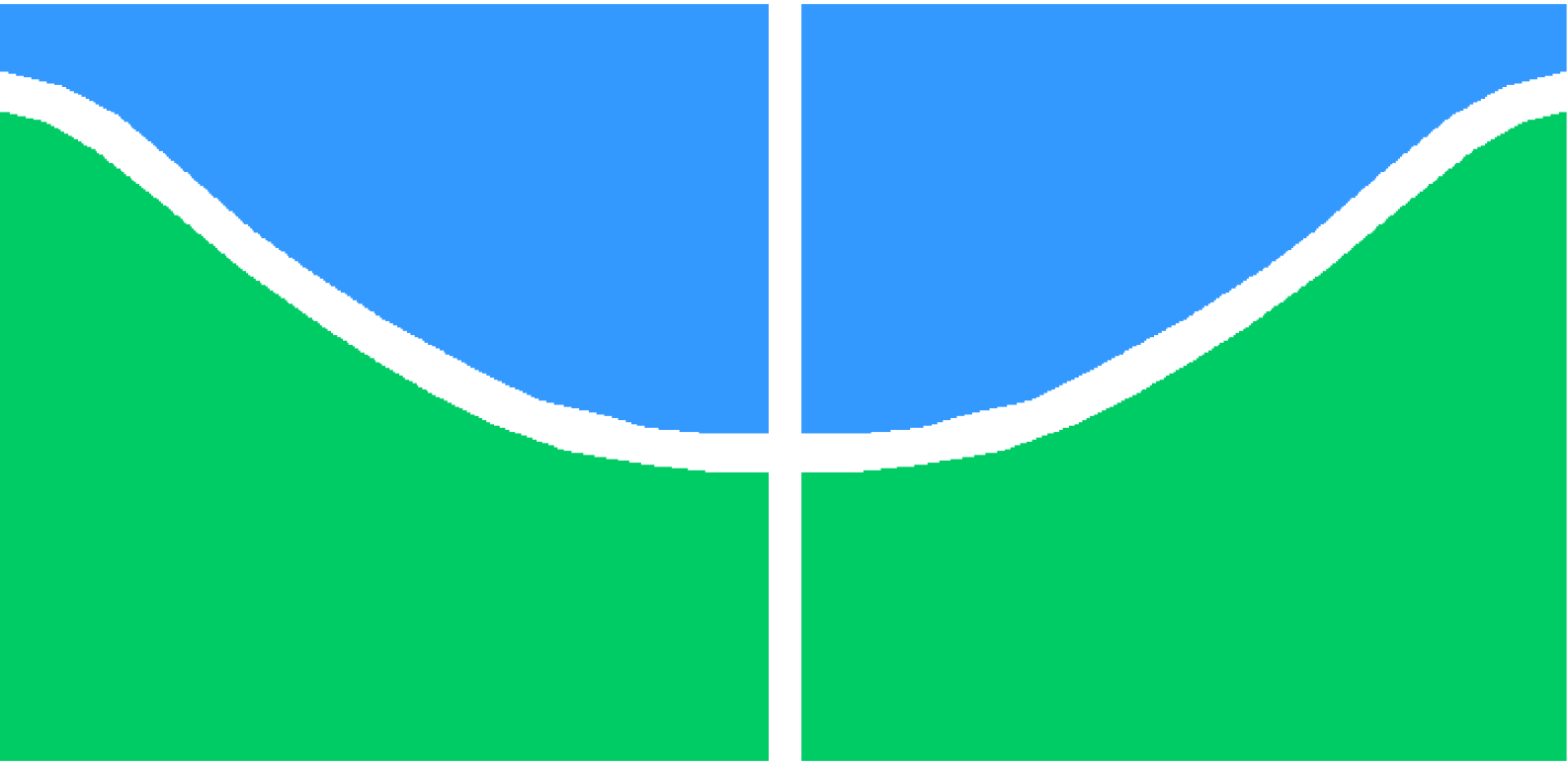

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo

Autor: Gabriela Matias Navarro
Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF
2014

Gabriela Matias Navarro

Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software .

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF

2014

Gabriela Matias Navarro

Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo/ Gabriela Matias Navarro. – Brasília, DF, 2014-

98 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. balanceamento. 2. cifra de fluxo. I. Prof. Dr. Luiz Augusto Fontes Laranjeira. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo

CDU 02:141:005.6

Gabriela Matias Navarro

Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software .

Trabalho aprovado. Brasília, DF, 25 de novembro de 2014:

Prof. Dr. Luiz Augusto Fontes Laranjeira
Orientador

Prof. Dr. Fabricio Ataides Braz
Convidado 1

Prof. Dr. Edson Alves da Costa Júnior
Convidado 2

Brasília, DF
2014

Resumo

A criptografia tem duas formas principais para cifrar um texto utilizando chave simétrica e chave assimétrica. Os algoritmos que utilizam chave simétrica são divididos em algoritmos de cifra de bloco e cifra de fluxo. Uma falha muito explorada por atacantes que desejam quebrar um texto cifrado é da análise de frequência de ocorrência dos caracteres do mesmo, pois a frequência média de ocorrência é conhecida para cada língua e muitos dos algoritmos não se preocupam com o balanceamento dessa frequência enquanto cifrando o texto. Este trabalho de conclusão de curso apresenta uma proposta de algoritmo que irá realizar o balanceamento completo da frequência de caracteres, aumentando sua segurança contra atacantes e curiosos.

Palavras-chaves: criptografia. cifra de fluxo. balanceamento.

Abstract

The cryptography has two ways to encrypt a text using symmetric key and asymmetric key. The algorithm using symmetric key algorithms are divided into block cipher and stream cipher. A failure much exploited by attackers who wish to break a cipher text is the analysis of frequency of occurrence of the same characters, because the medium frequency of occurrence is known for each language and many of the algorithms do not worry about balancing this frequency while encrypting the text. This course conclusion work proposes a algorithm that will perform the complete balancing of the frequency of characters, increasing their security against attackers and curious.

Key-words: crytography. stream cipher. balancing.

Lista de ilustrações

Figura 1 – Máquina Enigma 23
Figura 2 – Máquina SZ40 24
Figura 3 – Funcionamento básico da criptografia simétrica 25
Figura 4 – Modo Electronic Codebook 25
Figura 5 – Modo Cipher Block Chaining 26
Figura 6 – Modo Cipher Feedback 26
Figura 7 – Modo Output Feedback 27
Figura 8 – Modo Counter 27
Figura 9 – Cifra *DES* 28
Figura 10 – Cifra*AES* 29
Figura 11 – Funcionamento da cifra de fluxo 31
Figura 12 – Estrutura do algoritmo A5/1 32
Figura 13 – Estrutura do algoritmo A5/2 34
Figura 14 – Estrutura do algoritmo E0(FLUHRER; LUCKS, 2001) 35
Figura 15 – Tabela de *byte* para uso do algoritmo 50
Figura 16 – Esquema do algoritmo de cifração 50
Figura 17 – Esquema do algoritmo de decifração 52
Figura 18 – Arquitetura utilizada no projeto 55
Figura 19 – Diagrama de Sequência 59

Lista de tabelas

Tabela 1 – *Bits* importantes do algoritmo A5/1 32
Tabela 2 – *Bits* importantes do algoritmo A5/2 33
Tabela 3 – Tempo gasto para encontrar chaves 41
Tabela 4 – Frequência média de letras em um texto em inglês 41
Tabela 5 – Parâmetros para sequência com periodicidade 4 44
Tabela 6 – Parâmetros substituindo o valor de **m** 45
Tabela 7 – Tempos obtidos no experimento 1 62
Tabela 8 – Tempos obtidos no experimento 2 63
Tabela 9 – Tempos obtidos no experimento 3 63
Tabela 10 – Experimento tempo total enviando 2 *bytes* 64
Tabela 11 – Tempo enviando sinal de fim do arquivo '000' 64
Tabela 12 – Quantidade de conflitos e tempo médio de resolução. 65

Lista de abreviaturas e siglas

DES	Data Encryption Standard.
IBM	International Business Machines.
NIST	National Institute of Standards and Technology
AES	Advanced Encryption Standard
ECB	Electronic codebook
CBC	Cipher-block chaining
CFB	Cipher feedback
OFB	Output Feedback
CTR	Counter
SSL	Secure Sockets Layer
GSM	Global System for Mobile
LFSR	Linear feedback shift register
TLS	Transport Layer Security
WEP	Wired Equivalent Privacy
RSA	Algoritmo criptográfico de chave assimétrica
MIT	Massachusetts Institute of Technology
TRNG	True random number generator
PRNG	Pseudorandom number generator
RC4	Rivest Cipher 4

Lista de símbolos

$\oplus$	Símbolo lógico ou-exclusivo
ϕ	Função Totiente de Euler

Sumário

1 INTRODUÇÃO 21
1.1 Problema 21
1.2 Objetivos do Trabalho 21
1.2.1 Objetivo Geral 21
1.2.2 Objetivos Específicos 21
1.3 Organização do Trabalho 22

2 CRIPTOGRAFIA 23
2.1 Criptografia de Chave Simétrica 24
2.1.1 Cifra de Bloco 24
2.1.1.1 Algoritmo DES 27
2.1.1.2 Algoritmo AES 29
2.1.2 Cifra de Fluxo 30
2.1.2.1 Algoritmo A5/1 31
2.1.2.2 Algoritmo A5/2 33
2.1.2.3 Algoritmo E0 35
2.1.2.4 Algoritmo RC4 36
2.2 Criptografia Assimétrica 38
2.2.1 Algoritmo RSA 39
2.3 Criptoanálise 40
2.3.1 Força Bruta 40
2.3.2 Análise de Frequência 41

3 GERADOR DE NÚMEROS PSEUDO-ALEATÓRIOS 43
3.1 Geradores 44
3.1.1 Gerador Linear Congruente 44
3.1.2 Gerador Blum Blum Shub 45
3.1.3 Gerador Blum Micali 46
3.1.3.1 Exemplo 47
3.2 Segurança comprovada em geradores de números pseudo-aleatórios 47
3.2.1 Segurança do Blum Blum Shub 47
3.2.2 Segurança do Blum Micali 47
3.2.3 Comparação do Blum Blum Shub e Blum Micali 48

4 PROPOSTA DE ALGORITMO 49
4.1 Motivação 49

4.2 Funcionamento . . . 49
4.3 Vantagens . . . 54

5 IMPLEMENTAÇÃO . . . 55
5.1 Arquitetura . . . 55
5.1.1 Inicializador do Gerador . . . 56
5.1.2 Gerador de *Bytes* . . . 56
5.1.3 Fila de Mensagem . . . 56
5.1.4 *Thread* de Envio para Decifrador . . . 56
5.1.5 *Thread* para Receber do Cifrador . . . 56
5.1.6 *TCP/IP* . . . 56
5.1.7 Decifrador . . . 57
5.1.8 Cifrador . . . 57
5.2 Diagrama de Sequência . . . 57

6 RESULTADOS . . . 61
6.1 Ambiente Utilizado para Experimentos . . . 61
6.2 Tempos de Execução . . . 62
6.2.1 Experimento 1 . . . 62
6.2.2 Experimento 2 . . . 62
6.2.3 Experimento 3 . . . 63
6.2.3.1 Tempo Total de Execução . . . 64
6.2.4 Resolução de Conflitos . . . 65

7 CONCLUSÃO . . . 67
7.1 Trabalhos Futuros . . . 68

Referências . . . 69

APÊNDICES 71

APÊNDICE A – CÓDIGOS . . . 73
A.1 Bibliotecas . . . 73
A.1.1 Conexão . . . 73
A.1.2 Funções . . . 74
A.2 Conexão e Funções . . . 78
A.2.1 Conexão . . . 78
A.2.2 Funções . . . 81
A.3 Programas Principais . . . 90
A.3.1 Cifrador Algoritmo Proposto . . . 90

A.3.2 Decifrador Algoritmo Proposto 93
A.3.3 Cifrador *RC4* 95
A.3.4 Decifrador *RC4* 97

1 Introdução

Desde tempos remotos, segredos sempre existiram e métodos para deixá-los indecifráveis foram construídos. Enquanto algumas pessoas estavam escondendo informações, outras curiosas estavam tentando descobri-los. (MAXIMOV, 2006)

Na constante luta entre manter e quebrar segredos, métodos foram desenvolvidos para se esconder informações e falhas nesses métodos foram exploradas para se obter as informações.

Um dos métodos para obter informações sem a autorização necessária é a análise de frequência da ocorrência de caracteres dessas informações escondidas. Há estudos que indicam qual a frequência média de utilização de cada letra em um texto. Analisando um texto cifrado e calculando a frequência de ocorrência dos caracteres do mesmo, pode ser possível, através da substituição dos mesmos pelos caracteres com frequência similar da língua, quebrar a cifragem utilizada e ter acesso à mensagem original.

1.1 Problema

Como há a possibilidade de se obter as informações simplesmente analisando a frequência de ocorrência de caracteres, isso torna a comunicação insegura. Como consequência, o principal objetivo de se utilizar o algoritmo criptográfico, que é proteger informações importantes, torna-se obsoleto.

1.2 Objetivos do Trabalho

1.2.1 Objetivo Geral

Esse trabalho visa apresentar um algoritmo de criptografia que dificulte a análise de frequência de um texto permitindo assim a comunicação privada e segura.

1.2.2 Objetivos Específicos

- Identificar e estudar as principais formas de criptografia.
- Identificar e estudar as principais formas de criptoanálise.
- Identificar e estudar geradores de números pseudo-aleatórios.
- Realizar a proposta de algoritmo.

- Implementar o algoritmo proposto.
- Realizar experimentos afim de compará-los com resultados de outros algoritmos conhecidos.

1.3 Organização do Trabalho

Este trabalho tem a seguinte organização:

No capítulo 2 são apresentados os principais tipos de algoritmos criptográficos, incluindo suas descrições e exemplos, assim como são descritos métodos de criptoanálise.

No capitulo 3 é descrito o que é um gerador de números pseudo-aleatórios, alguns exemplos e suas implementações e é explicada a importância desses geradores para a criptografia.

No capitulo 4 é explicada a proposta do algoritmo com suas soluções, seu funcionamento e as principais vantagens em relação a outros algoritmos.

No capítulo 5 é explicado como foi feito a implementação com a arquitetura que foi utilizada e os passos para utilizar o algoritmo.

No capítulo 6 é explicado os experimentos realizados e como foi feita a coleta de dados desses experimentos.

No capitulo 7 é apresentada a conclusão deste trabalho, a descrição dos trabalhos futuros e suas importâncias.

2 Criptografia

A criptografia é a ciência dos códigos secretos, permitindo a confidencialidade de uma comunicação que utiliza um meio inseguro. (VAUDENAY, 2006). A necessidade de manter informações em sigilo impulsionou a evolução dos estudos da criptografia. O exato início da criptografia é incerto, mas na Renascença, assim como outros em muitos outros campos, o estudo da criptografia começou a ser aprofundado e suas técnicas armazenadas e ensinadas. (DAVIES, 1997)

Uma das primeiras formas de criptografia utilizada foi a substituição de caracteres. A substituição primeiramente era feita de forma fixa, ou seja, era usada uma única tabela em que as letras do alfabeto eram trocadas por outras letras. Leon Battista Alberti introduziu uma maneira diferente de fazer a substituição, criando o algoritmo *polyalphabetic substitution*. O princípio desse algoritmo era a criação de várias tabelas de substituição e a utilização de uma chave para definir a ordem que cada tabela seria usada para cifrar cada letra do texto em claro. O conjunto de tabelas de substituição é chamado de *Vigenere tableau*.

Durante a II Guerra Mundial, informações relacionadas às estratégias de guerra deveriam ser mantidas em sigilo. Para isso duas máquinas foram criadas e merecem destaque: a *Enigma* e a *Lorenz SZ40*. A máquna *Enigma* foi baseada nas capacidades criptográficas de uma série de motores conectados por fio e *plugboard*. (WILCOX, 2006) O *plugboard* fazia a substituição de uma letra por outra letra.

Figura 1 – Máquina Enigma[1]

A máquina *Lorenz* foi um impressor anexado que automaticamente cifrava o texto em claro digitado pelo operador e enviava o texto cifrado por um fio de comunicação. No

[1] Disponível e adaptado de: <http://www.bbc.co.uk/history/topics/enigma>

outro lado da comunicação, tinha uma máquina *Lorenz* idêntica que fazia a decifração automaticamente e imprimia o texto em claro no papel. (COLLINS, 2006)

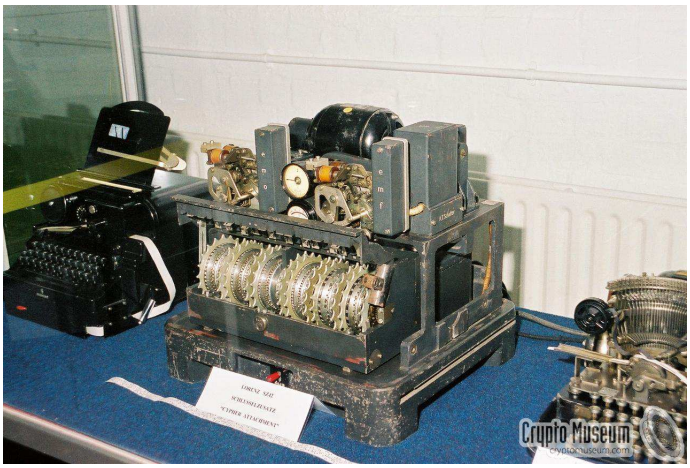

Figura 2 – Máquina SZ40[2]

Após a guerra, a publicação do algoritmo *DES* serviu de incentivo para cientistas do mundo todo aprofundarem suas pesquisas em novos algoritmos. Nos dias de hoje existem duas linhas de estudo sobre a criptografia, a criptografia com chave simétrica e a com chave assimétrica.

2.1 Criptografia de Chave Simétrica

A criptografia simétrica utiliza a mesma chave[3] para cifrar[4] e decifrar[5] uma mensagem. É objeto de estudo para muitos cientistas e existem muitos algoritmos que ainda são utilizados pelo mundo. Sua vantagem é que o custo para ser utilizada não é tão elevado e sua possibilidade de uso é maior. Sua desvantagem é que a forma que é feita a troca de chaves não é segura.

