UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIAS DE COMPUTAÇÃO

Engenharia de Segurança - SSC0747 - 2018

Trabalho 1 - Criptografia

Prof. Dr. Kalinka Regina Lucas Jaquie Castelo Branco

Eduardo Garcia Misiuk	**Número USP: 9293230**
Lucas Yudi Sugi	**Número USP: 9293251**
Maurício Caetano da Silva	**Número USP: 9040996**

Introdução

Está sendo cada vez mais frequente encontrarmos dispositivos conectados em rede, de modo que, é comum ver aparelhos domésticos inteligentes que realizam uma série de processamentos e troca de dados via internet. Nesse ponto, origina-se o famoso termo Internet of Things (IoT).

Assim, podemos afirmar que quase tudo está conectado. Tal fato implica em uma grande quantidade de informações trafegando na rede, o que nos leva a seguinte pergunta: como proteger tais dados ?

Muitas dessas informações são privadas, não podendo ser expostas para terceiros. Logo, com o intuito de garantir uma segurança em tais dados, surge a idéia de se utilizar criptografia.

Objetivos

O objetivo deste documento é estudar quatro algoritmos de criptografia, abordando os seguintes pontos:

- Tipo do algoritmo
- Força da chave
- Modo de compartilhamento das chaves
- Característica de funcionamento

Após estudá-los, será escolhido um para que possamos realizar modificações com a finalidade de torná-lo mais forte, isto é, fazer com que seja mais difícil descobrir a chave que o algoritmo usa via força bruta com um sistema computacional extremamente potente.

Por fim, será discutido porque tais modificações foram realizadas e como elas podem tornar o algoritmo mais forte.

Discussão dos algoritmos

A seguir iremos estudar quatro algoritmos de criptografia:

Data Encryption Standard (DES)

DES é um algoritmo de criptografia simétrica em blocos que utiliza a cifra clássica de Feistel somado a duas permutações. Abaixo podemos ver uma figura do seu funcionamento:

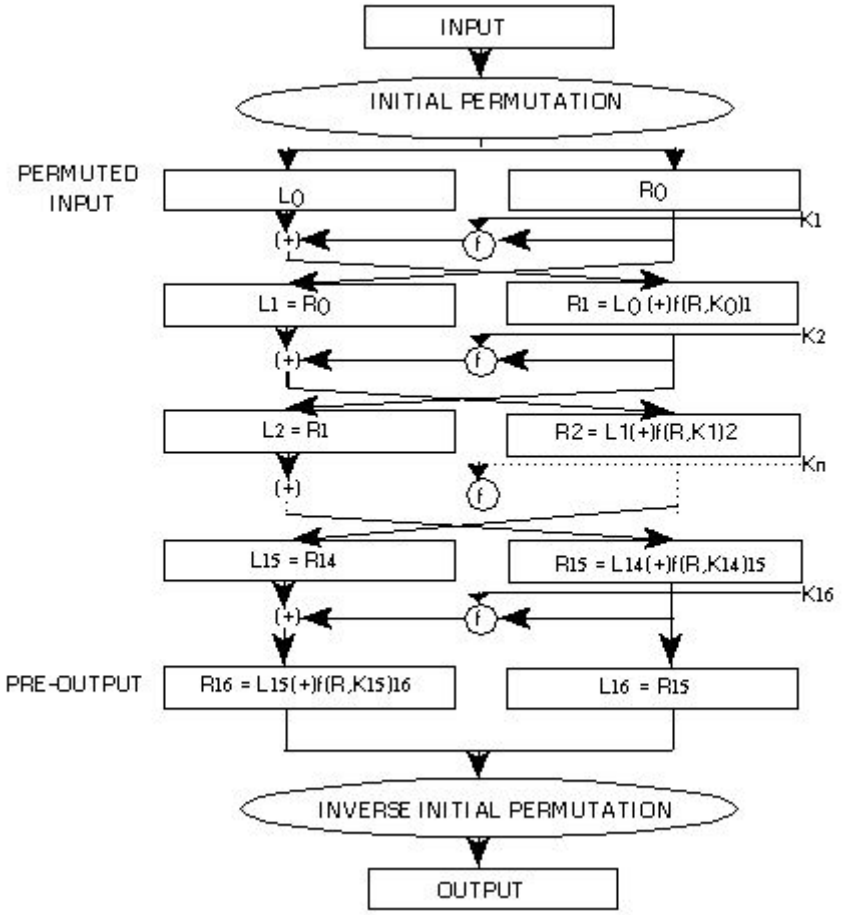

Figura 1: Cifra de Feistel.

Dada a imagem acima, podemos explicar o passo a passo do DES:

1. Entramos com um texto plano que será convertido para binário e quebrado em blocos de 64 bits. Caso não seja possível completar o bloco, é realizado um preenchimento com bits 0.
2. Cada bloco receberá uma permutação inicial.
3. Dividir a chave K de 56 bits em subchaves (K_i) para cada rodada.
4. Dividimos o bloco em dois (32 bits cada), chamados de L_i (Left i) e R_i(Right i), em que i indica qual rodada estamos.
5. Para i = 0 até i = 15, faremos:
   5.1. Aplicar R_i junto com K_i na função rodada (F).
   5.2. A saída do passo anterior é utilizada com um XOR L_i.
   5.3. Trocar os resultados, isto é, $L_{i+1} = R_i$ e $R_{i+1} = L_i$ XOR $F(R_i + K_i)$
6. Realizar a permutação inversa.
7. Juntar todos os blocos, converter para texto e enviar para a saída.

Basicamente estas são as etapas do algoritmo visto em alto nível. Devemos agora aprofundar certos passos que foram omitidos:

1. Função rodada(F):
    1.1. Aplicamos uma função de expansão em R_i (divisão direita do bloco de 64 bits) na qual os dois últimos bits de cada linha são expandidos, ou seja, dispomos os seis primeiros bits de R_i em uma linha. Na próxima, os dois primeiros serão iguais aos dois últimos da linha anterior. Utilizamos dessa lógica até termos 8 linhas por 6 colunas totalizando 48 bits.
    1.2. Realizamos uma operação XOR da saída anterior com a subchave K_i.
    1.3. O resultado servirá de entrada para as S-Boxes. Existe um total de 8 delas, na qual cada uma recebe 6 bits e produz como saída 4 bits. Terminando a função de rodada.
2. S-Boxes: Nós temos uma tabela em que cada célula possui algum valor de quatro bits. Esta célula está indexada da seguinte maneira: cada linha é representada por 2 bits e cada coluna por 4. Logo, com os 6 bits de entrada nós selecionamos uma célula da tabela e realizamos a substituição.
3. Permutação inicial: Temos uma tabela 8x8 que iremos chamar de IP, na qual o valor de cada célula representa uma posição bit e os índices dela representam a nova posição. Por exemplo, imagine que temos [0][0] = 58 em IP, então o bit da posição 58 no bloco, será colocado em [0][0] em um novo bloco permutado.
4. Permutação inversa: A permutação inversa ocorre da mesma forma que a etapa anterior, diferindo que ela tentará desfazer a permutação inicial (realiza a operação contrária).
5. Geração das subchaves: Dada a chave inicial K, nós realizamos uma permutação da mesma forma que anteriormente, segundo uma tabela (CP1). Dividimos o resultado em duas partes chamadas de C_i e D_i que irão receber um rotação circular a esquerda cada um. A quantidade de vezes que a rotação ocorre depende de uma tabela. Após essa operação C_i e D_i são permutados em uma outra tabela (CP2) que resultará em uma chave K_i de 48 bits.

