INSTITUTO MILITAR DE ENGENHARIA

1º TEN FELIPE QUITETE CURI

PROPOSTA DE SISTEMA EFICIENTE E SEGURO DE ENCRIPTAÇÃO SEQUENCIAL BASEADO NO ONE-TIME PAD

Dissertação de Mestrado apresentada ao Curso de Mestrado em Engenharia Elétrica do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Ciências em Engenharia Elétrica.

Orientador: Paulo Roberto Rosa Lopes Nunes, Ph.D.

Rio de Janeiro
2009

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80-Praia Vermelha
Rio de Janeiro-RJ CEP 22290-270

005.8 Curi, F.Q.
c975p

Proposta de sistema eficiente e seguro de encriptação sequencial baseado no One-time pad / Felipe Quitete Curi. - Rio de Janeiro : Instituto Militar de Engenharia, 2009.
113 p.: il.

Dissertação (mestrado) - Instituto Militar de Engenharia - Rio de Janeiro, 2009.

1. Criptografia. 2. Processamento de Sinais. I. Título. II. Instituto Militar de Engenharia.

CDD 005.8

INSTITUTO MILITAR DE ENGENHARIA

1º TEN FELIPE QUITETE CURI

PROPOSTA DE SISTEMA EFICIENTE E SEGURO DE ENCRIPTAÇÃO SEQUENCIAL BASEADO NO ONE-TIME PAD

Dissertação de Mestrado apresentada ao Curso de Mestrado em Engenharia Elétrica do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Ciências em Engenharia Elétrica.

Orientador: Paulo Roberto Rosa Lopes Nunes, Ph.D.

Aprovada em 09 de Fevereiro 2009 pela seguinte Banca Examinadora:

Paulo Roberto Rosa Lopes Nunes, Ph.D. do IME - Presidente

Weiler Alves Finamore, Ph.D. da PUC

Maria Thereza Miranda Rocco Giraldi, Dr. do IME

Rosângela Fernandes Coelho, Dr. ENST do IME

Rio de Janeiro
2009

A Deus, meus pais e esposa.

AGRADECIMENTOS

Primeiramente a Deus, que sempre guiou a minha vida em jornadas prósperas e repletas de ensinamento e felicidade.

Aos meus pais Walter e Edmere, com quem sempre pude contar e muito primaram pelos meus estudos.

A minha esposa, Carla, que com seu carinho, amor e companheirismo sempre me apoiou em minhas escolhas.

A minha Vó Miramar, que me introduziu no mundo da leitura e lá me manteve, a que posso atribuir o início das minhas vitórias como estudante.

A minha Vó Adyles, cuja ajuda foi fundamental nas duas semanas anteriores ao vestibular 1998/1999 do IME, semanas particularmente importantes em minha vida.

As minhas irmãs Tatiane, Ercy e Laura e meu Irmão Iuri, que sempre com muito carinho me trataram e com isso me ajudaram a chegar nesse momento.

Aos amigos e aos colegas de mestrado que me acompanharam ao longo deste caminho compartilhando as alegrias, incentivando e aconselhando nos momentos difíceis. Em especial, Amorim, Arantes, Arthur, Ballestero, Bruno, Diogo, Fábio Bandeira, Fortunato, Guilherme & Bárbara, Ottoni, Sauders e Sebastian.

Ao meu orientador, Professor Paulo Roberto Rosa Lopes Nunes pelas sugestões e apoio nos momentos difíceis deste trabalho.

Aos professores do Departamento de Engenharia Elétrica do IME, em especial, a Professora Maria Thereza Miranda Rocco Giraldi, pelo apoio oferecido durante o Curso, a Professora Rosângela Fernandes Coelho, cujo contato inicial foi fundamental para a conclusão deste curso, ao Maj Alexandre Pimentel Mendonça, que muito me ajudou com a programação necessária e ao Professor Ernesto Leite Pinto, pelas excelentes aulas ministradas.

Aos funcionários da secretaria da SE/3: Maria de Lourdes, Ronaldo e Souza pela boa vontade que sempre me foi ofertada.

Este trabalho se insere nas atividades desenvolvidas dentro do projeto *Estudo de Tecnologias para Provimento de Comunicação em Sistemas Estratégicos de Defesa (Aux Prodefesa 940/2005)*, financiado pela CAPES e pelo Ministério da Defesa.

"O impulso para descobrir segredos está profundamente enraizado na natureza humana."

John Chadwick

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

RESUMO

Este trabalho procura apresentar uma nova técnica criptográfica baseada em outra, amplamente conhecida e difundida: o one-time pad. Inicialmente é apresentada uma introdução às primeiras técnicas criptográficas utilizadas pela humanidade, para posterior explicação do criptosistema perfeito, o one-time pad, e discussão sobre a impossibilidade prática de seu uso. Após estes aspectos introdutórios é apresentado o sistema proposto, discutindo-se cada parte do projeto, em especial, a utilização da operação soma-módulo como alternativa à operação XOR e a escolha da função SHA-256 como a função não-inversível necessária para a segurança do algoritmo. Discorre-se também a respeito da geração aleatória e pseudo-aleatória de sequências de bits e os métodos utilizados para avaliá-las. Os critérios utilizados para avaliação de um sistema criptográfico são também abordados. Finalmente, programas para implementação do sistema são desenvolvidos e apresentados.

ABSTRACT

This work aims to present a new cryptographic technique that is based on an algorithm widely known and disseminated, the one-time pad. Initially, it is presented an introduction to the first cryptographic techniques used by humanity, further, an explanation of the one-time pad, a perfect cryptographic system, is presented, and the practical impossibility of its use is discussed. After these introductory aspects the proposed system is presented, an each part of the project is discussed, specially, the use of the modular sum operation as an alternative to the XOR operation and the choice of the SHA-256 function as the non-reversible function needed to make the algorithm secure. Generation of random and pseudo-random bit sequences is discussed too, as well as randomness validation techniques. Evaluation criteria for cryptographic systems are also addressed. Finally, implementation code is developed and presented.

1 INTRODUÇÃO

1.1 CONSIDERAÇÕES INICIAIS

Desde que o mundo é mundo, o ser humano busca formas de esconder segredos. Seja para comunicar-se com segurança ou guardar alguma descoberta, a *criptologia* sempre esteve presente. Guerras foram vencidas, impérios foram destruídos e vidas foram tomadas devido ao bom uso ou não desta ciência que fascina, seja na forma de simples palavras cruzadas ou complexos sistemas de encriptação.

A criptologia é dividida em dois ramos, a *criptografia* e a *criptoanálise*. A criptografia é a arte de desenvolver sistemas que possibilitem a codificação de mensagens, mantendo-as seguras e inalcançáveis a pessoas não autorizadas. Seus praticantes são os *criptógrafos*. A criptoanálise, por sua vez, é o ramo que objetiva quebrar os códigos desenvolvidos pelos criptógrafos. Usada tanto para vencer os códigos inimigos como para testar os seus próprios. Seus praticantes são os *criptoanalistas* (SCHENEIER, 1996).

Informalmente, um *algoritmo criptográfico* é uma função que transforma uma mensagem (*texto em claro*) em um *criptograma* (*texto cifrado*). Esta transformação é dependente de uma chave que pode ser de conhecimento público ou não, dependendo da natureza do algoritmo em questão. A transformação inversa é necessariamente dependente de uma chave que só deve ser conhecida pela parte autorizada a tomar conhecimento da informação cifrada (LAMBERT, 2004).

Esse esquema está evidenciado na FIGURA 1.1.

FIG. 1.1: Sistema de Criptografia

Os algoritmos criptográficos podem ser classificados como *simétricos* ou *assimétricos*. Os simétricos são os que utilizam a mesma chave para encriptar e decriptar. Os assimétricos utilizam uma chave para encriptar e outra para decriptar (SCHENEIER, 1996). Essa

dissertação trabalhará com a teoria voltada para algoritmos simétricos, então, a partir deste ponto, todas as referências serão feitas a este tipo de algoritmo.

1.2 BREVE HISTÓRICO SOBRE CRIPTOGRAFIA

Os primeiros registros sobre a criptografia datam de 4000 anos atrás, quando esta arte era utilizada de maneira bem incipiente pelos egípcios (LAMBERT, 2004). Já o primeiro aparelho criptográfico militar conhecido foi o citale espartano, ferramenta que permitia a criptografia de frases utilizando a técnica básica da *transposição*, que consiste no embaralhamento das letras utilizadas para compor a mensagem que se deseja transmitir (SINGH, 2007).

Contudo, para este trabalho, maior interesse há na segunda técnica básica de criptografia, a *substituição*. O primeiro registro de uso desta técnica foi com o poderoso e conhecido Júlio César. Este utilizava a escrita secreta com tanta frequência que Valerius Probus escreveu todo um tratado sobre cifras, o qual, infelizmente, não sobreviveu até a época atual. No entanto, graças a *As vidas dos Césares*, escrito no século II por Suetônio, tem-se uma descrição detalhada de um dos tipos de cifra de substituição (SINGH, 2007). Esta descrição também será vista na SEÇÃO 2.1.

Apesar de importante para a análise que será feita do *one-time pad*, esta cifra foi quebrada no século X D.C. através da conhecida técnica de análise de frequências. Em 1540, Leon Battista Alberti introduziu o conceito de cifra polialfabética e construiu o primeiro disco de cifras da história. Em 1586, Blaise de Vigénere publicou um tratado com uma cifra de 26 alfabetos, que ficou conhecida como cifra de Vigénere, sendo esta, por muito tempo, considerada inquebrável. Porém, em 1854, quase 300 anos depois, Charles Babbage decifra a cifra de Vigénere, utilizando um método de análise de tamanho de chave com posterior redução à vários problemas de cifra monoalfabética (análise de frequência). Em 1917, o major Joseph Mauborgne e Gilbert Vernam introduziram o conceito de chave aleatória e inventaram o *bloco de cifras de uma única vez*(*one-time pad* - Apesar de apresentado o nome traduzido, será utilizado neste trabalho o nome conhecido, em inglês), sistema totalmente seguro, porém completamente sem praticidade (KAHN, 1967).

Neste momento, ao se chegar em um algoritmo perfeito, porém impraticável, os criptográfos naturalmente voltaram seus esforços para a criação de algoritmos que misturassem ambas as técnicas, transposição e substituição. Estas técnicas aliadas a conceitos como o de redes Fiestel podem fornecer algoritmos seguros, porém não perfeitos

(SCHENEIER, 1996). O importante é notar que a partir do histórico descrito abaixo, os criptógrafos voltaram seus esforços para algoritmos que não seriam perfeitos, porém seriam impossíveis de quebrar em tempo hábil, ou seja, caso fosse testada cada chave possível (o chamado *ataque de força-bruta*) aliado à probabilidade daquela chave ser a procurada, quando se achasse a chave correta, de nada mais serviria.

Então, em 1918, Arthur Scherbius, inventor alemão, construiu a Enigma, máquina de cifras largamente utilizada pelos alemães na 2ª Grande Guerra. Prevendo os tempos difíceis, os poloneses começaram a trabalhar em cima da criptoanálise da Enigma. No final da década de 30, Marian Rejewski, matemático polonês, decifrou a Enigma e construiu uma máquina chamada bomba, que ajudava no processo de criptanálise. Em 1938, os alemães aumentaram o nível de segurança da Enigma. No início de 1940, utilizando o que já tinha sido deduzido pelos poloneses e o princípio da máquina de Turing, Alan Turing criou, em Bletchey, UK, uma máquina capaz de quebrar a nova Enigma.

No início da década de 60, Horst Feistel, alemão e criptógrafo da IBM, criou e lançou o Lúcifer, que veio a se transformar no padrão de cifragem de dados (com o nome de DES - emphData Encryption Standard) em concurso proposto pelo NIST (*National Institute of Standards and Technology*).

Em 1975, Whitfield Diffie visualizou a possibilidade da criptografia assimétrica, ou seja, a chave que encripta é diferente da que decripta. Em junho de 1976, Whitfield Diffie, Martin Hellman e Ralph Merkle inventaram o esquema para troca de chaves Diffie-Hellman-Merkle, solucionando o problema da distribuição de chaves. Em abril de 1977, Ron Rivest, Adi Shamir e Leonard Adleman encontraram enfim uma função *one-way* que difundiu rapidamente a criptografia assimétrica. Os mesmos feitos descritos neste parágrafo foram alcançados anteriormente por Malcolm Willianson em 1969, James Ellis em 1973 e Clifford Cocks em 1974, pesquisadores ingleses; porém não foram publicados por questões de segurança nacional.

Em junho de 1991, Phil Zimmermann disponibilizou um *software* na internet, o PGP, que juntava as funcionalidades da criptografia simétrica e da criptografia de chave-pública, utilizando pela primeira vez, dessa maneira, a assinatura digital.

Em 1998, surgiu o algoritmo Rijndael, que substituiria o antigo padrão (DES), com o nome de AES (*Advanced Encryption Standard*), em novo concurso proposto pelo NIST.

Na Europa, foi criado um projeto de pesquisa multinacional voltado para a Tecnologia da Informação, o *New European Schemes for Signatures, Integrity and Encription* - NESSIE. O projeto, apesar de não ser um instituto de padrões, se propõe a ser uma inter-

face entre a comunidade de pesquisa e a comunidade de usuários, testando e comparando algoritmos antes que estes sejam propostos como padrões (PRENEEL, 2003).

Para o século XXI, o futuro da criptografia e da criptanálise parece realmente se encontrar na física quântica, pois é nesse campo que os maiores esforços estão concentrados. Se tais esforços forem recompensados, a criptanálise não mais terá dificuldade computacional para quebrar a criptografia atual e, por outro lado, a criptografia terá inventado um padrão mais seguro.

1.3 MOTIVAÇÃO

Da seção anterior pode-se então concluir que desde muito tempo, esta ciência, a criptologia, é um dos principais focos de cada governo existente no nosso planeta. Uma forma segura de guardar segredos é algo que sempre foi necessário e é cada vez mais perseguido e vital para a segurança das nações.

Obviamente, essa situação não é diferente para o Brasil. Como país em crescente desenvolvimento, com grande potencial científico e enormes riquezas naturais, é evidente o interesse estrangeiro em nosso país. Para se ter o controle de todas essas variáveis são necessários diversos projetos em vários campos científicos. Alguns destes, como por exemplo, o SIVAM (Sistema de Vigilância da Amazônia) e o SIPAM (Sistema de Proteção da Amazônia) já são executados pelo Exército Brasileiro. Para uma maior eficácia destes projetos é necessário evitar que os dados coletados estejam a disposição de qualquer pessoa ou entidade, sendo esta proteção feita em diversos quesitos, sendo um deles o quesito segurança da informação, mais precisamente na criptografia.

Porém, como dito anteriormente, todas as formas conhecidas atualmente (vindouras a partir do *one-time pad*) baseiam-se na suposição que o oponente não possui tempo hábil suficiente para quebrar os algoritmos por força-bruta. Contudo, com o contínuo aumento de velocidade computacional que as novas tecnologias vêm apresentando, essa suposição pode estar ameaçada.

Cada vez que se pensa neste problema, é normal que se pense na descoberta feita por Joseph Mauborgne e Gilbert Vernam em 1917, o *one-time pad* (SCHENEIER, 1996). *One-time pad* é um esquema de encriptação perfeito, independentemente de qualquer poder computacional - ver CAPÍTULO 2. Este esquema para ser perfeito, porém, exige condições atualmente impraticáveis. É inspirado neste tema, *one-time pad*, que esta dissertação pretende trabalhar, propondo um novo sistema seguro, prático e eficaz, contribuindo assim para a eficácia dos sistemas criptográficos utilizados no Brasil.

1.4 OBJETIVOS

São os seguintes os objetivos deste trabalho:

- Estudar variações de sistemas criptográficos que tenham como base o one-time pad. Avaliar as potencialidades e fraquezas destes sistemas e sua imunidade aos diferentes tipos de ataques criptográficos;
- Estudar o problema de geração de sequências pseudo-aleatórias e verdadeiramente aleatórias, para que estas últimas sejam utilizadas no sistema proposto;
- Desenvolver um sistema onde parte do canal é utilizada para enviar, de forma segura, a sequência aleatória utilizada na encriptação;
- Implementar o sistema desenvolvido; e
- Analisar este sistema.

1.5 ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação está organizada em 7 (sete) capítulos, incluindo esta introdução, e 1 (um) apêndice.

O Capítulo 2 dá os conceitos básicos necessários ao entendimento do *One-time pad*.

O Capítulo 3 descreve o sistema proposto.

O Capítulo 4 apresenta os conceitos necessários ao entendimento da análise que foi feita.

O Capítulo 5 mostra os resultados obtidos na análise do sistema proposto e seus significados.

O Capítulo 6 apresenta como foi feita a implementação do sistema.

O Capítulo 7 finaliza o trabalho com conclusões a respeito do sistema proposto e apresenta propostas de estudos.

2 ONE-TIME PAD

2.1 A CIFRA DE SUBSTITUIÇÃO MONOALFABÉTICA

Uma cifra de substituição monoalfabética é um procedimento criptográfico simples que consiste em substituir cada letra do alfabeto por outra, evitando, obviamente, que duas letras diferentes levem à mesma substituição.

2.1.1 A CIFRA DE SUBSTITUIÇÃO MONOALFABÉTICA POR DESLOCAMENTO

A primeira cifra de substituição utilizada para fins militares foi a cifra de deslocamento de César, ou simplesmente a cifra de César. Ela consistia em um deslocamento de n casas do alfabeto com a subsequente substituição(SINGH, 2007). Por exemplo, considerando o alfabeto inglês, com 26 letras, para um $n = 5$, tería-se que a letra A seria representada pela letra F, a letra B seria representada pela letra G, a letra C seria representada pela letra H e assim por diante. Com esta configuração, a mensagem "ESTOU NA PONTE" seria representada pelo texto cifrado "JXYTZ SF UTSYJ". Para este exemplo ,então, a chave seria exatamente o valor de n, pois conhecido este valor, é trivial achar a mensagem original. Na FIGURA 2.1 pode ser visualizado um exemplo para $n = 3$.

Alfabeto original	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Alfabeto cifrado	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

FIG. 2.1: Exemplo de deslocamento para n=3

2.1.2 A CIFRA DE SUBSTITUIÇÃO MONOALFABÉTICA GERAL

Uma fraqueza óbvia do uso da técnica descrita acima é o número de encriptações possíveis. Para um alfabeto de m letras o número de encriptações possíveis vale $m - 1$. Com os modernos meios computacionais, testar todas as possibilidades seria uma tarefa muito simples.

Uma variação simples, porém eficaz, que poderia ser proposta, seria a possibilidade da substituição ocorrer por qualquer rearranjo do alfabeto, sem que houvesse qualquer lógica de deslocamento nele contida. O número de possibilidades aumentaria consideravelmente

(em um alfabeto de m letras, teria-se $(m-1)!$ possibilidades), e teria-se também uma chave mais complexa, que desta vez não poderia ser apenas um deslocamento, mas teria que indicar a ordem das m letras do alfabeto, rearranjadas no alfabeto cifrado (SINGH, 2007). A FIGURA 2.2 mostra essa técnica.

Alfabeto original	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Alfabeto cifrado	J	L	P	A	W	I	Q	B	C	T	R	Z	Y	D	S	K	E	G	F	X	H	U	O	N	V	M

FIG. 2.2: Rearranjo do alfabeto

2.2 ANÁLISE DE FREQUÊNCIAS

Conforme visto na SEÇÃO 1.2, a cifra monoalfabética funcionou por quase 8 séculos antes de ser quebrada. Só no século X D.C é que foi proposta a chamada técnica de análise de frequências, responsável pela quebra desta cifra. A análise de frequência consiste em analisar a frequência com que cada caracter de uma linguagem aparece e com isso diminuir o trabalho de força bruta necessário à quebra de determinado texto cifrado (SINGH, 2007).

Primeiramente, analisa-se o texto cifrado e verifica-se a taxa de ocorrência de cada letra. Após isso esta tabela é comparada com uma tabela como a TABELA 2.1 que nos mostra a taxa de frequência de cada caracter da linguagem utilizada para escrever a mensagem que foi ciptografada. A tabela citada mostra a taxa de frequência da língua inglesa[1].

É natural que as letras mais frequentes no texto cifrado correspondam as letras mais frequentes daquela linguagem específica. Por exemplo, se ao analisar o texto cifrado for constatado que as letras mais frequentes são X, F e J com respectivamente as porcentagens 11.95, 8.98 e 8.37, provavelmente estas letras estão representando as letras e, t e a. Partindo dessa premissa, e considerando peculiaridades da linguagem, chega-se à mensagem original. Por exemplo, na língua inglesa é muito comum a aparição de palavras com duas letras e em seguida, como em *been* ou emphthree. Então se o texto cifrado tiver algumas aparições de XX, é bem provável que a letra X realmente represente a letra e. Outro exemplo é que a letra t raramente é vista antes ou depois de b, d, g, j, k, m, q ou v (SINGH, 2007).

Obviamente, o mesmo procedimento pode ser feito para mensagens escritas em português. Para isso deve se analisar a TABELA 2.2 que mostra a taxa de frequência de

[1]Retirado de (BEUTELSPACHER, 1994)

Letra	Porcentagem
a	8,167
b	1,492
c	2,782
d	4,2533
e	12,702
f	2,228
g	2,015
h	6,094
i	6,966
j	0,153
k	0,772
l	4,025
m	2,406
n	6,749
o	7,507
p	1,929
q	0,095
r	5,987
s	6,327
t	9,056
u	2,758
v	0,978
w	2,360
x	0,150
y	1,974
z	0,074

TAB. 2.1: Tabela de frequências da língua inglesa

cada caracter da língua portuguesa[2].

2.3 SEGURANÇA PERFEITA EM UM CRIPTOSISTEMA

2.3.1 DEFINIÇÃO DE CRIPTOSISTEMA

Antes de mostrar-se a definição de segurança perfeita, deve-se mostrar a definição formal de Criptosistema, mas precisamente a que interessa neste trabalho, a simétrica. Essa definição vem de Claude E. Shannon (SHANNON, 1949), não apenas o pai da Teoria da Informação, mas também da criptologia moderna (BEUTELSPACHER, 1994).

Seja a FIGURA 2.3. Nela está representado um conjunto de textos em claros (M1, M2 e M3), um conjunto de textos cifrados (C1, C2, C3 e C4) e um conjunto de transformações

[2]Retirado de (VICKISOFT, 2004)

Letra	Porcentagem
a	14,63
b	1,04
c	3,88
d	4,99
e	12,57
f	1,02
g	1,30
h	1,28
i	6,18
j	0,4
k	0,02
l	2,78
m	4,74
n	5,05
o	10,73
p	2,52
q	1,20
r	6,53
s	7,81
t	4,34
u	4,63
v	1,67
w	0,01
x	0,21
y	0,01
z	0,47

TAB. 2.2: Tabela de frequências da língua portuguesa

(representadas pelas linhas F1 e F2).

Um *Criptosistema Simétrico* consiste de um conjunto M de textos em claros, um conjunto C de textos cifrados e um conjunto F de transformações inversíveis de M em C. Pode ser representado por S=(M,C,F), e já que todas as transformações são inversíveis, não pode acontecer de duas flechas indicativas da mesma transformação chegarem em um mesmo texto cifrado. Por exemplo, não poderia acontecer de as flechas que representam a transformação F1 em M1 e M3 chegarem ao mesmo texto cifrado C3, pois ao inverter a transformação não sabería-se se aquele C3 teve como origem M1 ou M3.

Atentando para isso e que a notação |X| representa o número de elementos do conjunto X, chega-se a primeira conclusão:

$$|M| \leq |C|$$

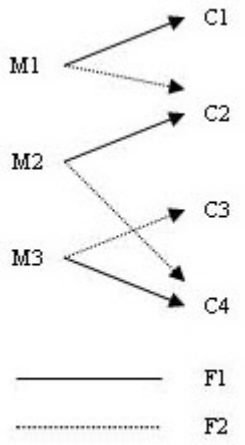

FIG. 2.3: Criptosistema

2.3.2 PROBABILIDADES A PRIORI E A POSTERIORI

Sabendo-se agora o que significa um criptosistema, deve-se partir para o estudo do que o torna perfeitamente seguro. Perfeição em segurança significa a *impossibilidade* de um criptoanalista aumentar o seu conhecimento sobre o criptosistema, por mais que este possua potência computacional ou *know-how* (BEUTELSPACHER, 1994).

Por exemplo, se há o conhecimento de que um adversário utiliza a cifra de César sempre com uma mesma chave, quanto mais textos cifrados forem capturados, mais informações (frequências) serão obtidas sobre a ocorrência das letras cifradas. Com isso, chegar-se-á em estimativas de probabilidades mais próximas das reais (aquelas conhecidas de um determinado alfabeto, como a língua inglesa ou portuguesa) e, finalmente, se estará mais perto de quebrar qualquer mensagem deste sistema. Através destas colocações pode-se concluir obviamente que a cifra de César que utiliza sempre a mesma chave não é um sistema perfeitamente seguro.