Os algoritmos são divididos em: cifra de bloco e cifra de fluxo.

2.1.1 Cifra de Bloco

A cifra de bloco tem como princípio a cifra de blocos de caracteres com tamanho definido e a mensagem criptografada deve ter o mesmo tamanho da mensagem em claro. Um algoritmo muito conhecido é o *DES* que foi desenvolvido pela *IBM*. Foi a primeira cifra

[2] Disponível e adaptado de: <http://www.cryptomuseum.com/crypto/lorenz/sz40/>
[3] Entrada para as funções cifrar e decifrar, é usada para que duas pessoas consigam se comunicar em segredo.
[4] Função que tem como objetivo transformar uma mensagem que deve estar em sigilo e transformá-la em algo ininteligível
[5] Função que transforma uma mensagem ininteligível para a mensagem original
[6] Disponível e adaptado de: <http://www.codeproject.com/Articles/21076/Securing-Data-in-NET>

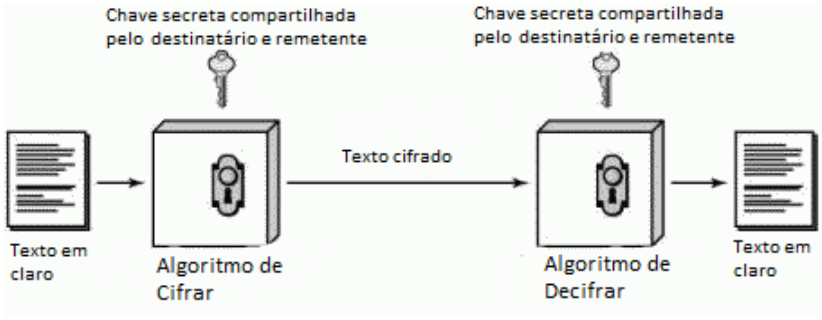

Figura 3 – Funcionamento básico da criptografia simétrica[6]

comercialmente desenvolvida e sua estrutura foi divulgada por completo. (BIRYUKOV,).

O tamanho da chave utilizada no *DES* é considerado um problema nos dias atuais. Muitos pesquisadores sabendo desse problema tentaram remediá-lo com novas variantes criptográficas. Um exemplo é o *3-DES* que utiliza duas chaves no processo, mas que, como consequência, tem seu desempenho prejudicado.

Outra medida tomada para contornar o problema de ataques ao *DES* foi a publicação de um concurso do *NIST* para a escolha de um novo padrão criptográfico. Assim surgiu o *AES*, que é um algoritmo que utiliza o sistema de permutação e substituição, tem o tamanho do bloco fixo de 128 bits e tamanho de chave variável entre 128, 192 ou 256 bits.

Existem operações que podem ser aplicadas nas cifras de blocos e que podem ser utilizadas nos algoritmos *DES* e *AES* para a produção de mensagens criptografadas mais seguras, são elas:

ECB a mensagem é dividida em blocos e consiste em cifrar os blocos de forma independente um do outro e com uma chave fixa. A desvantagem desse modo de cifra é que blocos de texto em claro iguais produzem blocos de texto cifrado iguais.

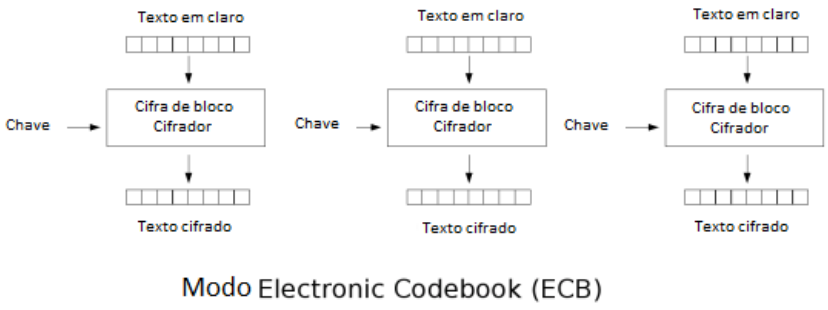

Figura 4 – Modo *Electronic Codebook*[7]

[7] Disponível e adaptado de: <http://cryptodox.com/Block_cipher_modes_of_operation>

CBC a mensagem é dividida em blocos e o modo de cifra utiliza da operação de ou-exclusivo entre um vetor de inicialização (para o primeiro bloco) e o texto em claro[8] e por fim um ou-exclusivo do resultado com a chave. A partir do segundo bloco, ao invés de um vetor de inicialização, é utilizado o bloco criptografado anterior.

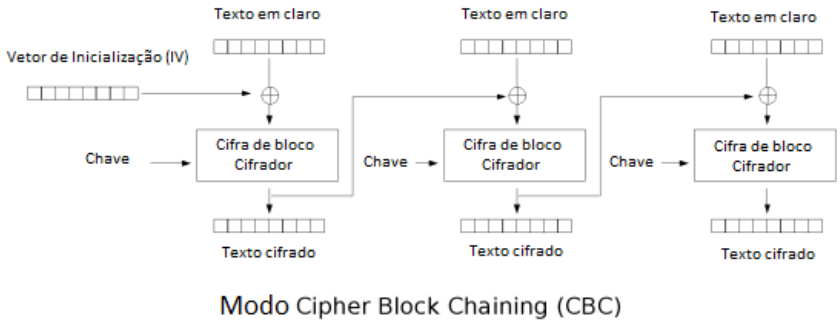

Figura 5 – Modo *Cipher Block Chaining*[9]

CFB a mensagem também é dividida em blocos e esse modo de cifra é muito similar ao CBC, porém é feito um ou-exclusivo da chave com o vetor de inicialização, o bloco criptografado anterior e com o resultado é feito um ou-exclusivo com a mensagem em claro.

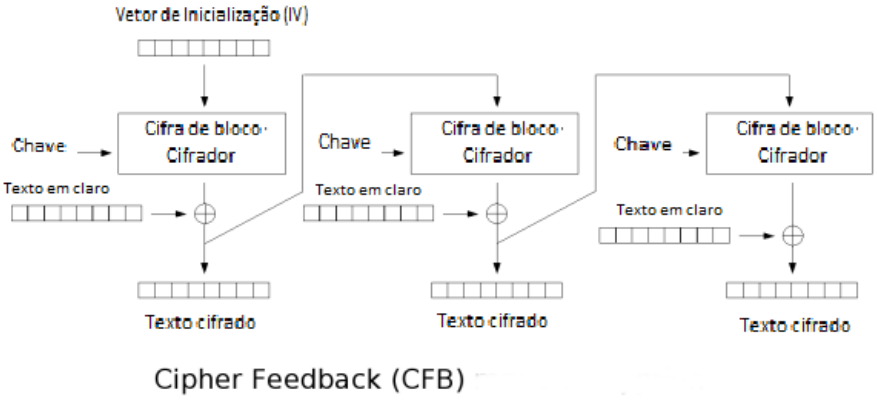

Figura 6 – Modo *Cipher Feedback* [10]

OFB a diferença em relação ao CFB é que o resultado entre o vetor de inicialização e a chave ou resultado da operação anterior e a chave, serve de entrada para o próximo bloco, ao invés do bloco criptografado.

[8] Mensagem que deve ser mantida em sigilo e que se deseja cifrar.

[9] Disponível e adaptado de:<https://www.adayinthelifeof.nl/2010/12/08/encryption-operating-modes-ecb-vs-cbc/>

[10] Disponível e adaptado de: <http://crypto.stackexchange.com/questions/2476/cipher-feedback-mode>

[11] Disponível e adaptado de: <http://commons.wikimedia.org/wiki/File:Ofb_decryption.png>

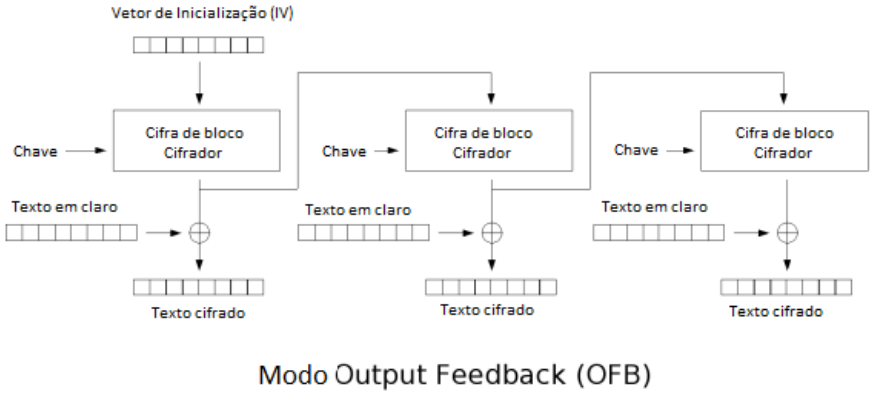

Figura 7 – Modo *Output Feedback*[11]

CTR utiliza como entrada para a função criptográfica um contador que é acrescentado de um em um.

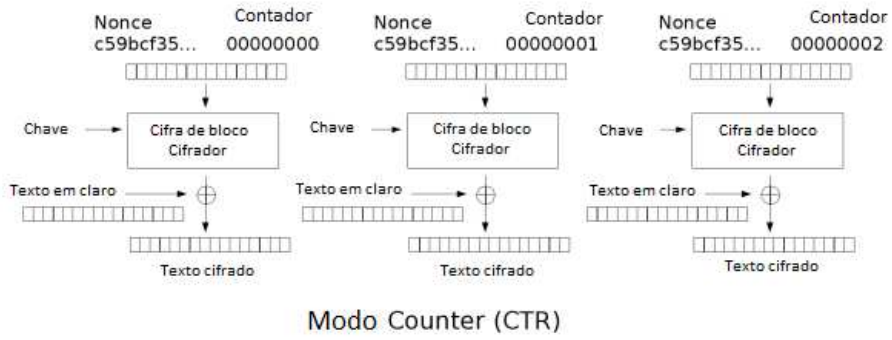

Figura 8 – Modo *Counter*[12]

2.1.1.1 Algoritmo DES

Como dito anteriormente, o DES foi o primeiro algoritmo que foi publicado e utilizado de forma comercial. Esse algoritmo utiliza o princípio da cifra de *feistel*[13] com 16 *rounds* de processamento.

Esse algoritmo utiliza chave de 56 *bits* e opera em blocos de dados de 64 *bits*. A fase de inicialização é feita uma permutação do bloco de dados iniciais. O outro passo da fase de inicialização é dividir o bloco de dados em blocos de 32 *bits*.

Depois dessa divisão, as operações são executadas no bloco que fica a direita como mostrado na figura 9. Esse algoritmo introduz dois atributos: confusão e difusão. O atributo de confusão é feito por uma camada de substituição *S-box*. A camada de permutação

[12] Disponível e adaptado de: <http://crypto.stackexchange.com/questions/8151/counter-mode-static-iv-but-different-keys>

[13] Estrutura de criptografia simétrica que inclui dividir o bloco de dados e depois realizar operações em um deles e ao final realizar um ou-exclusivo nos dois blocos.

[14] Disponível e adaptado de: <http://www.eventid.net/show-DocId-19.htm>

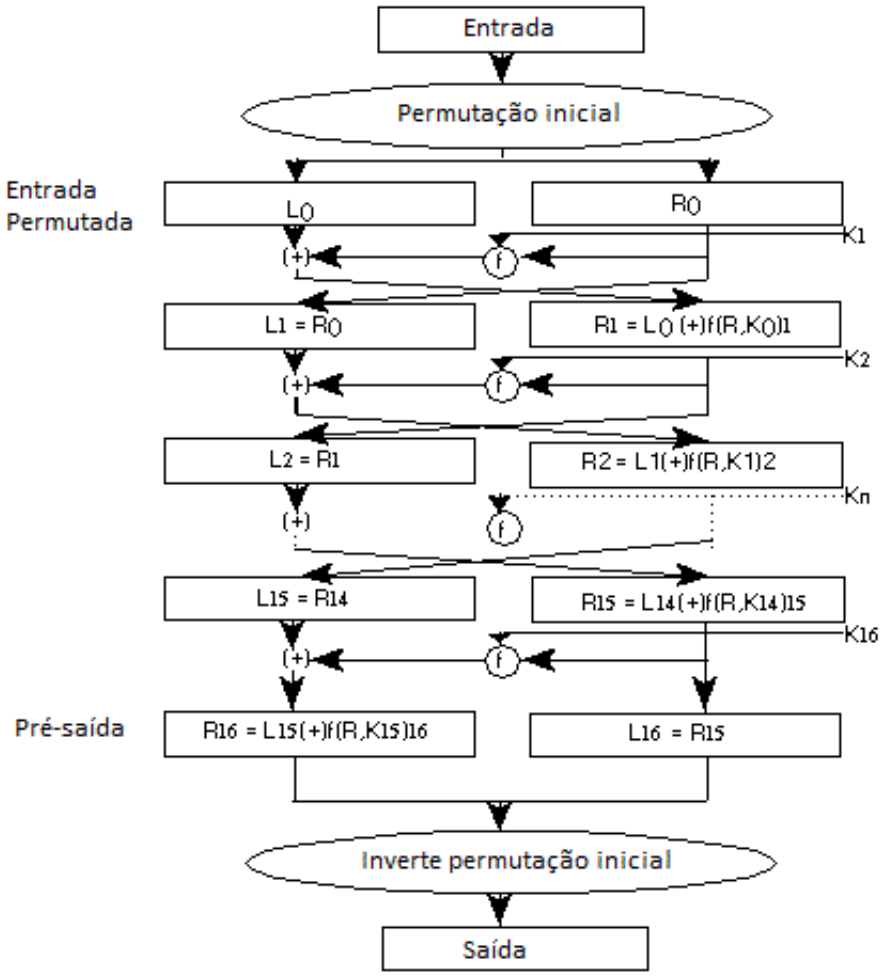

Figura 9 – Cifra DES[14]

é feita pela *p-box* e é responsável pelo atributo de difusão na cifra. O resultado dessas duas operações no bloco do lado direito é feito um ou-exclusivo com o lado esquerdo e isso resulta no lado direito do próximo *round*. Essas operações se repetem em todos os 16 *rounds*. No último passo é realizada a inversão da permutação inicial. No pseudo-código 2.1 pode ser demonstrado os passos acima descritos.

Código 2.1 Pseudo-código DES

```
1 void desCipher(char plainBlock[64], char roundKeys[56], char
    cipherBlock[64]){
2
3   char tempBlock[64] = permutation(plainBlock, initialPermutation
      )
4   char leftBlock[32] = split(0,32,tempBlock)
5   char rightBlock[32] = split(32,64,tempBlock)
6   for(i = 0; i < 16; i++){
7     sbox = sbox(rightBlock)
```

```
8       pbox = pbox(sbox)
9       rightBlock = pbox ^ leftBlock
10    }
11
12    tempBlock = combination(leftBlock, rightBlock)
13    cipherBlock = permutation(tempBlock, finalPermutation)
14  }
```

2.1.1.2 Algoritmo AES

O algoritmo *AES* foi inventado com o intuito de dar mais segurança ao processo de criptografia e foi adotado como algoritmo padrão pelo *NIST*. Esse algoritmo, ao contrário do *DES*, não utiliza a cifra de *feistel* como base.

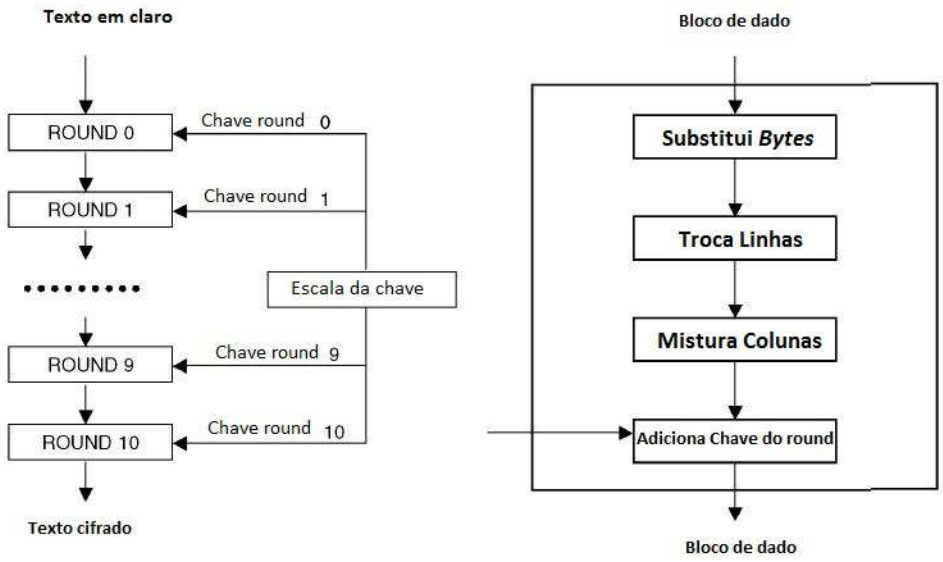

Figura 10 – Cifra AES[15]

O algoritmo *AES* utiliza chaves com tamanhos variáveis e essas dependem da quantidade de *rounds* que são utilizados. Exemplo é que para 10 *rounds* o tamanho da chave que deve ser utilizado é de 128 *bits*, para 12 *rounds* o tamanho da chave é de 192 *bits* e para 14 *rounds* o tamanho da chave correspondente é 256 *bits*. Isso é feito pois para cada *round*, um pedaço da chave é utilizado.

