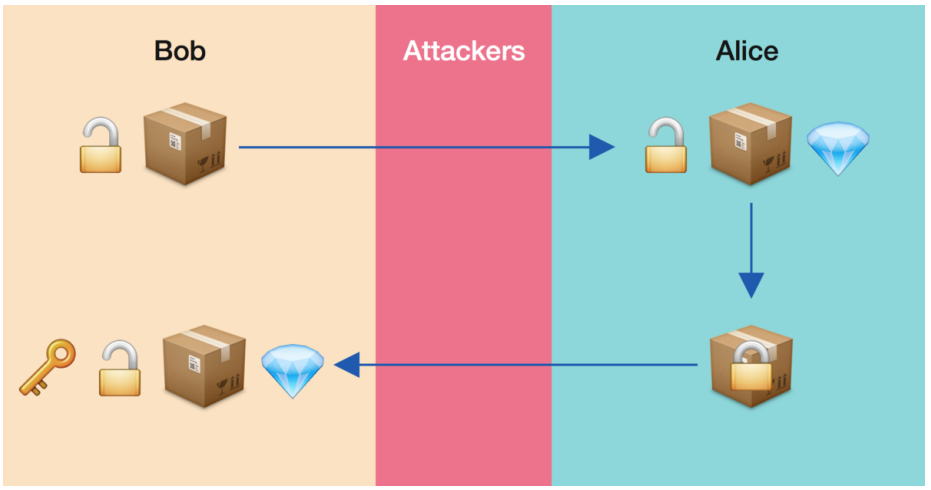
Trabalho 3 - RSA

Aluno: Daltro Oliveira Vinuto Mat: 160025966



UnB Departamento de Ciência da Computação

RSA - Processo:

1. São escolhidos dois primos ‘p’ e ‘q’ aleatoriamente.
2. ‘p’ e ‘q’ são compartilhados entre A, B que querem se comunicar. Chame fi(n) = (p-1)*(q-1)
3. A partir de ‘p’, ‘q’, A gera ‘e’, sendo que ele é escolhido como o coprimo de fi(n), isso é, para escolher ‘e’ basta encontrar um número entre (2, fi(n)) tenha MDC de 1 com f(n).
4. A manda ‘e’ para B.
5. B escolhe ‘d’ sendo ‘d’ a solucao da equacao: d*e mod f(n) == 1.
6. A partir daqui A pode mandar um numero ‘n1’ arbitrário de mensagem mensagens para B e B pode mandar um numero n2 arbitrário de mensagens para A pois a comunicação estará protegida pela criptografia RSA.

O processo que A segue para codificar uma mensagem M que ele quer mandar para B é o seguinte:

C = (M**e) mod n.

O processo que B segue para decodificar uma mensagem C que A mandou esse o seguinte:

M = (C**d) mod n.

O processo que B segue para mandar uma mensagem M para A e em seguida A decodificar e o simétrico inverso isso e cada um codifica com a chave pública e decodifica com a chave privada.

Detalhes de implementação:

Foi codificada a função chamada rsa_encode() que recebe uma mensagem na forma de string o valor de ‘n’ e de ‘e’ e ele então converte cada caractere para o código correspondente em unicode e então ela criptografa cada código de caractere usando RSA gerando assim uma lista de integers que é convertida para lista de bytes, essa lista de bytes é então retornada para a função que chamou rsa_encode().

Foi codificada a função chamada rsa_decode() que recebe uma lista de bytes, o valor de ‘n’ e de ‘d’ e então converte cada byte para o valor em integer correspondente e então decodifica usando RSA gerando assim uma lista de inteiros que por sua vez é convertida em caracteres unicodes sendo concatenados e assim a função retorna uma única string com a mensagem decodificada.

Segue print_screen:

```python
def rsa_encode(message: str, n: int, e: int) -> list[bytes]:
    values_list_message: list[int] = []
    values_list_encrypted: list[int] = []

    for char in message:
        value:int = ord(char)
        values_list_message.append(value)


    for char_code in values_list_message:
        char_encrypted:int = pow(char_code, e, n) # same as (message_int**e) % n
        values_list_encrypted.append(char_encrypted)

    encrypted_message: list[bytes] = []

    for char_code in values_list_encrypted:
        encrypted_message.append( char_code.to_bytes(4,"big") )

    return encrypted_message


def rsa_decode(encrypted_message: list[bytes], n: int, d: int) -> str:
    values_list_encrypted: list[int] = []
    values_list_decrypted: list[int] = []


    for value_byte in encrypted_message:
        value:int = int.from_bytes(value_byte, "big")
        values_list_encrypted.append(value)

    for char_code in values_list_encrypted:
        char_decrypted:int = pow(char_code, d, n) # same as (message_int**d) % n;
        values_list_decrypted.append(char_decrypted)

    decrypted_message: str = ""

    for char_code in values_list_decrypted:
        decrypted_message += chr(char_code)

    return str(decrypted_message)
```

Para calcular d usamos a funcao choose_coprime() que recebe fi(n) = (p-1)*(q-1) que tenta escolher entre primos comuns 65537, 257, 17, 5 porém se nenhum dele for coprimo com fi(n) então começa em 5 e busca o primeiro número que é coprimo com fi(n) isso é um número tal o máximo divisor comum dele e de fi(n) é igual a 1.

