RSA *Rivest-Shamir-Adleman*

Relatório do trabalho de implentação do algoritmo *Rivest-Shamir-Adleman* (RSA) Versão implementada em Java

Félix Dal Pont Júnior Michels, Nicolas Peter Lane

[1]Centro de Ciências Tecnológicas – Universidade do Estado de Santa Catarina (UDESC)
Departamento de Ciência da Computação
Rua Paulo Malschitzki, 200 Zona Industrial Norte, CEP: 89.219-710 Joinville – SC – Brazil

felix_dpm@hotmail.com, nicolas@colmeia.udesc.br

Abstract. *Existem diversos problemas que a humanidade deparou-se ao longo de sua existência, dos quais alguns já foram resolvidos. Dentre as classes de problemas P e NP, o presente trabalho visa apresentar uma implementação do algoritmo de cifra assimétrica RSA cuja dificuldade em ser quebrada está atrelada ao problema da fatoração de primos, um problema NP. Por fim, são apresentadas considerações sobre os diferentes aspectos que impactam a realização da cifra e da tentativa de quebra a partir das experimentações.*

Lista de abreviaturas

RSA *Rivest-Shamir-Adleman*

1. Introdução

A criptografia baseia-se na transmissão de mensagens sigilosas entre duas partes, isto é, cujo conteúdo só possa ser compreendido pelas partes envolvidas. Durante muito tempo a humanidade utilizou-se de um padrão de criptografia denominado criptografia de chave simétrica em suas correspondências sigilosas. Tal tipo de criptografia utiliza a mesma chave sobre a mensagem para cifrá-la e para decifrá-la de volta para a mensagem original. No entanto, essa forma de criptografia requer que ambas as partes conheçam de antemão a chave a ser utilizada a fim de cifrar/decifrar a mensagem [1, 2]. O que torna necessário o envio da chave para o outro lado, momento na qual o mensageiro poderia ser interceptado, tornando conhecido ao interceptador a forma de decifrar a mensagem. Com o advento dos computadores e da Internet, o problema de transmitir-se mensagens sigilosas permaneceu e tal problema levou a criação da criptografia assimétrica [1, 2]. Em algoritmos assimétricos de criptografia existe um par de chaves, a chave pública que é de conhecimento de todos e a chave privada que é conhecida apenas pelo criador do par de chaves. Assim, a comunicação entre as partes ocorre com a mensagem original sendo cifrada com a chave pública desse indivíduo destinatário e o tal indivíduo decifra a mensagem de volta à sua forma original com a sua chave privada. De forma semelhante, a réplica da mensagem original para o seu correspondente ocorre seguindo o mesmo princípio, o emissor utiliza a chave pública desse correspondente e o correspondente decifra a mensagem com a sua própria chave privada. Nesse contexto, este trabalho apresenta uma implementação do algoritmo de criptografia assimétrica RSA cuja dificuldade em quebrar a sua cifra reside no fato da atividade da fatoração de primos ser um problema NP. O presente trabalho é organizado de modo que na Seção 2 é apresentado

o detalhamento da funcionalidade do RSA implementado em conjunto com a complexidade das funções. Enquanto na Seção 3 é apresentado o ambiente de testes que foi utilizado para as experimentações. A implementação do RSA utiliza o hardware descrito na Seção 3, para a experimentação que leva em consideração a cifra, a quebra de cifra com força bruta e a quebra de cifra com força bruta + Pollard Rho na Seção 4. Na Seção 5 são apresentadas as considerações finais em função dos resultados da experimentação e discutido possíveis melhorias que foram observadas.

2. Funcionamento do RSA e suas complexidades de tempo

A implementação realizada pela autoria deste trabalho foi é apresentada segundo as partes que o compõem: o makefile e as classes utilizadas no RSA.

1. makefile
    - *all*: compila o fonte rsa.java
    - *run*: executa a classe rsa.class
    - *plot*: plota os gráficos .dat
    - *clean*: executa a comando de bash clean com argumentos para excluir *.class, *.k, *.dat e *.png
2. public class millerRa.java
    - *public boolean millerRabin(BigInteger n, BigInteger digits, BigInteger repetitions)*: Esse método testa um número n candidato a ser primo e o avalia quanto a chance dele ser primo ou a certeza dele ser um número composto. Assim, o Miller-Rabin é um algoritmo de primacidade probabilístico. Seu funcionamento testa a certeza de o número ser composto, isto é, caso o algoritmo afirme em sua saída o n é composto, então ele é composto e de certeza não é primo. Contudo, caso ele afirme que o valor é primo isso indica que há uma chande de ele ser primo, não a certeza. A redução de incerteza ocorre ao repetir a execução do algoritmo por um certo valor de iterações que pode ser pequeno como 100 que foi utilizado na implementação dessa função. A motivação do pequeno número de repetições é porque a cada iteração o algoritmo proporciona uma redução da chance de ser composto em 1/4. Então com uma reduzida quantidade de iterações é possível determinar que o candidato não é composto, mas sim provavelmente primo. O custo de complexidade de tempo do teste de Miller-Rabin é $O(k * \log^3 n)$, na qual n é o candidato a primo e o k é o número de iterações utilizadas. Contudo, como os valores são BigInt o custo de complexidade de tempo fica em $O(n^3)$.
3. public class eucliExd.java
    - *public BigInteger gcdExtended(BigInteger a, BigInteger b, BigInteger x, BigInteger y)*: O algoritmo é a implementação do euclides extendido que a partir dos resultados obtidos pelo algoritmo de euclides os rearranja de forma alternativa para obter o maior divisor comum dentre os valores a e b como uma combinação linear. Os valores de x e y nessa implementação são necessários para ter-se os valores intermediários da recursão para montar a fórmula da combinação linear. Tal processo é interessante para encontrar valores que sejam coprimos entre si e possui uma complexidade de tempo de $O(\log^2 m)$, onde m é o valor do módulo. Contudo, por tratar-se de valores BigInt a complexidade de tempo fica em $O(m^2)$.