Abaixo temos 4 figuras representando a função de rodada e as S-boxes.

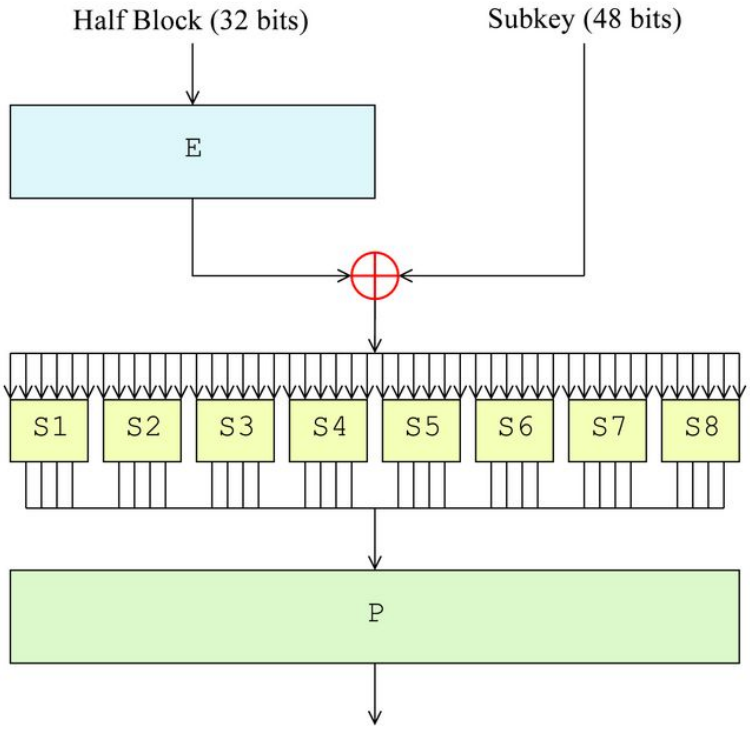

Figura 2: Esquema da função de rodada.

S6		Middle 4 bits of input															
		0000	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
Outer bits	**00**	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
	01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1010	0011	1001	1000	0110
	10	0100	0010	0001	1011	1010	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
	11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1010	0100	0101	0011

Figura 3: Exemplo de uma tabela que o S-box 6 usa.

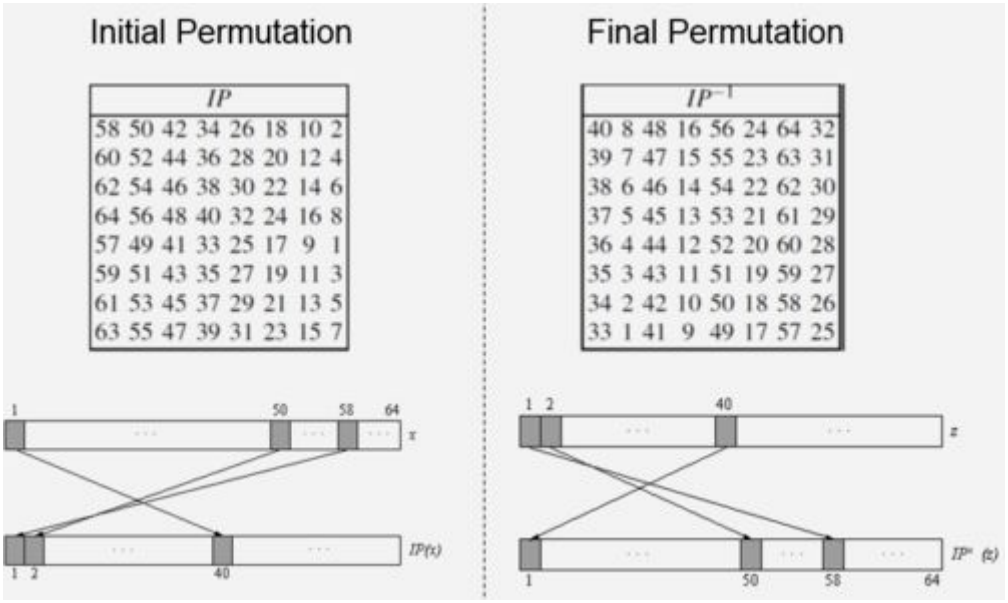

Figura 4: Tabela IP e IP^{-1} que são utilizadas para permutação inicial e inversa, respectivamente.

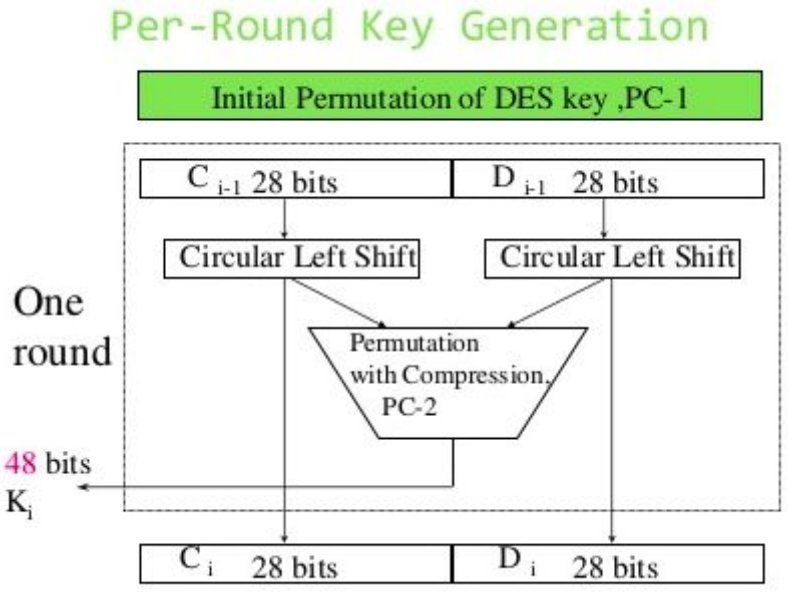

Figura 5: Geração das subchaves de rodada.

