

Trabalho 3: Gerador e Verificador de Assinaturas Digitais com RSA-PSS

Leandro Beloti Kornelius - 211020900
Lucca Magalhães Boselli Couto - 222011552
Vitor Caldas Danelon Lopes - 222031822

¹Departamento de Ciência da Computação – Universidade de Brasília (UnB)
Disciplina: Segurança Computacional, 2025/1

1. Contextualização

A criptografia assimétrica é um pilar da segurança da informação moderna, permitindo a comunicação segura e a verificação de autenticidade em canais inseguros. Diferentemente da criptografia simétrica, que utiliza uma única chave, a abordagem assimétrica emprega um par de chaves: uma pública, para cifrar ou verificar, e uma privada, para decifrar ou assinar.

As assinaturas digitais são uma das aplicações mais importantes dessa tecnologia, garantindo a autenticidade, integridade e o não-repúdio de documentos digitais. Elas funcionam como uma contraparte digital de uma assinatura manuscrita, mas com garantias de segurança matemática.

Neste trabalho, foi desenvolvida uma ferramenta para gerar e verificar assinaturas digitais utilizando o algoritmo RSA-PSS, conforme as especificações propostas. Este relatório detalha a fundamentação teórica, a arquitetura da implementação e os resultados dos testes realizados.

2. Implementação do Gerador/Verificador

A solução foi desenvolvida em Python, com o código modularizado em três partes principais, espelhando as etapas do processo de assinatura digital. O código-fonte completo pode ser encontrado no seguinte repositório do GitHub: <https://github.com/LeandroKornelius/UnB-Security/tree/main/Trabalho%203>

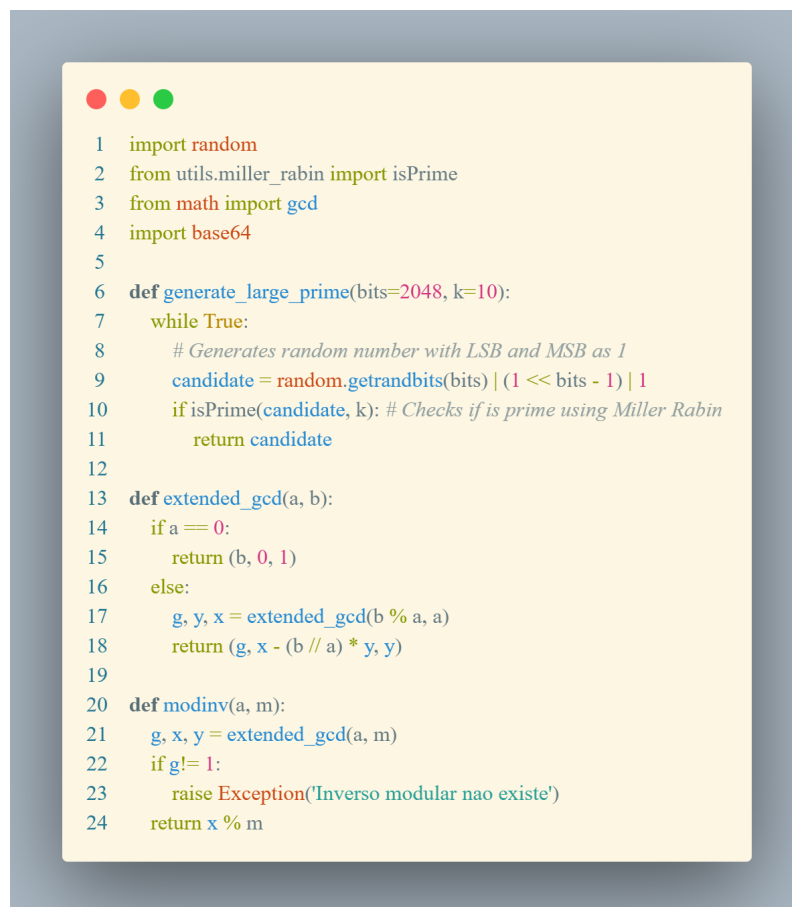
2.1. Parte 1: Geração de Chaves e Cifra

A base para o sistema RSA é a geração de um par de chaves seguro. Este processo foi implementado do zero, seguindo as etapas matemáticas fundamentais.

