Assinatura Digital utilizando RSA-PSS

Lucas Gabriel de Oliveira Lima - 231003406
Pedro Lucas Pereira Neris - 231018964

18 de julho de 2025

Universidade de Brasília (UnB)
Departamento de Ciência da Computação (CIC)
Disciplina: Segurança Computacional
Semestre: 2025.1
Professora: Lorena de Souza Bezerra Borges

Resumo

Este trabalho tem como objetivo implementar o algoritmo de geração e verifação de assinaturas digitais tendo como base o algoritmo de chave pública RSA-PSS. Será apresentada a implementação do programa capaz de gerar gerar, serializar e armazenar as chaves, além de verificar a autenticidade de documentos/textos assinados utilizando este algoritmo.

1 Introdução

Com a crescente digitalização de processos e documentos, surge a necessidade da criação de ferramentas no mundo eletrônico que simulem ou até alcancem as mesmas funcionalidades de ferramentas do mundo físico. Uma destas ferramentas é a assinatura, utilizada para comprovar autoria ou concordância de um documento por parte de uma pessoa (física ou jurídica). Com isso, surge a assinatura digital, ferramenta que fornece todos os benefícios de uma assinatura física e até mais benefícios.

A assinatura digital utiliza uma chave privada gerada para um indivíduo para autenticar os documentos que ele deseja assinar. Com a chave pública deste indivíduo, é possível verificar se um documento foi realmente assinado por quem ele diz ser. Para fornecer estas funcionalidades, podem ser utilizados diversos algoritmos criptográficos que utilizam criptografia assimétrica (uma chave privada, conhecida apenas por um agente, e uma chave pública, conhecida pela rede), como o RSA e o RSA-PSS.

Neste trabalho, adentramos no funcionamento do RSA-PSS (que utiliza o RSA como base) e sua implementação prática, mostrando como ele pode ser utilizado para autenticação de documentos e verificação de identidade. Todos os códigos do projeto estão disponíveis no seguinte repositório: https://github.com/lucasdbr05/digital-signature-with-rsa-pss.

2 RSA

O RSA (Rivest–Shamir–Adleman) é um dos algoritmos de criptografia assimétrica mais utilizados no mundo. Ele foi desenvolvido em 1977 e baseia-se na dificuldade de fatorar grandes números primos, o que garante sua segurança.

O funcionamento do RSA envolve duas chaves: uma chave pública, usada para criptografar mensagens, e uma chave privada, usada para descriptografá-las.

O algoritmo utiliza propriedades algébricas para esse processo. Para isso definimos as seguintes chaves:

- p, q: chaves privadas primas
- n: chave pública resultante de $p \cdot q$
- e: chave pública de expoente na encriptação
- d: chave privada de expoente na decriptação

2.1 Teste de Primalidade

Apesar de chamarmos de teste de primalidade, por trabalharmos com números muito grandes, não conseguimos fatorar esse número em tempo computacional satisfatório o suficiente para checar se ele realmente é primo.

No entanto, utilizamos o paradoxo do aniversário para checar que a probabilidade desse número ser composto seja muito próxima de 0, ou seja, que tenha uma grande probabilidade de ser primo.

Para isso, o teste se utiliza de um teorema que enuncia que, para um número n grande, a cardinalidade do conjunto de números primos p tais que $1 \leq p \leq n$ é aproximadamente $n/\ln(n)$.

Ou seja, para um número de 1024 bits, temos que a probabilidade de o teste falhar é de aproximadamente $1/(1024 \cdot \ln(2))$.

O teste em si funciona da seguinte forma: tomando como base o Teorema de Fermat, se n é primo, $n-1$ pode ser escrito na forma $2^t \cdot q$, tal que q é ímpar. Portanto, se n for um número primo, podemos selecionar um número a qualquer pertencente a $(1, n)$ e uma das duas condições precisam ser atendidas:

- $a^q \equiv 1 \pmod{n}$
- existe $i \in [0, t-1]$ tal que $a^{q \cdot 2^i} \equiv -1 \pmod{n}$

```
    def miller_composite_test(self, a : int, n : int, t : int, q : int) -> bool
        :
        r = fast_exponentiation(a, q, n)

        if (r == n-1 or r == 1):
            return False

        for r in range(1, t):
            r = (r * r) % n
            if (r == n-1):
                return False
        return True
```

2.2 Teste de Miller-Rabin

O teste de Miller-Rabin utiliza-se do teorema de Rabin, que enuncia que se um número n é composto, ao menos 75% dos números no intervalo $(1, n)$ são testemunhas da não primalidade de n ao aplicarmos o teste de primalidade explicado acima.