Agora que sabemos como funciona a criptografia no DES, podemos falar sobre a descriptografia. Uma das vantagens do DES é que não precisamos de dois algoritmos, um para criptografar e outro para descriptografar. Para levar o texto cifrado ao texto plano basta inverter a ordem das subchaves nas rodadas. Apenas realizando esse processo contrário é possível obter o texto original.

Após todas essas explicações, será descrito como os parâmetros do DES podem ser modificados a fim de torná-lo mais forte:

- Tamanho do bloco: Quanto maior for o tamanho do bloco maior será a segurança, contudo, isso implicará em uma maior lentidão no processo.
- Tamanho da chave: Mesma coisa que o tamanho do bloco. Sendo que inicialmente era utilizado chaves de 64 bits (são utilizados 56 bits pois o resto é utilizado para paridade), o que mostrou ser muito frágil. Atualmente, utiliza-se 128 bits.
- Número de rodadas: Quanto mais rodadas tivermos maior será a segurança, porém, maior será o tempo de execução.
- Geração da subchave: Métodos mais complexos para sua geração irão dificultar a criptoanálise.
- Função rodada: Novamente, quanto mais complexa maior resistência a criptoanálise.

Analisando tais parâmetros, pode-se acreditar que DES é um bom algoritmo de criptografia e que sua chave é forte, mas na verdade não é bem assim. Em julho de 1998 a Electronic Frontier Foundation realizou um ataque ao DES com uma máquina de 250 mil dólares, conseguindo quebrá-lo em apenas 3 dias, mostrando a fragilidade do algoritmo.

Além disso, muitos acreditam que existem fraquezas nas S-boxes (ainda não foram descobertas) que permitem realizar uma criptoanálise. Portanto, o DES não é um algoritmo de criptografia seguro e seu uso é desencorajado, apesar de ainda ser utilizado em algumas aplicações.

Advanced Encryption Standard(AES)

AES é um algoritmo de cifra simétrica em blocos que surgiu com o intuito de substituir o DES devido as suas fraquezas. O NIST (National Institute of Standards and Technology) realizou um concurso para escolher um algoritmo de criptografia, na qual o vencedor acabou sendo Rijndael, por atender melhor os quesitos propostos por NIST.

Dado isso vamos discorrer um pouco sobre o AES. Ele tem a vantagem de combinar os blocos e chaves em qualquer tamanho dentro de 128 bits, 192 bits e 256 bits. Além disso, deve-se salientar que o número de rodadas dele é determinado pelo quantidade de chaves que utiliza (ele utiliza esse processo que é parecido com o esquema de Feistel mas não igual).

Tendo essas informações vamos explicar como ocorre seu funcionamento considerando um bloco e uma chave de 128 bits:

1. Separar o texto plano e a chave em blocos de 4x4 sendo que cada célula é um valor em hexadecimal, portanto, 1 byte.
2. Faça um AddRoundKey com a chave original.
3. Para i = 0 até i = 8 faça:

3.1. SubBytes: Cada célula do bloco será substituída segundo uma tabela chamada de S-box, com o seguinte critério: Os 4 bits mais à esquerda da célula em hexadecimal definem a linha, os 4 mais à direita à coluna em que você irá acessar a S-box para realizar a substituição.

3.2. ShiftRows: Para cada linha do bloco será realizado um deslocamento circular à esquerda segundo o seguinte critério: A linha j irá rotacionar j-1 vezes, sendo que j inicia em 1. Realizar isso até a última linha do bloco.

3.3. MixColumns: Cada coluna será mapeada para um novo valor que é uma combinação linear de todos os outros bytes desta coluna.

3.4. AddRoundKey: Faça um XOR byte a byte do bloco com a chave da rodada.

4. Para i = 9 (última rodada) faça SubBytes, ShiftRows e AddRoundKey.
5. Una todos os blocos e você possui o texto criptografado.

Abaixo nós temos imagens que tentam demonstrar as 4 operações de forma visual:

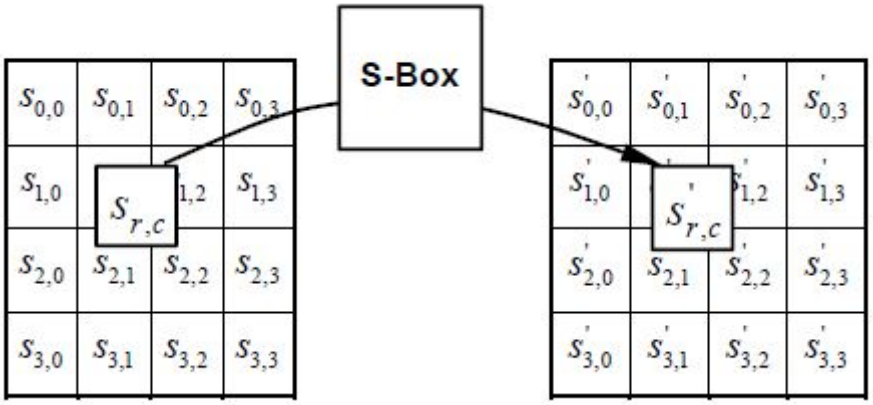

Figura 6: Funcionamento de SubBytes via uma S-box

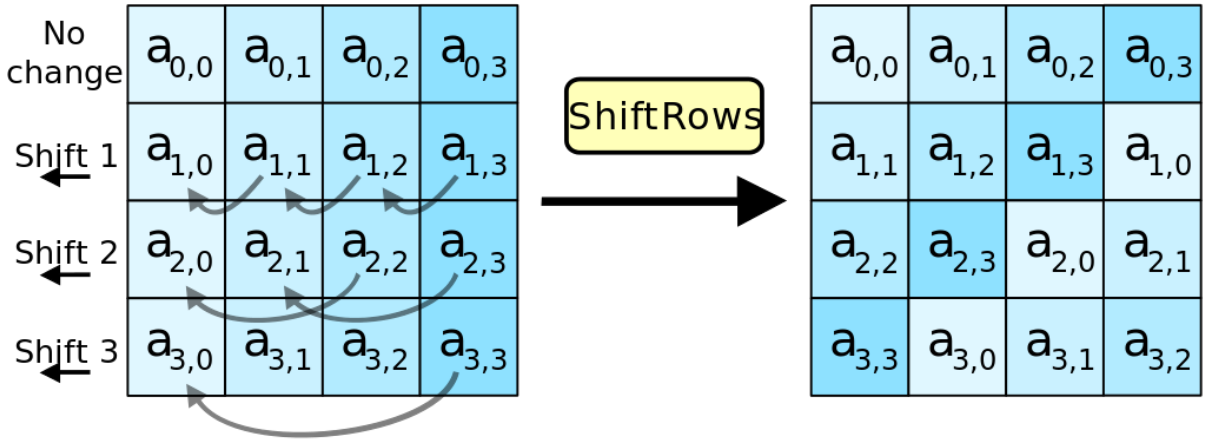

Figura 7: Funcionamento de ShiftRows.