Primeiramente, foi implementada a função `generate_large_prime`, que utiliza o teste de primalidade de Miller-Rabin para encontrar números primos grandes (com 2048 bits por padrão), garantindo a robustez das chaves. Em seguida, a função `generate_keypair` orquestra a criação do par de chaves:


1. Gera dois primos distintos, p e q .
2. Calcula o módulo $n = p * q$ e o totiente de Euler $\phi = (p-1)*(q-1)$.
3. Define o expoente público e (usualmente 65537) e calcula o expoente privado d como o inverso modular de e em relação a ϕ , utilizando uma implementação do Algoritmo Estendido de Euclides.

Por fim, as chaves são salvas em arquivos no formato PEM/Base64 para facilitar o armazenamento e o uso posterior. A implementação pode ser visualizada nas Figura 1, 2 e 3



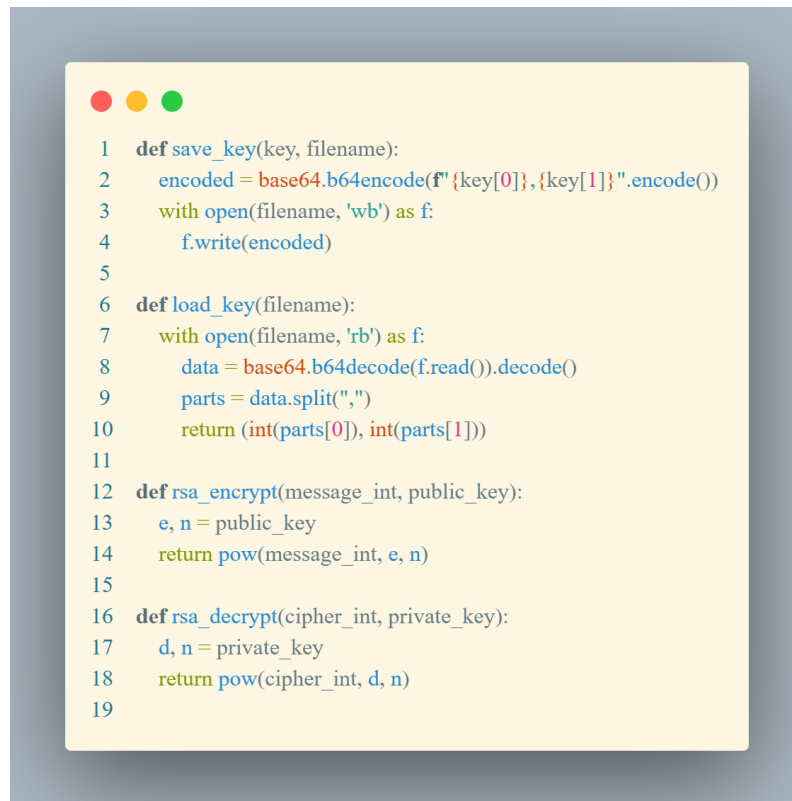
```
1 import random
2 from utils.miller_rabin import isPrime
3 from math import gcd
4 import base64
5
6 def generate_large_prime(bits=2048, k=10):
7     while True:
8         # Generates random number with LSB and MSB as 1
9         candidate = random.getrandbits(bits) | (1 << bits - 1) | 1
10        if isPrime(candidate, k): # Checks if is prime using Miller Rabin
11            return candidate
12
13 def extended_gcd(a, b):
14     if a == 0:
15         return (b, 0, 1)
16     else:
17         g, y, x = extended_gcd(b % a, a)
18         return (g, x - (b // a) * y, y)
19
20 def modinv(a, m):
21     g, x, y = extended_gcd(a, m)
22     if g != 1:
23         raise Exception("Inverso modular nao existe")
24     return x % m
```

Figure 1. Funções principais para geração de chaves RSA



```
1 def generate_keypair(bits=2048):
2     # Finds p
3     p = generate_large_prime(bits)
4     q = generate_large_prime(bits)
5     while p == q:
6         q = generate_large_prime(bits)
7
8     n = p * q
9     phi = (p - 1) * (q - 1)
10
11     e = 65537
12     if gcd(e, phi) != 1:
13         e = 3
14         while gcd(e, phi) != 1:
15             e += 2
16
17     d = modinv(e, phi)
18
19     public_key = (e, n)
20     private_key = (d, n)
21
22     return public_key, private_key
```

