Segurança Computacional
Trabalho 03

Eduardo Pereira de Sousa - 231018937
Luca Heringer Megiorin - 231003390
Universidade de Brasília

17 de Julho, 2025

1 Introdução

Com a expansão da computação no cotidiano da população, a necessidade de garantir a autenticidade e integridade de documentos digitais e transações se fez presente no contexto da tecnologia. A fim de assegurar que as mensagens não fossem adulteradas e permitir identificar inequivocavelmente o responsável por tal mensagem, foi criada a assinatura digital.

Para este terceiro trabalho de Segurança Computacional, a dupla desenvolveu um gerador e verificador de assinaturas, utilizando o RSA-PSS para assinar e validar arquivos diversos. Neste documento, serão abordados os processos presentes no algoritmo RSA (seção 2), bem como o fluxo de assinatura do RSA-PSS (seção 3). Os códigos em Python desenvolvidos estão disponíveis no GitHub (vide o apêndice A).

2 RSA

O RSA (Rivest-Shamir-Adleman) é um algoritmo de criptografia assimétrica amplamente utilizado em sistemas de segurança digital. Desenvolvido em 1977 por Ronald Rivest, Adi Shamir e Leonard Adleman, a segurança do algoritmo é baseada na dificuldade de se fatorar números inteiros muito grandes, especialmente o produto de dois primos [1]. Este algoritmo não só pode ser usado de forma a criptografar e esconder mensagens, mas também pode ser usado no contexto de assinatura digital.

O funcionamento do RSA envolve a geração de um par de chaves: uma pública, utilizada para encriptação ou verificação de assinaturas, e uma chave privada, utilizada para a decriptação ou geração da assinatura. No caso da criptografia, a função de encriptação recebe a chave pública do receptor e a de decriptação, a privada do receptor, enquanto que, para assinar, usa-se a chave privada do assinante para encriptar a mensagem (gerar a assinatura) e a chave pública do mesmo para decriptar (parte da verificação da assinatura).

2.1 Geração de Chaves

As chaves são compostas por um expoente (público e, ou privado d) e o valor de n, utilizado como módulo em todas as operações. O valor de n é obtido a partir do produto de dois grandes números primos p e q, que são gerados aleatoriamente. Como o tamanho de n afeta na segurança, escolheu-se ter n como um número de 2048 bits, ou seja, os primos p e q devem ser de 1024 bits (mais sobre sua geração na seção 2.1.1). Com base neles, calcula-se a função totiente de Euler, definida por $\phi(n) = (p-1) \times (q-1)$.

Escolhe-se, então, o expoente público e, $1 < e < \phi(n)$ e coprimo de $\phi(n)$ (isto é, $mdc(e, \phi(n)) = 1$). Apesar de poder ser qualquer valor que respeite tais condições, o valor mais utilizado na prática é $e = 65537$, por ser uma opção que oferece eficiência e segurança, visto que, por se tratar de um número de Fermat ($2^{2^4} + 1$), é um número de fácil manipulação para as operações computacionais que o utilizam. Como os primos são gerados aleatoriamente, deve-se checar se, realmente, $mdc(e, \phi(n)) = 1$ e, caso e não seja coprimo de $\phi(n)$, gerar novos primos p e q até que essa condição seja satisfeita, já que e tem o seu valor fixo nesta implementação.

Após garantir que e é coprimo de $\phi(n)$, calcula-se n a partir de p e q, e, a partir do expoente público, o expoente privado d, sendo este o inverso multiplicativo de e módulo $\phi(n)$, satisfazendo a congruência $d \cdot e \equiv 1 \pmod{\phi(n)}$. Os valores então de n, e, d são retornados como as partes das chaves, que, no programa deste trabalho, são processados pela função controladora (`generate_keys`, presente no arquivo `operations.py`, vide apêndice A), a qual irá armazenar os valores em um arquivo de chave.

