Signature Generator and Verifier

Lucas Rocha
Universidade de Brasília
211055325

Gabriela de Oliveira
Universidade de Brasília
211055254

Eduardo Marciano
Universidade de Brasília
211055227

1 Gerador e Verificador de Assinaturas Virtuais

Geradores de assinaturas digitais são ferramentas as quais garantem a autenticidade e/ou a integridade de uma dada mensagem. Para isso, utiliza-se técnicas de criptografia, normalmente criptografia assimétrica, onde uma chave privada é empregada para assinar digitalmente um dado, e o hash da mensagem é incorporado na assinatura como uma forma de validação do conteúdo.

Já verificadores de assinaturas são responsáveis por confirmar a autenticidade e/ou a integridade de uma assinatura digital. Considerando um cenário de criptografia assimétrica como o dado acima, o verificador utiliza a chave pública correspondente do emissor para decifrar a mensagem cifrada, caso a mensagem realmente seja do emissor declaro, o hash deste texto em claro deve ser correspondente ao hash enviado pelo o emissor.

Assim, neste notebook iremos implementar um gerador e verificador de assinatura digitais de documentos, utilizando técnicas avançadas em segurança da computação. Para isso, o projeto consta com três módulos principais: geração de chaves, encriptação e decriptação, assinatura e verificação.

O primeiro módulo será responsável pela geração de chaves criptográficas com base no algoritmo RSA, onde essas chaves serão derivadas de números primos de 1024 bits. O segundo módulo dedica-se ao processo de encriptação e decriptação de mensagens utilizando o OAEP (Optimal Asymmetric Encryption Padding). O terceiro realiza o cálculo do hash da mensagem, formatação do resultado em BASE64 e sumariza a verificação de assinaturas digitais com um exemplo prático.

1.1 Geração de Chaves

Primeiramente começaremos nosso projeto com a parte mais crucial nos sistemas modernos de criptografia, a geração das chaves. As chaves são responsáveis por garantir a confidencialidade e a integridade de qualquer algoritmo de criptografia moderno, então é necessário uma alta atenção para se evitar qualquer tipo de padrão ou rastreabilidade em sua geração. Para isso, usaremos duas bibliotecas que irão nos trazer a aleatoriedade necessária para a geração das Chaves.

[1]:
```
from random import getrandbits, randrange
```

O algoritmo que será responsável pela encriptação e decriptação do nosso projeto será o algoritmo de criptografia assimétrica RSA, amplamente reconhecido por sua segurança e aplicabilidade em sistemas modernos de criptografia assimétrica.

Dito isso, por ser assimétrico, iremos gerar duas chaves, uma pública e uma privada. A chave

pública será composta por n=p×q, onde “p” e “q” são números primos com no mínimo 1024 bits, e por “e”, o qual será um número inteiro escolhido que satisfaça a propriedade 1< “e” < (n) e sendo coprimo de (n) = (p1)×(q1). Já a chave privada será representada pelo mesmo n da chave pública e por “d”, calculado como “d” “e”^{1} (mod (n)), sendo “d” o inverso modular de “e” em relação a (n).

Com isto, o primeiro passo será gerar os primos “q” e “p”. Faremos isso usando o teste de primalidade de Miller–Rabin, o que nos retornará dois primos diferentes entre si com uma alta probabilidade.

[2]:
```
def is_prime(n: int, k=128) -> bool:
    """Teste de primalidade usando o algoritmo de Miller-Rabin."""
    s = 0
    r = n - 1
    while r & 1 == 0:
        s += 1
        r //= 2

    for _ in range(k):
        a = randrange(2, n - 1)
        x = pow(a, r, n)
        if x != 1 and x != n - 1:
            j = 1
            while j < s and x != n - 1:
                x = pow(x, 2, n)
                if x == 1:
                    return False
                j += 1
            if x != n - 1:
                return False
    return True

def get_prime():
    """Gera um número primo de 1024 bits."""
    while True:
        p = getrandbits(1024)
        p |= 1
        p |= 1 << 1023
        if is_prime(p):
            return p

p = get_prime()
q = get_prime()

while p == q:
    q = get_prime()
```

Com “p” e “q” gerados, iremos definir algumas funções auxiliares para calcularmos a operação mdc e os valores de “n” e “phi”.

