Primeiro Projeto de Matemática Discreta

Henrique da Silva
hpsilva@proton.me

14 de agosto de 2022

Sumário

1 Introdução

Neste relatório, vamos discutir e implementar o sistema RSA.

Para ter as ferramentas necessárias para construí-lo, primeiro precisamos construir algumas ferramentas, estas serao discutidas nas seções 2, 3, 4, e 5.

A inspiração para escolha do projeto de chaves RSA é que algo que eu já utilizo diariamente, a autenticação feita pelo *git* utiliza um par privado e público de chaves RSA.

Objetivo do projeto é conseguir gerar um par seguro de chaves *RSA* e utilizá-lo para fazer minha autenticação no *git*.

Todos arquivos utilizados para criar este relatório, e o relatorio em si estão em: `https://github.com/Shapis/ufpe_ee/tree/main/4thsemester/`

2 O Codificador de texto

Este foi criado para transformar uma *string* de texto em um *int*, através de dois métodos. TextoParaInteiro(string) e InteiroParaTexto(int).

2.1 TextoParaInteiro

Este método recebe um texto e o torna em um m do tipo *int* da seguinte maneira:

$$m = \sum_{i=0}^{N-1} cod(a_i) * 27^i \tag{1}$$

Com *a* ate *z* sendo definidos como 1 ate 26, "espaco"sendo definido como 27.

2.2 InteiroParaTexto

Para retornar o texto, este método recebe um inteiro m e faz a seguinte operação:

$$a_i = cod\left(\frac{m}{27^i} \pmod{27}\right) \tag{2}$$

Para todo i que nao faca m ser menor que 1

2.3 Restrições e limitações

A principal restrição é que isto foi implementado usando o tipo int do $C\#$ que tem 32 bits. Porém, já que ele contém tanto números positivos quanto negativos o valor máximo dele é de:

$$\frac{2^{32}}{2} - 1 = 2147483647 \tag{3}$$

Estamos codificando o texto de maneira que cada dígito ocupa ate: $2^N = 27$, $N = \frac{\log 27}{\log 2}$ bits

Então a quantidade máxima de bits ocupados eh simplesmente $N * L$

Para o nosso caso em específico, que o tipo int tem $2^{31} - 1$ de tamanho, ou seja, seguramente ate 31 bits. Temos que:

$$L * \frac{\log 27}{\log 2} \leq 31 \tag{4}$$

Que nos dá $L = 6$, ou seja, podemos seguramente converter ate 6 caracteres para tipo int e convertê-los de volta.

Vale notar, que isto é um limite inferior de seguranca. Na verdade temos 6.3 dígitos disponíveis, que nos permitiria por exemplo, guardar e recuperar, uma frase de sete dígitos do tipo $zzzzzd$, Mas para ter certeza. Tem de ser 6 ou menos dígitos.

3 A classe *BigNumber*

Esta será uma classe que armazenará os números que utilizarei para a criação do RSA.

Utilizarei como base para meu $BigNumber$ a classe $BigInteger$ do $C\#$, que tem limite de tamanho tão grande quanto couber na memória do computador que o está utilizando.

Para o nossos fins, queremos um $BigNumber$ que tenha no maximo 2048 bits. Então para todas operações de $BigNumber$ incluindo a sua própria criação, criarei um $SafetySizeCheck$ que caso o $BigNumber$ exceda 2048 bits, ele irá lançar uma exceção e parar o programa com a mensagem de erro apropriada.

Importante lembrar que inclui o zero no $BigNumber$, entao na verdade o limite superior dele fica da seguinte maneira:

$$BigNumber \leq 2^{2048} - 1 \tag{5}$$

É também importante lembrar que todas operações de checagem de segurança ocorreram ***após*** a operação ser realizada.

Ou seja, o programa permitirá operações inseguras, desde que o $BigNumber$ resultante desta operação insegura não exceda 2048 bits.