A partir deste exemplo e generalizando para criptosistemas que o texto claro não é necessariamente representado por apenas uma letra, pode-se definir as chamadas probabilidades *a priori* e probabilidade *a posteriori*.

Seja um criptosistema $S = (M, C, F)$. Seja $\mu \in M$ e $\gamma \in C$, sendo γ a cifra de μ. A probabilidade *a priori* de μ, ou seja, a probabilidade de qualquer texto em claro μ ocorrer, é representada por:

$$p(\mu)$$

Após a encriptação, ao analisar-se o texto cifrado, procura-se qualquer informação adicional que possa tornar a quebra daquele sistema mais fácil. Após esta análise, a

probabilidade *a posteriori* de μ, ou seja, a probabilidade de μ ocorrer quando já se conhece γ, pode ser representada por:

$$p_\gamma(\mu) = P[\mu|\gamma]$$

Como o conceito de perfeição passa pelo fato do criptoanalista não conseguir obter qualquer informação de μ ao analisar γ, pode-se conclur que em um criptosistema perfeito tem-se a seguinte relação:

$$p_\gamma(\mu) = p(\mu)$$

2.3.3 CRITÉRIOS DE PERFEIÇÃO

Para se reconhecer um Criptosistema perfeito, é necessário que este satisfaça os três critérios[3] descritos a seguir.

Critério 1. *Se S tem segurança perfeita então qualquer texto em claro do sistema pode ser transformado, usando uma chave de S, em qualquer texto cifrado de S.*

Ou seja, para cada texto em claro existe uma chave que o transforma em qualquer um dos textos cifrados possíveis. Isso pode ser provado se atentar-se para o fato de que partiu-se da premissa que ele é perfeito e consequentemente $p_\gamma(\mu) = p(\mu)$. Com isso, não pode haver um texto cifrado que não possa ser transformado em um texto em claro, o que faria a premissa cair em contradição, já que tería-se para um certo μ, sendo $p(\mu) > 0$, $p_\gamma(\mu) = 0$ e consequentemente $p_\gamma(\mu) \neq p(\mu)$.

Critério 2. *Se S é perfeito, então:*

$$|F| \geq |C| \geq |M|$$

A segunda desigualdade já foi esclarecida em 2.3.1, faltando a prova da primeira. Do primeiro critério, tem-se que μ pode ser transformado em qualquer texto cifrado por meio de uma transformação e ao mesmo tempo é necessário uma transformação diferente para transfromá-lo em cada texto cifrado possível. Daí conclui-se que o número de transformações necessárias é ao menos tão grande quanto o número possível de textos cifrados.

[3]Retirados de (BEUTELSPACHER, 1994)

Critério 3. *Seja S=(M,C,F) um criptosistema com:*

$$|F| = |C| = |M|$$

Onde todas as chaves ocorrem com a mesma probabilidade e qualquer texto em claro μ possui exatamente uma transfromação para cada texto cifrado possível. Se essas condições ocorrerem, S é perfeito.

Um exemplo de um criptosistema assim pode ser visto na FIGURA 2.4.

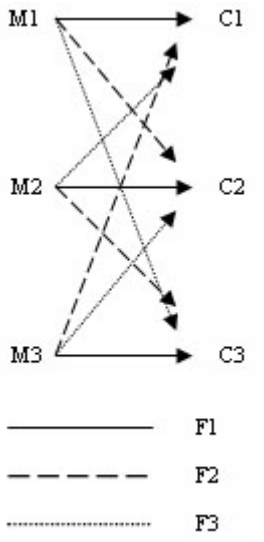

FIG. 2.4: Criptosistema perfeito

2.4 O ONE-TIME PAD

Nesta seção será apresentado o *One-time pad*, criptosistema perfeito que serve de inspiração para o algoritmo desenvolvido neste trabalho.

2.4.1 ESQUEMA GERAL

O *One-time pad* é um procedimento simples, porém perfeito, representado na FIGURA 2.5.

A operação $\oplus$ representa uma soma módulo caracter a caracter, assumindo que os caracteres são letras. Para realizar esta soma, basta atribuir um valor para cada letra do alfabeto, que, geralmente, significa atribuir o valor 1 (um) à letra A, o valor 2 à letra B e assim por diante até chegar na letra Z que terá o valor 26 (trabalhando com o quantitativo de letras existente na língua inglesa, que é o utilizado mundialmente) e quando

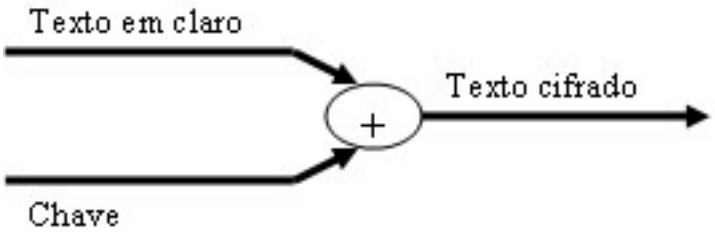

FIG. 2.5: One-time pad

ocorrer de chegar ao final (26), recomeçar a contagem. Por exemplo, $A(1)+G(7) = H(8)$ e $W(23) + E(5) = B(2)$. Notando-se que $2 = 28$(soma das letras W e E) $- 26$(tamanho do alfabeto), percebe-se facilmente que quando a *soma* das letras ultrapassar 26, basta diminuir de 26 para achar o valor correspondente à letra cifrada. Para obter a mensagem original, basta pegar o texto cifrado e diminuir da chave, seguindo os mesmos procedimentos. Por exemplo, a mensagem "SALVEME" criptografada com a chave "HJEMPSB" resultaria no texto cifrado "AKQIUFG". Interceptada esta mensagem, o inimigo poderia optar por testar TODAS as chaves possíveis. Sabendo ele que a encriptação é feita utilizando-se o *One-time pad*, que tem a chave do mesmo tamanho que a mensagem em claro, que resulta em uma mensagem cifrada de também mesmo tamanho, concluiria ele que a nossa possui 7 caracteres. Ao começar os testes, por exemplo, pela chave "ABCDEFG", encontraria ele a mensagem "ZINEPZZ", que não faz qualquer sentido. Porém, mesmo continuando com este procedimento, nada se concluiria. Quando ele testasse a chave utilizada, "HJEMPSB", ele encontraria a mensagem real, "SALVEME", que é obviamente coerente. Porém, quando ele testasse a chave "WFHKPSB" ele encontraria a mensagem "DEIXEME", que também é uma mensagem coerente, porém não passa a informação desejada. Fazendo uma rápida análise, pode-se perceber que qualquer mensagem de 7 letras pode ser encontrada por este método, ou seja, após realizar TODOS os testes possíveis, o inimigo terá sobre sua mesa todas as mensagens coerente de 7 letras que podem ser feitas e só terá um caminho para selecionar a que ele julga ser a real mensagem enviada: a probabilidade de cada uma destas chaves ocorrer.

Deste fato, conclui-se que a geração da chave utilizada deve ser *verdadeiramente* aleatória, mais especificamente, as probabilidades de cada uma das 26 letras possíveis para cada caracter da chave devem ser iguais e os caracteres devem ser estatisticamente independentes. Para cada nova mensagem deve ser gerada uma nova chave, pois se o criptoanalista tiver múltiplos textos cifrados cujas chaves foram repetidas, ele pode reconstruir o texto em claro (SCHENEIER, 1996). Isso porque com os vários textos cifrados

de chaves repetidas, ele poderá obter novas informações probabilísticas sobre a ocorrência dos caracteres, obtendo uma probabilidade a *posteriori* diferente da probabilidade *a priori*. E como visto na SEÇÃO 2.3.2, quando isso acontece tem-se que o criptosistema não tem perfeita segurança e evidentemente pode ser quebrado. Este é o motivo para que no **Critério 3**, na SEÇÃO 2.3.3, exige-se para a perfeição do criptosistema, a geração de chaves com mesma probabilidade.

Pode-se mostrar que este sistema, o *one-time pad*, é perfeito. É trivial mostrar que $|F| = |C| = |M|$, visto que cada caracter do alfabeto possui exatamente n caracteres cifrados possíveis, correspondentes a n transformações possíveis, onde n é o tamanho do alfabeto. Isto adicionado à exigência de produção de chaves aleatórias igualmente prováveis, torna o *one-time pad* um sistema criptográfico perfeito.

2.4.2 OPERAÇÃO XOR

Hoje em dia, no mundo informatizado, não se utiliza mais diretamente os caracteres de um alfabeto na representação de uma mensagem, pois tornou-se mais conveniente trabalhar com bits. Quando insere-se uma letra ou qualquer caracter em um computador, este é interpretado como um conjunto de bits. Assim, cada caracter possui uma representação única que o especifica. Por exemplo, a letra *a* é representada pelo conjunto de bits 1100001 e o caracter & é representado pelo conjunto 0100110.

Obviamente, se o alfabeto é representado por bits, a chave também o será. Com isso, a operação realizada agora no *one-time pad* não será mais uma operação caracter a caracter e sim uma operação bit a bit e será representada a partir deste ponto pelo símbolo $\oplus$, que representa uma soma módulo 2, mais conhecida como a operação lógica *ou exclusivo(XOR)*. Na TABELA 2.3 pode-se verificar os resultados possíveis nesta operação.

$\oplus$	0	1
0	0	1
1	1	0

TAB. 2.3: Operação XOR

Ou seja, $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$ e $1 \oplus 1 = 0$.

É fácil mostrar que se $A \oplus B = C$, então $C \oplus B = A$ e $C \oplus A = B$. Daí se A for a mensagem em claro, B a chave utilizada para encriptar este texto e C o texto cifrado, é possível obter a mensagem em claro utilizando apenas o texto cifrado e chave:

$$A = (A \oplus B) \oplus B$$

2.4.3 A DIFICULDADE DE SE USAR O ONE-TIME PAD

Infelizmente, a perfeição do *One-time pad* vem acompanhada de algumas dificuldades que tornam o seu uso ineficiente. A exigência da chave ser aleatória, só poder ser usada uma única vez e o seu tamanho ser do tamanho da mensagem em claro trazem sérios impedimentos para a utilização do *one-time pad*, pois torna necessário uma produção em larga escala de números verdadeiramente aleatórios. Como nos tempos atuais os canais de comunicação trabalham na casa de Gbps, existe um déficit entre a velocidade desejável para a produção de números aleatórios e a velocidade que efetivamente é atingida, pois os geradores de números verdadeiramente aleatórios ainda trabalha na casa dos 2 Mbps (BORGES JR., 2008).

Esse problema pode ser minimizado com a pré-produção e armazenamento de números aleatórios para posterior uso, porém neste caso deverá haver uma segurança especial para os dispositivos de armazenamento de dados que conterão as futuras chaves, como políticas de armazenamento, acesso e distribuição destas, pois caso um destes dispositivos(que nada mais são que os impressos ou manuscritos utilizados na segunda grande guerra) seja interceptado pelo inimigo, este só precisará testar as chaves que estão contidas no dispostivo e não todo o espaço amostral existente para as chaves[4].

[4]Por exemplo, caso esteja se trabalhando com 512 bits, o espaço amostral contém 2^{512} possibilidades

3 PROPOSTA DE SISTEMA

3.1 SISTEMA

O Sistema que será agora proposto é o escopo principal deste trabalho. Inspirado no *one-time pad*, pretende-se que este trabalho apresente um método rápido e simples de criptografar uma sequência utilizando uma chave longa. Essa chave longa garante a dificuldade de uma criptoanálise por força bruta ser eficaz contra este método. A proposta pode ser visualizada na FIGURA 3.1.

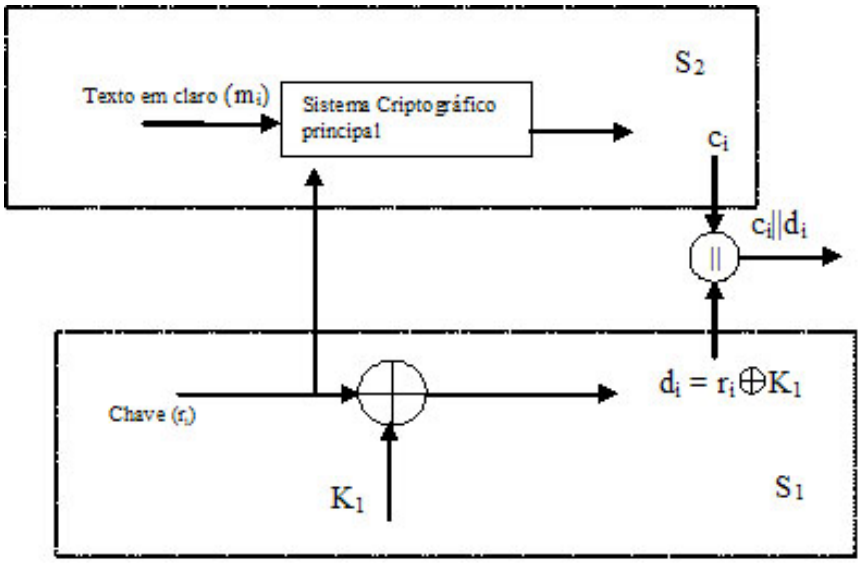

FIG. 3.1: Visão Geral do sistema

O sistema possui dois sub-sistemas S_1 e S_2. O primeiro, S_1, tem como objetivo criptografar a sequência de chaves r_i que será utilizada por S_2. r_i e d_i são números inteiros que representam sequências binárias com o mesmo comprimento. A operação que r_i sofre com a máscara K_1 resulta em d_i. K_1 é a verdadeira chave desse sistema, posto que para o seu efetivo funcionamento deve ser mantida secreta, pois se for descoberta, o sistema resultará quebrado.

Como d_i pode ser interceptado no canal de comunicação, para que K_1 não possa ser estimado é necessário que a distribuição de probabilidade de d_i seja idêntica a de r_i, o que acontecerá se r_i for constituida por símbolos estatisticamente independentes e uniformemente distribuídos, assumindo que cada valor de r_i mapeia em um único valor de d_i e vice-versa, condição esta necessária para que r_i possa ser recuperado no decodificador.

As operações soma-módulo e XOR são inversíveis e rápidas, e serão utilizadas em S_1. (Ver SEÇÃO 5.3). O sistema S_1 é um sistema criptográfico perfeito, desde que a sequência r_i seja verdadeiramente aleatória e inacessível a elementos externos ao sistema.

O subsistema S_2 é a parte referente à encriptação propriamente dita do texto em claro m_i. S_2, o sistema criptográfico principal, indicado na FIGURA 3.1, pode ser qualquer sistema criptográfico conhecido, tal como o DES, AES, entre outros. Porém, deseja-se que S_2 seja, em termos de simplicidade, o mais próximo possível de S_1. A uma primeira vista, essa simplicidade pode ser obtida utilizando-se outra operação soma-módulo (ou xor), resultando então no sistema mostrado na FIGURA 3.2.

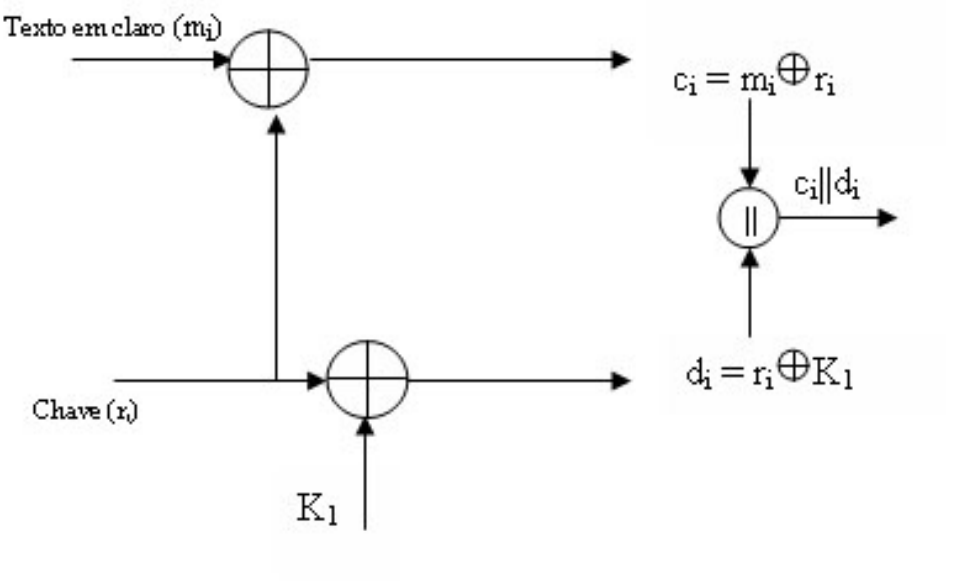

FIG. 3.2: Sistema simplificado

Como deve-se considerar que o adversário tem acesso ao canal de comunicação, deve-se então considerar que ele possui também acesso a c_i e d_i. Tendo conhecimento de c_i, se o adversário inserir uma mensagem conhecida em m_i, ele poderá deduzir o r_i utilizado[5]. Conhecido r_i e d_i, pode-se facilmente deduzir o valor de K_1 e com isso quebrar o algoritmo. Ou seja, o uso conjunto de S_1 e S_2 pode resultar em informação sobre r_i ou K_1. Dessa maneira, verifica-se a necessidade de isolar S_1 de S_2.

Este isolamento pode ser feito por uma função inversível ou pelo menos de difícil inversão. Estas funções de difícil inversão, as chamadas *one-way functions*, são relativamente fáceis de computar. Ou seja, para estas, dado um x é simples computar $f(x)$, porém, dado $f(x)$ é muito difícil computar x. Difícil significa que o poder computacional

[5]Este ataque é conhecido como ataque de texto em claro em conhecido e será melhor explicado na SEÇÃO 4.3

a disposição não será suficiente para esta operação (SCHENEIER, 1996). Então, a necessidade de garantir que a função não era inversível, fez com que fosse escolhida para esse trabalho uma função *hash one-way*, a SHA-256, pois, para este tipo de função, dado o valor *hash*, é inviável computar a origem daquele valor (SCHENEIER, 1996). O ponto negativo desta função é a provável perda de eficiência, já que sua saída será geralmente reduzida em relação à entrada. Uma apresentação mais detalhada sobre esta função pode ser vista na SEÇÃO 3.3. Daí, finalmente tem-se a proposta final na FIGURA 3.3.

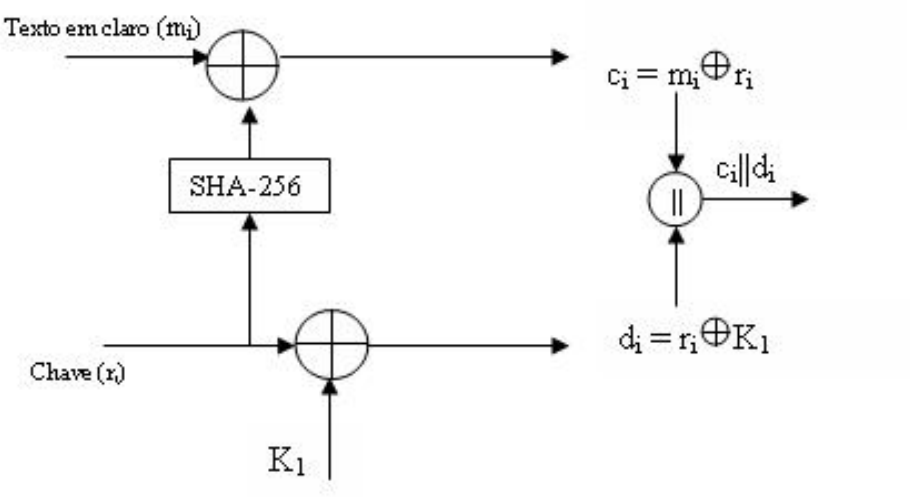

FIG. 3.3: Sistema proposto

Para decriptar, no receptor, será utilizado o esquema da FIGURA 3.4

Conforme pode-se observar na FIGURA 3.3, a chave gerada é utilizada em duas operações. Na primeira, sofrerá a ação de uma máscara K_1 e na segunda, sofrerá ação da função SHA-256 para posteriormente fazer a encriptação propriamente dita da mensagem em claro. Após ambas operações, os dois resultados, c_i e d_i, são concatenados e enviados por um canal de comunicação.

No receptor, estes são desconcatenados. Na sequência, d_i sofre ação da máscara K_1 e dessa operação obtém-se r_i. Com este r_i e utilizando novamente a função SHA-256, obtém-se a mensagem em claro m_i que se desejou transmitir.

Percebe-se que este sistema, por carregar a chave junto a ele, não possui o problema de geração e armazenamento de chaves, inerente ao *one-time pad*.

Neste Capítulo serão apresentados os três componentes deste sistema[6]: os responsáveis pela encriptação, a função SHA-256 e a máscara K_1. As operações de encriptação podem ser feitas utilizando tanto operações de soma módulo quanto operações XOR.

[6]Além da chave e do texto em claro

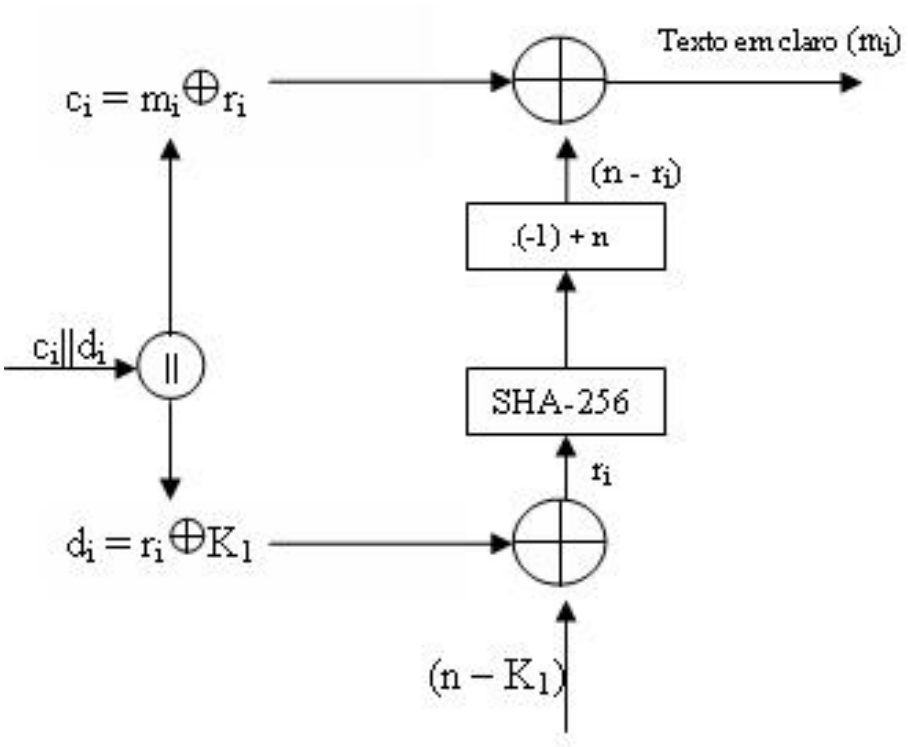

FIG. 3.4: Decriptação para o Sistema proposto

Para tanto, ambas serão apresentadas e comparadas. A função SHA-256 é uma função cuja necessidade surgiu no curso deste trabalho. Esta será, então, também apresentada e discutida neste Capítulo. A forma como r_i será gerada, será discutida no CAPÍTULO 6.

3.2 ARITMÉTICA MODULAR

3.2.1 CONGRUÊNCIA

A operação soma módulo n entre os inteiros A e B (restringindo-se aos inteiros) tem como resultado um inteiro positivo menor que n que, somado a um determinado múltiplo de n, iguala o valor obtido quando A e B são somados algebricamente. O resultado de uma soma módulo, portanto, possui um limite máximo. Generalizando, qualquer operação usual definida no conjunto dos inteiros, quando dita módulo n, resulta em um inteiro não-negativo menor que n. Pode-se estabelecer então uma relação entre um inteiro qualquer e um inteiro não-negativo menor que n. Utiliza-se a seguinte notação:

$$a \equiv b(mod\ n),$$

e diz-se que a e b são congruentes módulo n (COUTINHO, 2005).

Por exemplo, se for desejoso representar o número 231 como uma hora do dia, este será representado por 15, pois:

$$231 \equiv 15(mod\ 24)$$

O número 231 é o número que se deseja representar, 15 é o número 231 já representado e 24 é o número de horas em um dia. Diz-se que 231 e 15 são congruentes módulo 24.

Com base nas informações acima, pode-se chegar a uma maneira simples de calcular números congruentes:

- $a = k \cdot n + b$, onde a representa os números congruentes que serão calculados, $k \in Z$, n é o módulo e b é o número para o qual se quer encontrar congruentes. Por exemplo, os diversos congruentes de 15 (mod 24), são 39 ($1 \cdot 24 + 15$), 63 ($1 \cdot 24 + 15$), 87 ($1 \cdot 24 + 15$) e assim por diante.