Figure 2. Funções principais para geração de chaves RSA - Parte 2



```

1  def save_key(key, filename):
2      encoded = base64.b64encode(f"{key[0]},{key[1]}".encode())
3      with open(filename, 'wb') as f:
4          f.write(encoded)
5
6  def load_key(filename):
7      with open(filename, 'rb') as f:
8          data = base64.b64decode(f.read()).decode()
9          parts = data.split(",")
10         return (int(parts[0]), int(parts[1]))
11
12 def rsa_encrypt(message_int, public_key):
13     e, n = public_key
14     return pow(message_int, e, n)
15
16 def rsa_decrypt(cipher_int, private_key):
17     d, n = private_key
18     return pow(cipher_int, d, n)
19

```

Figure 3. Funções principais para geração de chaves RSA - Parte 3

2.2. Parte 2: Assinatura

Para assinar uma mensagem, não basta apenas cifrar seu hash. É crucial utilizar um esquema de preenchimento (padding) para evitar ataques. O trabalho exigiu o uso do RSA-PSS, um padrão robusto e seguro. A função `sign_message` realiza o processo:

1. A mensagem original é processada pela função de codificação `emsa_pss_encode`.
2. Dentro desta função, um hash SHA-3 da mensagem é gerado. Um salt aleatório é criado para garantir que a assinatura seja probabilística.
3. A estrutura de dados do PSS é montada, e uma máscara é gerada pela função MGF1 (Mask Generation Function) para ofuscar parte da estrutura.
4. A mensagem codificada resultante (EM) é então "assinada" através da aplicação da operação RSA com a chave privada.

O resultado é uma assinatura binária, que é então codificada em Base64 e salva no arquivo `assinatura.sig`. O código desta etapa está nas Figuras 4, 5 e 6

```

1  import hashlib
2  import os
3  from parte_1 import rsa_decrypt
4  import base64
5
6  def mgf1(seed, mask_len, hash_func=hashlib.sha3_256):
7      # Generates mask of len mask_len using MGF1 based on SHA-3
8      counter = 0
9      output=b''
10
11     while len(output) < mask_len:
12         c = counter.to_bytes(4, byteorder='big')
13         output += hash_func(seed + c).digest()
14         counter += 1
15
16     return output[:mask_len]

```

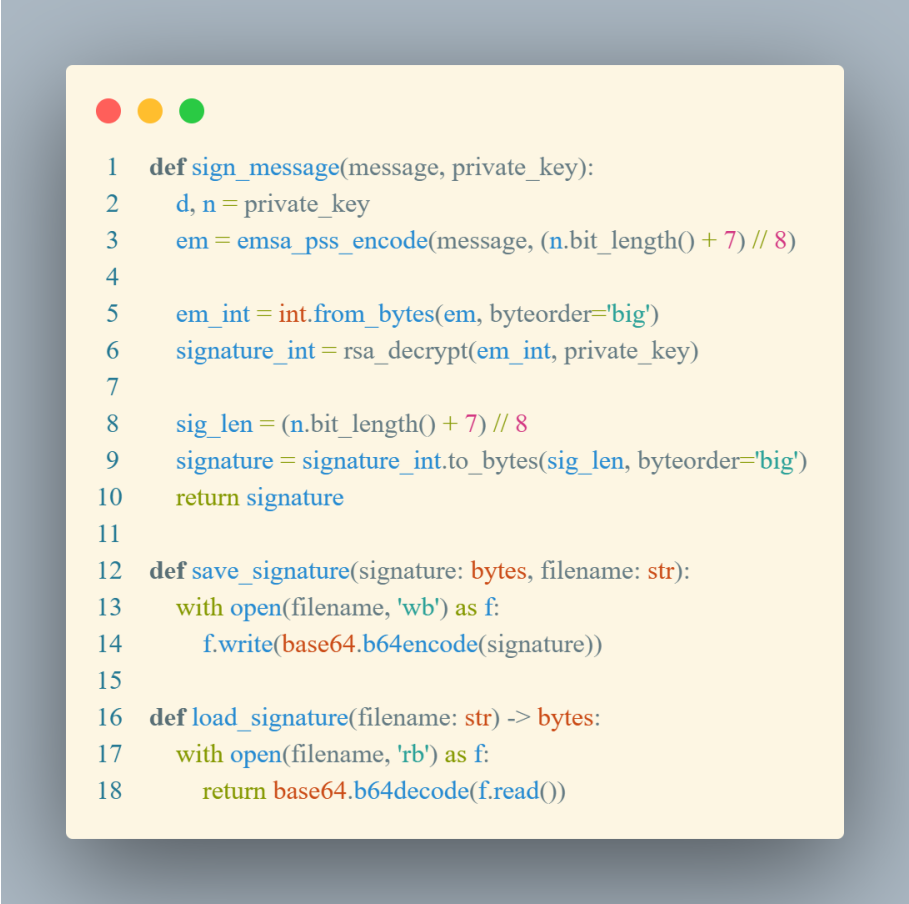
Figure 4. Implementação da assinatura com codificação RSA-PSS

```

1  def emsa_pss_encode(message, em_len, hash_func=hashlib.sha3_256, salt_len=32):
2      h_len = hash_func().digest_size
3      if em_len < h_len + salt_len + 2:
4          raise ValueError('em_len pequeno')
5
6      # Message hash
7      m_hash = hash_func(message).digest()
8
9      # Random salt
10     salt = os.urandom(salt_len)
11
12     m_prime = b'\x00' * 8 + m_hash + salt
13     h = hash_func(m_prime).digest()
14
15     # Generates mask
16     ps = b'\x00' * (em_len - salt_len - h_len - 2)
17     db = ps + b'\x01' + salt
18     db_mask = mgf1(h, em_len - h_len - 1, hash_func)
19     masked_db = bytes(x ^ y for x, y in zip(db, db_mask))
20
21     em = masked_db + h + b'\xbc'
22     return em