[3]:
```python
def gcd(a, b):
    """Calcula o máximo divisor comum (GCD) usando o algoritmo de Euclides."""
    while b:
        a, b = b, a % b
    return a

def get_n(p, q):
    """Calcula o valor de n = p * q."""
    return p * q

def get_phi(p, q):
    """Calcula o valor de phi = (p - 1) * (q - 1)."""
    return (p - 1) * (q - 1)

n = get_n(p, q)
phi = get_phi(p, q)
```

Agora, iremos definir duas função que irão encapsular as equações de geração de “e” e “d”

[4]:
```python
def choose_e(phi):
    """Escolhe um valor de 'e' que seja coprimo a 'phi'."""
    e = 2
    while e < phi and gcd(e, phi) != 1:
        e += 1
    return e

def get_d(e, phi):
    """Encotra o inverso modular de 'e'."""
    return pow(e, -1, phi)

e = choose_e(phi)
d = get_d(e, phi)
```

Pronto, com “n”, “phi”, “e” e “d” estamos prontos para gerarmos nossas chaves com a função generate_keys();

[5]:
```python
def generate_keys(n, e, d):
    """Gera as chaves pública e privada."""
    public_key = (e, n)
    private_key = (d, n)

    return public_key, private_key

pk, sk = generate_keys(n, e, d)

print(f"Chave Pública: {pk}.\n Chave privada: {sk}")
```

```
Chave Pública: (3, 124181592357443953745223989234822383716629393227683839899877904011383461973814437446103250728416164992254804448189711418036061506800797923293
```

```
524525700058489638454037084849970235667365137918362960655329129907172937389242380
483564165479179508291510699487855122285227547303125522674380177003463302328009252
250035149074121118040692310060500695750405054995415367298742649501867428485813997
700588703787639383287981795461193440824581643151769282446351344084582757173546807
752219695542158052829871675821932446062592970358365952188567813742355266419428000
0041942135788014627534377403965635942098639437365027581308343809056631570093).
 Chave privada: (827877282382959691634826594898815891444195954851225599332519360
075889746492096249640688338189441099948365362987931409453573433786719861552901 6
350466705659758969358056566646823778243425278908640436886086604781958259494923 22
376110319453005527673799658570081523485031535417015116253451335642201552006168 33
356751081771269262274449256792303874685292887487512664823424957245535200271973 07
766157766788222702719131185865223061970586413835882491779291393411358285330111 28
764492954594394130987053370819800455945352398015684463376269405763826946143585 93
367862853004931407334027642513644096268623931305515260829869118415555210 7, 12418
159235744395374522398923482238371662939322768383989987790401138346197381443744 61
032507284161649922548044481897114180360615068007979232935245257000584896384540 37
084849970235667365137918362960655329129907172937389242384835641654791795082915 10
699487855122285227547303125522674380177003463302328009252500351490741211180406 92
310060500695750405054995415367298742649501867428485813997005887037876393832879 81
795461193440824581643151769282446351344084582757173546807522196955421580528298 71
675821932446062592970358365952188567813742355266419428000041942135788014627534 37
740396563594209863943736502758130834380905663157009 3)
```

1.2 Encriptação e Decriptação

Agora, já em porte da chave pública e a privada, podemos avançar para o processo de encriptação e decriptação do algoritmo RSA. A encriptação será realizada com a chave privada (sk), seguindo a fórmula C = M^"e" mod n, onde "M" é a mensagem original, "d" é o expoente da chave privada e "n" é o produto dos números primos gerados. Já para a decriptação, utilizaremos a chave pública (pk), seguindo essa fórmula: M = C^"d", onde "C" é o texto cifrado e "e" é o expoente da chave pública.