E desta, pode-se chegar a duas forma de se verificar congruência:

- Calcula-se a divisão entre a e n e pega-se o resto. O número a é congruente ao resto módulo n. No exemplo acima, $\frac{231}{24}$ é igual 9 com resto 15. Ou seja, 231 é congruente a 15 módulo 24 ou,

- Subtrai-se b de a. Se o número resultante for um múltiplo de n, b e a são congruentes. Voltando ao exemplo acima, tem-se que $231 - 15 = 216$, que é múltiplo de 24. Mais uma vez conclui-se a congruência de 231 a 15 módulo 24.

Os itens vistos serão demonstrados na próxima seção.

3.2.2 DEFINIÇÃO FORMAL DE CONGRUÊNCIA

Da definição de múltiplos tem-se que dois números são múltiplos desde que sua divisão resulte em um número inteiro, o que permite concluir que é perfeitamente viável trabalhar com números negativos na aritmética modular. Da mesma maneira que 39, 63 e 231 são congruentes a 15 módulo 24, também o são -9 e -33 - esses valores são facilmente encontrados utilizando-se a equação $a = k \cdot n + b$.

Visto isso, será definido agora o subconjunto de Z pela relação de congruência módulo n. Este grupo, cuja notação é Z_n, será chamado de conjunto dos inteiros módulo n (COUTINHO, 2005). Por definição, sabe-se que este grupo é subconjunto de Z. Seja

$a \in Z$, a classe $\overline{a}$ é formada pelos $b \in Z$ que satisfazem $b - a$ múltiplo de n; isto é $b - a = k \cdot n$, para algum $k \in Z$. Pode-se assim descrever a classe a na forma:

$$\overline{a} = \{a + k \cdot n \mid k \in Z\}$$

Uma observação importante que pode ser ressaltada é que $\overline{0}$ é o conjunto dos múltiplos de n.

Continuando, se $a \in Z$, então pode-se dividí-lo por n, obtendo-se q e r, tal que:

$$a = n \cdot q + r,\ 0 \leq r \leq n - 1$$

Logo $a - r = n \cdot q$ é um múltiplo de n. Portanto $a \equiv r\ (mod\ n)$. Isto é, um inteiro qualquer é congruente módulo n a um inteiro no intervalo que vai de 0 a $n - 1$. Resumindo:

$$Z_n = \{\overline{0}, \overline{1}, ...\overline{n-1}\}$$

Por exemplo, o subconjunto das horas pode ser representado por:

$$Z_{24} = \{\overline{0}, \overline{1}, ...\overline{23}\}$$

As classes 0, 1, ...23 são representadas por:

$$\overline{0} = \{0 + k \cdot 24 \mid k \in Z\}$$

$$\overline{1} = \{1 + k \cdot 24 \mid k \in Z\}$$

...

$$\overline{23} = \{23 + k \cdot 24 \mid k \in Z\}$$

E como pode-se imaginar este subconjunto Z_n? Imagine todos os pontos de Z marcados ao longo de uma reta horizontal, de uma em uma unidade. Imagine agora que enrola-se esta reta em uma circunferência colando o ponto n sobre o ponto 0. Como a reta é infinita, continuar-se-á enrolar a reta. Desta maneira os pontos cujas coordenadas são múltiplos de n coincidirão com o ponto 0. A imagem geométrica de Z_n é, portanto, a de uma circunferência, onde estão marcados n pontos equidistantes. Cada ponto corresponde a uma das classes de Z_n (COUTINHO, 2005).

3.2.3 OPERAÇÕES BÁSICAS DA ARITMÉTICA MODULAR

Visto o conteúdo da SEÇÃO 3.2.2, pode-se agora definir 3 das 4 operações básicas da aritmética modular para as classes em questão:

a) $\overline{a+b} = \overline{\overline{a}+\overline{b}}$

b) $\overline{a-b} = \overline{\overline{a}-\overline{b}}$

c) $\overline{a \cdot b} = \overline{\overline{a} \cdot \overline{b}}$

E destas pode-se concluir[7]:

a) $(a+b)\ mod\ n = ((a\ mod\ n) + (b\ mod\ n))\ mod\ n$

b) $(a-b)\ mod\ n = ((a\ mod\ n) - (b\ mod\ n))\ mod\ n$

c) $(a \cdot b)\ mod\ n = ((a\ mod\ n) \cdot (b\ mod\ n))\ mod\ n$

A quarta operação básica, a divisão, não será utilizada neste trabalho, porém para um rápido esclarecimento, na aritmética modular, nem toda divisão é possível de se realizar. Para entender isso, comece por esse exemplo: quando normalmente se quer resolver a equação $7 \cdot x = 21$, divide-se 21 por 7 e acha-se a resposta, 3. O que na verdade foi feito foi multiplicar os dois lados da equação pelo inverso de 7. Daí, resultam duas divisões, de 7 por 7, que deixa a variável sozinha do lado esquerdo da equação e de 21 por 7, que é, consequentemente, a resposta. E por que foi escolhido o inverso de 7 para multiplicar os dois lados da equação? Exatamente porque ao multiplicar-se um número pelo seu inverso o resultado vale 1. Na operação de divisão modular é feita a mesma sequência de passos.

Por exemplo, para resolver a equação $7 \cdot x \equiv 3(mod\ 15)$, precisa-se multiplicar ambos os lados da equação pelo inverso de 7 em Z_{15}. Ou seja, o primeiro problema é descobrir o inverso de 7 em Z_{15}. Como dito no parágrafo anterior, o raciocínio é igual. O inverso de 7 em Z_{15} é o número que multiplicado por ele dá 1 em Z_{15}. Neste exemplo, tem-se que $-2 \cdot 7 = -14$ e $-14(mod\ 15) = 1$. Logo -2 é o número que multiplicado por 7 é igual a 1. Como $-2 \equiv 13(mod\ 15)$, tem-se que o inverso de 7 vale 13. E realmente $7 \cdot 13 = 91 = 15 \cdot 6 + 1$ ou $91 \equiv 1(mod\ 15)$. Com isso, a equação fica $x \equiv 13 \cdot 3 \equiv 39 \equiv 9(mod\ 15)$. O número 9, então, é a solução da equação.

Por outro lado, a equação $2 \cdot x \equiv 7(mod\ 14)$ é mais complicada de resolver, pois 2 não tem inverso em Z_{14}. O método de multiplicar pelo inverso não funcionará então. Terá que ser usado o método da tentativa e erro, que nem sempre levará a uma resposta.

[7] Retirado de (SCHENEIER, 1996)

3.2.4 CASO PARTICULAR: A FUNÇÃO XOR

A função XOR, como já apresentada na SEÇÃO 2.4.2, é uma função realizada bit a bit, que pode ser resumida pela TABELA 2.3. Em uma rápida análise desta função, pode-se concluir que ela é um caso particular da operação soma módulo, existente na aritmética modular e apresentada na SEÇÃO 3.2.3.

Observa-se que apenas dois valores de entradas e saídas são possíveis, 0 e 1. Naturalmente pensa-se, então, em operações módulo 2, e esta conclusão é realmente verdadeira, pois:

- $0 \oplus 0 = 0$, pois $0 + 0 = 0 \equiv 0 \ (mod\ 2)$
- $0 \oplus 1 = 1$, pois $0 + 1 = 1 \equiv 1 \ (mod\ 2)$
- $1 \oplus 0 = 1$, pois $1 + 0 = 1 \equiv 1 \ (mod\ 2)$
- $1 \oplus 1 = 0$, pois $1 + 1 = 2 \equiv 0 \ (mod\ 2)$

Para decidir-se sobre qual operação, XOR ou soma módulo, usar foi feito um teste, que utilizou uma rotina de medição de tempo que utiliza como entrada três arquivos de texto. Estes arquivos, que são compostos por 128, 512 e 1024 kilobytes tiveram os seus processos de encriptação medidos. A encriptação de cada arquivo foi medida duas vezes, uma feita com a operação XOR e a outra com a operação soma módulo(SM). Essas medições, em segundos, com erro de $\pm$1s, podem ser vistas na TABELA 3.1.

Tamanho do arquivo	128KB	512MB	1MB
Duração da encriptação utilizando SM	17	58	114
Duração da encriptação utilizando XOR	19	66	128

TAB. 3.1: Duração de processos de encriptação

Percebe-se que a diferença de execução é muito pequena para gerar qualquer preferência, porém optou-se pela operação soma módulo e desta maneira foi implementado o algoritmo do sistema proposto.

Com isto, é proposto, então, uma pequena mudança no criptosistema do *one-time pad*, como pode ser visto na FIGURA 3.5.

É importante ressaltar que todas as demonstrações de criptosistema perfeito feitas na SEÇÃO 2.4.1 são análogas para o caso da soma módulo n. Isso é rapidamente verificado constatando que, assim como no *one-time pad* original, a relação $|F| = |C| = |M|$ é verdadeira, ou seja, qualquer texto em claro possui exatamente uma transformação para

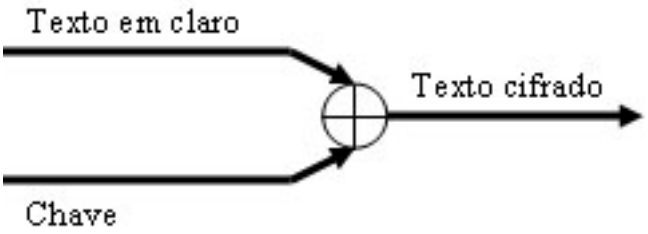

FIG. 3.5: One-time pad como soma módulo

cada texto cifrado possível. Considerando ainda que as chaves continuam aleatórias, pelo critério 3, fica demonstrado que este criptosistema também é perfeito.

Finalizando, a teoria de aritmética modular é bem mais extensa do que aqui foi apresentado, porém, para o presente trabalho, as seções estudadas são suficientes para a sua compreensão[8].

3.3 FUNÇÃO HASH

3.3.1 DEFINIÇÃO DE FUNÇÃO HASH

Como visto na SEÇÃO 3.1, a função escolhida foi a função *hash one-way*, cuja principal característica é inviabilidade de inversão. Muito cuidadosa teve que ser a escolha do algoritmo que seria utilizado neste trabalho, entre os diversos existentes. Esse cuidado deve-se ao fato que o criptosistema que será apresentado nesta dissertação, como visto também na SEÇÃO 3.1 é tão forte quanto a função *hash one-way* utilizada, pois a quebra desta, implica na quebra do criptosistema.

De qualquer maneira, é importante salientar que o criptosistema proposto, que foi implementado em C e que se encontra no APÊNDICE 9.1, foi implementado de maneira modular, ou seja, se houver a necessidade de substituir, em algum momento, a função *hash one-way* utilizada, basta substituir o módulo desta pelo da função que se deseja utilizar.

Uma função *hash one-way*, $h(M)$, que opera sobre uma mensagem M, retorna um valor chamado de *hash*, de tamanho fixo e designado por h (SCHENEIER, 1996). Além disso, esta função tem as seguintes características (MERKLE, 1979):

- Dado M, é fácil de computar h;
- Dado h, é difícil de computar M;

[8]Para um maior aprofundamento desta área da matemática, ver (SCHENEIER, 1996) e principalmente (COUTINHO, 2005)

- Dado *M*, é difícil de achar outra mensagem *M'*, tal que *H(M)=H(M')*.

Além destas, uma boa função *hash one-way* é livre de colisões[9], ou seja, é difícil achar duas mensagens quaisquer que possuam o mesmo *hash*.

3.3.2 OPÇÕES DE ESCOLHA DE FUNÇÕES HASH

Existem na literatura e no meio científico diversos algoritmos que serviriam ao propósito deste trabalho. Porém, como dito na SEÇÃO 3.3.1, essa escolha deveria ser cuidadosa. Entre as diversas possibilidades e após pesquisas minuciosas, foi dado prioridade aos quesitos rapidez e segurança, com ênfase na segurança.

Entre os algoritmos estudados, tem-se:

- SNEFRU (MERKLE, 1990) foi descartado por ser inseguro (BIHAM, 1993) seu uso com 2 estágios. E apesar de com 8 estágios ele ser seguro, ele fica mais lento que outros algoritmos, como MD5 e SHA (SCHENEIER, 1996);
- NHash (MIYAGUCHI, 1990) também foi descartado por ser inseguro (PRENEEL, 1993);
- MD4 (RIVEST, 1990) também foi descartado pela insegurança (DEN BOER, 1992);
- MD5 (RIVEST, 1992), apesar de ter sido provada a existência de algumas colisões, não é considerado inseguro (ROBSHAW, 1993). É uma boa possibilidade;
- MD2 (ROBSHAW, 1994) é considerado lento (PRENEEL, 1993);
- Enfim, SHA (NIST, 1994a) e suas variantes (NIST, 2002), mostraram-se bem aptos ao trabalho proposto nesta dissertação, principalmente as versões mais novas (SHA-256, SHA-384 e SHA-512), que serão elucidadas na SEÇÃO 3.3.3[10].

Fora estes, alguns mais novos ainda não foram suficientes testados, como por exemplo o *hash matrix* (INAYATULLAH, 2007), para serem utilizados neste trabalho.

[9] Tradução livre, retirada de SCHENEIER sobre o termo *collision-free*

[10] O SHA é chamado seguro porque foi projetado para ser computacionalmente inviável recuperar a mensagem correspondente, dado a mensagem processada (REGISTER, 1992).

3.3.3 SECURE HASH ALGORITHM

3.3.3.1 SHA-1

O NIST (*National Institute of Standards and Technology*) junto com a NSA (*National Security Agency*), ambos do governo dos Estados Unidos, desenvolveram o *Secure Hash Algorithm* (SHA) para ser usado como padrão de assinatura digital (SCHENEIER, 1996) em 1994. Alguns anos mais tarde, variações do SHA foram criadas (NIST, 2002).

Para fins didáticos, este trabalho explicará o algoritmo da primeira versão do SHA nesta seção. Apesar desta versão já ter sido quebrada, ela terá importância fundamental para a explicação, na SEÇÃO 3.3.3.2, da versão que será utilizada. Por fim, na SEÇÃO 3.3.3.3, será dada uma rápida elucidação das diferenças entre a versão escolhida e as outras versões, que apesar de mais modernas e seguras, têm suas desvantagens[11].

O primeiro passo feito pelo algoritmo SHA é o ajustamento dos dados de entrada em blocos de 512 bits. Para isso é necessário que o que será processado tenha um tamanho de um múltiplo de 512 bits. A regra para tanto é a descrita abaixo:

- Após o último bit da mensagem, concatena-se o bit 1;
- Em seguida contanena-se tantos bits 0 quanto seja necessário para se atingir o tamanho de 64 bits menor que o próximo múltiplo de 512;
- Por fim adiciona-se 64 bits, que conterão a informação que diz qual era o tamanho da mensagem antes das concatenações.

Tal sequência pode ser melhor visualizada na FIGURA 3.6. A primeira informação que pode-se tirar desta sequência e, consequentemente, da FIGURA 3.6 é que a entrada terá um tamanho máximo de 2^{64}, já que é o máximo que pode ser identificado pela última concatenação. Como o bloco total deve ser múltiplo de 512, então a concatenação dos três primeiros blocos (entrada + bit 1 + bits 0) é igual a um múltiplo de 512 menos 64 bits, como exigido. Quando se concatenar o último bloco, será atingido o múltiplo de 512.

Após as concatenações, o SHA trabalhará com cada sub-bloco de 512 bits separadamente, gerando para cada um deles uma saída de 160 bits, o *hash*. Cada um destes sub-blocos será dividido em 16 blocos menores de 32 bits (M_0 a M_{15}).

Após essa etapa, cinco variáveis de 32 bits são inicializadas como se segue[12]:

[11] Todas informações contidas nas seções citadas neste parágrafo foram retiradas de (SCHENEIER, 1996) e (NIST, 2002)

[12] Números em hexadecimal

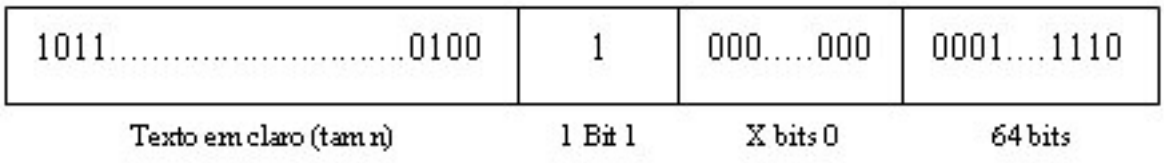

FIG. 3.6: Concatenações do SHA

- A = 0x67452301
- B = 0xefcdab89
- C = 0x98badcfe
- D = 0x10325476
- E = 0xc3d2e1f0

Em seguida, a recebe o valor de A, b recebe o valor de B e assim por diante:

- $a \leftarrow A$
- $b \leftarrow B$
- $c \leftarrow C$
- $d \leftarrow D$
- $e \leftarrow E$

O *loop* do SHA, então, inicia-se:

FOR t=0 to 79

TEMP = (a<<<5)+f_t(b,c,d)+e+W_t+K_t

e = d

d = c

c = b<<<30

b = a

a = TEMP

END FOR

Onde K_t são quatro constantes:

- K_t = 0x5a827999, para t=0 até 19;
- K_t = 0x6ed9eba1, para t=20 até 39;
- K_t = 0x8f1bbcdc, para t=40 até 59;
- K_t = 0xca62c1d6, para t=60 até 79, e

$f_t(X, Y, Z)$ são funções não lineares utilizadas no SHA:

- $f_t(X, Y, Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$, para t=0 até 19;
- $f_t(X, Y, Z) = X \oplus Y \oplus Z$, para t=20 até 39;
- $f_t(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$, para t=40 até 59;
- $f_t(X, Y, Z) = X \oplus Y \oplus Z$, para t=60 até 79;

E por fim:

- $W_t = M_t$, para t=0 até 15
- $W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1$, para t=16 até 79

Com o símbolo $<<< s$ significando um deslocamento circular de s bits, e os operadores binários $\wedge, \vee, \neg$ e $\oplus$ correspondendo, respectivamente, às operações binárias AND, OR, NOT e XOR. Uma operação SHA pode ser resumida pela FIGURA 3.7.

Caso só tenha se processado uma palavra de 512 bits (M_0 a M_{16}), por exemplo um arquivo de tamanho menor ou igual a 512 bits (64 Bytes), ficam encerrados os cálculos e assim determina-se o hash após os 80 *loops*:

- $H_0 = a + A$
- $H_1 = b + B$
- $H_2 = c + C$
- $H_3 = d + D$
- $H_4 = e + E$

E finalmente o *hash* será assim formado:

$$H_0|H_1|H_2|H_3|H_4$$

Porém se for processada mais de uma palavra de 512 bits, após o processamento da primeira palavra, serão feitos novos 80 loops para cada nova palavra de 512 bits e as constantes de entrada A, B, C, D e E serão os valores finais de a, b, c, d e e do último *loop*. Isso será melhor evidenciado quando, na SEÇÃO 3.3.3.2, for mostrado todo o algoritmo.

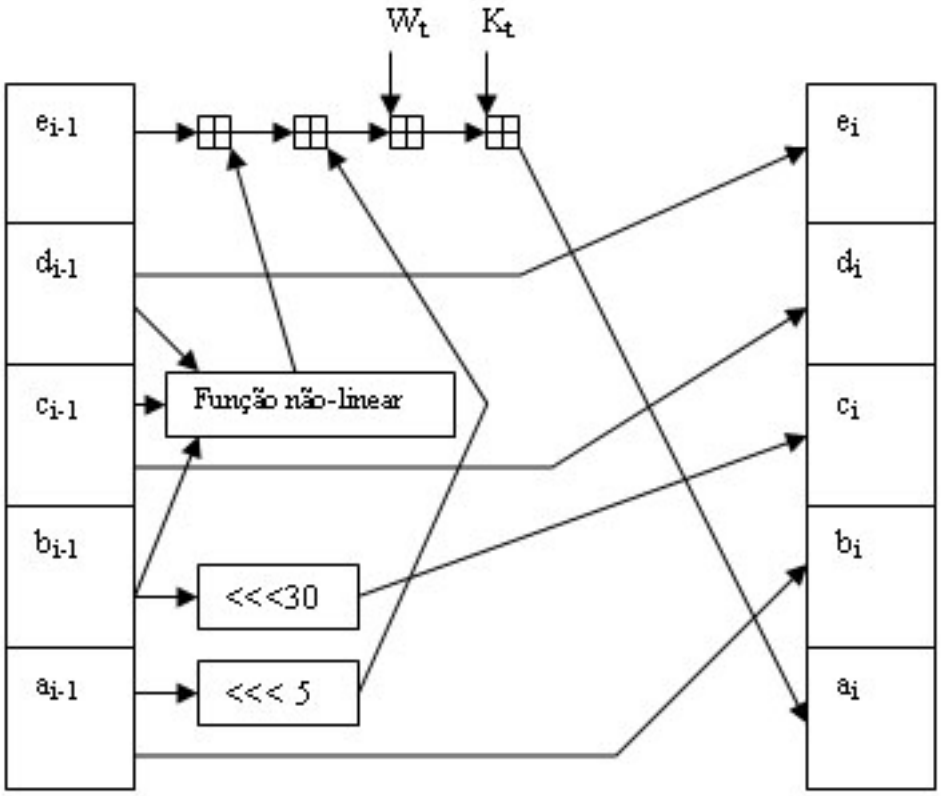

FIG. 3.7: Uma operação SHA

Este algoritmo, apesar de ter sido utilizado durante muitos anos, foi finalmente quebrado em 2005 pela equipe de pesquisa de Xiaoyun Wang, Yiqun Lisa Yin, e Hongbo Yu da Universidade Shandong, localizada na China (SCHENEIER, 2005). Ou seja, durante 11 anos, o SHA mostrou-se um algoritmo extremamente forte e só após muito trabalho, a sua criptoanálise foi finalmente alcançada. Por isso, apesar da quebra deste, esta dissertação utilizará uma de suas variações (SHA-256), descrita na próxima seção, que, provavelmente, gozará de segurança por, pelo menos, alguns anos mais, já que nenhuma notícia de criptoanálise bem sucedida deste foi encontrada no meio acadêmico até

a presente data.

De qualquer maneira é sempre bom lembrar que a idéia principal deste trabalho não é utilizar o SHA e sim uma variação do *one-time pad*. E já que a sua programação foi realizada modularmente, como já antes dito, no momento que o SHA-256 não for mais satisfatório, este pode ser substituído.

3.3.3.2 SHA-256

O algoritmo do SHA-256 é muito parecido com o do SHA-1. Assim, nesta seção só serão comentadas as diferenças e após isso será mostrado o algoritmo desta versão do SHA.

A primeira diferença reside no próprio nome. O número no SHA-256 significa que a saída (hash) é de 256 bits contra apenas 160 bits do SHA-1. As outras diferenças estão no algoritmo. Será seguida a mesma sequência do algoritmo que foi mostrada na SEÇÃO 3.3.3.1 para que seja feito o perfeito acompanhamento das diferenças existentes.

O primeiro passo, assim como na SEÇÃO 3.3.3.1, é o ajustamento dos dados de entrada em blocos de 512 bits. Para este passo não há qualquer diferença. Ele deve ser feito seguindo exatamente as mesmas regras:

- Após o último bit da mensagem, concatena-se o bit 1;
- Em seguida contanena-se tantos bits 0 quanto seja necessário para se atingir o tamanho de 64 bits menor que o próximo múltiplo de 512;
- Por fim adiciona-se 64 bits, que conterão a informação que diz qual era o tamanho da mensagem antes das concatenações.

Após essa etapa, o SHA trabalhará com cada sub-bloco de 512 bits separadamente, gerando para cada um deles uma saída de 256 bits, diferentemente do SHA-1, que gerava uma saída de 160 bits. Novamente, cada um destes sub-blocos será dividido em 16 blocos menores de 32 bits (M_0 a M_{15}).

Após essa etapa, oito variáveis de 32 bits (contra 5 do SHA-1) são inicializadas como se segue[13]:

- A = 0x6a09e667
- B = 0xbb67ae85

[13]Números em hexadecimal

- C = 0x3c6ef372
- D = 0xa54ff53a
- E = 0x510e527f
- F = 0x9b05688c
- G = 0x1f83d9ab
- H = 0x5be0cd19

Agora, para a primeira palavra de 512 bits da entrada, a recebe o valor de A, b recebe o valor de B e assim por diante. Porém, a partir da segunda palavra de 512 bits da entrada, a recebe o valor de H_0, b recebe o valor de H_1 e assim por diante. H_i significa a i-ésima parte da saída(hash) calculado para a palavra anterior. Esta situação fica melhor elucidada com o auxílio da figura 3.8. Destas informações:

- $a \leftarrow A$ se primeira palavra ou $a \leftarrow H_0$ caso contrário;
- $b \leftarrow B$ se primeira palavra ou $b \leftarrow H_1$ caso contrário;
- $c \leftarrow C$ se primeira palavra ou $c \leftarrow H_2$ caso contrário;
- $d \leftarrow D$ se primeira palavra ou $d \leftarrow H_3$ caso contrário;
- $e \leftarrow E$ se primeira palavra ou $e \leftarrow H_4$ caso contrário;
- $f \leftarrow F$ se primeira palavra ou $f \leftarrow H_5$ caso contrário;
- $g \leftarrow G$ se primeira palavra ou $g \leftarrow H_6$ caso contrário;
- $h \leftarrow H$ se primeira palavra ou $h \leftarrow H_7$ caso contrário.

