Instituto Tecnológico de Aeronáutica
Curso de Engenharia da Computação
Disciplina de CTC-21

Sistema criptográfico em RSA

Daniel Prince Carneiro
Dylan N. Sugimoto
Gabriel Adriano de Melo
Laurival Siqueira Calçada Neto
Thiago Filipe de Medeiros

Docente: Professor Carlos Ribeiro

São José dos Campos
11/07/2017

Designação de responsabilidades

Daniel: Pesquisa e compreensão do método RSA. Responsável pela composição parcial do relatório. Além disso, revisão final do relatório e ajuste de incorreções semânticas e sintáticas.

Dylan: Pesquisa e compreensão do método RSA. Pesquisa e compreensão da correlação entre o método RSA e a matemática discreta. Redação da motivação e da metodologia do relatório que propôs o tema desse projeto. Redação do esqueleto do relatório. Redação da capa, motivação, metodologia e recursos utilizados.

Gabriel: Pesquisa e compreensão do método RSA. Descrição do funcionamento do código. Avaliação e otimização do algoritmo. Análises de tempo e de implementação. Pesquisa e avaliação da segurança da implementação.

Laurival: Pesquisa e compreensão do método RSA. Responsável pela avaliação e analise do método utilizado. Além disso, composição do relatório sobre os dados obtidos para comprovar a eficiência do método RSA

Thiago: Implementação do algoritmo RSA na linguagem Python. Comentários do código explicando e exemplificando o funcionamento do mesmo. Testes e análise do código implementado em relação ao espaço utilizado e tempo consumido durante execução.

_____
Daniel Prince Carneiro

_____
Dylan Nakandakari Sugimoto

_____
Laurival Siqueira Calçada Neto

_____
Gabriel Adriano de Melo

_____
Thiago Filipe de Medeiros

Motivação

Nesta Era da Informação, a humanidade está cada vez mais dependente da internet, pois a facilidade que essa tecnologia trouxe para o compartilhamento de informação mudou a forma como os indivíduos se relacionam consigo mesmo e com os outros ao seu redor. Contudo, um problema desse compartilhamento de informação é a segurança dos dados compartilhados, no sentido de evitar que agentes invasores sequestrem esses dados ou informações compartilhadas. Afinal, como praticamente todos os setores da economia e da política utilizam computadores para armazenar e compartilhar informações, torna-se evidente que haja informações sensíveis ou confidenciais armazenas na forma de bits; logo, esses computadores podem sofrer ataques cibernéticos, como ocorreu recentemente, no mês de Maio e Junho de 2017, em que "*hackers*" sequestraram informações de computadores de empresas e pediram um valor de resgate dessas informações para essas empresas.

Assim, nesse contexto da segurança da informação, destaca-se a criptografia RSA por possibilitar a comunicação segura de informações e, pelo menos até agora, ter resistido a todas as tentativas de quebra. Além disso, é o principal método de criptografia utilizado na segurança de dados compartilhados via internet, e por tudo isso foi escolhido como tema para este trabalho aplicar a criptografia RSA em textos em português, bem como pelo fato desse método de criptografia utilizar conhecimento embasado na Teoria dos Números, que está inserida no contexto da matemática discreta, ou seja, este tema está correlaciona com o conteúdo ministrado na disciplina CTC-21. De forma mais específica, foi utilizado conceitos da aritmética módulo n para aplicação do método RSA[1][3], porém é sabido que a criptografia faz uso de teorias mais avançadas da matemática discreta, como por exemplo, um problema computacional da criptografia é encontrar vetores curtos em reticulados modulares, em que nesse caso faz uso de uma teoria de reticulados.[4] [5]

Metodologia

Este projeto foi executado em três etapas:

1. Busca por informações no sentido de compreender o funcionamento do algoritmo do método de criptografia RSA e a sua correlação com a disciplina de CTC-21.
2. Implementação do método em código utilizando as linguagens recomendadas, em especial, a linguagem python e a documentação do código.
3. Análise do código e redação do relatório.

Na primeira etapa, percebeu-se que há uma grande quantidade de materiais sobre a metodologia RSA, e de diferentes formatos, ou seja, há diferentes fontes sobre essa metodologia disponíveis na internet que variam desde web enciclopédias e outros sites da área de criptografia e matemática discreta até monografias e artigos científicos, e videoaulas sobre a metodologia RSA. Diante dessa grande variedade de fontes, deu-se prioridade para as fontes com referências confiáveis, ou para as documentações advindas de instituições de educação e de pesquisa de conceito e de alto prestígio. Assim, consultando essas fontes compreendeu-se o algoritmo de geração, de encriptação e de decriptação, e que tal método fundamenta-se na matemática módulo n. [1][3] E com uma pesquisa mais aprofundada, no sentido, de investigar mais a fundo a correlação entre a matemática discreta e a criptografia, descobriu-se a aplicação da teoria de reticulados em criptografia, em específico, na resolução do problema do vetor mais curto.[4][5]

De forma resumida, o método RSA é composto por geração das chaves, encriptação e decriptação da mensagem. Na geração das chaves, primeiramente se escolhe dois números primos e realiza-se o produto deles, obtendo-se o primeiro número da chave pública e da chave privada. O segundo número da chave pública é escolhido de forma que seja primo com a função totiente do primeiro número, e seja menor que a função totiente do primeiro número. O segundo número da chave privada é tal que o produto entre o segundo número das duas chaves deixa resto unitário na divisão pela função totiente do primeiro número da chave pública. Na encriptação, cada caractere é transformado para um número de dois dígitos, e o texto é quebrado em blocos, chamados de mensagem. O número criptografado na mensagem é tal que deixa o mesmo resto que a mensagem elevada à potência do segundo número da chave pública na divisão pelo primeiro número da chave pública. Na decriptação, recupera-se a mensagem ao calcular o número que deixa o mesmo que a mensagem criptografa elevada à potência do segundo número da chave privada na divisão pelo primeiro número da chave privada. Essa lógica matemática foi implementada computacionalmente na etapa seguinte.