4. public class modExpv2.java
    - *public BigInteger modExp(BigInteger n, BigInteger exp, BigInteger moduleNumber)*: O Java já provê uma implementação de potenciação modular, contudo a autoria do trabalho testando quanto ao desempenho observou que a implementação ofertada por Bruce Schneier [2] foi um pouco melhor. O método é conhecido em inglês como "Right-to-left binary method". A complexidade de tempo da implementação se a função não trabalhasse com BigInt seria $O(loge)$ onde e é o expoente, contudo com BigInt a função passa a custar $O(e^3)$.
5. public class pollardRho.java
    - *public BigInteger rho(BigInteger n)*: O algoritmo implementado visa obter um chute melhor do que o obtido de forma aleatória pelo algoritmo de força bruta regular. O algoritmo beneficia a força bruta porque permite que o espaço de busca seja reduzido de acordo com os valores obtidos de uma função de referência g(x) que no caso desta implementação é $g(x) = x^2 + 1 * modn$, na qual o n é o valor sendo testado. O algoritmo retorno um fator não-trivial de n ou 0 em caso de falha. A complexidade do algoritmo em termos de tempo é $O(n^{1/4})$, contudo quando utilizado junto a BigInteger seu custo passa a ser $O(n^4)$.
6. public class rsa.java
    - *public int menu()*: O algoritmo apresenta 5 opções de atividades dentro do RSA e retorna o inteiro correspondente a atividade selecionada. Assim, sua complexidade de tempo é $O(1)$.
    - *public static BigInteger cipher(String msg, BigInteger e, BigInteger n)*: A função apenas executa a multiplicação modular para cifrar a msg. A função implementada custa em complexidade de tempo $O(e^3)$.
    - *public static String uncipher(BigInteger C, BigInteger d, BigInteger n)*: a função decifra a mensagem cifrada e converte bloco BigInt para String e então converte a mensagem para o seu valor original em char. A complexidade de tempo dessa função é $O(e^3)$.
    - *public static void encryption(String name, int keySize)*: a função gera o par de chaves que é utilizado no RSA para as atividades de cifra e decifração da mensagem. A função internamente converte keySize para BigInt e em suas atividades a de maior custo computacional em tempo culminam no custo de $O(n^6)$.
    - *public static void bruteForce()*: Essa função tenta encontrar um p válido a partir de d que é possível por meio da operação de produto modular quando o p correto for encontrado. Uma vez encontrado p o algoritmo encontra q que o permite decifrar a mensagem. A complexidade da função é $O(2^n)$.
    - *public static void bruteForce20min()*: Funcionamento semelhante a função *bruteForce()*, porém o objetivo é gerar a maior chave possível em 20 minutos de execução.
    - *public static void bruteForceProb()*: Em princípio é semelhante ao *bruteForce()*, mas este algoritmo diferencia-se por utilizar como chute inicial o Pollard Rho. Inicialmente foram feitos testes utilizando em todas as iterações valores do Pollard Rho, contudo os autores observaram que o

resultado ao utilizar somente na estimativa inicial do algoritmo o tornavam verdadeiramente mais rápido para quebrar a cifra. A complexidade da função é $O(n^4)$.

3. Ambiente de testes

O seguinte ambiente de testes foi utilizado na Seção 4 para executar a implementação do RSA apresentado na Seção 2. Esse ambiente compreende um computador que usa como sistema operacional uma distribuição linux baseada no Debian com as seguintes especificações:

- CPU: Core i5-3333 3.00 GHz
- RAM: 16GB DDR3 798MHz
- SDD: 256 Hitachi Sata
- Compilador: Oracle Java JDK 11

A autoria do trabalho optou por usar-se da linguagem de programação Java por possuir um suporte estável a utilização de inteiros grandes. Além disso, a autoria concordou que o desempenho não seria muito afetado ao utilizar Java, permitindo a implementação do algoritmo com um bom desempenho.