O *loop* do SHA-256, então, inicia-se:

FOR t=0 to 63

TEMP1 = h + $\sum_1^{256}$(e) + Ch(e,f,g) + W_t + K_t^{256}

TEMP2 = $\sum_0^{256}$(a) + Maj(a,b,c)

h = g

g = f

f = e

e = d + TEMP1

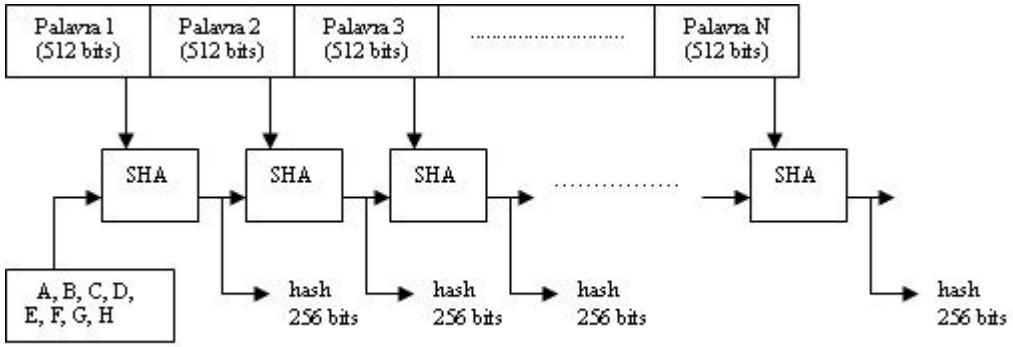

FIG. 3.8: A recursividade do SHA-256

d = c

c = b

b = a

a = TEMP1 + TEMP2

END FOR

Onde K_t^{256} são 64 constantes, descritas na FIGURA 3.9, em hexadecimal e da esquerda para direita:

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

FIG. 3.9: As 64 constantes do SHA-256

As funções $Ch(x, y, z)$, $Maj(x, y, z)$, $\sum_0^{256}$(x) e $\sum_1^{256}$(x) são:

- $Ch(x, y, z) = (x \wedge y) \vee ((\neg x) \wedge z)$
- $Maj(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$
- $\sum_0^{256}$(x)= $ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
- $\sum_1^{256}$(x)= $ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$

Onde $ROTR^n(x)$ significa rotação(circular) à direita de n casas.

W_t se resume a:

- $W_t = M_t$, para t=0 até 15

- $W_t = \sigma_1^{256}(W_{t-2}) \oplus W_{t-7} \oplus \sigma_0^{256}(W_{t-15}) \oplus W_{t-16}$, para t=16 até 63

As funções σ_0^{256}(x) e σ_1^{256}(x) são:

- σ_0^{256}(x)= $ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
- σ_1^{256}(x)= $ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

Onde $SHR^n(x)$ significa deslocamento(não-circular) à direita de n casas.

Novamente os operadores binários $\wedge, \vee, \neg$ e $\oplus$ correspondem, respectivamente, às operações binárias AND, OR, NOT e XOR.

E por fim, as partes do hash são calculadas:

- $H_0 = a + A$ se primeira palavra ou $H_0 = a + H_0$ caso contrário;
- $H_1 = b + B$ se primeira palavra ou $H_1 = b + H_1$ caso contrário;
- $H_2 = c + C$ se primeira palavra ou $H_2 = c + H_2$ caso contrário;
- $H_3 = d + D$ se primeira palavra ou $H_3 = d + H_3$ caso contrário;
- $H_4 = e + E$ se primeira palavra ou $H_4 = e + H_4$ caso contrário;
- $H_5 = f + F$ se primeira palavra ou $H_5 = f + H_5$ caso contrário;
- $H_6 = g + G$ se primeira palavra ou $H_6 = g + H_6$ caso contrário;
- $H_7 = h + H$ se primeira palavra ou $H_7 = h + H_7$ caso contrário;

E finalmente o *hash* será assim formado:

$$H_0|H_1|H_2|H_3|H_4|H_5|H_6|H_7$$

O *loop* acima descrito pode ser visualizado na FIGURA 3.10[14]

[14]Figura retirada de (AISOPOS, 2005)

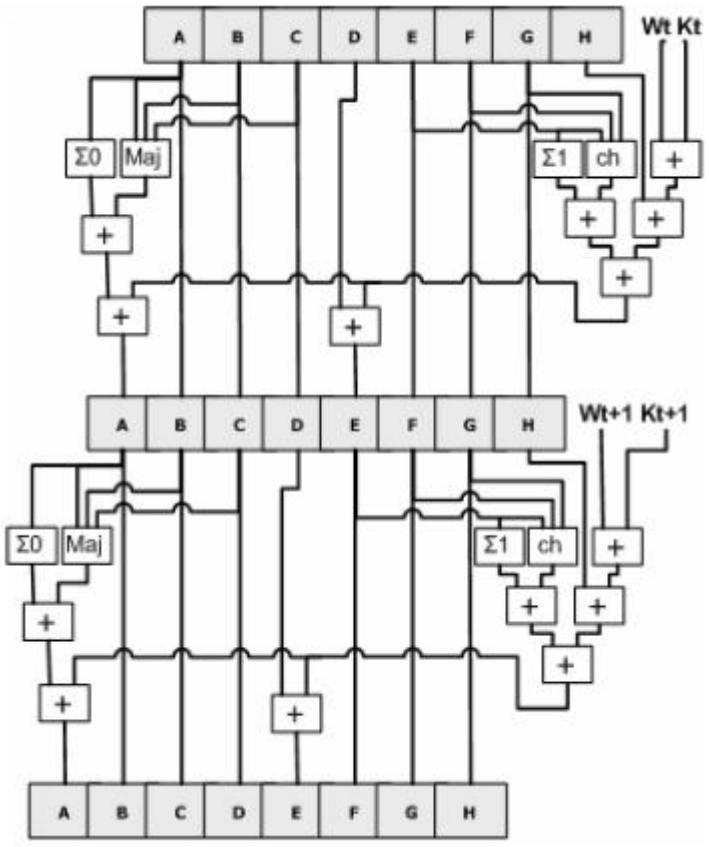

FIG. 3.10: Esquema Geral para o SHA-256

3.3.3.3 OUTRAS VERSÕES SHA

Além das versões 1 e 256 do SHA, existem também as versões que geram saídas de 224, 384 e 512 bits, chamadas de, respectivamente, SHA-224, SHA-384 e SHA-512. Além das diferenças encontradas nos tamanhos das saídas, podem ser encontradas outras diferenças entre todos esses algoritmos, tais como a forma de ajustamento dos dados de entrada, a forma de cálculo dos valores de W_t, os valores das constantes K_t e algumas pequenas diferenças nos algoritmos. Todos, porém, seguem o mesmo ciclo lógico, a exemplo do que foi mostrado para as versões 1 e 256. Posto isso, coloca-se então na TABELA 3.2 as diferenças principais existentes entre esses algoritmos.

Versão	Mensagem	Bloco	Palavra	Saída	Segurança
SHA-1	$<2^{64}$	512	32	160	80
SHA-224	$<2^{64}$	512	32	224	112
SHA-256	$<2^{64}$	512	32	256	128
SHA-384	$<2^{128}$	1024	64	384	192
SHA-512	$<2^{128}$	1024	64	512	256

TAB. 3.2: Principais diferenças entre as diversas versões do SHA

Os valores das colunas 2, 3, 4, 5 e 6 estão expressos em bits.

De maneira geral, as diversas versões SHA podem ser resumidas pela FIGURA 3.11.

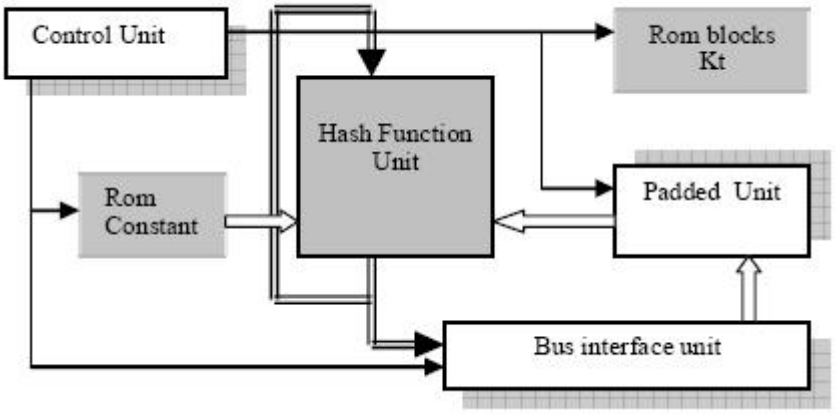

FIG. 3.11: Esquema Geral para as versões do SHA

É importante ressaltar novamente que a função dificilmente inversível (que no caso deste trabalho será o SHA) não é o foco principal do trabalho. Por isso, sempre será possível, quando necessário, trocar a utilizada por outra que seja mais conveniente. A escolha da versão 256 levou em conta segurança e rapidez (ZEGHID, 2007; AISOPOS, 2005).

4 TEORIA DE ANÁLISE DE PROJETO CRIPTOGRÁFICO

4.1 INTRODUÇÃO

Este capítulo baseia-se no estudo dos processos de seleção de algoritmos realizados pelo NIST (NECHVATAL, 1999) e na teoria de criptoanálise. O primeiro, o estudo dos processos de seleção de algoritmos, é voltado para cifradores de blocos iterado [15](LAMBERT, 2004), porém, neste trabalho, sua filosofia de projeto será adequada ao projeto apresentado no CAPÍTULO 3[16], que apesar de ser um cifrador de blocos[17], não é iterado[18]. O segundo, teoria de criptoanálise, trata da teoria voltada para a quebra de sistemas criptográficos considerando os possíveis ataques existentes.

4.2 FILOSOFIA UTILIZADA

A filosofia na qual este trabalho se baseia se traduz no respeito aos cinco princípios de projeto apresentados hierarquicamente a seguir: segurança, eficiência, flexibilidade, simplicidade, credibilidade (LAMBERT, 2004). Dentro de cada princípio, é sugerido um conjunto de requisitos para o projeto e para transformações criptográficas que este algoritmo incorpora. A satisfação destes requisitos é condição suficiente para projetar um cifrador simétrico de blocos dentro dos atuais padrões de aceitação internacionais. No escopo desta dissertação só serão apresentados requisitos que tenham sentido para um encriptador sequencial, que é o objeto de estudo deste trabalho.

[15]Um cifrador de blocos iterado é é um cifrador de blocos que inclui uma repetição seqüencial de uma função interna chamada função de passo. Os parâmetros de um cifrador de blocos iterados incluem o número de passos Nr, o tamanho dos blocos n, o tamanho k da chave K a partir da qual as Nr sub-chaves de passo K_i são geradas. Cada valor de K_i deve determinar uma bijeção de um passo (para permitir a decifração)(LAMBERT, 2004). Um exemplo de cifrador de bloco iterado é o AES ou DES(BIHAM, 1999)

[16]Quando trata-se de um cifrador de blocos iterado, outras características devem ser avaliadas, tais como a completude, o efeito avalanche e a matriz de dependências(FEISTEL, 1973)

[17]Segundo (SCHENEIER, 1996), um cifrador de blocos é aquele que opera o texto em claro em determinadas quantidades de bits

[18]É usual utilizar a expressão cifrador de blocos para representar um cifrador de blocos iterado, tipo de cifrador amplamentente utilizado e alvo de estudos acadêmicos

4.2.1 PRIMEIRO PRINCÍPIO: SEGURANÇA

A segurança de um algoritmo simétrico é função de dois parâmetros: a força do algoritmo e o tamanho do espaço de chaves. Ambas são igualmente importantes, mas a primeira torna-se mais crítica, por ser muito mais difícil de avaliar. O algoritmo é considerado forte se o método mais rápido para quebrar o algoritmo é a busca exaustiva da chave. Provar isso é um problema mais difícil que a própria criptoanálise. Apesar disso, é possível estabelecer limites inferiores para o fator trabalho da criptoanálise que são aceitos pela comunidade científica. Para isso, não deve haver nenhum tipo de atalho ou *backdoor*, o algoritmo deve ser resistente contra todas as formas de criptoanálise conhecidas e deve ser considerado improvável o surgimento de uma nova forma de criptoanálise que o ameace. Portanto, o algoritmo deve ser baseado em alguma estrutura criptográfica bem conhecida e consagrada[19]. Por fim, o espaço de chaves deve ser grande o suficiente para tornar a busca exaustiva um problema intratável[20]. Além disso, a especificação completa do algoritmo deve ser pública ou suficientemente semelhante a alguma estrutura de conhecimento público que esteja submetida à comunidade científica há muitos anos, sem ser comprometida, e a chave, os blocos de texto em claro e os blocos de texto cifrado devem ter, no mínimo, 128 bits.

4.2.2 SEGUNDO PRINCÍPIO: EFICIÊNCIA

Este conceito está relacionado ao aspecto da velocidade ou desempenho do algoritmo nas plataformas em que se pretende utilizá-lo, sendo de grande importância técnica e comercial. A importância fica clara sobretudo nas aplicações que exigem maior velocidade como por exemplo máquinas tipo ATM (e.g. caixas eletrônicos). Geralmente, a eficiência de um algoritmo tem uma relação intrínseca com o nível de segurança do mesmo, principalmente em algoritmos iterativos. Porém, mesmo no algoritmo sequencial deste trabalho, essa relação pode, a princípio, ser verificada. A utilização de uma função menos complexa que o SHA-256, provavelmente seria mais rápida. De qualquer maneira, atualmente, mais evidentemente do que ocorria no final dos anos 70, a tendência ao aumento de velocidade dos computadores torna boa prática dar ênfase à segurança mesmo que,

[19]No caso deste trabalho, o *one-time pad*

[20]Na prática, é suficiente que a busca exaustiva não seja compensadora. Isso depende do valor da informação. Se o algoritmo se prestará a proteger informações capazes de comprometer grandes empresas ou interesses de estado, a solução do problema deve estar além de toda a capacidade de processamento disponível(BLAZE)

possivelmente, em detrimento de alguma eficiência (velocidade). Esta consideração deve contribuir para aumentar a expectativa de vida útil de um padrão criptográfico e está de acordo com o que sugeriu Lars Knudsen em (KNUDSEN, 2000).

4.2.3 TERCEIRO PRINCÍPIO: SIMPLICIDADE

Este princípio está associado à facilidade de compreensão do algoritmo, simplicidade das operações realizadas por ele, tais como deslocamentos, soma módulo e etc.. O algoritmo deve ser suficientemente simples para compreensão e implementação em hardware e em software. É desejável que o conceito de simplicidade se estenda aos processos de construção das transformações, sobretudo das não lineares. A simplicidade também é importante quanto à validação do algoritmo, quando é desejável que se possam utilizar as técnicas já consagradas. Resumindo, a preocupação com a eficiência, a flexibilidade e a credibilidade devem naturalmente conduzir a um cifrador simples.

4.2.4 QUARTO PRINCÍPIO: FLEXIBILIDADE

A flexibilidade em relação à plataforma de utilização também é de grande importância comercial, pois espera-se que um padrão criptográfico seja utilizado em diversos tipos de componentes, como máquinas tipo ATM , cartões inteligentes, terminais de computadores, equipamentos de criptografia de voz, etc. Ou seja, esta característica é importante pois proporciona a boa compatibilidade entre os diferentes equipamentos em que o algoritmo será executado, contribuindo para a popularização do cifrador.

O não respeito a este princípio pode levar ao abandono do projeto. Por exemplo, o tamanho fixo da chave condenou o DES à substituição pelo AES, o que sugere que a flexibilidade em relação ao tamanho da chave é um aspecto desejável em um projeto de algoritmo criptográfico e fundamental para a segurança e a credibilidade do algoritmo, além de aumentar a sua expectativa de vida útil.

Ainda é desejável a flexibilidade em relação ao tamanho dos blocos, pois o ideal é que seja possível implementar o algoritmo para cifrar blocos de tamanhos diferentes.

Por fim, apesar de não haver uma relação rígida entre simplicidade e flexibilidade, a simplicidade do projeto tende a torná-lo mais flexível, uma vez que, sendo melhor compreendido, este pode ser mais fácil e eficientemente adaptado para diferentes aplicações.

4.2.5 QUINTO PRINCÍPIO: CREDIBILIDADE

A combinação dos princípios segurança e simplicidade conduz ao quinto princípio, credibilidade, pois é esperado que a simplicidade das operações evidencie qualquer fraqueza potencial da cifra. O tempo que o padrão leva para ser popularizado também influi na credibilidade do projeto. Este aspecto sugere que a simplicidade do algoritmo é um dos princípios que deve nortear o seu projeto, pois quanto mais simples, mais rápido estará sendo objeto de testes por estudiosos e a partir daí, mais rápido pode ter sua credibilidade atestada.

Com isso, um algoritmo criptográfico será tão mais útil quanto mais for utilizado pela comunidade, pois para que um remetente utilize uma cifra ele precisa saber que o destinatário também a utiliza. Para que um algoritmo seja bem aceito pela comunidade, a credibilidade é mais importante que a segurança do algoritmo. Embora seja extremamente improvável que um algoritmo inseguro tenha credibilidade na comunidade científica. Por outro lado, é relativamente fácil a credibilidade de um algoritmo ser comprometida e isso pode acontecer mesmo que não seja encontrada uma fraqueza no algoritmo que comprometa de fato a segurança. É suficiente que não fiquem claras as razões de projeto e que a estrutura do algoritmo seja difícil de analisar com as técnicas tradicionais. Isso ficou claro quando a IBM não publicou na década de 70 as regras para a distribuição das constantes nas tabelas de substituição e a razão para o número de iterações adotados no DES (SCHENEIER, 1996). Portanto, as regras seguidas para a construção das transformações utilizadas pelo cifrador devem ser claras, simples e divulgadas, para que não haja suspeitas da existência de *backdoors* ou atalhos.

4.3 TEORIA DE CRIPTOANÁLISE

O ponto crucial da criptografia é manter o texto em claro secreto. Para isso, é necessário também manter a chave secreta[21]. Partindo desse princípio básico, a criptoanálise é elaborada. Serão descritos agora os quatro tipos gerais de ataques[22]

- **Ataque utilizando apenas textos cifrados**[23] - O criptoanalista tem o texto cifrado de várias mensagens, todas encriptadas utilizando-se o mesmo algoritmo e a mesma chave. Seu trabalho é obter por dedução a chave utilizada e com isso poder decriptar os textos em claro correspondentes e os futuros;

[21] Assume-se que o adversário tem total acesso ao meio de comunicação utilizado (SCHENEIER, 1996).
[22] Uma tentativa de criptoanálise é chamada de ataque (SCHENEIER, 1996).
[23] *Ciphertext-only attack* (SCHENEIER, 1996)

- **Ataque quando se conhece os textos em claro**[24] - O criptoanalista tem acesso não apenas aos textos cifrados como também aos textos em claro correspondentes. Seu trabalho é obter por dedução a chave utilizada e com isso poder decriptar os textos em claro futuros;

- **Ataque utilizando texto em claro escolhido**[25] - O criptoanalista não só tem acesso aos textos cifrados e seus correspondentes textos em claro como também pode escolher qual texto em claro ele deseja encriptar. Este é mais poderoso que o anterior pois ele pode escolher específicos textos em claro, que poderão dar maiores informações sobre a chave. Seu trabalho é obter por dedução a chave utilizada e com isso poder decriptar os textos em claro futuros;

- **Ataque adaptado utilizando texto em claro escolhido**[26] - Este é um caso especial do anterior. Com esse ataque, o criptoanalista, além das vantagens do método, pode adaptar algumas características do texto em claro. Pode, por exemplo, escolher, com base nos resultados anteriores, um bloco menor de texto em claro;

[24] *Known-plaintext attack* (SCHENEIER, 1996)
[25] *Chosen-plaintext attack* (SCHENEIER, 1996)
[26] *Adaptative-chosen-plaintext attack* (SCHENEIER, 1996)

5 ANÁLISE DO SISTEMA PROPOSTO

5.1 INTRODUÇÃO

A Análise feita neste capítulo é baseada na teoria discorrida no CAPÍTULO 4. Nesse foram apresentadas duas vertentes: a primeira trata da filosofia de projeto utilizada, enquanto que a segunda trata dos ataques utilizados por criptoanalistas. Neste capítulo, então, o projeto será analisado segundo essas vertentes.

5.2 PRIMEIRA VERTENTE: FILOSOFIA UTILIZADA

Como já explicado, a filosofia na qual este trabalho se baseia se traduz no respeito aos cinco princípios a seguir: segurança, eficiência, flexibilidade, simplicidade, credibilidade (LAMBERT, 2004). A análise será feita segundo esses princípios.

5.2.1 PRINCÍPIO: SEGURANÇA

No decorrer deste trabalho não foi verificado qualquer indicação de não cumprimento dos requisitos destacados na SEÇÃO 4.2.1. Não há qualquer tipo de atalho ou *backdoors* no programa, o projeto é inspirado em uma estrutura criptográfica bem conhecida e consagrada, tem resistência às principais formas de criptoanálise existentes, pois a maioria destas é baseada em algoritmos iterados e possui um espaço de chaves grande: 512 bits. Satisfatórias considerações sobre este princípio podem ser consultadas na SEÇÃO 5.3.

5.2.2 PRINCÍPIO: EFICIÊNCIA

A eficiência está diretamente ligada à velocidade com que as informações são processadas. Neste quesito não foram feitos grandes testes, pois o computador utilizado para os testes não fornecia grande desempenho devido as suas características(um Pentium M, 1.7 GHZ, 512 MB de memória RAM), porém resultados viáveis foram conhecidos[27].

Como resultados satisfatórios foram encontrados, maior importância não foi dada a este princípio, pois devido aos avanços computacionais de hoje a ênfase dos projetos conhecidos tem sido quanto à segurança(KNUDSEN, 2000).

[27] ver TABELA 3.1

5.2.3 PRINCÍPIO: SIMPLICIDADE

A simplicidade é outro ponto forte deste projeto, visto que ele é baseado no algoritmo criptográfico mais simples conhecido, o *one-time pad*. Como visto na SEÇÃO 4.2.3, a simplicidade é verificada pela facilidade de compreensão do algoritmo e de implementação, seja em *hardware* ou *software*. O projeto apresentado por este trabalho possui pouca complexidade, utilizando apenas as funções de soma módulo e concatenação, que são funções básicas e podem ser implementadas em poucas linhas de programação. As funções utilizadas pelas funções SHA's existentes são igualmente simples.

5.2.4 PRINCÍPIO: FLEXIBILIDADE

A flexibilidade também está presente neste projeto: ele encripta qualquer quantidade de bits de entrada, não há qualquer restrição aparente de uso, seja *hardware* ou *software*, e o tamanho da chave pode ser regulado pela função não linear que for escolhida(no caso deste trabalho, o SHA-256). Além destes fatores, o programa implementado, como foi feito em C, pode ser utilizado tanto em plataformas Linux como Windows.

Por fim, o fato de ser um projeto simples, torna o princípio flexibilidade inerente a este, pois uma vez que, sendo melhor compreendido, este pode ser mais fácil e eficientemente adaptado para diferentes aplicações.

5.2.5 PRINCÍPIO: CREDIBILIDADE

A combinação dos princípios segurança e simplicidade conduz ao quinto princípio, credibilidade, pois é esperado que a simplicidade das operações evidencie qualquer fraqueza potencial da cifra. Daí, pode-se concluir que este projeto possui credibilidade, pois é, a princípio, simples e seguro. Claro, porém, que mais e mais testes devem ser feitos para se assegurar que este sistema não possui fraquezas que o tornem inutilizável.

5.3 SEGUNDA VERTENTE: TEORIA DE CRIPTOANÁLISE

O sistema proposto, como visto na SEÇÃO 3.1, pode ser dividido em duas partes. A primeira, que pode ser visualizada na FIGURA 5.2, tem a sua força atestada por uma simples análise. Pela própria proposta do sistema, r_i pode ser modelado como uma variável aleatória estatisticamente independente com distribuição uniforme. Esta variável aleatória passa por uma operação de soma com a máscara K_1, e desta operação obtém-se d_i. Tem-se que r_i é uma variável aleatória com distribuição uniforme e tem sua função

distribuição de probabilidade(fdp) dada por:

$$f_X(x) = \frac{1}{b-a},\ para\ a \leq x \leq b\ e\ 0\ c.c.$$

, onde a é o valor mínimo que r_i pode assumir e b é o valor máximo. A média desta função é igual a $\frac{b+a}{2}$

Caso a soma realizada no sistema fosse a soma aritmética, a soma desta variável aleatória r_i com uma constante, que no caso do sistema proposto é K_1, faria com que a função distribuição de probabilidade deste resultado fosse:

$$f_X(x + K_1) = \frac{1}{b-a},\ para\ a + K_1 \leq x \leq b + K_1\ e\ 0\ cc$$

, onde a média seria igual a

$$\frac{b + K_1 + a + K_1}{2} = \frac{b + a + 2K_1}{2} = \frac{b+a}{2} + K_1$$

, ou seja, bastaria obter experimentalmente a média de várias amostras r_i, a média de várias amostras de d_i, e fazer a diferença entre elas. O resultado seria K_1 e com isso o sistema estaria quebrado. Isto pode ser melhor visualizado na FIGURA 5.1.