O tamanho do bloco que é utilizado é de 128 *bits* e é chamado de estado. O bloco de estado é dividido em uma matriz 4×4. A forma como cada *round* é feita pode ser vista

[15] Disponível e adaptado de: <http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/AES-L/help.html>

na figura 10. Todos os *rounds* são idênticos, com exceção do último *round*. Em cada *round* a operação de substituição com a função *S-box* é feita e essa operação introduz confusão a cifra. Outra operação que é realizada em cada *round* é a de permutação com a função de *p-box* e introduz difusão a cifra. A permutação é composta pela mistura de colunas e linhas da matriz que o bloco foi dividido. A diferença entre o último *round* e os anteriores é o passo de mistura das colunas, o documento de proposta de algoritmo explica o porque: Para que a cifra e o inverso da mesma sejam mais similares em estrutura, a camada de mistura de colunas do último *round* é diferente das camadas de permutações de *rounds* anteriores.(DAEMEN; RIJMEN, 1998)

Código 2.2 Pseudo-código AES

```
1  void aesCipher(char plainState[128], char roundKeys[rounds], char
       cipherState[128]){
2
3    char matrix[4][4] = transformInMatrix(plainState)
4    for(i = 0; i < rounds; i++){
5      char tempMatrix[4][4] = sBox(matrix)
6      tempMatrix = shiftRows(tempMatrix)
7      tempMatrix = mixColumns(tempMatrix)
8      matrix = AddKeys(tempMatrix, roundKeys[i])
9    }
10   tempMatrix = shiftRows(matrix)
11   cipherBlock = AddKeys(tempMatrix, roundKeys[round])
12 }
```

2.1.2 Cifra de Fluxo

Os algoritmos que utilizam a cifra de fluxo o fazem *bit* a *bit* ou *byte* a *byte*. Uma das vantagens da cifra de fluxo em relação a cifra de bloco é seu desempenho superior e, por esse motivo, é muito utilizado em sistemas de redes, tais como *bluetooth*, *SSL* e outros.

Outra vantagem é a possibilidade de transformar uma cifra de bloco em cifra de fluxo. Para isto basta definir que o tamanho do bloco seja um bit ou um byte. Alguns dos algoritmos de fluxo mais populares, que hoje cobrem mais de 80% do mundo na telecomunicação e *cyber space*, são: A5/1, A5/2, E0 e RC4. (BAKHTIARI; MAAREF, 2011)

A figura 11 representa o funcionamento da cifra de fluxo, que funciona a partir de um gerador de *keystream*,[16], que é gerado a partir de uma chave. Esse *keystream* é

[16] Sequência de caracteres aleatórios ou pseudo-aleatórios que são utilizados para a cifra de fluxo.

utilizado para cifrar os *bits* do texto em claro. Para decifrar, o mesmo *keystream* deve ser gerado para que possa fazer a operação ou-exclusivo com os *bits* do texto cifrado.

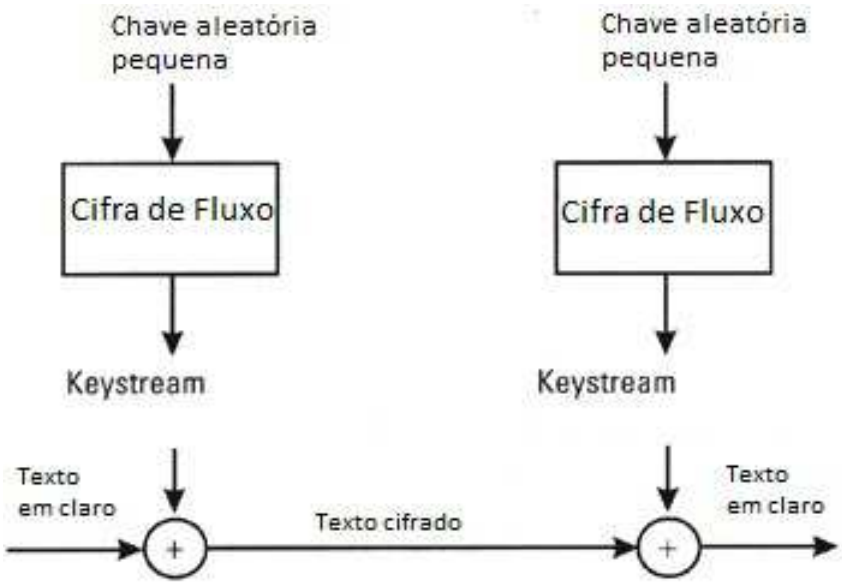

Figura 11 – Funcionamento da cifra de fluxo[17]

2.1.2.1 Algoritmo A5/1

Principal algoritmo usado no mundo para a comunicação *GSM*, foi desenvolvido em 1987 e teve seu funcionamento mantido em segredo e somente em 1999 foi revelado por engenharia reversa. Consiste de três *Linear Feedback Shift register* ou *LFSR* binários R1, R2 e R3. Um *LFSR* é um registrador que mantém valores de estado anterior e é atualizado por uma função que normalmente é um ou-exclusivo. Esses registradores são atualizados com cronômetro irregular, o que significa que a forma de atualização dos três registradores acontecem de forma independente.

Cada registrador contém *bits* importantes para a criptografia e suas posições são predeterminadas. São os *bits* do *clocking* e *tapping*, que estão definidos na tabela 1. Os registradores tem tamanhos diferentes sendo que, o R1 tem 19 *bits*, o R2 tem 22 *bits* e o R3 tem 23 *bits*. O algoritmo tem como entradas a chave secreta *Kc*, que tem 64 *bits*, e um vetor de inicialização de 22 bits composto do *frame number*. O *keystream*, de 228 *bits*, é a saída do processo, sendo que os primeiros 114 *bits* são o *keystream* de *downlink* e os outros 114 *bits* são o *keystream* de *uplink*. Cada ciclo no algoritmo é composto por passos específicos em cada registrador:

A operação de ou-exclusivo é realizada entre os *tapping bits* de cada registrador. Utilizando a regra de maioridade[19] com os *clocking* bits. O registrador é acionado se o

[17] Disponível e adaptado de: <http://www.globalspec.com/reference/81191/203279/2-6-stream-ciphers>

[18] Disponível e adaptado de: <http://cryptome.info/0001/a51-bsw/a51-bsw.htm>

[19] Essa regra funciona da seguinte maneira, os três *clocking bits* de cada registrador são comparados

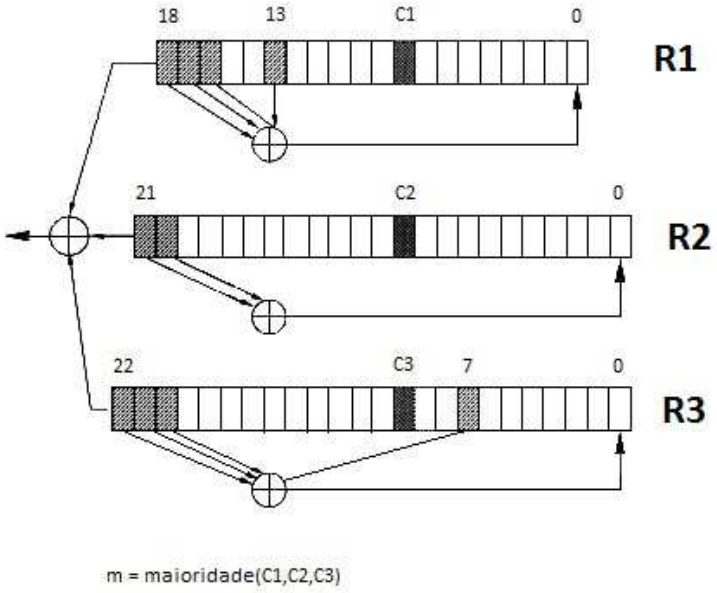

Figura 12 – Estrutura do algoritmo A5/1[18]

Registrador	*Clocking Bits*	*Tapping Bits*
R1	8	13, 16, 17, 18
R2	10	20,21
R3	10	7,20,21,22

Tabela 1 – *Bits* importantes do algoritmo A5/1

valor contido no *clocking bits* for igual ao resultado da regra de maioridade e então os *bits* do registrador são deslocado da direita para a posição mais a esquerda do registrador.

Código 2.3 Pseudo-código A5/1

```
1  void a51Generation(key key[64]){
2
3    int i;
4    for(i = 0; i < 64; i++){
5      R1[0] = R1[0] ^ key[i]
6      R2[0] = R2[0] ^ key[i]
7      R3[0] = R3[0] ^ key[i]
8    }
9
10   registers R1[19] ,R2[22], R[23]
11
```

e os registradores que tiverem os *bits* que teve maior ocorrência na comparação são acionados, por exemplo o R1 tem *bit* 1, o R2 tem *bit* 0 e o R3 tem *bit* 1, então o *bit* com maior ocorrência é o 1, sendo assim somente os registradores R1 e R3 são acionados naquele ciclo.

```
12   resultR1 = R1[13] ^ R1[16] ^ R1[17] ^ R1[18]
13   resultR2 = R2[20] ^ R2[21]
14   resultR3 = R3[7] ^ R3[20] ^ R3[21] ^ R3[22]
15
16   clockingBit = marjorityRule(R1[8], R2[10], R3[10])
17   if(R1[8] == clockingBit){
18     deslocateToTheLeft(resultR1)
19   }else if(R2[10] == clockingBit){
20     deslocateToTheLeft(resultR2)
21   }else if(R3[10] == clockingBit){
22     deslocateToTheLeft(resultR3)
23   }
24
25   output = R1[18] ^ R2[21] ^ R3[22]
26 }
```

Se o registrador for acionado, então o resultado do ou-exclusivo do primeiro passo deve ser passado para a posição zero do registrador correspondente.

Finalmente, a saída é definida pelo ou-exclusivo dos *bits* dos registradores nas posições: R1[18], R2[21] e R[22]. Para a fase de iniciação é utilizada a chave de 64 *bits* e passa de *bit* por *bit* fazendo ou-exclusivo da posição 0 de cada registrador com o *bit* da chave. O algoritmo ignora o resultado das primeiras 100 execuções e depois retorna os *bits* do *keystream* a cada 228 ciclos, gerando os 228 *bits* de saída.

2.1.2.2 Algoritmo A5/2

Esse algoritmo foi desenvolvido por motivos de restrições de exportação do A5/1 e tem uma estrutura similar ao A5/1, com algumas exceções como por exemplo: o A5/2 utiliza 4 registradores. Outra diferença é o fato que o *clocking* dos registradores R1, R2 e R3 são acionados baseados na regra de maioridade do R4. Ou seja cada registrador é acionado dependendo dos *bits* do registrador R4.

Registrador	*Clocking Bits*	*Tapping Bits*	*Bits* utilizados na função maioridade
R1	8	13, 16, 17, 18	12, 14*[20], 15
R2	10	20, 21	9, 13, 16*
R3	10	7, 20, 21, 22	13*, 16, 18
R4	3, 7, 10	-	-

Tabela 2 – *Bits* importantes do algoritmo A5/2

[20] *Bit* complementar

[21] Disponível e adaptado de: <http://blog.cryptographyengineering.com/2012/02/sattelite-phone-encryption-is-terrible.html>

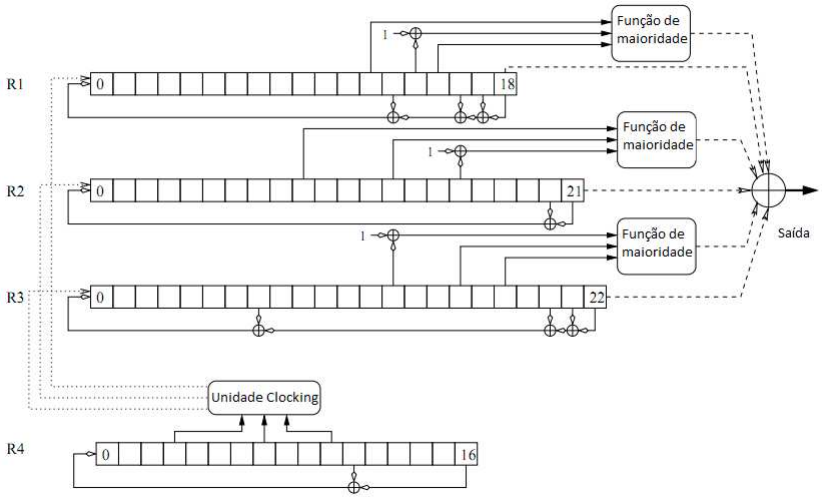

Figura 13 – Estrutura do algoritmo A5/2[21]

Caso o resultado da regra da maioridade dos *clocking bits* do R4 seja igual ao do *bit* que se encontra na posição 3 então o registrador R2 é acionado, caso seja igual ao *bit* que se encontra na posição 7 então o registrador R3 é acionado e se for igual ao *bit* que está na posição 10 então o registrador R1 é acionado. Depois que os registradores são acionados, então o registrador R4 é acionado também. A saída desse algoritmo é feita da forma demonstrada pelo pseudo-código 2.4

Código 2.4 Pseudo-código A5/2

```
1  void a52Cipher(key key[64]){
2
3    registers R1[18] ,R2[21], R3[22], R4[16]
4
5    clockingBit = marjorityRule(R4[3], R4[7], R4[10])
6
7    resultR1 = 0
8    resultR2 = 0
9    resultR3 = 0
10   if(clockingBit == R4[3]){
11     // R2
12     complement = 1 ^ R2[16]
13     resultR2 = marjorityRule(complement, R2[9], R2[13]) ^ R2[21]
14     deslocateToLeft(R2, R2[21] ^ R2[20])
15
16   } else if(clockingBit == R4[7]) {
17   // R3
18     complement = 1 ^ R3[13]
```

```
19      resultR3 = marjorityRule(complement, R3[16], R3[18]) ^ R3[22]
20      deslocateToLeft(R3, R3[22] ^ R3[21] ^R3[20])
21
22    } else {
23    //R1
24      complement = 1 ^ R1[14]
25      resultR1 = marjorityRule(complement, R1[12], R1[15]) ^ R1[18]
26      deslocateToLeft(R1.R1[18] ^ R1[17] ^R1[16] ^ R1[13])
27    }
28
29    deslocateToLeft(R4, R4[16] ^ R4[11])
30
31    output = resultR1 ^ resultR2 ^ resultR3
32 }
```

1. É calculado o complementar do *bit* da posição especificada de cada registrador.
2. Em cada registrador é utilizado a regra da maioridade para dois *bits* e o complementar do *bit* calculado anteriormente.
3. É feito um ou-exclusivo do resultado do item anterior com o *bit* mais significativo do registrador de cada registrador.

2.1.2.3 Algoritmo E0

Utilizado para proteger informações que utilizam a tecnologia *bluetooth*. O principal objetivo dessa tecnologia é conectar dois aparelhos para transferência de algum tipo de informação.

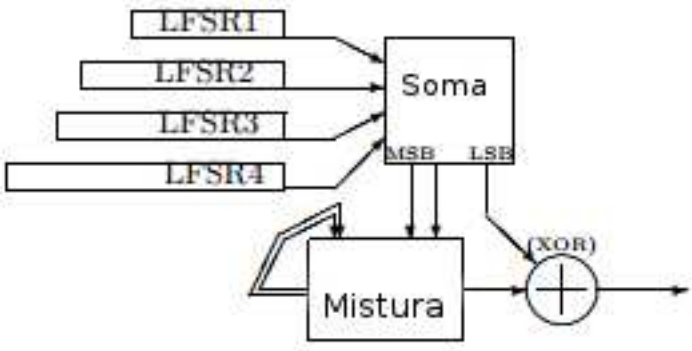

Figura 14 – Estrutura do algoritmo E0(FLUHRER; LUCKS, 2001)

O algoritmo também é baseado no uso de registradores e a saída é gerada em ou-exclusivo de *bits* de cada registrador. Nessa estrutura são utilizados 4 registradores com

tamanhos 25, 31, 33 e 39 *bits*. O sistema da geração do *keystream* é um pouco diferente, a cada *clock tick* todos os registradores são acionados e é feito um ou-exclusivo da saída de cada um. A operação é descrita a seguir:

Código 2.5 Pseudo-código E0

```
1  void e0Cipher(key key[64]){
2
3    registers R1[25] ,R2[31], R3[33], R4[39]
4
5
6    while(clockTick) {
7      result = sum(R1,R2, R3, R4)
8      char finiteStateMachine[3] = msb(result)
9      output = finiteStateMachine ^ lsb(result)
10   }
11 }
```

2.1.2.4 Algoritmo RC4

Projetado em 1987 por Ron Rivest, é muito utilizado em sistemas de redes como TLS/SSL e WEP por conta de sua simplicidade e alto desempenho. Tem tamanho de chave variável e se baseia em permutações aleatórias. Utiliza um gerador de números aleatórios e as permutações são realizadas sobre esses números aleatórios.

O funcionamento do algoritmo é realizado com uma leitura do texto em claro feita a cada *byte* e cada *byte* produz um *byte* de texto cifrado. Seu funcionamento está ligado a um gerador de *keystream*, ou seja, utilizando como entrada uma chave é criada uma sequência de números aleatórios.

O RC4 é dividido em três passos fundamentais: a inicialização, geração aleatória de *bytes* e cifração.

Inicialização RC4 aceita chaves de tamanho variáveis. Na fase de inicialização, o vetor S é inicializado com valores iniciais seguindo o código mostrado a seguir:

Código 2.6 Código Inicialização

```
1 typedef struct RC4{
2     int i;
3     int j;
4     char state[255];
```

```
5 }
6
7 void rc4Initializer(RC4 * params, char key[]){
8
9   int i,j = 0;
10  size_t keylen = strlen(key);
11  unsigned char temp[255];
12  for(i = 0; i <= 255; i++){
13    params->state[i] = i;
14    temp[i] = key[i % keylen];
15  }
16 }
```

Como pode ser visto no código anterior, um vetor temporário de 256 *bytes* é criado juntamente com a inicialização do vetor de 256 *bytes* S. O vetor temp é inicializado com os valores da chave K. Se a chave tem tamanho 256 *bytes*, então T é igual a K. Se K for menor que 256 *bytes*m então a chave é repetida até preencher as 256 posições de T.