1.2.1 RSA

Para encapsular a lógica dessas fórmulas, criaremos uma classe chamada RSA que irá gerenciar as operações de encriptação e decriptação. No método de inicialização da classe iremos passar as chaves pública e privada já definidas.

[6]:
```python
class RSA:
    def __init__(self):
        self.public_key, self.private_key = pk, sk

    def encrypt(self, message: int):
        """Criptografa uma mensagem usando a chave privada."""
        d, n = self.private_key
        result = pow(message, d, n)
        return str(result)
```

```
    def decrypt(self, message: int):
        """Descriptografa uma mensagem usando a chave pública."""
        e, n = self.public_key
        result = pow(message, e, n)
        return str(result)
```

1.2.2 OAEP

Mas, antes de aplicarmos diretamente o RSA na nossa mensagem a ser encriptada, primeiramente, iremos utilizar uma técnica de padding pseudo aleatória, OAEP (Optimal Asymmetric Encryption Padding), com o intuito de aumentar a segurança da nossa criptografia tornando mais difícil técnicas baseadas em análises matemáticas e/ou estruturais, as quais podem possuir alguma efetividade em mensagens com tamanho muito pequeno.

O algoritmo funciona mesclando a mensagem original com um valor pseudo aleatório r de tamanho de 64 bytes, esse r servirá para gerarmos x e y que serão a base da nossa nova mensagem com seu padding. Já o algoritmo de decriptação é determinismo, o qual realiza operações reversas para obter a mensagem original. Vale ressaltar que esse algoritmo isoladamente não traz segurança a mensagem, qualquer um que tenha acesso a cifra será capaz de fazer o processo reverso. No entanto, ele adiciona uma camada de aleatoriedade, dificultando inferências sobre o conteúdo da mensagem antes da descriptografia. Por isso, iremos implementá-lo a seguir encapsulando-o em um classe chamada OAEP.

[7]:
```
from hashlib import sha3_512
from os import urandom

class OAEP:
    k0 = 512
    k1 = 256

    def __init__(self):
        self.x = None
        self.y = None

    def sha3_512(self, data: bytes) -> bytes:
        """Aplica SHA3-256 e retorna o hash como bytes."""
        return sha3_512(data).digest()

    def encrypt(self, message: int) -> int:
        """Encriptação do algoritmo OAEP"""
        r = urandom(64)

        message = message << self.k1

        x = message ^ int.from_bytes(self.sha3_512(r))

        y = int.from_bytes(r) ^ int.from_bytes(self.sha3_512(x.to_bytes(128)))
```

```
        return (x << self.k0) | y

    def decrypt(self, ciphertext: int) -> int:
        """Decriptação do algoritmo OAEP"""

        y = ciphertext & ((1 << self.k0) - 1)
        x = ciphertext >> self.k0

        r = y ^ int.from_bytes(self.sha3_512(x.to_bytes(128)))

        pm = x ^ int.from_bytes(self.sha3_512(r.to_bytes(self.k0 // 8)))

        return pm >> self.k1
```

1.3 Assinatura, Transmissão e Verificação de Assinatura

Agora que já temos ferramentas o suficiente para realizarmos a geração das chaves, encriptação e decriptação, iremos nos preocupar com questionamentos fundamentais na nossa aplicação: Como podemos assinar a mensagem? Como podemos transmitir essa mensagem de uma maneira adequada? Como podemos verificar nossa assinatura?

1.3.1 Assinatura

Para assinarmos digitalmente nossa mensagem a ser transmitida, utilizaremos uma função hash, mais especificamente as funções SHA3-256 e SHA3-512. Uma função de hash é um algoritmo que dado uma entrada de dados de qualquer tamanho, sua saída será de tamanho fixo, a qual é chamada de hash. Utilizaremos essa função, pois ela possui a propriedade de que a computação reversa do hash para a mensagem original é extremamente custosa computacionalmente, propriedade da irreversibilidade. Ainda, uma função hash possui uma segunda propriedade de interesse para nós, a propriedade de resistência à colisão, que define que o processo de descoberta de uma mensagem que gere um mesmo hash é extremamente caro computacionalmente.

