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, 153130435540715886857042415276984142328322296164438033031183598041058468093454174800809430168218132040922044059041233752731122733636271702891
```

```
626429026703966817708972806432175675297915867199212308539782352420159284936832880180291367369854404996963957465775780802348806898231713150195651119997270739069755850023882216512128909462524519520971112985081059976065447842618844649857969174654644252817477543075565840639766725953994400262337744957450834908443718261081202660084982654497218640028743633735364500770223829963310832266476215741996253906380631201337806495432071208082749592367751187338712236419929269738671530795479).
 Chave privada: (102086957027143924571361610184656094885548197442958688687455732027372312062302783200539620112145421360614696039360822501820748489090847801927750952684469311211805981870954783783531943911466141539026521568280106189957888586786860911579902936664642638310517187201565871265487808766797100746664847159379837233347596589047011142662273969871010342818872500059281767556378865536133034541413525841662551002494071607966284199660253658942915170791040073579080812753591780149636107354948664752351142591712855863936048775700089228006280612371471086239438639829417153670606781457226073511579627999421129764506489631868138027164111, 1531304355407158868570424152769841423283222961644380330311835980410584680934541748008094301682181320409220440590412337527311227336362717028916264290267039668177089728064321756752979158671992123085397823524201592849368328801802913673698544049969639574657757808023488068982317131501956511199972707390697558500238822165121289094625245195209711129850810599760654478426188446498579691746546442528174775430755658406397667259539944002623377449574508349084437182610812026600849826544972186400287436337353645007702238299633108322664762157419962539063806312013378064954320712080827495923677511873387122364199292697386715307954720808274959236775118733871223641992926973867153079547)
```

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: BCGxOWNbrq2OWcoV/U5ysM7ZzJNRAe65tqtzI+M2JEOEBLw/l
77BIUhJPJfT6jPKnOjo8UIa7cLxaTA6DViTebeG4T5VFqjF7GOKvVUEchLtKfQKqV5jeR4hMIGUIl9IG
pLwcPldabvUpNxWNkRLnrL+fsNDr464AygdGauwDQarJlTL8FkBEqpuOhgXnYfh36eXIf55KdSilPApV
B+xddFNh1shIS7/IUXop28xrnrNfXEccwt+rGOv09WavOebmO8Y9tTOxlR9EMORi85BvyRVD5e7r76z+
zidFGlFBciqpyNlTf9KbuGbj43TyAO1O+PuFxrtdMvZF8larr5szxFpYVV6PFEg/QTn4ybiS4TV2X+9I
hXJzbjDFzpXSguSybgfn5yfZ4r70doYBQb6de0mhQGHzQtkyC9/YaGORMmDNaqbeGRouKQL6PxQrMGO7
/ZAWar4dMab0KkDYhK9dWXB3pUiWolZg6Lfid+BSWtB1CIajyjyel73q7CSC9za4J30F9c0vbLXjgbEN
fjhRnMhrgEYElpFeEYKsj7Bq1W7kgrCCgsPRoeDOwigb0czhLeSrCTItzNdtGQ8pVlXC5Yfd6Lfsjkyl
xqwfLU3WNFSjEB9oUXG3jw8AKpbVIjHIvO4wx7YP9ylgYzIQ31VH62fk7pDmOndF6AwHXwc79lL0sM6g
Dgp4Vqidf/Hi2OJUdoy8xaaEzK3fURBMMlbBk/PXaAll7IcEz+hQ+DHKDI3xNDrYp85Dnelr9Vt7sele
t1WSt8ll+26aDdyfoAS2Tli2n1g5+8gWbejPV6XKj9PzpwfTtSMaqcyVEod51HBNvMs6Y0cniy6f/xIm
9jIrCP2kESS/t8GOITPvacXV372AQduKVIkFQbWSuOlShTGEoZ265kByv9sFsdBptctjk1BXI5EDYGRc
0cgCBwsZB661/1XZj+vV7zUrnfUZWZzIF8NOjkvxn0Dc23GZa4toKAuMp2vMmIv0KX6bU5twQt5MscN9
nb05A5TLpx+J+u+TkahRKf+hvp00PFHnqzTJ/eWi46ghSMWGdkkwM++TrQYHNbC7mjnm9tfmMSno3ibd
WdPlaJ7e5Asl9LdcsltTvpAPyM19SJlb0AlZARAczthu/rWs5dxp3K17s5NJTEwAPKpH+adfZBG7uSOc
nH4ChZIxBy7+EwHurFYkAloE1HdnevYmU6gzGFrfx9QlRPjm07KPWiqDHFO3M8HVRxffUluy4QEuwcC2
FhmgJ1PDAdlz4u+tWeJyRaAUlLtnOiWV37SSFyo0bt8Y7J+zkouD4IeVlDuPSsXBKCVtiRoyzvHpANeQ
rPhyjkuVHjbyZ7w2P/QIvmrrYb2WNZVfJsTfcYOAnlWMqzlIqb6/jgAGeXp8lUwYH9MEftwtKOxk5efB
hP1Huo2JVqB2rp2UoQ1wZgMHZtCUg5IVVvZwX1JfN/43Ytr8Zjcw/a9wJ0dK6jEikUwckobMB5RBH3Qw
ZMzy6JBoFAYEAu16gRlrvez0mOXR3gwPTIL9VJHq1Ek9Woe+i/1hiwU7dWCnqQWbAOTtawYpKLb7dJ3v
RlrV9D4s29iZlSrLeLqk3prfyfxw0Cla0pNS1FmxqlrSzFN6AZDgBjfUOgsf841Irs+EbH7XINLUNfrv
+LECyXvcT/bycRZVeDsKAOgXKh2KlE/xs3I7TT8lp+2q6oH7JMEiHJWlfh4d0ZkisWs/C4HQNoiWIBW0
3SgypY5Q5ueRFoqtMvk/t+2G203PmFAIVttsGKKS5Mw9JTNTdDYV26d9Z0jpAfjQ3cSUkXSI7zvCTIQ1
KJcpON9um5VIH0gBcCdhK8E8gzlRc9UZY2ZHG+bDfFFpon4P3Vljb4KjpajapmTF7OuQXjK50aYVU5Rp
WTjR+AR9hdP84eBhtkHMqX3PkCv7Wq/ko6jmEmI7g4di62H4Y9bslyYzR+gAD/FDqmasr9ISa9PAgclZ
7fkloqImrjcxCJu48M7juJYe+dZEpO2ULg1x3lJF4tTV/yoB8o58jLcNLEqA+aiuJ51Xo+6yj3c+7v+b
HpH64t4cUe5JNkGEX0y6uMAB/unYWusEESHiPvpdyHPexLCMPOCMDkpRTp+fmQjEyUvZlbD67Ai252VL
C8g/IjYWcEYhISVQqFFxnWBUkWrYQFRMWW14t/c4XTnqGmxO4lAaRo4HWk4zwtPRZfhAS8TCpGAaePPY
wUogzUeLB604cJ3iaNaNandpUi8aLpdmXwI3F92VHRsObb9Or2qlGXpYglSRkuDPGguLWx/2PY6mvIuw
aJHvLHAdgxJNK5hfheJSLc+n2w93i6usVFagCFMY
Hash da mensagem:
10155982480115756648127591059143648659079994826288477104413493581540835761445 6
```

```
Hash decodificado:
10155982480115756648127591059143648659079994826288477104413493581540835761445 6
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.