Após a inicialização do vetor T, é feita uma permutação inicial utilizando variáveis de controle i e j e é realizada obedecendo o código abaixo:

Código 2.7 Código Permutação Inicial

```
1 void intialPermutation(RC4 * params, char key[]){
2
3   unsigned char swap;
4   for(i = 0; i <= 255; i++) {
5     j = (j + params->state[i] + temp[i]) % 256;
6     swap = params->state[j];
7     params->state[j] = params->state[i];
8     params->state[i] = swap;
9   }
10
11  params->i = 0;
12  params->j = 0;
13 }
```

Geração Aleatória de *bytes* Após a permutação inicial, a geração de bytes é feita utilizando de permutações aleatórias do vetor S. Essa geração utiliza das variáveis de controle i, j e k. Essa geração é realizada a cada cifração realizada pelo algoritmo. A geração é feita da seguinte forma:

Código 2.8 Código Geração Aleatório de *bytes*

```
1  char rc4Generator(RC4 * params){
2
3    unsigned char t;
4    unsigned char k;
5    unsigned char swap;
6    params->i = (params->i + 1) % 256;
7    params->j = (params->j + params->state[params->i]) % 256;
8    swap = params->state[params->j];
9    params->state[params->j] = params->state[params->i];
10   params->state[params->i] = swap;
11   t = (params->state[params->i] + params->state[params->j]) %
         256;
12   k = params->state[t];
13
14   return k;
15 }
```

Cifração Para a fase de cifração é preciso que se obtenha um *byte* da função de gerador aleatório. A cifração desse algoritmo é feita com a operação ou-exclusivo entre o *byte* proveniente do gerador e o *byte* correspondente do texto em claro.

Código 2.9 Código Cifração de *bytes*

```
1  char rc4Encryption(RC4 * params, char plainText){
2
3     byteToCipher = rc4Generator(params);
4     return plainText ^ byteToCipher;
5  }
```

2.2 Criptografia Assimétrica

A criptografia assimétrica é usada principalmente para cifrar dados pequenos. Um dos principais usos é a assinatura digital e para realizar a troca de chaves de forma segura. Esse tipo de criptografia está associada à criptografia de chave pública. Nela é estabelecido que um par de chaves (chave pública e privada) sejam usadas para a cifração e decifração da mensagem.

O funcionamento é simples. Bob tem suas duas chaves e disponibiliza sua chave pública de forma que qualquer pessoa que tenha interesse em lhe enviar uma mensa-

gem criptografada utilize sua chave pública. Quando Bob recebe a mensagem cifrada, é realizada a decifração dessa mensagem utilizando sua chave privada.

Alguns dos algoritmos mais conhecidos são o *Elgamal* e o *Diffie-Hellman*. O algoritmo *RSA* foi o primeiro algoritmo a cifrar e decifrar com sucesso utilizando o princípio da chave pública. O *Elgamal* tem sua força proveniente da dificuldade de se calcular um logaritmo discreto. Com o Elgamal pode-se realizar a assinatura digital de um documento. *Diffie-Hellman* foram os visionários da chave pública, primeiramente definiram a ideia e algum tempo depois definiram um algoritmo que serve apenas para trocar uma chave secreta entre duas pessoas.

2.2.1 Algoritmo RSA

Tem seu nome baseado nas iniciais dos cientistas do *MIT* que o inventaram: *Ron Rivest, Adi Shamir* e *Len Adleman*. A geração das chaves desse algoritmo é feita utilizando números primos e sua força está associada ao fato de que é fácil realizar a multiplicação de dois números primos grandes, mas é difícil a decomposição desse resultado. Portanto a chave é gerada da seguinte forma:

1. Dois números **p** e **q** são escolhidos de forma aleatória e são primos.
2. $\mathbf{p} \neq \mathbf{q}$
3. **n** é composto pela multiplicação de **p** e **q**.
4. **e**[22], tal que mdc[23](ϕ(n, e)) = 1
5. **d**[24], (e $^{(-1)}$) $\times$ (mod ϕ(n))

A chave pública **e** é usada para cifrar a mensagem e a chave privada **d** é usada para decifrar. O método de cifra é simples, o texto é tratado como um conjunto de inteiros e a cada inteiro é feito o seguinte cálculo:

Cifrar C = M ^{e} mod n. Sendo que M é o inteiro correspondente a parte do texto em claro e n o resultado da multiplicação dos dois números primos selecionados na confecção das chaves, C é o texto cifrado e **e** é a chave pública.

Decifrar M = C ^{d} mod n. Sendo que C é o texto cifrado, M o texto em claro e **d** é a chave privada.

[22] Chave pública, para cifrar um texto.
[23] Operação matemática que visa descobrir o maior divisor comum entre dois ou mais números
[24] Chave privada, para decifrar um texto

Código 2.10 Pseudo-código RSA

```
1  int phi(p, q){
2      return (p - 1) * (q - 1)
3  }
4
5  void rsaCipher(key key[64], char plainText){
6      int p = getBigPrime()
7      int q = getBigPrime()
8
9      while( p == q){
10         q = getBigPrime()
11     }
12
13     int n = p * q
14
15     e = mdc(phi(n, e)
16     while(e != 1){
17         e = mdc(phi(n, e))
18     }
19
20     d = pow(e, (-1)) % (n - 1)
21
22     c = pow(plainText, e) % n
23
24 }
```

2.3 Criptoanálise

Estudo que tem como objetivo a criação de técnicas e métodos que ajudam a obter o texto em claro a partir de um texto cifrado sem ter a devida autorização. Conforme a criptografia cresceu e seus estudos se aprofundaram, os estudos da criptoanálise também cresceram. Abaixo algumas técnicas de criptoanálise.

2.3.1 Força Bruta

Esse ataque visa procurar por exaustão a chave usada para cifrar a mensagem. Testa todas as possibilidades de chave e analisa o texto produzido.

É eficiente para chaves pequenas pois a quantidade de chaves possíveis é pequena. Por se tratar de uma procura pela chave, esse método pode ser aplicado em diferentes

tipos de algoritmo criptográfico.

Tamanho da chave	Tempo para quebrar
32	2.15 milissegundos
56	10.01 horas
128	5.4 X 10^{18} anos
168	5.9 X 10^{30} anos

Tabela 3 – Tempo gasto para encontrar chaves adaptado de (STALLINGS, 2011)

2.3.2 Análise de Frequência

Esse método visa analisar a frequência de ocorrência de cada carácter de um texto cifrado. Existem diversos estudos que indicam a frequência média de ocorrência de cada letra em um texto.

Letra	Frequência no texto	Letra	Frequência no texto
a	8.167	n	6.749
b	1.492	o	7.507
c	2.782	p	1.929
d	4.253	q	0.095
e	12.702	r	5.987
f	2.228	s	6.327
g	2.015	t	9.056
h	6.094	u	2.758
i	6.966	v	0.978
j	0.153	w	2.360
k	0.772	x	0.150
l	4.025	y	1.974
m	2.406	z	0.074

Tabela 4 – Frequência média de letras em um texto em inglês (LEWAND, 2000)

A ideia é identificar as frequências do texto cifrado e substituir as letras a medida que for identificado, até conseguir produzir um texto coeso.

Esse é o método que o algoritmo proposto neste trabalho visa dificultar, visto que sua aplicação pode ser utilizada em diversos tipos de algoritmos criptográficos e seus resultados podem ser satisfatórios. Com o balanceamento das ocorrências do texto cifrado, as frequências médias conhecidas não servirão de apoio para possíveis ataques, visto que cada carácter terá a mesma frequência de ocorrência.

3 Gerador de Números Pseudo-Aleatórios

Há muitos anos a geração de números aleatórios é objeto de estudo da matemática. Por muito tempo não existiu um algoritmo determinístico para a geração de números que tivesse um período de repetição suficientemente longo para aplicações complexas. Em 1998, os matemáticos japoneses Matishumote e Nishimura inventaram o primeiro algoritmo em que a periodicidade (2^{19937} - 1) excede o número de mudanças rotatórias de elétrons desde a criação do universo. (DUTANG; WUERTZ, 2009)

Apesar de existir o *TRNG* ou *True Random Number Generator*, que oferece sequências de números que são aleatórios, as mesmas não são determinísticas e isso faz com que sua utilização seja limitada, visto que em aplicações como a criptografia é necessário reproduzir a mesma sequência para a função de cifrar e de decifrar.

Com isso a utilização de geradores de números pseudo-aleatórios se faz necessária, porém a sequência gerada deve seguir dois critérios para que seus números possam ser considerados aleatórios:

1 - Aleatoriedade O principal requisito para uma sequência ser considerada aleatória é seus números serem aleatórios. Dois critérios podem ser usados para validar uma sequência:

Distribuição uniforme Os bits de uma sequência devem ter frequência aproximada, ou seja, a quantidade de 0 deve ser aproximada da quantidade de 1.

Independência Os números não devem ter relação entre si.

2 - Imprevisibilidade Esse critério serve para garantir que sabendo um número da sequência, é impossível prever o número anterior ou posterior.

Um gerador de números pseudo-aleatórios tem como entrada uma semente.[1] A semente normalmente é gerada a partir de um algoritmo *TRNG*. Pelo fato de serem utilizadas como entrada em algoritmos determinísticos as sementes devem ser guardadas de forma segura.

[1] Termo para especificar um número para geração de uma sequência.

3.1 Geradores

3.1.1 Gerador Linear Congruente

Uma técnica muito usada para geração de números pseudo-aleatórios é um algoritmo que primeiramente foi proposto por *Lehmer*, que é conhecido como método Linear Congruente. (STALLINGS, 2011) O algoritmo gera uma sequência dependendo das entradas e a função utilizada para essa geração é:

$$\mathcal{X}_n = (a \times X_{n-1} + c) \mod m \tag{3.1}$$

Para cada número produzido na sequência, essa formula é aplicada. A eficiência dessa sequência está diretamente relacionada a escolha das variáveis. A variável **m** é o módulo da equação e a condição de escolha dessa variável é ser maior que 0. A variável **a** é o multiplicador da função e sua condição é ser maior que 0 e menor que **m**. A variável c é o incremento e sua condição é estar entre 0 e **m**. O X_0 é a semente para a sequência, será o primeiro elemento da sequência e sua condição é estar entre 0 e **m**.

A escolha dessas entradas determina a periodicidade da sequência e por consequência seu desempenho. Como exemplo de uma sequência com periodicidade 4, as entradas seriam:

Variável	Valor
a	5
c	0
m	16
X_0	1

Tabela 5 – Parâmetros para sequência com periodicidade 4

Portanto o primeiro valor da sequência é o 1 por ser a semente. O segundo valor é calculado da seguinte maneira:

$$\mathcal{X}_n = (5 \times 1 + 0) \mod 16 \tag{3.2}$$

$$\mathcal{X}_n = 5 \tag{3.3}$$

Para o terceiro valor, o cálculo é feito utilizando o resultado do anterior:

$$\mathcal{X}_n = (5 \times 5 + 0) \mod 16 \tag{3.4}$$

$$\mathcal{X}_n = 9 \tag{3.5}$$

Finalmente para o quarto valor, também se utiliza o valor anterior em seu cálculo:

$$\mathcal{X}_n = (5 \times 9 + 0) \mod 16 \tag{3.6}$$

$$\mathcal{X}_n = 13 \tag{3.7}$$

A partir do quinto valor, a sequência se repete:

$$\mathcal{X}_n = (5 \times 13 + 0) \mod 16 \tag{3.8}$$

$$\mathcal{X}_n = 1 \tag{3.9}$$

Ou seja, a escolha dos parâmetros é um passo extremamente importante para esse gerador. Um exemplo com uma periodicidade relativamente grande pode ser obtida trocando apenas o valor de **m** da resolução anterior:

Variável	Valor
a	5
c	0
m	37
X_0	1

Tabela 6 – Parâmetros substituindo o valor de **m**

A sequência gerada a partir dessas variáveis, é:

[1, 5, 25, 14, 33, 17, 11, 18, 16, 6, 30, 2, 10, 13, 28, 29, 34, 22, 36, 32, 12, 23, 4, 20, 26, 19, 21, 31, 7, 25, 24, 9, 8, 3, 15]

Apesar de ter sido maior que a sequência anterior, isso não necessariamente prova sua eficiência, visto que para aplicações como a criptografia, números muito maiores seriam utilizados e sua periodicidade teria de ser maior também.

3.1.2 Gerador Blum Blum Shub

Uma estratégia para se gerar números pseudo-aleatórios seguramente é conhecida como Blum Blum Shub, nomeada pelos seus desenvolvedores. Tem a prova pública de sua força criptográfica de algoritmos para qualquer propósito. (STALLINGS, 2011) Os primeiros passos para o algoritmo são:

1. Escolha números primos **p** e **q**.
2. **p** e **q** devem ter resto 3 quando divididos por 4.

3. Obtenha **n**, multiplicando **p** e **q**.

4. Escolha **s** tal que **s** é primo relativo de **n**[2].

Com isso o gerador produzirá uma sequência com as seguintes condições:

$$\mathcal{X}_0 = s^2 \mod n \tag{3.10}$$

$$\mathcal{X}_n = (X_{n-1})^2 \mod n \tag{3.11}$$

$$\mathcal{B}_n = X_n \mod 2 \tag{3.12}$$

Ou seja, os *bits* menos significantes são retirados a cada número produzido.

3.1.3 Gerador Blum Micali

O gerador Blum Micali é um gerador de números pseudo-aleatórios provadamente seguro e foi criado por Manuel Blum, que também participou do desenvolvimento do Blum Blum Shub, e Silvio Micali. (AGERBLAD; ANDERSEN,)

O gerador é executado utilizando os seguintes passos iniciais:

1. Escolha um número primo **p**

2. Então o grupo cíclico[3] Z_p = [1, p-1]

3. a é um elemento de Z_p e é o número aleatório resultante

Com esses passos, o algoritmo tem as entradas para que possa ser realizada a geração dos números. A função que de geração é:

$$\mathcal{G}^k \equiv a \bmod p \tag{3.13}$$

Em que **k** é um número inteiro positivo e g um inteiro que representará um grupo de números inteiros aleatórios.

[2] p e q não são fatores de s
[3] Grupo que é gerado por um elemento que e que o grupo finaliza nesse mesmo elemento.

3.1.3.1 Exemplo

Dado que:

1. p = 17
2. Z_p = [1,16]

Então para g = 3 e k = 4: $3^4 \equiv 13 \mod 17$. Ou seja, o número aleatório 13 satisfaz a condição de estar incluso no grupo Z_p e é o número aleatório do grupo 3.

3.2 Segurança comprovada em geradores de números pseudo-aleatórios

Segurança comprovada em criptografia significa que quebrar um algoritmo criptográfico específico é tão difícil quanto resolver um problema específico conhecido por ser difícil. (AGERBLAD; ANDERSEN,)

Uma das aplicações de números pseudo-aleatórios seguros é servir de geradores de *keystream* para a cifra de fluxo. O algoritmo nesse trabalho proposto, irá utilizar esse de tipo de gerador de *keystream*.

Existem dois geradores de números pseudo-aleatórios que são publicamente comprovados em segurança: *Blum Blum Shub* e *Blum Micali*.

3.2.1 Segurança do Blum Blum Shub

A segurança do *Blum Blum Shub* é obtida pelo fato de que seu funcionamento se baseia na residualidade quadrática. Esse problema é relacionado em decidir se **x** é resíduo quadrático nas condições: mdc(x, N) = 1 e que **x** está entre $1 \leq x \leq N$. [4]

Para ser um resíduo quadrático, **x** tem que ter a seguinte propriedade $y^2 \equiv x \mod N$, para um inteiro y. Para esse cálculo, um algoritmo recebe N e x como parâmetros e retorna 1 se for resíduo quadrático e 0 se não for. (AGERBLAD; ANDERSEN,)

3.2.2 Segurança do Blum Micali

Sua segurança é provida pelo problema do logaritmo discreto. Esse gerador utiliza da seguinte função:

$$\mathcal{G}^k \equiv a \mod p \tag{3.14}$$

Realizar a exponenciação logarítmica é um problema fácil de resolver, porém o inverso, o logaritmo discreto, é uma operação comprovadamente difícil.

[4] N é igual a multiplicação dos números primos escolhidos para a geração dos números.

3.2.3 Comparação do Blum Blum Shub e Blum Micali

No trabalho de Josefin Agerblad e Martin Andersen em *Provably Secure Pseudo-Random Generators*, foi realizada uma comparação dos dois geradores de números pseudo aleatórios, que são comprovadamente seguros, nos quesitos de aplicação, velocidade e segurança.

No quesito aplicação, o *Blum Blum Shub* foi considerado mais vantajoso pelo fato que esse gerador é melhor aplicado para cifração de chave pública e o *Blum Micali* ser mais vantajoso para cifrações de chave privada. Como a cifração de chave pública é mais utilizada na criptografia, o *Blum Blum Shub* foi ganhador nesse quesito.

No quesito velocidade, os autores não conseguiram determinar qual gerador é mais rápido, visto que para um gerador ser considerado seguro, o desempenho do mesmo é prejudicado já que suas operações são mais complexas.

No quesito segurança, os autores também não conseguiram chegar a um consenso, visto que o problema que dificulta a quebra dos dois geradores são distintos e igualmente difíceis de serem resolvidos.

4 Proposta de Algoritmo

Neste capítulo será apresentado o algoritmo que este trabalho visa propor, detalhes de seu funcionamento e suas vantagens.

4.1 Motivação

A evolução no campo da criptografia é constante e há sempre novas propostas de algoritmos que focam em solucionar um obstáculo imposto por possíveis atacantes e seus métodos para se obter informações sigilosas.

O algoritmo proposto neste trabalho visa diminuir as possibilidades de análises de frequência de um texto cifrado. É uma nova modalidade na produção de texto cifrado utilizando cifra de fluxo.

4.2 Funcionamento

O algoritmo de cifração terá como entrada um texto em claro e uma sequência de números pseudo-aleatórios e terá como objetivo produzir um texto cifrado onde as frequências de ocorrência dos valores de *bytes* cifrados seja totalmente balanceada.

O mecanismo utilizado para este balanceamento será produzir o texto cifrado em "rodadas" de 256 *bytes* em que cada valor de *byte* cifrado só aparece uma vez por rodada. Isto é, os 256 *bytes* cifrados de cada rodada são distintos, porém a ordem em que aparecem em cada rodada é aleatória (ou melhor dizendo, pseudo-aleatória).

