Implementação e Análise de Complexidade do Sistema de Criptografia de Chave Pública RSA

Jackson Machado

Universidade Estadual de Santa Catarina (UDESC)
Joinville – SC – Brasil

***Resumo**: O objetivo deste artigo é realizar uma análise da criptografia assimétrica RSA, descrevendo sobre seu funcionamento e abordando sobre sua implementação e complexidade, tanto para geração quanto para a quebra das chaves, se utilizando da linguagem Python para estruturação dos algoritmos que compõem o sistema de criptografia.*

1. Introdução

Um sistema de criptografia de chave pública, conforme menciona Cormen (2009, p.958-959), pode ser utilizado para codificar mensagens entre dois participantes sem que um intruso que possa escutar essa mensagem consiga decodifica-la, ela pode ser conferida com facilidade por qualquer pessoa, porém não pode ser forjada por ninguém e perde sua validade caso algum bit seja alterado.

Um dos primeiros registros sobre a utilização de criptografia é a Cifra de César, datado da época do Império Romano, tendo seu nome em homenagem a Júlio César, que a utilizava para comunicação com os seus generais. Se tratava de uma criptografia simples de substituição, porém bem útil a época.

No começo do século XX, a criptografia foi fundamental no meio militar, participando de duas grandes guerras, onde os militares omitiam suas mensagens por meio desta.

Por mais que seja antiga, a criptografia só começou a ganhar importância recentemente, tornando-se objeto de estudos, sendo que atualmente o seu grande objetivo é garantir a segurança da informação.

2. Criptografia

Conforme menciona Cormen (2009, p. 959), o sistema de criptografia de chave pública RSA se baseia na diferença drástica entre a facilidade de se encontrar números primos grandes e a dificuldade de fatorá-los.

O sistema RSA é um sistema de criptografia assimétrico, ou seja, utiliza-se de duas chaves para criptografar e descriptografar as mensagens, sendo uma pública e outra privada respectivamente.

Cormen (2009, p. 959) se utiliza do exemplo de dois participantes, Alice e Bob, que querem transmitir uma mensagem criptografada um para o outro. Bob obtém a chave pública de Alice e criptografa o seu texto com ela, Alice então com sua chave privada descriptografa o texto original. Logo, é de extrema importância que a chave privada seja mantida em sigilo, para que a mensagem não possa ser quebrada.

O processo de geração das chaves de RSA consiste em algoritmos com testes de primalidade, geração de números aleatórios e aritmética modular, sendo que, na prática, os números inteiros utilizados podem ultrapassar 2^{2048}.

Abaixo está um exemplo da aplicação do procedimento de criação das chaves, utilizando-se de números menores para facilitar o cálculo e o entendimento:

- Seleciona-se dois números primos aleatórios e diferentes, por exemplo, $p = 17$ e $q = 43$.
- Calcula-se o n como sendo $n = p \,.\, q$, por exemplo, $n = 731$.
- Então, calcula-se o *totiente* de Euler de n, que é dado por $\varphi(n) = (p-1)(q-1)$, sendo $\varphi(731) = 16 \,.\, 42 = 672$.
- Após, aleatoriamente se escolhe um número primo e que satisfaça o Máximo Divisor Comum com $\varphi(n)$, neste exemplo sendo $e=43$.
- Por fim, calcula-se o d que é o inveso modular de e com fórmula $d \equiv 1 \ (mod\ \varphi(n))$, neste exemplo sendo $d=547$

Com isso a chave pública seria {e=43, n=731} e a chave privada seria {d=547, n=731}. Com isso, para criptografar a mensagem basta se utilizar da chave pública através da seguinte fórmula $C = M^e \ mod\ N$, já na descriptografia basta fazer o caminho inverso $M = C^d \ mod\ N$.

A segurança do algoritmo encontra-se justamente na definição de d pois por mais que se saiba n para que se consiga $\varphi(n)$ é necessário saber os primos que lhe originaram, p e q, logo, se é necessário fatorar seu valor para encontrar os valores originais.

3. Algoritmos

Para a criação de um algoritmo RSA, utilizam-se de teorias matemáticas, com a finalidade de aumentar o aproveitamento computacional. Para tanto utiliza-se principalmente as três teorias matemáticas abaixo para a construção do algoritmo.

A implementação completa das funções pode ser consultada no seguinte endereço: < https://github.com/jacksjm/rsa-python>

3.1. Teste de Primalidade

Conforme já abordado na estrutura exemplo da criação das chaves, por vezes é necessário selecionar um número primo para utilização. Contudo esta tarefa se torna complicada quando falamos de números inteiros grandes.