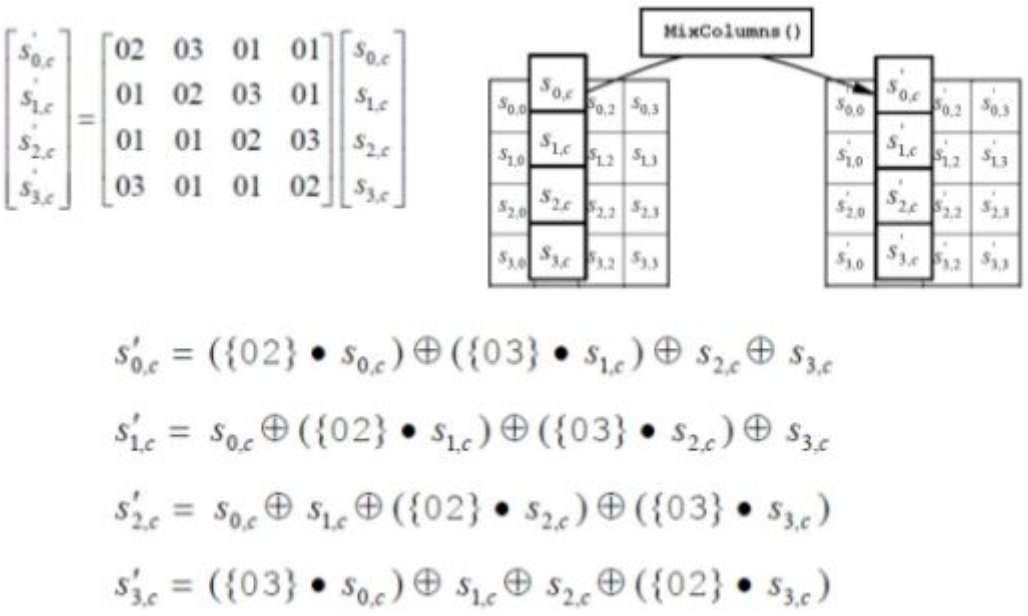

Figura 8: Funcionamento de MixColumns.

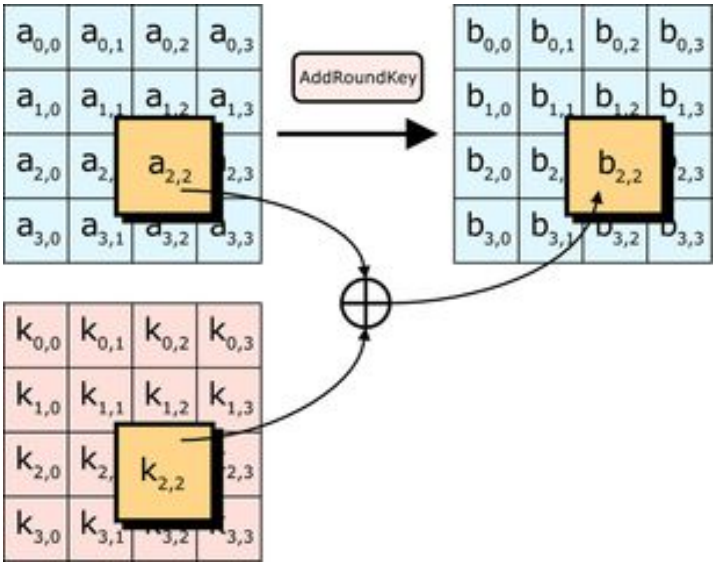

Figura 9: Funcionamento de AddRoundKey.

Esta é a estrutura de funcionamento básica do AES. Falta apenas explicar como é gerada a chave em cada rodada que é utilizada em AddRoundKey:

1. Crie uma matriz w que possua 4 linhas e 44 colunas. Nós iremos colocar os blocos das chaves nelas.
2. Copie para as 4 primeiras colunas a chave dada.
3. Dado que a primeira coluna é i = 0, então, de i = 4 até i = 43 faça:

3.1. Copie a coluna anterior para um vetor temporário, isto é, temp = w[i-1].
3.2. Se (i mod 4 = 0) então realize uma rotação circular a esquerda em w[i-1], seguido de uma substituição via SubBytes. Com o resultado faça um XOR com uma constante da rodada chamada de Rcon e armazene em temp.
3.3. Faça novamente um XOR com temp e w[i-4].

É dessa maneira que são geradas as novas chaves de cada rodada. Devemos apenas explicar o que é a constante da rodada (Rcon). Ela é um vetor que é calculado da seguinte maneira: Rcon[j] = (Rc[j],0,0,0) na qual Rc[j] = $x^{(i-4)/4}$ mod $(x^8 + x^4 + x^3 + x + 1)$

Explicado todo o processo de criptografia, podemos falar sobre a descriptografia. Para realizá-la, podemos manter a mesma estrutura explicada anteriormente, só que devemos modificar as 4 operações: SubBytes, ShiftRows, MixColumns e AddRoundKey, além de aplicar as chaves de rodada na ordem inversa. Tais operações devem ser realizadas no modo inverso. Feito isso, é possível descriptografar o texto cifrado.

Realizadas as explicações, podemos falar sobre a força do algoritmo. Sua segurança está baseada nas 4 operações descritas anteriormente. Todas elas podem parecer bem simples, mas possuem um caráter matemático muito forte. Para cada uma delas, existe uma elaboração lógica e bem estruturada que permite até mesmo provar que o algoritmo é seguro. Como se não bastasse, o AES permite chaves muito grandes, por exemplo, 256 bits. Tal tamanho faz com que seja impraticável realizar ataques de força bruta com o poder computacional atual.

Além disso, podemos citar que o AES veio com o intuito de substituir o DES, tentando ser imune aos ataques da época. Podemos concluir, portanto, que esse algoritmo de criptografia é bem forte.

