Implementação e Análise de Complexidade do Sistema Criptográfico RSA

Brayan Eduardo Rosa[1], **Paulo Henrique Cuchi**[1]

[1]Departamento de Ciência da Computação
Universidade do Estado de Santa Catarina (UDESC)
Joinville – SC – Brazil

brayan_sx@hotmail.com, paulo.cuchi@gmail.com

Resumo. *Este artigo tem como objetivo levantar uma análise em torno do sistema de criptografia assimétrica RSA, abordando tópicos quando o seu funcionamento, implementação e complexidade. O documento também descreve a base teórica e matemetica que dão base a aplicação dos algoritmos que compõem o sistema de criptografia.*

1. Introdução

No inicio do século XX, a criptografia já era considerada uma ferramenta muito importante, seu principal uso foi no meio militar durante as duas grandes guerras mundiais, era utilizada com o objetivo de obscurecer as informações importantes que eram obtidas em campo de batalha.

A criptografia é, provavelmente, tão antiga quanto a própria escrita. Entretanto, só recentemente, se tornou alvo de extenso estudo científico. Uma das grandes motivações para este estudo é a segurança, em um mundo cada vez mais dependente do fluxo de dados [Carvalho 2000].

Hoje em dia, a utilização da criptografia mantém um objetivo em comum, que é garantir a segurança da informação. Em decorrência do avanço da tecnologia, foi ficando cada vez mais fácil explorar novos métodos e realizar experiências para descobrir o limite da segurança dos algoritmos de criptografia.

2. Criptografia RSA

O sistema de criptografia por chave pública RSA é sustentado pela dramática diferença entre a facilidade de encontrar números primos grandes e a dificuldade de fatorar o produto de dois números primos grandes [Cormen et al. 2009].

RSA é um sistema de criptografia assimétrico, o que significa que a informação é criptografada e descriptografada com a utilização de duas chaves diferentes. As duas chaves utilizadas no RSA são demoninadas chaves públicas e chaves privadas, que são respectivamente utilizadas para criptografar e descriptografar dados.

O processo de geração de um par de chaves RSA consiste em testes de primalidade, geração de números aleatórios e aritimética modular, o exemplo abaixo exemplifica o procedimento de criação de um par de chaves utilizando inteiros relativamente pequenos. Na sua utilização prática, o RSA pode utilizar inteiros grandes que podem ultrapassar 2^{2048}.

1. Inicialmente, são selecionados dois números primos aleatórios, $p = 71$ e $q = 83$.
2. É calculado $n = pq$, logo: $n = 71 \times 83 = 5893$.
3. Em seguida, calcula-se o totitente de Euler de n, dado por $\phi(n) = (p-1)(q-1)$: $\phi(5893) = 70 \times 82 = 5740$
4. É escolhido aleatoriamente um e que satisfaça mdc$(e,\ \phi(n)) = 1$
   $e = 593$
5. Em seguida, calcula-se d, que é o inverso modular de e, isso significa que:
   $de \equiv 1 \pmod{\phi(n)}$, d deve ser um número entre 2 e $\phi(n)$
   O resultado da função inverso modular é 997.

As chaves resultantes são:

- Chave pública = $\{e,\ n\} = \{593,\ 5893\}$
- Chave privada = $\{d,\ n\} = \{997,\ 5893\}$

A criptografia de um dado numérico utilizando a chave pública é feita utilizando a exponenciação modular:

$$C = M^e \bmod n$$

Onde C é a mensagem criptografada e M é a mensagem original. É importante garantir que a entrada satisfaça $M < n$. Na descriptografia, é feito o processo inverso, utilizando a chave privada:

$$M = C^d \bmod n$$

3. Implementação

O programa foi desenvolvido utilizando a linguagem C, a implementação de inteiros grandes utilizada foi a da GMPlib – *GNU Multiple Precision Arithmetic Library*, a plataforma usada para o desenvolvimento foi o Linux 3.17 com o GCC 4.9.2.

O software está disponível na página < `https://github.com/cuchi/cal-rsa` > sob a licensa livre *GNU General Public License* – Versão 2.

3.1. O Programa

O programa é dividido em três executáveis diferentes:

- *generate-rsa* – Gera chaves RSA com tamanho arbitrário.
- *rsa-file* – Criptografa e descriptografa arquivos utilizando chaves RSA pré-criadas.
- *simulate-break* – Simula a quebra de uma chave RSA.

A interface do programa é por linha de comando, onde cada executável realiza o *parsing* dos comandos digitados pelo usuário. Abaixo, um exemplo de uso do programa para gerar uma chave e criptografar um arquivo:

```
generate-rsa -b 2048 > my_keypair.txt
rsa-file -i my_passwords.txt -o top_secret -k my_keypair.txt -e
```

Nesse exemplo, o usuário gera um par de chaves de 2048 bits e redireciona a saída, criando o arquivo *my_keypair.txt*. Em seguida, o arquivo *my_passwords.txt* é criptografado gerando o arquivo de saída *top_secret*.