Vale notar que, para evitar desnecessariamente retornar p e q, checa-se se a chave deve ser serializada no formato PEM (dado que a biblioteca utilizada para essa parte necessita desses valores para serializar a chave privada) e, se não for o caso, não retorna tais valores sensíveis (mais disso abordado na seção 2.1.2). A implementação da lógica de geração de chaves pode ser vista no código 1:

```python
def rsa_generate_keys(choice = 1) -> tuple[int,int,int]:
    e = 65537
    while True:
        p = generate_prime()
        q = generate_prime()
        # e has to be coprime with phi, so the gdc is checked
        phi = (p - 1) * (q - 1)
        if math.gcd(e, phi) == 1:
            break
    # Primes have been chosen, get n
    n = p * q
    d = int(Mod(e, phi).inverse)
    if choice == 1: return (e,d,n,None,None)
    else: return (e,d,n,p,q)
```

Código 1: Implementação do código de geração de chaves RSA, presente no arquivo rsa.py.

2.1.1 Geração de Primos e Verificação de Primalidade com Miller-Rabin

Como parte da segurança do RSA depende da dificuldade de se fatorar o módulo n, é fundamental que os números p e q utilizados na geração das chaves sejam primos, dado que isso dificulta ainda mais a fatoração de n. Porém, verificar se um número muito grande é primo com métodos tradicionais pode ser computacionalmente inviável. Por isso, o algoritmo de Miller-Rabin é amplamente utilizado como um teste probabilístico de primalidade, oferecendo alta confiabilidade com um bom desempenho.

O teste não prova de forma determinística que um número é primo, mas identifica com grande probabilidade se um número é composto, dependendo de quantas vezes for repetido [1]. O algoritmo funciona reescrevendo um número N qualquer a ser testado como $N-1 = 2^S \times D$, com D sendo ímpar. Em seguida, é escolhida uma base aleatória A, escolhida entre 2 e $N-1$, e verifica-se a congruência $A^D \pmod N$. Se essa congruência ou alguma de suas potências sucessivas $A^{2^R \times D} \pmod N$ resultar em $1 \pmod N$ ou $-1 \pmod n$ (ou seja, $N-1 \pmod N$), então N passa no teste para aquela base, e caso contrário, o número é dado como composto.

Esse processo é repetido com várias bases A, geralmente de forma aleatória ou com um conjunto pré-definido, para reduzir a chance de erro. No caso do projeto, o processo é repetido 64 vezes, garantindo uma chance praticamente certa de definir corretamente se o número é primo ou não. Também foi decidido que seria gerado um A entre 2 e $N-2$, pois é certo que $(N-1)^D \pmod N$ é garantido ser igual a $1 \pmod N$ ou $N-1 \pmod N$, o que seria irrelevante para os testes.

Além do teste de primalidade de Miller-Rabin, foram feitas outras decisões para agilizar o cálculo dos números primos antes de checar a primalidade, dentre elas garantir que o número gerado é ímpar e de 1024 bits, com os dois bits mais significativos forçados para 1, a fim de ter certeza que a multiplicação entre eles daria um número de 2048 bits (linhas 4 a 6 no código 2), e checar se o número gerado é divisível por algum primo pré-calculado (linha 7 código 2).

A Tabela 1 abaixo mostra como a otimização com primos pré-gerados afeta o tempo médio de geração de números e, por padrão, o programa usa 500 números primos, dado que forneceu a melhor média de execução. Com isso, não é perdido tempo com números que não condizem com as necessidades do algoritmo RSA e não é necessário depender de Miller-Rabin em todo número gerado, pois este é um algoritmo custoso.

	0 primos	**250 primos**	**500 primos**	**750 primos**	**1000 primos**
Execução 1	794.7295 ms	586.2654 ms	950.0675 ms	635.2132 ms	732.0134 ms
Execução 2	722.2315 ms	1684.0656 ms	581.9469 ms	720.0001 ms	949.8194 ms
Execução 3	1115.1788 ms	485.58 ms	661.9075 ms	596.8753 ms	741.2017 ms
Execução 4	610.3267 ms	674.0245 ms	628.8119 ms	737.6826 ms	1347.2036 ms
Execução 5	957.3981 ms	540.9086 ms	474.8103 ms	757.8381 ms	596.1561 ms
Média das 5 execuções	**839.97292 ms**	**794.16882 ms**	**659.50882 ms**	**689.52186 ms**	**873.27884 ms**

Tabela 1: Tempo de 5 execuções de geração de chave, dependendo do tamanho da lista de primos pré-gerados, com o tempo em negrito indicando a média das execuções