3.1 Multiplicacao de *BigNumber*

Aqui podemos observar o seguinte:

$$2^a * 2^b = 2^{a+b} \tag{6}$$

Então a multiplicação de dois $BigNumber$ de tamanho a e b, pode no máximo nos dar um $BigNumber$ de tamanho $a + b$

3.2 Soma de *BigNumber*

Neste caso temos o seguinte:

$$2^a + 2^a = 2 * (2^a) = 2^1 * 2^a = 2^{a+1} \tag{7}$$

Logo podemos concluir que no máximo a soma de dois números de tamanho N bits dará um numero de tamanho $N + 1$ bits.

3.3 Codificando numeros grandes

Inicialmente, notamos que o ***Codificador de Texto*** discutido na seção dois, tinha limitação de utilizar o tipo *int* de 31 bits. Que não será suficiente para nossos propósitos.

Então escrevi dois novos métodos ***TextoParaBigNumber***, e ***BigNumberParaTexto***

Estes tendo as mesmas limitações porém alterando o tamanho do nosso número de 31 bits para 2048 bits.

Resolvendo a equação (4) para 2048, teremos que o nosso L deve se limitar a no máximo 430 caracteres.

4 Aritmetica Modular

4.1 AddMod

As limitações aqui sao as mesmas da soma de dois $BigNumber$ como vimos acima em (7).

A função AddMod pode no máximo dar um $BigNumber$ de tamanho $N + 1$ bits, N sendo o tamanho do maior dos dois $BigNumber$.

4.2 MulMod

Vimos acima em (6) as limitações de multiplicação de dois $BigNumber$.

Então no máximo a soma dos tamanhos em bits dos nossos $BigNumber$ deve dar 2048 que eh o tamanho que escolhemos para o nosso $BigNumber$

4.3 ExpMod

No caso da exponencial precisamos que o produto dos tamanhos dos dois $BigNumber$ seja menor que 2048

4.4 InvMod

Para resolver a congruência linear utilizamos o algoritmo de euclides estendido. E a operação de máxima ordem que utilizamos em todas operações eh a de multiplicação de $BigNumber$ que descrevemos em (6)

Logo, nossa limitação para garantir que não vamos exceder os **2048** bits do $BigNumber$ eh que a soma em pares, de a, b, e n não exceda 2048 bits.

5 Busca por números primos

utilizarei o método de Miller Rabin para testar a primalidade dos números.

5.1 Testando os números dados:

$2^{521} - 1$	$\rightarrow$	primo
$2^{523} - 1$	$\rightarrow$	nao primo
$2^{607} - 1$	$\rightarrow$	primo

5.2 Achando novos primos:

Para achar novos números primos criarei um novo $BigNumber$ de tamanho n, e testarei numeros impares menores que este $BigNumber$ ate o teste de Miller Rabin me retorna que provavelmente é um primo.

Para adicionar um elemento de aleatoriedade. Após checar um número, vamos subtrair a este $2 * x$ com x variando entre 0 e n aleatoriamente.

Esta maneira de gerar números aleatórios, me garante que todo número gerado terá tamanho menor do que n bits. Porém, ela me retorna números primos repetidos com alguma frequência.

Com alguns testes de números, tive a chance de aproximadamente 1 em 5000 de obter dois números primos idênticos. O que para um sistema que sofre ataques com certeza não seria aleatoriedade suficiente. Mas para nossos propósitos de testar a criação de um sistema RSA que não vai sofrer ataques. Será o suficiente.

6 O Sistema RSA

Vamos utilizar de todas ferramentas que criamos acima para montar o nosso sistema.

6.1 Geração do par de chaves RSA

Inicialmente vamos achar dois números primos, p, e q de 512 bits aleatórios.

Vamos definir $n = pq$ e $\phi(n) = (p-1)(q-1)$

Vamos também escolher um ϵ tal que $MDC(\epsilon, \phi(n)) = 1$. Para simplicidade, escolhi o menor ϵ para qual esse MDC seja 1.

E através do sistema de resolução de congruências lineares que criamos na seção 4.4, vamos computar um d da seguinte maneira:

$$\epsilon d \equiv 1(mod\phi(n)) \tag{8}$$

Então, afinal temos um ϵ e um n que serão nossa chave pública, e um d que será nossa chave privada.

