Trabalho de Implementação Segurança Computacional - Gerador / Verificador de assinaturas

Carlos Gabriel V.N. Soares, 18/0056298 - Noite
Eduardo Ferreira de Assis , 170102289 - Manhã

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)

cgvilasnovas@gmail.com, eduardoffassis@gmail.com

1. Introdução

Neste trabalho iremos implementar um gerador e verificador de assinaturas **RSA** (*Rivest–Shamir–Adleman*) utilizamos também o **teste de primalidade** *Miller–Rabin* para assegurar números primos. A implementação ocorreu em *Python 3* por motivos de facilidade e flexibilidade da linguagem.

2. Execução do programa

Para executar o programa devemos executar dentro da pasta *src/*. Onde passamos um argumento numérico com o número de bits de cada número primo. Na imagem a seguir, cada número primo terá 512 bits.

```
src$ python3 ./main.py 512
```

Figure 1. Código executado: "python3 ./main 512"

3. RSA

Primeiramente, foi estabelecido que a chave gerada *n* deveria conter no mínimo 1024 bits. Inicialmente devemos utilizar dois números para gerar *n*: p e q onde $n = p * q$ e p e q possuem 512 bits.

3.1. Cálculo dos números primos

Vamos utilizar números primos onde cada número gerado deve ter 512 bits. Para isto primeiramente geramos um número candidato a primo (dizemos candidato a primo pois apenas vamos assegurar que ele é primo mais à frente) através da seguinte função:

```python
def getRandomNumber(n):
    '''Random number of n bits.'''
    return random.randrange(2**(n-1)+1, 2**n - 1)

def getPrime(n):
    '''Generate a prime'''
    while True:
        prime_candidate = getRandomNumber(n)

        for divisor in initial_primes:
            if prime_candidate % divisor == 0 and divisor**2 <= prime_candidate:  # candidate is composite
                break  # not prime
            else:
                if isMillerRabinPassed(prime_candidate, 40):
                    return prime_candidate  # prime
                else:
                    break  # not prime
```

Figure 2. Função geradora de primos

A função recebe um número que define a quantidade de bits que o número gerado deve possuir. A função *getRandomNumber(n)* gera um número de n bits ímpar, fator que aumenta a probabilidade dele ser primo. Em seguida, comparamos com o módulo dos cem primeiros números primos para melhorar a performance do algoritmo. Uma vez que temos nosso número candidato a primo, realizamos o teste de primalidade de *Miller-Rabin*.

```
initial_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
                  31, 37, 41, 43, 47, 53, 59, 61, 67,
                  71, 73, 79, 83, 89, 97, 101, 103,
                  107, 109, 113, 127, 131, 137, 139,
                  149, 151, 157, 163, 167, 173, 179,
                  181, 191, 193, 197, 199, 211, 223,
                  227, 229, 233, 239, 241, 251, 257,
                  263, 269, 271, 277, 281, 283, 293,
                  307, 311, 313, 317, 331, 337, 347, 349]
```

Figure 3. Lista com os cem primeiros números primos

3.2. O Teste de primalidade Miller–Rabin

O teste de *Miller-Rabin* é um teste de primalidade probabilístico, ou seja, em uma única execução ele não é capaz de garantir se um número é primo ou não, entretanto caso executarmos o teste inúmeras vezes a chance de erro se torna insignificante. Para este programa, utilizamos 40 iterações.

```
def isMillerRabinPassed(candidate, iterations):
    '''Run Rabin Miller Primality check `iterations` times'''
    maxDivisionsByTwo = 0
    ec = candidate-1

    while ec % 2 == 0:
        ec >>= 1
        maxDivisionsByTwo += 1

    assert(2**maxDivisionsByTwo * ec == candidate-1)

    def _isComposite(round_tester):
        if pow(round_tester, ec, candidate) == 1:
            return False

        for i in range(maxDivisionsByTwo):
            if pow(round_tester, 2**i * ec, candidate) == candidate-1:
                return False
        return True

    for i in range(iterations):
        round_tester = random.randrange(2, candidate)
        if _isComposite(round_tester):
            return False

    return True
```

Figure 4. Função com o teste de *Miller-Rabin*

3.3. Geração das chaves

Em seguida calculamos *n=p*q*, calculamos o *totient de Carmichel* (geralmente indicado pela letra grega λ), onde: λ(n)=lcm((p-1),(q-1))
Em seguida devemos escolher um número inteiro e tal que 1 $<$ e $<$ λ(n), por convenção é comum escolher e = 65537, dentre as razões se encontram as que ele é um número primo, é um número relativamente alto, e seu cálculo é rápido(devido a relativamente poucos bits com valor 1). Para finalizar, obtemos a variável *d* que representa o inverso multiplicativo modular, ou seja, $d \equiv e^{-1}(mod\lambda(n))$

```
n = p * q    # RSA modulo

phi = (p - 1) * (q - 1) # Totient de Carmichael

d = libnum.invmod(e, phi)   # Inverso modular multiplicativo
```

Figure 5. Código para calcular aritmética RSA

3.4. Hashing

Para calcular o *hash* da mensagem vamos utilizar o algoritmo *SHA-3*, já implementado em *python* na biblioteca *hashlib*. Em seguida, vamos codificar para base64, garantindo assim a compatibilidade, assim como a função *SHA-3*, a codificação já esta implementada em bibliotecas *python*. Na última etapa antes de gerar o texto cifrado, convertemos para inteiros novamente.

```
# Gera hash
msg_hash = hashlib.sha3_256(message.encode())
# Codifica hash
encoded_bytes = base64.b64encode(msg_hash.hexdigest().encode('UTF-8'))
int_byte = int.from_bytes(encoded_bytes, 'big')      # cast para int
```

Figure 6. Função de hashing da mensagem

3.5. Criptografia

Para conseguirmos a mensagem cifrada *c* a partir do hash gerado anteriormente, basta calcularmos: $c = (m^e)mod\, n$, onde m é o hash gerado na mensagem anterior, $e = 65537$ e *n* o número de bits indicados pelo usuário. Em Python temos que a função *pow(m,e,n)* realiza automaticamente o cálculo almejado.

```
c = pow(int_byte, e, n)
```

Figure 7. Função para criptografar em python

3.6. Descriptografia

Uma vez que obtivermos a mensagem cifrada podemos utilizar a mesma função com argumentos diferentes para descriptografar-la. Temos então: $dc = (c^d)mod\, n$ onde *d* é o inverso multiplicativo modular calculado anteriormente. Agora devemos realizar o caminho "inverso", transformamos *dc* para bytes novamente e em seguida realizamos a decodificação utilizando a base64, por último transformamos em string.

```python
# Desencripta
dc = pow(c, d, n)
# print("Deciphered-int:\n%s\n" % dc)
dc = (dc).to_bytes(bytes_needed(dc), byteorder='big') # cast para bytes
encoded_str = base64.b64decode(dc)
res = str(encoded_str, 'UTF-8')
```

Figure 8. Função para descriptografar em python

3.7. Verificação

Vamos supor que A envia mensagem para B, A realiza a etapa de criptografia da mensagem, e enviar a cifra em conjunto com a mensagem. Para verificar a originalidade da mensagem, B utiliza do mesmo algoritmo de hash em conjunto com a chave publica e compara o hash da mensagem com o hash obtido, caso os dois sejam iguais, tem-se a certeza que a mensagem é original.