print screen abaixo:

```python
def choose_coprime(fi: int) -> int:
    e:int
    finded_coprime: bool = False

    list_primes: list[int] = [65537, 257, 17, 5]

    for value in list_primes:
        if (math.gcd(fi, value) == 1):
            finded_coprime = True
            e = value

    i: int = 5
    while not finded_coprime:
        if (math.gcd(fi, i) == 1):
            finded_coprime = True
            e = i
        i+= 2
        print(f"i: {i}")

    print(f"new value of e: {e}")
    return e
```

Para calcular d eficientemente usamos o algoritmo de euclides estendido, abaixo descrição do algoritmo:

Algorithm: Extended Euclidean algorithm. (Ref: [MENE97], Algorithm 2.107)

INPUT: Two non-negative integers a and b with $a \geq b$.
OUTPUT: $d = \gcd(a, b)$ and integers x and y satifying $ax + by = d$.

1. If $b = 0$ then set $d = a$, $x = 1$, $y = 0$, and return(d, x, y).
2. Set $x_2 = 1$, $x_1 = 0$, $y_2 = 0$, $y_1 = 1$
3. While $b > 0$, do
   a. $q = \text{floor}(a/b)$, $r = a - qb$, $x = x_2 - qx_1$, $y = y_2 - q\,y_1$.
   b. $a = b$, $b = r$, $x_2 = x_1$, $x_1 = x$, $y_2 = y_1$, $y_1 = y$.
4. Set $d = a$, $x = x_2$, $y = y_2$, and return(d, x, y).

Print screen abaixo:

```python
def inverse_of_modulo(a: int, b: int ) -> int:
    x1: int; x2: int; x3: int;
    y1: int; y2: int; y3: int;
    q:int

    if (b == 0):
        return 1

    x1, x2, x3 = 0, 1, b
    y1, y2, y3 = 1, 0, a
    while x3 != 0:
     q = y3 // x3
     x1, x2, x3, y1, y2, y3 = (y1 - q * x1), (y2 - q * x2), (y3 - q * x3), x1, x2, x3
    return y1 % b
```

Algoritmo de Miller-Rabin:

Como gerar os números primos ‘p’, e ‘q’ usando Miller-Rabin.

O algoritmo de Miller-Rabin funciona da seguinte maneira:

1. A partir de um ‘n’ gerado aleatoriamente.
2. Calculamos n-1
3. A partir de ‘n-1’ calculamos ‘s’ e ‘d’ tal que n - 1 = (2**s)*d, para isso calculamos 2**s começando com s==1 e enquanto (n-1) mod (2**s) == 0 incrementamos o ‘s’ em 1 unidade. Quando (n-1) mod (2**s) != 0 então utilizamos o valor anterior de ‘s’, isto é , o último valor de ‘s’ que resultou em (n-1) mod (2**s) == 0 e nesse ponto calculamos ‘d’ como d = (n-1)/(2**s).
4. Escolhamos ‘a’ aleatoriamente dentro do intervalo (2, n-2)
5. Calculamos ‘x’ tal que x == (a**d) mod n.
6. Calculamos s times y tal que y = (x**2) mod n (a**d) % number. Calculamos ‘s’ times ou até que seja detectado que ‘n’ não é primo.
   a. Após cada cálculo de y verificamos se é verdadeira a expressão: (y == 1 and x != 1 and x != n-1) se ela for verdadeira em qualquer iteração então o número ‘n’ não é primo.

    b. Substituímos x por y.
7. Se o loop anterior terminou porque calculou ‘y’ um número de vezes ‘s’ e mesmo assim não detectou que o número não é primo então verificamos se y != 1 se for verdade então o numero nao e primo.
8. Se o numero nao foi detectado como nao primo até esse ponto então ‘n’ é provavelmente nao composto isso é ele é provavelmente primo.
9. Se o número ‘n’ foi declarado como não primo, então geramos aleatoriamente outro ‘n’ e testamos novamente usando o algoritmo de Miller-Rabin voltando assim para o passo 1.

Problemas e Dificuldades encontradas:

1. A operação de base**power % value isso é de modulo nao era rápida suficiente por isso foi utilizada a função do python pow(a,b,c).
2. Os valores de Unicode precisaram ser convertidos para bytes antes de serem enviados e precisavam ser recebidos como uma lista de bytes. As conversões antes de enviar e após receber foram necessárias.
3. Embora difícil de entender foi necessário usar o algoritmo de euclides estendido para calculator d como solução de d*e = 1 (mod n) pois as outras abordagens eram muito lentas.

Bibliografia:

1. Slides da disciplina.
2. https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
3. https://www.di-mgt.com.au/euclidean.html#code-binarygcd
4. https://www.youtube.com/watch?v=8i0UnX7Snkc&t=366s&ab_channel=NesoAcademy
5. https://www.youtube.com/watch?v=qph77bTKJTM&ab_channel=TomRocksMaths
6. https://www.youtube.com/watch?v=4zahvcJ9glg&ab_channel=EddieWoo
7. https://www.youtube.com/watch?v=oOcTVTpUsPQ&ab_channel=EddieWoo
8. https://www.youtube.com/watch?v=ZwPGE5GgG_E&ab_channel=Udacity
9. https://www.di-mgt.com.au/rsa_alg.html#:~:text=To%20compute%20the%20value%20for,ed%3D1mod%CF%95.
10.