Para se garantir que nenhum valor de *byte* cifrado se repita em uma rodada, utiliza-se uma tabela de 256 posições. No início da rodada a tabela tem todas a suas 256 posições marcadas como não ocupadas. Para cada *byte* de texto em claro que se deseja cifrar será feito um ou-exclusivo com o valor correspondente fornecido pelo gerador de números aleatórios resultando no *byte* pré-cifrado. Tendo-se obtido o *byte* pré-cifrado verifica-se se este valor já está ocupado na tabela.

Caso o valor do *byte* pré-cifrado obtido corresponda a uma posição já ocupada na tabela,temos um conflito. Este valor não será aceito como *byte* cifrado e o próximo valor da sequência pseudo-aleatória será utilizado para tentar novamente a cifração do *byte* em claro. Este procedimento será repetido até que o valor pré-cifrado obtido corresponda a uma posição não ocupada na tabela, quando então este valor será aceito como o *byte* cifrado (correspondente ao byte de texto em claro que está sendo processado pelo cifrador).

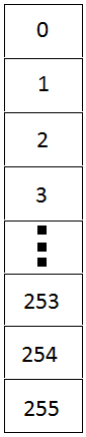

Figura 15 – Tabela de *byte* para uso do algoritmo

O número de valores da sequência pseudo-aleatória não utilizados será contabilizado em uma variável j.

Caso o valor do *byte* pré-cifrado obtido corresponda a uma posição não ocupada na tabela, este valor será aceito como *byte* cifrado, a posição correspondente na tabela será marcada como ocupada, e a variável j será reinicializada com o valor 0.

Aqui se introduz o problema de como enviar ao decifrador as seguintes informações:

1. Que houve um conflito na cifração (quando o mesmo ocorrer)
2. Em ocorrendo o conflito, qual o número de valores na sequência pseudo-aleatória deverão ser desprezados na decifração.

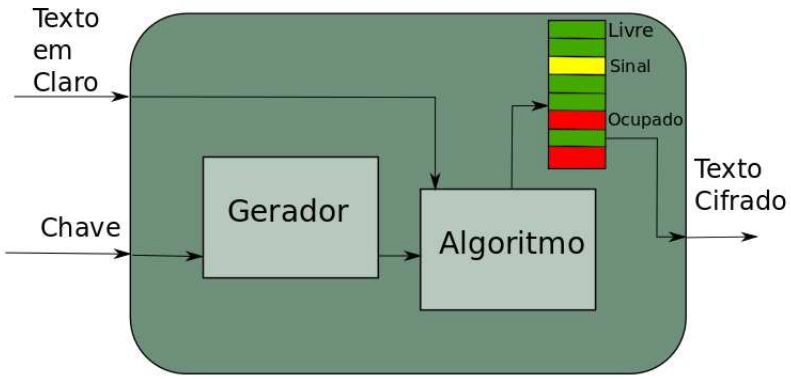

Figura 16 – Esquema do algoritmo de cifração

Para informar ao decifrador que houve um conflito na cifração propomos a utilização de um valor de *byte* entre 0 e 255 chamado de **sinal**.Este valor é o primeiro *byte* do gerador de *bytes*, e será atualizado a cada vez que a tabela se completa, se tornando o primeiro valor do gerador pseudo-aleatório da próxima rodada. Este valor será enviado pelo cifrador como parte da sequência de *bytes* cifrados indicando ao decifrador que o *byte* seguinte na sequência de *bytes* cifrados corresponderá ao número de valores da sequência produzida pelo gerador pseudo-aleatório que deverão ser desprezados. O decifrador saberá o *byte* recebido é sinal por meio de um contador, que será incrementado a cada vez que um *byte* for decifrado. Após o envio do *byte* de sinal, o cifrador enviará o valor da variável j (também cifrado), e finalmente enviará o *byte* cifrado.

O processo de cifração, que pode ser visto no pseudo-código 4.1, segue os seguintes caminhos:

O caminho ideal é o que o resultado da operação ou-exclusivo entre o *byte* do texto em claro e o número aleatório correspondente sempre corresponda a uma posição livre na tabela de balanceamento.

Código 4.1 Pseudo-Código Cifração

```
signal = S[0] // primeiro sinal do gerador
table[255] = '0'
count = 0;
while(plainText != EOF)
    if(table[plainText[k] ^ S[k]] != '1'){
        if(j == 0)
            output(plainText[k] ^ S[k]);
        if( j > 0){
            output(S[old] ^ j);
            output(S[k+j] ^ plainText[k]);
            j = 0;
        }
        count++;
    }
    if(table[plainText[k] ^ S[k] == '1'){
        if(j == 0){
            output(signal);
            S[old] = S[k+1];
            j = 1;
        }
        if(j > 0){
            j = j + 1;
```

```
23          }
24      }
25      //se tabela completar
26      if(count == 256){
27          resetTable();
28          signal = S[k]
29          count = 0;
30      }
31  }
```

Caso se detecte um conflito de resultados na tabela de balanceamento, é enviado o *byte* de sinal para o receptor da mensagem, pois com isso o receptor saberá que o próximo valor recebido será a quantidade de números pseudo-aleatórios que deverão ser descartados da sequência.

Então o algoritmo vai executar o comando de descartar os números pseudo-aleatórios, até que se encontre um resultado em uma posição que esteja livre na tabela de balanceamento.

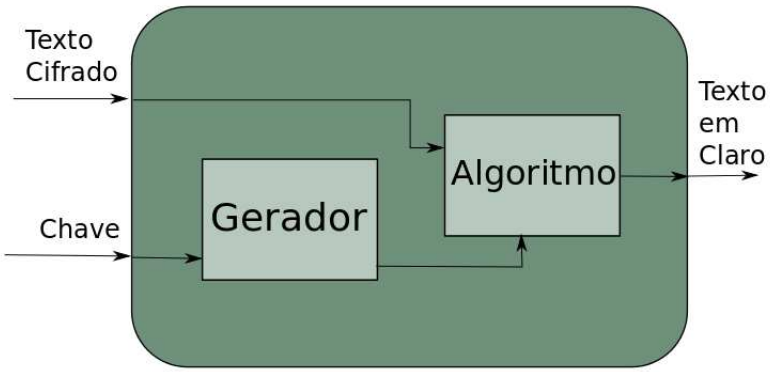

Figura 17 – Esquema do algoritmo de decifração

Ao encontrar essa posição livre, o algoritmo (no lado da cifração) enviará o *byte* de sinal (sem cifrar) e em seguida enviará o resultado do ou-exclusivo entre **j** (quantidade de números descartados) e S_{old}. Então o valor de **j** é novamente reinicializado como 0.

Quando a tabela for totalmente concluída, a rodada termina e o processo se inicia de novo com uma nova rodada, reinicializando-se a tabela (todas as suas posições são marcadas como desocupadas) e um novo sinal é gerado. Esse processo termina quando os *bytes* do texto em claro forem todos cifrados. A cada rodada o *byte* de sinal terá um

valor diferente. Escolhemos o primeiro *byte* da sequência de números pseudo-aleatórios da rodada como o *byte* de sinal.

Podemos observar o processo de decifração no pseudo-código 4.2. Neste não é necessário se utilizar a tabela de balanceamento. Para se realizar a decifração é necessário que o decifrador tenha a capacidade de produzir a mesma sequência de números aleatórios que o cifrador. Isto implica que ele deve ter a mesma chave que foi utilizada no processo de cifrar.

No início de cada rodada na decifração, o primeiro *byte* da sequência aleatória é separado como o *byte* de sinal. Os valores pseudo-aleatórios seguintes serão usados no processo de decifração. O primeiro passo no processo de decifração é comparar o *byte* cifrado recebido com o valor de sinal utilizado na rodada.

Código 4.2 Pseudo-Código Decifração

```
1  signal = S[0] // primeiro sinal do gerador
2  count = 0;
3  while(cipherText != EOF)
4      if(C[k] != sinal){
5          output(C[k] ^ S[k];
6          count++;
7      }
8      if((C[k] == signal){
9          j = C[k+1] ^ S[k+1];
10         S = S[k+j];
11         output(C[k] ^ S);
12         count++;
13     }
14     //se tabela completar
15     if(count == 256){
16         resetTable();
17         signal = S[k]
18         count = 0;
19     }
20 }
```

Se o valor do *byte* cifrado for diferente do valor do sinal utilizado na rodada, realiza-se a operação ou-exclusivo entre o *byte* cifrado e o valor fornecido pelo gerador pseudo-aleatório, obtendo-se assim o *byte* de texto em claro correspondente.

Se o valor do *byte* cifrado for igual ao *byte* de sinal utilizado na rodada, isso quer dizer que o próximo *byte* do texto cifrado conterá o valor da variável **j** enviado pelo

cifrador, que indica quantos números aleatórios devem ser descartados para o processo de decifração. Ao se descartar esses números, realiza-se a decifração do *byte* cifrado e o processo de decifração volta ao curso normal de execução.

4.3 Vantagens

A maior vantagem deste algoritmo é a equalização das frequências de ocorrência de cada valor de texto cifrado enviado. Excluindo-se os valores dos *bytes* de sinal e os valores dos números de sequência pseudo-aleatórios a serem desprezados na decifração, a frequência de cada valor possível dos *bytes* de texto cifrado será idêntica. Isto é, o algoritmo produz um texto cifrado perfeitamente balanceado.

Outra vantagem do algoritmo de balanceamento de frequência proposto é que ele é totalmente genérico. Isto é, ele funciona com qualquer gerador de números pseudo-aleatórios e algoritmos de cifra de fluxo, tais como: A5/1, A5/2, E0, RC4, *Blum Blum Shub*, *Blum Micali*, etc.

5 Implementação

Neste capítulo serão apresentados detalhes da implementação deste trabalho. Será apresentada a arquitetura que foi criada para o desenvolvimento do algoritmo proposto, um diagrama de sequência, que proverá melhor visibilidade sobre os passos que são seguidos na sua execução e uma explicação sobre os elementos que o compõe.

5.1 Arquitetura

A figura 18 apresenta a arquitetura que foi utilizada para a realização dos experimentos. A arquitetura foi desenvolvida com o intuito de reproduzir uma aplicação real de um algoritmo de criptografia. As partes do cifrador e do decifrador são executadas em dois computadores distintos que se comunicam utilizando o protocolo de rede TCP/IP.

De maneira geral, cada lado terá uma *thread* que auxiliará no envio/recebimento do *byte* cifrado usando o *socket* TCP/IP. A parte de cifração/decifração será utilizando o núcleo do *RC4* alterado de forma a implementar o algoritmo de balanceamento proposto

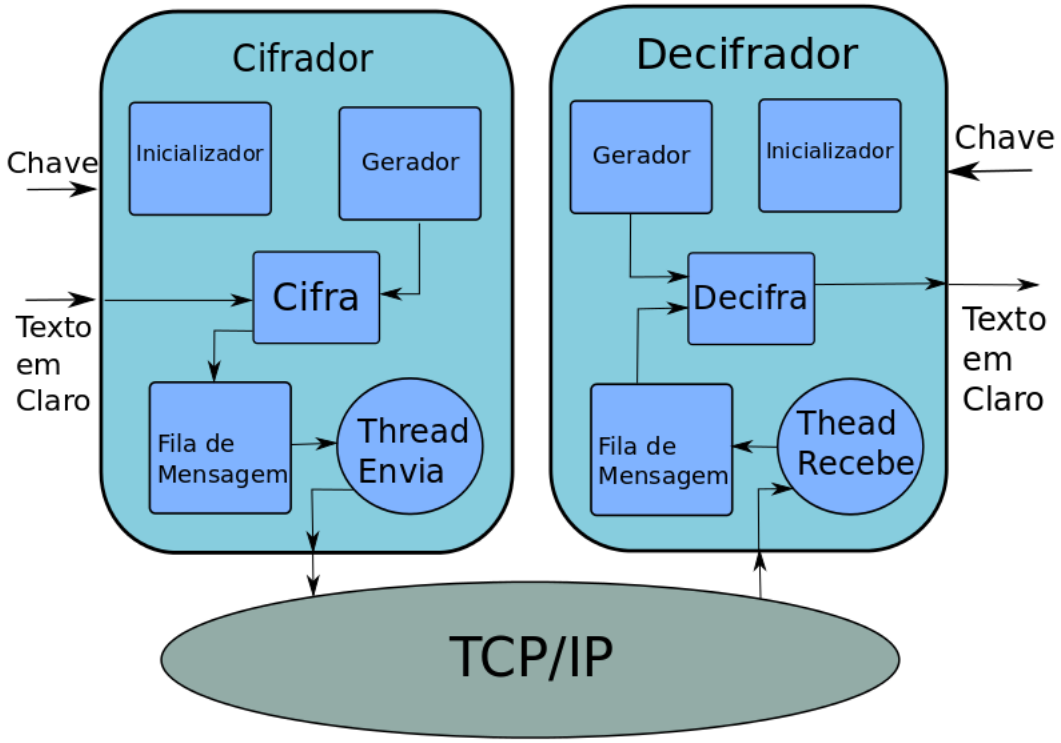

Figura 18 – Arquitetura utilizada no projeto

5.1.1 Inicializador do Gerador

Esse elemento é responsável por fazer a inicialização do gerador que será utilizado no algoritmo.

O gerador escolhido para os experimentos foi o do próprio *RC4*. O *RC4* tem como entradas o texto em claro que será cifrado e uma chave, esta chave é utilizada para fazer a inicialização do gerador. A chave utilizada foi gerada pelo *ssh-keygen*.

5.1.2 Gerador de *Bytes*

Como explicado anteriormente, o gerador escolhido foi o do próprio *RC4* e sua geração foi feita de acordo com o que o algoritmo aconselha.

5.1.3 Fila de Mensagem

A fila de mensagem foi utilizada para servir de intermediário entre o cifrador/decifrador e a *thread* que envia/recebe os *bytes* cifrados.

5.1.4 *Thread* de Envio para Decifrador

Esta thread é responsável por retirar os bytes cifrados da fila de mensagens e os enviar, via *TCP/IP*, para o decifrador.

Optou-se pelo uso desta *thread* para minimizar o *overhead* do envio dos bytes pela rede através da paralelização da execução, evitando, com isso, que o cifrador fique parado aguardando a rede.

5.1.5 *Thread* para Receber do Cifrador

Assim como no cifrador, o decifrador também conta com uma *thread* auxiliar para receber os *bytes* que são enviados pelo cifrador e escrever na fila de mensagem.

Neste caso, provavelmente, o uso da *thread* não traz um ganho apreciável, visto que a rede é o maior gargalo, por isso, a leitura do *socket* é feita a medida que o *buffer* do *TCP/IP* se completa.

5.1.6 *TCP/IP*

Quando estávamos analisando o melhor protocolo de comunicação para ser utilizado em conjunto com o algoritmo, decidimos pelo TCP/IP pela reciprocidade que o mesmo contém„ além da garantia de entrega e sequência dos dados

Em algoritmos de criptografia, a ordem com que os *bytes* são enviados/lidos é de extrema importância, para que o texto decifrado tenha consistência e integridade.

5.1.7 Decifrador

Processo responsável em fazer a tradução dos *bytes* cifrados.

O decifrador irá ter acesso aos *bytes* cifrados na fila de mensagem que será populada pela *thread* de recebimento. A operação que é efetuada é somente um ou-exclusivo do *byte* proveniente do gerador e o *byte* correspondente.

5.1.8 Cifrador

Processo responsável em cifrar os *bytes* em claro que se deseja obter segurança.

Uma vez cifrado, postará na fila de mensagens para que a *thread* de envio faça a sua parte e envie o mesmo para o decifrador

5.2 Diagrama de Sequência

O diagrama de sequência representado na figura 19 tem como objetivo demonstrar como os processos,cifra e decifra, interagem com os outros elementos, tais como: fila de mensagem, *threads* e a comunicação entre os mesmos.

1. O processo de decifração deve ser o primeiro a ser iniciado, pois ele é responsável por iniciar o servidor em que o processo da cifra irá se conectar para a comunicação e envio de *bytes* cifrados.

2. Em seguida, o processo de decifração irá iniciar a *thread* que será responsável por ler as mensagens proveniente do *socket* de comunicação.

3. O processo de decifração também inicia a fila de mensagem que irá utilizar.

4. Quando o processo de cifração inicia, ele tenta se conectar com o servidor e o servidor o aceita ou não.

5. Quando a comunicação é estabelecida, o processo de cifração inicia a *thread* que será responsável por enviar os *bytes* cifrados para o decifrador.

6. Por fim, o processo de cifração também inicia a fila de mensagem.

7. Os passos anteriores são somente para inicialização dos processos e *threads*. A parte de cifração começa quando tudo já está iniciado e acontece da seguinte forma:

    a) O processo de cifração faz a leitura do arquivo que contém o texto em claro e junto com o *byte* do gerador executa a operação de ou-exclusivo.

    b) Em seguida, o *byte* cifrado é postado na fila de mensagem.

c) Por sua vez, a *thread* que envia os *bytes* cifrados, faz a leitura da fila de mensagem e envia para o decifrador o *byte* cifrado.

d) A *thread* que faz a leitura do *socket*, recebe o valor do *byte* cifrado e o escreve na fila de mensagem do decifrador.

e) Por fim, o decifrador faz a leitura da fila de mensagem e o decifra realizando a operação ou-exclusivo com o *byte* do gerador.