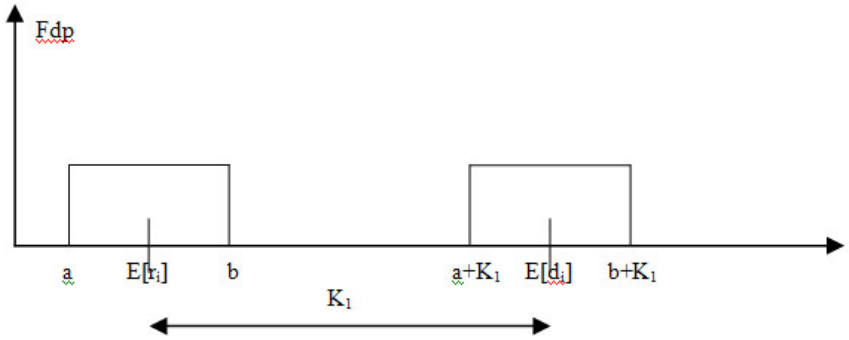

FIG. 5.1: Fdp no caso da soma aritmética

Já com a utilização da soma módulo não há este problema, como a operação soma módulo é uma operação cíclica, quando o valor da soma ultrapassasse o valor de b, começaria a recontagem. Isso significa que, no caso da soma módulo, não há deslocamento da fdp, ou seja, a fdp de r_i e a fdp de d_i ficariam sobrepostas.

Com isso, após esta operação de soma módulo a distribuição mantém-se uniforme, ou seja, conclui-se que d_i também é uma variável aleatória estatisticamente independente com distribuição uniforme e, desta maneira, essa primeira parte do sistema é inquebrável.

A segunda parte do sistema, que pode ser visualizada na FIGURA 5.3, pode ser analisada segundo um ataque de texto em claro. Dado c_i, que é conhecido do adversário,

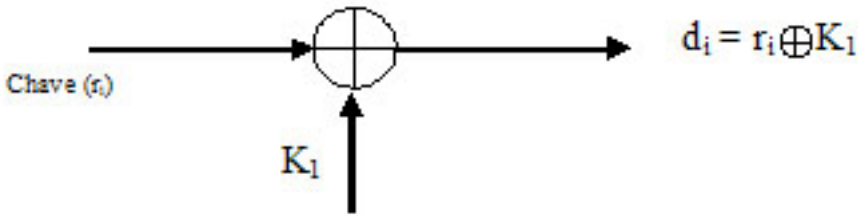

FIG. 5.2: Primeira parte do sistema

caso ele tenha acesso ao sistema, um m_i qualquer poderia ser colocado na entrada. A partir daí ter-se:

$$c_i = f(r_i) \oplus m_i \Rightarrow f(r_i) = c_i \oplus m_i$$

Desta maneira, para que seja possível calcular r_i, a função $f(x)$ terá que ser inversível. Porém a função escolhida, SHA-256, entre suas características, tem a da não inversibilidade, já que sua entrada é composta por 512 bits enquanto que sua saída é composta por 256 bits. Com isso, a criptoanálise, por este caminho, é ineficiente.

Esta análise permite afirmar que este sistema é tão forte quanto a função escolhida para esse passo. Devido a isso, a reconhecida força da função SHA-256 é o motivo pelo qual esta foi escolhida, mesmo que para isso a eficiência seja um pouco prejudicada, já que a saída desta função tem metade do tamanho da sua entrada. Com isso, caso venha a existir, nos futuros trabalhos científicos, alguma contra-indicação de uso para a função SHA-256, esta deve ser trocada por outra que não comprometa o sistema proposto.

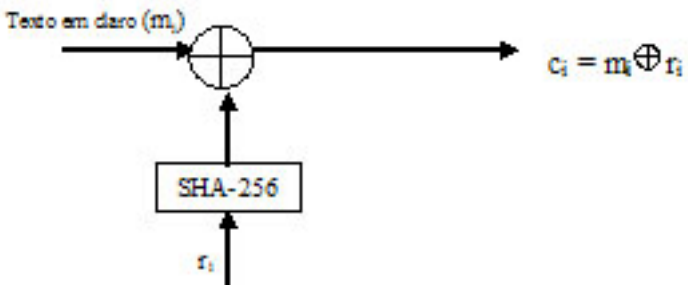

FIG. 5.3: Segunda parte do sistema

Outra observação importante, é que o adversário, caso tenha acesso ao sistema, seja este desenvolvido em *software* ou *hardware*, não deve ter acesso a entrada de números aleatórios (r_i), pois caso tenha, basta colocar a entrada 0 no lugar de r_i, que obterá o valor de K1 na saída d_i.

5.4 CONCLUSÃO DA ANÁLISE

A análise segundo os cinco princípios: segurança, eficiência, simplicidade, flexibilidade e credibilidade, não evidenciou contra-indicação ao uso do esquema proposto. Também a análise segundo ataques de criptoanálise não mostrou fraquezas evidentes.

É recomendável verificar se existem novas técnicas de análise disponíveis e utilizá-las para testes, melhorando, assim, a credibilidade do sistema.

6 IMPLEMENTAÇÃO DO SISTEMA

6.1 SEQUÊNCIAS ALEATÓRIAS E PSEUDO-ALEATÓRIAS

Conforme visto no CAPÍTULO 1.1, mais precisamente através da FIGURA 1.1, o conceito básico de criptografia passa e depende da existência de uma chave que possa ser utilizada para encriptação e outra que possa ser utilizada para decriptação (caso essas chaves sejam iguais, trata-se da criptografia simétrica, caso estas chaves sejam diferentes, trata-se da criptografia assimétrica). Como a mensagem a ser encriptada deve ser, obviamente, mantida secreta, as sequências de bits que compõem as chaves devem ser aleatórias, pois casos estas fossem previsíveis, trivial seria descobrir o conteúdo das mensagens.

Um importante tema de estudo é exatamente a geração de números verdadeiramente aleatórios. Estes não podem ser obtidos apenas por *software*, pois um computador pode gerar apenas um número finito de estados (mesmo que bem grande, ainda é finito) e com isso essa geração será sempre uma função determinística. Isto significa que qualquer gerador de números em um computador é periódico. Um gerador verdadeiramente aleatório necessita de alguma entrada verdadeiramente aleatória e um computador não pode fornecer isso apenas por *software* (SCHENEIER, 1996).

Fenômenos verdadeiramente aleatórios só existem no mundo real. Por isso, para essa geração, é necessária a observação de algum fenômeno físico, tal como o ruído branco[28] gerado por um resistor, por exemplo. O estudo da geração aleatória vem a cada dia dando mais resultados, porém, hoje, as taxas máximas atingidas por esses geradores, cerca de 2Mbps (TOSHIBA, 2008), não são aquelas requeridas pelos modernos sistemas de comunicação, cujas taxas de transmissão frequentemente ultrapassam os 1000Mbps (1Gbps)(DLINK, 2008),(UNICAMP, 2008).

Devido a esse problema é comum a criptografia utilizar os chamados números pseudo-aleatórios, gerados por geradores também chamados de pseudo-aleatórios. Os geradores pseudo-aleatórios são periódicos e suas sequências são reproduzíveis, já que estas sempre partem de um ponto inicial, chamado semente. O período da sequência deve ser longo o bastante (alguns chegam a 2^{256} (SCHENEIER, 1996) bits de tamanho) para que uma

[28] O termo ruído branco usualmente refere-se a um processo aleatório cuja densidade espectral de potência é igual para todas as frequências (GARCIA, 1994)

sequência finita retirada desta, de comprimento razoável (de acordo com o uso que será feito dela), não seja periódica (BORGES JR., 2008). Ou seja, se há a necessidade de 100.000 bits, não deve ser utilizado um gerador cujo período é 10.000 bits. Outras características são desejáveis a um gerador pseudo-aleatório, entre elas:

- deve ter igual quantidade de números zeros e uns;
- das sequências corridas, sequências de um mesmo bit, metade deve ser de comprimento um bit, um quarto de comprimento dois bits e assim por diante;
- não deve ser passível de compressão, e
- a distribuição de sequências corridas de zeros e uns deve ser a mesma[29].

Estas características podem ser verificadas utilizando-se o teste qui-quadrado (*chi-square*) encontrado em (GARCIA, 1994).

Além destas características, os geradores pseudo-aleatórios devem possuir as seguintes propriedades:

a) deve parecer aleatório, ou seja, deve passar nos principais testes de aleatoriedade, e

b) deve ser computacionalmente inviável predizer o próximo bit da sequência, ou seja, a probabilidade de acertar o próximo número da sequência deve ser baixa, mesmo havendo completo conhecimento do algoritmo que gera aquelas sequências.

   Adicionalmente a estas propriedades, se for desejável testar se uma determinada sequência é verdadeiramente aleatória deve-se testar uma outra propriedade:

c) a sequência não pode ser reproduzida, ou seja, se o gerador aleatório for acionado duas vezes com a mesma entrada, ele não fornecerá a mesma saída.

6.2 ALGORITMOS DE GERADORES PSEUDO-ALEATÓRIOS

Apesar da vasta utilização de números pseudo-aleatórios gerados por compiladores em sistemas, como por exemplo, de jogos, não é qualquer um que pode ser utilizado como fonte geradora de sequências porque nem todos são suficientemente seguros para o propósito da criptografia, pois podem não estabeler a segurança devida aos sistemas

[29]Retirado de (RUEPPEL, 1986)

criptográficos, já que a criptografia é extremamente sensível às propriedades destes geradores.

Ao usar um gerador de números pseudo-aleatórios fraco pode-se obter uma seqüência com certo grau de correlação que irá influenciar no resultado final da encriptação. Essa correlação, ao ser descoberta pelos criptoanalistas, será utilizada como forma de quebrar o gerador de sequências pseudo-aleatórias, ou seja, predizer o número seguinte a ser gerado.

O estudo de algoritmos de geração pseudo-aleatórios serve para criar mecanismos auxiliares para a implementação de geradores de seqüências de números aleatórios, pois, como visto, a geração de números verdadeiramente aleatórios não é simples e na maioria das aplicações a taxa de geração não atende aos critérios desejados pelo projeto. A utilização, então, dos algoritmos pseudo-aleatórios auxiliam ao atendimento dos critérios de segurança, velocidade e praticidade necessários quando utilizados em conjunto com os geradores aleatórios.

Esse uso conjunto baseia-se em uma idéia simples, a geração de números verdadeiramente aleatórios serve de alimentação para a geração de números pseudo-aleatórios, ou seja, os números verdadeiramente aleatórios são utilizados como as sementes dos geradores pseudo-aleatórios. Estes, portanto, fornecem como saída números ainda pseudo-aleatórios, mas com maior grau de segurança, pois têm como sementes números verdadeiramente aleatórios. Com essa técnica ganha-se maior velocidade e maior praticidade. A utilização de mais de um tipo de algoritmo pseudo-aleatório pode ser utilizado na tentativa de aumentar o período da seqüência pseudo-aleatória.

Serão apresentados agora dois algoritmos pseudo-aleatórios de importância fundamental nesta área, pela vasta utilização de suas idéias e variações.

6.2.1 MÉTODO CONGRUENTE LINEAR

Muitos geradores de números pseudo-aleatórios utilizados são modificações do método linear congruente, daí a importância de apresentá-lo. Ele se baseia na seguinte relação de recorrência:

$$x_{n+1} = (a.x_n + c) mod\ m, n \geq 0$$

O valor x_0 inicial é a semente, a é o multiplicador, c o incremento, m é a quantidade de número diferentes que se deseja gerar. A escolha adequada desses valores é fundamental para a determinação do período da seqüência de saída. Pela simples observação constata-se que o período será sempre menor que m, e que para se ter uma sequência maior deve-se aumentar o valor de m. Este algoritmo é extremamente simples e atende muito bem

aos requisitos de geradores de seqüências de números pseudo-aleatórios, desde que seja realizada a escolha adequada dos parâmetros.

Para obter um período máximo(SCHENEIER, 1996):

- c e m devem ser primos entre si;
- $b = (a - 1)$ é um múltiplo de p, para todo p primo divisor de m;
- b é um múltiplo de 4, se m é múltiplo de 4.

Para a escolha dos parâmetros, aconselha-se fazer uma primeira análise em m. Como foi visto, o período da seqüência será no máximo igual a m. De acordo com o valor de n, o valor de m é escolhido de modo a ser suficientemente maior que n, mas não tão grande, para poupar recursos computacionais. Sugere-se que m seja uma função de n:

$$m = (menorprimo(\sqrt{10n}))^2$$

A função menorprimo(x) retorna o maior número primo d, tal que $d < x$.

Com esta função garante-se que m será suficientemente maior que n e ao mesmo tempo, m é da forma $m = p^2$ com p primo, e logo os únicos divisores de m serão 1, m e p. Então, convenientemente, escolhe-se como valor de a :

$$a = menorprimo(\sqrt{10n}) + 1$$

sendo p:

$$p = menorprimo(\sqrt{10n})$$

q é o único divisor primo de m, e pela segunda condição sabe-se que $(a - 1)$ deve ser múltiplo de p para todo p primo divisor de m. Então deve ser de forma:

$$a - 1 = np \rightarrow a = np + 1 (n \in \aleph)$$

O valor de n que se adapta às funções sugeridas é 1. O valor de c é escolhido entre 0 e $\frac{m}{10}$, sendo c diferente de p, assim a primeira condição é satisfeita. Como m é apenas múltiplo de p, sendo p primo, a terceira condição fica sem sentido e perde significado. Para o valor de semente x_0 a escolha é realizada aleatoriamente entre 0 e m.

6.2.2 LINEAR FEEDBACK SHIFT REGISTER(LFSR)

Esse algoritmo tem sido utilizado na criptografia militar dos Estados Unidos desde o começo da engenharia eletrônica. O LFSR é formado por duas partes: um deslocador

à direita e uma função de retorno na formação de bits. O(s) bit(s) menos significativo(s) é (são) requisitado(s) e o bloco restante é deslocado para direita. Um número igual ao de bits requisitados é inserido à esquerda, sendo estes bits inseridos o resultado de uma determinada operação realizada com os demais bits. A saída utilizada para formar uma sequência é o conjunto de bits mais à direita, que saem do registrador a cada ciclo de deslocamento. O período desta geração depende do tamanho do registrador e do número de bits requisitados. A FIGURA 6.1 exemplifica o esquema geral e a FIGURA 6.2 exemplifica um LFSR de período 4 e tamanho também igual a 4.

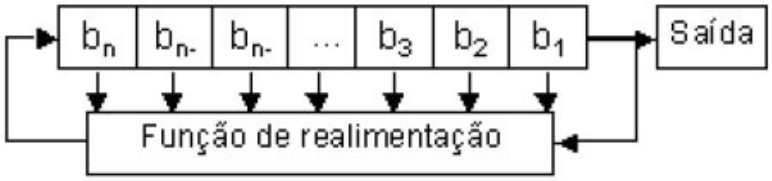

FIG. 6.1: Esquema Geral do LFSR

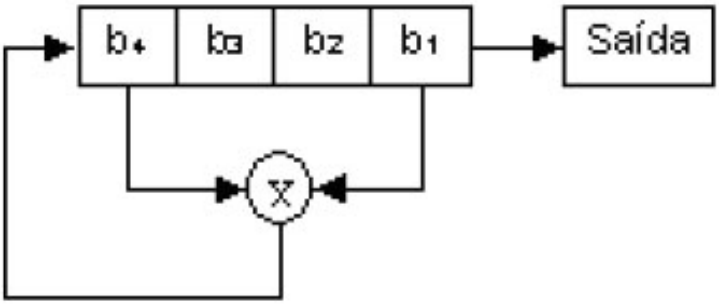

FIG. 6.2: LFSR de período e tamanho 4

O LFSR é o tipo mais simples de registrador e o tipo mais comum usado em criptografia. A função de retorno correspondente a este algoritmo é a função XOR de bits remanescentes no registrador, como mostrado na figura 6.2. Analisando esta mesma figura e tendo como semente a sequência 0001 tem-se a sequência mostrada na TABELA 6.1.

É possível observar que seu período é de tamanho 15 e que por motivo óbvio a semente nunca pode ser nula. Um gerador LFSR pode gerar uma sequência pseudo-aleatória de período até $2n - 1$, sendo n o número de bits que o registrador comporta. O grifo na palavra até é devido a necessidade de ressaltar que neste caso o período foi 15 não porque

registrador	função de retorno	saída
0001	1	1
1000	1	0
1100	1	0
1110	0	1
1111	0	1
0111	1	1
1011	0	1
0101	1	1
1010	1	0
1101	0	1
0110	0	0
0011	1	1
1001	0	1
0100	0	0
0010	0	0

TAB. 6.1: Ciclos LFSR de período 4

o tamanho do registrador é 4 e sim devido a semente escolhida. Esta deve ser escolhida utilizando o conceito de polinômios primitivos e campos de Galois, que fogem ao escopo deste trabalho. Para entendimento básico, basta saber que, para que o período seja máximo, os bits da semente mais um deve ser um polinômio primitivo módulo 2, onde um polinômio primitivo de grau n é um polinômio irredutível que divide $x^g + 1$, onde $g = 2^{n-1}$, mas não divide $x^d + 1$, para qualquer d que divide $2^n - 1$ (SCHENEIER, 1996). Para maiores informações sobre esta área da matemática ver (ZIERLER, 1969, 1968). No caso, os bits da semente geram o polinômio $0.x^4 + 0.x^3 + 0.x^2 + 1.x + 1 = x + 1$, que é polinômio primitivo módulo 2.

O LFSR não deve ser utilizado sozinho, pois por ser linear e geralmente esparso (poucos bits 1) apresenta algumas fraquezas que podem ser exploradas. O seu uso porém combinado com alguma função não linear pode resultar em bons geradores pseudo-aleatórios, como é o caso do STOP and GO. A tradução literal do nome deste gerador já explica seu funcionamento, parada e progresso alternando. Este gerador não sofre tantos problemas no teste de correlação. Funciona com três LFSRs (LFSR-1, LFSR-2 e LFSR-3) de diferentes comprimentos de seus registradores. O LFSR-1 ativa o clock do LFSR-2 quando sua saída é 1, e ativa o LFSR-3 quando sua saída é 0. A saída do gerador é um XOR entre as saídas de LFSR-2 e LFSR-3. Este pode ser visualizado na FIGURA 6.3. A função $\phi(t)$ é o clock do conjunto.

Uma versão melhorada do STOP and GO é a cascata de Gollmann (SCHENEIER,

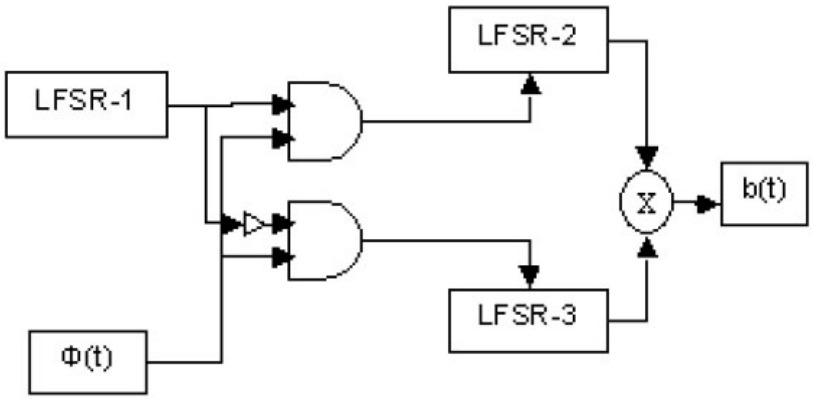

FIG. 6.3: Uso eficiente de LFSR's para geração pseudo-aleatória

1996), apresentada na FIGURA 6.4. Este consiste de uma série de LFSRs, cuja saída pertencente a um LFSR aciona o *clock* do próximo LFSR, ou seja, se a saída do LFSR-1 for igual a 1, o *clock* de LFSR-2 é ativado, se a saída de LFSR-2 for igual a 1, o *clock* de LFSR-3 é ativado, e assim por diante. A saída do último LFSR é a saída do gerador. Este projeto em cascata é bastante simples e pode ser usado para gerar sequências com períodos extremamente grandes. Possui bom desempenho nos testes de aleatoriedade, mas é vulnerável ao ataque de criptoanálise chamado *lock-in*, este ataque consiste da reconstrução da entrada do último registrador, e assim, reconstrói-se a seqüência de registradores. Uma prevenção para este tipo de ataque é o uso de registradores de tamanho mínimo de 15 bits. Pela análise de quebra de código e falha de segurança, é melhor que sejam usados muitos registradores de poucos bits que poucos registradores de muitos bits.

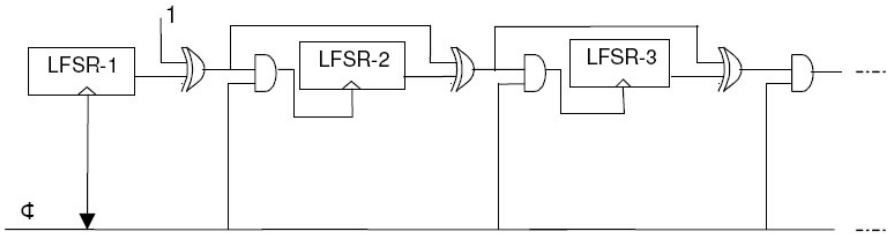

FIG. 6.4: Casacata de Gollmann

6.3 ALGORITMOS DE GERADORES ALEATÓRIOS

Apesar de sempre se falar em número aleatório, este termo só adquire sentido quando visto dentro de um conjunto, pois não é possível dizer se um número é mais ou menos

aleatório sem conhecer a seqüência a que ele pertence e o número de amostras dessa seqüência. Uma sequência aleatória deve ser composta por dados não-correlatados, independentes e que não possam se prever. Nas próximas subseções serão apresentadas algumas formas de obtenção dessas sequências[30].

6.3.1 LANÇAMENTO DE UMA MOEDA

De maneira mais teórica, uma sequência de zeros e uns obtida pelos lançamentos de uma moeda basta para a obtenção de uma sequência, assumindo que esta moeda não seja viciada, ou seja, a probabilidade de sair cara ou coroa é meio. É claro que esta forma de geração de bits aleatórios não tem nenhum valor prático, até porque a taxa de geração não é suficiente e, por isso, esta forma é ineficiente. Uma variação desta forma seria o lançamento de um dado, também não viciado, e considerar que três de suas seis faces correspondem a um bit e as outras três faces correspondem ao outro bit.

6.3.2 RUÍDO DE DIODO

O ruído de um diodo pode servir para produzir sequências de números aleatórios. Isso acontece devido aos defeitos inerentes aos materiais semicondutores do diodo e ao seu comportamento na região limite de polarização. A única maneira de saber se determinado circuito que contenha um diodo gerará números verdadeiramente aleatórios é realizando testes estatísticos de aleatoriedade. Em geral, estes circuitos são fontes de sequências binárias que tem uma distribuição estatística uniforme, e por isso, geralmente, promovem uma maneira eficiente de geração de sequências aleatórias. Um aprofundamento desta teoria pode ser visto em (BORGES JR., 2008), onde um circuito contendo um diodo Zener é utilizado como fonte aleatória. O circuito gerador projetado, utiliza-se basicamente das características do diodo Zener operando nas proximidades da região de ruptura, para obtenção de um sinal aleatório, de modo a ser posteriormente amplificado por dois estágios constituídos por transistores bipolares de junção (TBJ), devidamente polarizados, dispostos na configuração emissor comum, a fim de prover maior ganho de tensão, com o intuito de gerar um sinal com amplitude variando na faixa de 1V a 2V. Este sistema pode ser visualizado na FIGURA 6.5.

