Trabalho de Implantação 2
Gerador/Verificador de Assinaturas

Paulo Victor França de Souza - 20/0042548
Thais Fernanda de Castro Garcia - 20/0043722

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0201 - Segurança Computacional (2022.1)
Prof. João José Costa Gondim - Turma 1

1. Introdução

Sendo um dos primeiros sistemas de criptografia de chave pública, RSA (Rivest-Shamir-Adleman) é abundantemente usado para transmissão segura de dados, sendo usado por exemplo, em mensagens de emails, compras on-line e entre outros. Este sistema se baseia em ter uma chave de encriptação pública diferente da chave de decriptação privada (secreta), onde esta assimetria é baseado na dificuldade prática da fatorização do produto de dois números primos grandes. É considerado um dos mais seguros, tendo várias tentativas sem sucesso de quebrá-lo, sendo também uma das grandes inovações em criptografia de chave pública e o primeiro algoritmo a possibilitar criptografia e assinatura digital.

O funcionamento do RSA se fundamenta na criação e publicação de uma chave pública baseada em dois números primos grandes, junto com um valor auxiliar, onde os números primos devem ser mantidos secretamente. Qualquer um pode usar a chave pública para encriptar a mensagem, porém apenas com grande conhecimento dos números primos pode decodificar a mensagem de forma viável, tendo esse problema chamado de problema RSA. [4] [3] [2] [1]

2. Implementação

A implementação do projeto consiste em um gerador e verificador de assinaturas RSA em arquivos, com as funcionalidades de geração de chaves, cifra simétrica, geração de assinatura e verificação.

2.1. Parte I: Geração de Chaves

Nesta etapa há a geração de chaves (p e q primos com no mínimo de 1024 bits), onde além da geração de chaves é feita também a leitura do arquivo .txt e sua passagem para base 64. Dessa forma, para a geração das chaves primeiramente são gerados 2 números primos ”p” e ”q” com no mínimo 1024 bits, os quais são verificados usando o teste de primalidade Miller-Rabin, um teste probabilístico da primitividade de um dado número n. Neste teste dado um numero n, caso ele não passe pelo teste, n com certeza é um número composto e se n passar no teste, ele é primo. A seguir, foi utilizado p e q para gerar n, que é o produto entre os dois, gerando também com ajuda de f(n) = (p-1)(q-1) os números ”e” e ”d”. Foi usado ”d” e ”n” para gerar a chave privada e ”e” e ”n” para gerar a chave pública.

```python
def miller_rabin(n,k): # Teste de primalidade Miller-Rabin.
    if n == 2 or n == 3:
        return True

    if n%2 == 0:
        return False

    r,s = 0, n-1
    while s%2 == 0:
        r+=1
        s//=2
    for _ in range(k):
        a = random.randrange(2, n-1)
        x = pow(a,s,n)
        if x == 1 or x == n-1:
            continue
        for _ in range(r-1):
            x = pow(x,2,n)
            if x == n - 1:
                break
        else:
            return False
    return True

def gera_primos(): # Geração de chaves primas com no mínimo 1024 bits com teste de primalidade (Miller-Rabin).
    n = random.getrandbits(1024)
    if miller_rabin(n,40) == True:
        return n
    return gera_primos()
```

Figure 1. Geração de chaves com teste de primalidade (Miller-Rabin).

2.2. Parte II: Cifra Simétrica

Nesta seção do código foi feita a geração de chaves simétrica e cifração simétrica de mensagem (AES modo CTR), onde houve a preocupação em primeiramente gerar as chaves simétricas, para isso foi usado um contador que consiste em um número de 16 bytes como chave. Já para a cifração houve o uso da cifra simétrica AES modo CTR, para isso primeiro é dividido o conteúdo da mensagem em blocos de 16 bytes, podendo ou não ser necessário a extensão da mensagem para que se gere blocos completos de 16 bytes, dessa forma percorremos os blocos formados da mensagem aplicando um ”ou exclusivo (xor)” do bloco atual com a chave atual, esse bloco resultante será concatenado à *string* do resultado da cifração, após cada laço da interação aumentamos o contador em 1 e damos continuidade cifrando o próximo bloco.