Na segunda etapa, implementou-se o método RSA em linguagem Python começando pela geração das chaves, depois, a encriptação de texto, e finalmente, a decriptação, sendo que a documentação do código foi feita à medida que parte era completada. Após o término de cada parte do código (geração, encriptação e decriptação), testes foram realizados para assegurar a funcionalidade do código. E ao fim das três partes estarem completas, iniciou-se os testes finais para assegurar o funcionamento global do código sendo seguidos pelos testes de análise de desempenho. Os resultados desses teste são descritos na secção "Resultados e Análise" deste relatório.

Na terceira etapa, foi feito uma análise do código, e melhorias foram sugestionadas e implementadas. E o projeto foi documentado na forma do presente relatório.

Recursos e Avaliação

Neste projeto, foi utilizado as seguintes ferramentas de pesquisa e codificação:

- Pesquisa na internet;
- Consulta a materiais acadêmicos relacionados ao assunto (artigos científicos, monografia, livros, etc);
- Codificação em Python;
- Código redigido utilizando o programa "Sublime Text".
- Redação do relatório no programa "LibreOffice Writer";
- Geração de primos no programa "OpenSSL"

A avaliação da implementação do sistema criptográfico baseado no método RSA para textos em português deu-se na análise do tempo de processamento, da capacidade de criptografar mensagens com caracteres especiais e da segurança contra ataques de força bruta.

A análise de tempo foi feita utilizando-se recursos do próprio Python, contabilizando-se os tempos para cada etapa de execução do programa.

Implementação

O código foi dividido em três arquivos de forma a organizá-lo melhor. O arquivo "RSA.py" implementa efetivamente o algoritmo RSA, desde a conversão e criptografia até a descriptografia de uma mensagem de texto presente no arquivo "mensagem.txt". No arquivo "classes.py" utilizou-se orientação a objetos para encapsular o código que definem o mensageiro e o destinatário da

mensagem. No arquivo “suporte.py” definiram-se funções matemáticas básicas e computacionais auxiliares, como o máximo divisor comum entre dois números, o inverso e a exponenciação modulares.

Deve-se utilizar dois primos maiores com mais de 100 dígitos decimais para que o código possa funcionar, apesar de que também foi codificada uma função que gera primos aleatórios, mas que não é utilizada em toda a execução por ser lenta. Além da utilização de primos gerados por implementação própria, obtendo-se uma chave de 768 bits, também se gerou primos com o programa OpenSSL, de forma a se obter uma chave de 2048 bits.

No arquivo “RSA.py”, as chaves públicas e privadas são geradas a partir dos números primos, isto é, primeiro faz-se a multiplicação de ambos os primos para obter a chave pública e o seu expoente é obtido como o co-primo do produto dos dois primos menos um. O expoente da chave privada é obtido como o inverso modular do expoente da chave pública. Em seguida, a mensagem é lida de um arquivo e criptografada utilizando-se a chave pública, sendo a mensagem criptografada impressa na tela. Por fim, a mensagem é descriptografada utilizando-se a chave privada, sendo então impressa na tela.

É importante notar que como a criptografia ocorre em texto, é necessário que o mesmo seja convertido para uma forma numérica. Para melhorar a segurança da implementação, cada grupo de 95 caracteres é convertido para um único número, e este número pode ser aumentado caso se escolham números primos ainda maiores. É importante, contudo, que a codificação da mensagem original esteja no formato UTF-8, pois cada caractere poderá ser codificado com até 1000 códigos diferentes, na implementação utilizada em python, o que cobre todos os acentos e caracteres especiais utilizados em português. Esse passo de converter o texto em número é realizado pela função “conversor” presente na classe “mensageiro”.

Após ter a mensagem de texto expressa como um vetor de números, cada número é então criptografado utilizando-se a chave pública, isto é, é realizada uma exponenciação modular desse número na base da chave pública.

Assim, a mensagem criptografada, que corresponde a esse vetor de números criptografados pode ser transmitida com segurança para o destinatário, que ao recebê-la, utilizará a sua chave privada para descriptografar cada número desse vetor de números. Assim, com um vetor de números descriptografados, basta converter cada número em uma sequência de 95 caracteres, que poderá ser lido no formato UTF-8.

Resultados e Análise

A implementação do algoritmo teve um tempo linear para a quantidade de caracteres criptografados, o que pode ser observado nas Figuras de 1 a 5. Isso se deve ao fato de que a criptografia empregada se deu por blocos de caracteres, e não pela a mensagem como um todo. Dessa forma, como cada bloco tem um tempo de processamento aproximadamente constante, o tempo de execução do algoritmo depende apenas da quantidade de blocos de caracteres a serem criptografados, e como cada bloco contém um número constante de caracteres, o tempo de execução do algoritmo depende linearmente da quantidade de caracteres a serem criptografada.