4. Experimentos

Nessa seção são apresentados 3 casos de teste, respectivos a atividade de cifra da mensagem com o RSA, da quebra de cifra via força bruta e da quebra da cifra com força bruta + Pollard Rho. Os casos de teste são então comparados de acordo com o tamanho da chave utilizada. Em cada caso de testes o algoritmo executou 20 vezes e os valores de tempo obtidos são a média dessas execuções para os valores de 8, 16 e 32- bits junto ao RSA. O objetivo desses casos de teste é apresentar o comportamento do RSA para as diferentes atividades de modo que seja explicitado o porque o RSA é um importante algoritmo para a segurança dos dados de transações que ocorrem pela Internet. O caso de teste mensurando as cifras de mensagens é apresentada na Subseção 4.1, enquanto o casos de testes com força bruta está na Subseção 4.2 e o de força bruta com Pollard Rho está na Subseção 4.3. Por fim, a comparação de tempo das atividades de acordo com o respectivo tamanho da chave utilizada é apresentado na Subseção 4.4.

4.1. Caso de teste 1: atividade de cifra da mensagem

Nessa seção são apresentados os resultados para as 20 iterações com as quais foram mensurados os tempo de execução para a cifra de tamanhos de 8, 16 e 32 bits. Ao final todas as execuções, são apresentadas as considerações a respeito deste caso de teste:

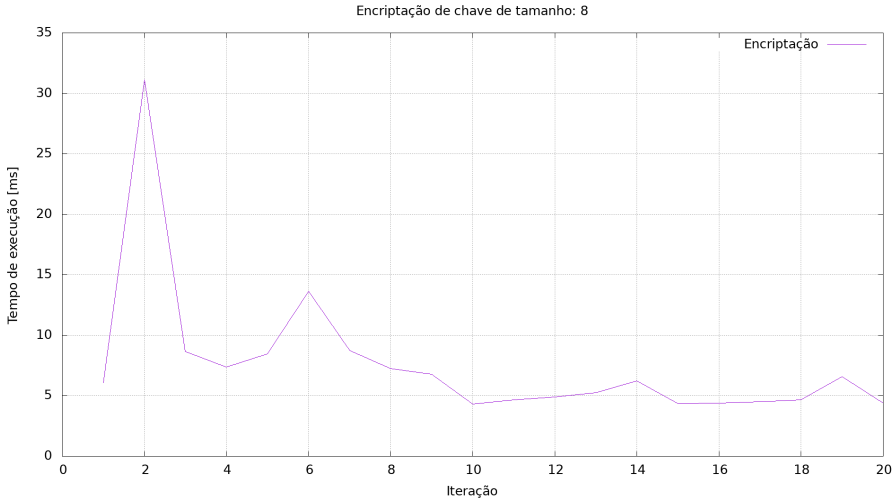

Figura 1. *cifra*: 20 iterações para cifrar com 8 bits

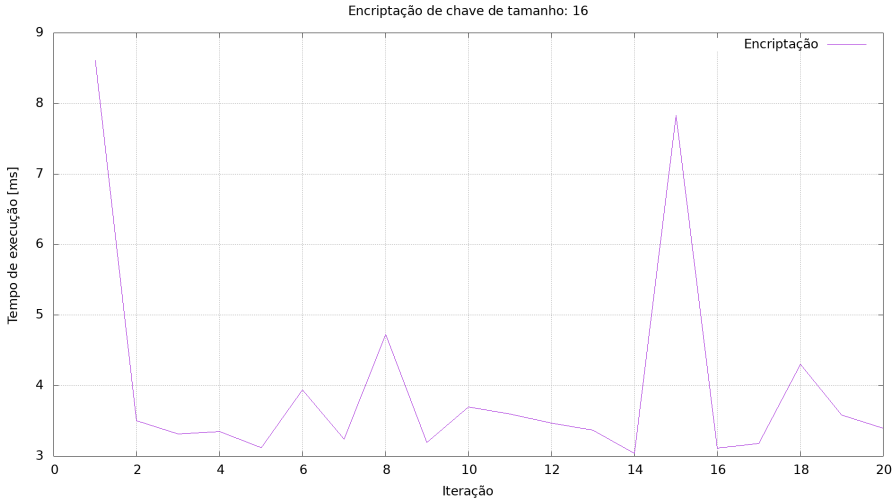

Figura 2. *cifra*: 20 iterações para cifrar com 16 bits

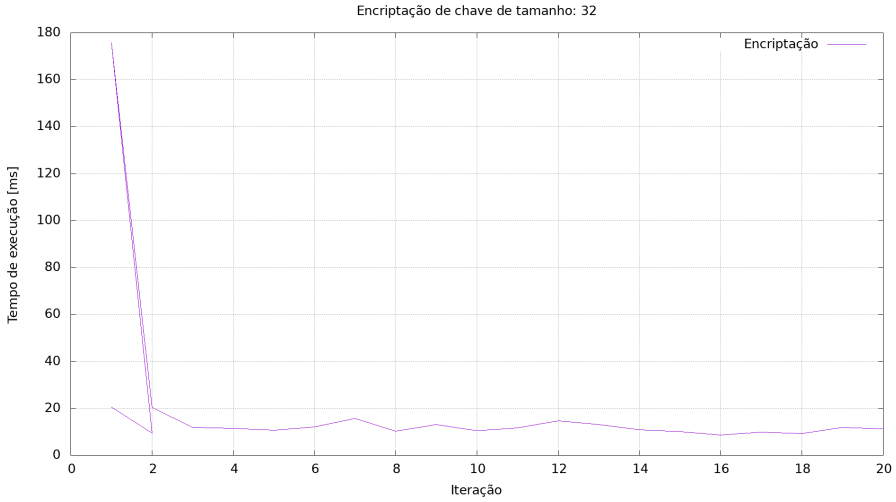

Figura 3. *cifra*: 20 iterações para cifrar com 32 bits

Nas Figuras 1, 2 e 3 é ilustrado que o algoritmo executou em curto espaço de tempo as encriptações sobre as mensagens em questão. Pode-se observar um gradual aumento no tempo a medida que a cifra aumenta de forma geral. O que faz sentido uma vez que a cifra aumente, eleve o consumo de processamento e de tempo da aplicação. A comparação das médias dos tempos de encriptação vai ser apresentada na Subseção 4.4. Além disso, observa-se o aumento no tempo de execução está em conformidade com a complexidade de tempo $O(n^6)$.

4.2. Caso de testes 2: atividade de quebra da cifra por meio de força bruta

A quebra da cifra utilizando força bruta foi executada 20 vezes para cada tamanho de chave: 8, 16 e 32 bits. Em todos os casos, foram utilizadas diferentes chaves criadas pela função de encriptação. São expostos os tempos de execução para cada iteração da função, bem como a comparação entre as médias dos tempos em relação ao seu tamanho de chave na Subseção 4.4. Ao final de todas as execuções, são apresentadas as considerações a respeito deste caso de teste:

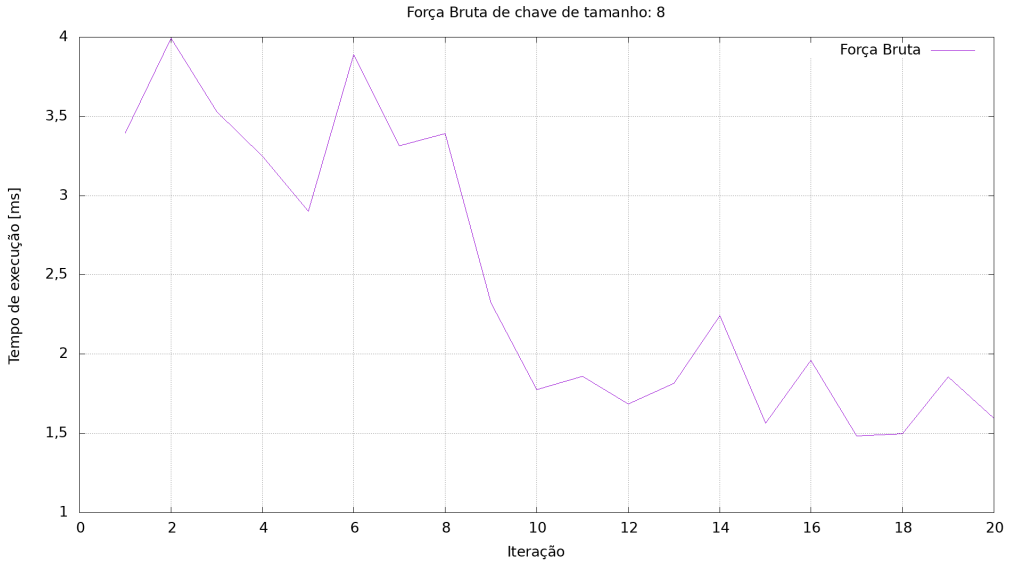

Figura 4. Tempo de execução de cada iteração da função força bruta para a chave de tamanho 8 bits.

