Implementação de um Gerador e Verificador de Assinaturas RSA

Eduardo Freire dos Santos, 211010299
Ruan Petrus Alves Leite, 211010459

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
Segurança Computacional - Turma Noturna

`dudufreiresantos@gmail.com, 211010459@aluno.unb.br`

1. Introdução

Nesse relatório é descrito o sistema de criptografia de chave pública RSA e o algoritmo OAEP. Além disso, é feito um programa em Python que implementa o RSA utilizando OAEP, permitindo cifrar, decifrar, assinar e verificar a assinatura de mensagens.

O repositório com o programa implementado pode ser encontrado em `github.com/duduFreire/compsec-03`.

2. Funcionamento do RSA

De agora em diante, dado $a \in \mathbb{Z}$ e $n \in \mathbb{Z}^+$, denotaremos por $a \,\%\, n$ o único inteiro x tal que $0 \leq x < n$ e $x \equiv a \pmod{n}$.

2.1. Geração de chaves

Para cifrar e decifrar mensagens com RSA é necessário primeiro gerar dois pares de números (n, e) e (n, d), conhecidos como chave pública e chave privada, respectivamente.

Para isso, geram-se dois números primos aleatórios p e q suficientemente grandes e tomamos $n = pq$. Feito isso, escolhe-se também um número e coprimo a $\phi(n)$ aleatório, em que ϕ é a função tociente de Euler (note que $\phi(n) = (p-1)(q-1)$). Pela coprimalidade, existe um único natural $0 < d < n$ tal que $ed \equiv 1 \pmod{\phi(n)}$. Tendo os números n, e e d, tomamos o par (n, e) como a chave pública e (n, d) como a chave privada.

2.2. Cifragem e Decifragem

Suponha agora que Alice deseja mandar uma mensagem m para Bob (assumiremos que m é um número natural). Primeiramente, é necessário que Bob gere um par de chaves conforme o processo descrito acima e disponibilize sua chave pública para Alice. É importante que o n gerado seja maior que m. Feito isso, Alice calcula $c = m^e \,\%\, n$ e envia a mensagem cifrada c para Bob.

Tendo em mãos c, Bob utiliza sua chave pública para calcular o valor $c^d \,\%\, n$. Mas note que $c^d \equiv m^{ed} \pmod{n}$. Como $ed = 1 \pmod{\phi(n)}$, temos um $k \in \mathbb{Z}$ tal que $ed = 1 + k\phi(n)$. Daí, segue que

$$
\begin{aligned}
c^d &\equiv m^{ed} \pmod n \\
&= m^{1+k\phi(n)} \\
&= m(m^{\phi(n)})^k \\
&\equiv m \pmod n
\end{aligned}
$$

em que a última equivalência segue do Teorema de Euler.

Dessa forma, reduzindo c^d módulo n Bob recupera a mensagem mandada por Alice.

2.3. Assinatura e Verificação

Suponha agora que Alice deseja mandar uma mensagem a Bob de tal forma que Bob consiga verificar que a mensagem foi mandada por ela.

Para isso, é necessário que Alice e Bob gerem pares de chaves. Denotaremos a chave pública de Alice por (n_a, e_a) e a privada por (n_a, d_a). Similarmente, as chaves de Bob serão (n_b, e_b) e (n_b, d_b). Ambos disponibilizam suas chaves públicas ao outro.

Para enviar sua mensagem m para Bob, primeiro Alice computa $c = m^{e_b} \,\%\, n_b$, conforme feito anteriormente. Em seguida, Alice utiliza alguma função de hash h e calcula $s = h(m)^{d_a} \,\%\, n_a$ e envia o par (c, s) para Bob. O número s é chamado de assinatura da mensagem.