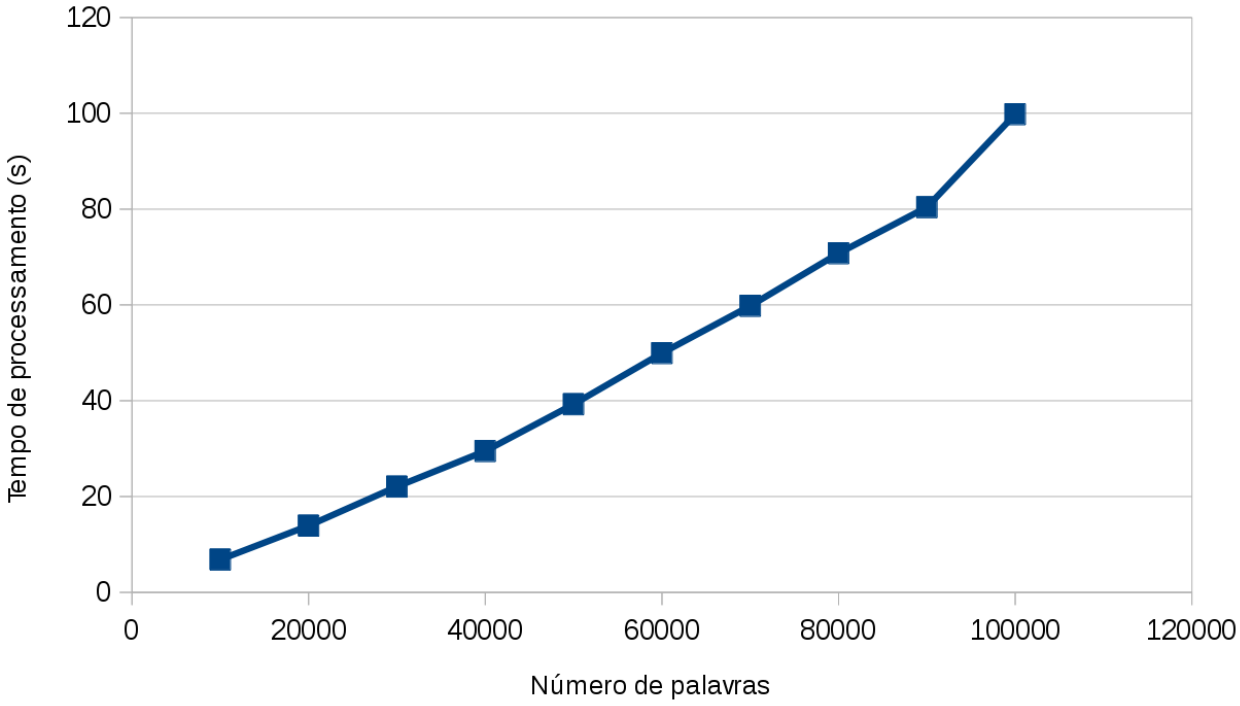

Figura 1 – Tempo de processamento em função do número de palavras.

É importante notar que cada bloco de caractere deve ter um tamanho considerável, caso contrário, um atacante que conhecesse a chave pública poderia simplesmente gerar todas as combinações de blocos possíveis. Por exemplo, considerando o caso no qual cada caractere fosse individualmente criptografado, um atacante poderia simplesmente gerar todos os 255 caracteres possíveis e assim teria a tabela numérica de transformação do caractere para o número criptografado e assim poderia descobrir a mensagem criptografada.

A segurança da implementação dá-se pelo tamanho do bloco de caracteres e pelo tamanho dos números primos empregados. O tamanho do bloco de caracteres é importante para impedir que um atacante, por força bruta, obtenha todas as possíveis combinações de blocos e seus respectivos números criptográficos e, uma vez que o processo de criptografia é determinístico, obter a mensagem original. Os grandeza da chave empregada, isto é, dos números primos utilizados, é

importante não só por determinar o tamanho máximo do bloco, como também por ser necessário fatorá-lo para obter a chave privada e poder ler a mensagem original.

A fim de avaliar a segurança do método RSA pela fatoração da chave pública, efetuou-se uma pesquisa bibliográfica. A maior chave encontrada a qual foi possível ser quebrada tinha um tamanho de 768 bits, consideravelmente próximo à chave de 781 bits utilizada inicialmente no projeto. Em um processador de um único núcleo, AMD Opteron, operando a 2.2 GHz e utilizando 2 GB de RAM, essa fatoração levaria cerca de 1500 anos [6].

Outro fator importante de segurança é o fato de que uma pequena variação na mensagem, seja de apenas um caractere, afeta significativamente o código criptografado do seu respectivo bloco de criptografia. Dessa forma um atacante não consegue avaliar, em um ataque de força bruta, se a sua tentativa está próxima da mensagem interceptada.

