



GIOVANNI MINARI ZANETTI, 202014280  
GIULIA MOURA FERREIRA, 200018795  
TIAGO LEÃO BUSON, 200034162



# Implementação de Assinatura Digital com RSA e OAEP

DEP. CIÊNCIA DA COMPUTAÇÃO – UNIVERSIDADE DE BRASÍLIA (UNB)  
CIC0201 - SEGURANÇA COMPUTACIONAL



# Introdução

- A **criptografia** é fundamental para a segurança da informação, garantindo confidencialidade, autenticidade e integridade das comunicações.
- O **RSA** é um dos algoritmos mais populares de criptografia assimétrica.
- Para aumentar a segurança do RSA, o projeto implementa o **OAEP**
- O projeto desenvolve e implementa um sistema de assinatura digital, garantindo proteção contra modificações não autorizadas.

# Criptografia RSA

Baseado na dificuldade  
da fatoração de números  
primos grandes

```
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)
    }
```

## UTILIZA UM PAR DE CHAVES

- **Chave pública:** utilizada para encriptação da mensagem.
- **Chave privada:** utilizada para decryptação da mensagem e assinatura digital.

## PROCESSO BÁSICO

- Escolha de dois números primos grandes **p** e **q**.
- Cálculo de  **$n=p \times q$** .
- Definição do expoente público **e**, nesse caso **65537**.
- Cálculo do expoente privado **d**, o inverso modular de  **$e \bmod \varphi(n)$** .



```
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
```

# Teste de Primalidade de Miller-Rabin

Um teste probabilístico que verifica se um número é primo

## NÚMEROS PRIMOS GRANDES

- Para garantir segurança, os números **p** e **q** precisam ser primos grandes

```
def miller_rabin_test(n, k=20): 1 usage
    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
```

```
def generate_large_prime(bits=1024): 2 usages
    while True:
        candidate = random.getrandbits(bits)
        candidate |= (1 << bits - 1) | 1
        if miller_rabin_test(candidate):
            return candidate
```

## FUNCIONAMENTO

- Escolhe um número aleatório **a**.
- Verifica se **n** passa nos critérios matemáticos de número primo.
- O número de iterações aumenta a precisão do teste.

# OAEP

## Optimal Asymmetric Encryption Padding

Adiciona aleatoriedade, tornando a criptografia mais segura

### VULNERABILIDADES DO RSA

- O **RSA** sem padding pode ser vulnerável a ataques

### ETAPAS

1. Geração de um **hash** de uma label opcional
2. Preenchimento (**padding**) estruturado com bytes 0x00
3. Aplicação da função de geração de máscara (**MGF**)
4. Ofuscação da mensagem original antes da encriptação

```
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
```

```
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]
```

# Função Hash SHA-3

Um algoritmo de hash seguro que protege contra colisões e ataques de pré-imagem.

## FUNÇÃO HASH NO PROJETO

- Transforma mensagens de tamanho variável em um valor **fixo**.
- Garante **integridade** da mensagem ao compará-la após a assinatura digital.

## SHA3-256

- O projeto utiliza **SHA3-256**, que gera hashes de **256** bits

```
import hashlib
HASH_FUNCTION = lambda x: hashlib.sha3_256(x).digest()
HASH_LENGTH = len(HASH_FUNCTION(b''))
```

```
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
```

# Encriptação

1. A mensagem é transformada em um hash com **SHA-3**
2. Aplica-se o **padding OAEP** para segurança extra.
3. O resultado é cifrado com a **chave pública**

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

    # Decrypta
    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
```

# Deccriptação

1. A mensagem cifrada é deccriptada com a **chave privada**.
2. O padding OAEP é **removido**.
3. A mensagem original é **restaurada** e comparada com o hash armazenado.



# Remoção do padding, ou Unpadding

1. O resultado da decifragem contém os elementos ofuscados por OAEP, incluindo:
  - a. **maskedSeed** (semente mascarada).
  - b. **maskedDB** (bloco de dados mascarado).
2. A seed original é extraída usando a função **MGF1** e uma operação **XOR** com **maskedSeed**.
3. O bloco de dados original (**DB**) é recuperado aplicando **XOR** entre **maskedDB** e a **máscara gerada pela seed**.
4. O código percorre DB até encontrar 0x01, que marca o **início da mensagem**.
5. Todos os bytes após 0x01 são extraídos como a **mensagem original**.