2.3. Parte III: Geração da Assinatura

A seguir, damos início a seção que é responsável pela geração da assinatura, isto é, feito o cálculo de *hashes* da mensagem em claro (função de *hash SHA-3*), assinatura da mensagem (cifração do *hash* da mensagem usando OAEP) e a formatação do resultado (caracteres especiais e informações para verificação em BASE64), onde foi feito primeiro o *hash SHA-3* da mensagem e posteriormente aplicando a cifração OAEP (*Optimal Asymmetric Encryption Padding*) no hash conquistado. O OAEP consiste em primeiramente denotar a mensagem original como ”M” e ela após o *padding* como ”PS” e ”s” dados aleatórios, assim podemos utilizar das duas funções determinísticas G e H para a geração de máscara, para obter DB usaremos da concatenação de ”PS”, o byte 0x01 e mensagem ”M”, usaremos do ”ou exclusivo (xor)” e da função ”G” para prosseguirmos com a criptografia e gerar a máscara de DB. Por fim iremos realizar a formatação do resultado em BASE64.

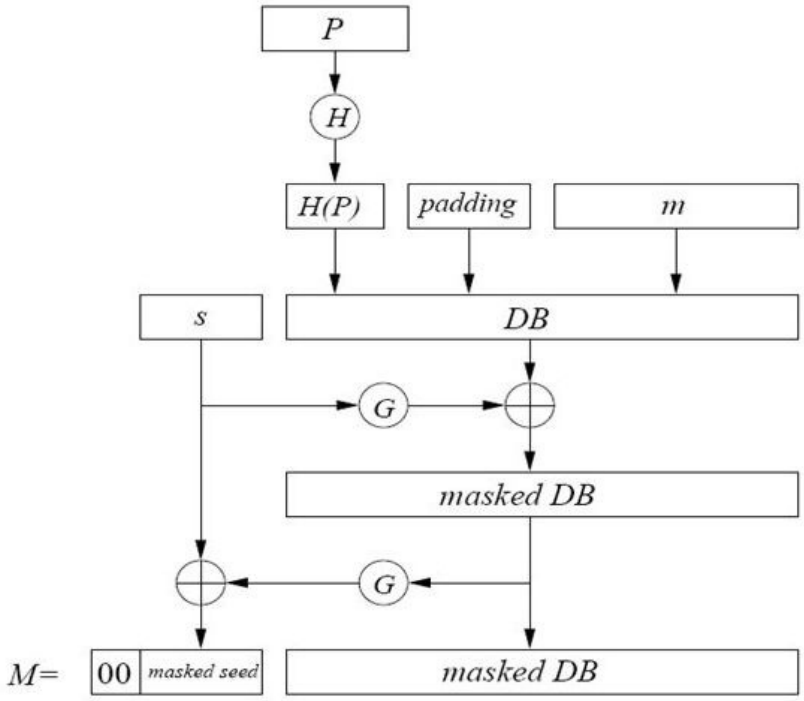

Figure 2. Algoritmo OAEP.

2.4. Parte IV: Verificação

Para a etapa final do projeto, será feito o *parsing* do documento assinado e decifração da mensagem (de acordo com a formatação usada, no caso BASE64), depois a decifração da assinatura (decifração do hash) e a verificação (cálculo e comparação do hash do arquivo). Iremos utilizar das funções de decifração, para isso, iremos traduzir a mensagem cifrada em um inteiro com a mesma quantidade de octetos do módulo da chave privada, dessa forma ,com a mensagem cifrada no formato inteiro, podemos prosseguir e fazer a mensagem elevada a ”d” com modulo ”n”. Dessa forma obtemos novamente o *hash SHA-3* da mensagem.

References

[1] B. Daniel and P. Rogaway. Optimal asymmetric encryption - how to encrypt with RSA.

[2] P. Daniel, M. Pitchaiah, and P. . Implementation of advanced encryption standard algorithm.

[3] B. Lynn. Miller-rabin test.

[4] Wikipédia, a enciclopédia livre. RSA (sistema criptográfico).