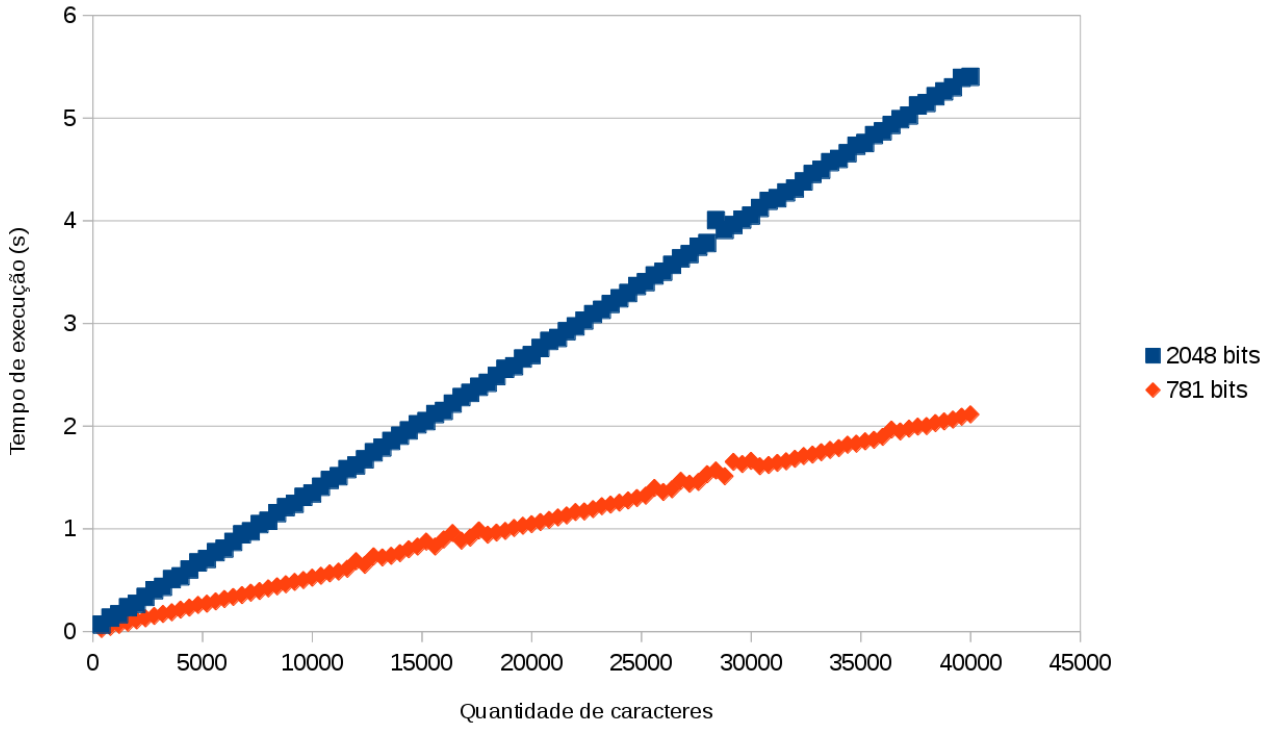

Figura 2 – Comparação entre os tempos de execução de chaves de 2048 e 781 bits.

Para gerar uma chave e 2048 bits aleatória, utilizou-se o programa OpenSSL para gerar os números primos aleatórios. Dessa forma, efetuou-se os testes com o algoritmo utilizando-se um chave de 2048 e blocos de 250 caracteres. Como cada caractere foi representado por 8 bits na codificação utilizada, uma codificação de 2048 bits permite blocos de até 256 caracteres para que a mensagem fosse descriptografada unicamente.

Dessa forma, a implementação do algoritmo pode ser avaliada também utilizando-se primos ainda maiores do que os utilizados inicialmente. Tais primos, apesar de aumentarem a segurança contra fatoração e força bruta, aumentam o tempo de processamento, como observado na Figura 2.

Com relação aos tempos de execução, observou-se que entre os tempos de conversão e de desconversão de texto para número, o último é consideravelmente mais lento que o primeiro, o que pode ser explicado pela realização da operação módulo, presente apenas no último. Dessa forma, a operação de exponenciação modular, que realiza várias vezes a operação de módulo, é ainda mais lenta, o que explica os maiores tempos de processamento na encriptação e decriptação em todo o algoritmo.

Sobre uma análise espacial e temporal, foram feitos alguns testes que serão discutidos a seguir. Primeiramente, para a codificação ASCII, cada caractere corresponde a 1 byte. Para armazenar inteiros positivos até K são necessários cerca de $\log_2 K$ bits. Exemplificando com os valores utilizados no algoritmo implementado: a criptografia foi realizada agrupando o texto em blocos de 95 caracteres cada, 97x8 = 776 bits de informação. Os primos escolhidos resultam em um valor 'n' de aproximadamente 10^{235}, valor que limita 'm' em cerca de $\log_2 10^{235}$, ou seja, 781 bits de informação. Assim, essencialmente, a quantidade de espaço ocupado após a encriptação é a mesma da mensagem original. Isso ocorre porque o protocolo reversível a ser utilizado é uma função bijetiva e o algoritmo RSA é, portanto, reversível, resultando em uma correspondência entre o limitante 'n', produto dos primos grandes, e a quantidade de caracteres que podem ser agrupados em um bloco a ser criptografado. Dessa forma, para a análise temporal, foi construído a tabela a seguir baseada no número de palavras.

Tabela 1 – Tempo de processamento do algoritmo para cada etapa.