As opções *i*, *b*, *k* e *e* indicam respectivamente o arquivo de entrada, o arquivo de saída, o arquivo que possui o *keypair* e o modo de operação – *e* para criptografar e *d* para descriptografar.

4. Algoritmos

De acordo com [Cormen et al. 2009] a teoria dos números era vista como tópico bonito porém bastante inútil na matemática pura. Hoje em dia nos algoritmos que se baseiam na teoria dos números são usados amplamente, em maior parte devido a invenção de sistemas criptográficos baseados em números primos grandes.

Os algoritmos usados no programa foram feitos a partir de teoremas elaborados por matemáticos teóricos. A necessidade em utilizar esses algoritmos é o objetivo de conseguir um bom aproveitamento dos recursos computacionais.

4.1. Teste de Primalidade

Para muitos algoritmos criptográficos, é necessário selecionar um ou mais números primos muito grandes de maneira aleatória. Com isso estamos frente-a-frente com o problema de determinar se um número grande é primo ou não. Não há maneira simples e eficiente de completar essa tarefa [Stallings 2006].

Na teoria dos números, primo é um número inteiro positivo que possui apenas dois divisores exatos. Um exemplo algoritmo elaborado de maneira ingênua para determinar se um número n é primo, seria um que contasse o número de divisores exatos de n. O que não foi levado em consideração é que o custo computacional desse algoritmo é muito alto quando se trata de inteiros grandes.

Na implementação do programa foi utilizado o algoritmo de *Miller-Rabin*, esse algoritmo é pobabilístico e pode ser usado para testar a primalidade de um número com uma chance de acerto próxima à 100% e com um custo computacional significativamente baixo. Seu funcionamento é demonstrado no Algoritmo 1.

4.2. Inverso Modular e Maior Divisor Comum

Para as aplicações como criptografia e teste de primalidade é necessário manipular números que são significativamente maiores que 32 bits [Dasgupta et al. 2006]. Na aritimética modular utilizam-se algoritmos capazes de fazer operações com números grandes utilizando pouco recurso computacional, os algoritmos de aritimética modular implementados no projeto foram o Algoritmo de Euclides e sua versão extendida, que são usados respectivamente para calcular o maior divisor comum e o inverso modular.

Originalmente, no problema do maior divisor comum há o processo de fatoração, que por ser um problema NP, não é aplicavel quando se trata de números grandes. O algorítmo de Euclides busca obter o maior divisor comum em tempo polinomial com a utilização da regra de Euclides:

- Se x e y são inteiros positivos tal que $x \geq y$, então mdc(x, y) = mdc(x mod y, y).

Há também a extensão do algoritmo de Euclides, demonstrada no Algoritmo 2 abaixo.

4.3. Fatoração de Inteiros Grandes

A dificuldade na fatoração de números inteiros grandes é o fator que garante a segurança do sistema criptográfico RSA.

No programa é implementado um algoritmo de força bruta que dado um módulo n, eoncontra seus fatores, que no caso da chave RSA, são p e q. O Algoritmo 3 abaixo demonstra o seu funcionamento.

Entrada: Um inteiro $n > 3$ para ser testado
Entrada: Um parâmetro k que determinará a precisão do teste
Saída: *composto* se n for composto, caso contrário *provavelmente primo*

início
  escreva $n - 1$ como $2^s d$;
  enquanto $k > 0$ **faça**
    $k \leftarrow k - 1$;
    $a \leftarrow$ número aleatório no intervalo $[2, n - 2]$;
    $x \leftarrow a^d \bmod n$;
    se $x = 1$ *ou* $x = n - 1$ **então**
      Próximo loop;
    fim
    para $r = 1$ *até* $s - 1$ **faça**
      $x \leftarrow x^2 \bmod n$ **se** $x = 1$ **então**
        retorna *composto*
      fim
      se $x = n - 1$ **então**
        Próximo loop;
      fim
    fim
    retorna *composto*
  fim
  retorna *provavelmente primo*
fim

Algoritmo 1: Algoritmo de Miller-Rabin para teste de primalidade.

Entrada: Dois inteiros positivos a e b tal que $a \geq b \geq 0$
Saída: Inteiros x, y e d tal que $d = \text{mdc}(a, b)$ e $ax + by = d$

início
  se $b = 0$ **então**
    retorna *(1, 0, a)*
  fim
  $(x, y, d) =$ *euclides-ext*$(b, a \bmod b)$;
  retorna *(y, $x - \lfloor a/b \rfloor y$, d)*
fim

Algoritmo 2: Algoritmo Extendido de Euclides.

Entrada: Um número inteiro n
Saída: Números inteiros p e q de tal modo que $p \times q = n$

início
  $p \leftarrow \sqrt{n}$;
  se $p \equiv 0$ (*mod* 2) **então**
    $p \leftarrow p + 1$;
  fim
  repita
    $p \leftarrow p - 2$;
  até $n \equiv 0$ (*mod* p);
  $q \leftarrow n/p$;
  retorna p, q
fim

Algoritmo 3: Algoritmo de força bruta para fatoração de número inteiro.