```
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")
```

# Assinatura Digital RSA

Garante autenticidade e integridade da mensagem.

```
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
```

```
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
```

## PROCESSO DE ASSINATURA

1. Gera-se o **hash da mensagem** usando SHA-3
2. O hash é **cifrado com a chave privada**, gerando a assinatura digital
3. A assinatura é **anexada** à mensagem

## VERIFICAÇÃO DA ASSINATURA

1. O destinatário **decifra a assinatura** com a chave pública
2. O hash obtido é **comparado** com o hash da mensagem recebida.
3. Se forem idênticos, a assinatura é **válida**.

# Parsing da Assinatura Digital

```
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}")
```

## ANÁLISE DO DOCUMENTO ASSINADO

1.

O documento assinado é lido e convertido de **Base64** para bytes

2.

A função **separa** a assinatura da mensagem original

3.

**Retorna** esses dois elementos para a verificação posterior

# Verificação da Assinatura Digital

```
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)
```

## VERIFICAÇÃO DA ASSINATUR

1. Chama a função de análise para obter a **mensagem** e a **assinatura**.
2. **Recalcula** o hash da mensagem original usando **SHA-3**.
3. **Decifra** a assinatura usando a **chave pública** do remetente.
4. Compara o hash da **assinatura decifrada** com o hash da **mensagem original**.
5. Retorna **verdadeiro (válido)** se os hashes forem idênticos, caso contrário, a assinatura é considerada **inválida**.



# Execução do Programa

Fluxo de funcionamento do código

1. Geração das **chaves RSA**.
2. Leitura da mensagem a partir de um **arquivo**.
3. Geração da **assinatura digital**.
4. Codificação em **Base64** para facilitar transmissão e armazenamento.
5. **Verificação** da assinatura e **validação** da autenticidade da mensagem.

```
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}")
```

# Resultados

O programa exibe os resultados no console, permitindo acompanhar cada etapa

1.

```
Gerando chaves RSA...  
Chaves geradas com sucesso.
```

2.

```
Mensagem lida do arquivo: Mensagem de teste de assinatura e verificação!  
Gerando a assinatura da mensagem...  
Assinatura gerada: 569380564539652949241734436449496740751537178124694021 ...
```

3.

```
Formatando o documento assinado em Base64...  
Documento assinado formatado: AAAAME1lbnNhZ2VtIGRlI ...
```

4.

```
Iniciando a verificação do documento assinado...  
Iniciando o parsing do documento assinado.  
Documento decodificado de Base64 com sucesso.
```

5.

```
Comprimento da mensagem extraído: 48 bytes.  
Mensagem extraída: Mensagem de teste de assinatura e verificação!  
Assinatura extraída: 5693805645396529492417344364494967407515371781 ...
```

6.

```
Verificando a assinatura...  
Verificação da assinatura concluída.  
A assinatura é válida. Mensagem original: Mensagem de teste de assinatura e verificação!
```



# Conclusão

- O projeto demonstrou a **viabilidade** da assinatura digital usando **RSA e OAEP**.
- O uso de **SHA-3** como função hash aumenta a **segurança**.
- O sistema consegue detectar **alterações na mensagem** e **rejeitar mensagens forjadas**.

## Possíveis melhorias:

- Uso de chaves **RSA maiores** para segurança extra.
- Otimização do código para **maior desempenho**.
- Implementação de suporte a outros algoritmos de hash, como **SHA3-512**.

# Referências

- Baseado em materiais acadêmicos, artigos e documentação sobre RSA, OAEP e SHA-3.
- **Fontes principais:**
  - Wikipedia
  - GeeksforGeeks
  - Medium
  - AWS Docs
  - Documentação oficial do NIST