Nº de palavras	Leitura da mensagem[s]	Conversão[s]	Encriptação[s]	Decriptação[s]	Desconversão[s]	**Total[s]**
100	0.004	0.000	0.032	0.036	0.004	**0.076**
200	0.000	0.000	0.060	0.064	0.004	**0.128**
300	0.000	0.004	0.096	0.096	0.012	**0.204**
400	0.000	0.004	0.128	0.128	0.012	**0.268**
500	0.004	0.004	0.180	0.192	0.020	**0.400**
600	0.000	0.008	0.188	0.188	0.020	**0.404**
700	0.000	0.004	0.220	0.220	0.020	**0.464**
800	0.000	0.008	0.260	0.256	0.024	**0.548**
900	0.004	0.008	0.284	0.280	0.032	**0.608**
1000	0.004	0.004	0.328	0.308	0.032	**0.676**
2000	0.000	0.012	0.625	0.620	0.068	**1.325**
3000	0.000	0.016	0.962	0.936	0.104	**2.018**
4000	0.004	0.020	1.270	1.272	0.152	**2.719**
5000	0.000	0.024	1.608	1.572	0.212	**3.416**
6000	0.000	0.032	1.888	1.878	0.248	**4.046**
7000	0.000	0.036	2.212	2.180	0.308	**4.736**
8000	0.000	0.040	2.544	2.496	0.360	**5.440**
9000	0.000	0.044	2.884	2.828	0.444	**6.200**
10000	0.000	0.048	3.112	3.124	0.564	**6.8485**
20000	0.010	0.090	6.081	6.085	1.650	**13.916**
30000	0.000	0.133	9.118	9.123	3.700	**22.074**
40000	0.000	0.180	12.160	12.210	4.937	**29.487**
50000	0.000	0.240	15.296	15.684	8.051	**39.272**
60000	0.008	0.280	18.848	18.954	11.848	**49.939**
70000	0.008	0.332	21.980	22.095	15.396	**59.812**
80000	0.008	0.376	25.214	25.432	19.755	**70.785**
90000	0.008	0.428	28.252	27.514	24.188	**80.389**
100000	0.004	0.484	31.776	31.520	36.028	**99.812**

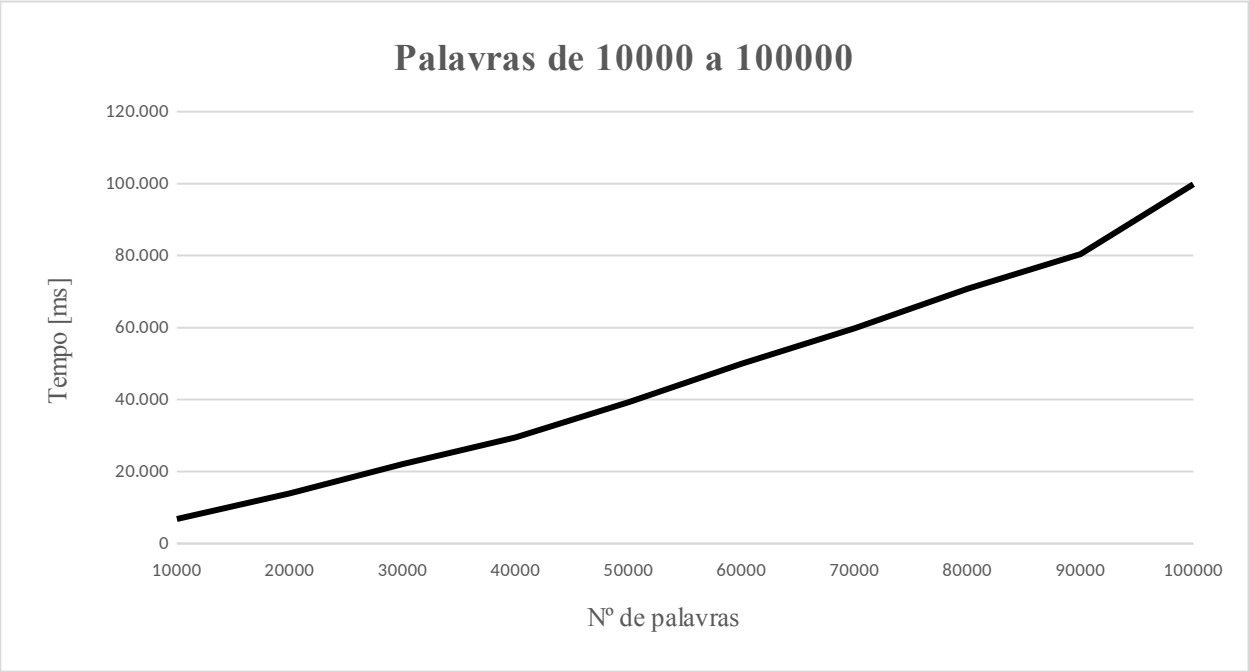

Figura 3 – Tempo de processamento de até 100.000 palavras.