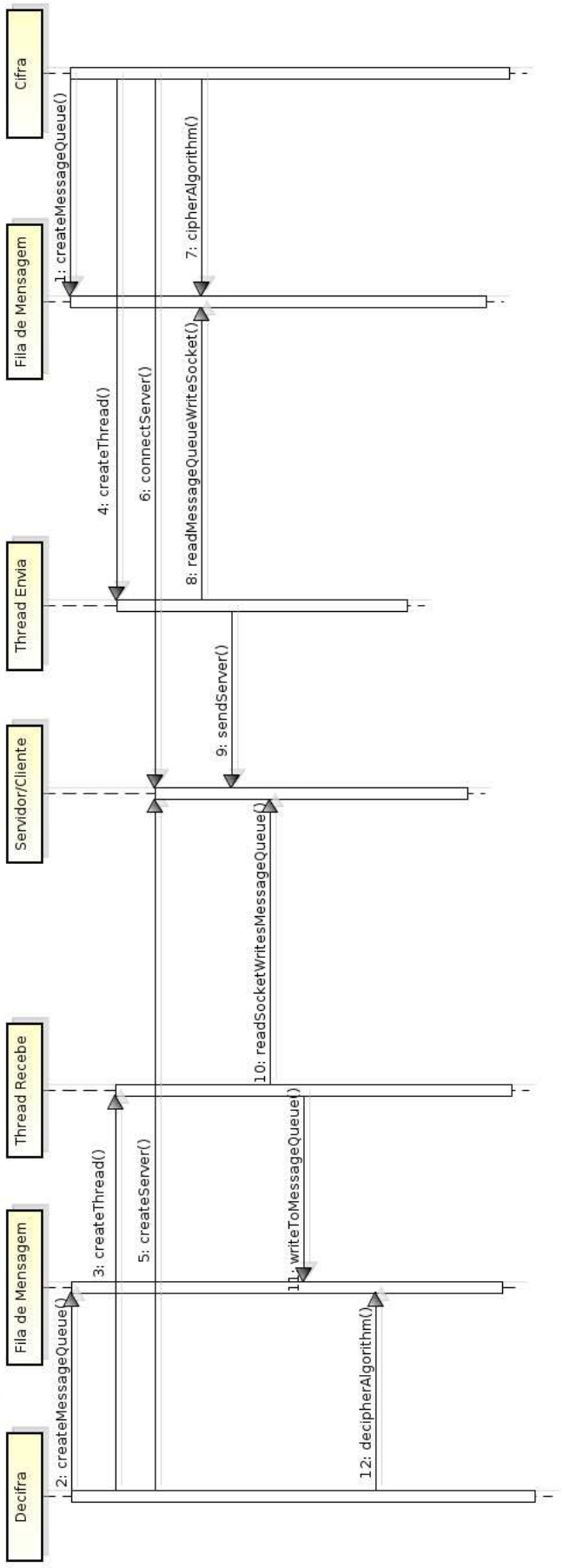

Figura 19 – Diagrama de Sequência

6 Resultados

Neste trabalho foram desenvolvidos diversos experimentos para se chegar a um resultado considerado satisfatório para a implementação do algoritmo proposto.O critério utilizado para a avaliação e validação dos experimentos foi o tempo total de execução. O tempo total de execução é a soma do processo da cifração combinada com a *thread* de envio e a decifração.

Os primeiros experimentos analisaram os tempos do processo de cifração e *thread* de envio e a medição dos tempos médios necessários para a resolução dos conflitos.

Depois de alguns experimentos com os geradores: linear congruente, *Blum Blum Shub* e *Blum Micali*, chegou-se a conclusão que a utilização destes geradores complicaria a análise de desempenho entre o *RC4* e o algoritmo aqui proposto. Isso se deve ao fato de que esses geradores são usados principalmente para geração de números, sem a preocupação com a sua repetibilidade. Para conseguir números com uma repetibilidade menor, seria necessário estudar o uso da chave fornecida para a cifração e transformá-la em entradas satisfatórias para estes geradores. Estes geradores demandam valores muito específicos para que o algoritmo tenha resultados satisfatórios e se esses requisitos não forem satisfeitos pode-se chegar a um *loop* infinito na procura de um *byte* que satisfaça as condições do algoritmo de balanceamento proposto.

O *RC4*, por outro lado, se demonstrou bastante eficaz na geração de números esparsos o suficiente para minimizar os conflitos e evitar a possibilidade do *loop* infinito, sendo, por esta razão, o único utilizado para este estudo.

Uma vez escolhida a melhor implementação do algoritmo, usou-se o *RC4* puro para servir de base de tempo para verificar o *overhead* imposto na cifrarão pelo algoritmo proposto.

6.1 Ambiente Utilizado para Experimentos

Foram utilizados dois computadores para os experimentos aqui descritos. O primeiro computador, que foi o responsável pela cifração do texto em claro, tem processador Intel i7, 16GB de memória RAM e utiliza o sistema operacional Ubuntu 14.04. O segundo computador, responsável pela decifração, tem processador Intel i7, 8GB de memória RAM e também utiliza Ubuntu 14.04 como sistema operacional. O compilador utilizado para ambos os computadores foi o *gcc* 4.8.2.

Durante os experimentos, cabos de rede ligados diretamente no *switch* de rede foram utilizados para que interferências da rede *wireless* não prejudicassem os resultados

dos experimentos.

Como existem fatores externos aos experimentos(internet, por exemplo) os resultados tendem a variar alguns milissegundos. Para se obter o valor mais acurado, foram coletados dez resultados por tamanho de texto para cada algoritmo em cada experimento e calculada a média, que será apresentada nas tabela de resultados dos experimentos

6.2 Tempos de Execução

Esses experimentos foram realizados principalmente para a aproximação de um resultado satisfatório. O fato de se ter tratamento de conflitos faz com que o algoritmo fique com o desempenho inferior ao do *RC4*, porém alguns ajustes foram feitos na arquitetura que foi planejada para a implementação do algoritmo para um resultado mais próximo possível.

6.2.1 Experimento 1

O primeiro experimento foi realizado utilizando somente a arquitetura que foi explanada no capítulo 5, sem qualquer intervenção sobre o algoritmo. Os resultados foram totalmente insatisfatórios, como podem ser observados na tabela 7.

Tamanho do texto(caracteres)	RC4(ms)		RC4 + algoritmo(ms)	
	Cifração	Cifração + *Thread*	Cifração	Cifração + *Thread*
10000	12.3	13.4	18.75	19.75
50000	51.6	54.2	97.875	98.875
100000	95.3	96.3	172.125	173.125
500000	1779.2	1836.6	3737.75	3740.5
1000000	6890.3	7023.6	9318.5	9325.75
5000000	48861.6	48905.4	54068	54083.625

Tabela 7 – Tempos obtidos no experimento 1

6.2.2 Experimento 2

Depois de análise dos resultados do primeiro experimento e a comparação dos resultados do *RC4* e *RC4* em conjunto com o algoritmo proposto, percebeu-se que algo que estava sendo utilizado na arquitetura do algoritmo e que não era o algoritmo em si estava afetando o desempenho da execução. A constatação se se deve ao fato de que o *RC4* também ter obtido resultados insatisfatórios. Depois de uma análise, descobriu-se que o tamanho da fila de mensagem *default*, que estava sendo utilizado, era o problema. Devido ao seu tamanho estar limitado a poucos caracteres, a *thread* de envio não estava sendo suficientemente rápida e isso estava fazendo com que o tempo da cifração também

fosse prejudicado. Outro indício que esse era o problema e não o algoritmo em si é que o tempo de cifração e tempo da *thread* de envio era praticamente o mesmo.

Com isso, foi efetuado o aumento da fila de mensagem e vimos que com a arquitetura implementada nesse trabalho isso é de extrema importância para o bom desempenho do algoritmo. Os resultados com a fila de mensagem expandida está na tabela 8.

Tamanho do texto(caracteres)	RC4(ms)		RC4 + algoritmo(ms)	
	Cifração	Cifração + Thread	Cifração	Cifração + Thread
10000	6.2	10.4	11.8	17.4
50000	22.6	34.4	40	60.6
100000	36.2	60	73	114.2
500000	154.4	279.6	333.6	548.2
1000000	304.2	544.2	663.2	1169.6
5000000	1400.4	3528	3139	6743.2

Tabela 8 – Tempos obtidos no experimento 2

6.2.3 Experimento 3

Com os resultados obtidos pelo experimento anterior, resolvemos reavaliar a necessidade de enviar *byte* a *byte* o texto cifrado. Isso porque depois de algumas análises, percebemos que a quantidade de conflitos em uma cifração era equivalente a metade do texto em claro e para cada conflito três *bytes* a mais eram enviados, ou seja, a quantidade de bytes a mais enviados pelo algoritmo proposto era:

$$tcifrado = tclaro + 2 \times (tclaro/2) \tag{6.1}$$

Isso resulta no dobro do texto em claro e que somente o algoritmo proposto envia. Portanto, para diminuir a quantidade de envios decidimos enviar os três *bytes* referentes ao conflito(sinal, quantidade de *bytes* descartados e *byte* cifrado) de uma só vez , aproximando, com isso, da quantidade de envios que o *RC4* realiza. O resultado desse experimento está descrito na tabela 9.

Tamanho do texto(caracteres)	RC4(ms)		RC4+Algoritmo(ms)	
	Cifração	Cifração + *Thread*	Cifração	Cifração + *Thread*
10.000	7.8	15.1	12.6	15.9
50.000	19.3	33.6	28.2	34.9
100.000	35.9	60.1	50.5	61
500.000	157	279.2	207.2	427.5
1.000.000	302.8	548.6	397.6	1020.4

Tabela 9 – Tempos obtidos no experimento 3

6.2.3.1 Tempo Total de Execução

Como os resultados anteriores foram os mais satisfatórios que obtivemos durante toda a fase de testes desse trabalho, decidimos dar uma visão de tempo de execução total. Nessa parte do experimento, o tempo de execução que foi levado em conta foram: cifração, *thread* de envio e recebimento e decifração. Para se obter o tempo total foi necessário a utilização de um meio de informar para o lado da decifração que o lado da cifração havia acabado,porque senão a decifração ficaria esperando novos *bytes* para decifrar. Portanto duas opções foram testadas:

Envio de dois *bytes* Nesse experimento, para cada *byte* cifrado enviado era enviado um *byte* que indicava o final do texto em claro para o lado da decifração.

Tamanho do texto(caracteres)	RC4(ms)		RC4 + algoritmo(ms)	
	Cifração(ms)	Cifração + *Thread*	Cifração	Cifração + *Thread*
10000	8.2	11.6	10.4	17.6
50000	21.2	33.4	27	61.8
100000	36.2	61.8	47.4	426.8
500000	159.6	281.2	212.4	426.8
1000000	311.2	574.4	419	835
5000000	1540.8	2794.2	2086	4501.4

Tabela 10 – Experimento tempo total enviando 2 *bytes*

Essa opção de envio foi descartada, visto que o tempo que a *thread* utiliza para enviar aumentou consideravelmente. Portanto, decidimos pela estratégia que mostraremos a seguir.

Envio de sequência de caracteres Nesse experimento, ao final do texto em claro foi enviado uma sequência de três bytes 0 como um marcador de fim de cifragem.

Tamanho do texto(caracteres)	RC4(ms)	RC4+Algoritmo(ms)	Tempo a mais(ms)
10.000	125.2	127.6	2.4
50.000	605.9	623	17.1
100.000	1144.8	1214.6	69.8
500.000	5730	6042.2	312.2
1.000.000	11,429.6	12,048.7	619.1

Tabela 11 – Tempo enviando sinal de fim do arquivo '000'

Com esses resultados obtivemos em média 4,34% de tempo a mais. A maior porcentagem obtida foi com o texto de tamanho 100.000 caracteres, 6,09%.

6.2.4 Resolução de Conflitos

Esse experimento visa dar maior visibilidade sobre a quantidade média de conflitos que ocorrem em diversos tamanhos de textos e o tempo médio das resoluções. Esses tempos podem ter uma variação, visto que para cada conflito a função de obter o tempo do sistema foi utilizada duas vezes. Para textos pequenos a variação pode ser pequena, porém a medida que o texto fica maior, esse tempo tende a aumentar também. A tabela 12 mostra os resultados obtidos.

Tamanho do texto(caracteres)	Tempo Extra(ms)	Quantidade de Conflitos	Tempo médio de resolução(ms)
10.000	10.4	5.037	0.002065
50.000	32.8	25.174	0.001303
100.000	67	49.998	0.001340
500.000	313.7	250.086	0.001254
1.000.000	611.8	499.890	0.001224
5.000.000	3029.1	2.497.605	0.001213

Tabela 12 – Quantidade de conflitos e tempo médio de resolução.

Percebe-se que a média de conflitos para cada tamanho de texto é aproximadamente a metade do tamanho do texto em claro. Outra percepção que pode ser obtida pela tabela é que conforme a quantidade de conflitos aumenta, o tempo médio de resolução diminui.

7 Conclusão

Conforme visto ao longo deste trabalho, existem formas de se obter informações, que devem ser mantidas em sigilo, mesmo não tendo a devida autorização e isto faz com que haja uma insegurança, por parte dos usuários dos algoritmos criptográficos, em utilizá-los. Visando resolver esse problema, a proposta deste trabalho é fazer a definição de um algoritmo de cifra de fluxo com balanceamento de frequência de ocorrência de caracteres.

Existem diversas técnicas de criptoanálise. Uma dessas é a análise de frequência de ocorrência de caracteres de um texto cifrado. Com essa técnica, o atacante pode ter acesso ao texto em claro, utilizando referências de frequência de ocorrência média de caracteres para cada idioma. O algoritmo, neste trabalho proposto, tem como foco dificultar a utilização dessa técnica, pois as frequências de ocorrência dos caracteres do texto cifrado serão balanceadas e assim torna-se impossível determinar qual caractere terá maior frequência no mesmo.

Outro ataque conhecido como *Statistical Mobius Analysis* tem como hipótese que um sistema criptográfico(tal como um gerador pseudo aleatório) obedece a uma geração de *bytes* com distribuição fixa(FILIOL, 2002). Como o algoritmo proposto neste trabalho não utiliza todos os *bytes* que são produzidos pelo gerador pseudo aleatório(em cada conflito em sua tabela interna alguns *bytes* da sequência pseudo aleatória precisam ser descartados, dependendo do texto em claro que está sendo cifrado) este ataque não funcionaria sobre uma mensagem cifrada utilizando o algoritmo proposto.

O Klein (2006) propõe um ataque utilizando uma relação, descoberta por ele, entre a saída do gerador pseudo aleatório e os estados internos do *RC4*. Dada uma distribuição de estados internos uniformes Klein obtém duas sequências possíveis de *bytes* que serão gerados pelo gerador pseudo aleatório. Por essa abordagem é possível determinar todos os *bytes* que serão produzidos pelo *RC4*. O algoritmo proposto neste trabalho também mitiga esse ataque, pois o descarte de *bytes* da sequência pseudo aleatória(que ocorre durante a resolução de conflitos em sua tabela interna) confunde as previsões de um atacante que utiliza a relação de Klein.

Após a execução dos experimentos, podemos afirmar que a utilização do algoritmo em conjunto com o *RC4* é possível com um impacto pequeno no desempenho do tempo total. Algumas observações devem ser feitas, por exemplo:

1. a utilização da fila de mensagem do *Linux* impacta em haver uma alteração em seu tamanho para que o algoritmo funcione sem impactos

2. a utilização desse algoritmo é indicado para comunicações síncronas, pois somente

dessa forma, os desempenhos do *RC4* e do *RC4* e o algoritmo se assemelham.

7.1 Trabalhos Futuros

Comunicações assíncronas Após análises, foi percebido que uma melhoria que deve ser implementada é a possibilidade de se utilizar o algoritmo em comunicações assíncronas.

Multi-*threading* Uma maneira de auxiliar o algoritmo em questão de desempenho, seria implementá-lo de forma a aceitar ser multi-*threading* para a cifração e envio. Uma maneira de fazer com que as mensagens cheguem na ordem deve ser pensada.

Fila de mensagem estudar uma forma de implementar a comunicação entre as threads sem o seu uso para evitar que o seu tamanho seja um gargalo

Geradores Encontrar uma maneira eficiente de utilizar os geradores: linear congruente, *Blum Blum Shub* e *Blum Micali*

Referências

AGERBLAD, J.; ANDERSEN, M. Provably secure pseudo-random generators, a literary study. Citado 2 vezes nas páginas 46 e 47.

BAKHTIARI, M.; MAAREF, M. A. An efficient stream cipher algorithm for data encryption. University Technology Malaysia, 2011. Citado na página 30.

BIRYUKOV, A. Block ciphers and stream ciphers: The state of the art. Katholieke Universiteit Leuven. Citado na página 24.

COLLINS, C. A comparative analysis of mark 1, colossus and zuse z4. 2006. Citado na página 24.

DAEMEN, J.; RIJMEN, V. Aes proposal: Rijndael. 1998. Citado na página 30.

DAVIES, D. A brief history of cryptography. 1997. Citado na página 23.

DUTANG, C.; WUERTZ, D. A note on random number generation. 2009. Citado na página 43.

FILIOL, E. A new statistical testing for symmetric ciphers and hash functions. 2002. Citado na página 67.

FLUHRER, S. R.; LUCKS, S. Analysis of the e0 encryption system. 2001. Citado 2 vezes nas páginas 9 e 35.

KLEIN, A. Attacks on the rc4 stream cipher. 2006. Citado na página 67.

LEWAND, R. E. *Cryptological Mathematics*. [S.l.]: The Matematical Association of America, 2000. Citado na página 41.

MAXIMOV, A. *Some Words on Cryptanalysis of Stream Ciphers*. Tese (Doutorado) — Lund University, 2006. Citado na página 21.

STALLINGS, W. *Cryptography and Network Security, Principles and Practice*. [S.l.]: Prentice Hall, 2011. Citado 3 vezes nas páginas 41, 44 e 45.

VAUDENAY, S. *A Classical Introduction To Cryptography, Applications for Communications Security*. [S.l.: s.n.], 2006. Citado na página 23.

WILCOX, J. Solving the enigma: History of the cryptanalytic bombe. 2006. Citado na página 23.