O sinal gerado pelo diodo Zener é produzido pelo efeito de avalanche. Esse efeito é causado basicamente pela polarização reversa do diodo, que faz com que os elétrons se choquem desordenadamente, e com isso, ganhem energia para se chocarem com outros,

[30]Retirados de (BORGES JR., 2008)

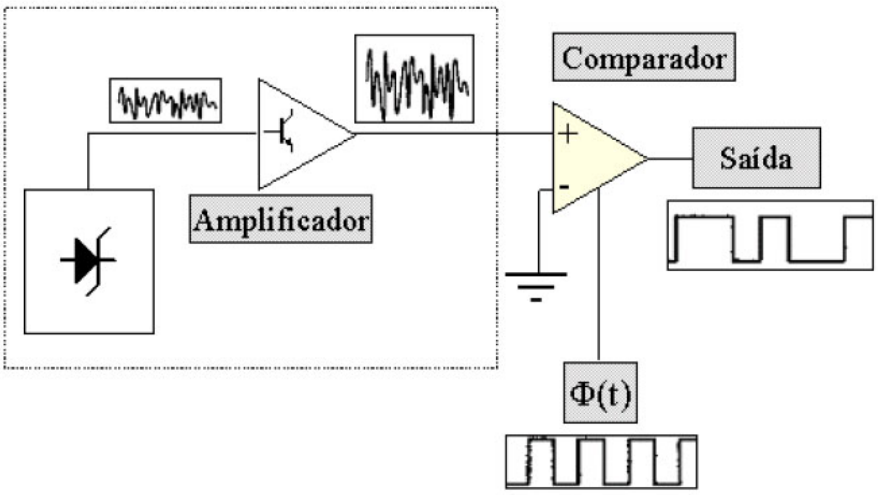

FIG. 6.5: Gerador de números aleatórios utilizando o diodo Zener

e assim sucessivamente, resultando em um efeito em cadeia, ou avalanche que produz o ruído branco com banda limitada.

6.3.3 MATERIAL RADIOATIVO

Os materiais radioativos também podem ser utilizados como geradores de números aleatórios, uma vez que emitem fótons de forma aleatória. Um contador Geiger, conectado a um computador, pode ser usado para transformar estes dados em informação útil e administrável em criptografia. Essa geração é rara e cara devido ao preço dos equipamentos utilizados e o custo de manutenção.

6.3.4 T12RNG

O T12RNG é um dispositivo capaz de gerar números aleatórios através de *hardware* sendo conectado a uma porta serial de computador. A aleatoriedade na geração é devido ao ruído branco produzido pelos semicondutores instalados no circuito. Pode ser usado diretamente em senhas, chaves criptográficas, *one-time-pads* etc, ou indiretamente como semente para um gerador pseudo-aleatório. Atinge taxas de 60 bytes por segundo, o que pode ser considerado lento em aplicações práticas.

Este trabalho utilizará um dispositivo cuja idéia é parecida com a do T12RNG para atingir seus objetivos. Trata-se do *Linux Random Number Generator*(LRNG), que utiliza as atividades inerentes aos dispositivos de *hardware* de um computador como entrada para o seu gerador pseudo-aleatório, que será visto na próxima seção.

6.4 ALGORITMO UTILIZADO POR ESTE TRABALHO

Como visto, fontes verdadeiramente aleatórias são originadas por fenômenos físicos, e estes são de difícil captação e transformação em dados utilizáveis. Devido a isso, *apenas para efeito de implementação do sistema proposto*, optou-se por utilizar esses números aleatórios como sementes de geradores de números pseudo-aleatórios. Estas sequências são suficientemente aleatórias desde que a semente não seja conhecida. Esse caso geral é utilizado também pelo LRNG, processo de geração de números pseudo-aleatórios inerente ao kernel (núcleo) do Linux, ou seja, presente em todas as suas distribuições. Por ser um gerador de fácil utilização e razoável qualidade (BARAK, 2005; GUTTERMAN, 2006) foi escolhido como fonte dos números aleatórios que servirão de entrada no sistema que foi proposto no CAPÍTULO 3. Claro que tem-se a consciência que a utilização de geração verdadeiramente aleatória, como visto na SUBSEÇÃO 6.3.2, melhoraria os resultados que serão apresentados no CAPÍTULO 5, o que permite concluir que bons resultados com esse gerador apresentará ainda melhores resultados com geradores verdadeiramente aleatórios.

6.4.1 ESTRUTURA GERAL DO LRNG

A estrutura do LRNG pode ser resumida em 3 componentes assíncronos (GUTTERMAN, 2006). O primeiro traduz eventos de componentes de sistema, tais como teclado, mouse, interrupções e discos rígidos em bits que compõe a entropia do sistema. O segundo componente adiciona bits a uma pool primária e o terceiro, quando bits são lidos pelo gerador, aplica o algoritmo SHA-1 três vezes consecutivas para gerar a saída do gerador e retro-alimentar a própria pool de bits. Cada amostra de aleatoriedade gera 2 palavras de 32 bits, a primeira diz o momento do evento e a segunda o valor do evento.

Essa *pool* primária alimenta outras duas pools, a secundária e a urandom, que têm características diversas. A *pool* secundária abastece o arquivo *random* dentro do diretório *dev* e a *pool* urandom abastece o arquivo *urandom* - unblocked random - também dentro do diretório *dev* e pode ser acessada pelo comando *get_random_bytes*. A diferença entre as duas reside no fato que */dev/random* não retorna mais bits do que sua atual entropia estimada, ou seja, após este número de bits ele bloqueia a emissão de bits, o que o torna mais seguro. Já */dev/urandom* devolve a quantidade de bits solicitada independentemente do valor da entropia do sistema, o que o torna mais inseguro que o primeiro. O esquema geral do funcionamento do LNRG pode ser visualizado na FIGURA 6.6.

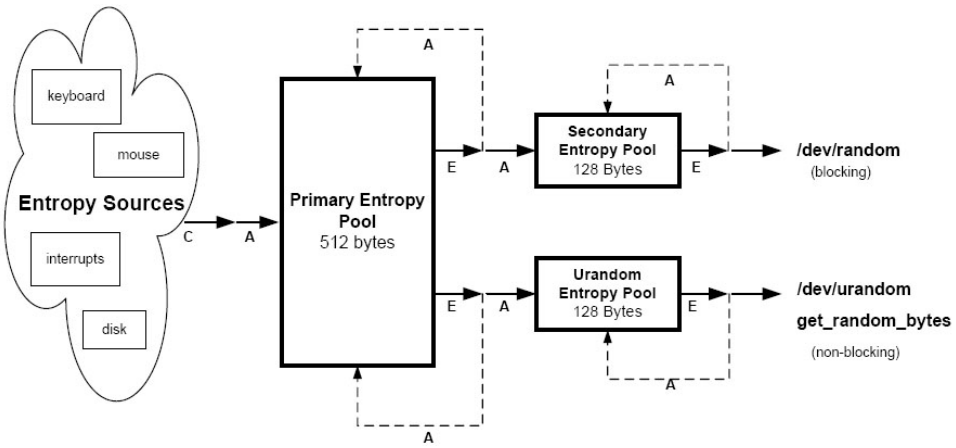

FIG. 6.6: Esquema Geral do LRNG. C - Coleta de bits, A - Adição de bits e E - extração de bits.

Cada pool(a primária, a secundária e a urandom) possui um contador próprio. Elas têm, respectivamente, o tamanho *default* de 512, 128 e 128 bytes, modificáveis (THEODORE, 2008). Esses contadores sempre estão em um estado que varia de zero ao tamanho da *pool*, que indica a entropia estimada da *pool*. Quando há saída de bits de uma *pool*, o seu contador é decrementado e quando há entrada de bits, o seu contador é incrementado. O contador é decrementado sempre da exata quantidade de bits que foram extraídos. Quando os bits são adicionados, a situação é um pouco mais complexa, pois esses bits podem ter vindo das fontes de entropia ou da *pool* primária. Se vieram das fontes de entropia, esta é então reestimada e o contador incrementado de acordo com essa reestimação. Se os bits vêm da *pool* primária, então o contador é adicionado dos bits recebidos. O contador da pool secundária tem uma importância fundamental na função de bloqueio do arquivo */dev/random*, pois quando a quantidade de bits solicitada é maior que o valor de seu estado, a entrega de bits terá que ser bloqueada até que mais bits sejam fornecidos pelos eventos de sistema.

Devido à natureza assíncrona do sistema, a entropia coletada não pode ser simplesmente adicionada, mas ela é coletada e armazenada. Algumas vezes por minuto esses dados são utilizados para fazer a estimação da entropia que será adicionada à pool primária e evidenciada em seu contador. Após encher esta, o sistema passa a enviar a entropia para a *pool* secundária. Após isso o ciclo recomeça. O sistema nunca envia entropia diretamente para a pool *urandom*.

Por fim, a extração de bits ocorre quando um usuário utiliza os arquivos */dev/random* e */dev/urandom*, retirando assim, respectivamente, bits da pool secundária e da *pool urandom* e quando a *pool* secundária e *urandom* não possuem bits suficientes e daí

extraem bits da *pool* primária.

6.4.2 INICIALIZAÇÃO DO LRNG

Como já dito, quando eventos de sistemas são capturados pelo sistema, duas palavras de 32 bits são geradas, uma contendo o tempo do evento em *jiffies*(número de milisegundos passados desde a inicialização da máquina) e a outra contendo um valor para o evento ocorrido. A primeira palavra seria muito previsível caso esta contagem de tempo fosse zerada cada vez que se inicializasse a máquina, pois, por exemplo, se houvesse o conhecimento de que a máquina é sempre desligada ao final do dia, então o ataque de força-bruta só teria que testar a quantidade de milissegundos existentes em 24 horas, cerca de 2^{26} opções. Para evitar isso, o sistema simula uma continuidade. Cada vez que o sistema é desligado, ele lê 512 bytes de */dev/urandom* e escreve em um arquivo. Durante a próxima inicialização, o sistema recupera os 512 bytes e escreve de volta no dispositivo */dev/urandom*. Analisando a FIGURA 6.6, verifica-se que quando escreve-se na pool *urandom*, há uma retro-alimentação para a pool primária. Os resultados dessa operação são muito semelhantes a receber 512 bytes através de eventos de sistema (GUTTERMAN, 2006).

6.4.3 COLETANDO ENTROPIA

As duas palavras geradas pela observação de eventos de sistema ocorrem como explicitado na SUBSEÇÃO 6.4.2. A primeira palavra, como explicado, vem do número de milisegundos passados desde a inicialização da máquina. Já a segunda palavra depende do processamento dos eventos observados, pois por si só apresentaria pouca entropia, como evidenciado na TABELA 6.2. Esse processamento dos eventos permite concluir que o LRNG é, então, um gerador pseudo-aleatório, como anteriormente dito. Serão apresentados aqui as formas de obtenção de bits através da observação de teclados, *mouses*, discos rígidos e interrupções:

- Teclado - a palavra de 32 bits contém diversos códigos de que indicam o pressionar e soltar de teclas, dentro de um *range* de 0 a 255;
- Interrupções - a palavra de 32 bits contém diversos códigos de que indicam as IRQs - *Interrupted Requests*, dentro de um *range* de 0 a 16;
- Mouse - a palavra de 32 bits é formada pela expressão:

$$palavra = (tipo << 4) \oplus codigo \oplus (codigo >> 4) \oplus valor$$

Onde tipo é a representação binária da indicação de início e fim de movimento, codigo é a representação do pressionar ou soltar os botões do mouse e valor é a indicação booleana(verdadeiro ou falso) do pressionar do mouse;

- Disco rígido - a palavra de 32 bits é formada pela expressão:

$$palavra = 0x100 + ((maior << 20)|menor)$$

Onde maior e menor são números resultantes da ação de acesso a um disco e variam conforme o disco em questão (IDE, SCSI, Floppy e etc.).

É importante ressaltar que esses não são os únicos meios de se coletar entropia do sistema. Os casos acima prestam-se apenas a exemplificar a coleta de entropia. Em sistemas Linux, por exemplo, que trabalhem em um roteador, que, geralmente, não possui teclados, discos ou mouses, outras formas de obtenção de dados aleatórios podem ser implementadas, tais como a leitura das diversas portas de rede desse específico *hardware*.

Teclado	Mouse	Disco rígido	Interrupções
8	12	3	4

TAB. 6.2: Número de bits desconhecidos resultantes dos eventos de sistema

6.4.4 ESTIMAÇÃO DA QUANTIDADE DE ENTROPIA

O conceito de entropia para o LRNG não tem exatamente o mesmo significado clássico conhecido, que é a medida de incerteza de uma variável aleatória (COVER, 1991), definido por:

$$H(X) = -\sum p(x).log[p(x)]$$

Para o LRNG, entropia significa apenas a quantidade de bits que o sistema já coletou ao observar os eventos de *hardware*, tais como interrupções, apertar de teclas de teclados e etc., ou seja, a quantidade disponível para ser solicitada pelo sistema.

Conforme dito, ao se observar os eventos de sistemas, novos bits são adicionados à entropia, e essa é evidenciada pelo contador de cada *pool*. O LRNG estima a quantidade de entropia utilizando apenas a palavra gerada que indica o tempo do evento, não utilizando a palavra que indica o valor do evento. Essa estimação é feita da seguinte maneira:

Seja t_n o tempo do evento n. Define-se:

$$\delta_n = t_n - t_{n-1}$$

$$\delta_n^2 = \delta_n - \delta_{n-1}$$

$$\delta_n^3 = \delta_n^2 - \delta_{n-1}^2$$

t_n, δ_n, δ_n^2 e δ_n^3 são palavras de 32 bits. A quantidade de entropia adicionada é o evento definido pelo $log_2(minimo(|\delta_n|, |\delta_n^2|e|\delta_n^3|)_{[19-30]})$, onde $S_{[a-b]}$ indica os bits da posição a até a posição b, sendo o 0 o bit mais significativo. Se $minimo(|\delta_n|, |\delta_n^2|e|\delta_n^3|)_{[19-30]} = 0$, então a estimação vale 0. Se diferente de zero, aumenta-se o valor do contador da *pool* onde os bits foram adicionados.

Mais uma vez, é importante ressaltar que a estimação é utilizada apenas nos casos em que os bits adicionados vem da observação dos eventos de sistema, pois quando vem da *pool* primária, os contadores da *pool* secundária e da *pool urandom* são apenas incrementados da quantidade de bits transferidos.

6.4.5 ATUALIZAÇÃO DAS *POOLS*

A coleta de entropia para a *pool* primária e a consequente estimação de entropia foram assuntos das SUBSEÇÕES 6.4.3 e 6.4.4. Esta subseção tem o propósito de indicar como as atualizações das três *pools* é feita. Essa estimação do valor da entropia ou o simples valor adicionado aos contadores, no caso de bits transferidos da *pool* primária para as *pools* secundária e *urandom*, passa por operações matemáticas, resultando no fim em um novo valor de bits para serem adicionados a *pool* correspondente. Esta operação é feita quando há requisição de bits (representada pela letra A na FIGURA 6.6), quando há coleta de bits (representada pela letra C) e quando há extração de bits (representada pela letra E) (GUTTERMAN, 2006).

Esta operação segue a forma (o modelo exato varia conforme a *pool*) do algoritmo *add(pool,j,g)* encontrado no APÊNDICE 9.2. A entropia adicionada aos contadores ou a entropia estimada é representada pela letra g e o estado atual da *pool* é representado pelo índice j. Maiores informações em (GUTTERMAN, 2006; BARAK, 2005; DERAADT, 2003).

6.4.6 EXTRAÇÃO DE BITS

A entropia é extraída a partir da *pool* secundária se utilizado o */dev/random* e da poll *urandom* se utilizado */dev/urandom* ou o comando *get_ random_ bytes*. Também é extraída a partir da *pool* principal para efeito de alimentar as outras polls. Extrair entropia de uma poll não é simples. Envolve operações hash dos bits extraídos, modificando

o estado da *pool* e decrementando a entropia existente pelo número de bits extraídos. O APÊNDICE 9.3 apresenta o algoritmo que o LRNG utiliza quando bits são solicitados. A FIGURA 6.7 apresenta o seu diagrama.

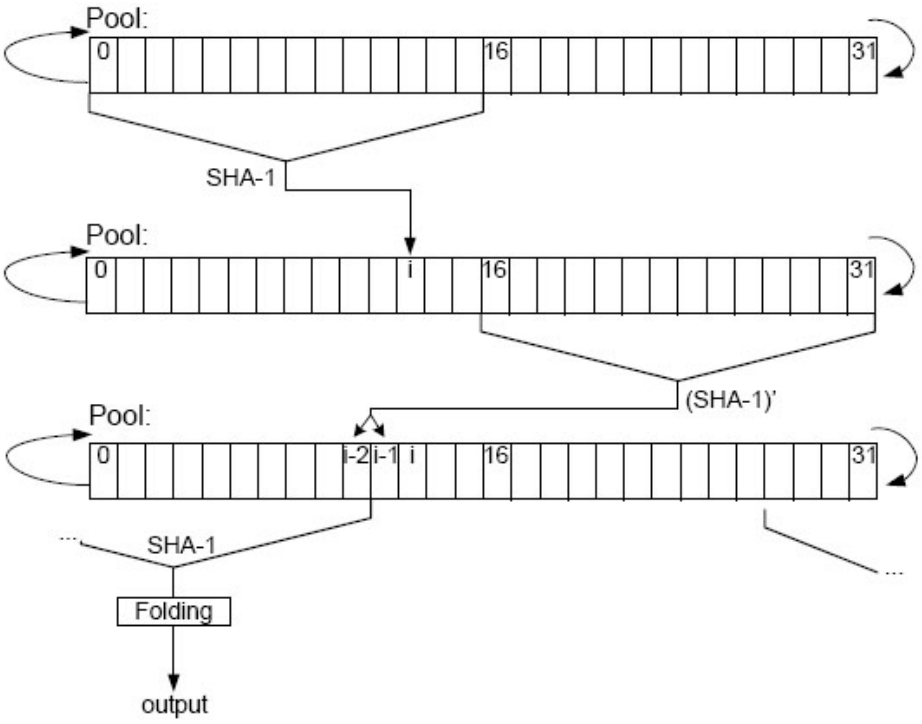

FIG. 6.7: Diagrama da extração de bits realizada pelo LRNG.

A extração é feita em blocos de 10 bytes. O processo é descrito para o caso das *pools urandom* ou secundária, que utilizam palavras de 32 bits. Para simplificar a descrição não estão incluídas as etapas de decremento da estimativa da entropia e o processo de recarga desta. O algoritmo aplica a função SHA-1 para as primeiras 16 palavras e adiciona o resultado no estado de índice j. É então aplicada uma variação do SHA-1, que será chamado SHA-1' na metade direita do *pool*. Após isso acrescenta partes do resultado para os estados de índices j-1 e j-2. Finalmente, ele se aplica SHA-1' para as 16 palavras que finalizam o índice j-2 e utiliza o resultado para calcular a saída da seguinte maneira:

- A saída do SHA-1' é de 5 palavras (20 bytes) de comprimento;
- Estas são comprimidas em 2,5 palavras de comprimento através de outra operação matemática, descrita no APÊNDICE 9.4;
- Estas são os 10 bytes de saída deste ciclo de operação;
- Esta saída final é copiada para quem solicitou os bits (kernel ou usuário);

- O ciclo continua até o número de bytes solicitados.

A mencionada variante da função SHA-1 é obtida modificando as contantes de entrada do SHA-1 original a cada loop efetuado.

Para o algoritmo de extração da *pool* principal há uma pequena diferença, o SHA-1 (ou SHA-1') é aplicada a cada um dos oito blocos de 16 palavras da *pool* primária e uma vez mais as 16 palavras que estão no índice j-8. As primeiras oito operações SHA-1 atualizam oito palavras da *pool* e a última operação gera a saída (GUTTERMAN, 2006).

Estas operação são internas ao Sistema Operacional Linux, não prescisando serem operacionalizadas pelo usuário. Um método eficiente de coletar-se bits pseudo-aleatórios seguros, ou seja, utilizando o */dev/random*, é através do seguinte comando:

$$cat\ /etc/random > arquivoDEbits.bin$$

Com esse comando, o sistema lê os bits existentes na *pool* secundária e os transfere para o arquivo arquivoDEbits.bin, que será depois utilizado. O */dev/random* também pode ser atingido pelo caminho */proc/sys/kernel/random*. Neste diretório são encontrados diversos arquivos importantes de configuração e acesso do LRNG, como o arquivo *entropy_avail* que indica a entropia disponível ou o arquivo *poolsize* que pode ser manipulado para modificar o tamanho da *pool* (THEODORE, 2008).

Para testar os números pseudo-aleatórios, coletados pelo processo acima descrito, foram utilizados alguns testes de aleatoriedade que serão descritos no CAPíTULO 4. Tais resultados encontram-se na SEÇÃO 6.6.1 e foram considerados satisfatórios.

6.5 TESTES PARA ALGORITMOS CRIPTOGRÁFICOS

6.5.1 INTRODUÇÃO

Como explanado na SEÇÃO 4.2, os testes aqui apresentados serão voltados para algoritmos sequenciais. Na literatura e no material científico pesquisado, diversos métodos de avaliação de cifras foram encontrados, porém a grande maioria voltava-se para a avaliação de algoritmos baseados em estruturas iteradas (também chamadas de estruturas Feistel)[31]. Esta facilidade de se encontrar algoritmos de testes para estruturas iteradas

[31]Feistel concluiu que à medida que o texto em claro atravessa os diversos níveis de transformação ao longo da função criptográfica, o efeito da modificação de um bit na entrada é amplificado em uma avalanche imprevisível. Ao final do processo, em média, metade dos bites da saída deve ser igual 0 e metade igual a 1(FEISTEL, 1973)

e consequente dificuldade de se encontrar os mesmos algoritmos para outras estruturas deve-se a clara tendência da comunidade científica em trabalhar com algoritmos baseados em estruturas iteradas.

Para estes, diversos testes podem ser encontratados, entre eles:

- **Resistência contra criptoanálise linear**, ou seja, resistência contra o ataque criado por Mitsuru Matsui (MATSUI, 1994), que utiliza aproximações lineares[32] para descrever a ação de um cifrador de blocos. Uma apresentação detalhada pode ser verificada em (LIMA, 1999; CASTANO, 1998);

- **Resistência contra criptoanálise diferencial**. A criptoanálise diferencial baseia-se na análise de pares de textos cifrados pela mesma chave e oriundos de textos em claros que possuem diferenças específicas. Ela avalia a evolução dessas diferenças, conforme os textos em claros são transformados, a cada rodada do algoritmo iterado a ser testado[33];

- **Avaliação do efeito avalanche e da completude**. Em um algoritmo iterado, a cada iteração, a associação de transformações lineares e não-lineares existentes nele deve fazer com que cada vez mais bits de saída dependam do valor de mais bits de entrada, produzindo o chamado efeito avalanche. Quando atinge-se o estado[34] em que qualquer bit de saída dependa do valor de todos os bits de entrada diz-se que a função é completa. Esses conceitos, avalanche e completude, são amplamente difundidos (BIHAM, 1993; FEISTEL, 1973; SCHENEIER, 1996) e aceitos por especialistas como necessário a qualquer função criptográfica iterada. Uma técnica simples para se fazer estas avaliações é utilizando o conceito de matriz de dependências (LAMBERT, 2004).

Para os encriptadores sequenciais, no entanto, os principais testes são baseados na aleatoriedade da saída do sistema e são nesses testes que este trabalho será baseado para avaliação de seu projeto.

[32]Aproximações lineares são equações lineares de operações XOR de bits do texto em claro, do criptograma e de uma chave provável cuja probabilidade de acontecer seja $p \neq 1/2$

[33]Essa técnica foi apresentada por Eli Bihan e Adi Shamir em 1990 e pode ser melhor conhecida em (BIHAM, 1991)

[34]número de ietrações

6.5.2 TESTES DE ALEATORIEDADE

No FIPS 140-1, testes estatísticos de aleatoriedade (NIST, 1994b), são especificados quatro testes de aleatoriedade de forma bastante objetiva. Os testes são descritos nas SEÇÕES 6.5.2.2 a 6.5.2.4 a seguir e são baseados na distribuição χ^2, descrita na SEÇÃO 6.5.2.1. Outro ponto importante deste documennto é que o NIST, em vez de deixar que o usuário escolha usar níveis de significância estatística para os testes, o NIST definiu limites para as frequências analisadas na estatística dos testes, estatísticas estas descritas na SEÇÃO 6.5.2.5.

6.5.2.1 A DISTRIBUIÇÃO QUI-QUADRADO

A distribuição Qui-Quadrado é um caso especial importante da distribuição Gama(BUSSAB, 2002), que tem sua função densidade de probabilidade dada por:

$$f(x;\alpha,\beta) = \frac{1}{\Gamma(\alpha)\beta^{\alpha}} x^{\alpha-1} e^{\frac{-x}{\beta}},\ x > 0$$

$$f(x;\alpha,\beta) = 0,\ x < 0\ e\ \alpha \geq 1\ e\ \beta > 0$$

Pode-se demonstrar que E(X)[35]=$\alpha\beta$ e Var(X)[36]=$\alpha\beta^2$

Para se obter a função densidade de probabilidade da distribuição Qui-Quadrado, substitui-se, na função densidade de probabilidade da distribuição Gama, α por $\frac{v}{2}$ e β por 2, com $v > 0$ inteiro, obetndo-se:

$$f(y;v) = \frac{1}{\Gamma(\frac{v}{2})2^{\frac{v}{2}}} y^{\frac{v}{2}-1} e^{\frac{-y}{2}},\ y > 0$$

$$f(y;v) = 0,\ y < 0$$

Tem-se consequentemente, E(Y)=v e Var(Y)=2v.

O gráfico para esta distribuição, segundo diferentes graus de liberdade, se encontra na FIGURA 6.8, porém o meio mais comum de se obter valores para ela é utilizando a TABELA 9.1, encontrada no APÊNDICE 9.5.