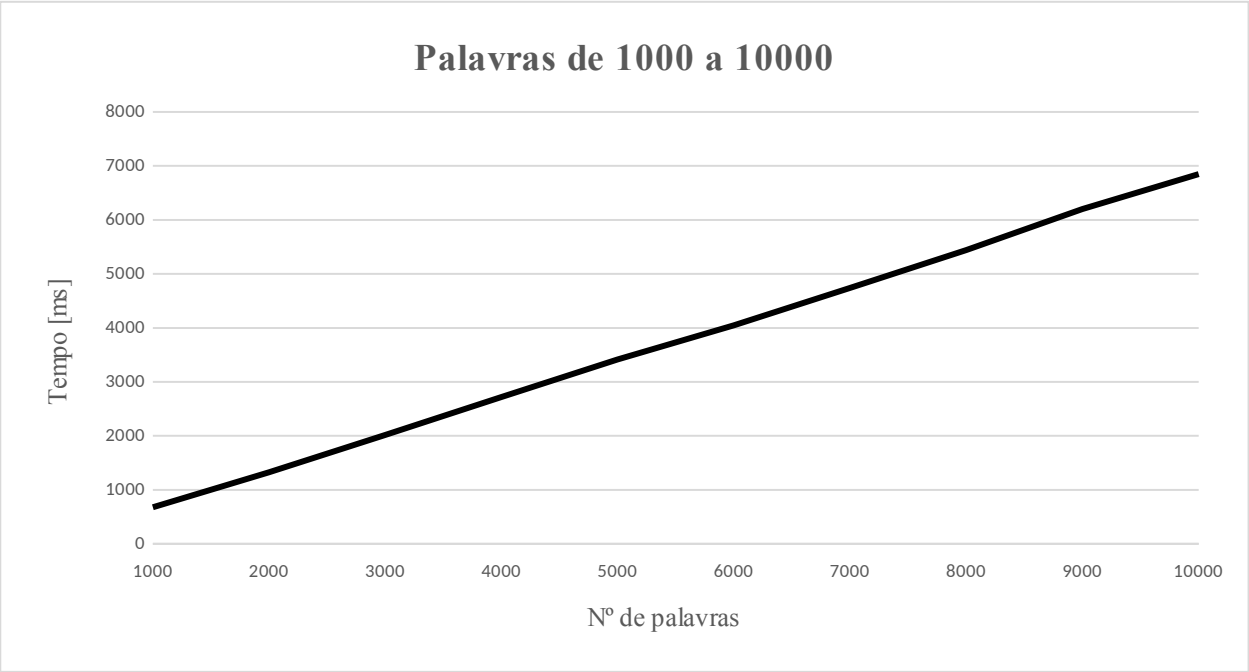

Figura 4 – Tempo de processamento de até 10.000 palavras.

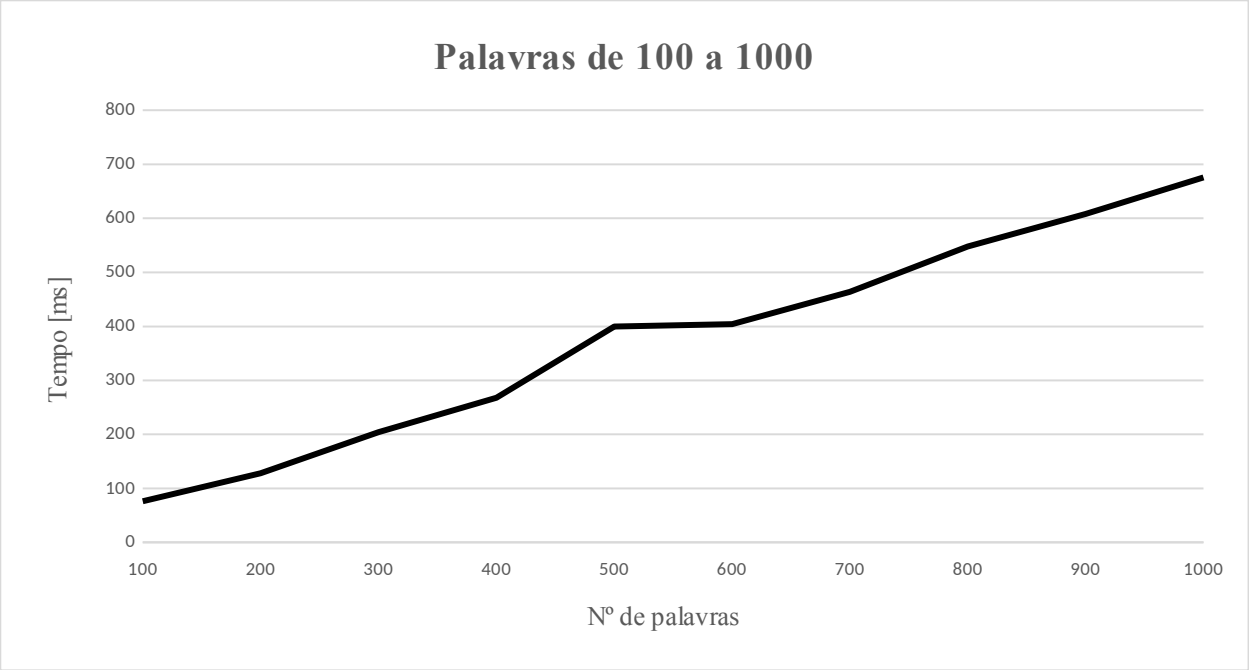

Figura 5 – Tempo de processamento de até 1.000 palavras.

Como é possível observar, todos os gráficos têm o comportamento linear do tempo de execução em função do número de palavras no texto, palavras de 2 a 11 caracteres geradas aleatoriamente dentro dos caracteres da tabela ASCII – em média palavras de 5 letras. Devido ao valor constante de ‘n’ nas simulações, o tempo de execução depende praticamente apenas do número de blocos a serem criptografados, que cresce linearmente com o número de caracteres presentes no texto.

Considerações Finais

A implementação do sistema criptográfico baseado no método RSA para textos em português foi bem sucedida ao realizar a criptografia de caracteres especiais, ao ter uma ordem linear e a utilizar números primos e blocos suficientemente grandes para inviabilizar a sua quebra por força bruta.

Observou-se que a maior parte do tempo de processamento da implementação se deu na encriptação e na decriptação das mensagens, o que pode ser justificado pela da exponenciação modular. Uma melhoria do algoritmo seria na otimização dessa função matemática, diminuindo assim o tempo de processamento.