Apêndices

APÊNDICE A – Códigos

A.1 Bibliotecas

A.1.1 Conexão

```
1 #ifndef CONNECTIONS_H
2 #define CONNECTIONS_H
3
4 #include <functions.h>
5
6 /**
7  * Variables for network tcp
8  */
9 int sockdecipher, sockcipher, portno, clilen;
10 struct sockaddr_in serv_addr, cli_addr;
11 struct hostent *server;
12
13 /**
14  * Create server
15  */
16
17 int createServer(int argc, char *argv[]);
18
19 /**
20  * Connect to decipher(server)
21  */
22
23 int connectServer(int argc, char *argv[]);
24
25 /**
26  * Accept cipher(client)
27  */
28
29 int acceptClient();
30
31 /**
32  * Send byte to decipher(server)
33  */
```

```
34
35 int sendServer(message msg);
36
37 /**
38  * Accept byte from cipher(client)
39  */
40
41 int receiveMessage(unsigned char * cipherByte);
42
43 /**
44  * Close socket
45  **/
46
47 void closeSocket(int sockfd);
48
49 /**
50  * Print error for network
51  */
52
53 void error(char *msg);
54
55 #endif
```

A.1.2 Funções

```
1 #ifndef FUNCTIONS_H
2 #define FUNCTIONS_H
3 /**
4  * Libraries neccessary to the program
5  */
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <sys/types.h>
9 #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12 #include <string.h>
13 #include <unistd.h>
14 #include <pthread.h>
15 #include <sys/ipc.h>
16 #include <sys/msg.h>
17 #define TABLE_SIZE 255
```

```
18 #define CHARACTERS 255
19
20 /**
21  *  Structs for the program
22  **/
23
24 typedef struct RC4 {
25   int i;
26   int j;
27   unsigned char state[256];
28 } RC4;
29
30 typedef struct threadParameters {
31   char * fileOut;
32   char * fileName;
33   int eof;
34 } threadParameters;
35
36 typedef struct message {
37   long type;
38   short eof;
39   short bytes;
40   unsigned char characteres[30];
41   char filler[20];
42 } message;
43
44 /**
45  * Message queue id
46  **/
47
48 static int msqid;
49
50 /**
51  * Global variables for the program
52  */
53
54 char table[TABLE_SIZE];
55 unsigned char signal;
56
57 /**
58  *  Stores child as a thread
59  */
```

```
60
61 static pthread_t readsMessage;
62
63 /**
64  * Reset table
65  */
66
67 void resetTable();
68
69 /**
70  * RC4 Initializer
71  */
72
73 void rc4Initializer();
74
75 /**
76  * RC4 Get new byte
77  */
78
79 char rc4Generator();
80
81 /**
82  * Cipher/Decipher byte
83  */
84
85 unsigned char xorOperation(char textByte, char byteGenerated);
86
87 /**
88  * Checks if table is occupied
89  */
90
91 int checkTable(unsigned char cipherByte);
92
93 /**
94  * Cipher text with algorithm
95  */
96
97 int cipherAlgorithm(RC4 * params, char plaintext);
98
99 /**
100  * Thread to decipher a text with algorithm
101  */
```

```
102
103 void decipherAlgorithm(RC4 * params);
104
105 /**
106  * Rc4 plain cipher algorithm
107  */
108
109 void rc4CipherAlgorithm( RC4 * params,char plaintext);
110
111 /**
112  * Rc4 Plain Decipher Algorithm
113  */
114
115 void rc4DecipherAlgorithm( RC4 * params);
116
117 /**
118  * Creates thread
119  */
120
121 void createThread(threadParameters * dataThread,int type);
122
123 /**
124  * Creates message queue
125  */
126
127 int createMessageQueue(char * fileName, int type);
128
129 /**
130  * Destroy message queue
131  */
132
133 void destroyMessageQueue();
134
135 /**
136  * Writes character to message queue
137  */
138
139 int writeMessageToQueue(message msg, int count);
140
141 #endif
```

A.2 Conexão e Funções

A.2.1 Conexão

```
1 #include <connections.h>
2 #include <functions.h>
3
4 /**
5  * Create server
6  */
7
8 int createServer(int argc, char *argv[]){
9
10   if (argc < 2) {
11     fprintf(stderr,"ERROR, no port provided\n");
12     exit(1);
13   }
14   sockdecipher = socket(AF_INET, SOCK_STREAM, 0);
15   if (sockdecipher < 0)
16     error("ERROR opening socket");
17
18   bzero((char *) &serv_addr, sizeof(serv_addr));
19   portno = atoi(argv[1]);
20   serv_addr.sin_family = AF_INET;
21   serv_addr.sin_addr.s_addr = inet_addr("0.0.0.0");
22   serv_addr.sin_port = htons(portno);
23   if (bind(sockdecipher, (struct sockaddr *) &serv_addr,sizeof(
       serv_addr)) < 0) {
24     error("ERROR on binding");
25     return 1;
26   }
27   return 0;
28
29 }
30
31 /**
32  * Connect to decipher(server)
33  */
34
35 int connectServer(int argc, char *argv[]){
36   if (argc < 3) {
37     fprintf(stderr,"usage %s hostname port\n", argv[0]);
```

```
38      exit(0);
39    }
40    portno = atoi(argv[2]);
41    sockdecipher = socket(AF_INET, SOCK_STREAM, 0);
42    if (sockdecipher < 0)
43      error("ERROR opening socket");
44
45    server = gethostbyname(argv[1]);
46    if (server == NULL) {
47      fprintf(stderr,"ERROR, no such host\n");
48      exit(0);
49    }
50
51    bzero((char *) &serv_addr, sizeof(serv_addr));
52    serv_addr.sin_family = AF_INET;
53    bcopy((char *)server->h_addr,
54    (char *)&serv_addr.sin_addr.s_addr,
55    server->h_length);
56    serv_addr.sin_port = htons(portno);
57    if (connect(sockdecipher,(struct sockaddr *)&serv_addr,sizeof(
          serv_addr)) < 0){
58      error("ERROR connecting");
59      return 1;
60    }
61    return 0;
62 }
63
64 /**
65  * Accept cipher(client)
66  */
67
68 int acceptClient(){
69    listen(sockdecipher,5);
70    clilen = sizeof(cli_addr);
71    sockcipher = accept(sockdecipher, (struct sockaddr *) &cli_addr
          , &clilen);
72    if (sockcipher < 0){
73      error("ERROR on accept");
74      return 1;
75    }
76    return 0;
77 }
```

```
78
79 /**
80  * Close some socket
81  **/
82
83 void closeSocket(int sockfd) {
84   shutdown(sockfd, SHUT_RDWR);
85   close(sockfd);
86 }
87
88 /**
89  * Print error for network
90  */
91
92 void error(char *msg){
93   perror(msg);
94   exit(1);
95 }
96
97 /**
98  * Send byte to decipher(server
99  */
100
101 int sendServer(message msg){
102   int n;
103   n = write(sockdecipher, msg.characteres, msg.bytes);
104   if (n < 0){
105       error("ERROR writing to socket");
106       return 0;
107   }
108   return 1;
109 }
110
111 /**
112  * Accept byte from cipher(client)
113  */
114
115 int receiveMessage(unsigned char * cipherByte){
116   int n;
117   n = read(sockcipher,cipherByte,8192);
118     if (n < 0){
119       error("ERROR reading from socket");
```

```
120       return 1;
121     }
122   return n;
123 }
```

A.2.2 Funções

```
1  #include <functions.h>
2  #include <connections.h>
3  #include <pthread.h>
4  #include <sys/ipc.h>
5  #include <sys/msg.h>
6  #include <arpa/inet.h>
7  #include <netinet/in.h>
8
9  /**
10  * Reset table
11  */
12
13 void resetTable(){
14   int i = 0;
15   for(; i <= TABLE_SIZE; i++){
16     table[i] = '0';
17   }
18 }
19
20 /**
21  * RC4 Initializer
22  */
23
24 void rc4Initializer(RC4 * params, char key[]){
25
26   int i,j = 0;
27   size_t keylen = strlen(key);
28   unsigned char temp[255];
29   for(i = 0; i <= CHARACTERS; i++){
30     params->state[i] = i;
31     temp[i] = key[i % keylen];
32   }
33
34   unsigned char swap;
35   for(i = 0; i <= CHARACTERS; i++) {
```

```
36      j = (j + params->state[i] + temp[i]) % 256;
37      swap = params->state[j];
38      params->state[j] = params->state[i];
39      params->state[i] = swap;
40    }
41
42    params->i = 0;
43    params->j = 0;
44  }
45
46  /**
47   * RC4 Get new byte
48   */
49
50  char rc4Generator(RC4 * params){
51
52    unsigned char t;
53    unsigned char k;
54    unsigned char swap;
55    params->i = (params->i + 1) % 256;
56    params->j = (params->j + params->state[params->i]) % 256;
57    swap = params->state[params->j];
58    params->state[params->j] = params->state[params->i];
59    params->state[params->i] = swap;
60    t = (params->state[params->i] + params->state[params->j]) %
        256;
61    k = params->state[t];
62
63    return k;
64  }
65
66  /**
67   * Cipher/Decipher byte
68   */
69
70  unsigned char xorOperation(char textByte, char byteGenerated){
71
72    return textByte ^ byteGenerated;
73  }
74
75  /**
76   * Checks if table is occupied
```

```
77  */
78
79 int checkTable(unsigned char cipherByte){
80
81   if(table[cipherByte] == '1'){
82     //Value is occupied
83     return 1;
84   } else {
85     //Value is not occupied
86     return 0;
87   }
88 }
89
90 /**
91  * Writes character to message queue
92  */
93
94 int writeMessageToQueue(message msg, int count){
95   // Fix the problem with the queue when the cipher result is 0
96   if(msgsnd(msqid, &msg, 30, 0) == -1)
97   {
98     perror("msg_sender: Error while writing message to queue");
99     exit(1);
100  }
101  return 0;
102 }
103
104
105 /**
106  * Thread to cipher with algorithm
107  */
108
109 int cipherAlgorithm(RC4 * params, char plaintext){
110
111   unsigned char byteToCipher, cipherTextByte, oldS, cipherJ;
112   int result, j = 0, position = 0;
113   int count = 1;
114   message msg;
115   msg.type = 1;
116   msg.eof = 0;
117   do{
118     if(j == 255){
```

```
119       cipherJ = xorOperation(j, oldS);
120       msg.characteres[position] = cipherJ;
121       position++;
122       msg.characteres[position] = signal;
123       position++;
124       oldS = rc4Generator(params);
125       j = 1;
126     }
127     byteToCipher = rc4Generator(params);
128     cipherTextByte = xorOperation(plaintext, byteToCipher);
129
130     result = checkTable(cipherTextByte);
131
132     if(result == 0) {
133       if(j == 0){
134         msg.characteres[position] = cipherTextByte;
135         position++;
136         msg.bytes = position;
137         writeMessageToQueue(msg,position);
138         position = 0;
139
140       }else {
141         // cipher j with old_s and put on ipc along with
              ciphertext
142         cipherJ = xorOperation(j, oldS);
143         msg.characteres[position] = cipherJ;
144         position++;
145         msg.characteres[position] = cipherTextByte;
146         position++;
147         msg.bytes = position;
148         writeMessageToQueue(msg,position);
149         position = 0;
150
151       }
152       table[cipherTextByte] = '1';
153       break;
154     }else{
155       if(j == 0){
156         //put signal on ipc
157         msg.characteres[position] = signal;
158         position++;
159         oldS = byteToCipher;
```

```
160         j = 1;
161       }else{
162         j = j + 1;
163       }
164     }
165   }while(result == 1);
166
167   return 0;
168 }
169
170 /**
171  * Thread to decipher a text with algorithm
172  */
173
174 void decipherAlgorithm(RC4 * params){
175
176   unsigned char byteToDecipher, plainTextByte, cipherTextByte;
177   int j = 0, countTable = 256;
178   int charsRead, n;
179   message msg;
180   msg.type = 1;
181   while(1){
182     if(countTable > 255){
183       signal = rc4Generator(params);
184       countTable = 1;
185     }
186     charsRead = msgrcv(msqid, &msg, 30, 0 , 0);
187     byteToDecipher = rc4Generator(params);
188     if(msg.characteres[0] == signal) {
189       charsRead = msgrcv(msqid, &msg, 30, 0, 0);
190       j = xorOperation(msg.characteres[0],byteToDecipher);
191       int i = 1;
192       for(i = 1; i < j; i++){
193         byteToDecipher = rc4Generator(params);
194       }
195       continue;
196     }
197     plainTextByte = xorOperation(msg.characteres[0],
          byteToDecipher);
198     countTable++;
199     printf("%c", plainTextByte);
200     if(msg.eof == 1){
```

```
201         unsigned char test = '1';
202         n = write(sockcipher, &test, 1);
203         break;
204       }
205     }
206 }
207
208 /**
209  * Rc4 Plain Cipher Algorithm
210  */
211
212 void rc4CipherAlgorithm( RC4 * params,char plaintext){
213
214   unsigned char byteToCipher, cipherTextByte, oldS, cipherJ;
215   int result, j = 0;
216   int count = 1;
217   message msg;
218   msg.type = 1;
219   msg.eof = 0;
220
221   byteToCipher = rc4Generator(params);
222   cipherTextByte = xorOperation(plaintext, byteToCipher);
223
224   //put cipherTextByte on IPC
225   msg.characteres[0] = cipherTextByte;
226   msg.bytes = 1;
227   writeMessageToQueue(msg,msg.bytes);
228
229 }
230
231
232 /**
233  * Rc4 Plain Decipher Algorithm
234  */
235
236 void rc4DecipherAlgorithm( RC4 * params){
237   unsigned char byteToDecipher, plainTextByte;
238   int charsRead, n;
239   message msg;
240   msg.type = 1;
241   msg.eof = 0;
242   while(1){
```

```
243     charsRead = msgrcv(msqid, &msg, 30, 0 , 0);
244
245     if(msg.eof == 1){
246       unsigned char test = '1';
247       n = write(sockcipher, &test, 1);
248       break;
249     }
250     byteToDecipher = rc4Generator(params);
251     plainTextByte = xorOperation(msg.characteres[0],
          byteToDecipher);
252     printf("%c", plainTextByte);
253   }
254 }
255
256 /**
257  * Creates message queue
258  */
259
260 int createMessageQueue(char * fileName, int type){
261     #ifdef PRINT_DEBUG
262         printf("msg_receiver: Initiating\n");
263     #endif
264
265   key_t key;      /* Key to access message queue */
266
267     /* Create a key  */
268     key = ftok(fileName, 'A');
269     if(key == -1)
270     {
271         perror("msg_receiver: Error while creating message key");
272         exit(1);
273       }
274     #ifdef PRINT_DEBUG
275         printf("msg_receiver: message key created: %i\n", key);
276     #endif
277
278     /* With the created key, tries connect/create to a message
          queue */
279     msqid = msgget(key, 0644 | IPC_CREAT);
280     if(msqid == -1)
281     {
282         perror("msg_receiver: Coundn't create or connect to a
```

```
            message queue");
283         exit(1);
284     }
285     #ifdef PRINT_DEBUG
286         printf("msg_receiver: message id created: %i\n", msqid);
287     #endif
288
289   return 0;
290 }
291
292 /**
293  * Destroy message queue
294  */
295
296 void destroyMessageQueue(){
297     /* Destroys message queue */
298     if(msgctl(msqid, IPC_RMID, NULL) == -1)
299     {
300         perror("msgctl");
301         exit(1);
302     }
303 }
304
305
306 /**
307  * Reads from message queue and writes on file
308  */
309
310 static void * readMessageQueueWriteSocket(void * params){
311     threadParameters * data = params;
312   message msg;
313   msg.type = 1;
314   int count = 0;
315   int charsRead;
316
317     while(1){
318       charsRead = msgrcv(msqid, &msg, 30, 0, 0);
319         if(msg.eof == 1){
320           msg.characteres[0] = '0';
321           msg.bytes = 1;
322           for(count = 0; count < 3; count++){
323             sendServer(msg);
```

```
324             }
325             data->eof = 1;
326             break;
327           }
328           count++;
329           sendServer(msg);
330       }
331 }
332
333 /**
334  * Read file and writes on message queue
335  */
336
337 static void * readSocketWritesMessageQueue(void * params){
338   threadParameters * data = params;
339   message msg;
340   msg.type = 1;
341   unsigned char byte[8192];
342   int charsRead;
343   short notFirst = 0;
344   int count = 0;
345   int eof = 0;
346
347   while(1){
348     charsRead = receiveMessage(&byte[0]);
349     if(charsRead == 0){
350       if(notFirst){
351         msg.eof = 1;
352         writeMessageToQueue(msg,count);
353         break;
354       }
355       notFirst = 1;
356       continue;
357     }
358     msg.bytes = 1;
359     for(count = 0; count < charsRead; count++){
360       if(byte[count] == '0'){
361         eof++;
362         if(eof == 3){
363           //eof has arrived
364           msg.eof = 1;
365           writeMessageToQueue(msg,1);
```

```
366             break;
367           }
368         }else{
369           if(eof){
370             int j;
371             for(j = 0; j < eof; j++){
372               msg.characteres[0] = '0';
373               writeMessageToQueue(msg,1);
374             }
375           }
376           if(eof == 3){
377             break;
378           }
379           msg.characteres[0] = byte[count];
380           writeMessageToQueue(msg, 1);
381           eof = 0;
382         }
383       }
384     }
385 }
386 void createThread(threadParameters * dataThread,int type){
387
388   if(type == 1){
389     pthread_create(&readsMessage, NULL,
          readMessageQueueWriteSocket , dataThread);
390   }else{
391     pthread_create(&readsMessage, NULL,
          readSocketWritesMessageQueue , dataThread);
392   }
393 }
```