Primeiramente, conceitua-se primo um número interno positivo que possua apenas dois divisores exatos (1 e ele mesmo). Basicamente, se poderia fatorar o número e contar a quantidade de divisões exatas que ocorreria com o mesmo, contudo em números inteiros grandes este processamento seria extremamente lento.

Para esta aplicação foram considerados dois testes diferentes de primalidade, através do método de Fermat e de Miller-Rabin, sendo que em ambos os casos não se obtêm a exatidão com relação a se um número seria ou não primo, porém devido ao ganho computacional e a baixa probabilidade de erro, pode-se dizer que conceitualmente o número seria primo. A implementação está descrita nos arquivos *primalityFermat.py* e *primalityMillerRabin.py*.

Conforme menciona Cormen (2009, p. 971), a complexidade do algoritmo de Miller-Rabin é definida por n sendo um número de β bits, exige $O(s\beta)$ operações aritméticas e $O(s\beta^3)$ operações de bits, pois ele não exige assintoticamente mais trabalho que s exponenciações modulares. Sendo assim, sua complexidade seria $O(k \log_2 n)$, sendo k o número de iterações e n o número de bits.

Já o algoritmo de Fermat possui uma pior complexidade comparado ao de Miller-Rabin, chegando a, aproximadamente, $O(k \log_2 n)$ sendo k o número de iterações e n o número a ser testado.

3.2. Inverso Modular, Maior Divisor Comum e Exponenciação Modular

Com a finalidade de agilizar operações com números inteiros grandes, utilizou-se no projeto os algoritmos de aritmética modular de Euclides e sua versão estendida, de maneira que o primeiro foi utilizado para calcular o Maior Divisor Comum (função MDC com complexidade $O(\log_2 n)$ sendo n o valor do número a ser testado) e o Inverso Modular (função MDCEst com complexidade $O(\log_2 \varphi(n))$).

Também para otimizar as operações de exponenciação seguidas de módulo, foi-se criado a função testMod com a finalidade de realizar a fórmula de criptografia e descriptografia das mensagens de maneira mais eficiente.

A implementação está descrita no arquivo *genericsFunctions.py*.

3.3. Totiente de Euler

Se utilizou da função Totiente de Euler para determinar o valor de $\varphi(n)$, através da busca de dois números aleatórios, p e q, definiu-se o valor de n e para a definição do valor de $\varphi(n)$, utilizou-se de $(p-1) * (q - 1)$.

A implementação está descrita no arquivo *totiente.py*.

3.4. Fatoração de Inteiros Grandes

Como já mencionando anteriormente, a dificuldade na fatoração de números inteiros é o principal fator que garante a segurança da criptografia RSA.

No programa citado são implementados dois algoritmos para quebra da criptografia, sendo um que aplica a força bruta e outra que aplica a heurística que permite quebrar chaves um pouco maiores, que é o algoritmo de Pollard-Rho.

Para o algoritmo de força bruto, que foi implementado no arquivo *brutalForce.py*, realiza-se uma fatoração de 3 até a raiz quadrada de n, pois por heurística um dos números gerados (p ou q) estará entre este espaço, esta busca possui complexidade $O(\sqrt{n})$.

Já para o algoritmo de Pollard-Rho consiste em se utilizar do conceito de Máximo Divisor Comum, Paradoxo do Aniversário e Algoritmo de Busca de Ciclo de Floyd para determinar um dos números gerados. Com isso, utiliza-se a aleatoriedade e heurística para determinar os valores e, como menciona Cormen (2009, p. 976), não há como garantir nem seu tempo de execução nem seu sucesso, porém, espera-se que encontre o valor dentro de $O(\sqrt{p})$, sendo este o pior cenário e *p* sendo o fator modular encontrado. Sua implementação está no arquivo *pollardRho.py*.

4. Análise de Complexidade

4.1. Geração da Chave

Conforme já mencionado, o processo de geração do par de chaves RSA consiste basicamente em testes de primalidade de inteiros grandes e aleatórios, exponenciação de números inteiros grandes e aritmética modular.

A complexidade para geração deste par de chaves está na aleatoriedade da geração dos números primos e em seu teste de primalidade, sendo a complexidade da aleatoriedade $O(n^2)$ que é assintoticamente superior ao teste de primalidade.

4.2. Quebra da Chave

Como mencionado, para a quebra de chave utilizou-se dois algoritmos, um de força bruta, onde este faz iterações de 3 até a raiz quadrada de *n*, e o algoritmo de Pollard-Rho.