```

Figure 5. Implementação da assinatura com codificação RSA-PSS - Parte 2



```

1  def sign_message(message, private_key):
2      d, n = private_key
3      em = emsa_pss_encode(message, (n.bit_length() + 7) // 8)
4
5      em_int = int.from_bytes(em, byteorder='big')
6      signature_int = rsa_decrypt(em_int, private_key)
7
8      sig_len = (n.bit_length() + 7) // 8
9      signature = signature_int.to_bytes(sig_len, byteorder='big')
10     return signature
11
12  def save_signature(signature: bytes, filename: str):
13      with open(filename, 'wb') as f:
14          f.write(base64.b64encode(signature))
15
16  def load_signature(filename: str) -> bytes:
17      with open(filename, 'rb') as f:
18          return base64.b64decode(f.read())

```

Figure 6. Implementação da assinatura com codificação RSA-PSS - Parte 3

2.3. Parte 3: Verificação da Assinatura

A verificação é o processo inverso da assinatura. A função `verify_signature` carrega a mensagem original, a assinatura e a chave pública de quem está enviando. As etapas são:

1. A assinatura é decodificada de Base64 para seu formato binário.
2. A operação RSA com a chave pública é aplicada à assinatura para reverter a cifragem e obter a mensagem codificada (EM).
3. A função `emsa_pss_verify` é chamada para validar a EM. Ela reconstrói a estrutura PSS, extrai o salt, calcula um novo hash da mensagem original e o compara com o hash contido na EM.

Se ambos os hashes forem idênticos e toda a estrutura do preenchimento PSS for válida, a assinatura é considerada autêntica. Caso contrário, ela é rejeitada. A implementação da verificação pode ser vista nas Figuras 7 e 8.

```

1  import hashlib
2
3  from parte_1 import rsa_encrypt, load_key
4  from parte_2 import mgf1
5
6
7  def emsa_pss_verify(message, em, em_len, hash_func=hashlib.sha3_256, salt_len=32):
8      h_len = hash_func().digest_size
9      if em_len < h_len + salt_len + 2:
10         return False
11
12     if em[-1] != 0xbc:
13         return False
14
15     masked_db = em[:em_len - h_len - 1]
16     h = em[em_len - h_len - 1:-1]
17
18     # Remove mask
19     db_mask = mgf1(h, em_len - h_len - 1, hash_func)
20     db = bytes(x ^ y for x, y in zip(masked_db, db_mask))
21
22     # Ignores the highest bits of the first byte
23     num_unused_bits = 8 * em_len - (8 * em_len - 1)
24     db = bytes([db[0] & (0xFF >> num_unused_bits)]) + db[1:]
25
26     # Verifies padding and extracts salt
27     ps_len = em_len - h_len - salt_len - 2
28     if not db[:ps_len] == b'\x00' * ps_len or db[ps_len] != 0x01:
29         return False
30
31     salt = db[-salt_len:]
32
33     # Reconstructs h
34     m_hash = hash_func(message).digest()
35     m_prime = b'\x00' * 8 + m_hash + salt
36     h_prime = hash_func(m_prime).digest()
37
38     return h == h_prime