A.3 Programas Principais

A.3.1 Cifrador Algoritmo Proposto

```
1 #include <functions.h>
2 #include <connections.h>
3 #include <time.h>
4 #include <sys/time.h>
5
6
7 int main(int argc, char *argv[]){
```

```
8
9   connectServer(argc, argv);
10  char key[] = {"pAatOAgJqpdy7CECQbRojYLqy11O2W38nJm+
       Sfcz94QrNAH3EXhKFxx4t0FsjfiadGDlHkdLMG8jvgjG5jlNI72gkODO9IroRk
       +
       vy151xeBQoout9ZQ6wZktuqibySxKxGy7g0RcgfW6D7vnpV9bNfsxtOZDirJOhgQWErOq
       +GKYze25B9g0gheVEumbBpgz376VwJoHPZcFRVQw6lOc/EAaozgS1Uwh1/
       Zs1ThagZC4EP85ATVPfFJ45xNUAD1LsLqGjSOc1Y7MBU7EINwXDrDJIeKeyn+
       txKu0EoJ9QAgZyXqK4gOeUyZ1br1NMZ222Cgx+
       Q5unZA6qq8zUrHsi024wldyLJL0uqHpbHn6k3IrNrNcw45BLKBbsHf4JICaPIr825XeQb0
       +VrNUS9eSCeErUIF7iNxryaW2H67EXKds2R7pGMh+
       JlogxUUMbJ5THV5HcV2rSPLugeCWDW8MACwW2Q3d6E2t1/
       v4recq3IohzX4rBoDTz4Px91wCajdgPdJpVwP82Cyv210fqI4KBZidEgtqEzJbbh7fQvGc
       +iVGhCXs5S6XihWadbfa2XyM5hF56O3kZNFG0UITbd0/
       cClVtfA7PDNEgg2pZNjVoNvwuR+/85
       b0YhDnlX6xawNBdO6xBUuRSs5oHvmd2aTTphlxlnjAlXnfgW9Oo0ACq5YbDgo
       +npCkcE4xEyO19Pf92YORUbbk4j+
       RJHcCuDqC0cdNDN5HGSTWvJn2KbzPpdg3UAcrLc7jd4usrTuiJ7QVu6uXT6FOl2h7hMGEy
       /1kol4v5KChHq6iQ5/jkQV2049c4kN03BLceIXkyXs+y5hyVlmx/
       QBNYa3etsAiw58yMVyojsB0AWgVnUkT00IY3nw2UIgaj5ENZtcZz/
       yhRNJOwDQwG4d0n2IHZYmwgD47wXW20g1o3avaThsshBMiI6dMmcogpb2R8gtCkuGvkjwl
       /8cjGAA+wcVaOU9EndKUqT56uB4hcSQ96N63n1w07D1h6AK/
       CuELXE5wNtgPzQkgi83r+6qrY/
       rFeeO0dZAUUlR01Pyw15oGw7nmRBMBR2lak1Cjsi84I9K4JyW4t/
       LEuIvc122FB7YNCg9mRbTVMnUi3vWWcvVNur6/WQBtQllSdQ9Moo/
       mfYHA0TIyf9DBWBHtaJoWP7KvVfAgvjYyMU90ijiLDrASY8580Q1g3j6LioLEaXijfGst3
       +c7K2gTk+V1KBZEEphGXQgFrJClZwoz/
       zohtn53xEooCME2cDEVaDvBf2B9Wch1BaOFJIcJdJ8JrheUM3l6eUBuUtoy5XNHb9oJsWf
       +azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYfqs+
       hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+
       GhPZxlLDkogfSkQYZ5De6gNkwUi8qI3T0IaHjKs8tYOJU3+
       hlMSLSfUTwKn4JEx+V7FUiAUP0xI1Cxh5ySRQQoJqOA9eNj85"};
11
12  int messageCipher = 1;
13  int rc = createMessageQueue("cipher", messageCipher);
14
15  if(rc !=  0){
16    printf("Problem creating message queue");
17    return 1;
18  }
19
20  FILE *fp;
```

```
21  fp = fopen("teste.txt", "r");
22
23  // Starts thread cipher side
24  threadParameters dataThread;
25  dataThread.fileName = "cipher";
26  dataThread.fileOut = "cipher.txt";
27  dataThread.eof = 0;
28  int type = 1;
29  createThread(&dataThread, type);
30
31  //Message struct
32  message msg;
33  msg.type = 1;
34
35  char plaintext;
36
37  RC4 params;
38
39  int result, countTable = 0, count = 0;
40
41  resetTable();
42  rc4Initializer(&params, key);
43  signal = rc4Generator(&params);
44  table[signal] = '1';
45
46  while( ( plaintext = fgetc(fp) ) != EOF ){
47   if(countTable > 254){
48     resetTable();
49     signal = rc4Generator(&params);
50     table[signal] = '1';
51     countTable = 0;
52   }
53    do{
54      result = cipherAlgorithm(&params, plaintext);
55      if(result == 0){
56        countTable++;
57        break;
58      }else{
59        printf("ERROR");
60      }
61
62    }while(1);
```

```
63    }
64
65    msg.eof = 1;
66    writeMessageToQueue(msg, count);
67    fclose(fp);
68
69    printf("Waiting thread ends\n");
70    do{
71      usleep(100);
72    }while(dataThread.eof == 0);
73
74    destroyMessageQueue();
75
76
77  }
```

A.3.2 Decifrador Algoritmo Proposto

```
1  #include <functions.h>
2  #include <connections.h>
3
4  int main(int argc, char *argv[]){
5
6    createServer(argc, argv);
7    acceptClient();
8
9    char key[] = {"pAatOAgJqpdy7CECQbRojYLqy11O2W38nJm+
       Sfcz94QrNAH3EXhKFxx4t0FsjfiadGDlHkdLMG8jvgjG5jlNI72gkODO9IroRk
       +
       vy151xeBQoout9ZQ6wZktuqibySxKxGy7g0RcgfW6D7vnpV9bNfsxtOZDirJOhgQWErOq
       +GKYze25B9g0gheVEumbBpgz376VwJoHPZcFRVQw6lOc/EAaozgS1Uwh1/
       Zs1ThagZC4EP85ATVPfFJ45xNUAD1LsLqGjSOc1Y7MBU7EINwXDrDJIeKeyn+
       txKu0EoJ9QAgZyXqK4gOeUyZ1br1NMZ222Cgx+
       Q5unZA6qq8zUrHsi024wldyLJL0uqHpbHn6k3IrNrNcw45BLKBbsHf4JICaPIr825XeQb0
       +VrNUS9eSCeErUIF7iNxryaW2H67EXKds2R7pGMh+
       JlogxUUMbJ5THV5HcV2rSPLugeCWDW8MACwW2Q3d6E2t1/
       v4recq3IohzX4rBoDTz4Px91wCajdgPdJpVwP82Cyv210fqI4KBZidEgtqEzJbbh7fQvGc
       +iVGhCXs5S6XihWadbfa2XyM5hF56O3kZNFG0UITbd0/
       cClVtfA7PDNEgg2pZNjVoNvwuR+/85
       b0YhDnlX6xawNBdO6xBUuRSs5oHvmd2aTTphlxlnjAlXnfgW9Oo0ACq5YbDgo
       +npCkcE4xEyO19Pf92YORUbbk4j+
       RJHcCuDqC0cdNDN5HGSTWvJn2KbzPpdg3UAcrLc7jd4usrTuiJ7QVu6uXT6FOl2h7hMGEy
```

```
    /1kol4v5KChHq6iQ5/jkQV2049c4kN03BLceIXkyXs+y5hyVlmx/
    QBNYa3etsAiw58yMVyojsB0AWgVnUkT00IY3nw2UIgaj5ENZtcZz/
    yhRNJOwDQwG4d0n2IHZYmwgD47wXW20g1o3avaThsshBMiI6dMmcogpb2R8gtCkuGvkjwIkCed
    /8cjGAA+wcVaOU9EndKUqT56uB4hcSQ96N63n1w07D1h6AK/
    CuELXE5wNtgPzQkgi83r+6qrY/
    rFeeO0dZAUUlR01Pyw15oGw7nmRBMBR2lak1Cjsi84I9K4JyW4t/
    LEuIvc122FB7YNCg9mRbTVMnUi3vWWcvVNur6/WQBtQllSdQ9Moo/
    mfYHA0TIyf9DBWBHtaJoWP7KvVfAgvjYyMU90ijiLDrASY8580Q1g3j6LioLEaXijfGst3Rp
    +c7K2gTk+V1KBZEEphGXQgFrJClZwoz/
    zohtn53xEooCME2cDEVaDvBf2B9Wch1BaOFJIcJdJ8JrheUM3l6eUBuUtoy5XNHb9oJsWfZ
    +azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYfqs+
    hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+
    GhPZxlLDkogfSkQYZ5De6gNkwUi8qI3T0IaHjKs8tYOJU3+
    hlMSLSfUTwKn4JEx+V7FUiAUP0xI1Cxh5ySRQQoJqOA9eNj85"};
  int messageDecipher = 2;
  int rc = createMessageQueue("decipher", messageDecipher);
  if(rc !=  0){
    printf("Problem creating message queue");
    return 1;
  }

  // Starts thread decipher side
  threadParameters dataThreadDecipher;
  dataThreadDecipher.fileName = "test";
  dataThreadDecipher.fileOut = "cipher.txt";

  message msgCipher;
  msgCipher.type = 1;
  int type = 2;
  createThread(&dataThreadDecipher, type);

  //Message struct
  message msg;
  msg.type = 1;
  char cipherText;
  RC4 paramsDecipher;

  rc4Initializer(&paramsDecipher, key);

  decipherAlgorithm(&paramsDecipher);

  destroyMessageQueue();
```

```
38 }
```

A.3.3 Cifrador *RC4*

```
1  #include <functions.h>
2  #include <time.h>
3  #include <sys/time.h>
4  #include <connections.h>
5
6
7  int main(int argc, char *argv[]){
8
9    connectServer(argc, argv);
10   char key[] = {"pAatOAgJqpdy7CECQbRojYLqy11O2W38nJm+
       Sfcz94QrNAH3EXhKFxx4t0FsjfiadGDlHkdLMG8jvgjG5jlNI72gkODO9IroRk
       +
       vy151xeBQoout9ZQ6wZktuqibySxKxGy7g0RcgfW6D7vnpV9bNfsxtOZDirJOhgQWErOq
       +GKYze25B9g0gheVEumbBpgz376VwJoHPZcFRVQw6lOc/EAaozgS1Uwh1/
       Zs1ThagZC4EP85ATVPfFJ45xNUAD1LsLqGjSOc1Y7MBU7EINwXDrDJIeKeyn+
       txKu0EoJ9QAgZyXqK4gOeUyZ1br1NMZ222Cgx+
       Q5unZA6qq8zUrHsi024wldyLJL0uqHpbHn6k3IrNrNcw45BLKBbsHf4JICaPIr825XeQb0
       +VrNUS9eSCeErUIF7iNxryaW2H67EXKds2R7pGMh+
       JlogxUUMbJ5THV5HcV2rSPLugeCWDW8MACwW2Q3d6E2t1/
       v4recq3IohzX4rBoDTz4Px91wCajdgPdJpVwP82Cyv210fqI4KBZidEgtqEzJbbh7fQvGo
       +iVGhCXs5S6XihWadbfa2XyM5hF56O3kZNFG0UITbd0/
       cClVtfA7PDNEgg2pZNjVoNvwuR+/85
       b0YhDnlX6xawNBdO6xBUuRSs5oHvmd2aTTphlxlnjAlXnfgW9Oo0ACq5YbDgo
       +npCkcE4xEyO19Pf92YORUbbk4j+
       RJHcCuDqC0cdNDN5HGSTWvJn2KbzPpdg3UAcrLc7jd4usrTuiJ7QVu6uXT6FOl2h7hMGEy
       /1kol4v5KChHq6iQ5/jkQV2049c4kN03BLceIXkyXs+y5hyVlmx/
       QBNYa3etsAiw58yMVyojsB0AWgVnUkT00IY3nw2UIgaj5ENZtcZz/
       yhRNJOwDQwG4d0n2IHZYmwgD47wXW20g1o3avaThsshBMiI6dMmcogpb2R8gtCkuGvkjwl
       /8cjGAA+wcVaOU9EndKUqT56uB4hcSQ96N63n1w07D1h6AK/
       CuELXE5wNtgPzQkgi83r+6qrY/
       rFeeO0dZAUUlR01Pyw15oGw7nmRBMBR2lak1Cjsi84I9K4JyW4t/
       LEuIvc122FB7YNCg9mRbTVMnUi3vWWcvVNur6/WQBtQllSdQ9Moo/
       mfYHA0TIyf9DBWBHtaJoWP7KvVfAgvjYyMU90ijiLDrASY8580Q1g3j6LioLEaXijfGst3
       +c7K2gTk+V1KBZEEphGXQgFrJClZwoz/
       zohtn53xEooCME2cDEVaDvBf2B9Wch1BaOFJIcJdJ8JrheUM3l6eUBuUtoy5XNHb9oJsWf
       +azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYfqs+
       hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+
       GhPZxlLDkogfSkQYZ5De6gNkwUi8qI3T0IaHjKs8tYOJU3+
```

```
    hlMSLSfUTwKn4JEx+V7FUiAUP0xI1Cxh5ySRQQoJqOA9eNj85"};
  int messageCipher = 1;
  int rc = createMessageQueue("cipher", messageCipher);
  if(rc !=  0){
    printf("Problem creating message queue");
    return 1;
  }

  FILE *fp;
  fp = fopen("teste.txt", "r");

  // Starts thread cipher side
  threadParameters dataThread;
  dataThread.fileName = "cipher";
  dataThread.fileOut = "cipher.txt";
  dataThread.eof = 0;
  int type = 1;
  createThread(&dataThread, type);

  //Message struct
  message msg;
  msg.type = 1;
  char plaintext;
  RC4 params;
  int result, countTable = 0, count = 0;
  rc4Initializer(&params, key);
  while( ( plaintext = fgetc(fp) ) != EOF ){
      rc4CipherAlgorithm(&params, plaintext);
  }
  msg.eof = 1;
  writeMessageToQueue(msg, count);
  fclose(fp);

  printf("Waiting thread ends\n");
  do{
    usleep(100);
  }while(dataThread.eof == 0);

  destroyMessageQueue();

  return 0;
}
```

A.3.4 Decifrador *RC4*

```
1 #include <functions.h>
2 #include <connections.h>
3
4 int main(int argc, char *argv[]){
5
6   createServer(argc, argv);
7   acceptClient();
8
9   char key[] = {"pAatOAgJqpdy7CECQbRojYLqy11O2W38nJm+
      Sfcz94QrNAH3EXhKFxx4t0FsjfiadGDlHkdLMG8jvgjG5jlNI72gkODO9IroRk
      +
      vy151xeBQoout9ZQ6wZktuqibySxKxGy7g0RcgfW6D7vnpV9bNfsxtOZDirJOhgQWErOq
      +GKYze25B9g0gheVEumbBpgz376VwJoHPZcFRVQw6lOc/EAaozgS1Uwh1/
      Zs1ThagZC4EP85ATVPfFJ45xNUAD1LsLqGjSOc1Y7MBU7EINwXDrDJIeKeyn+
      txKu0EoJ9QAgZyXqK4gOeUyZ1br1NMZ222Cgx+
      Q5unZA6qq8zUrHsi024wldyLJL0uqHpbHn6k3IrNrNcw45BLKBbsHf4JICaPIr825XeQb0
      +VrNUS9eSCeErUIF7iNxryaW2H67EXKds2R7pGMh+
      JlogxUUMbJ5THV5HcV2rSPLugeCWDW8MACwW2Q3d6E2t1/
      v4recq3IohzX4rBoDTz4Px91wCajdgPdJpVwP82Cyv210fqI4KBZidEgtqEzJbbh7fQvGo
      +iVGhCXs5S6XihWadbfa2XyM5hF56O3kZNFG0UITbd0/
      cClVtfA7PDNEgg2pZNjVoNvwuR+/85
      b0YhDnlX6xawNBdO6xBUuRSs5oHvmd2aTTphlxlnjAlXnfgW9Oo0ACq5YbDgo
      +npCkcE4xEyO19Pf92YORUbbk4j+
      RJHcCuDqC0cdNDN5HGSTWvJn2KbzPpdg3UAcrLc7jd4usrTuiJ7QVu6uXT6FOl2h7hMGEy
      /1kol4v5KChHq6iQ5/jkQV2049c4kN03BLceIXkyXs+y5hyVlmx/
      QBNYa3etsAiw58yMVyojsB0AWgVnUkT00IY3nw2UIgaj5ENZtcZz/
      yhRNJOwDQwG4d0n2IHZYmwgD47wXW20g1o3avaThsshBMiI6dMmcogpb2R8gtCkuGvkjwl
      /8cjGAA+wcVaOU9EndKUqT56uB4hcSQ96N63n1w07D1h6AK/
      CuELXE5wNtgPzQkgi83r+6qrY/
      rFeeO0dZAUUlR01Pyw15oGw7nmRBMBR2lak1Cjsi84I9K4JyW4t/
      LEuIvc122FB7YNCg9mRbTVMnUi3vWWcvVNur6/WQBtQllSdQ9Moo/
      mfYHA0TIyf9DBWBHtaJoWP7KvVfAgvjYyMU90ijiLDrASY8580Q1g3j6LioLEaXijfGst3
      +c7K2gTk+V1KBZEEphGXQgFrJClZwoz/
      zohtn53xEooCME2cDEVaDvBf2B9Wch1BaOFJIcJdJ8JrheUM3l6eUBuUtoy5XNHb9oJsWf
      +azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYfqs+
      hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+
      GhPZxlLDkogfSkQYZ5De6gNkwUi8qI3T0IaHjKs8tYOJU3+
      hlMSLSfUTwKn4JEx+V7FUiAUP0xI1Cxh5ySRQQoJqOA9eNj85"};
10
11  int messageDecipher = 2;
```

```
12   int rc = createMessageQueue("decipher", messageDecipher);
13   if(rc !=  0){
14     printf("Problem creating message queue");
15     return 1;
16   }
17
18   // Starts thread decipher side
19   threadParameters dataThreadDecipher;
20   dataThreadDecipher.fileName = "test";
21   dataThreadDecipher.fileOut = "cipher.txt";
22   message msgCipher;
23   msgCipher.type = 1;
24
25   int type = 2;
26   createThread(&dataThreadDecipher, type);
27   int charsRead;
28
29   //Message struct
30   message msg;
31   msg.type = 1;
32   char cipherText;
33   RC4 paramsDecipher;
34
35   rc4Initializer(&paramsDecipher, key);
36
37   rc4DecipherAlgorithm(&paramsDecipher);
38
39   destroyMessageQueue();
40 }
```