Para ambos os casos foram realizadas 20 execuções, de maneira a garantir a média do tempo de execução de cada algoritmo. Ainda, também conforme mencionado, foram utilizados dois cenários para o teste de primalidade, com o algoritmo de Fermat e de Miller-Rabin. O programa utilizado para execução foi o *main.py*.

Os tempos em segundos obtidos foram:

- Algoritmo de Primalidade Miller-Rabin:

Quantidade de Bits	Tempo Força Bruta	Tempo Pollard-Rho
4	0.0008195499999999999	0.0008510999999999999
5	0.00245105	0.0018988999999999998
6	0.0022013000000000002	0.0013487
7	0.0047989999999999994	0.00185085
8	0.00995065	0.0058995
9	0.0276664	0.01506875
10	0.083744	0.0467461
11	0.45683090000000004	0.37473155
12	1.15199055	1.0414135500000001
13	2.4702056	4.311936
14	12.3293228	11.8301114
15	53.604564849999996	50.44135525
16	220.92275145	272.4591924

Tabela 1 - Comparação de Performance Algoritmo de Força Bruta e Pollard-Rho com o Teste de Primalidade Miller-Rabin

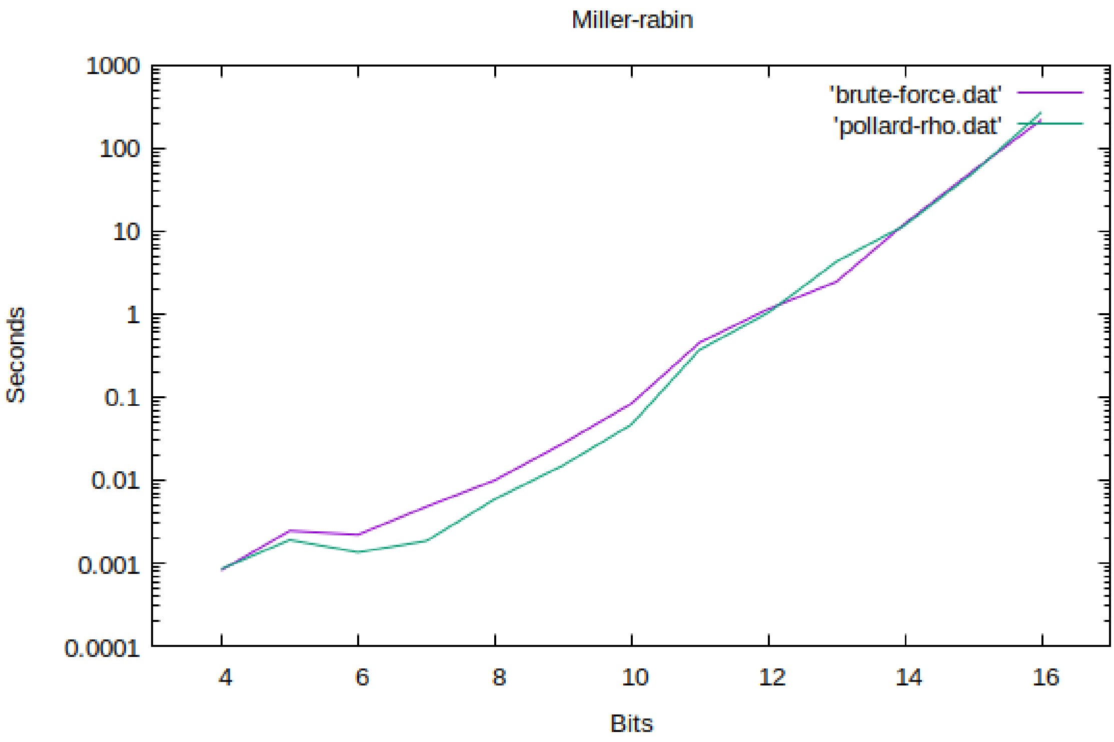

Figura 1 - Gráfico da Comparação de Performance Algoritmo de Força Bruta e Pollard-Rho com o Teste de Primalidade Miller-Rabin

- Algoritmo de Primalidade Fermat:

Quantidade de Bits	Tempo Força Bruta	Tempo Pollard-Rho
4	0.00090235	0.0008510999999999999
5	0.0028778000000000002	0.0011495
6	0.0027924	0.00059965
7	0.00464835	0.0021514999999999998
8	0.0151782	0.00950135
9	0.03143265	0.02595135
10	0.09020275	0.050296450000000006
11	0.2324939	0.20653755000000001
12	1.03579115	0.7800814500000001
13	2.496547	1.55168765
14	9.6308006	12.9153705
15	44.0716983	58.63505535
16	223.0730403	243.4707103