Então, ao aplicarmos o teste de primalidade em 40 iterações, com a probabilidade de o teste falhar de $1 - 0{,}75 = 0{,}25 = \frac{1}{4}$, temos que a probabilidade do teste de Miller-Rabin falhar é de $(\frac{1}{4})^{40}$.

O teste funciona da seguinte forma: é passado um número n, decompomos e realizamos 40 testes de primalidade, passando números a distintos. Se em algum dos testes retornar que o número é composto, temos certeza que o número não é primo. Caso contrário, assumimos que ele é primo.

```
    def miller_rabin(self, n: int, its: int = 40) -> bool:
        if n <= 4:
            return (n == 2 or n == 3)

        (t, q) = self.simplify_n_minus_1(n)
        for _ in range(its):
            a = randint(2, n-2)
            if (self.miller_composite_test(a, n, t, q)):
                return False
        return True
```

2.3 Geração das Chaves p e q

Para gerar p e q, tomamos o seguinte processo: aleatoriamente selecionamos um número ímpar qualquer entre 2^{1023} e $2^{1024} - 1$, para garantirmos que o seu bit mais significativo seja o 1024^{o} bit. Então, vamos iterando pelos números ímpares da sequência iniciada por esse número selecionado até encontrarmos um número que aponte ser primo no teste de Miller-Rabin.

Para encontrar esse número, são feitas em média $1024 \cdot \ln(2)/2 \approx 305$ tentativas até encontrar um número possivelmente primo.

Com isso, a probabilidade da geração de um número primo falhar acaba sendo de aproximadamente $305 \cdot 4^{-40}$, o que é uma probabilidade extremamente baixa de erro.

```
    # Metodo usado para gerar p e q
    def get_random_prime(self, msb: int = 1024) -> int:
        p = randint(1 << (msb-1), (1 << (msb)) - 1)
        if p % 2 == 0:
            p += 1
        while True:
            if self.miller_rabin(p):
                return p
            p += 2
            if p >= (1 << (msb)):
                p = (1 << (msb-1)) + 1
```

2.4 Chave Pública n e seu totiente

A chave pública n é dada por:

$$n = p \cdot q$$

n será usado diretamente como módulo nos processos de encriptação e decriptação.

Por p e q serem números primos, e o totiente de Euler ser uma função multiplicativa, temos que:

$$\phi(n) = (p-1) \cdot (q-1)$$

$\phi(n)$ será usado diretamente como para a geração das chaves e e d.

2.5 Geração das Chaves e e d

Tomando a chave n já criada, temos que e precisa ser um número tal que $\gcd(e, \phi(n)) = 1$. Portanto, começamos com o número 65537, que frequentemente é adotado no mercado para aplicações com RSA, por ser um número primo suficientemente grande, além de ser um número de Fermat (ou seja, tal que $F_n = 2^{2^n} + 1$), que traz propriedades interessantes para ser usado no RSA.

Caso o número inicial não seja coprimo a $\phi(n)$, vamos iterando até encontrar um número que tenha essa propriedade.

Após encontrarmos e, temos que d é um número tal que:

$$e \cdot d \equiv 1 \pmod{\phi(n)}$$

Ou seja, d é o inverso modular multiplicativo de e. Para encontrar d, utilizamos o algoritmo de Euclides estendido.

```
    # Metodo usado para encontrar o expoente publico 'e'
    def gen_public_exponent(self, phi_n:int) -> int:
        e = 65537

        while gcd(e, phi_n) != 1:
            e += 2
            if e >= phi_n:
                e = 3
        return e

    # Metodos usados para encontrar 'd', ou seja, o inverso mutiplicativo de 'e
        '
    def multiplicative_inverse(self, e: int, phi_n:int) -> int:
        (g, inv_e, d) = self.extended_gcd(e, phi_n)
        return inv_e % phi_n

    def extended_gcd(self, a: int, b: int)-> tuple[int, int, int]:
        x, y = 1, 0
```

```
        aux_x, aux_y = 0, 1
        aux_a, aux_b = a, b

        while (aux_b) :
            q = aux_a // aux_b
            (x, aux_x) = (aux_x, x - q * aux_x)
            (y, aux_y) = (aux_y, y - q * aux_y)
            (aux_a, aux_b) = (aux_b, aux_a - q * aux_b)
        return (aux_a, x, y)
```