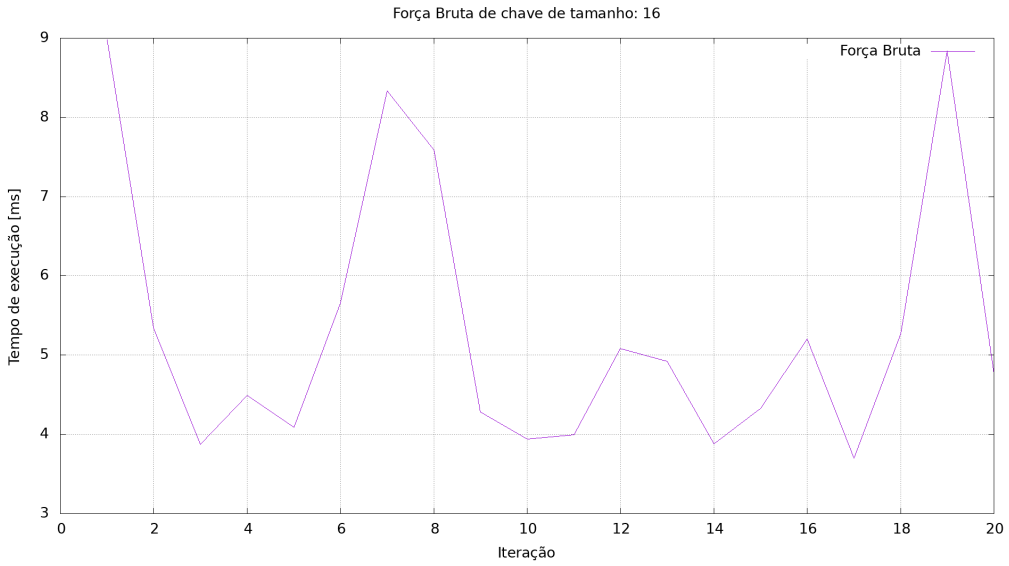

Figura 5. Tempo de execução de cada iteração da função força bruta para a chave de tamanho 16 bits.

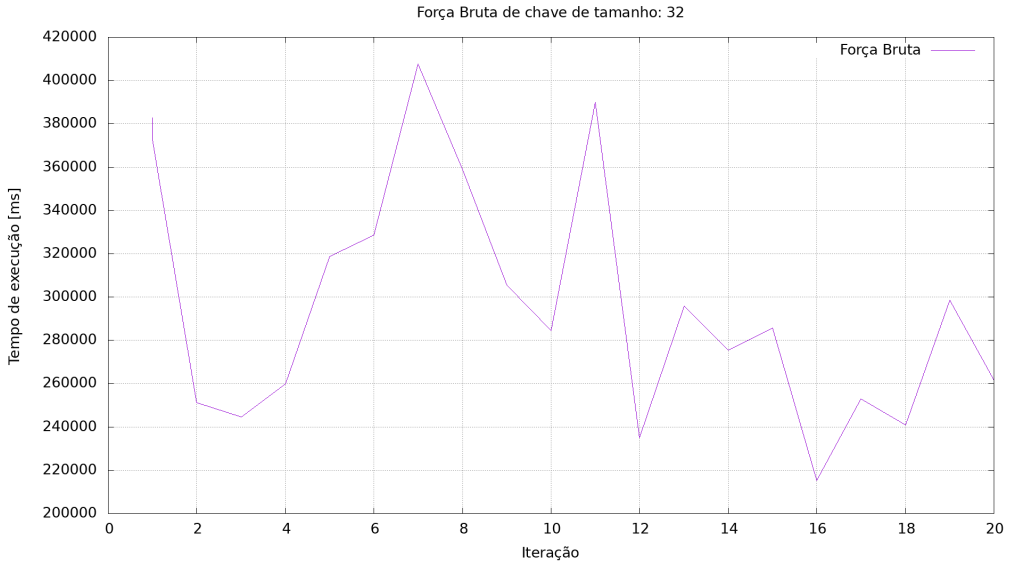

Figura 6. Tempo de execução de cada iteração da função força bruta para a chave de tamanho 32 bits.

Analisando as Figuras 4, 5 e 6 é perceptível o aumento em tempo execução conforme o crescimento do tamanho da chave. A diferença na ordem de grandeza é compatível com a complexidade de tempo da função, $O(2^n)$. A comparação completa entre as médias dos tempos é apresentada na seção 4.4.

4.3. Caso de testes 3: atividade de quebra da cifra por meio de força bruta e de Pollard Rho

A função de fatoração de Pollard Rho foi adicionada ao chute inicial da quebra de cifra utilizando força bruta. Assim para estudar melhor seu impacto, a função foi executada

20 vezes para cada tamanho de chave: 8, 16 e 32 bits. Em todos os casos dispostos nas Figuras 7, 8 e 9, foram utilizadas diferentes chaves criadas pela função de encriptação.

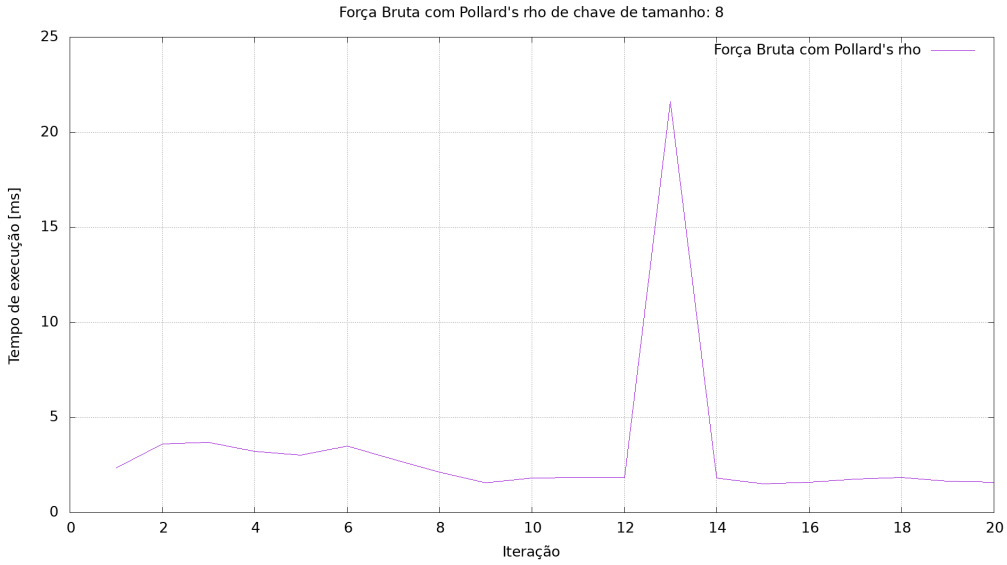

Figura 7. Tempo de execução de cada iteração da função força bruta com Pollard's Rho para a chave de tamanho 8 bits.