RC4

É um algoritmo de cifra de fluxo simétrico, logo, diferentes dos outros dois algoritmos abordados em que se realiza uma criptografia por bloco; aqui, nós criptografamos byte a byte. Ele foi criado por Ron Rivest para a RSA Security sendo bem famoso e utilizado. Na verdade, protocolos como SSL/TLS, WPA e WEP utilizam-se dessa cifra.

O algoritmo permite utilizar chaves de tamanho variável que vão de 8 a 2048 bits, na qual são utilizadas para inicializar um vetor de estado S de 256 bytes. Tal vetor irá conter em suas células uma permutação dos valores de 0 a 255.

Dado isso podemos explicar o funcionamento do algoritmo:

1. Para i = 0 até i = 255 faça:
   1.1. S[i] = i
   1.2. T[i] = K[i mod keylen]
2. Inicialize j = 0 e para i = 0 até i = 255 faça:
   2.1. j = (j + S[i] + T[i]) mod 256
   2.2. Swap(S[i],S[j])
3. Inicialize j = 0 e faça infinitamente:
   3.1. i = (i+1) mod 256
   3.2. j = (j + S[i]) mod 256
   3.3. Swap(S[i],S[j])
   3.4. t = (S[i] + S[j])
   3.5. k = S[t]
4. Com o k da etapa anterior, nós o utilizamos para realizar um XOR com o próximo byte do texto plano (criptografar) ou do texto cifrado (descriptografar).

Note que em 1 estamos inicializando S de forma crescente e copiando os keylen valores da chave K, que caso seja menor que 255 será repetido em T.

Em 2 estamos realizando uma permutação de S com a ajuda de T. Como a única operação sobre S é uma troca, então ela conterá todos os valores de 0 a 255.

Já em 3, aplicamos a geração de fluxo, isto é, o algoritmo gerará infinitos valores de k que serão utilizados para criptografar byte a byte uma mensagem.

Perceba que é bem simples explicar RC4, na verdade, esta é uma das vantagens dos algoritmos de fluxo. Além de serem rápidos, possuem códigos relativamente simples de implementar.

Discutido o seu funcionamento devemos falar agora sobre a sua força. RC4 consegue utilizar chaves bem grandes para criptografia o que dificulta em muito ataques de força bruta. Ao longo dos anos tentaram-se realizar ataques mas nenhum deles se mostrou prático o suficiente. O único problema que houve é com relação ao protocolo WEB em que seu modo de geração de chaves deixa vulnerabilidades. Mas isso não é um problema em si do RC4, bastando apenas mudar o modo como são geradas tais chaves.

Portanto, podemos concluir que o algoritmo é bem forte para criptografia.

RSA

É um algoritmo de criptografia de chave pública que realiza a cifra por blocos. A estrutura de funcionamento da sua criptografia e descriptografia é bem simples. Considere que:

M = Texto plano
C = Texto Cifrado

e = Expoente
d = Expoente
n = número produzido pela multiplicação de dois primos p,q

Então, temos que a criptografia e descriptografia são dadas pelas seguintes equações:

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

$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

Note que, a partir das equações, as chaves privada e pública são dadas respectivamente por {d,n} e {e,n}. Lembre-se que algoritmos criptográficos de chave pública possuem um par de chaves.

Apesar de ser bem fácil definir sua estrutura, para que possa funcionar e tenha segurança, certas características devem ser atendidas:

1. Os valores determinados devem satisfazer $M^{ed} \bmod n = M$ para $M < n$.
2. Ser inviável determinar d dados e {e,n}.
3. Ser relativamente fácil encontrar M^e mod n e $C^d \bmod n$ para todos os valores de $M < n$.
4. $n = pq$ onde p e q são primos.
5. Calcular a função totiente $tot = mmc((p-1)(q-1))$
6. Calcular o inverso multiplicativo de e mod totiente dado por: $ed \equiv 1 \bmod$ tot

Essas considerações são necessárias por razões matemáticas, mais especificamente em aritmética modular. Com elas, podemos provar o porque devemos ter tais restrições. Como não é intuito deste documento demonstrar tais provas, iremos concentrar nossos esforços em explicar outros detalhes do algoritmo.

Dados os pontos acima, discutiremos como calcular os valores necessários para o algoritmo:

1. Selecione dois primos p e q.
2. Calcule $n = pq$.
3. Calcule $(p-1)(q-1)$.
4. Selecione e de modo que seja coprimo de e e menor que $(p-1)(q-1)$.
5. Determine d tal que $de \equiv 1 \bmod (p-1)(q-1)$. Isso é facilmente calculado utilizando o algoritmo estendido Euclides.

Com isso, temos todos os valores necessários para constituir o par de chaves do algoritmo. Iremos agora discorrer sobre a força do RSA.

A chave do RSA é bem segura dado que podemos utilizar valores bem grandes de modo a impedir ataques de força bruta. Nesse ponto, o algoritmo acaba se saindo muito bem. O seu maior problema, na verdade, se dá por conta de ataques matemáticos. Isso porque o RSA é forte por basear-se na ideia de que é muito difícil fatorar o produto de dois números primos grandes.

Contudo, cada vez mais tenta-se desenvolver técnicas que buscam realizar tal fatoração de modo rápido. Deve-se levar em conta isto na hora de escolher as chaves.

Portanto, o RSA é um algoritmo seguro até o momento em que ninguém consiga, em tempo satisfatório, resolver a fatoração do produto de dois números primos grandes.

Distribuição das chaves

Nas seções anteriores nós discutimos sobre a força do algoritmo, seu funcionamento e tipo. Para concluir esta etapa, falta explicar como ocorre a troca de chaves de cada um deles, fato que é extremamente necessário para que possa haver o uso da criptografia.

Para os algoritmos simétricos, o mais comum é utilizar um Centro de Distribuição de Chaves (CDC ou KDC em inglês) que funciona da seguinte forma:

- Uma entidade A envia para o CDC sua identidade, a identidade de B com que deseja-se comunicar e um nonce N_1 que é utilizado para identificar a conexão de modo único. Tudo isso criptografado com uma chave K_a que somente A e CDC conhecem.
- O CDC envia uma resposta que contém uma chave de sessão K_s, as identidades de A e B e o mesmo nonce N_1 que são criptografados com K_a. Além disso, é enviado K_s e a identidade de A criptografados com K_b, chave que somente B e CDC conhecem.
- A envia o conteúdo criptografado com K_b para B de modo que somente ele irá saber o conteúdo.
- B responde a mensagem com um novo nonce N_2.
- A responde com a aplicação de uma função $F(N_2)$ que pode ser somar 1 ao nonce de modo que eles consigam identificar a conexão de maneira única.
- Agora eles podem realizar a sua comunicação.