A geração dos primos utiliza da biblioteca nativa do Python, Secrets, que faz uso da geração verdadeiramente aleatória de números, a partir de fatores de entropia capturados pelo sistema (como ruídos de hardware). O algritmo para gerar números primos aleatórios pode ser visto no código 2 abaixo:

```
NBITS = 1024 # Lenght needed for each prime number
def generate_prime() -> int:
    while True:
        n = secrets.randbits(NBITS)
        n |= (1 << (NBITS - 1))
        n |= (1 << (NBITS - 2))
        n |= 1
        if not any(n % prime == 0 for prime in get_first_n_primes()):
            if miller_rabin(n): return n

def miller_rabin(number) -> bool:
    if number <= 2: return False
    d = number - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1
    for _ in range(64):
        a = random_in_range(2, number - 2)
        x = pow(a, d, number)
        if x == 1 or x == number - 1:
            continue
        for _ in range(s - 1):
            x = pow(x, 2, number)
            if x == 1:
                return False
            if x == number - 1:
                break
        else:
            return False
    return True
```

Código 2: Implementação do código para padding, presente no arquivo rsa.py.

2.1.2 Armazenamento das Chaves

O armazenamento de chaves também é uma parte importante para evitar forjas de assinatura. Por simplicidade, a implementação neste projeto para guardar as chaves foi apenas a codificação das partes das chaves (módulo n, expoente público e,e expoente privado d) em base 64 e a inserção dos conjuntos (n,e) e (n,d) em arquivos de texto com a extensão `.custom_key`. Estes arquivos não seguem o padrão de serialização PEM, em razão da complexidade de implementar tal forma de serialização manualmente. Todavia, com o uso da biblioteca Cryptography [4] (apenas usada para essa parte e para o cálculo do hash), foi dada a possibilidade para o usuário guardar suas chaves no formato simplificado ou no formato PEM.

O uso do formato PEM é ideal para a distribuição de certificados digitais (distri-

buição das chaves públicas, por exemplo), dado que é um padrão utilizado por muitas bibliotecas e, portanto, por diversos serviços. Todavia, caso a chave privada não seja criptografada antes de ser serializada, continua exposta, de forma que a responsabilidade fica para o usuário de garantir que ninguém terá acesso à sua chave. Isso é comum em sistemas como o PJe, em que é necessária uma chave externa (geralmente em um pen-drive) para assinar, mas implica na vulnerabilidade caso esta seja perdida.

Outro ponto interessante seria o uso de Autoridades Certificadoras para manejar as chaves públicas, dado que, quanto menos responsabilidade for atribuída ao usuário, melhor a capacidade de um sistema garantir a sua segurança, já que haverá menos fatores para contribuir para riscos de segurança. No caso do trabalho, o usuário pode escolher qualquer chave pública e privada que ele desejar, dado que, como o projeto foi feito com visão acadêmica, visou-se a simplicidade, e esses foram apenas tópicos de discussão, e nada mais foi feito quanto ao armazenamento das chaves além do que foi dito no início desta seção.

Vale notar que, como essa biblioteca precisa ter conhecimento dos valores de p e q para serializar a chave privada, isso pode causar um possível risco ao sistema, visto que os valores desses primos circulam o código por um tempo maior do que o de geração de chaves. Isso, porém, não seria um risco em um sistema real, pois, usando as implementações do RSA diretamente da biblioteca permitem que esses valores não sejam necessários na hora de serializar a chave privada, ou seja, essa vulnerabilidade descrita é apenas causada por uma necessidade de funcionamento deste programa específico, que não usa da implementação do RSA presente na Cryptography [4].

2.2 Encriptação e Decriptação

A parte mais simples em quesito de código é a encriptação e decriptação com o RSA. No caso da assinatura, que foi o tópico do projeto, a chave privada é usada na encriptação (assinatura) e a chave pública é usada na decriptação (verificação), como pode ser visto abaixo (em que A é a assinatura gerada a partir de uma mensagem M):

$$A = M^d \pmod{n}$$

$$M = A^e \pmod{n}$$

Este processo pode ser lento, dado que trabalha com potências de números extremamente grandes. Todavia, são esses números grandes que garantem a segurança do algoritmo, pois tentar fatorar n para achar p e q e, portanto, achar o expoente privado, é um processo computacionalmente impossível com a tecnologia atual. Dessa forma, escolher uma quantidade grande de bits para n é essencial e, com o avanço da tecnologia, o tamanho padrão de 1024 bits para n já não é mais considerado seguro e, portanto, são indicados números de tamanho maior ou igual a 2048 bits. No caso desse projeto, decidiu-se usar o tamanho de 2048 bits, como dito anteriormente.