A fim de melhorar a segurança da implementação cada bloco de caracteres poderiam ter pelo menos um espaço destinado a um caractere aleatório, o que deixaria a mensagem criptografada não determinística. Dessa forma, não seria possível descobrir a mensagem por força bruta, uma vez que a mesma mensagem teria um código criptografado diferente. Para o destinatário bastaria descriptografar a mensagem e retirar o caractere aleatório, presente em uma posição conhecida, como no final de cada bloco.

Finalmente, o projeto cumpriu com a proposta pedagógica de aprofundar os conceitos de matemática discreta aprendidos em sala de aula, além de contextualizar o conteúdo com a aplicação da criptografia RSA, uma importante área de conhecimento da Engenharia de Computação.

Referência Bibliográfica

[1] de Oliveira, P. E. R.; de Andrade, P. T. E.; D'Oliveira, R. L. G. *Rsa*. Disponível em: <http://www.ime.unicamp.br/~ftorres/ENSINO/MONOGRAFIAS/oliv_RSA.pdf>, acessado em: 11 julho. 2017.

[2] Disponível em: <https://pt.wikipedia.org/wiki/Fun%C3%A7%C3%A3o_totiente_de_Euler>, acessado em: 11 julho. 2017.

[3] RSA; Wikipedia. Disponível em: <https://pt.wikipedia.org/wiki/RSA>, acessado em: 20 maio. 2017.

[4] Zanon, G. H. M.. *Criptografia Homomórfica*. São Paulo: Novembro de 2016. Disponível em: <http://bcc.ime.usp.br/tccs/2016/gustavozanon/monografia.pdf>, acessado em: 20 maio. 2017.

[5] Bollauf, M. F. *Criptografia baseada em Reticulados*. Campinas: 21 de Novembro de 2014. Disponível em: <http://www.ime.unicamp.br/~campello/geometria/maiara.pdf>, acessado em: 20 maio. 2017.

[6] Kleinjung et. al.; Factorization of a 768-bit RSA modulus; Disponível em: <https://eprint.iacr.org/2010/006>. Acessado em 4 de julho de 2017.

Apêndice

Arquivo RSA.py

```
import math, time # para medição dos tempos de execução
from ferramentas import suporte, classes # módulos auxiliares

# menor primo maior que 10^115
p =
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000079

# menor primo maior que 10^120
q =
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000079

n = p * q
ln = (p - 1) * (q - 1) # totiente(n)

e = suporte.encontra_coprimo(ln) # expoente da chave publica
d = suporte.inverso_modular(e, ln) # expoente da chave privada

chave_publica = classes.chave_criptica(n, e)
chave_privada = classes.chave_criptica(n, d)
#'p', 'q' e 'ln' devem ser mantidas em sigilo, pois podem ser usadas para calcular 'd'

# distribuição da chave
# suponha que Bob queira enviar uma mensagem secreta para Alice:
# usando o RSA, Bob deve saber a chave pública de Alice para
# encriptar a mensagem e Alice deve usar a chave privada para
# decriptar a mensagem. para que Bob possa enviar mensagens
# encriptadas, Alice transmite a chave pública (n, e) para Bob
# de uma forma não necessariamente segura. a chave privada 'd' de
# Alice nunca é distribuída.

Alice = classes.destinatario(chave_publica)
Bob = classes.mensageiro(chave_privada)

# encriptação
# depois de Bob obter a chave pública de Alice, ele pode enviar a
# mensagem 'M'. primeiro ele converte o texto em 'M' em um número
# inteiro 'm' tal que 0 ≤ m < n usando um protocolo reversível
# conhecido por Alice. em seguida, a mensagem é encriptada em 'c'
# utilizando a chave pública 'e' de Alice (c = m^e (mod n)).
# isto pode ser feito de forma razoavelmente fácil usando a
# exponenciação modular. enfim, Bob transmite 'c' para Alice.

Bob.M = suporte.ler_mensagem() # lê a mensagem salva no arquivo 'mensagem.txt'
print('> mensagem original: ' + str(len(Bob.M)) + ' caracteres')
print(Bob.M)
print()

Bob.conversor()

Bob.encriptar()
print('> mensagem encriptada: ' + str(len(Bob.c)) + ' blocos de dados (' +
str(classes.tam_bloco) + ' caracteres/bloco)')
for k in Bob.c:
   print(k, end='\n\n')

# decriptação
# Alice pode recuperar 'm' de 'c' usando o expoente 'd' da chave privada
# (que apenas Alice conhece) ao computar 'c^d = m (mod n)'. a partir de
# 'm', utilizando o protocolo reversível, Alice pode recuperar a mensagem
# 'M' original.
```