Abaixo temos uma figura que demonstra graficamente o uso de um CDC para distribuir as chaves simétricas:

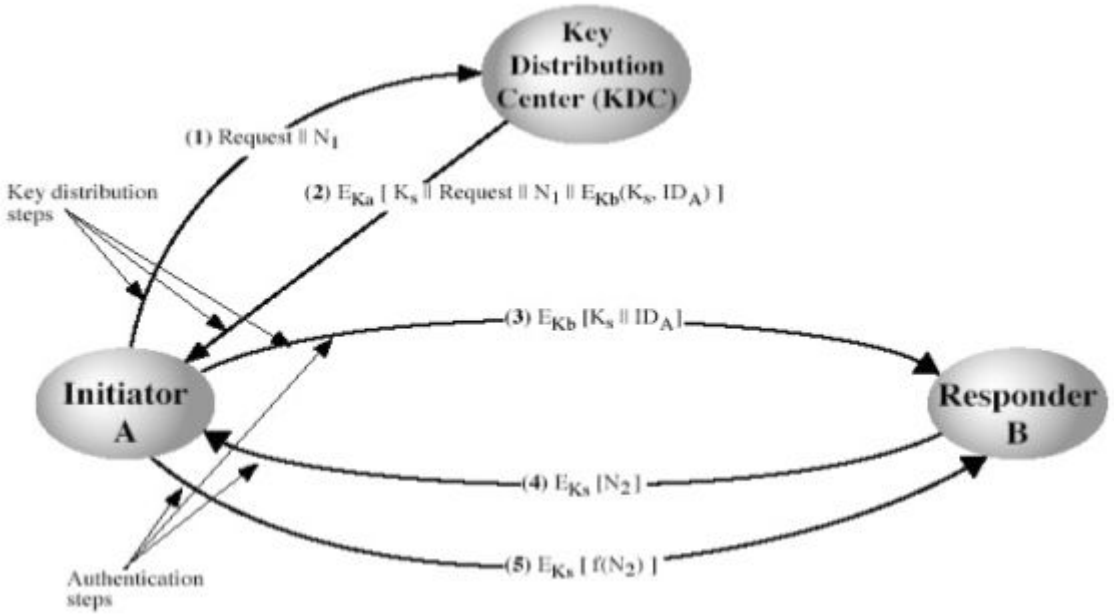

Figura 10: Distribuição de chaves simétricas.

Para os algoritmos de chave pública, inicialmente surgiu-se a ideia de que, como nós temos uma chave que é pública, basta torná-la disponível para os outros que desejam se comunicar com você. Realmente isso funciona, pois quem deseja lhe enviar uma mensagem realiza uma criptografia com sua chave pública, logo, somente você, que possui a chave privada correspondente, pode descriptografar e ver o conteúdo da mensagem.

O problema aqui está que outra pessoa pode se passar por você oferecendo a chave pública dela. Logo, ela poderá ver o conteúdo da mensagem. Esse cenário é muito comum de ocorrer em ataques de homem do meio.

Dado esse problema, para que fosse possível resolvê-lo foi criado uma Autoridade de Certificadora (AC). Ela será responsável por emitir certificados que irão afirmar que a entidade A realmente é A, ou seja, que a chave pública K_a pertence a quem diz ser seu dono. Desse modo, é possível que outras pessoas possam enviar mensagens criptografadas sem correr o risco de estar utilizando uma chave pública errada.

Abaixo temos uma figura do seu funcionamento:

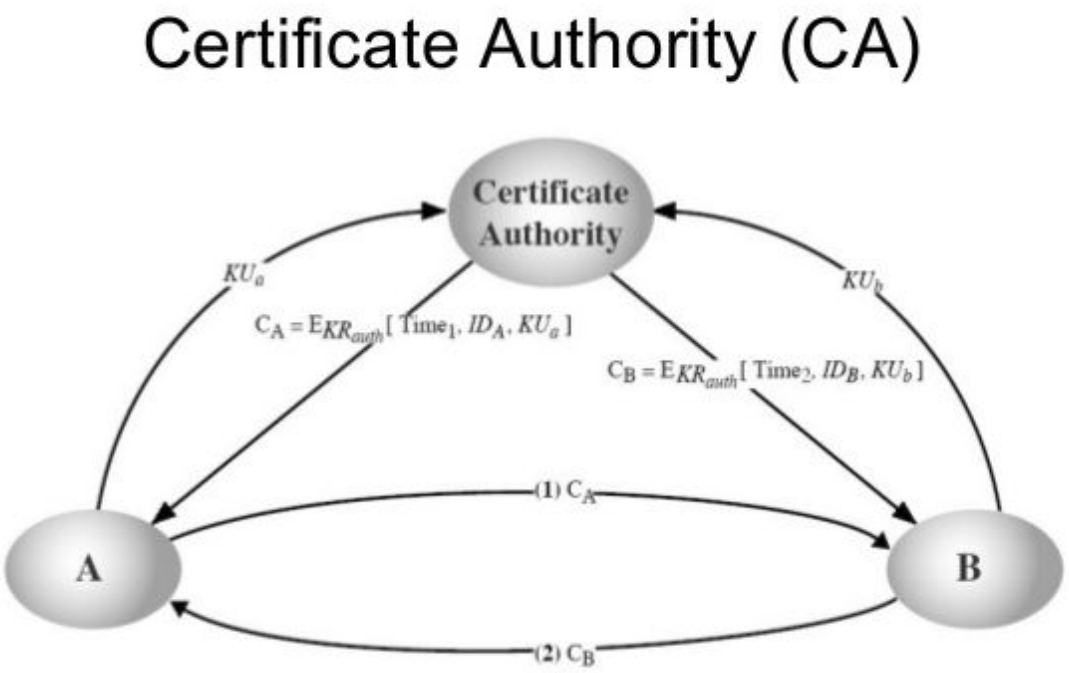

Figura 11: Distribuição de chaves assimétricas.

Escolha do algoritmo de implementação

Discorrido sobre quatro algoritmos de criptografia, nós devemos escolher um para que possamos realizar modificações de modo a torná-lo mais forte. No caso, foi escolhido o RSA pelas seguintes razões:

- Facilidade de entendê-lo
- Facilidade em implementá-lo
- Cifra forte
- Amplamente utilizado
- Conhecimento prévio em aritmética modular

Tais pontos foram cruciais para a escolha do algoritmo, pois com eles será possível melhor adaptá-lo de modo a torná-lo mais forte.

EML RSA

A nossa implementação do RSA é chamada de EML RSA, sendo EML as iniciais dos autores, para diferenciar-se da implementação original. O código pode ser encontrado no repositório https://github.com/eduardogmisiuk/eml-rsa e conta com a licença GPL v3.

