Explicacão do Código Python para Criptografia RSA com OAEP e Base64

Davi de Araújo Garcez Bueno - 211060586
Erick Hideki Taira - 222011525

Janeiro 2025

1 Introdução

O código apresentado implementa um sistema de criptografia baseado no algoritmo RSA, combinando técnicas como OAEP (Optimal Asymmetric Encryption Padding), codificacão Base64 e teste de primalidade de Miller-Rabin. Além disso, ele realiza assinatura digital utilizando RSA e verifica a autenticidade da assinatura.

2 Explicação das Funções

2.1 Função `decompondo`

Esta funcão decompõe um número na forma , onde é impar.

```
def decompondo(n):
    # Decompoe n-1 como 2 elevado a e vezes m, onde n e impar.
    # Args:
    #     n: numero para decompor
    # Returns:
    #     int: expoente
    #     int: numero
        e = 0
        m = n - 1
        while m % 2 == 0:
            m //= 2
            e += 1
        return e, m
```

2.2 Função `miller_rabin`

Implementa o teste de primalidade de Miller-Rabin, retornando verdadeiro se o número for provavelmente primo e falso caso contrário.

```
    def miller_rabin(n, rodadas=10):
    """
```

```
    Miller-Rabin primality test
    Args:
        n: numero para ser testado
        rodadas: quantidade de rodadas a serem feitas
    Returns:
        bool: True se e primo, False se nao e primo
    """
        if n <= 1 or n == 4:
            return False
        if n <= 3:
            return True
        if n % 2 == 0:
            return False

        expo_k, m = decompondo(n)  #exponte K e m tal que n - 1 =
    2^k * m
        for _ in range(rodadas):
            a = random.randrange(2, n - 2) # pega um numero
    aleatorio entre 2 e n-2
            """ x = a^m mod n """
            x = pow(a, m, n)
            if x == 1 or x == n-1: # se x for 1 ou n-1, entao n e
    primo
                continue
            else:
                teste = False
                for _ in range(expo_k - 1): # para i de 0 ate k-1
                    """ x = x^2 mod n """
                    x = pow(x, 2, n)
                    if x == n-1:
                        teste = True
                        break
                if(teste):
                    continue
                else:
                    return False
        return True

```

2.3 Função `mgf1`

Mascara uma entrada utilizando uma funcão de hash para gerar uma saida pseudoaleatória de comprimento especificado.

```
    def mgf1(seed: int, length: int, hash_func=hashlib.sha1) ->
    bytes: #mascara de geracao de funcao
        if length > (hash_func().digest_size * (2**32)): #tamanho
    da mascara
            raise ValueError("mask too long")
        # Converte o tamanho de bits para bytes
        byte_length = (length + 7) // 8
        # Converte a seed para um tamanho fixo de bytes
        seed_bytes = seed.to_bytes((seed.bit_length() + 7) // 8,
    byteorder='big')
        T = b""
        counter = 0
```

```
        while len(T) < byte_length: # enquanto o tamanho de T for menor que o tamanho de bytes
            C = counter.to_bytes(4, byteorder='big')
            T += hash_func(seed_bytes + C).digest()
            counter += 1
        return T[:byte_length]

```

2.4 Funções `shift_left` e `shift_right`

Realizam deslocamento de bits para a esquerda e direita, respectivamente.

```
    def shift_left(shift_values, n): #deslocamento para a esquerda
    """
    Shift Left
    Args:
        shift_values: quantidade de shift
        n: numero para ser shiftado
    Returns:
        int: numero shiftado
    """
        return n << shift_values

    def shift_right(shift_values, n): #deslocamento para a direita
    """
    Shift Right
    Args:
        shift_values: quantidade de shift
        n: numero para ser shiftado
    Returns:
        int: numero shiftado
    """
        return n >> shift_values

```

2.5 Função `msb`

Determina o bit mais significativo de um número.

```
    def msb(n): #bit mais significativo
    """
    Most Significant Bit
    Args:
        n: numero para ser calculado o MSB
    Returns:
        int: MSB
    """
        msb = 0
        while n > 0:
            n >>= 1
            msb += 1
        return msb

```

2.6 Função `number_of_bits`

Calcula a quantidade de bits necessária para representar um número.

```
    def number_of_bits(n): #calcula o numero de bits de um certo
    numero
    """
    Number of Bits
    Args:
        n: numero para ser calculado a quantidade de bits
    Returns:
        int: quantidade de bits
    """
        temp = n
        count = 0
        while(temp != 0):
            temp = shift_right(1, temp)
            count += 1
        return count

```

2.7 Função `DB`

Concatena um hash e uma mensagem, adicionando um preenchimento de no minimo 8 bits.

```
    def DB(pHash, mensagem): #faz a concatenacao de pHash e
    mensagem
    """
    DB
    Obs:
        padding minimo de 8 bits
    Args:
        pHash: pHash
        mensagem: mensagem
    Returns:
        int: DB com padding
    """
        pHash = shift_left(1536 , pHash)
        padding = shift_left(number_of_bits(mensagem), 1)
        pHash = pHash | padding
        return (pHash | mensagem)

```

2.8 Função `prime_numbers`

Gera dois números primos utilizando o teste de Miller-Rabin.

```
    def prime_numbers(): #descobre os numeros primos usando a
    funcao de Miller-Rabin
    """
    Descobrindo os numeros primos
    Args: none
    Returns:
        list: lista com os numeros primos
    """
```

```
        n = 1 << 1024
        contador = 0
        par_de_numeros_primos = []
        while True:
            if miller_rabin(n):
                par_de_numeros_primos.append(n)
                contador += 1
            if contador == 2:
                break
            n += 1
        return par_de_numeros_primos

```

2.9 Função `enc_oaep`

Implementa a etapa de encapsulamento OAEP para preparacão da mensagem antes da criptografia RSA.

```
    def enc_oaep(mensagem): #encriptacao OAEP
    """
    Encapsulamento OAEP
    Args:
        mensagem: mensagem
    Returns:
        int: EM
    """
        seed = random.getrandbits(256).to_bytes(32, byteorder="big"
    )                                   # gera uma seed aleatoria
        seed = int.from_bytes(seed, byteorder='big')
                                        # converte a seed para int
        pHash = hashlib.sha3_256().digest()
                                        # hash de NULL
        pHash = int.from_bytes(pHash, byteorder='big')
                                        # converte o hash da seed para
     int
        maskedDB = DB(pHash, mensagem) ^ int.from_bytes(mgf1(seed,
    1792, hashlib.sha3_256))            # faz a concatenacao de pHash
    e mensagem
        maskedSeed = seed ^ int.from_bytes(mgf1(maskedDB, 32,
    hashlib.sha3_256))                      # faz a mascara da seed
        maskedSeed = shift_left(1792, maskedSeed)
                                        # desloca a seed para a
    esquerda em 1792 bits
        EM = maskedSeed | maskedDB
                                        # concatena a mascara da seed
    e a mascara da mensagem
        return EM

```

2.10 Função `dec_oaep`

Realiza a decodificacão OAEP para recuperar a mensagem original após a descriptografia RSA.

```
    def dec_oaep(c): #decriptacao OAEP
    """
    Decriptando OAEP
    Args:
        c: EM
    Returns:
        int: m
    """
        bitMask = shift_left(1792, 1) - 1
                                        # mascara de bits
        maskedDB = c & bitMask
        maskedSeed = shift_right(1792, c)
                                                # mascara de DB
        seed = maskedSeed ^ int.from_bytes(mgf1(maskedDB, 32,
    hashlib.sha3_256), byteorder='big')   #
        db = maskedDB ^ int.from_bytes(mgf1(seed, 1792, hashlib.
    sha3_256), byteorder='big')          #
        bitMask = shift_left(1536, 1) - 1
                                        #
        m = db & bitMask
                                        #
        MSBit = msb(m) - 1
                                        #
        MSBit = shift_left(MSBit, 1)
                                        #
        m = m ^ MSBit
                                        #
        return m

```

2.11 Função `multiplicative_inverse`

Calcula o inverso multiplicativo de um número utilizando o algoritmo estendido de Euclides.

```
    def multiplicative_inverse(a, b): # calcula o inverso
    multiplicativo usando o Algoritmo extendido de Euclides
    """
    Inverso multiplicativo
    Obs
        a > b
        a x t1 e congruente 1 mod b
    Args:
        a: numero
        b: numero
    Returns:
        int: inverso multiplicativo
    """
        if b > a:
            a, b = b, a
        old_a = a
        if gcd(a, b) != 1:
            return None
        q = a // b
        r = a % b
        t1 = 0
```

```
        t2 = 1
        t = t1 - t2 * q
        while b != 0:
            a, b = b, r
            if b == 0:
                t1 = t2
                if t1 < 0:
                    t1 += old_a
                return t1
            q = a // b
            r = a % b
            t1, t2 = t2, t
            t = t1 - t2 * q

```

2.12 Funções base64_encode e base64_decode

Implementam codificacão e decodificacão Base64.

```
    def base64_encode(mensagem): #codificacao usando a base64
    """
    Base64 Encode
    Args:
        mensagem: mensagem
    Returns:
        bytes: mensagem codificada
    """
        mensagem = mensagem.encode('utf-8')
        mensagem = base64.b64encode(mensagem)
        return mensagem

    def base64_decode(mensagem_encriptada): #decodificacao usando a
     base64
    """
    Base64 Decode
    Args:
        mensagem_encriptada: int mensagem encriptada
    Returns:
        str: mensagem decodificada
    """
        mensagem_encriptada = mensagem_encriptada.to_bytes((
    mensagem_encriptada.bit_length() + 7) // 8, byteorder='big')
        mensagem_encriptada = base64.b64decode(mensagem_encriptada)
    .decode('utf-8')
        return mensagem_encriptada

```

2.13 Funções enc_rsa e dec_rsa

Realizam a criptografia e descriptografia RSA.

```
    def enc_rsa(n, e, m): #encriptacao RSA
    """
    RSA Encryption
    Args:
```

```
        n: produto dos primos n
        e: chave public e
        m: mensagem
    Returns:
        int: c
        int: n
    """
        c = pow(m, e, n)
        return c, n

    def dec_rsa(p, q, e, c): #decriptacao RSA
    """
    RSA Decryption
    Args:
        p: primo p
        q: primo q
        e: chave publica e
        c: mensagem criptografada
    Returns:
        int: mensagem descriptografada m
    """
        n = p * q
        phi = (p - 1) * (q - 1)
        d = multiplicative_inverse(phi, e)
        m = pow(c, d, n)
        return m

```

2.14 Funções `assinatura_com_rsa` e `verificar_assinatura_com_rsa`

Geram uma assinatura digital usando RSA e verificam sua autenticidade.

```
    def assinatura_com_rsa(message, key, n): #assinatura com RSA
        message = base64_encode(message)
        message = int.from_bytes(message, byteorder='big')
        hash_message = hashlib.sha3_256(message.to_bytes(32,
byteorder='big')).digest()
        hash_message = int.from_bytes(hash_message, byteorder='big'
)
        encrypted_hash = enc_rsa(n, key, hash_message)[0]
        return message, encrypted_hash

    def verificar_assinatura_com_rsa(message, enc_hash, e, n):
        decrypted_hash = enc_rsa(n, e, enc_hash)
        hash_message = hashlib.sha3_256(message.to_bytes(32,
byteorder='big')).digest()
        hash_message = int.from_bytes(hash_message, byteorder='big'
)
        return decrypted_hash[0] == hash_message

```

2.15 Código de teste

O código de teste gera dois números primos, calcula e , codifica uma mensagem em Base64, realiza a criptografia RSA, descriptografa a mensagem, gera e

verifica uma assinatura digital.

```
    # Teste de execucao do programa
    lista = prime_numbers()
    p = lista[0]
    q = lista[1]
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 65537
    message = "Bluey Heeler"

    print("Texto original:", message)
    print("====================================================")
    message = base64_encode(message)
    print("Message com BASE64:", message)
    print("====================================================")
    message = enc_oaep(int.from_bytes(message, byteorder='big'))
    print("Message com OAEP:", message)
    print("====================================================")
    enc_message = enc_rsa(p * q, e, message)[0]
    print("Message encriptada com RSA:", enc_message)
    print("====================================================")
    dec_message = dec_rsa(p, q, e, enc_message)
    print("Message descriptografada com RSA:", dec_message)
    print("====================================================")
    dec_message = dec_oaep(dec_message)
    print("Message descriptografada com OAEP:", dec_message)
    print("====================================================")
    dec_message = base64_decode(dec_message)
    print("Message descriptografada com BASE64:", dec_message)

    d = multiplicative_inverse(e, phi)
    message = "Bluey Heeler"
    c = (assinatura_com_rsa(message, d, n))
    verificar_assinatura_com_rsa(c[0], c[1], e, n)
```

2.16 Impressão dos resultados

Por fim, o código imprime os resultados obtidos durante o teste.

```
Texto original: Bluey Heeler
============================================================
Message com BASE64: b'Qmx1ZXkgSGVlbGVy'
============================================================
Message com OAEP: 123934030549754492491210231517666468099730
369442567688928578558286419964863381939669809836534438834640
574310406666447300575050808586264352549897458907546763535965
984452608526562908810643416967711008306866672881682194684283
740557775781735790577949197826806275977650831281749354860437
523127708065751832847683773984717204032294259713046385753159
905749941944973553402221672293014068464668257642064375871658
774153288925901118853075942714803710368797697455635279938888
554817566393562761016967537375626053087500121574139942083547
872647694721457958554497259380122494023896678864888018821502
070572652576557617081555911780555546
============================================================
Message encriptada com RSA: 150560028313053864293695780953 55
```

```
508273616502034331236071775124786842864065033001042693931720
646944781588159691880277391553798572617769934379623251912328
846197149916658589532294448631519904152853405610787998586710
899393634545821672318441986195833808438244444473133093529842
908609702557069293870537639358475093499886731555097184848691
033170877550709922480953798163720998271677316482433149367349
280445062421288233406709124016609068591255854451085685936615
294531036455966095834693157492962824461428369243671654428706
376324900791509868420949038355712002219951420371594635475017
482891061696236046241908471012372146994055492
============================================================
Message descriptografada com RSA: 123934030549754492491210231517666468099730369442567688928578558286419964863381939669809836534438834640574310406666447300575050808586264352549897458907546763535965984452608526562908810643416967711008306866672881682194684283740557775781735790577949197826806275977650831281749354860437523127708065751832847683773984717204032294259713046385753159905749941944973553402221672293014068464668257642064375871658774153288925901118853075942714803710368797697455635279938888554817566393562761016967537375626053087500121574139942083547872647694721457958554497259380122494023896678864880188215020705726525765576170815559117805554 6
============================================================
Message descriptografada com OAEP: 1082358658154514268713979
21985126684281
============================================================
Message descriptografada com BASE64: Bluey Heeler
```

3 Fluxo Geral do Código

1. Geração de dois números primos p e q.
2. Cálculo de n e de phi.
3. Definição da chave pública e cálculo da chave privada.
4. Codificação da mensagem em Base64, seguida do algoritmo OAEP.
5. Criptografia da mensagem com RSA.
6. Descriptografia da mensagem e reversão do processo de OAEP e Base64.
7. Geração e verificação de uma assinatura digital.
8. Teste de execução do programa.
9. Impressão dos resultados.

4 Conclusão

O código implementa um sistema robusto de criptografia e assinatura digital utilizando RSA. As funções desenvolvidas garantem a integridade e segurança dos dados manipulados, demonstrando o funcionamento prático da criptografia assimétrica combinada com técnicas adicionais de segurança.