Assim, a nossa estratégia de assinatura será transmitir nossa mensagem e o seu hash, porém este hash estará criptografado pelo algoritmo RSA. Com isso, ao realizar a decriptação utilizando a chave pública do emissor, o receptor poderá fazer o hash da mensagem recebida e comparar com o hash decriptado, e, caso eles sejam iguais, o emissor terá a garantia que a mensagem é realmente do seu emissor, já que apenas ele possui acesso a sua chave privada capaz de gerar a encriptação correta.

Nesta parte do nosso projeto, utilizaremos a biblioteca "hashlib" para o desenvolvimento da nossa função Hash, sendo a única parte do nosso projeto que utilizará código de terceiros.

Função Hash

[8]:
```
from hashlib import sha3_512, sha3_256
class sha3:
  @staticmethod
  def hash_512(message):
    if type(message) == int:
      message = str(message)
    elif type(message) == str:
```

```
      message = message.encode()
    return int.from_bytes(sha3_512(message).digest())

  @staticmethod
  def hash_256(message):
    if type(message) == int:
      message = str(message)
    elif type(message) == str:
      message = message.encode()
    return int.from_bytes(sha3_256(message).digest())
```

1.3.2 Transmissão

Para a transmissão da nossa cifra e da menssagem, primeiramente, iremos converter a concatenação de ambas para o formato Base64. Isso será feito, pois a Base64 é uma forma de codificação que transforma dados binários em uma sequência de caracteres ASCII, o que normalmente é um formato mais adequado a transmissão de dados.

Nossa implementação é um modelo simplificado da Base64, mas suficientemente robusto para o escopo deste projeto.

BASE64

[9]:
```
class base_convert:

  @staticmethod
  def format(message: int):
    base64_chars =␣
↪"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
    res = ""
    #Enquanto a mensagem não se tornar 0 realiza o método de divisões sucessivas
    while message > 0:
      remainder = message % 64
      res = base64_chars[remainder] + res
      message = message // 64
    return res

  @staticmethod
  def parse(base64_message: str):
    base64_chars =␣
↪"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
    res = 0
    # procura o caracter encontrado na string
    for char in base64_message:
      res = res * 64 + base64_chars.index(char)
    return res
```

1.3.3 Verificação com Exemplo Prático

Agora, finalmente partiremos para a demonstração do nosso projeto. Nesta demonstração, geramos o contéudo do nosso plain text utilizando o lorem ipsum, um gerador randômico de texto. O arquivo em si está disponível no diretório resources com nome "input.txt".

O processo começa com a leitura do texto em formato binário, que é convertido diretamente em um valor numérico inteiro. Em seguida, calculamos o hash da mensagem para criar uma representação única e compacta do texto. Após isso, utilizamos o padding OAEP sobre o hash da mensagem para adicionar segurança à encriptação. Com o hash preparado, realizamos a sua criptografia utilizando a chave privada do algoritmo RSA. Por fim, para realizarmos a transmissão, concatenamos a mensagem ao hash criptografado e, assim, colocamos o resultado da concatenação em Base64.

Já o processo de verificação é o reverso no processo de assinatura. Primeiramente, desfazemos o formato Base64, separamos a mensagem do hash criptografado e realizamos a decriptação com a chave pública do emissor e retiramos o padding resultado do OAEP. Assim, ao final dessa série de processos, temos dois dados fundamentais, a mensagem e o hash do emissor, caso o hash do emissor seja compatível com o hash da mensagem, realmente essa mensagem é do emissor declarado, garantindo assim confiabilidade a origem dessa mensagem, caso não seja, concluímos que ou o hash ou a mensagem foram interceptadas e trocadas durante a nossa transmissão, logo, ambas devem ser invalidadas e descartadas.