Antes de passarmos à explicação das mudanças feitas no RSA original, serão descritas as versões de programas, bibliotecas e sistemas operacionais que foram utilizados para sua confecção e teste.

Plataforma, linguagem e utilitários

O EML RSA foi implementado em C++11 e testado nas seguintes plataformas:

- Máquina virtual de Ubuntu 18.04 LTS 64 bits, kernel release version 4.15.0-22-generic, VM feita no Oracle VM VirtualBox versão 5.2.6 r120293;
- Máquina nativa com sistema operacional Linux Mint 18.2 KDE, kernel version 4.13.0-36-generic, 64 bits.

Utilitários utilizados:

- g++ versão 7.3.0;
- g++ versão 5.4.0;
- GNU Make versão 4.1.

Bibliotecas externas utilizadas:

- GNU Multi Precision versão 6.1.2. Para o correto funcionamento no Ubuntu, instalar os seguintes pacotes com o apt-get:
    - sudo apt install libgmp10 libgmp3-dev libgmpxx4ldbl.

Compilação e execução do EML RSA

O EML RSA conta com um Makefile para compilação, retirado do repositório https://gitlab.com/monaco/posixeg/ e modificado de acordo com as necessidade do nosso projeto.

Antes de executar o Makefile, certifique-se que existe a pasta ‘obj’ na raiz do projeto. Caso ela já exista, execute o comando ‘make’. Será criado um arquivo executável chamado ‘main’. Para executá-lo, digite ‘./main’, passando os argumentos como descrito na especificação do trabalho. **Atente-se ao fato de que o argumento de escolha de algoritmo não deve ser passado**.

A seguir iremos demonstrar como realizar as etapas de instalação, compilação e execução necessárias do EML RSA.

Após baixar o arquivo zipado, será necessário realizar sua descompactação. Feito isso, podemos entrar na pasta. Abaixo temos uma foto exemplificando o processo:

```
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop$ unzip eml-rsa-master.zip
Archive:  eml-rsa-master.zip
08358a69e0b87d53db0fa9608d91561adb8316b6
   creating: eml-rsa-master/
  inflating: eml-rsa-master/.gitignore
  inflating: eml-rsa-master/LICENSE
  inflating: eml-rsa-master/Makefile
   creating: eml-rsa-master/include/
  inflating: eml-rsa-master/include/eml-rsa.h
   creating: eml-rsa-master/src/
  inflating: eml-rsa-master/src/eml-rsa.cpp
  inflating: eml-rsa-master/src/main.cpp
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop$ cd eml-rsa-master/
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$
```

Figura 12: Descompactação e mudança de pasta.

Estando no diretório correto, iremos criar a pasta obj e logo em seguida compilar o programa com o comando make:

```
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$ mkdir obj
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$ make
Makefile:49: obj/main.d: No such file or directory
Makefile:49: obj/eml-rsa.d: No such file or directory
g++ -I./include -std=c++11  -c src/eml-rsa.cpp  -MM -MT 'src/eml-rsa.o obj/eml-rsa.d' src/eml-rsa.cpp > obj/eml-rsa.d
g++ -I./include -std=c++11  -c src/main.cpp  -MM -MT 'src/main.o obj/main.d' src/main.cpp > obj/main.d
g++ -I./include -std=c++11  -Wall  -c src/main.cpp -o obj/main.o
g++ -I./include -std=c++11  -Wall  -c src/eml-rsa.cpp -o obj/eml-rsa.o
g++ obj/main.o obj/eml-rsa.o -lgmpxx -lgmp  -o main
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$
```

Figura 13: Criação da pasta obj e compilação do algoritmo.

Com a compilação pronta podemos gerar as chaves pública e privada do algoritmo, como será demonstrado a seguir, onde key é o nome passado para gerar os arquivos da chave (key.prv e key.pub) e 100 é a semente (não obrigatória).

```
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$ ./main K key 100
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$
```

Figura 14: Geração das chaves pública e privada.

Geradas as chaves, iremos criptografar o arquivo image_c.png com os seguintes comandos:

```
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$ ./main C key.pub image_c.png image_d.png
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$
```

Figura 15: Criptografia de uma imagem com a chave pública gerada.

A seguir mostramos a imagem original (image_c.png) e a compactada(image_d.png). Note que não é possível visualizar o conteúdo da imagem compactada:

Figura 16: Imagem original e criptografada.

Por fim realizaremos sua descompactação, finalizando o processo:

```
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$ ./main D key.prv image_d.png image_c2.png
sugi@sugi-Lenovo-YOGA-510-14ISK:~/Desktop/eml-rsa-master$
```

Figura 17: Descriptografando a imagem com a chave privada gerada.

Figura 17: Imagem descriptografada.

Tipos de arquivos permitidos

O EML RSA trata o arquivo de entrada como uma sequência de bytes sem distinção de tipo, o que permite a criptografia de qualquer tipo de arquivo. Foram testados arquivos textuais com codificação UTF8, além de arquivos MP3 e JPEG.

Chaves

A escolha do tamanho da chave levou em consideração os tamanhos atualmente utilizados em aplicações reais. O tamanho padrão da chave do EML RSA é 1024 bits. As chaves são compostas por 3 números, sendo os dois primeiros vindos do algoritmo clássico do RSA - par (n, e) para a chave pública e par (n, d) para a chave privada - e o último (n1), igual para as duas chaves, é um número primo utilizado para a criação de uma sub chave e para a realização da Cifra de César.

Pseudo código do EML RSA

O algoritmo utilizado tem como base o RSA original complementado por operações XOR, cifra de César e permutação baseada no algoritmo RC4. Abaixo explicamos o seu funcionamento de criptografia e descriptografia:

Cifragem

1. for i in [0, message.size()[do S[i] = i end
2. Com j = 0, faça:
   for i in [0, message.size()[do
   ```
   j = (j + S[i] + key[i % key.size()]) % message.size()
   swap (S[i], S[j])
   ```
   end
3. for i in [0, message.size()[do swap (message[i], message[S[i]]) end
4. Pega-se 64 bytes da mensagem, transformando-a em um grande número positivo, e separa cada símbolo transformado em número com um delimitador.
5. Cria-se um vetor *subkeys* com duas sub chaves, sendo a primeira os 8 primeiros dígitos de *n* e a segunda os 8 primeiros dígitos de *n1*.
6. Sendo *chunks* um vetor dos blocos gerados em (4), faça:
   for each block in chunks do
   ```
   cypher = block^e mod n
   cypher = cypher xor subkeys[0]
   cypher = cypher xor subkeys[1]
   cypher = cypher + n1
   cypher = cypher xor subkeys[0]
   cypher = cypher xor subkeys[1]
   cypher = cypher + “ ”
   ```
   end

Na etapa 6, adicionamos espaços entre os blocos criptografados para conseguir capturar os blocos para decifragem.