```

Figure 7. Implementação da verificação da assinatura

```

1  def verify_signature(message, signature, public_key):
2      e, n = public_key
3      em_len = (n.bit_length() + 7) // 8
4      if len(signature) != em_len:
5          raise ValueError("Tamanho da assinatura inválido.")
6
7      signature_int = int.from_bytes(signature, byteorder='big')
8      em_int = rsa_encrypt(signature_int, public_key)
9      em = em_int.to_bytes(em_len, byteorder='big')
10
11     return emsa_pss_verify(message, em, em_len)
12
13
14 def verify_file_signature(message_path, signature_path, pub_key_path):
15     import base64
16
17     public_key = load_key(pub_key_path)
18
19     with open(message_path, "rb") as f:
20         message = f.read()
21
22     with open(signature_path, "rb") as f:
23         signature = base64.b64decode(f.read())
24
25     valid = verify_signature(message, signature, public_key)
26     if valid:
27         print("Assinatura VÁLIDA!")
28     else:
29         print("Assinatura INVÁLIDA!")

```

Figure 8. Implementação da verificação da assinatura - Parte 2

3. Testes e Resultados

Para validar a corretude da implementação, foi criado um script de testes automatizados no arquivo `main.py`. Este script executa um ciclo completo de operações e verifica os resultados esperados:

1. **Geração de Chaves:** Um novo par de chaves é gerado e salvo em `public_key.pem` e `private_key.pem`.
2. **Assinatura:** Uma mensagem de teste é criada e assinada com a chave privada recém-gerada.
3. **Teste de Verificação Válida:** O script verifica se a assinatura gerada é válida para a mensagem original. O resultado esperado é a confirmação da validade.
4. **Teste de Verificação Inválida:** O script tenta verificar a mesma assinatura, mas com uma mensagem adulterada. O resultado esperado é a falha na verificação, provando que o sistema detecta qualquer modificação na integridade do dado.

Ambos os testes passaram com sucesso, confirmando que a implementação atende a todos os requisitos de funcionalidade. A saída do console com os resultados dos testes pode ser visualizada ao testar o código.

4. Conclusão

Este trabalho teve êxito em desenvolver uma ferramenta completa para geração e verificação de assinaturas digitais, cumprindo todos os requisitos da especificação. O PSS introduz aleatoriedade no processo através de um "sal" (salt). Essa característica o torna seguro e robusto contra ataques de falsificação, mesmo em cenários onde a função de hash possa apresentar fraquezas, garantindo um nível de segurança superior e alinhado aos padrões criptográficos modernos.

A decisão de utilizar chaves RSA de 2048 bits foi deliberada para refletir as melhores práticas de segurança atuais. Chaves de 1024 bits, embora ainda funcionais, são consideradas obsoletas para a maioria das aplicações nos dias atuais, pois se tornaram vulneráveis a ataques de fatoração com o aumento do poder computacional. Seguindo as recomendações de órgãos como o NIST (National Institute of Standards and Technology), a dimensão de 2048 bits oferece um balanço robusto entre a segurança contra os atuais ataques por computação clássica e um desempenho computacional aceitável para a geração e verificação das assinaturas.

A tecnologia implementada neste projeto não é meramente teórica, ela é o que garante a confiança em inúmeros sistemas digitais. É fundamental em certificados digitais (TLS/SSL) que protegem a navegação web, na assinatura de softwares para garantir sua procedência e integridade, na segurança de transações financeiras e na validação de documentos eletrônicos governamentais. Portanto, o projeto não apenas cumpriu um requisito acadêmico, mas também proporcionou uma experiência prática e profunda em uma tecnologia que é fundamental para a segurança da infraestrutura digital global.