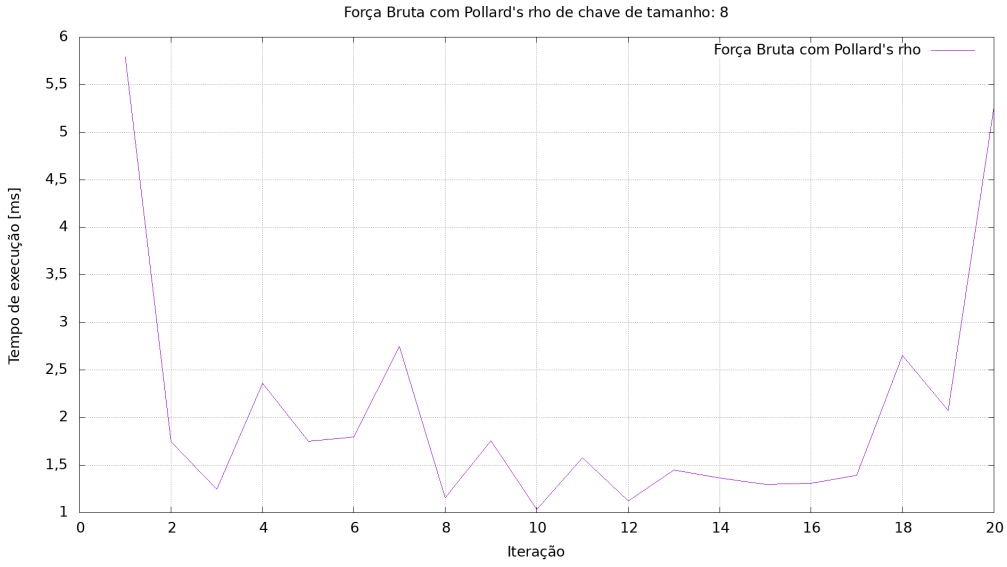

Figura 8. Tempo de execução de cada iteração da função força bruta com Pollard's Rho para a chave de tamanho 16 bits.

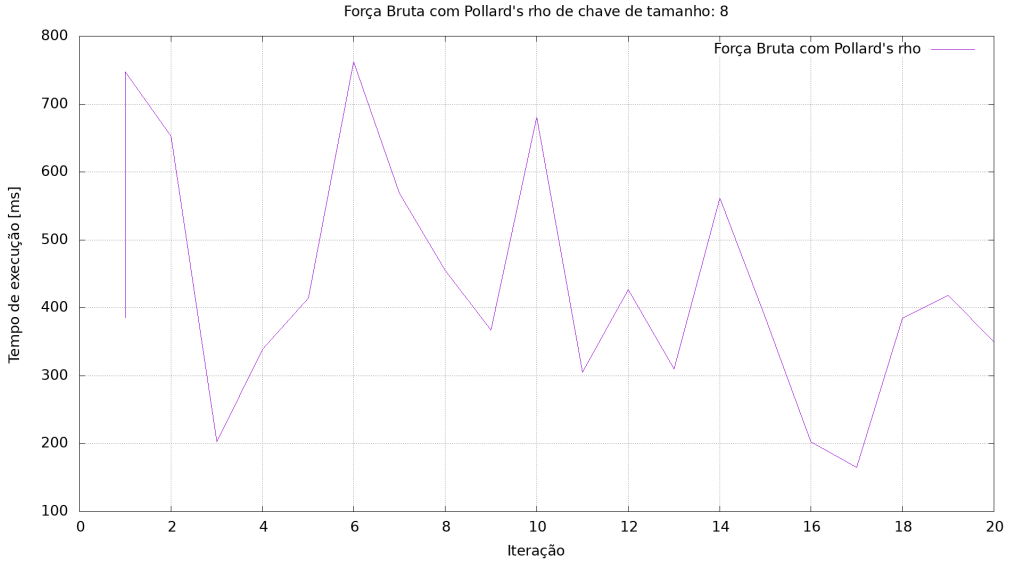

Figura 9. Tempo de execução de cada iteração da função força bruta com Pollard Rho para a chave de tamanho 32 bits.

Utilizando-se da função de Pollard Rho para o chute inicial da fatoração de primos, percebe-se que os tempos de execução foram aprimorados. Ao examinar as Figuras apresentadas, 7, 8 e 9, vemos que o aumento no tempo de execução conforme o crescimento do tamanho da chave está de acordo com a complexidade de tempo $O(n^4)$.