Ambos chaves privadas e públicas serão salvas em arquivo.

6.2 Codificando e decodificando

Vamos utilizar do nosso ***Codificador de Texto*** nesta etapa.

Com tudo que temos em mãos, os passos para codificação e decodificação são simples como veremos a seguir:

6.2.1 Codificacao

$$Y \equiv X^\epsilon \pmod n \tag{9}$$

Onde X é a mensagem em texto limpo. Y a mensagem criptografada, e ϵ e n são a chave pública.

Isto foi implementado na classe RSA, e para simplificar, $RSA.Codificar$ recebe textos limpos e a chave pública, e retorna textos criptografados ao invés de cifrar o $BigNumber$

6.2.2 Decodificando

$$Z \equiv Y^d \pmod n \tag{10}$$

Onde Z é a mensagem em decodificada. Y a mensagem criptografada, e d é a chave privada.

Importante de notar que o seguinte eh verdade:

$$Z \equiv X \pmod n \tag{11}$$

Isto foi implementado na classe RSA, e para simplificar, $RSA.Decodificar$ recebe textos cifrados e a chave privada, e retorna textos limpos

6.3 Testando para um texto pré-determinado

Quando executado o programa irá mostrar um texto ao usuário, codificá-lo, mostrar o texto codificado, e após isso decodificar e mostrar o texto decodificado. O texto decodificado deve ser o mesmo que o texto original.

O embasamento teorico disto esta descrito na secao 6.2.

6.4 Testado para texto arbitrario Y

O programa irá pedir ao usuário um texto, vamos codificar este texto, exibi-lo codificado, e após isso decodificar e exibi-lo novamente. O texto decodificado deve ser o mesmo que o texto original.

Lembrando que há limitação de tamanho do texto em 430 caracteres como visto na seção 3.3

6.5 Assinatura digital

Para assinar um documento X, basta divulgar o Y de:

$$Y \equiv X^d \pmod n \tag{12}$$

E para verificar, fazemos o oposto que é:

$$Z \equiv Y^d \pmod n \tag{13}$$

Lembrando da equação (10) que nos dá que $Z \equiv X$ já que:

$$Z \equiv X^{de} \equiv X \pmod n asdas \tag{14}$$

Nesta etapa, o programa irá assinar o texto que o usuário escreveu anteriormente, e verificar a assinatura.

Importante notar que assinatura é um caso parecido com o de codificação de texto. Exceto que neste caso vamos cifrar o texto limpo com nossa chave privada, e decodificar o texto cifrado com a chave pública.

6.6 Autenticação de remetente e destinatário

Neste caso, temos dois usuários A e B.

O usuário A fará a codificação da mensagem com a chave pública de B, assim garantido que só B lerá a mensagem, e envia junto com a mensagem a sua assinatura digital.

O usuário B usara sua chave privada para decodificar a mensagem, e usara a chave pública de A para verificar a assinatura digital.

O programa vai guiar o usuário A em enviar uma mensagem assinada para o usuário B, e o usuário B verificará que a mensagem foi assinada por A.

7 Conclusões

Neste projeto conseguimos criar um sistema simples que criptografa, descriptografa mensagens de texto, e assina mensagens.

As maiores vulnerabilidades que enxergamos no projeto sao a geração dos números primos, ja que o método "aleatório"que usamos não foi um método extremamente robusto, havendo a possibilidade de repetição de números primos com probabilidade em torno de $1/5000$

Também há problemas com a segurança do arquivo de texto que contém as chaves privadas. Qualquer acesso indevido a ele, permitiria leitura das mensagens por terceiros.

Mas em geral, creio que atingimos os objetivos que foram descritos na introdução do projeto.

Alterei as chaves que uso para acesso a git por ssh para chaves geradas por este programa, mesmo que haja estes problemas descritos acima.

Creio que o maior ganho de segurança será trabalhando na aleatoriedade dos primos gerados.