Decifragem

O processo de decifragem é somente o inverso do de cifragem.

1. Sendo o arquivo criptografado separado por espaços, faça:
   for each c in file do
   ```
   m = c xor subkeys[1]
   m = m xor subkeys[0]
   m = m - n1
   m = m xor subkeys[1]
   m = m xor subkeys[0]
   m = m^d mod n
   m = separate_tokens(m, delimiter)
   ```
   end

2. for i in [0, message.size()[do S[i] = i end
3. Com j = 0, faça:
   for i in [0, message.size()[do
   j = (j + S[i] + key[i % key.size()]) % message.size()
   swap (S[i], S[j])
   end
4. for i = message.size()-1; i >= 0; i++ do swap (message[i], message[S[i]]) end

Realizadas tais explicações, é necessário esclarecer porque foram utilizadas as modificações propostas no algoritmo RSA.

A implementação do XOR e Cifra de César foram usadas por serem bem simples de codificar, além do processo de descriptografia ser fácil. Ademais, tais operações já conseguem dificultar um pouco que haja ataques de criptoanálise em cima do algoritmo.

A operação de permutação, que foi inspirada pelo RC4, foi implementada pois, como visto neste documento, não existem ataques efetivos contra tal algoritmo de fluxo. Portanto, para que houvesse uma maior segurança, foi decidido adicionar ao RSA o código que se baseia em RC4.

Tempo de execução

O RSA é um algoritmo relativamente lento e, por isso, não é geralmente utilizado para criptografar dados de usuários, mas sim chaves criptográficas para transmissão por meios inseguros. Tendo em mente isto, fizemos uma análise de seu tempo de execução com arquivos de tamanho variando de 1B a aproximadamente 8MB. O resultado pode-se ver no gráfico abaixo.

Apesar de ser um poderoso algoritmo de criptografia, ele demora em torno de 220s (ou 3min40s) para criptografar um arquivo de aproximadamente 8MB. Porém, como o uso dele normalmente é para criptografar chaves, há um bom desempenho.

Para comparar o tempo que demoraria para criptografar uma chave criptográfica, criamos uma chave usando o GPG, que utiliza RSA de 3072 bits. O tamanho em disco desta chave é de aproximadamente 2000B. Nosso algoritmo demoraria, portanto, em torno de 0.05s para criptografá-la, sendo portanto viável o seu uso.

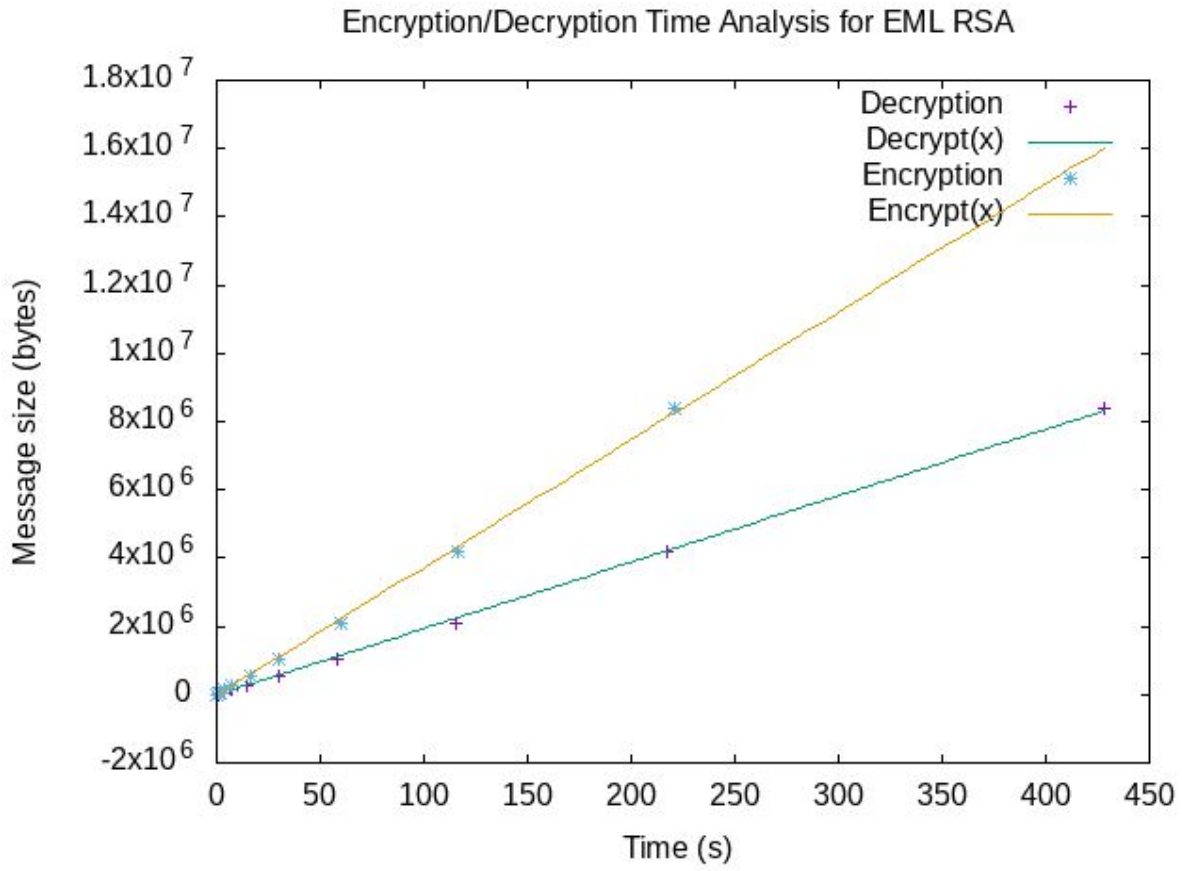

Gráfico 1: relação entre tamanho da chave (eixo Y) e tempo para criptografar/descriptografar utilizando o algoritmo EML RSA.

Conclusão

A criptografia não possui poder para resolver todos os problemas relacionados à segurança e certamente não é 100% segura, mas sua utilização é imprescindível visto que é necessário proteger as informações produzidas.

Desse modo, o estudo apresentado neste trabalho, assim como o algoritmo EML RSA, vêm com o intuito de estimular a utilização da criptografia nos softwares produzidos atualmente.