2.6 Encriptação e Decriptação

Para esse processo, utilizamos o algoritimo de exponênciação rápida para executar as operações de potência em complexidade $O(log(n))$, ao invés de ser feito em complexidade $O(n)$ como nos algoritmos clássicos ().

```
def fast_exponentiation(b:int, e:int, MOD: int) ->int:
    value = 1
    b %= MOD
    while (e > 0) :
        if (e & 1):
            value = value * b % MOD
        b = (b * b) % MOD
        e >>= 1

    return value
```

Com todas as chaves definidas, temos que a encriptação de um bloco M, que gera um bloco cifrado C, é dada por:

$$C = M^e \mod n$$

```
    def encrypt(self, M: int) -> int:
        n = self.load_pem_key(self.key_paths["n"])
        e = self.load_pem_key(self.key_paths["e"])
        return fast_exponentiation(M, e, n)
```

E a decriptação de um bloco cifrado C, que gera um bloco original M, é dada por:

$$M = C^d \mod n$$

```
    def decrypt(self, M: int) -> int:
        n = self.load_pem_key(self.key_paths["n"])
        inv_e = self.load_pem_key(self.key_paths["inv_e"])
        return fast_exponentiation(M, inv_e, n)
```

2.7 Assinatura Digital

A assinatura digital utiliza o algoritmo e as chaves descritas acima da seguinte forma:

1. Para uma dada mensagem m, é calculado o hash $h = H(m)$. Geralmente é utilizado o algoritmo SHA-256 ou alguma variação dele. Depois disso

$$assinatura = h^d \pmod n.$$

2. A assinatura então é verificada por quem recebe fazendo o cálculo com a chave pública

$$h' = assinatura^e \pmod n$$

E confere se $h' = H(m)$

2.8 Análise de segurança

3 RSA-PSS

O esquema RSA-PSS (Probabilistic Signature Scheme) é uma variação do algoritmo RSA tradicional para assinatura digital, que introduz aleatoriedade e técnicas de probabilidade no processo de assinatura para aumentar a segurança, sendo considerado mais seguro contra ataques modernos do que o RSA padrão.

A nossa implementação utilizando o esquema de assinatura digital RSA-PSS funciona da seguinte forma: é feita uma rodada de RSA na primeira etapa do projeto para criptografia e decriptografia, são aplicadas aplica funções de *hash* e geração de máscara (MGF1) para construir a assinatura RSA-PSS.

3.1 Diferenças em relação ao RSA tradicional

- **Determinismo x Aleatoriedade:** O RSA tradicional usa a operação ($assinatura = H(m)^d \bmod n$) para assinatura, que torna cada assinatura idêntica para uma mesma mensagem. Já o RSA-PSS introduz aleatoriedade a partir de funções de padding probabilístico que garante assinaturas diferentes mesmo para mensagens iguais.
- **Esquema de padding:** O RSA-PSS utiliza um padding baseado em máscara, derivado do hash da mensagem. Esse processo protege contra ataques de estrutura.
- **Segurança provada:** Diferente do RSA tradicional com PKCS#1 v1.5, o RSA-PSS é provadamente seguro sob suposições criptográficas. [MKJR16]

Assinatura

A assinatura de uma mensagem é feita da seguinte da seguinte forma:

1. Calcula o hash da mensagem com SHA-256.
2. Gera um *salt* aleatório de tamanho definido.
3. Concatena o hash da mensagem com o salt e aplica a função MGF1 para gerar o padding `PS`.
4. Constrói a codificação EM: `0x00 || 0x01 || PS || 0x00 || H(M) || SALT`.
5. Converte EM para inteiro e aplica a operação de descriptografia RSA (com chave privada) para gerar a assinatura.
6. Codifica a assinatura e o salt em base64 para facilitar o armazenamento/transmissão.

```
    def sign(self, message: bytes) -> tuple[str, str]:
        """
        EM = 0x00 || 0x01 || PS || 0x00 || H(M) || SALT
        """
        em_len = self.get_em_length()

        hlen = hashlib.new(self.hash_alg).digest_size
        message_hash = hashlib.new(self.hash_alg, message).digest()
        salt = os.urandom(self.salt_len)

        ps_len = em_len - hlen - self.salt_len - 3
        ps = self.mgf1(message_hash + salt, ps_len, self.hash_alg)

        em = b'\x00' + b'\x01' + ps + b'\x00' + message_hash + salt
        em = int.from_bytes(em, byteorder='big')

        # Use RSA decrypt to sign with private key
        signature = self.rsa.decrypt(em)

```

```
        signature_b64 = base64.b64encode(signature.to_bytes(em_len, byteorder='
            big')).decode()
        salt_b64 = base64.b64encode(salt).decode()
        return (signature_b64, salt_b64)
```

Verificação

A verificação da assinatura é feita da seguite forma:

1. Decodifica a assinatura e o salt de base64.
2. Aplica a operação de criptografia RSA (com chave pública) sobre a assinatura para recuperar EM.
3. Verifica a estrutura de EM e extrai o padding, hash e salt.
4. Recalcula o hash da mensagem e o padding esperado com MGF1.
5. Compara os valores extraídos com os valores esperados. Se todos coincidirem, a assinatura é válida.

```
    def verify(self, message: bytes, signature_b64: str, salt_b64: str) -> bool
        :
        em_len = self.get_em_length()

        hlen = hashlib.new(self.hash_alg).digest_size
        m_hash = hashlib.new(self.hash_alg, message).digest()

        signature_bytes = base64.b64decode(signature_b64)
        salt = base64.b64decode(salt_b64)

        signature = int.from_bytes(signature_bytes, byteorder='big')

        # Use RSA encrypt to verify with public key
        em = self.rsa.encrypt(signature)
        em = em.to_bytes(em_len, byteorder='big')

        if not (em[0] == 0x00 and em[1] == 0x01):
            return False
        ps_len = em_len - hlen - self.salt_len - 3
        ps = em[2:2+ps_len]
        ps_check = self.mgf1(m_hash + salt, ps_len, self.hash_alg)
        if ps != ps_check:
            return False
        if em[2+ps_len] != 0x00:
            return False
        m_hash2 = em[3+ps_len:3+ps_len+hlen]
        salt2 = em[3+ps_len+hlen:]
        return m_hash2 == m_hash and salt2 == salt
```

Função MGF1

A função `mgf1` é usada para gerar uma máscara determinística a partir de um *seed*, utilizando SHA-256. Ela é importante para gerar o padding pseudoaleatório no esquema PSS.

```
    def mgf1(self, seed: bytes, mask_len: int, hash_alg: str = 'sha256') ->
        bytes:
        hlen = hashlib.new(hash_alg).digest_size
        mask = b''
        for i in range((mask_len + hlen - 1) // hlen):
            c = i.to_bytes(4, byteorder='big')
            mask += hashlib.new(hash_alg, seed + c).digest()
        return mask[:mask_len]
```

RSA-PSS melhora a segurança das assinaturas RSA ao introduzir aleatoriedade e um padding mais robusto. Seu uso é recomendado por padrões como o PKCS#1 v2.2 e pelo NIST em aplicações que exigem alta resistência a ataques criptográficos modernos [Nat].

4 Implementação

4.1 Fluxo da aplicação

O algoritmo RSA-PSS foi utilizado em uma interface implementada com a biblioteca *Streamlit* do Python. Ela possui uma única tela, com as funcionalidades de assinar e verificar assinaturas, tanto de arquivos como de textos. Instruções sobre como executar o projeto e todos os códigos-fonte estão no repositório do projeto.

