UFV - Universidade Federal de Viçosa
DPI - Departamento de Informática
Prof. André Gustavo dos Santos
INF 230 - Matemática Discreta - 2022/1

Trabalho 2
Para terça, 2/ago/22

Teoria dos números – criptografia RSA

Neste trabalho devem ser implementadas funções auxiliares para suporte so sistema de criptografia de chave pública RSA, bem como programas para codificação, decodificação e assinatura digital nesse sistema. Um resumo do funcionamento do RSA é dado a seguir. Mais detalhes podem ser vistos no material da disciplina.

1. Introdução

Cada participante possui uma chave pública (n, e) e uma chave secreta d:

- n é multiplicação de dois primos, p e q
- e é um número coprimo com $\phi = (p-1)(q-1)$, ou seja, $\text{mdc}(e, \phi) = 1$
- d é o inverso de e módulo ϕ, ou seja, $de \equiv 1 (\text{mod } \phi)$

Uma mensagem $M = [m_1, m_2, \ldots, m_k]$ é codificada com a chave pública do destinatário:

$$c_i = {m_i}^e \bmod n$$

A mensagem $C = [c_1, c_2, \ldots, c_k]$ é enviada ao destinatário, que a decodifica com sua chave secreta:

$$m_i = {c_i}^d \bmod n$$

A segurança do método está na dificuldade de se fatorar números grandes. Não é conhecido um método para se encontrar d a partir de n e e sem conhecer ϕ. O valor de ϕ é facilmente calculado conhecendo-se p e q, mas se n é grande, é computacionalmente inviável descobrir p e q.

2. Chaves

Para gerar as chaves deve-se então:

1. Escolher dois primos grandes p e q
   Para gerar um primo grande, pode-se usar o pequeno teorema de Fermat, que afirma que, se p é primo, então $a^{p-1} \equiv 1 (\text{mod } p)$ para todo a coprimo de p. A ideia é geraar um número aleatório p e fazer esse cálculo para vários números a com $\text{mdc}(a, p) = 1$. Se $a^{p-1} \not\equiv 1 (\text{mod } p)$, então p certamente não é primo. Se $a^{p-1} \equiv 1 (\text{mod } p)$ para vários a, então p é muito provavelmente primo.
2. Calcular $n = pq$
3. Calcular $\phi = (p-1)(q-1)$
4. Escolher e com $\text{mdc}(e, \phi) = 1$
   Pode-se gerar números aleatórios até encontrar um que atende a propriedade.
5. Calcular d com $de \equiv 1 (\text{mod } \phi)$
   Deve-se calcular o inverso de e módulo ϕ. Isso pode ser feito pelo algoritmo de Euclides estendido. Dados a e b, o algoritmo encontra os coeficientes de Bézout s e t com $sa + tb = \text{mdc}(a, b)$. Assim, para os valores e e ϕ, o algoritmo encontra os coeficientes s e t com $se + t\phi = 1$, já que $\text{mdc}(e, \phi) = 1$. Fazendo módulo ϕ temos $se + t\phi \equiv 1 (\text{mod } \phi)$, logo $se \equiv 1 (\text{mod } \phi)$ pois $t\phi \equiv 0 (\text{mod } \phi)$. Portanto, d é o coeficiente de Bézout de e, o valor s encontrado pelo algoritmo de Euclides estendido no cálculo de $\text{mdc}(e, \phi)$. (obs. o algoritmo pode retornar $s < 0$; nesse caso, basta somar ϕ: $d = s + \phi$).

3. Codificação

Considere que a mensagem é uma sequência de letras maiúsculas, A até Z. Cada uma é transformada em um inteiro de 2 dígitos: A = 00, B = 01, ..., Z = 25, que são agrupadas em blocos, formando uma sequência numérica $M = m_1, m_2, \ldots, m_k$. O tamanho de cada bloco é o maior tal que 2525...25 não ultrapassa n. Cada bloco m_i é codificado em um bloco c_i por $c_i = {m_i}^e \bmod n$, formando uma sequência numérica $C = c_1, c_2, \ldots, c_k$, que é a mensagem codificada enviada ao destinatário.

4. Decodificação

A mensagem codificada $C = c_1, c_2, \ldots, c_k$ é decodificada em $M = m_1, m_2, \ldots, m_k$ pelo destinatário por $m_i = {c_i}^d \bmod n$. A sequência numérica obtida é transformada de volta em letras: 00 = A, 01 = B, ..., 25 = Z.

5. Assinatura

Uma mensagem $M = m_1, m_2, \ldots, m_k$ pode ser assinada por $a_i = {m_i}^d \bmod n$, criando uma mensagem assinada $A = a_1, a_2, \ldots, a_k$, ou seja, codificada com a própria chave secreta. A autenticidade pode ser verificada usando-se a chave pública, formando de volta a mensagem original por $m_i = {a_i}^e \bmod n$. Este cálculo com e só retorna a mensagem original se ela tiver sido codificada (assinada) com d, secreta.