[10]:
```
# Instanciação da classe RSA
cy = RSA()
oaep = OAEP()

f = open('../resources/input.txt', 'rb')
plain_text = f.read()
f.close()

plain_text = str(int.from_bytes(plain_text))

# Calculo do hash 256 da mensagem
h = sha3.hash_256(plain_text)

# Encryptação desse hash
e = oaep.encrypt(h)
e = cy.encrypt(e)

# Concatenação entre plain_text e o hash encriptado
message_to_encode = plain_text + str(e)

# Conversão para a Base64
encoded_message = base_convert.format(int(message_to_encode))

print("Mensagem codificada em base64:", encoded_message)

# Parse da base64
decoded_message = str(base_convert.parse(encoded_message))
```

```python
# Separação entre o plain_text e o hash encriptado
decoded_plain_text = decoded_message[:len(plain_text)]
decoded_plain_text_hash = sha3.hash_256(decoded_plain_text)

encrypted_hash = decoded_message[len(plain_text):]

# Decryptação do hash encriptado
decrypted_hash = cy.decrypt(int(encrypted_hash))
decrypted_hash = oaep.decrypt(int(decrypted_hash))

# Comparação entre os hashs para verificação da assinatura
print("Hash da mensagem:", decoded_plain_text_hash)
print("Hash decodificado:", decrypted_hash)
print("Verificação de assinatura:", decrypted_hash == decoded_plain_text_hash)
```

```
Mensagem codificada em base64: EOxOvgF3h+Y2r0PvkqnFY5DL+ipBkSGxALH4YdfoARSLVvrfn
KbuGbuKL3w4/sfZjPFLikYQF5ld18/G3pnpXxivosoQvx3KcC9GnQIef0dg7WBtNLXQyS+ikW7qBZ7Zg
2ABzr3pHaEZKm/4rR/uDybjTuZ1fzYk6h0jOgsCQaSHyANDVWRegbPwi6ZEDnZNTnUbKvvLa9mkLM12F
PyyA1lN8RL2Snrdt2P6fKi7HJa+bE5HeQQTJGk2lt+o6RTr9PpBY6kyRvvbZ3spJh25VIyf/lTsSK7pZ
UIAU5q0UQjf27PL//0YTVZ0/CAPfPzBeGIEdzb/0hwMqyhc6Gw8qDbvIBZvjpwtAiBh16mCO3goWA9Qa
tg4ih/LH6dI3V6CNBkU/Fypl7pNFevngSK6NPJHwpWg+gnN9HjSdUXcPJF1mDvU3sphKYloFGvJa4uAx
HO3tOlZ7rKe/I9qBGVV2KsJtI/o32GwL6l28STidGd0CLzXSFXMtb5UqsLZRx4gRBAxiJAOL+rfdNgho
a2WoZYK+d+d/Vks7rBMlHXJZblF67EvQ8LN7bAuACNVv5psnpXNgZzPuzCacDIfUU0rMYGHCL/+wW1ae
O6ZjB2Ou6tOBSC65n4P2AD1xUkkKyYAjyneQgeOVXNJwLdtn7OJuQzLEVD0HFFcibsjSapybfkFa5JYE
UhZnyVqVgMxVY+CVWlDYJ4c+qrfMDIx7r3fVT61uNWX7l6oPnVEq4hOd5Zlc/uS/GdQGn2H7u9TsUY7T
1FAr5ozlTYuuTUIKwJZfdifJNJyTY3UYSVthqIpos4TsJP0+SkonlK5qyt8iHngHHeAqjGcIXvQ5VEIo
zZpJ3zG6pG2QQIlxK9bGTOSztTVAiq4+yuV1yY+o2Lmj7E/yM6z5cavR6u2ou6eUURQGc/kzxOFVgqE6
pwpFvqQ9kBt86th5FjSTL/Ro7WTB8h/fnLmTr0uLw5kJp7ShdvdtL0nuMuB1IPWSEmjiniKQbg2D3kOL