Agora Bob pode decifrar a mensagem c de Alice como antes, recuperando m. Para verificar a assinatura de Alice, Bob calcula o valor $s^{e_a} \,\%\, n_a$. Analogamente ao que foi demonstrado acima, note que $s^{e_a} = (h(m)^{d_a})^{e_a} \equiv h(m) \pmod{n_a}$. Assim, Bob calcula o valor $h(m) \,\%\, n_a$ e verifica se este coincide com $s^{e_a} \,\%\, n_a$. Se os valores forem iguais, a mensagem muito provavelmente foi gerada por Alice, já que apenas Alice conhece o valor d_a necessário para gerar a assinatura correta da mensagem. Se os valores forem distintos, certamente a mensagem não foi enviada por Alice.

3. Funcionamento do OAEP

Foi usado o esquema de OAEP para fazer o padding da mensagem. O OAEP adiciona um elemento randômico ao RSA, e uma permutação de uma porta para prevenir qualquer vazamento de informações da mensagem.

A codificação do OAEP funciona nos seguintes passos:

- $lHash = Hash(L)$
- $PS = 0x00$, onde $|PS| = k - mLen - 2hlen - 2$ bytes.
- $DB = lHash|PS||0x01|M$
- Gera uma seed aleatória, $seed$
- $dbMask = MGF(seed, -k - hLen - 1)$
- $maskedDB = DB \oplus dbMask$
- $seedMask = MGF(maskedDB, hlen)$
- $maskedSeed = seed \oplus seedMask$
- o resultado é $EM = 0x00||maskedSeed||maskedDB$

Onde:

- $Hash$ é uma função de hash
- MGF é a função geradora de mascara
- $hlen$ É o tamanho do output da função de hash em bytes
- k é o tamanho do módulo de RSA em bytes
- M é a mensagem
- L é uma label opcional
- PS é uma byte string de null-bytes

Foi utilizada a função $MGF1$ como MGF e a função $hashf\_sha3\_256$ como função de hash.

4. Implementação do RSA

4.1. Exponenciação binária

Todas as etapas do RSA (cifragem, decifragem, assinatura e verificação) envolvem a exponenciação de números naturais. Logo, é essencial que essa operação possa ser feita rapidamente. A seguinte equação

$$a^b \,\%\, n = \begin{cases} 1, & b = 0 \\ (a^{b/2})^2 \,\%\, n, & b \,\%\, 2 = 0 \\ a(a^{(b-1)/2})^2 \,\%\, n, & b \,\%\, 2 = 1 \end{cases}$$

demonstra como calcular $a^b \,\%\, n$ utilizando apenas $O(\log b)$ multiplicações. Dessa forma as exponenciações são executadas de maneira relativamente rápida.

4.2. Teste de Primalidade de Miller–Rabin

Para gerar o par de chaves é necessário escolher dois números primos aleatórios grandes (normalmente em torno de 2048 bits). Para isso, necessita-se de uma maneira rápida de conferir se um número é primo.

O teste de primalidade escolhido foi o de Miller-Rabin, descrito por exemplo em [Schoof 2004]. Este é um teste probabilístico que recebe um número n cuja primalidade será testada e um número k de iterações. Se o teste retorna verdadeiro então há uma chance de $1 - 1/4^k$ de n ser primo. Se o teste retorna falso, então n certamente é composto. Por exemplo, escolhendo $k = 5$ iterações a chance de um falso positivo é de 1 em 1024. A complexidade desse teste é dada por $O(k\, log(n)^3)$.

Tendo esse teste em mãos, o procedimento para gerar um primo grande aleatório é simples: gere um número aleatório grande n. Enquanto n não é julgado como primo pelo teste de Miller-Rabin, incremente n.

4.3. Coprimalidade e Inverso multiplicativo

Durante a geração das chaves também é necessário gerar um número aleatório e coprimo a $(p-1)(q-1)$. Para isso utiliza-se o algoritmo de Euclides, que encontra o máximo divisor comum de dois inteiros a e b em tempo $O(\log(\min(a, b)))$. O mesmo algoritmo pode ser utilizado para encontrar o inverso multiplicativo de e.

References

[Schoof 2004] Schoof, R. (2004). Four primality testing algorithm). https://www.mat.uniroma2.it/~schoof/millerrabinpom.pdf. [Online; accessed 1-December-2023].