6.5.2.2 TESTE DE FREQUÊNCIA

Seja $s = s_0|s_1|s_2|...|s_{n-1}$ uma sequência binária de tamanho n. O objetivo deste teste é verificar se o número de bits iguais a zero é aproximadamente igual ao número de bits

[35] média

[36] variância

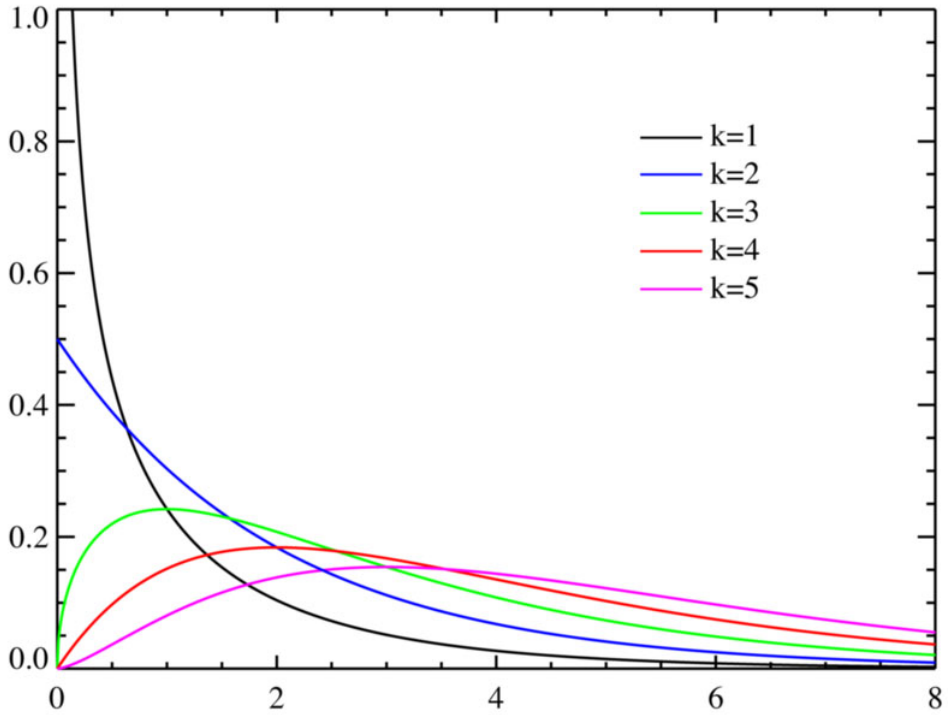

FIG. 6.8: Distribuição Qui-Quadrado para os graus de liberdade v = k = 1, 2, 3, 4 e 5.

iguais a um, como seria esperado em uma sequência aleatória. Sejam n_0 o número de bits iguais a zero e n_1 o número de bits iguais a um. A estatística utilizada é a seguinte:

$$X_1 = \frac{(n_0 - n_1)^2}{n}$$

Esta estatística deve seguir aproximadamente uma distribuição χ^2 com 1 grau de liberdade se $n > 10$, porém na prática recomenda-se $n \geq 20000$.

Percebe-se que, para $n = 20000$, o ideal seria a ocorrência de 10000 bits iguais a zero e 10000 bits iguais a um. Analisando a estatística acima, chega-se a conclusão que se $n_0 = n_1 = 10000$, $X_1 = 0$, logo o ideal é que X_1 seja o mais próximo possível de zero. Se a essa análise for incluída a TABELA 9.1, percebe-se que para $v = 1$, tem-se:

- $P(X_1 > 0,001) = 97,5\%$;
- $P(X_1 > 3,841) = 5\%$;e
- $P(X_1 > 10,827) = 0,1\%$.

Daí, a forma mais correta de fazer o teste de frequência seria gerar o maior número possível de sequências aleatórias com $n > 10$, calcular a estatística de cada uma das sequências e ver se elas seguem a distribuição dada(no caso χ^2 com 1 grau de liberdade), ou seja, ver se 97,5% das sequências tem a sua estatística(X_1) maior que 0.001, se 5%

tem a sua estatística maior que 3.841, se 0,1% tem sua estatística maior que 10,827 e assim por diante conforme valores encontrados na TABELA 9.1.

Devido a dificuldade em se gerar tantas sequências para serem avaliadas, o NIST definiu, baseado na distribuição Qui-quadrado, limites para este teste de aleatoriedade. Por exemplo, para uma distribuição de 20000 bits, tanto o número de bits iguais a zero quanto o número de bits iguais a um devem estar entre os valores 9654 e 10346[37].

Raciocínio idêntico foi utilizado para cada um dos testes de aleatoriedade descritos a seguir. Os limites definidos pelo NIST para estes testes serão apresentados na SEÇÃO 6.5.2.5.

6.5.2.3 TESTE POKER

Seja m um inteiro positivo, tal que $\frac{n}{m} \geq 5.2^m$, e seja k o maior inteiro menor ou igual a $\frac{n}{m}$. Divide-se a sequência s em k blocos de tamanho m(sem superposição) e representa-se por n_i o número de ocorrências do *i-ésimo* tipo de sequência de tamanho $m(1 \leq i \leq 2^m)$. O teste poker determina se cada sequência de tamanho m aparece o mesmo número de vezes, como seria esperado em uma distribuição aleatória. A estatística utilizada é a seguinte:

$$X_2 = (\frac{2^m}{k}) \sum_{i=1}^{i=2^m} n_i^2 - k$$

Esta estatística deve seguir aproximadamente uma distribuição χ^2 com $2^m - 1$ graus de liberdade. Esse teste é uma generalização de teste de frequência apresentado em 6.5.2.2.

6.5.2.4 TESTE DE SEQUÊNCIAS CORRIDAS

O objetivo deste teste é verificar se o número de ocorrências de sequências compostas somente de bits iguais a zero(delimitadas a esquerda e a direita pelo bit 1) ou somente bits iguais a 1(delimitadas a esquerda e a direita pelo bit 0) é compatível com uma distribuição aleatória.

O número esperado de tais sequências de comprimento i em uma sequência de tamanho n é $e_i = \frac{(n-i+3)}{2^{i+2}}$. Seja k o maior inteiro i para o qual $e_i \geq 5$. Sejam U_i e Z_i os números de sequências de uns e zeros de tamanho i presentes em n, respectiva-

[37]Perceber que se o número de bits iguais a zero estiver dentro do intervalo o número de bits iguais a um também estará

mente, para cada $i(1 \leq i \leq k)$. A estatística usada é:

$$X_3 = \sum_{i=1}^{i=k} \frac{(U_i - e_i)^2}{e_i} + \sum_{i=1}^{i=k} \frac{(Z_i - e_i)^2}{e_i}$$

Esta estatística deve seguir aproximadamente uma distribuição χ^2 com $2k-2$ graus de liberdade.

Adicionalmente a esse teste, é verificado o tamanho máximo encontrado em uma sequência corrida, seja de zeros ou uns. A sequência é aprovada se o tamanho máximo não passa de um determinado valor.

6.5.2.5 LIMITES DEFINIDOS PELO NIST

Como explicado em 6.5.2.2, a dificuldade em se gerar a quantidade de sequências necessárias para o completo teste destas, levou o NIST a definir limites para cada um dos testes acima descritos. Esses limites foram calculados para serem utilizados como parâmetros de testes de sequências de 20000 bits. Caso algum dos testes dê negativo, a sequência é rejeitada.

a) **Teste de frequência** - Como já explicado, o número n_1 de bits iguais a um em s deve ser tal que $9654 \leq n_1 \leq 10346$;

b) **Teste Poker** - A estátística X_2, computada para $m = 4$, deve ser tal que $1,03 \leq X_2 \leq 57,4$;

c) **Teste de sequências corridas** - Os números U_i e Z_i de tamanho i são contados para cada i[38]. A sequência é aprovada se as contagens respeitarem os limites encontrados na TABELA 6.3. Além destes limites, é necessário que não haja qualquer sequência corrida, seja de zeros ou uns, maior que 33.

Tamanho da sequência	Valores permitidos para U_i e Z_i
1	2267 a 2733
2	1079 a 1421
3	502 a 748
4	223 a 402
5	90 a 223
6	90 a 223

TAB. 6.3: Valores permitidos para as contagens do teste de sequências corridas

[38] $1 \leq i \leq 6$. As sequências encontradas de tamanho maior do que 6 são consideradas iguais a 6

6.6 RESULTADO DOS TESTES ALEATÓRIOS

Os testes aleatórios foram realizados em dois tipos de sequências: a primeira, que era composta por bits procedentes do gerador pseudo-aleatório do Linux e a segunda, que era composta pela saída do sistema, após a criptografia. Dessa maneira, procura-se atestar o bom comportamento aleatório tanto do gerador pseudo-aleatório quanto do resultado da encriptação. Foram utilizados para ambos os casos 20000 bits, conforme explicado em 6.5.2.5. Os códigos utilizados encontram-se nos APÊNDICES 9.6, 9.7 e 9.8.

6.6.1 RESULTADOS PARA SEQUÊNCIA PROCEDENTE DO GERADOR PSEUDO-ALEATÓRIO

Resultado para o teste de frequência

Listagem 6.1: Resultado do teste de frequência

```
Nr de bits iguais a zero = 10008
Nr de bits iguais a um = 9992
9654 < bits 1's < 10346
APROVADO
```

Resultado para o teste poker

Listagem 6.2: Resultado do teste poker

```
Nr de sequencias iguais a 0000 = 297
Nr de sequencias iguais a 0001 = 334
Nr de sequencias iguais a 0010 = 312
Nr de sequencias iguais a 0011 = 321
Nr de sequencias iguais a 0100 = 315
Nr de sequencias iguais a 0101 = 306
Nr de sequencias iguais a 0110 = 293
Nr de sequencias iguais a 0111 = 334
Nr de sequencias iguais a 1000 = 306
Nr de sequencias iguais a 1001 = 355
Nr de sequencias iguais a 1010 = 280
Nr de sequencias iguais a 1011 = 326
Nr de sequencias iguais a 1100 = 323
Nr de sequencias iguais a 1101 = 310
Nr de sequencias iguais a 1110 = 293
Nr de sequencias iguais a 1111 = 295
Total de sequencias de 4 bits = 5000
estatistica = 17.78
```

```
1.03 < Estatistica < 57.4
APROVADO
```

Resultado para o teste de sequências corridas

Listagem 6.3: Resultado do teste de sequências corridas

```
TESTE DE SEQUENCIAS CORRIDAS
O numero de sequencias corridas que contem 1 bits 0 = 2446
2267 < 2446 < 2733
APROVADO
O numero de sequencias corridas que contem 1 bits 1 = 2460
2267 < 2460 < 2733
APROVADO
O numero de sequencias corridas que contem 2 bits 0 = 1300
1079 < 1300 < 1421
APROVADO
O numero de sequencias corridas que contem 2 bits 1 = 1263
1079 < 1263 < 1421
APROVADO
O numero de sequencias corridas que contem 3 bits 0 = 645
502 < 645 < 748
APROVADO
O numero de sequencias corridas que contem 3 bits 1 = 660
502 < 660 < 748
APROVADO
O numero de sequencias corridas que contem 4 bits 0 = 297
223 < 297 < 402
APROVADO
O numero de sequencias corridas que contem 4 bits 1 = 312
223 < 312 < 402
APROVADO
O numero de sequencias corridas que contem 5 bits 0 = 163
90 < 163 < 223
APROVADO
O numero de sequencias corridas que contem 5 bits 1 = 154
90 < 154 < 223
APROVADO
O numero de sequencias corridas que contem 6 bits 0 = 146
90 < 146 < 223
APROVADO
O numero de sequencias corridas que contem 6 bits 1 = 147
90 < 147 < 223
```

APROVADO
RESULTADO TESTE DE SEQUENCIAS CORRIDAS:APROVADO

TESTE DE TAMANHO MAXIMO DE SEQUENCIA
contMAX = 16
RESULTADO TESTE DE TAMANHO MAXIMO DE SEQUENCIA:APROVADO

6.6.2 RESULTADOS PARA SEQUÊNCIA PROCEDENTE DA SAÍDA DO SISTEMA

Resultado para o teste de frequência

Listagem 6.4: Resultado do teste de frequência

```
Nr de bits iguais a zero = 9928
Nr de bits iguais a um = 10072
9654 < bits 1's < 10346
APROVADO
```

Resultado para o teste poker

Listagem 6.5: Resultado do teste poker

```
Nr de sequencias iguais a 0000 = 319
Nr de sequencias iguais a 0001 = 311
Nr de sequencias iguais a 0010 = 319
Nr de sequencias iguais a 0011 = 315
Nr de sequencias iguais a 0100 = 288
Nr de sequencias iguais a 0101 = 290
Nr de sequencias iguais a 0110 = 284
Nr de sequencias iguais a 0111 = 339
Nr de sequencias iguais a 1000 = 316
Nr de sequencias iguais a 1001 = 329
Nr de sequencias iguais a 1010 = 310
Nr de sequencias iguais a 1011 = 290
Nr de sequencias iguais a 1100 = 307
Nr de sequencias iguais a 1101 = 340
Nr de sequencias iguais a 1110 = 311
Nr de sequencias iguais a 1111 = 332
Total de sequencias de 4 bits = 5000
estatistica = 14.98
1.03 < Estatistica < 57.4
APROVADO
```

Resultado para o teste de sequências corridas

Listagem 6.6: Resultado do teste de sequências corridas

```
Tamnho do arquivo (em bytes) = 625
TESTE DE SEQUENCIAS CORRIDAS
O numero de sequencias corridas que contem 1 bits 0 = 2490
2267 < 2490 < 2733
APROVADO
O numero de sequencias corridas que contem 1 bits 1 = 2459
2267 < 2459 < 2733
APROVADO
O numero de sequencias corridas que contem 2 bits 0 = 1236
1079 < 1236 < 1421
APROVADO
O numero de sequencias corridas que contem 2 bits 1 = 1225
1079 < 1225 < 1421
APROVADO
O numero de sequencias corridas que contem 3 bits 0 = 655
502 < 655 < 748
APROVADO
O numero de sequencias corridas que contem 3 bits 1 = 647
502 < 647 < 748
APROVADO
O numero de sequencias corridas que contem 4 bits 0 = 291
223 < 291 < 402
APROVADO
O numero de sequencias corridas que contem 4 bits 1 = 320
223 < 320 < 402
APROVADO
O numero de sequencias corridas que contem 5 bits 0 = 145
90 < 145 < 223
APROVADO
O numero de sequencias corridas que contem 5 bits 1 = 155
90 < 155 < 223
APROVADO
O numero de sequencias corridas que contem 6 bits 0 = 155
90 < 155 < 223
APROVADO
O numero de sequencias corridas que contem 6 bits 1 = 167
90 < 167 < 223
APROVADO
RESULTADO TESTE DE SEQUENCIAS CORRIDAS:APROVADO

```

TESTE DE TAMANHO MAXIMO DE SEQUENCIA

contMAX = 14

RESULTADO TESTE DE TAMANHO MAXIMO DE SEQUENCIA:APROVADO

7 CONCLUSÃO

7.1 SOBRE ESTE TRABALHO

Este trabalho procurou apresentar, sempre da forma mais objetiva e clara possível, uma nova idéia de técnica criptográfica baseada em outra técnica amplamente conhecida e difundida, o *one-time pad*.

No CAPÍTULO 1, após breve histórico da criptografia e exaltação de sua importância, explicitou-se os objetivos principais desta dissertação. No CAPÍTULO 2, como o trabalho trata de criptografia sequencial, foi apresentada uma introdução as primeiras técnicas básicas utilizadas pela humanidade nessa área, para posterior explicação do criptosistema perfeito, o *one-time pad*. Ainda nesse capítulo, foram feitas considerações a respeito da impossibilidade prática do seu uso.

Após estes aspectos introdutórios, foi apresentado, no CAPÍTULO 3, o projeto em si, discutindo-se e justificando-se cada parte do sistema: a utilização da operação soma módulo como alternativa e a escolha da função SHA-256 como a função não-inversível necessária para a segurança do algoritmo. Já no CAPÍTULO 6, discorreu-se a respeito da geração aleatória e pseudo-aleatória de bits e definiu-se o sistema que foi escolhido por este projeto para ser o gerador de suas chaves. No CAPÍTULO 4 foi explicado passo a passo quais critérios de avaliação seriam usados e o porque deles serem escolhidos. Finalmente, no CAPÍTULO 5, a análise discutida anteriormente foi realizada, e os resultados, que foram satisfatórios, evidenciados.

7.2 CONTRIBUIÇÕES DESTE TRABALHO

As contribuições, principais e secundárias, proporcionadas por este trabalho são:

- Foi apresentada uma nova técnica de criptografia, inspirada em outra bem conhecida e difundida, que, a princípio, não possui qualquer contra-indicação de uso. Esta passou por todos os testes aos quais foi submetida e parece cumprir todos os cinco princípios exigidos. A maior limitação desta técnica parece estar realmente na taxa existente hoje para a geração de sequências verdadeiramente aleatórias. Outra limitação é o fato da técnica apresentada utilizar no máximo $\frac{1}{3}$ da banda disponível no canal de comunicação para efetiva transmissão da mensagem, já que os outros

$\frac{2}{3}$ são utilizados para transmissão da chave. Essa escolha, porém, foi resultado de ser dada prioridade à segurança em detrimento da eficiência;

- Foi apresentado um estudo bem completo, proveniente de diversas fontes, sobre o uso do gerador pseudo-aleatório disponibilizado pelo Linux, que ajudará na compreensão do funcionamento desse gerador, visto que o mesmo tem pouca documentação, e, por isso, é de difícil compreensão;

- Foi apresentado o funcionamento detalhado da função SHA-256, o que facilitará o entendimento do funcionamento deste projeto;

- O projeto resultante desta dissertação foi devidamente implementado na linguagem de programação C, o que permite sua utilização em diversas plataformas de Sistemas Operacionais; e

- Os teste aleatórios também foram implementados e seus códigos estão disponíveis neste trabalho, o que facilitará qualquer pessoa que deles necessite.

7.3 POSSIBILIDADES DE CONTINUIDADE DE PESQUISA

Este trabalho, apesar de ter passado por diversas fontes de estudo e ter discorrido sobre diversos assuntos, não conseguiu esgotar todas as possibilidades. Daqui, então, podem continuar ou mesmo surgir estudos ainda não finalizados ou iniciados. Entre eles, pode-se sugerir:

- Verificar novas técnicas existentes, que possam ser úteis para uma análise mais completa a respeito deste projeto;

- Analisar o sistema sugerido utilizando-se novas funções no lugar do SHA-256 verificando como o sistema responde a essas funções;

- Acompanhar a criação de novos geradores verdadeiramente aleatórios com taxas de geração superiores às atuais, o que tornaria o sistema mais eficiente;

- Implementar o projeto em hardware, o que por si só aumentaria a eficiência do mesmo, e com isso acoplar aos geradores verdadeiramente aleatórios, que são oriundos de hardware, para futuros testes;

- Evidenciar as principais diferenças entre usar operações XOR e soma módulo, determinando com mais propriedade o porquê delas, definindo os casos em que seria mais propício utilizar uma ou outra opção; e
- Otimizar as rotinas criadas em C ou mesmo criar novas que executem o trabalho com mais eficiência e rapidez.

8 REFERÊNCIAS BIBLIOGRÁFICAS

AISOPOS, K., KAKAROUNTAS, A. P., MICHAIL, H. e GOUTIS, C. E. **High throughput implementation of the new Secure Hash Algorithm through partial unrolling**. *SIPS 2005*, 2005.

BARAK, B. e HALEVI, S. **A model and architecture for pseudo-random generation with applications to /dev/random**. *ACM Conference on Computer and Communications Security*, págs. 203–212, 2005.

BEUTELSPACHER, A. ***Cryptology***. Mathematical Association of America, first edition, 1994.

BIHAM, E. **A Note on Comparing the AES Candidates**. Technical report, http://secinf.net/cryptography/A_Note_on_Comparing_the_AES_Candidates%_Revised_Version.html, 1999.

BIHAM, E. e SHAMIR, A. **Differential Cryptanalysis of DES-like Cryptosystems**. *Journal of Crypotology*, 4(1):3–72, 1991.

BIHAM, E. e SHAMIR, A. **Differential Cryptanalysis of the Data Encryption Standard**. *Springer-Verlag*, 1993.

BLAZE, M., DIFFIE, W., RIVEST, R., SCHENEIER, B., SIMOMURA, T., THOMPSON, E. e WIENER, M. **Minimal Key Lenghts for Symmetric Ciphers to Provide Adequate Commercial Security. A report by an Ad Hoc group of cryptographers and computer scientists.** Technical report, http://www.bsa.or/policy/encryption/cryptographers.html.

BORGES JR., E. C. V. e DIAS, C. P. S. ***Geração de Números aleatórios, pseudo-aleatórios e suas aplicações em criptografia***. Projeto fim de curso, Instituto Militar de Engenharia, 2008.

BUSSAB, W. O. e MORETIN, P. A. ***Estatística Básica***. Editora Saraiva, quinta edition, 2002.

CASTANO, F. C. ***Análise das Cifras Rijndael e Serpent contra as Criptoanálises Linear e Diferencial***. Dissertação de mestrado, rio de janeiro, rj, brasil, 1998.

COUTINHO, S. C. ***Números inteiros e criptografia RSA***. IMPA, 2005.

COVER, T. M. e THOMAS, J. A. ***Elements of Information Theory***. Jonh Wiley & Sons, 1991.

DE RAADT, T., HALLQVIST, N., GRABOWSKI, A., KEROMYTIS, A. D. e PROVOS, N. **Extracting randomness from external interrupts**. The IASTED International Conference on Communication, Network and Information Security, págs. 141–146, 2003.

DEN BOER, B. e BOSSELAERS, A. **An Attack on the last two rounds of MD4**. *Advances in Cryptology Proceedings*, págs. 194–203, 1992.

DLINK. **Placa de rede 1Gbps**. Technical report, http://www.dlink.com, 2008.

FEISTEL, H. **Cryptography and computer Privacy**. In Scientific American, 228(5), 1973.

GARCIA, A. L. ***Probability and Random Processes for Electrical Engineering - Second Edition***. Addison-Wesley Publishing Company, 1994.

GUTTERMAN, Z., PINKAS, B. e REINMAN, T. **Analysis of the Linux Random Number Generator**. 2006.

INAYATULLAH, M., AHMAD, M. e KHAN, Q. **Message based Selection Matrix for Dynamic Hash based Security Systems**. *International Conference on Emerging Technologies*, 2007.

KAHN, D. ***The Codebreakers: The Story of Secret Writing***. Macmillan Publishing Co., 1967.

KNUDSEN, L. e HAVARD, R. **Recommendation to NIST for the AES**. Technical report, 2000.

LAMBERT, J. ***Cifrador Simétrico de Blocos: Projeto e Avaliação***. Dissertação de mestrado, Instituto Militar de Engenharia, 2004.

LIMA, A. P. e CASTAÑO, F. **Criptoanálise Linear**. RT034/DE9FEV99 - Instituto Militar de Engenharia, Rio de Janeiro, RJ, Brasil, 1999.

MATSUI, M. **Linear Cryptanalysis Method for DES cipher**. Advances in Cryptology: proceedings of EUROCRYOPT'93, Springer-Verlag, Berlim, págs. 386–397, 1994.

MERKLE, R. ***Secrecy Authentication, and Public Key Systems***. Dissertação (ph.d.), Stanford University, 1979.

MERKLE, R. **A Fast Software One-Way Hash Function**. *Journal of Cryptology*, págs. 43–58, 1990.

MIYAGUCHI, E., OHTA, K. e IWATA, M. **128-bit Hash Function(N-Hash)**. *Proceedings of securicom*, págs. 127–137, 1990.

NECHVATAL, J., BARKER, E., DODSON, D., DWORKIN, M., FOTI, J. e ROBACK, E. **Status Report on the 1^{st} Round of the Development of the Advanced Encryption Standard**. Journal of Research of the National Institute of Standards and Technology, 104(5), 1999.

NIST. **Digital Signature Standard**. *NIST FIPS PUB 186*, 1994a.

NIST. **SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES**. *Federal Information Processing Standards Publication 140-1*, 1994b.

NIST. **Secure Hash Standard**. *Federal Information Processing Standards Publication 180-2*, 2002.

PRENEEL, B. ***Analysis and Design of Cryptographic Hash Functions***. Dissertação (ph.d.), Katholieke Universiteit Leuven, 1993.

PRENEEL, B. **NESSIE Announces Final Selection of Cripto Algorithms**. Technical report, http://www.cryptonessie.org, 2003.

REGISTER, F. **Proposed Federal Information Processing Standard for Secure Hash Standard**. *Federal Register*, 57(21):3747–3749, 1992.

RIVEST, R. **The MD4 Message Digest Algorithm**. *RFC 1186*, 1990.

RIVEST, R. **The MD5 Message Digest Algorithm**. *RFC 1321*, 1992.