vWVDpCw37AZNvjODWsHPhR02P/SMOqeL9I5+8T3/i+f04Fgq2kJOp6xYyMzqVL9q6mX/2n6QJYjYi+LG
C5UUGx4wtGcB3hHd/V1/+hzsr1HgZMH7IvcvHvpRIi7U+KqzzkdTZyCzAE7/LDlZCN5V6hoaLVb4uOPJ
KO2bvLUCKbY9vn6y4qiDHBB2FmP1rewiG0RmyihxygziYjC/zZvkxUnHG/akqwmOcxHoveIYFH+5Xz5O
LXdMZ7189SSrkSWW8TDurpfhnJk0IV6O1+GRx1TOku33f3u0hU3i0xC6sRS5hDAlgZ9y1WlCzJMWip32
esJYSu/0Mgf5d79U/cKzLLf9d8T2ngm25G4SM0GLv9Ux1/Tf5mFF2yLramhksVhzMnrzsZpSkyRuLKsI
RZam0ykcApFCSCc9+AXk9c73Z7B0ywqU2O9dr2XgIL1NW5FZsw8ertbJVRvSmFaRr0w7sDybxO6cJrkY
JAwUGuLx/AAQbkuwJUVO0JdemzqnZIuCP4ksXHJZaZ6tewEAMRwvR8OC5uwnkX9odlnQj3VlAOE1ENzq
mQFgEWD5Rwyyg8G4NCNn6ebom5dPG0kiFtZROLspoz7Dx/3bKTrhYSY2LTvnOY55lLpwU/MO3HYVV/ul
xWRlhxoNkUNd+CpT1/wQZgR3dTVq0izTxGNAuVSSC/6jfwH4ZWTl1VadqndC13Ex8VT6CkTeDnkvngsV
+HN/k69f2r68xtOqzLElwkOxlr+8zhSdCaeeFxTIRM3HQE8oxXkuhcB4b8IxLv0ypvw7q9k7H60XwJRO
eTPcC8RUaMnbovGv39yw0/zfSJja8PpDIFs/7zZT6LoCmZGpbflmItu005kAnUKnoXcLv0aOnk4qTBpC
9qtLpfW4acqzvfVGgAde3zJszYDzHXAzxUMrTVkEJwe6CcJwMH6qWrkj92yLhOJc1qKBygRkB3vpgCZN
iRs55ewjjjHvwPSfgEUnrmoqJpDsmAUUfRadFX5BekN8ovsJWUj+7Pj1m1ZjjGnaGxtF/RimDde2mpJl
XEC1CNtdg1AKovA+Wl6O8wrPQwGM9xUXp+4FtXCchgq+eD59CBE/9DnCtZvWlsX9mt/QD31QqgXTUHA3
EvpvekMmDzC8rmhL4CCf51c1IugN62Nq5qI9qVSK0tok7iSFBE7eBq559AwF/pTvlbK/ihbhjqwzSN2J
xIUbnLPpnjWznkOpKIrkAvjg6FrhkVZ49IY5D5OSfgJoU06K6UiwXytGEIYcPI55DKhr05vzIfHgUtz4
kejKqxP+jOoF0B6c8gXs9oFD1s+7DBZOitOn2l
Hash da mensagem:
10155982480115756648127591059143648659079994826288477104413493581540835761445 6
```

```
Hash decodificado:
101559824801157566481275910591436486590799948262884771044134935815408357614456
Verificação de assinatura: True
```

Caso todos os passos tenham sido corretamente executados em sua devida ordem, a verificação da assinatura terá sido validada com a mensagem: "Verificação de assinatura: True". Caso não, execute novamente todos os blocos desse notebook em sua devida ordem.

Com isto, teremos efetivamente construído um software capaz de gerar assinaturas digitais, transmiti-las e verificá-las.

1.4 Repositório e Slides Da Apresentação

Para mais informações sobre a implementação, confira o repositório em: https://github.com/EduardoMarciano/Seminario-SC

Para mais informações sobre a apresentação, confira os slides em: https://www.canva.com/design/DAGd5pQ5obo/k4p1LVSKr-lb5PC8-SkCqw/view