4.4. Comparação dos resultados apresentados nos casos anteriores

Nessa seção são apresentadas comparações realizadas entre os dados obtidos nos casos de testes realizados. Isso foi feito, para proporcionar um entendimento maior acerca dos dados que foram obtidos a fim de verificar a conformidade com o que os autores sabiam a respeito do RSA. Os dados quanto a comparação do tempo de cifra são apresentados na Figura 10.

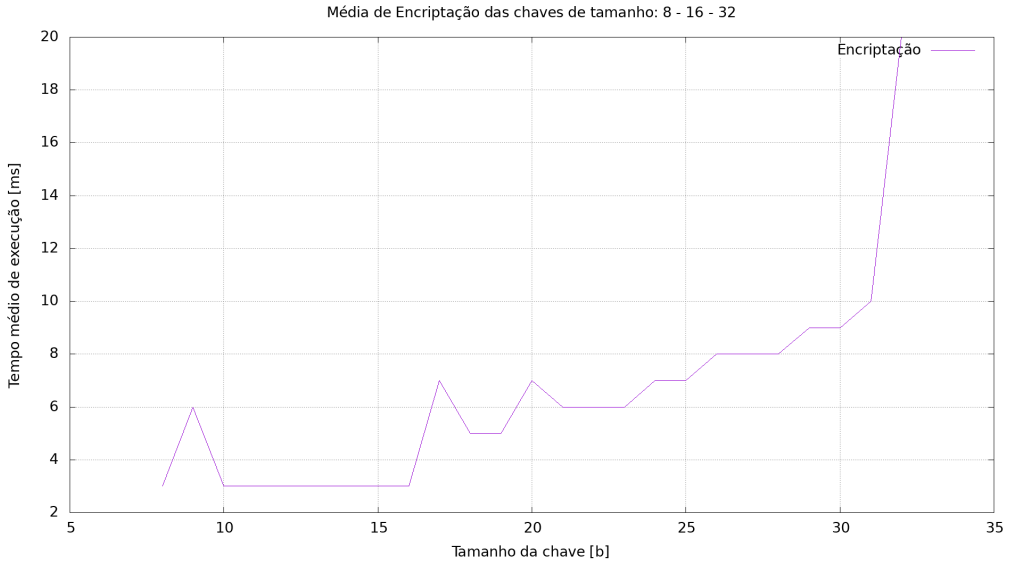

Figura 10. Média das execuções do caso de cifra/encriptação sobre mensagens

Na Figura 10 é possível observar que a cifra de mensagens utilizando o algoritmo do RSA possui uma relação de tempo muito pequena quando comparado com o custo proporcional de tempo da tentativa de quebrar a cifra por força bruta. Contudo, esse tempo a medida que a chave tende a aumentar significativamente. Observa-se também que para a atividade de encriptação de dados, em média o custo dela feita para 32 dígitos foi maior do que para os demais casos. Enquanto no que tange todos os casos, os tempos estão de acordo com o que era esperado em termos de crescimento dado a complexidade atribuída, $O(n^6)$. A Figura 11 representa os tempo médio da execução da função de força bruta para as três chaves.