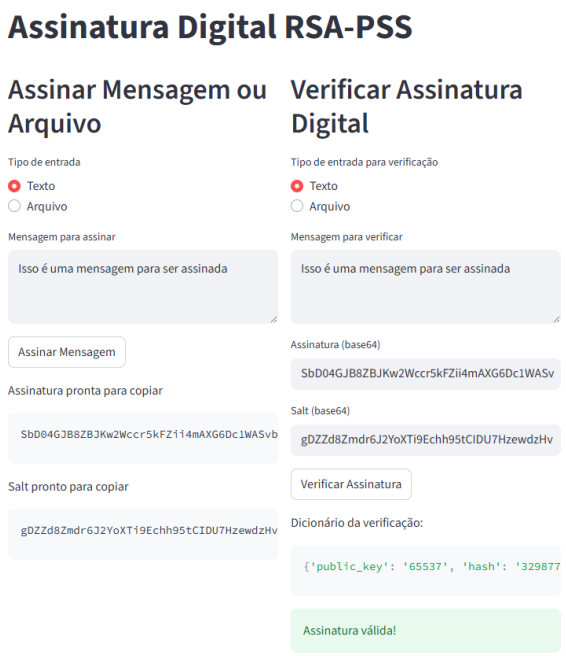

Figura 1: Interface com sucesso via texto

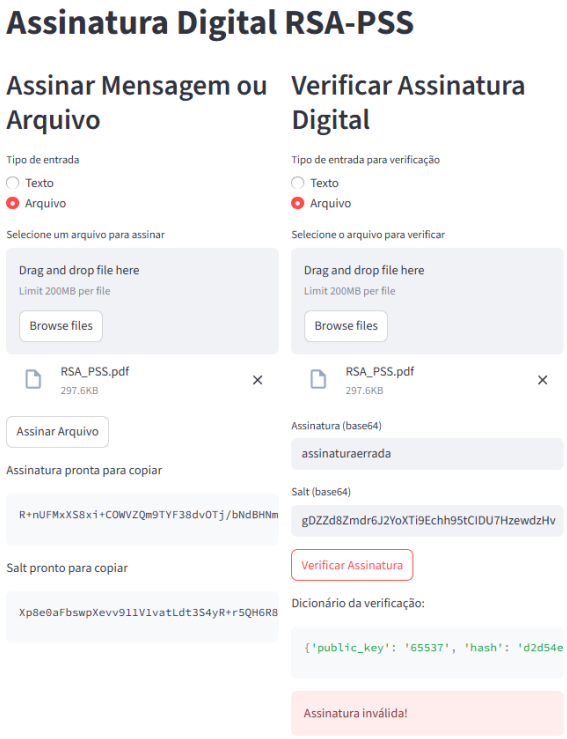

Figura 2: Interface sem sucesso via arquivo

- **Assinatura:** Ao inserir um arquivo ou texto na entrada e clicar no botão de assinar, o conteúdo é enviado para o método *sign* da classe *RSAPSS*. O sistema registra então a assinatura digital e o salt, ambos em base64 e exibe para o usuário.
- **Verificação:** O usuário escolhe entre verificar um arquivo ou texto. Ele então insere a assinatura e o salt (ambos em base64) e clica em verificar. O sistema envia o conteúdo para o método *verify* da classe *RSAPSS* para validar a assinatura. A interface devolve um dicionário da verificação, contendo a chave pública e o hash da assinatura do conteúdo e uma mensagem confirmando a validade da assinatura.

5 Análise de segurança

Como para decriptar a chave privada de algum indivíduo que foi gerada utilizando RSA, seria necessário fatorar n (chave pública), o que se mostra muito custoso computacionalmente. Além disso, o RSA-PSS ainda adiciona uma geração de chaves probabilística (para uma mesma entrada, podem ser geradas chaves diferentes), adicionando uma camada a mais de incerteza e segurança criptográfica. Portanto, o RSA-PSS se mostra um algoritmo seguro para a utilização em assinaturas digitais. Quanto maior forem os números primos utilizados como parâmetro no algoritmo, mais difícil de quebrá-lo ele se torna.

Vale ressaltar que, no projeto utilizamos chave p e q de 1024 bits unicamente por questão de performance. Apesar disso conferir um grau considerável de segurança, em aplicações reais são aconselháveis chaves de 2048, 3072 ou 4094 bits. Isso demonstra também um certo *trade-off* entre performance e um grau ainda mais elevado de segurança.

Apesar da robustez do RSA na computação clássica, é importante destacar que sua segurança se baseia na dificuldade de fatorar números grandes compostos por dois primos. Atualmente, essa operação é considerada inviável para computadores convencionais, pois requer tempo exponencial à medida que o tamanho dos primos aumenta. No entanto, o surgimento da computação quântica representa uma possível vulnerabilidade futura. Diferentemente dos bits tradicionais, os qubits podem existir em superposição de estados 0 e 1, permitindo que computadores quânticos processem múltiplas possibilidades simultaneamente. Esse comportamento viabiliza algoritmos como o de Shor, proposto em 1994, que é capaz de fatorar inteiros grandes em tempo polinomial. Caso computadores quânticos em larga escala se tornem realidade, algoritmos como o RSA poderão ser quebrados com relativa facilidade, comprometendo a confidencialidade e autenticidade de sistemas criptográficos baseados nesse esquema.

6 Conclusão

Neste trabalho, implementamos um esquema de assinatura digital utilizando o algoritmo de criptografia assimétrica RSA-PPS, uma evolução do algoritmo RSA, que introduz probabilidade na geração de chaves. Como citado no trabalho, o algoritmo RSA-PSS é seguro para a utilização em assinaturas digitais, devido à complexidade necessária para descobrir as chaves privadas de um usuário desse algoritmo (sendo ainda mais custoso que o RSA padrão por não ser determinístico).

Além da implementação das funcionalidades básicas para criação de uma assinatura digital, foi desenvolvida uma interface gráfica que permite interagir e testar nossa implementação do algoritmo. Com isso, os conhecimentos teóricos de segurança e criptografia puderam ser aplicados praticamente.

Referências

[MKJR16] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, RFC Editor, November 2016. Acesso em: 17 de julho de 2025.

[Nat] National Institute of Standards and Technology (NIST). Cryptographic algorithm validation program — digital signatures. https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/digital-signatures. Acesso em: 17 de julho de 2025.