6. Operações auxiliares

- Potenciação modular[1]: $a^b \bmod n$
  Esse cálculo é usado a todo momento, seja na geração das chaves (no pequeno teorema de Fermat) quanto na codificação e decodificação. Como o expoente b é grande, não é viável fazer na ordem de b multiplicações. Deve-se usar potenciação modular rápida. Uma forma foi mostrada em aula, que é calcular $a, a^2, a^4, a^8, \ldots$ e multiplicar as potências correspondentes a bits 1 na representação binária de b. Por exemplo, $a^{42} = a^2 \cdot a^8 \cdot a^{32}$. Note que são $\log_2 b$ multiplicações para calcular as potências intermediárias, $a^2 = a \cdot a, a^4 = a^2 \cdot a^2, a^8 = a^4 \cdot a^4, \ldots$ e na ordem de $\log_2 b$ multiplicações ao final, no pior caso. Outra é por divisão-e-conquista, calcular recursivamente potências com metade do expoente. Para calcular a^b, calcula-se $x = a^{\lfloor b/2 \rfloor}$. Se b é par, retorna $x \cdot x$; se b é ímpar retorna $x \cdot x \cdot a$; Condição de parada, $b = 1$, retorna a.
- Algoritmo de Euclides: $\text{mdc}(a, b)$
  Usado em várias etapas da geração de chaves, para verificar números coprimos (mdc = 1).
- Algoritmo de Euclides estendido: $\text{mdc}(a, b)$ e s, t com $sa + tb = \text{mdc}(a, b)$
  Usado no cálculo de d, o inverso de e módulo ϕ.
- **Aritmética modular**
  Note que o método usa módulo n e módulo ϕ em várias operações. Para evitar *overflow*, TODAS as operações devem ser feitas usando-se aritmética modular. Por exemplo, note que a codificação e a decodificação são feitas módulo n. Mas NÃO se deve calcular ${m_i}^e$ primeiro para depois tirar módulo n, pois ${m_i}^e$ é um número gigante. Deve ser feito módulo n em TODAS as multiplicações e adições intermediárias. Por exemplo, um cálculo $x \cdot y \cdot z \bmod n$ não deve ser feito $(x \cdot y \cdot z) \bmod n$, mas da seguinte forma: $((x \cdot y) \bmod n) \cdot z) \bmod n$.

[1] Há calculadoras online para potenciação modular, onde pode verificar seus resultados:
https://www.boxentriq.com/code-breaking/modular-exponentiation
https://www.mtholyoke.edu/courses/quenell/s2003/ma139/js/powermod.html
https://www.dcode.fr/modular-exponentiation

7. Tamanho das chaves

Desde 2015, a recomendação para o RSA é usar chaves de 2048 bits. Algumas linguagens, como Python e Java, permitem usar inteiros de precisão arbitrária. Outras não, sendo necessário implementar tipos personalizados que suportem esse tamanho juntamente com as operações aritméticas e outras.

Por questão de prazo de implementação, o trabalho deverá ser feito com números que cabem em inteiros de 64 bits. De fato, de 32 bits, para permitir cálculos intermediários dentro do limite de 64 bits. Por exemplo, se a, b têm mais de 32 bits, o cálculo $a \cdot b \bmod n$ terá um valor intermediário com mais de 64 bits (ab) antes de tirar módulo n. Por isso, p e q devem ser escolhidos de tal forma que n tenha até 32 bits (valor até $2^{31} - 1$, considerando variável *signed*).

No caso, o número n deve ter valor em torno de 1 ou 2 bilhões. Não existe segurança com chaves deste tamanho, pois a chave pública n pode ser fatorada relativamente rápido. Mas usaremos este tamanho devido ao tempo até o final do semestre.

Certifique-se de usar inteiro de 64 bits! Por exemplo, em C/C++, se `sizeof(long long int)` *vale 8.*

O que fazer e o que entregar

- Implementar as funções auxiliares mencionadas.
  Algumas linguagens contêm funções prontas, por exemplo, o método ModPow da classe BigInteger do Java, que calcula potenciação modular. Um dos objetivos do trabalho é conhecer a implementação das operações, então NÃO deve usar funções prontas da linguagem, mas uma implementada no código. Se usar implementações de terceiros, cite a fonte, e que esteja explícito o código delas.
- Implementar a geração de chaves do RSA e gerar suas chaves pública e secreta
- Implementar a codificação: dada a chave pública e uma mensagem, gerar a mensagem codificada
  Deve aceitar mensagens de texto (letras maiúsculas) ou numéricas (texto já convertido em números).
- Implementar a decodificação: dada a chave secreta e uma mensagem, decodificar a mensagem
  Deve aceitar mensagens numéricas.
- Implementar a assinatura e a verificação da assinatura
  (note que as operações acima têm praticamente o mesmo código, pode reusá-las à vontade)
- Usar o link disponível no PVANet para divulgar sua chave pública
- Usar o link disponível no PVANet para escrever mensagem codificada para pelo menos 3 destinatários
- Decodificar as mensagens que forem enviadas para você(s) no link anterior
- Usar o link disponível no PVANet para enviar mensagens assinadas (umas sim, outras não)
- Escolher algumas mensagens no link acima e verificar quais foram assinadas pelo remetente
- Descobrir a chave secreta de alguém por meio da chave pública correspondente (são valores "pequenos")
- Escrever um relatório contendo descrição das funções implementadas e os resultados dos passos acima
- Enviar os códigos implementados compactados em formato ZIP e o relatório em formato PDF