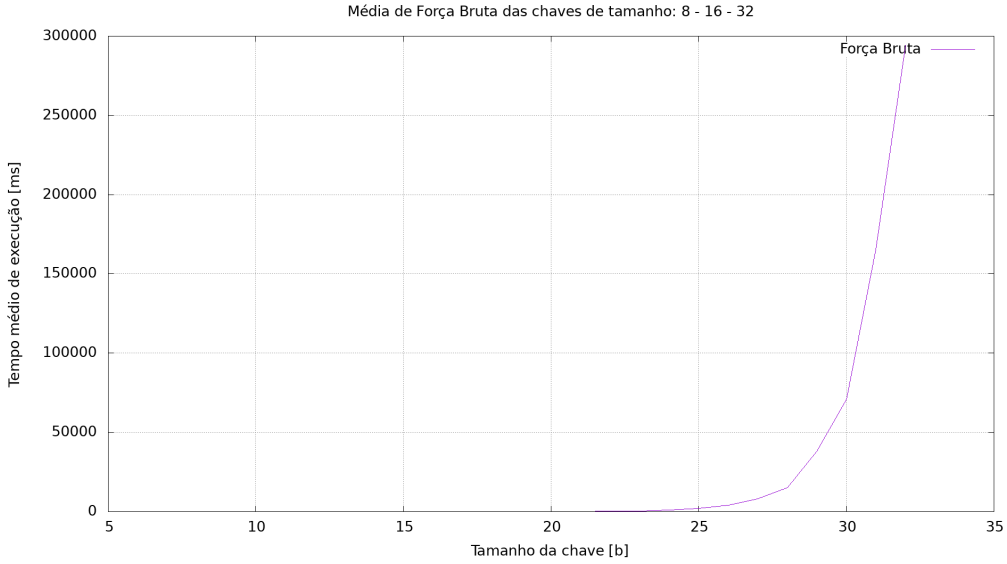

Figura 11. Média das execuções do caso de quebra via força bruta

Observamos que, como era previsto, o tempo médio de execução da chave de 32 bits é muito acima das demais. Os tempos médios de execução foram de 1,478 ms, 5,326 ms e 294147,333 ms, para as chaves de 8, 16 e 32 bits, respectivamente. Como já mencionado, a curva apresentada segue a complexidade de tempo estabelecida para o força bruta de $O(2^n)$. Com esses resultados, pode-se perceber que mesmo para decodificar chaves pequenas de 32 bits, o tempo de execução é muito elevado, aproximadamente 48000 vezes o tempo de execução da cifra da chave. Na Figura 12 é apresentado a comparação das média de tempo da função de força bruta utilizando Pollard Rho.

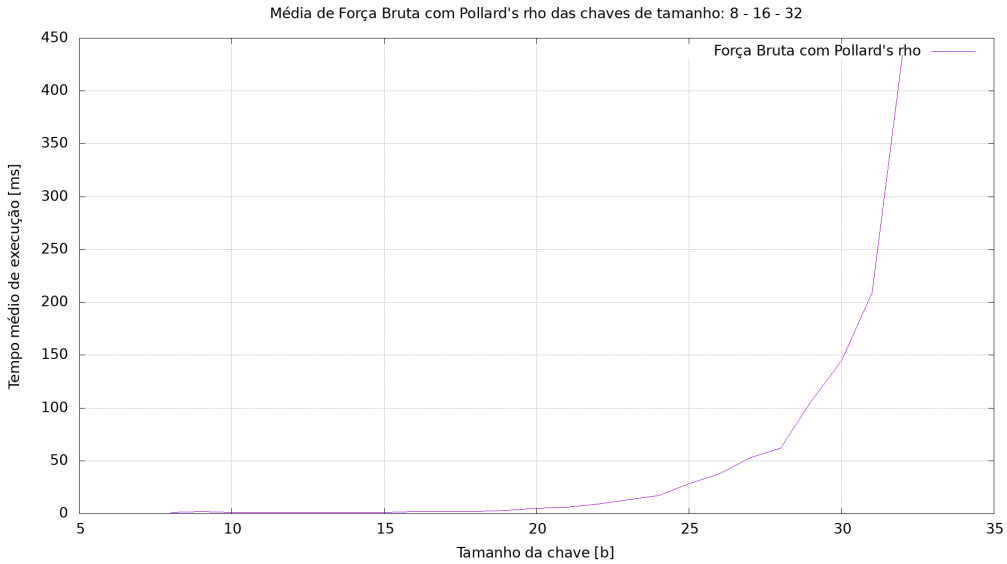

Figura 12. Média das execuções do caso de quebra via força bruta com Pollard's Rho

Os tempos médios de execução são de 1,704 ms, 2,045 ms e 434,418 ms, respectivamente para as chaves de 8, 16 e 32 bits. Assim, podemos rapidamente perceber que ao utilizar Pollard Rho, temos uma melhora substancial de até três casas decimais na quebra de chaves, aproximadamente 800 vezes menor. Novamente, o comportamento apresentado pela função na Figura 12 coincide com a complexidade da função de força bruta com Pollard Rho, como era previsto.

5. Considerações finais

Através desse estudo, pode-se experienciar as dificuldades na quebra de grandes chaves. Testes foram feitos com chaves de tamanhos maiores, como 128 bits, o qual tomou 6 horas de tentativas para uma chave gerada em 0,6 ms. Além disso, observa-se que a busca por implementações eficientes permitiu uma otimização da implementação em Java quando comparada com a implementação que tínhamos inicialmente que utilizada o crivo de Erastóteles para decidir quanto a primacidade, sendo uma abordagem ingênua a medida que o primo escalava em grandeza. De forma equivalente, a sua troca por Miller-Rabin melhorou muito o desempenho do algoritmo. Contudo, inicialmente a quantidade de iterações dele estavam muito pequenas (*e.g.* 20-40 iterações) de modo que ele de vez enquanto cometia erros ao afirmar que um número era provavelmente primo. Detalhe que nos levou a aumentar a iteração de Miller-Rabin para 100 a fim de possuir maior assertividade quanto ao resultado, o que eliminou o problema de Miller-Rabin errar quando dizia um n ser provavelmente primo. Outro ponto de interesse é que a implementação de Pollard Rho permitiu um decréscimo muito grande na complexidade para quebrar a cifra, o que contrariava as afirmações de que ele pouco ajudaria. Por fim, os autores concordam que os testes proporcionaram esclarecimento a cerca do algoritmo RSA, bem como uma nova perspectiva no que convem ao dilema P=NP.

Referências

[1] Stallings, William - “Cryptography and Network Security Principles and Practice”. 7th Edition, Pearson. 2017.

[2] Bruce Schneier - “Applied Cryptography: Protocols, Algorithms, and Source Code in C”. 20th Anniversary Edition, Willey. 2015.