3 RSA-PSS

O PSS (*Probabilistic Signature Scheme*) trata-se de uma forma de atribuir maior entropia no sistema de assinatura com o RSA. Enquanto que o seu antecessor, o PKCS#1 v1.5, aplicava um padding fixo à mensagem de hash que seria assinada no RSA, o PSS utiliza valores aleatórios, como o salt, para garantir que até a assinatura de uma mesma mensagem seria diferente a cada vez, aumentando, assim, a dificuldade de realizar ataques de forja de assinatura. Nesse esquema, após a mensagem passar por um algoritmo de hash, ela recebe um padding e é encriptada com a chave privada do assinante, por meio do RSA, gerando uma assinatura. Posteriormente, outra parte pode realizar o processo de verificação do PSS para ver a validade da assinatura.

3.1 PSS Encoding

Seguindo a RFC 8017 [2], o processo do padding PSS diferiu daquele apresentado nas aulas (EM = 0x00 || 0x01 || PS || 0x00 || H(M) || Salt), o qual possui uma estrutura próxima do PKCS#1 v1.5 (EB = 00 || BT || PS || 00 || D) [3], apesar de não existir salt nesta forma de padding. Por esta RFC [2], o formato obtido para o padding PSS é o seguinte:

$$EM = maskedDB \,||\, H \,||\, 0xbc$$

Neste caso, 0xbc é um byte fixo no final do padding, H é a aplicação do hash na estrutua M' = (Padding1 || mHash || salt), maskedDB trata-se da aplicação de um XOR entre DB = (Padding2 || salt) e a aplicação de uma função de geração de máscara a M', mHash é o hash da mensagem original e Padding1 e Padding2 são preenchimentos usados para M' e DB, respectivamente. Neste caso, o salt usado em M' é o mesmo que o em DB, pois assim, na hora de validar é possível identificar a parte que contém salt. Vale lembrar que foi escolhido o SHA3-512 como função de hash (presente na biblioteca Cryptography [4], usada apenas para o hash), justamente por ser robusto e diminuir as chances de colisão.

Seguindo as instruções contidas na RFC, o resultado final, EM, será um bloco de tamanho menor que o permitido pelo RSA (menor que 2048 bits, no caso deste projeto), mas também grande o suficiente para diminuir a chance de conflitos (nesse caso, foi-se determinado o tamanho de EM como 2047 bits). O código para a codificação PSS (seção 9.1.1 [2]) e da função de máscara (seção B.1 [2]) podem ser vistos nas funções `pss_encode` e `mgf` no arquivo `pss.py` (apêndice A), em que, para a geração do salt, foi usada a biblioteca Secrets.

3.2 Assinatura RSA

Com o cálculo da EM e com a encriptação por chave privada do RSA, é calculada a assinatura. No caso do assinador desenvolvido, após assinar um arquivo, é gerado um arquivo binário separado, com a extensão `.sig`. Tudo isso pode ser visto na função `sign`, presente no arquivo `operations.py` (apêndice A). Um exemplo de assinatura

pode ser visto abaixo, com um arquivo de 72 bytes, mostrando que a parte mais demorada da assinatura é o uso do RSA, em que é necessário realizar operações de potência modular com números enormes:

```
Signature Generator and Verifier: Key Generation - Sign File
Path to your file: README.md
Path to your private key file: priv/test-priv.pem
| File signed successfully, stored as README.md.sig
Time Elapsed During PSS Encoding: 0.1202 ms
Time Elapsed During RSA signing: 39.9184 ms
```

Terminal 1: Resultado de uma assinatura

3.3 PSS Verify

Seguindo a RFC 8017 [2], o processo inverso ao encoding PSS é a verificação, na qual, são propostos diversos casos para invalidar uma assinatura antes de afirmar que é válida. Esses casos envolvem verificação de uma possível alteração aos valores fixos, como o 0xbc, ou paddings, mas também uma alteração no M'. Nesse caso, calcula-se o hash de M' como na etapa de encoding, com a mensagem original, chamando o resultado de H'. Caso H' seja o mesmo que o H obtido na mensagem decriptada pelo RSA, EM = (maskedDB || H || 0xbc), valida-se a assinatura. Como será discutido posteriormente, diversos fatores podem alterar o resultado do RSA, o que invalidariam a assinatura. A função `pss_verify`, presente no arquivo `pss.py` (apêndice A) contém a implementação seguindo os passos da seção 9.1.2 [2].