Arquivo classes.py

```
import time
from ferramentas import suporte

tam_bloco = 95  # agrupa a mensagem em blocos de 95 caracteres


# classe da chave (publica ou privada)
class chave_criptica:
    def __init__(self, n, k):
        self.n = n  # produto de dois primos grandes
        self.k = k  # expoente


class pessoa:
    def __init__(self, chave_pessoal):
        self.pk = chave_pessoal
        self.M = ''  # mensagem a ser enviada
        self.m = []  # mensagem convertida em números
        self.c = []  # mensagem encriptada


class mensageiro(pessoa):
    def __init__(self, chave_pessoal):
        pessoa.__init__(self, chave_pessoal)

    # transforma a mensagem 'M' (caracteres) em um conjunto de números 'm'
    def conversor(self):
        # a mensagem 'M' é separada em blocos de 'tam_bloco' caracteres. caso a
mensagem tenha um tamanho
        # que não seja múltiplo de 'tam_bloco', o bloco é completado com '\0'.
        for k in range(tam_bloco - len(self.M) % tam_bloco):
            self.M += '\0'
        # os blocos de 'tam_bloco' caracteres são transformados em número seguindo a
codificação ASCII e trabalhando na
        # base 256 (uma vez que a codificação ASCII admite 256 caracteres). por
exemplo, supondo 'tam_bloco = 6' o bloco
        # 'cripto' seria convertido em '99 * 256^5 + 114 * 256^4 + 105 * 256^3 + 112 *
256^2 + 116 * 256 + 111'
        # que equivale a '109343046399087'. aqui os caracteres foram convertidos em
seus valores ASCII ('c = 99', 'r = 114',
        # 'i = 105', 'p = 112', 't = 116' e 'o = 111') e tratados como dígitos de um
número de 6 dígitos na base 256.
        for k in range(len(self.M) // tam_bloco):
            valor = 0
            for j in range(tam_bloco):
                valor = valor * 256 + ord(self.M[k * tam_bloco + j])
            self.m.append(valor)

    # encripta a mensagem convertida 'm'
    def encriptar(self):
        for k in self.m:  # -> O(len(m)) . O(log2(e)) . O(% n)
            self.c.append(suporte.exponenciacao_modular(k, self.pk.k, self.pk.n) %
self.pk.n)


class destinatario(pessoa):
    def __init__(self, chave_pessoal):
        pessoa.__init__(self, chave_pessoal)

    # transforma a mensagem 'm' (números) decriptada na mensagem original 'M'
(caracteres)
    def desconversor(self):
        # o processo é o inverso do exemplificado no 'conversor'. os blocos são
```

Arquivo suporte.py

```python
import math

# lê a mensagem salva no arquivo 'mensagem.txt' e retorna uma string com o texto lido
def ler_mensagem(nome = 'mensagem.txt'):
    f = open(nome, 'r')
    line = f.readlines()
    f.close()
    return line[0]

def atualiza(var, nova_var, q):
    aux = nova_var
    nova_var = var - q * nova_var
    var = aux
    return var, nova_var

# calcula o MDC de 'a' e 'b'
def MDC(a, b): # -> O(log2(min(a, b)))
    while b != 0:
        r = a % b
        a = b
        b = r
    return a

# calcula 'c = m^e (mod n)'
def exponenciacao_modular(x, e, n): # -> O(log2(e)) . O(% n)
    return pow(x, e, n)

# calcula o inverso de 'a' módulo 'n'
def inverso_modular(a, n): # -> log²(n)
    t, r = 0, n
    novo_t, novo_r = 1, a
    while novo_r != 0:
        q = r // novo_r
        t, novo_t = atualiza(t, novo_t, q)
        r, novo_r = atualiza(r, novo_r, q)
    if r > 1:
        return 'não inversível'
    elif t < 0:
        return t + n
    return t

# encontra um número menor que 'x' que seja coprimo com 'x'
divisor = 10 # esta variável pode assumir qualquer valor entre 1 e 'x'
             # sua finalidade é impedir que as chaves pública e privada seja iguais
def encontra_coprimo(x):
    y = x // divisor
    while y > 1: # -> O(log2(y) * x)
        if MDC(y, x) == 1:
            return y
        y -= 1
    y = (x // divisor) + 1
    while y < x: # -> O(log2(x) * x)
        if MDC(y, x) == 1:
            return y
        y += 1
```

Arquivo TempoRSA.py


```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


import math, time # para medição dos tempos de execução
from ferramentas import suporte, classes # módulos auxiliares



# Chave de 2048 bits, obtida pelo OpenSSL
p =
0x00f0d1d6f7167056d39e22cd86ae0c4cc7ef4673e9b7799191d32446c78d3e77e8c90a9192170f60f482d5af25216dac068067e71e8e
q =
0x00d35b3c8c74c84e0200baff2f400e405e353b9bcd089fc1dc684460512adea0359c17b972586900c224aaf73ee3ffb2258d731c44bc
n = p * q
ln = (p - 1) * (q - 1)
e = 0x10001 # Número primo arbitrário
d =
0x00ac879ff5a53c3cf392e2fcaafc610ddc759b861e93f880289452f5a6131432f322d30869962846e02859977532e5b37cbdfc9e6bb0

"""
# Chave de 781 bits criada pelo grupo
p =
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
q =
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
n = p * q
ln = (p - 1) * (q - 1) # totiente(n)
e = suporte.encontra_coprimo(ln) # expoente da chave publica
d = suporte.inverso_modular(e, ln) # expoente da chave privada
"""

chave_publica = classes.chave_criptica(n, e)
chave_privada = classes.chave_criptica(n, d)



mensagem = ''
tempos = []
quants = []
quant = 0
for i in range(100):
    mensagem +=
'5b37cbdfc9e6bb0bcb3746c5fbb68da270b91184e3861c48442795314bdf4ff351e2a3590657d41297b1043b5e0616d43d958d9f3b3e1
    quant += 400
    Alice = classes.destinatario(chave_publica)
    Bob = classes.mensageiro(chave_privada)
    Bob.M = mensagem
    ti = time.time() # início da execução
    Bob.conversor()
    Bob.encriptar()
    Alice.c = Bob.c
    Alice.decriptar()
    Alice.desconversor()
    tf = time.time()
    tempos.append(tf-ti)
    quants.append(quant)
    print(i)
```