP** e **hashing SHA-3** para reforçar a segurança. A verificação da assinatura confirma que o sistema implementado é capaz de detectar alterações não autorizadas, tornando-o um mecanismo robusto para garantir comunicações seguras. Futuras melhorias podem incluir a implementação de chaves **RSA** maiores para maior segurança e otimização do desempenho da verificação de assinaturas.

Futuras melhorias podem incluir:

- Implementação de chaves RSA maiores para aumentar a segurança.
- Otimização do tempo de verificação de assinaturas.
- Adaptação do projeto para suportar múltiplos formatos de documentos.

References

[1] AWS. O que é criptografia? `https://aws.amazon.com/pt/what-is/cryptography/#:~:text=Criptografia%20Ãl%20a%20prÃątica%20de,durante%20a%20computaÃğÃčo%20de%20dados).` Accessed: 2025-01-24.

[2] Wikipedia contributors. Optimal asymmetric encryption padding — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Optimal_asymmetric_encryption_padding&oldid=1264449538, 2024. Accessed: 2025-01-24.

[3] Prof Bill Buchanan OBE FRSE. So how does padding work in rsa? https://medium.com/asecuritysite-when-bob-met-alice/so-how-does-padding-work-in-rsa-6b34a123ca1f. Accessed: 2025-01-26.

[4] Geeks4Geeks. Primality test | set 3 (miller–rabin). https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/. Accessed: 2025-01-26.

[5] Rafael Sousa. Entendendo algoritmo rsa (de verdade). https://hackingnaweb.com/criptografia/entendendo-algoritmo-rsa-de-verdade/. Accessed: 2025-01-24.

[6] Equipe TOTVS. O que significa rsa e qual sua relação com criptografia? https://www.totvs.com/blog/gestao-para-assinatura-de-documentos/rsa/. Accessed: 2025-01-24.

[7] Wikipédia. Sha-3 — wikipédia, a enciclopédia livre. https://pt.wikipedia.org/w/index.php?title=SHA-3&oldid=66004885, 2023. Accessed: 2025-01-24.

[8] Wikipédia. Rsa (sistema criptográfico) — wikipédia, a enciclopédia livre. https://pt.wikipedia.org/w/index.phptitle=RSA_(sistema_criptogr%C3%A1fico)&oldid=68710646, 2024. Accessed: 2025-01-24.