3.4 Verificação RSA

Após receber um arquivo contendo a assinatura, é feita a decriptação deste com chave pública do RSA. A partir disso, e com o arquivo original, é feita a validação PSS, abordada anteriormente. Caso o arquivo, a assinatura e a chave pública estiverem corretos, como no terminal 2, a assinatura é válida. Caso a chave pública não for a correta (terminal 3) ou caso o arquivo não condizer com a assinatura (terminal 4), a assinatura é inválida, pois ambos os casos, ao realizar a decriptação RSA com chave pública, gerariam um EM em um formato que seria invalidado na verificação PSS. Dessa forma, vê-se que tanto a integridade do arquivo quanto a autenticidade são possíveis de ser garantidas.

Tudo isso pode ser visto na função `verify`, presente no arquivo `operations.py` (apêndice A). Os exemplos de verificação abaixo ainda corroboram com o fato de que a parte do RSA é a mais lenta. Todavia, a verificação é mais rápida que a assinatura, pois o expoente público (e = 65537) é um número pequeno comparado ao expoente privado, além de que, como dito anteriormente, por ser um número de Fermat, fornece maior eficiência computacional na hora dos cálculos presentes no RSA.

```
Signature Generator and Verifier: Key Generation - Verify Signed File
Path to your file: README.md
Path to your .sig file: README.md.sig
Path to your public key file: pub/test-pub.pem
| Valid signature
Time Elapsed During RSA Decrypting: 0.2347 ms
Time Elapsed During PSS Verification: 0.1305 ms
```

Terminal 2: Resultado de uma validação com a chave pública correta

```
Signature Generator and Verifier: Key Generation - Verify Signed File
Path to your file: README.md
Path to your .sig file: README.md.sig
Path to your public key file: pub/20250716_143050-pub.pem
| Invalid signature
Time Elapsed During RSA Decrypting: 0.2065 ms
Time Elapsed During PSS Verification: 0.0466 ms
```

Terminal 3: Resultado de uma validação com uma chave pública incorreta

```
Signature Generator and Verifier: Key Generation - Verify Signed File
Path to your file: README.md
Path to your .sig file: main.py.sig
Path to your public key file: pub/test-pub.pem
| Invalid signature
Time Elapsed During RSA Decrypting: 0.3367 ms
Time Elapsed During PSS Verification: 0.0830 ms
```

Terminal 4: Resultado de uma validação com arquivo de assinatura não condizente

4 Conclusão

A partir deste trabalho, foi possível se aprofundar de forma teórica quanto à assinatura digital por meio do RSA-PSS para arquivos, observando a vantagem desta, comparada à falta de entropia do PKCS#1 v1.5, mas também os problemas de eficiência que o RSA pode trazer, dado que sua segurança depende de um tamanho extremamente grande de chaves. Além disso, foi visível que existe uma necessidade de usar certificados digitais e formas de manejar as chaves para garantir melhor segurança ao usuário. Por fim, o código em Python do projeto está presente no GitHub, com o link no apêndice A.

Referências

[1] STALLINGS, W. HALL, P. *Cryptography and Network Security Principles and Practices, Fifth Edition*, 2010.

[2] MORIARTY, K. et al. RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2. Disponível em: <https://datatracker.ietf.org/doc/html/rfc8017>. Acesso em: 15 de julho de 2025.

[3] KALISKI, B. PKCS 1: RSA Encryption Version 1.5. Disponível em: <https://datatracker.ietf.org/doc/html/rfc2313>. Acesso em: 16 de julho de 2025.

[4] Biblioteca Cryptography para o Python. Disponível em: <https://cryptography.io/>.

A Código do Trabalho

O código foi desenvolvido e testado em Windows, com o Python 3.15.5, e pode ser executado pelo programa `main.py`, com funções auxiliares presentes no diretório `helpers`.

- Link para repositório: https://github.com/Luke0133/Trabalho03-SegurancaComputacional.