Tabela 2 - Comparação de Performance Algoritmo de Força Bruta e Pollard-Rho com o Teste de Primalidade Fermat

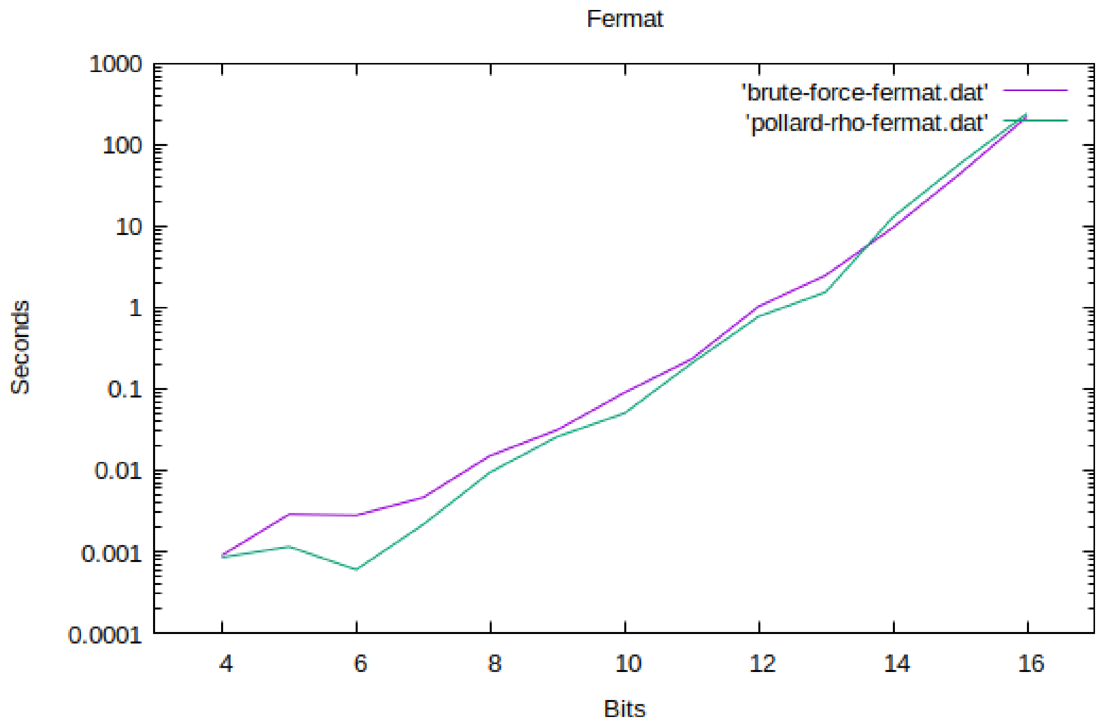

Figura 2 - Gráfico da Comparação de Performance Algoritmo de Força Bruta e Pollard-Rho com o Teste de Primalidade Fermat

Com isso, pode-se perceber que o Algoritmo de Pollard-Rho, na maioria dos casos, se apresenta ligeiramente melhor para a quebra do algoritmo RSA, pois a complexidade de $O(\sqrt{p})$ tende a ser menor que $O(\sqrt{n})$ na maioria dos casos. Ambos os testes demoraram cerca de 4 horas para serem executados, quebrando aproximadamente 640 chaves.

Também, pode-se perceber que o Algoritmo de Miller-Rabin possui melhor desempenho do que o de Fermat, pois os tempos de execução para geração são menores. Isto pode ser percebido haja visto que no teste de quebra é executada a geração da mensagem, logo é possível perceber que na utilização de Miller-Rabin os desempenhos foram melhores.

Desta forma, pode-se dizer que a quebra de chave possui uma complexidade exponencial ao tamanho de bits da entrada, sendo que a utilização de número inteiros grandes garante a segurança de sua informação.

5. Conclusão

É possível verificar que a criptografia RSA se baseia em métodos matemáticos para ser realizada, de maneira a otimizar a utilização de números inteiros grandes. Contudo, como para a fatoração de números grandes ainda não existe solução que possa ser executada em tempo polinomial, sua utilização tende a ser segura.

Ainda, pode-se perceber que a fatoração de números primos se trata de um problema NP e que sua complexidade apresentada uma grande variação de acordo com o método matemático aplicado.

Com isso, entende-se que a fatoração de números primos possui uma complexidade exponencial, no pior cenário, ao número de bits dos valores utilizados nas chaves, necessitando de muito tempo de processamento para conseguir ser quebrada.

Referencias

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