Segurança computacional
Trabalho 3

Guilherme Mattos - 17/0104508

Introdução

O trabalho a seguir implementa a cifração assimétrica pelo algoritmo RSA. Sem a utilização de bibliotecas específicas para criptografia.

Modos de operação

Existem três modos de operação determinados pelo primeiro argumento na linha de comando:

- 1) Modo de demonstração: O programa irá criar um par de chaves públicas e privadas, criptografar e descriptografar um arquivo de entrada. Criando dois arquivos com os sufixos “encrypted” e “decrypted”

- 2) Modo de criptografia: O programa irá criar um par de chave públicas e privada (d.txt n.txt) e criptografar um arquivo de entrada. Criando os arquivos n.txt, d,txt e “encrypted+<nome do arquivo>”

- 3) Modo de descriptografia: O programa irá ler dois arquivos de chave privada (d.txt n.txt) e descriptogrfar um arquivo de entrada (Para esse modo funcionar é necessário que os arquivos d.txt e n.txt existam). Criando assim o arquivo “decrypted+<nome do arquivo>”
-

É possível encontrar instruções de execução e descrição completa dos modos e exemplos no arquivo READ.ME .

Implementação:

A seguir serão discutidos os diversos passos para a criação de chaves e criptografia de um input.

1. Inicialmente é realizada a leitura de um arquivo em modo binário, esse dado é convertido para inteiro e se tornará o valor a ser criptografado.

2. Nesse passo temos a geração de um número primo muito grande de 1024 bits. Ou seja variando entre $(2^{(1023)} + 1 \ ; \ 2^{(1024)} - 1)$

```
def random_prime_generator():

    while(True):
        num = random.randrange(LOWER_PRIME_LIMIT, UPPER_PRIME_LIMIT,1)

        if simple_prime_test(num):
            if advanced_prime_test(num):
                break

    return num
```

3. Para garantir que esse número é primo, são realizados 2 testes:
    a. Teste de Eratosthenes: Consiste em dividir o número por uma sequência de números primos para determinar se ele é múltiplo de algum. Para essa implementação, foram testados os 1.000 primeiros primos

```
def simple_prime_test(num):  # Eratosthenes test
    if num%2 == 0:
        return False

    for prime in prime_list:
        if num%prime == 0:
            return False

    return True
```

b. Teste de Miller-Rabin: Consiste em testar se a propriedade de Fermat $candidato^{(num-1)} \equiv 1\ (mod\ candidato)$ é satisfeita para diferentes candidatos, sendo “num” o número que deseja-se testar a primalidade.

   Para realizar esse teste, inicialmente escolhemos um valor aleatório para um candidato de testemunho de primalidade. Em seguida, dividimos o valor $(num\ -\ 1)$ por 2 repetidas vezes até encontrar um número ímpar.

   Com esses valores disponíveis, é realizado o primeiro teste do teorema de Fermat, como foi mencionado acima.

   Após esse teste, temos o teste da expressão:

$$candidato^{2^r d} = -1\ (mod\ candidato),\ r \in \{0, 1, 2, \dots, s\}$$

   Sendo s o maior expoente tal que $2^s | (n\ -\ 1)$ (Referência)

   Caso o número passe no teste, temos um indício de que o número escolhido é primo. O erro associado ao teste é de $\frac{1}{4}$ por iteração. Para essa implementação, foram utilizadas 25 iterações, gerando um erro de $\left(\frac{1}{4}\right)^{25}$.

   Esse valor se mostra suficiente para a aplicação, porém para uso comercial, onde a segurança é um fator primordial, a literatura sugere pelo menos 40 iterações.

```python
def witness(candidate_witness, num):
    exp = num - 1

    while exp % 2 == 0:
        exp >>= 1

    congruent_try = pow(candidate_witness, exp, num)

    if congruent_try== 1 or congruent_try == num - 1:
        return True

    step = exp

    while step < (num-1):
        if pow(congruent_try,exp,num) == num - 1:
            return True
        step <<= 1

    return False

def advanced_prime_test(num):  # Miller-Rabin test

    for _ in range(MILLER_RABIN_ITERATIONS):
        candidate_witness = random.randrange(3, num - 1)

        if not witness(candidate_witness, num):
            return False

    return True
```

4. Com os dois primos p e q gerados, o número n é determinado por $n = pq$

5. A partir disso o número phi é determinado por $\phi(n) = (p - 1) * (q - 1)$

6. Tendo phi, são gerados números aleatórios menores que phi, até que seja encontrado um número *e* coprimo de phi.

```python
def e_generator(phi):

    while(True):
        e = random.randrange(3, phi-1,1)
        if (greatest_common_divisor(phi,e)==1):
            break

    return e
```

7. Por fim, é calculado o inverso multiplicativo de *e*, chamado de *d*

```python
def d_generator(e, phi):
    return pow(e,(-1),phi)
```

Com esses passos, foram criados os pares de chaves pública e privada.

Chave Pública: (n,e)

Chave Privada: (n,d)

A chave pública será utilizada para criptografia de um mensagem M, gerando uma cifra C, pela expressão:

$$C \equiv M^e (mod\ n)$$

A chave privada será utilizada para descriptografar uma cifra C, voltando à mensagem M original, pela expressão:

$$M \equiv M^d (mod\ n)$$

```python
def encrypt(M,e,n):
    return pow(M,e,n)

def decrypt(C,d,n):
    return pow(C,d,n)
```

Conclusão

A implementação do algoritmo RSA torna evidente seu brilhantismo. Baseando-se na dificuldade de determinar fatores de um primo extremamente grande, torna-se praticamente impossível derivar a chave privada a partir da chave pública. Com isso é possível ter uma comunicação criptografada sem o problema do envio das chaves (como ocorre na criptografia simétrica), aumentando, assim, a segurança dos dados enviados e permitindo a difusão de informação para qualquer um que o utilize.

É possível encontrar o código completo, instruções de uso e exemplos em [Github-Guilherme Camargo]