Relatório Trabalho 3 - Implementação RSA

Aluno: Felipe Renato Alvarenga Batalha (19/0012862) Turma 2

06 de Dezembro de 2023

1 Parte I - Geração de chaves e cifra

A primeira parte do trabalho consistiu na elaboração do RSA com p e q de 1024 bits, e contemplou os aspectos de cifração e decifração. A implementação se deu na linguagem *Python* e é necessário o uso de algumas bibliotecas, como *threading*, *queue* e *concurrent.futures* para lidar com a etapa de verificação de primalidade de forma um pouco mais performática em Python, fazendo o uso de concorrência, conforme a seguir:

```
1  def miller_rabin(n):
2
3      # trecho removido para abreviar o relatorio
4
5      with concurrent.futures.ThreadPoolExecutor() as
           executor:
6          futures = []
7
8          k = 64
9          for i in range(1, k):
10             a = random.randint(2, n -2)
11             future =
                   executor.submit(check_primality, a,
                   d, n, r)
12             futures.append(future)
13
14
15         results = [future.result() for future in
               concurrent.futures.as_completed(futures)]
```

```
16          return any(results)
17
18  def check_primality(a, d, n, r):
19      x = pow(a, d, n)
20      if x == 1 or x == n - 1:
21          return True
22
23      for j in range(1, r):
24          x = pow(x, 2, n)
25          if x == 1:
26              return False
27          if x == n - 1:
28              return True
29
30      return False
```

A implementação se encontra no arquivo *rsa.py* e pode ser executada individualmente, basta identificar a mensagem no arquivo e trocar pela desejada, e então executá-lo com o interpretador *Python*.

2 Parte 2/Parte 3 - Assinatura/Verificação

O OAEP foi implementado no arquivo *oaep.py*, e pode ser executado individualmente assim como a implementação da parte 1. Nele, há uma variável responsável por armazenar a mensagem, sendo assim possível verificar a cifração e decifração em OAEP implementada. Abaixo, trecho referente a cifração:

```
1  def oaep_encrypt(message, public_key):
2
3      # trecho removido para abreviar o relatorio
4
5      seed = os.urandom(hash_length)
6      data_block_length = k - hash_length - 1
7      data_block_mask = mgf1(seed, data_block_length)
8
```

```
9       masked_data_block = int.from_bytes(data_block,
            'big') ^ int.from_bytes(data_block_mask,
            'big')
10      masked_data_block =
            masked_data_block.to_bytes(data_block_length,
            'big')
11
12      #mask for the message
13      seed_mask = mgf1(masked_data_block, hash_length)
14      masked_seed = int.from_bytes(seed, 'big') ^
            int.from_bytes(seed_mask, 'big')
15      masked_seed = masked_seed.to_bytes(hash_length,
            'big')
16
17      #OAEP padding
18
19      encoded_message = b'\x00' + masked_seed +
            masked_data_block
20
21      # RSA encryption
22      encrypted =
            rsa_cipher(int.from_bytes(encoded_message,
            'big'), public_key)
23      encrypted =
            encrypted.to_bytes((encrypted.bit_length() +
            7) // 8, 'big')
24
25      return encrypted
```

Para decifração, foi executado o processo inverso, sendo tanto a cifração quanto a decifração implementadas baseadas na explicação do seguinte link: Explicação OAEP. A função *mgf1* implementada foi baseada na indicada no link Mask generation function, utilizando a implementação de SHA3-256 disponibilizada na lib *hashlib*.

Finalmente, o arquivo *main.py* permitiu a utilização do RSA com OAEP na cifração de arquivos como indicado no trabalho. Para executá-lo, é necessário o seguinte comando para cifração:

```
python3 main.py cifrar arquivo_de_entrada arquivo_cifrado
```

No processo de cifração, são geradas as chaves publica e privada, que são armazenadas nos arquivos *public_key.pub* e *private_key* em BASE64, imitando o funcionamento do *ssh-keygen*. A cifração e decifração utilizam essas chaves com os respectivos nomes e dispensam a sinalização das chaves no comando. Abaixo, exemplo do comando quando a intenção for decifrar:

```
python3 main.py decifrar arquivo_cifrado arquivo_decifrado
```

O arquivo resultande da execução do comando acima entitulado como *arquivo_decifrado* deve ser igual ao arquivo cifrado inicialmente, *arquivo_de_entrada*.

3 Conclusão

A realização do trabalho permitiu verificar empiricamente como se dá o funcionamento de um algoritmo de criptografia assimétrica, e apresentou alguns desafios ao lidar com os números primos. O principal deles, foi o de uma implementação rápida ao gerar primos grandes de 1024 bits, o que necessitou de bibliotecas de paralelismo e idealmente requeriria a implementação em uma linguagem de mais baixo nível, preferencialmente compilada.

Embora demore um pouco, o número de verificações através de testemunhas escolhido no teste de primalidade foi 64, e foi escolhido por apresentar um alto nível de certeza com relação a primalidade dos números em questão para p e q de 1024 bits.