5. Análise de Complexidade

5.1. Quebra de Chave RSA

A quebra da chave privada da criptografia RSA é feita pela fatoração do módulo da chave. O algoritmo de quebra utilizado no programa é semelhante ao Algoritmo 3 demonstrado neste artigo.

Sabendo que a fatoração de inteiros grandes não pertence à classe P, foi escolhido um intervalo limitado de *bits* para a realização de testes.

Os testes foram realizados com chaves de 40 bits até 80 bits, cada chave era gerada aleatoriamente e quebrada logo em seguida, o processo se repetia 10 vezes e o tamanho da chave era incrementado. No total 410 chaves foram quebradas em mais de 130 horas. Os gráficos das Figuras 1 e 2 mostram o crescimento de tempo da função de quebra de chave em relação a n bits.

A quebra de chave criptográfica feita pela utilização do Algoritmo 3, possui a complexidade exponencial de O(2^n), onde n é o número de bits da chave.

5.2. Geração de Chave RSA

O processo de gerar um par de chaves RSA consiste em testes de primalidade de inteiros grandes aleatórios, multiplicação de inteiros grandes e o uso de aritimética modular.

A geração de um par de chaves, contém no melhor caso, a complexidade de $\Omega(n^2)$ e no pior caso, O(∞), é importante destacar que o pior caso é probabilisticamente inviável, pois por probabilidade, há a chance do gerador de números aleatórios fornecer repetidamente apenas números compostos.

6. Conclusão

Pode-se concluir que o sistema de criptografia RSA trata de procedimentos puramente matemáticos, onde há a necessidade de utilizar algoritmos otimizados para manipular inteiros grandes. Por outro lado, o RSA depende da complexidade exponencial da fatoração

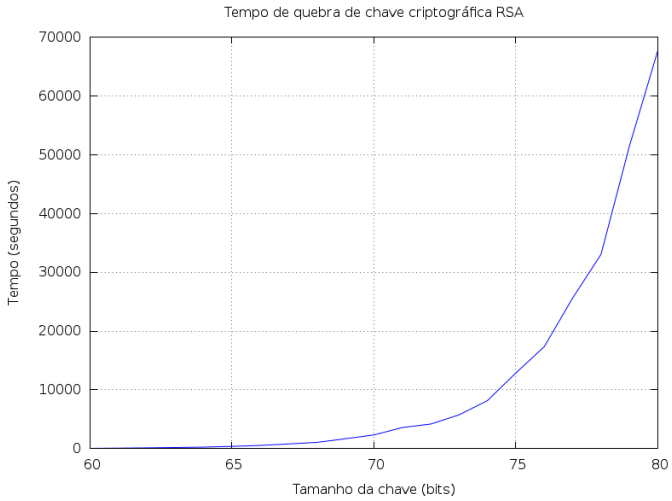

Figura 1. Tempo de quebra de chave em relação ao número de bits.

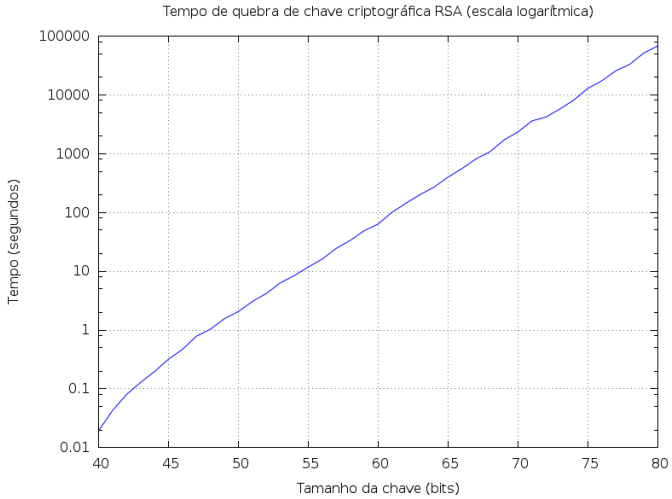

Figura 2. Tempo de quebra de chave em relação ao número de bits (escala logarítimica em y).

de números grandes, o que faz que com que as chaves RSA sejam seguras e amplamente utilizadas nos sistemas atuais.

Até o momento não foi descoberto nenhum algoritmo deterministico que fatore qualquer número inteiro em tempo polinomial, se fosse descoberto tal algoritmo, ou se fosse provado que $P = NP$, a segurança das chaves criptográficas que dependem da fatoração de números estariam comprometidas, pois qualquer chave poderia ser quebrada em tempo polinomial.

Referências

Carvalho, D. B. (2000). *Segurança de Dados com Criptografia: Métodos e Algoritmos*. Book Express LTDA.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Dasgupta, S., Papadimitriou, C. H., and V., V. U. (2006). *Algorithms*. McGraw-Hill Education (India) Pvt Limited.

Stallings, W. (2006). *Cryptography and Network Security: Principles and Practice*. Pearson Education, 4rd edition.