ROBSHAW, M. **Implementations of the search for Pseudo-Collisions in MD5**. *Technical Report TR-103*, 1993.

ROBSHAW, M. **MD2, MD4, MD5, SHA, and Other Hash Functions**. *Technical Report TR-101*, 1994.

RUEPPEL, R. A. **Analysis and design of stream ciphers**. *Springer-Verlag*, 1986.

SCHENEIER, B. ***Applied Cryptography***. John Wiley & Sons Ltd., 2ª edition, 1996.

SCHENEIER, B. **SHA-1 broken**. Technical report, http://www.schneier.com/blog/archives/2005/02/sha1_broken.html, 2005.

SHANNON, C. **Communication theory of secrecy systems**. *Bell Systems Technical Journal*, 28:656–715, 1949.

SINGH, S. ***O Livro dos Códigos***. Ed. Record, 6ª edition, 2007.

THEODORE, T. **random**. Technical report, http://linux.die.net/man/4/random, 2008.

TOSHIBA. **Generation random number at 2Mbit per second with small circuit only 1200 square micrometers in area**. Technical report, http://www.toshiba.co.jp/about/press/2008_02/pr0702.htm, 2008.

UNICAMP. **A fibra que alimentou a inovação completa 30 anos**. Technical report, http://www.unicamp.br/unicamp/divulgacao/2007/05/21/a-fibra-que-alimentou-a-inovacao-completa-30-anos, 2008.

VICKISOFT. **FREQUÊNCIA DA OCORRÊNCIA DE LETRAS NO PORTUGUÊS** . Technical report, http://www.numaboa.com.br/criptologia/matematica/estatistica/freqPortBrasil.php, 2004.

ZEGHID, M., BOUALLEGUE, B., BAGANNE, A., MACHHOUT, M. e TOURKI, R. **A reconfigurable Implementation of the New Secure Has Algorithm**. *Internatonal Conference on Availability, Reliability and Security - ARES'07*, 2007.

ZIERLER, N. **On primitive trinomials (mod 2)**. *information and Control, v.13*, págs. 541–544, 1968.

ZIERLER, N. **Primitive trinomials whose degree is a mersenne exponent**. *information and Control, v.15*, págs. 67–69, 1969.

9 APÊNDICES

Listagem 9.1: Programa principal - encriptação

```
#include <stdio.h>
#include<conio.h>
#include<math.h>

#define FourBytes unsigned int

FourBytes ChaveSHA[8];

#include "sha256.h"



FourBytes K1[] = {
0xd0efaafb,0x434d3385,0x45f9027f,0x503c9fa8,0xcd0c13ec,0x5f974417,0
    xc4a77e3d,0x645d1973,
0xe0323a0a,0x4906245c,0xc2d3ac62,0x9195e479,0x8ca1890d,0xbfe64268,0
    x41992d0f,0xb054bb16
};

//entrada do tipo => programa.exe mensagem.txt chave.bin
int main(int argc, void *argv[])
{
FILE *Input;
FILE *Key;
FILE *Out;
//Trabalharar-se-á com 256 bits por vez(256/32 = 8). Virá do arquivo
    mensagem.txt
FourBytes Entrada[8];
//A chave terá tamanho 512 bits(512/32 = 16). Virá do arquivo chave.txt
FourBytes Chave[16];
FourBytes *bufferChave;
FourBytes *bufferMensagem;
FourBytes SomaChaveK1[16];
FourBytes SomaChaveSHAmsg[8];
int i,j,k,carry,lSize,y;
float x;

if(argc<3){
```

```
printf("FORMATO: programa.exe mensagem.txt chave.bin");
return 1;
}
if( (Input=fopen((char *) argv[1],"rb" ))==NULL ) {
printf("Erro ao abrir arquivo %s.",argv[1]);
return 2;
}
if( (Key=fopen((char *) argv[2],"rb" ))==NULL ) {
printf("Erro ao abrir arquivo %s.",argv[2]);
return 3;
}
if( (Out=fopen("encriptado.bin","wb" ))==NULL ) {
printf("Erro ao criar arquivo encriptado.bin.");
return 4;
}
fclose(Out);

fseek (Input , 0 , SEEK_END);
lSize = ftell (Input);
rewind (Input);
bufferMensagem = (unsigned int*) malloc (sizeof(int)*lSize);
fread (bufferMensagem,1,lSize,Input); //tamanho de lSize em Bytes

fseek (Key , 0 , SEEK_SET); //Coloca o ponteiro do arquivo da chave no
    início

x = lSize/32.0;
y = ceil(x);

for (j=0;j<y;j++){
    for (i=0;i<8;i++){
        Entrada[i]=bufferMensagem[8*j+i];
    }

    bufferChave = (unsigned int*) malloc (sizeof(int)*16);
    fseek (Key , 0 , SEEK_CUR);
    fread (bufferChave,4,16,Key);
    for(k=0;k<16;k++){
        Chave[k]=bufferChave[k];
    }

```

```
        carry = 0;
        Out=fopen("encriptado.bin","ab" );
        for(k=0;k<16;k++){                  //SOMA MÓDULO K1+CHAVE
            SomaChaveK1[k] = K1[k]+Chave[k]+carry;

            fwrite (&SomaChaveK1[k],4,1,Out);
            if ((K1[k]+Chave[k]+carry) < (Chave[k]+carry))
                carry = 1;
            else carry = 0;
        }
        fclose(Out);

        sha256(Chave);

        carry = 0;
        Out=fopen("encriptado.bin","ab" );
        for(k=0;k<8;k++){                   //SOMA MÓDULO SHA+MSG
            SomaChaveSHAmsg[k] = ChaveSHA[k]+Entrada[k]+carry;
            fwrite (&SomaChaveSHAmsg[k],4,1,Out);
            if ((ChaveSHA[k]+Entrada[k]+carry) < (Entrada[k]+carry))
                carry = 1;
            else carry = 0;
        }
        fclose(Out);


    }

    fclose(Input);
    fclose(Key);
    fclose(Out);


    return 0;
    }
```

Listagem 9.2: Programa principal - decriptação

```
#include <stdio.h>
#include<conio.h>
#include<math.h>
```

```
#define FourBytes unsigned int

FourBytes ChaveSHA[8];

#include "sha256.h"


FourBytes K1[] = {
0xd0efaafb,0x434d3385,0x45f9027f,0x503c9fa8,0xcd0c13ec,0x5f974417,0
    xc4a77e3d,0x645d1973,
0xe0323a0a,0x4906245c,0xc2d3ac62,0x9195e479,0x8ca1890d,0xbfe64268,0
    x41992d0f,0xb054bb16
};

//entrada do tipo => decriptar.exe encriptado.bin
int main(int argc, void *argv[])
{
FILE *Input;
FILE *Key;
FILE *Out;
FourBytes Entrada[8];
FourBytes Chave[16];
FourBytes *bufferChave;
FourBytes *bufferMensagem;
FourBytes SomaChaveK1[16];
FourBytes SomaChaveSHAmsg[8];
int m,i,j,k,carry,lSize,y;
float x;

if(argc<2){
printf("FORMATO: decriptar.exe encriptado.bin");
return 1;
}
if( (Input=fopen((char *) argv[1],"rb" ))==NULL ) {
printf("Erro ao abrir arquivo %s.",argv[1]);
return 2;
}
if( (Out=fopen("decriptado.bin","wb" ))==NULL ) {
printf("Erro ao criar arquivo decriptado.bin.");
return 3;
```

```
    }
45  fclose(Out);

    fseek (Input , 0 , SEEK_END);
    lSize = ftell (Input);
    rewind (Input);
50  bufferMensagem = (unsigned int*) malloc (sizeof(int)*lSize);
    fread (bufferMensagem,1,lSize,Input);




55  x = lSize/32.0;
    y = ceil(x);

    m=0;
    carry = 0;
60
    for (j=0;j<y;j++){
        if (m!=2){
            for (i=0;i<8;i++){
                SomaChaveK1[8*m+i]=bufferMensagem[8*j+i];
65          }
            m++;
        } else {
                for (i=0;i<8;i++){
                    SomaChaveSHAmsg[i]=bufferMensagem[8*j+i];
70              }

                for(k=0;k<16;k++){                  //OBTENÇÃO DA CHAVE
                    Chave[k] = SomaChaveK1[k] - K1[k] + 0xffffffff + 1 - carry;
                    if ((K1[k]+Chave[k]) < (Chave[k]))
75                      carry = 1;
                    else carry = 0;
                }

                sha256(Chave);
80
                carry = 0;
                Out=fopen("decriptado.bin","ab" );
                for(k=0;k<8;k++){                   //SOMA MÓDULO SHA+MSG
```

```
                    Entrada[k] = SomaChaveSHAmsg[k] - ChaveSHA[k] + 0xffffffff
                        +1 - carry;
                    fwrite (&Entrada[k],4,1,Out);
                    if ((SomaChaveSHAmsg[k]) < (ChaveSHA[k]-carry))
                        carry = 1;
                    else carry = 0;
                }
                carry=0;
                fclose(Out);

                m=0;
        }
}
fclose(Input);
fclose(Out);

return 0;
}
```

Listagem 9.3: módulo SHA-256

```
#define A 0x6a09e667
#define B 0xbb67ae85
#define C 0x3c6ef372
#define D 0xa54ff53a
#define E 0x510e527f
#define F 0x9b05688c
#define G 0x1f83d9ab
#define H 0x5be0cd19

#define Ch(x, y, z)     (x & y) ^ (~x & z)
#define Maj(x, y, z)    (x & y) ^ (x & z) ^ (y & z)
#define Rot32(x, s)     (((x) >> s) | ((x) << (32 - s))) //à direita
#define sum0(x) (Rot32(x, 2) ^ Rot32(x, 13) ^ Rot32(x, 22))
#define sum1(x) (Rot32(x, 6) ^ Rot32(x, 11) ^ Rot32(x, 25))
#define sigma0(x)       (Rot32(x, 7) ^ Rot32(x, 18) ^ ((x) >> 3))
#define sigma1(x)       (Rot32(x, 17) ^ Rot32(x, 19) ^ ((x) >> 10))

FourBytes K[] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
        0x923f82a4, 0xab1c5ed5,
```

```
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
        0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
        0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
        0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
        0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
        0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
        0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
        0xbef9a3f7, 0xc67178f2
};

void sha256(FourBytes M[16]) {
    FourBytes a,b,c,d,e,f,g,h;
    int i;
    FourBytes temp1,temp2;
    FourBytes W[64],SHA[8];

    a = A;
    b = B;
    c = C;
    d = D;
    e = E;
    f = F;
    g = G;
    h = H;


    for (i=0;i<16;i++){
        W[i] = M[i];
    }
    for (i=16;i<64;i++){
        W[i] =  sigma1(W[i-2]) ^ W[i-7] ^ sigma0(W[i-15]) ^ W[i-16];
    }

    for (i=0;i<64;i++){
        temp1 = h + sum1(e) + Ch(e,f,g) + W[i] + K[i];
```

```
55            temp2 = sum0(a) + Maj(a,b,c);
              h = g;
              g = f;
              f = e;
              e = d + temp1;
60            d = c;
              c = b;
              b = a;
              a = temp1 + temp2;
          }
65
          ChaveSHA[0] = a + A;
          ChaveSHA[1] = b + B;
          ChaveSHA[2] = c + C;
          ChaveSHA[3] = d + D;
70        ChaveSHA[4] = e + E;
          ChaveSHA[5] = f + F;
          ChaveSHA[6] = g + G;
          ChaveSHA[7] = h + H;

75    }
```

9.2 ALGORITMO DE ATUALIZAÇÃO DAS POOLS SECUNDÁRIA E *URANDOM* DO LRNG - 32 BITS

Listagem 9.4: Extração de bits das pools

```
Algorito add(pool, j, g)
tabela[0] = 0x0
tabela[1] = 0x3b6e20c8
tabela[2] = 0x76dc4190
tabela[3] = 0x4db26158
tabela[4] = 0xedb88320
tabela[5] = 0xd6d6a3e8
tabela[6] = 0x9b64c2b0
tabela[7] = 0xa00ae278

temp = g
temp = temp xor pool[j]
temp = temp xor pool[(j+1) mod 32]
temp = temp xor pool[(j+7) mod 32]
temp = temp xor pool[(j+14) mod 32]
temp = temp xor pool[(j+20) mod 32]
temp = temp xor pool[(j+26) mod 32]
temp = (temp  3) xor tabela[temp & 7]

pool[j] = temp
```

9.3 ALGORITMO DE EXTRAÇÃO DE BITS DAS POOLS SECUNDÁRIA E *URANDOM* DO LRNG

Listagem 9.5: Extração de bits das pools

```
Algoritmo Extracao(pool, nbytes, j):
while nbytes > 0
   tmp := SHA-1(pool[0..15])
   add(pool, j, tmp[0])
   tmp := SHA-1'(pool[16..31])
   add(pool, j-1 mod 32, tmp[2])
   add(pool, j-2 mod 32, tmp[4])
   tmp := SHA-1'(pool[(j-2-15) mod 32 ... (j-2) mod 32])
   tmp := folding(tmp[0..4])
   output(tmp, min(nbytes, 10))
   nbytes := nbytes-min(nbytes, 10)
   j := j-3 mod 32
end while
```

9.4 ALGORITMO DE EXTRAÇÃO DE BITS DAS POOLS SECUNDÁRIA E *URANDOM* DO LRNG

Listagem 9.6: Extração de bits das pools

```
Algoritmo folding(W0, W1, W2, W3 e W4)
output W0 xor W3, W1 xor W4, W2[0-15] xor W2[16-31]
```

9.5 TABELA PARCIAL DA DISTRIBUIÇÃO QUI-QUADRADO

GL	p=99%	97,5%	95%	5%	0,1%
1	0,00016	0,001	0,004	3,841	10,827
2	0,020	0,051	0,103	5,991	13,815
3	0,115	0,216	0,352	7,815	16,266
4	0,297	0,484	0,711	9,488	18,467
5	0,554	0,831	1,145	11,070	20,515
10	2,558	3,247	3,940	18,307	29,588
15	5,229	6,262	7,261	24,996	37,697
20	8,260	9,591	10,851	31,410	45,315
30	14,953	16,791	18,493	43,773	59,703

TAB. 9.1: distribuição Qui-quadrado

Observações:

- GL = graus de liberdade = v;
- Seja y_c os valores do corpo da tabela. A tabela então dá os valores y_c tais que P($Y > y_c$)=p;
- Tabela retirada de (BUSSAB, 2002);
- Para valores de GL maiores que 30, usar aproximação normal, encontrada também em (BUSSAB, 2002).

Listagem 9.7: Código em C do teste de frequências

```
#include <stdio.h>
#include<conio.h>
#include<math.h>

#define FourBytes unsigned int

//entrada do tipo => monobit arquivo
int main(int argc, void *argv[])
{
FILE *Input;
FourBytes Entrada,lSize,potencia,numeroTeste,tamanho;
FourBytes bufferMensagem[625];
int i,j,bit0,bit1;
float x;

if(argc<2){
printf("FORMATO: monobit arquivo");
return 1;
}
if( (Input=fopen((char *) argv[1],"rb" ))==NULL ) {
printf("Erro ao abrir arquivo %s.",argv[1]);
return 2;
}

bit0 = 0;
bit1 = 0;

fseek (Input , 0 , SEEK_SET);
// 20000 bits/32 bits= 625 palavras de 32 bits
lSize = 20000/32;
//lerá 625(lSize) palavras de 4 Bytes
tamanho = fread (bufferMensagem,4,lSize,Input);
for (i=0;i<tamanho;i++){
        Entrada=bufferMensagem[i];
        for (j=31;j>=0;j--){
            potencia = pow(2,j);
            numeroTeste = Entrada & potencia;
```

```
            if (numeroTeste == 0) bit0++; else bit1++;
        }

    }

printf("Nr de bits iguais a zero = %d\n",bit0);
printf("Nr de bits iguais a um = %d\n",bit1);

if ((bit1 > 9654) && (bit1 < 10346)) {
   printf("9654 < bits 1's < 10346\n");
   printf ("APROVADO\n");
   }
   else {
   printf("bits 1's < 9654 ou bits 1's > 10346\n");
   printf("REPROVADO\n");
   }

fclose(Input);

return 0;
}
```

9.7 TESTE POKER

Listagem 9.8: Código em C do teste poker

```
#include <stdio.h>
#include<conio.h>
#include<math.h>

#define FourBytes unsigned int

//entrada do tipo => poker arquivo
int main(int argc, void *argv[])
{
FILE *Input;
FourBytes Entrada,lSize,numeroTeste[32],tamanho,somatorio1,somatorio2;
FourBytes bufferMensagem[625],controle,controle1,controle2,n[16];
int i,j;
float VarTeste;

if(argc<2){
printf("FORMATO: poker arquivo");
return 1;
}
if( (Input=fopen((char *) argv[1],"rb" ))==NULL ) {
printf("Erro ao abrir arquivo %s.",argv[1]);
return 2;
}

n[0] = n[1] = n[2] = n[3] = n[4] = n[5] = n[6] = n[7] = 0;
n[8] = n[9] = n[10] = n[11] = n[12] = n[13] = n[14] = n[15] = 0;
fseek (Input , 0 , SEEK_SET);
// 20000 bits/32 bits= 625 palavras de 32 bits
lSize = 20000/32;
//lerá 625(lSize) palavras de 4 Bytes
tamanho = fread (bufferMensagem,4,lSize,Input);

for (i=0;i<tamanho;i++){
        Entrada=bufferMensagem[i];
        for (j=0;j<8;j++){
            controle1 = pow(2,4*j);
            controle2 = pow(2,4*(j+1));
            controle = controle2 - controle1;
```

```
            numeroTeste[j] = Entrada & controle;
40          numeroTeste[j] = numeroTeste[j]/pow(2,4*j);
            switch(numeroTeste[j]) {
                case 0:
                n[0]++;
                break;
45              case 1:
                n[1]++;
                break;
                case 2:
                n[2]++;
50              break;
                case 3:
                n[3]++;
                break;
                case 4:
55              n[4]++;
                break;
                case 5:
                n[5]++;
                break;
60              case 6:
                n[6]++;
                break;
                case 7:
                n[7]++;
65              break;
                case 8:
                n[8]++;
                break;
                case 9:
70              n[9]++;
                break;
                case 10:
                n[10]++;
                break;
75              case 11:
                n[11]++;
                break;
                case 12:
                n[12]++;
```

```
80                  break;
                    case 13:
                    n[13]++;
                    break;
                    case 14:
85                  n[14]++;
                    break;
                    case 15:
                    n[15]++;
                    break;
                    default:
90                  printf("ERRO!");
                }
            }
    }
95
    printf("Nr de sequencias iguais a 0000 = %d\n",n[0]);
    printf("Nr de sequencias iguais a 0001 = %d\n",n[1]);
    printf("Nr de sequencias iguais a 0010 = %d\n",n[2]);
    printf("Nr de sequencias iguais a 0011 = %d\n",n[3]);
100 printf("Nr de sequencias iguais a 0100 = %d\n",n[4]);
    printf("Nr de sequencias iguais a 0101 = %d\n",n[5]);
    printf("Nr de sequencias iguais a 0110 = %d\n",n[6]);
    printf("Nr de sequencias iguais a 0111 = %d\n",n[7]);
    printf("Nr de sequencias iguais a 1000 = %d\n",n[8]);
105 printf("Nr de sequencias iguais a 1001 = %d\n",n[9]);
    printf("Nr de sequencias iguais a 1010 = %d\n",n[10]);
    printf("Nr de sequencias iguais a 1011 = %d\n",n[11]);
    printf("Nr de sequencias iguais a 1100 = %d\n",n[12]);
    printf("Nr de sequencias iguais a 1101 = %d\n",n[13]);
110 printf("Nr de sequencias iguais a 1110 = %d\n",n[14]);
    printf("Nr de sequencias iguais a 1111 = %d\n",n[15]);

    somatorio1 = 0;
    somatorio2 = 0;
115 for (i=0;i<=15;i++){
        somatorio1 = somatorio1 + n[i];
        somatorio2 = somatorio2 + n[i]*n[i];
    }

120 printf("Total de sequencias de 4 bits = %d \n",somatorio1);
```

```
    VarTeste = (float)16/5000;
    VarTeste = VarTeste*somatorio2 - 5000;
    printf("estatistica = %5.2f\n",VarTeste);
    if ((VarTeste > 1.03) && (VarTeste < 57.4)) {
125     printf("1.03 < Estatistica < 57.4\n");
        printf ("APROVADO\n");
        }
        else {
        printf("Estatistica < 1.03 ou Estatistica > 57.4\n");
130     printf("REPROVADO\n");
        }

    fclose(Input);

135 return 0;
    }
```

Listagem 9.9: Código em C do teste para sequências corridas

```
#include <stdio.h>
#include<conio.h>
#include<math.h>

#define FourBytes unsigned int

//entrada do tipo => seqCorrida arquivo
int main(int argc, void *argv[])
{
FILE *Input;
FourBytes Entrada,lSize,potencia,numeroTeste,tamanho;
FourBytes bufferMensagem[625];
int i,j,cont,antigoBit,n0[6],n1[6],cont33,contMAX,controle,limI[6],limS[6];
float x;

if(argc<2){
printf("FORMATO: monobit arquivo");
return 1;
}
if( (Input=fopen((char *) argv[1],"rb" ))==NULL ) {
printf("Erro ao abrir arquivo %s.",argv[1]);
return 2;
}

//Estes dois valores serve para não começar contando
//para no início não ser nem zero nem um, ou seja, para não contar
cont = 0;
antigoBit = 2;

cont33 = 33;
contMAX = 0;

n0[0] = n1[0] = 0; n0[1] = n1[1] = 0; n0[2] = n1[2] = 0;
n0[3] = n1[3] = 0; n0[4] = n1[4] = 0; n0[5] = n1[5] = 0;
n0[6] = n1[6] = 0;

fseek (Input , 0 , SEEK_SET);
// 20000 bits/32 bits= 625 palavras de 32 bits
```

```
lSize = 20000/32;
//lerá 625(lSize) palavras de 4 Bytes
tamanho = fread (bufferMensagem,4,lSize,Input);
printf("Tamnho do arquivo (em bytes) = %d\n",tamanho);
for (i=0;i<tamanho;i++){
        Entrada=bufferMensagem[i];
        for (j=0;j<32;j++){
            potencia = pow(2,j);
            numeroTeste = Entrada & potencia;

            if (numeroTeste == 0){
               if (antigoBit == 0) cont++;
               else {
                    antigoBit = 0;
                    if (cont > 5) n1[6]++; else n1[cont]++;
                    cont = 1;
               }
            }
            else {
               if (antigoBit == 1) cont++;
               else {
                    antigoBit = 1;
                    if (cont > 5) n0[6]++; else n0[cont]++;
                    cont = 1;
               }
            }
            if (cont > contMAX) contMAX = cont;
            if (cont > cont33) cont33 = cont;

        }
        if ((j == 32) && (i == (tamanho-1))){
                  if (antigoBit == 0) n0[cont]++; else n1[cont]++;
        }
    }

printf("TESTE DE SEQUENCIAS CORRIDAS\n");

limI[0] =2267; limS[0] =2733; limI[1] =1079; limS[1] =1421;
limI[2] =502; limS[2] =748; limI[3] =223; limS[3] =402;
limI[4] =90; limS[4] =223; limI[5] =90; limS[5] =223;
```

```
80  controle =0;

    for (i=1;i<7;i++){
        printf("O numero de sequencias corridas que contem %d bits 0 = %d\n", i
            , n0[i]);
        if ((n0[i] > limI[i-1]) && (n0[i] < limS[i-1])) {
85          printf("%d < %d < %d\n",limI[i-1],n0[i],limS[i-1]);
            printf ("APROVADO\n");
        }
        else {
             printf("%d < %d ou %d > %d\n",n0[i],limS[i-1],n0[i],limI[i-1]);
90           printf("REPROVADO\n");
             controle ++;
        }

        printf("O numero de sequencias corridas que contem %d bits 1 = %d\n", i
            , n1[i]);
95      if ((n1[i] > limI[i-1]) && (n1[i] < limS[i-1])) {
            printf("%d < %d < %d\n",limI[i-1],n1[i],limS[i-1]);
            printf ("APROVADO\n");
        }
        else {
100          printf("%d < %d ou %d > %d\n",n1[i],limS[i-1],n1[i],limI[i-1]);
             printf("REPROVADO\n");
             controle ++;
        }
    }
105

    printf("RESULTADO TESTE DE SEQUENCIAS CORRIDAS:");
    if (controle == 0) printf("APROVADO\n\n"); else printf("REPROVADO\n\n");

    printf("TESTE DE TAMANHO MAXIMO DE SEQUENCIA\n");
110 printf("contMAX = %d\n",contMAX);
    printf("RESULTADO TESTE DE TAMANHO MAXIMO DE SEQUENCIA:");
    if (contMAX < 33) printf("APROVADO\n"); else printf("REPROVADO\n");

    fclose(Input);
115

    return 0;
    }
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo