UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
CURSO DE INFORMÁTICA

Um estudo descritivo de novos algoritmos de criptografia

por

MARCO ANTÔNIO MIELKE HINZ

Monografia apresentada ao Curso de Bacharelado em Informática do Instituto de Física e Matemática da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Bacharel em Informática – Ênfase de Sistemas de Computação

Prof. Raul Fernando Weber, Dr. / UFRGS
Orientador

Prof. Gil Carlos R. Medeiros, MSc. / UFPEL
Co-orientador

Pelotas, dezembro de 2000

AGRADECIMENTOS

Ao meu orientador, Professor Raul Fernando Weber pela confiança depositada.

Ao meu co-orientador, Professor Gil Carlos Medeiros pelos momentos de apoio e cobrança.

A todos os professores que me auxiliaram durante o desenvolvimento deste trabalho.

Especialmente aos meus pais, que incondicionalmente acreditaram na minha capacidade e me apoiaram sempre apostando no meu futuro.

“Nada neste mundo é tão poderoso quanto uma idéia
cujo tempo tenha comprovado sua validade”

Victor Hugo

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIAÇÕES

DES – *Data Encryption Standard*

AES – *Advanced Encryption Standard*

NIST – *National Institute of Standards and Technology*

RESUMO

Este trabalho realiza uma análise que compara cinco dos mais importantes algoritmos de criptografia simétrica existentes no momento, que possivelmente poderão substituir o DES (*Data Encryption Standard*), usado desde a década de 70. Esta comparação é feita analisando os algoritmos em diversos fatores, tais como requisitos de memória, velocidade de processamento, funções criptográficas e complexidade computacional, mostrando suas vantagens e desvantagens.

O trabalho pode ser dividido em duas partes, a primeira uma parte inicial onde o DES e os 5 algoritmos comparados são descritos; na segunda parte é realizada a comparação dos algoritmos de onde são tiradas as conclusões sobre as vantagens e desvantagens do uso dos algoritmos nas mais diversas áreas.

Palavras Chave: criptografia, DES, AES, MARS, RC6, Rijndael, Serpent e Twofish

ABSTRACT

This essay makes a comparative analysis of the five most important criptography algorithms, one of which could replace the DES one, in use since 1977. This comparison is make by analyzing algorithms in several characteristics, such as memory requirements, processing speed, cryptographic functions and computational complexity, showing up their advantages and disadvantages.

The essay can be divided in two parts, the first of them is an introduction where DES and 5 finalist algorithms are compared; in the second one many comparisons are made between the algorithms, where are made conclusions, showing up the advantages and disadvantages of to use these algorithms in several areas.

Keywords: cryptography, DES, AES, MARS, RC6, Rijndael, Serpent e Twofish

1 INTRODUÇÃO

Manter a segurança de dados, importantes ou não, é uma preocupação constante na vida das pessoas. Quem um dia não precisou transmitir qualquer mensagem secreta, seja ela de que maneira for, e teve medo de que o conteúdo dessa mensagem fosse descoberto por um estranho? Esta preocupação tem afligido muitas pessoas a milhares de anos. Com a necessidade de manter dados secretos para quem não se deseje revelar o seu conteúdo, surgiu uma técnica chamada criptografia, que significa escrita desconhecida.

Desde tempos remotos, a criptografia sempre teve um papel importante na transmissão de dados secretos, pois é ela que torna, ou tenta tornar, estes dados irreconhecíveis para qualquer pessoa que não seja o remetente ou o recebedor.

Apesar da criptografia ter mudado na sua capacidade de criptografar textos, aumentando assim a segurança, sua forma de cifrar textos continua a mesma desde a história antiga, quando Júlio César fazia pequenas substituições de letras em suas mensagens de guerra para que inimigos não descobrissem seus planos.

A grande diferença dos algoritmos de hoje para os primeiros usados antigamente é que, com o surgimento dos computadores, substituições maiores se tornaram possíveis, tornando os algoritmos mais seguros.

Hoje, a criptografia continua tendo uma importância militar ainda muito grande, entretanto ela é usada também nas mais diversas áreas onde a transmissão de dados necessita de segurança. A crescente necessidade de métodos criptográficos mais seguros foi alavancada principalmente pela aumento da quantidade de informação armazenada e pela expansão das redes de computadores. Esta ligação íntima que existe entre a criptografia e a segurança tem feito aumentar o investimento de tempo e dinheiro para que novos algoritmos criptográficos mais seguros sejam descobertos.

Um modelo de criptografia que surgiu na década de 70 e se tornou padrão no mundo todo foi o DES (*Data Encryption Standard*). Implementado em 1977, o DES é usado por exemplo em navegadores internet ou *smart cards*, podendo assim ser implementado tanto em hardware como em software. Entretanto, depois de ser submetido a mais de 20 anos de criptoanálise, aliada a um aumento expressivo da capacidade computacional, a segurança do DES começou a se comprometer.

Preocupado com a necessidade de manter um padrão de algoritmo criptográfico seguro, em 1997 o NIST (*National Institute of Standards and Technology*) do governo Norte Americano, iniciou um projeto chamado AES (*Advanced Encryption Standard*) convidando a comunidade mundial especializada no assunto a submeter suas propostas de novos algoritmos

de criptografia, dos quais o vencedor irá substituir o DES como o novo modelo de algoritmo criptográfico. Este concurso realizou-se em três fases: num primeiro momento foram selecionados 15 algoritmos, dentre os quais, numa segunda etapa permaneceram somente 5 algoritmos, que numa etapa final, foram analisados permanecendo o Rijndael como escolhido.

O estudo desses novos algoritmos, no que diz respeito a seu funcionamento e principais características, representa a possibilidade de adquirir conhecimento de técnicas criptográficas que estão no estado da arte e também uma referência para estudos posteriores nas mais diferentes áreas onde estes algoritmos podem ser implementados.

Este trabalho tem o propósito de fazer uma análise computacional dos cinco algoritmos que chegaram a fase final deste concurso: MARS, RC6, Rijndael, Serpent e Twofish. Depois de uma análise destes cinco algoritmos, eles são comparados quanto:

- a complexidade computacional;
- aos requisitos de memória;
- a velocidade de processamento;
- as funções criptográficas;

No capítulo 2, é apresentado um histórico da criptografia e o seu desenvolvimento até os dias de hoje, como também algumas estruturas comumente usadas nos algoritmos criptográficos que ajudarão na descrição dos mesmos.

No capítulo 3, o algoritmo DES é descrito. Seu desenvolvimento e suas estruturas são detalhadas para facilitar a compreensão do porquê da procura de um novo algoritmo modelo de criptografia.

O capítulo 4, trata da descrição dos 5 algoritmos que este trabalho se propõe a comparar. São descritas suas estruturas de dados e princípios de funcionamento.

A seguir, no capítulo 5, é realizada uma comparação dos algoritmos, esta comparação é feita levando-se em conta aspectos que diferenciam os algoritmos quanto a sua eficiência e segurança.

O capítulo 6 traz a conclusão do trabalho.

2 DESENVOLVIMENTO DA CRIPTOGRAFIA

2.1Terminologia

Alguns termos específicos serão usados durante este trabalho. Para auxiliar na leitura do mesmo, apresenta-se aqui uma rápida explicação dos seus significados.

Um **transmissor** é aquele que transforma uma mensagem comum em uma mensagem criptografada e a manda para um **receptor** que a recebe e a transforma novamente na mensagem comum.

Uma **mensagem** é um texto simples. O processo de tornar o conteúdo de uma mensagem irreconhecível é chamado de **encriptação** ou **cifragem.** O processo contrário, ou seja, retornar uma mensagem em texto comum novamente é chamado de **decriptação** ou **decifragem**. A figura abaixo exemplifica melhor esses processos.

A mensagem depois de encriptada pode ser chamada de **criptografada** ou **cifrada**, e depois de decriptada esta é chamada de **decriptografada** ou **decifrada**.

A técnica de manter mensagens seguras é chamada de **criptografia**. A técnica ou a arte de tentar descobrir o conteúdo de mensagens cifradas é chamada de **criptoanálise**, e seus praticantes de **criptoanalistas** ou **atacantes**. E o conjunto destas duas técnicas é chamado de **criptologia**.

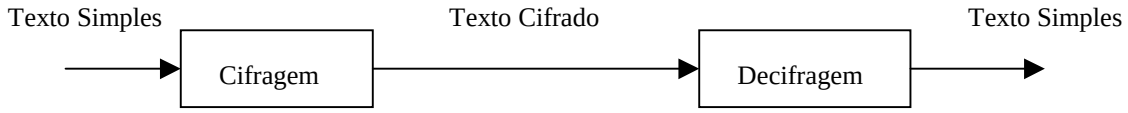

Figura 2.1. Os processos de cifragem e decifragem

2.2 História e fundamentos da criptografia

Para uma melhor compreensão do trabalho, uma breve descrição da evolução da criptografia através dos tempos faz-se necessária, uma vez que as mesmas técnicas empregadas nos primeiros algoritmos criptográficos de antigamente são empregadas hoje nos algoritmos mais modernos.

Mesmo tendo sido usada “informalmente” por muitas pessoas que, de alguma forma ou de outra, queriam manter segredo em relação ao conteúdo de suas mensagens, os militares foram o grupo de pessoas que mais usaram a técnica da criptografia para conseguir segurança

de seus dados transmitidos, principalmente durante as guerras; sendo eles também os maiores responsáveis pelo desenvolvimento da criptografia.

O primeiro relato de um algoritmo de criptografia que se tem é conhecido como algoritmo de César [WEB 95], usado pelo imperador Júlio César na Roma Antiga. Era um algoritmo simples que fazia substituições alfabéticas no texto da mensagem. As substituições aconteciam trocando as letras por outras, três posições a frente no alfabeto, ou seja, a letra *A* seria substituída por *D*, a letra *B* por *E*, *C* por *F*, e assim por adiante.

Texto Simples: Vamos atacar o norte durante a noite.
Texto Cifrado: Zdprv dxdfdu r qruxh gyudqxh d qrlxh.

Esse algoritmo desenvolvido por Júlio César enganou seus inimigos, mas depois de descoberto não enganou mais ninguém. Entretanto, uma pequena melhora neste algoritmo introduz um conceito que usamos até hoje, o conceito de chave. Se ao invés de substituirmos as letras três a três, substituíssemos elas *k* posições, então independentemente do algoritmo ser conhecido, a chave se torna mais importante que o algoritmo, pois sem o conhecimento dela não conseguiríamos decifrar a mensagem. O exemplo a seguir mostra um algoritmo de substituição que substitui as letras 10 posições a frente (*k*=10).

Com *k* = 10:
Alfabeto Simples: abcdefghijklmnopqrstuvwxyz
Alfabeto Cifrado: klmnopqrstuvwxyzabcdefghij

Texto Simples: Vamos atacar o norte durante a noite
Texto Cifrado: Fkwyc kdkmkb y xybdo nebkxdo k xysyo

Os algoritmos mais modernos usam os mesmos princípios deste relatado acima, onde o algoritmo é conhecido mas a chave não. É importante salientar que o tamanho da chave também é um fator importantíssimo na segurança do algoritmo, uma chave de dois dígitos possui 100 possibilidades de combinações, uma chave de 6 dígitos possui 1 milhão de possibilidades. Consequentemente quanto mais possibilidades de combinações, mais tempo um criptoanalista levará para decifrar um texto.

Esses algoritmos que substituem letras por outras letras, mas mantendo-as na mesma posição são chamados de algoritmos de cifras de substituição. Existem outros tipos de algoritmos que trocam a posição das letras sem trocar o seu valor, estes algoritmos são chamados de algoritmos de cifras de transposição.

Um algoritmo de transposição conhecido é o de transposição de colunas [TAN 97]. Neste algoritmo uma palavra ou frase sem letras repetidas é usada como chave. Então o número de letras dessa palavra ou frase é usado como número de colunas e o índice de cada letra como índice de cada coluna. Primeiramente escreve-se o texto preenchendo coluna por coluna formando um tipo de matriz, e finalmente se escreve o texto cifrado lendo a partir das colunas e não das linhas.

C	R	I	P	T	O
1	5	2	4	6	3
t	e	n	h	a	c
u	i	d	a	d	o
c	o	m	o	n	u
m	e	r	o	d	o
c	a	r	t	a	o
d	e	c	r	e	d
i	t	o	a	b	c

Chave : CRIPTO
Texto Simples: tenhacuidadocomonumerodocartaodecredito
TextoCifrado: tucmcdindmrrcocouoodchaootraeioeaetadndaeb

Figura 2.2. Criptografia por Transposição

2.2.1 Algoritmos de Chave Única

Os tipos de algoritmos mostrados anteriormente são algoritmos de chave única, isto significa que uma mesma chave é usada para cifrar a mensagem e para decifrá-la.

Esta técnica de chave única pode criar um cifra indecifrável e é conhecida a muito tempo. Primeiro se escolhe uma chave composta por uma string aleatória, depois converte o texto a ser criptografado em uma string de bits, finalmente calcule o XOR (ou exclusivo) dessas duas strings. O texto cifrado é inviolável, pois qualquer texto simples é candidato e igualmente equiprovável. Entretanto este método traz consigo algumas deficiências, uma delas é que não é possível memorizar a chave (ela é aleatória) obrigando ao transmissor a transporta-la junto com a mensagem. Então uma grande quantidade de texto criptografado capturado pelo criptoanalista pode dar a ele subsídios para decifrar a mensagem observando a repetição de caracteres.

Uma técnica mais apurada na cifragem de mensagens tenta esconder a chave dentro da mensagem de sorte que um criptoanalista não consiga identificá-la dentro da mensagem, e, além disso, não consiga obter qualquer informação do texto que o ajude a decifrá-lo.

Os algoritmos que serão descritos aqui são todos simétricos, ou seja, o processo de encriptação e decriptação é o mesmo. Então iremos descrever somente o processo de encriptação.

2.2.2 *S-Boxes*

Uma estrutura de uso freqüente nos algoritmos criptográficos é chamada de *S-Box*. *S-Boxes* são caixas de substituição não lineares que visam emaranhar o texto cifrado para que se torne mais difícil a sua decifragem. Elas variam de tamanho de entrada e saída e podem ser criadas algoritmicamente ou randomicamente, em outras palavras, o funcionamento de uma *S-Box* se baseia na simples troca de posição dos bits de entrada.

O primeiro algoritmo a implementar *S-Boxes* foi o *Lucifer*, depois o DES; hoje, os mais importantes algoritmos de criptografia usam *S-Boxes*.

2.2.3 *Feistel Networks*

Uma função *Feistel Network* [SCH 98] é um método que transforma qualquer função em uma permutação. Ela foi inventada por Horst Feistel em seu projeto do algoritmo *Lucifer* e se tornou popular no DES, sendo a base da maioria dos algoritmos criptográficos publicados desde então.

A construção de um bloco fundamental de um *Feistel Network* é uma função *F* que faz um mapeamento, dependente de uma chave, de uma string de entrada dentro de uma string de saída. Uma função *F* é sempre uma função não linear.

$$F: \{0,1\}^{n/2} \times \{0,1\}^{N} \rightarrow \{0,1\}^{n/2}$$

Onde n é o tamanho do bloco do *Feistel Network* e *F* é a função utilizando metade (*n*/2) dos bits de um bloco e *N* bits de uma chave como entrada, produzindo uma saída de tamanho de *n*/2 bits.

A cada volta, o bloco fonte de dados é a entrada da função *F*. E na saída desta função é aplicado um XOR com o bloco destino de dados, isso depois destes dois blocos trocarem de lugar para a próxima volta. O objetivo principal deste tipo de implementação está na quantidade de execuções, pois ela irá proporcionar uma criptografia fraca se for executada uma só vez, mas a força da cifra crescerá a medida que mais voltas forem sendo executadas.

Duas voltas de um *Feistel Network* são chamadas de um “ciclo”. Em um ciclo todos os bits do bloco fonte de dados são modificados pelo menos uma vez.

Outra característica importante de um *Feistel Network* é que uma palavra de dados é usada para modificar as outras.

2.3 ALGORITMOS PRÉ-SELECIONADOS

Os quinze algoritmos selecionados na fase inicial foram [WWW 1]: CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, MAGENTA, MARS, RC6, Rijndael, SAFER+, Serpent e Twofish. Alguns destes algoritmos eram desde seu projeto mais fracos que o DES, outros tiveram problemas na sua implementação.

Concluída a segunda etapa de avaliação, os algoritmos escolhidos foram: RC6, MARS, Rijndael, Twofish e Serpent.

Será descrito agora o algoritmo DES para que possa ser utilizado como parâmetro ao avaliarmos as mudanças e inovações que os cinco algoritmos criptográficos escolhidos estão trazendo para o mercado.

3 DES

Apesar da chamada criptografia moderna ter avançado muito na segunda guerra mundial, até a metade da década de 70 não se tinha um padrão que permitisse que diferentes equipamentos compartilhassem dados criptografados. A criação desse padrão tornaria a proteção desses dados muito mais barata e confiável. Pensando nisso, o NIST (que naquela época se chamava NBS - *National Bureau of Standards*), em 1973, fez uma chamada à comunidade para uma proposta de um algoritmo padrão.

Este algoritmo deveria obedecer algumas especificações, tais como [SCH 97]:

- deve ter alto nível de segurança;
- deve estar totalmente documentado e ser de fácil entendimento;
- a segurança do algoritmo deve se encontrar na chave e não depender do sigilo do algoritmo;
- deve estar disponível para todos os usuários;
- deve ser adaptável para uso em diversas aplicações;
- deve ser economicamente implementável em dispositivos eletrônicos;
- deve ser eficiente;
- deve estar habilitado para validação;
- deve ser exportável.

Embora tenha ocorrido um grande interesse da comunidade, nenhum algoritmo sequer se aproximava das especificações do NIST. Então no ano seguinte, uma nova requisição foi proposta e assim foi escolhido um algoritmo submetido pela IBM que era baseado no *Lucifer*, um algoritmo desenvolvido no início da década de 70 pela própria IBM. Este novo algoritmo, apesar de ser complicado, foi bem aceito uma vez que ele era de fácil implementação em hardware, sua principal característica era usar operações lógicas simples num grupo reduzido de bits.

3.1 Descrição

O DES é um algoritmo simétrico (o processo de encriptação é o mesmo de decriptação) de 64 bits. Para cada 64 bits de texto simples na entrada do algoritmo, surgem 64 bits de texto criptografado na saída. Apesar da chave do DES ser representada como um número de 64 bits, o seu tamanho é 56 bits pois todos os oitavos bits são usados somente como bits de paridade.

A transformação de texto simples em texto cifrado no DES dá-se em 18 estágios. No primeiro estágio é feita uma transposição da chave no texto de 64 bits. No último estágio ocorre uma transposição inversa do primeiro estágio. Estes dois estágios não afetam na segurança do DES, podendo ser retiradas do DES a fim de facilitar a implementação do DES em software. Contudo, essa modificação no algoritmo o coloca fora do padrão do DES. As 16 etapas intermediárias são transformações idênticas.

3.1.1 A permutação inicial

A permutação inicial não influi diretamente na segurança do DES pois ela apenas executa uma simples permutação de bits com o propósito de facilitar o uso do DES em chips quando ele é implementado em hardware.

Conforme podemos acompanhar na Tabela 3.1 [SCH 97], a função da permutação inicial é deslocar os bits para novas posições, como por exemplo o bit 58 para a posição 1, o bit 50 para a posição 2, o bit 42 para a posição 3 e assim por diante.

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	06	64	56	48	40	32	25	16	8
57	49	41	66	25	17	09	01	59	51	43	35	27	19	11	09
61	53	45	37	29	21	13	05	63	55	47	39	31	23	15	07

Tabela 3.1. A Permutação inicial

3.1.2 As 16 etapas intermediárias

A cada volta do DES, é executada a função *F*. Na entrada da função, são recebidas duas metades do bloco de 64 bits (metade esquerda e metade direita) divididas ao meio e uma sub-chave.

As duas metades do texto fonte são criadas antes da primeira volta, e ao final da última volta essas metades são reunidas novamente numa permutação final que é o inverso da permutação inicial.

A função *F* realiza a cada volta as seguintes operações:

- A chave é transformada para 48 bits (mais detalhes na seção 3.1.4 sobre o tratamento da chave);

- A metade direita dos dados é expandida de 32 para 48 bits através de uma permutação, a *expansion permutation*;
- Sobre esta metade direita é aplicado um XOR com a chave;
- Os dados da metade direita são mandados para oito *S-Boxes* produzindo 32 bits ao todo;
- Finalmente estes dados são permutados novamente numa *P-Box*.

Na saída da função *F* é aplicado um XOR com a metade esquerda dos dados. A metade direita dos dados se tornará a metade esquerda na próxima volta e a metade esquerda se tornará a metade direita.

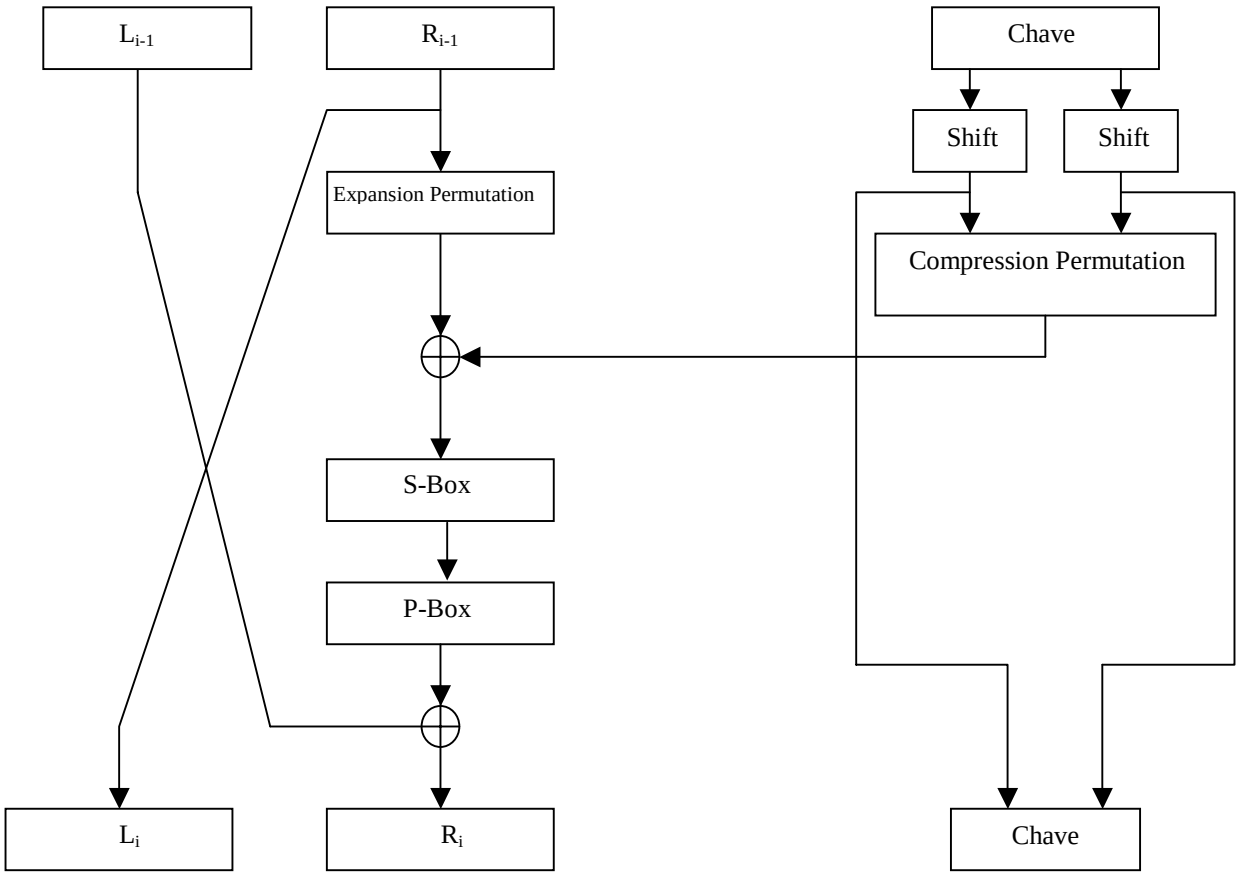

Figura 3.1. Cada ciclo do DES

3.1.2.1 *Expansion Permutation*

Além de deixar a metade direita dos dados do mesmo tamanho da chave a *Expansion Permutation* tem o objetivo de aumentar a dependência dos dados de forma que todos os bits dos dados de destino sejam dependentes de cada bit do texto fonte, ocasionando o que chamamos de efeito avalanche.

Para cada bloco de 4 bits de entrada os primeiros e quartos bits representam dois bits no bloco de saída, e os segundos e terceiros bits da entrada representam somente um bit.

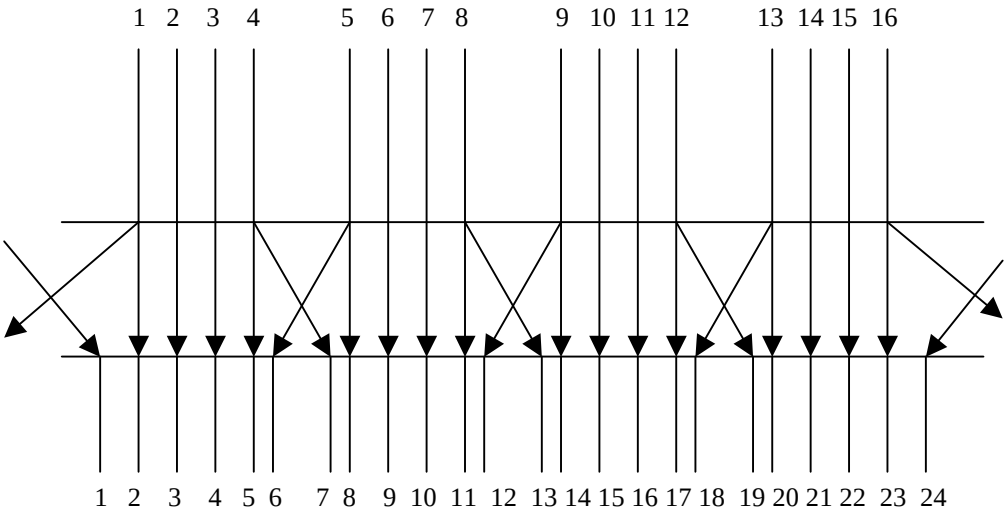

Figura 3.2. *Expansion Permutation*

3.1.2.2 *S-Boxes*

O DES trabalha com 8 S-Boxes diferentes que recebem cada uma 6 bits de entrada e devolvem na saída 4 bits.

Cada *S-Box* é uma matriz de 4 linhas por 16 colunas. O resultado final desta operação são oito blocos de 4 bits que são agrupados em um único bloco de 32 bits.

3.1.2.3 *P-Boxes*

Dentro das *P-Boxes* os bits são somente permutados para uma outra posição, com a característica de que nenhum bit é permutado mais de uma vez e todos os bits são permutados pelo menos uma vez. A tabela 3.2 [SCH 97] mostra a *P-Box* usada pelo DES.

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Tabela 3.2. P-Box

3.1.3 A permutação final

A permutação final é exatamente o inverso da permutação inicial, os dois blocos da saída da última volta do DES são concatenados de forma que possam ser usados na entrada da permutação.

40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

Tabela 3.3. A Permutação Final [SCH 97]

3.1.4 Tratamento da chave

A chave de 64 bits do DES é primeiramente reduzida para 56 bits, isso é feito ignorando-se todos os oitavos bits, dessa chave de 56 bits, a cada volta é gerada uma chave de 48 bits diferente que é usada no algoritmo.

Essa chave é gerada da seguinte maneira:

- a chave de 56 bits é dividida ao meio;
- cada metade é circularmente rotacionada para a esquerda em um ou dois bits, dependendo da volta;
- então 48 dos 56 bits são selecionados de uma tabela de permutação, essa operação é chamada de *compression permutation*.

4 ANÁLISE DESCRITIVA DOS NOVOS ALGORITMOS

4.1 MARS

MARS é um algoritmo de criptografia simétrico que foi apresentado ao AES pela IBM Corporation. Com um bloco de 128 bits e uma chave de tamanho variável, indo de 128 bits até 400 bits.

Este algoritmo trabalha com uma palavra de 32 bits, usando então, 4 palavras por bloco.

4.1.1 Estrutura de Funcionamento

A implementação do MARS se dá basicamente em três fases. A primeira fase implementa uma rápida mistura dos dados do texto fonte, a inserção da chave e 8 voltas da transformação *Type-3 Feistel*. A segunda fase é a fase mais importante do processo, ela consiste de 16 voltas da transformação *Type-3 Feistel*, destas 16 voltas, 8 foram implementadas em *forward mode* e 8 em *backward mode*. A terceira e última fase é, essencialmente, o inverso da primeira fase.

4.1.2 Fase Um

A primeira operação que é executada nesta fase é a inserção da chave em cada bloco de dados. Seguem-se então 8 voltas da transformação *Type-3 Feistel* sem chave. Cada volta usa um bloco de dados para modificar os outros três, esta operação será melhor descrita na Figura 4.1.

Usaremos os 4 *bytes* do texto fonte como índice das 2 *S-Boxes*, denotadas por *S0* e *S1*, depois executamos operações XOR ou ADD da correspondente entrada dentro das outras três. Se, por exemplo, nós tivéssemos os quatro bytes das palavras de texto fonte como *b0*, *b1*, *b2*, *b3*, onde *b0* é o byte menos significativo e *b3* o mais significativo, usaríamos *b0* e *b2* como índices da *S-Box S0* e consequentemente *b1* e *b3* como índices da *S-Box S1*. Primeiro, aplicamos um XOR em *S0[b0]* dentro da primeira palavra destino; depois aplicamos um ADD *S1[b1]* à mesma palavra. A seguir, aplicamos um ADD *S0[b2]* na segunda palavra destino e um XOR *S1[b3]* na terceira palavra destino. Ao final desta volta, rotacionamos as palavras do texto fonte em 24 bits à direita.

Estas operações são executadas em cada volta, sendo executadas 8 vezes. A cada volta, as palavras mudam de posição: a atual primeira palavra destino torna-se a segunda palavra fonte; a atual segunda palavra destino se torna a terceira palavra fonte, a atual terceira palavra destino torna-se a quarta palavra fonte e, finalmente, a quarta palavra destino torna-se a primeira palavra fonte.

Além disso, após cada quatro voltas específicas são adicionadas (ADD) de uma das palavras destino de volta dentro do texto fonte. Mais especificamente, depois da primeira e da quinta voltas adicionamos a terceira palavra destino de volta à palavra fonte e, após a segunda e a sexta voltas, adicionamos a primeira palavra destino de volta à palavra fonte. Estas operações extras são executadas para eliminar alguma fraqueza contra ataques diferenciais e para quebrar a simetria na próxima fase e tornar mais rápido o efeito avalanche.

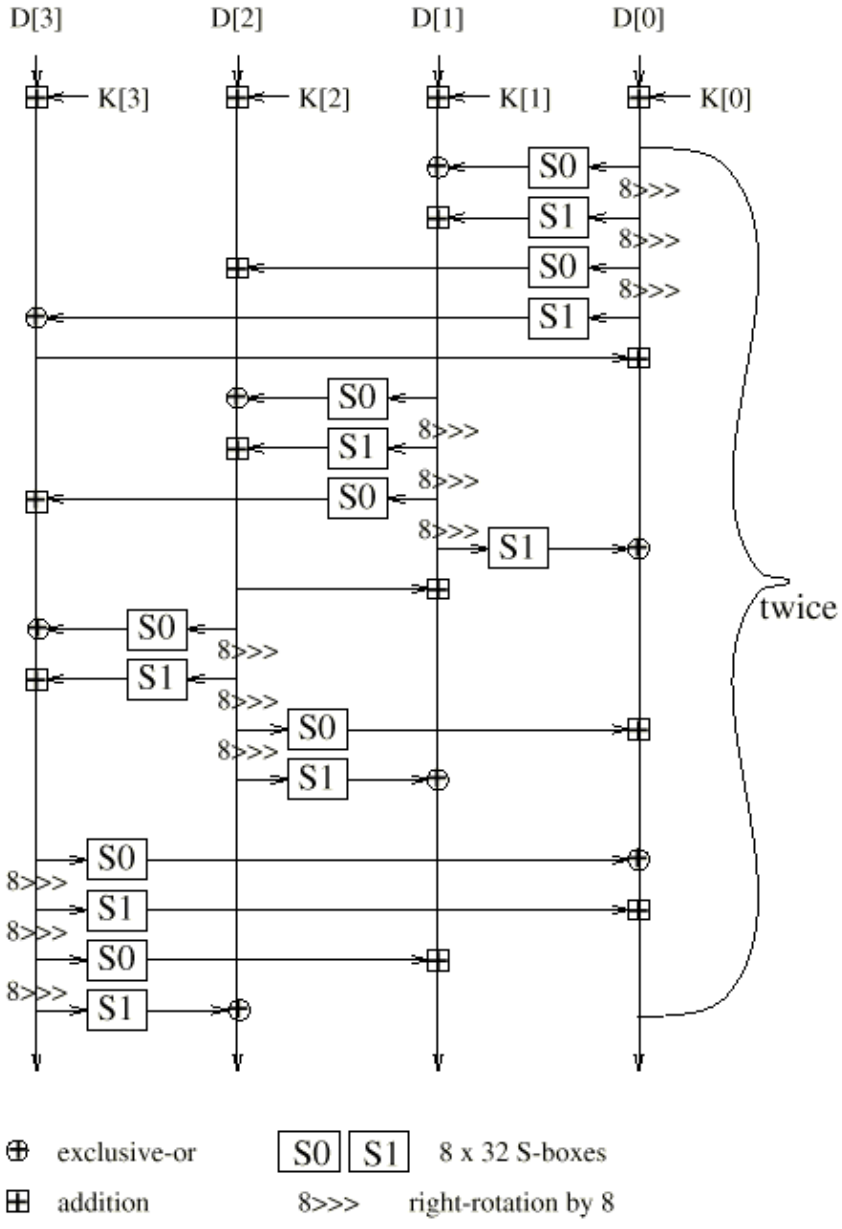

Figura 4.1 – Estrutura da Fase 1 [IBM 99]

4.1.3 Fase Dois

Esta fase é baseada em um laço de 16 voltas usando função *Type-3 Feistel*. A cada volta é aplicada uma função *E* que implementa uma combinação de multiplicações, rotações dependentes de dados e uma *S-Box*, para cada palavra de dados de entrada na função *E* ela retorna três palavras de dados, estas três palavras de dados são misturadas as outras três palavras de dados usando-se as funções XOR ou ADD; além disso, as palavras de dados de texto fonte são rotacionadas em 13 posições para a esquerda.

Uma outra forma de aumentar a resistência do algoritmo contra criptoanálise é usar as três saídas da função *E* em diferente ordem nas oito primeiras voltas do que nas oito últimas voltas. Mais especificamente, nas primeiras oito voltas são adicionadas (ADD) respectivamente a primeira e a segunda saídas da função *E* nas primeiras e segunda palavras destino de dados e aplicado um XOR na terceira saída da função *E* com a terceira palavra destino de dados. E nas oito voltas finais adicionamos (ADD) a primeira e a segunda saídas da função *E* a terceira e segunda palavra destino de dados respectivamente e aplicamos um XOR na terceira saída da função *E* com a primeira palavra alvo de dados. Para um maior entendimento do funcionamento da função *E* descreveremos ela a seguir. A figura 4.2 [IBM 99] mostra o funcionamento desta fase.

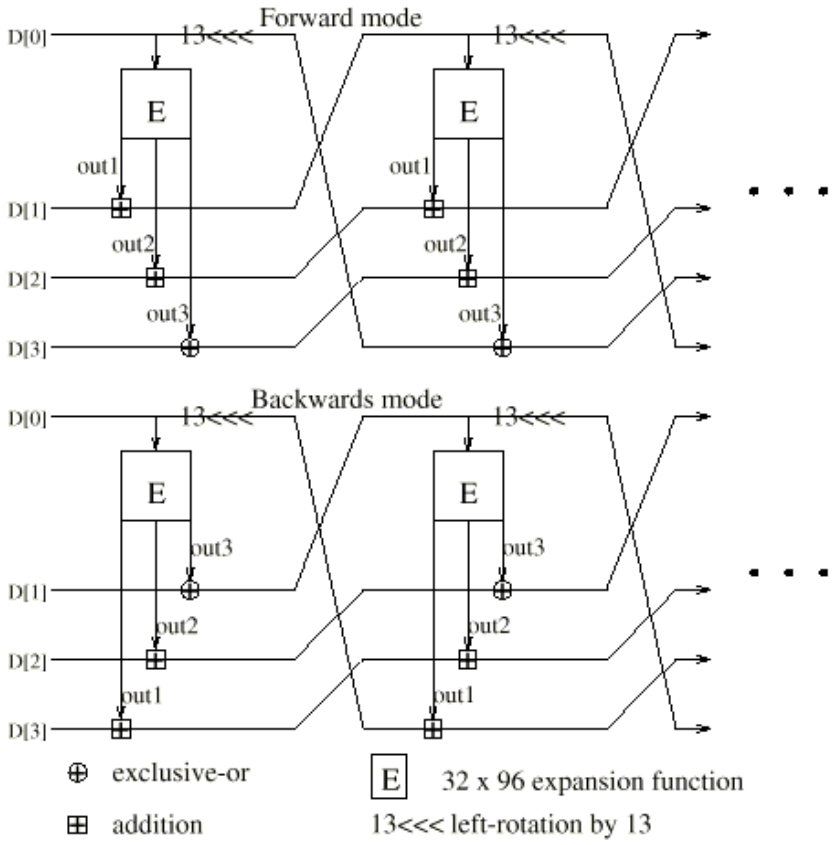

Figura 4.2 – As 8 voltas em *forward mode* e as 8 voltas em *backward mode* da Fase 2

4.1.3.1 A Função *E*

As três saídas da função *E* são montadas a partir de uma palavra de dados de entrada e mais duas chaves. Estas três saídas são tratadas dentro da função *E* como três variáveis temporárias: *L*, *M* e *R*. A função *E* usa uma *S-Box* que é obtida da concatenação das *S-Boxes S0* e *S1* da fase anterior, aqui usaremos *S* para identificar a *S-Box* de 512 entradas.

Primeiro, a variável *R* armazena o valor da entrada de dados rotacionado 13 posições para a esquerda, a variável *M* armazena o valor da entrada de dados adicionado (ADD) ao valor da primeira chave, a variável *L* armazena o valor de *S* que é conseguido usando os nove bits menos significativos da variável *M* como índices da *S-Box*.

Multiplicamos a variável *R* pela segunda palavra chave (que contém um grande número inteiro) e ainda rotacionamos *R* cinco posições para a esquerda de forma que os cinco bits mais significativos de *R* após a multiplicação se transformem nos cinco bits menos significativos. Depois é aplicado um XOR com *R* dentro de *L*, os cinco bits menos significativos de *R* são usados como parâmetro para definir quantas posições os bits da

variável *M* serão rotacionados para a esquerda, podendo variar de 0 à 31 posições. Mais uma rotação de 5 posições para a esquerda é feita em *M*, este resultado é usado para aplicarmos um XOR dentro de *L*, agora os cinco bits menos significativos de *R* são usados como parâmetro de quantas posições a variável *L* irá rotacionar para a esquerda.

A figura 4.3 [IBM 99] a seguir mostra o funcionamento da função *E*.

A variável *L* será a primeira saída da função, *M* a segunda, e *R* a terceira saída da função *E*.

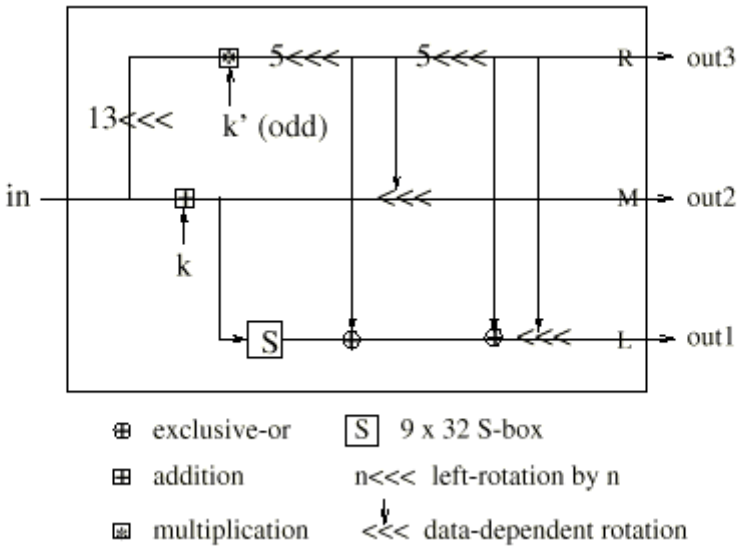

Figura 4.3 – A Função E

4.1.4 Terceira Fase

A terceira e última fase deste algoritmo é praticamente idêntica a primeira com a diferença de que os dados são processados na ordem inversa. A figura 4.4 [IBM 99] mostra em detalhes as operações realizadas nesta fase e descritas abaixo.

As operações são também as mesmas, aplicamos um XOR *S1[b0]* na primeira palavra destino, subtraímos *S0[b3]* da segunda palavra de dados, subtraímos *S1[b2]* da terceira palavra alvo e aplicamos um XOR *S0[b1]* dentro da terceira palavra destino; e por último, rotacionamos a palavra fonte em 24 posições para a esquerda.

A cada volta, rotacionamos as 4 palavras de dados, de modo que a atual primeira palavra destino de dados se tornará a quarta palavra fonte de dados, a atual segunda palavra

destino de dados se tornará a primeira palavra fonte de dados, a terceira palavra destino de dados se tornará a segunda palavra fonte de dados e a quarta palavra destino de dados se tornará a terceira palavra fonte de dados.

Antes de cada 4 específicas voltas, subtraímos uma das palavras destino de dados da palavra fonte, antes da quarta e da oitava volta subtraímos a primeira palavra destino de dados da palavra fonte e, antes da terceira e sétima volta, subtraímos a terceira palavra destino de dados da palavra fonte.

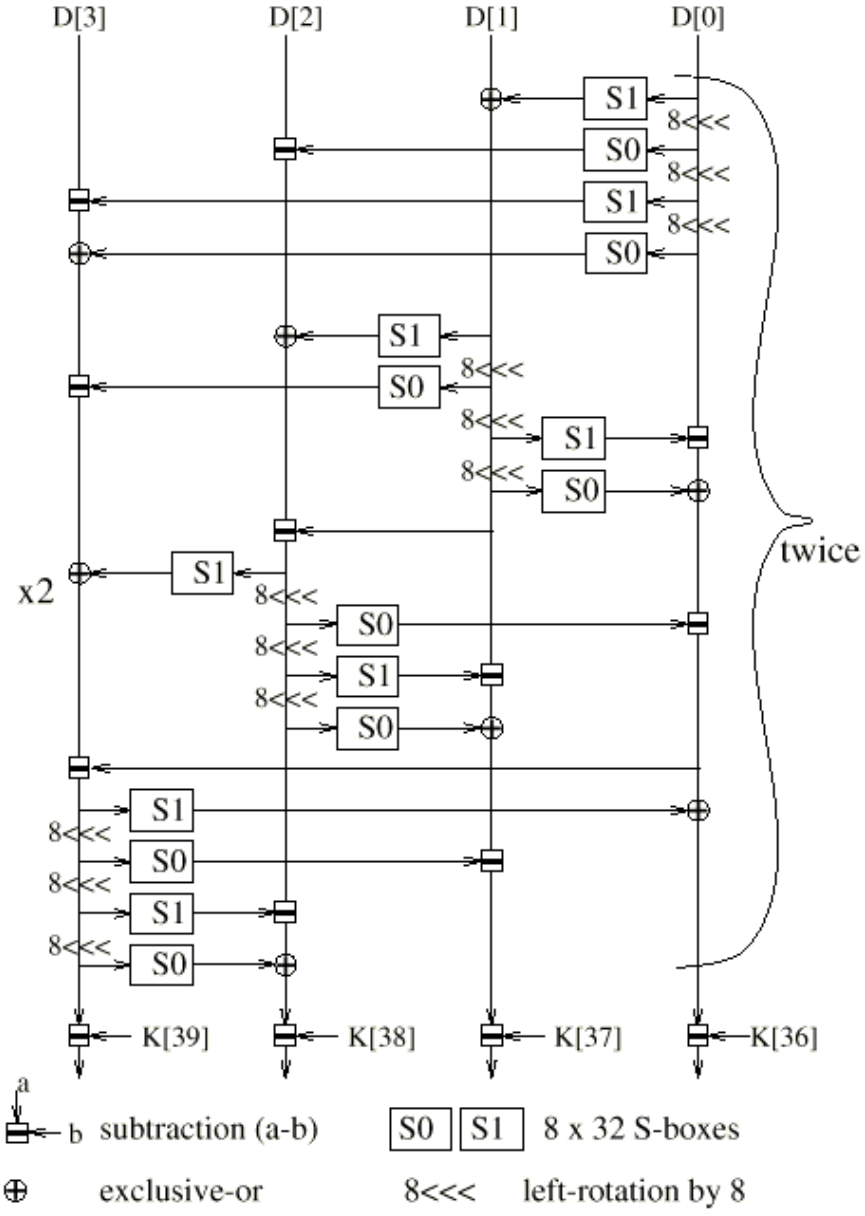

Figura 4.4 – Estrutura da Fase Três

4.1.5 *S-Box*

MARS usa uma *S-Box*, com 512 entradas. Porém, a fim de facilitar o entendimento do algoritmo trataremos ela como duas *S-Boxes* de 256 entradas cada uma. A *S-Box* de MARS é uma tabela de permutação de 8 bits.

4.1.6 Expansão da Chave

Este é um procedimento importantíssimo aplicado pelo MARS que não poderia deixar de ser demonstrado. A sua finalidade é transformar "qualquer" chave de entrada numa chave de tamanho padrão, acabando com o problema de chaves fracas que poderiam colocar em perigo a segurança do algoritmo.

A chave de entrada do algoritmo é um array *k[ ]* de *n* palavras de 32 bits, a única restrição existente é que *n* deve variar de 4 a 14 ($4<n<14$). O procedimento de expansão de chave transforma esse array *k[ ]* em um array *K[ ]* de 40 palavras.

O procedimento tem três passos, que estão descritos abaixo.

Primeiro, copiamos a chave para uma tabela(array) *T[ ]* de 15 posições, as primeiras n posições da tabela são ocupadas pelas *n* palavras da chave, as demais posições são ocupadas com 0.

A segunda etapa é repetida 4 vezes, onde cada iteração computa as próximas 10 palavras da chave expandida. Nesta etapa primeiro transformamos a tabela *T[ ]* usando a seguinte rotina, aqui especificada em pseudo-C:

```
for (i=0, 14 , i++ )  {
  if (i mod 2 = 0)
    T[i] = ((T[i – 7 mod 15] ⊕ T[i – 2 mod 15]) <<< 3) ⊕ (4i + 0)
  else
    T[i]=((T[i – 7 mod 15] ⊕ T[i – 2 mod 15]) <<< 3) ⊕ (4i + 1);
   }
```

Depois, esta operação é usada 4 vezes:

```
for (i=0, 14 , i++ )  {
   T[i] = (T[i] + S[low 9 bits of T[i – 1 mod 15]]) <<< 9;
  }
```

A última operação desta etapa é utilizar 10 das palavras da tabela *T[ ]* e reordená-las dentro do vetor da chave expandida *K[ ]*, isto é feito da seguinte maneira:

```
for (i=0, 9, i++)  {
  if (i mod 2 = 0)
    K[0 + 1] = T[4i mod 15]
 else
   K[10 + 1] = T[4i mod 15];
 }
```

Concluindo a expansão da chave, a última etapa é aplicada para preparar as 16 palavras chave para as multiplicações que são usadas no algoritmo, estas palavras são escolhidas como *k[5], k[7],... ,k[35]* e para elas serem fortes não podem ter dez ou mais 1's ou 0's consecutivos, uma escolha randômica destas palavras chave não asseguraria que elas seguissem este princípio.

4.2 RC6

Apresentado ao AES por pesquisadores do MIT e dos Laboratórios RSA e proveniente do RC5, o RC6 é talvez o mais simples dos algoritmos apresentados, além de sua simplicidade o RC6, ao contrário da maioria dos algoritmos criptográficos existentes, se destaca por não usar *S-Boxes*.

RC6 é parametrizado por 3 parâmetros: *w* que é o tamanho do bloco, *r* que denota o número de voltas e *b* que é o tamanho da chave em bytes. Para ir ao encontro das definições do NIST, o tamanho do bloco é de 32 bits, o número de voltas é igual a 20 e o tamanho da chave pode variar de 0 até 255 bits.

Da chave de entrada são derivadas *2r +4* chaves de tamanho *w* que serão usados durante o processo de cifragem ou decifragem.

O RC6 trabalha com 4 palavras de *w* bits cada uma que serão chamadas de registradores, estes 4 registradores (*A, B, C, D*) são usados tanto para receber o texto de entrada quanto para devolver o texto de saída.

4.2.1 Descrição do Algoritmo

A cada volta do RC6 são executadas as seguintes operações:

Primeiro são adicionados (ADD) aos registradores *B* e *D* uma palavra chave. Então em duas variáveis temporárias, *t* e *u*, são armazenados os resultados de uma função aqui chamada de *f* rotacionado para a esquerda *lg w* vezes, o valor da função *f* armazenado em *t* será usado em operações no registrador *A* e o valor da função *f* armazenado em *u* será usado em operações no registrador *B*. Paralelamente, são aplicadas aos registradores *A* e *C* operações idênticas, ao registrador *A* é aplicado um XOR com a variável *t*, seguido de uma rotação para a esquerda de *u* bits; a esse resultado é adicionado (ADD) uma palavra chave, e ao registrador *C* é aplicado um XOR com a variável *u* seguido de uma rotação para a esquerda de *t* bits, e a esse resultado também é adicionado (ADD) uma palavra chave. Segue-se, então, uma troca de valores: o valor do registrador *A* passa para o registrador *D*, o valor do registrador *B* passa para o *A*, o valor do registrador *C* passa para o *B* e o valor do registrador *D* passa para o *C*. Após essa troca, são adicionadas duas palavras chave, uma no registrador A e outra ao registrador C.

A figura seguinte irá demostrar com maior clareza as operações descritas acima.

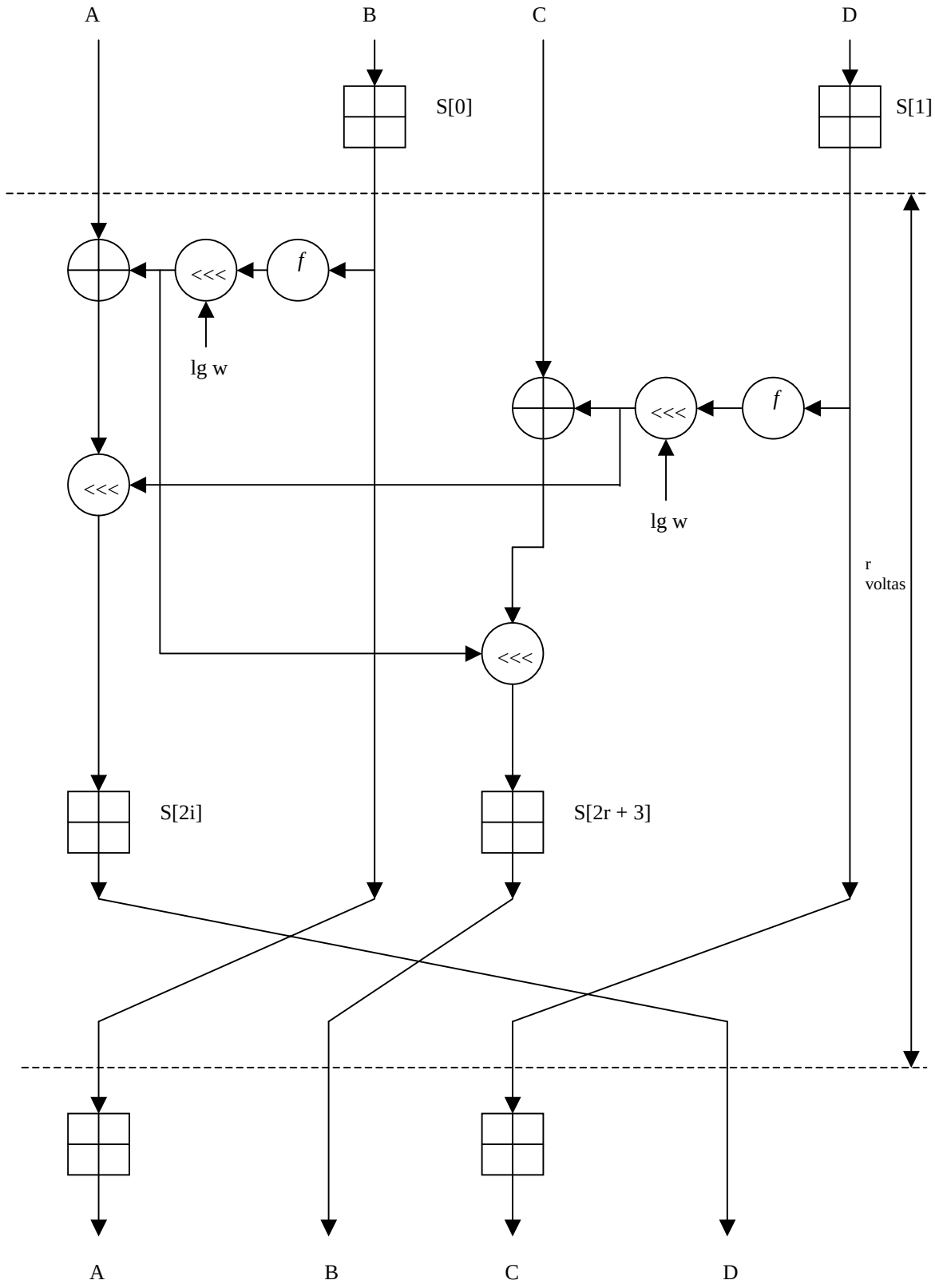

Figura 4.5. Processo de cifragem do RC6 [RIV 98]

4.2.1.1 A função *f*

A função *f* é aplicada em operações envolvidas nos registradores *B* e *D*, para o registrador *B* é aplicada a seguinte operação:

$F(B) = (B \times (2B + 1))$

e para o registrador *D*:

$F(D) = (D \times (2D + 1))$

4.2.2 O Tratamento da Chave

A partir de uma chave dada pelo usuário, o algoritmo de tratamento da chave, se necessário, completa a chave com zeros na saída as chaves que serão usadas pelo RC6 estão inseridas num vetor de *2r+4* posições de *w bits* de tamanho cada uma.

O seguinte algoritmo mostra como é feito o tratamento da chave no RC6:

As duas constantes existentes, *Pw* e *Qw*, na verdade são dois “números mágicos”, que valem:
Pw=B7E15163
Qw=9E3779B9

A entrada do algoritmo é um vetor *L[c];*

A saída do algoritmo é um vetor *S[2r+4];*

```
procedure escalonamento_da_chave   {
        S[0] = Pw;
        for i = = 1 to 2r + 3 do
          S[i] = S[i -1] + Qw;
        A = B = i = j = 0;
        V=3 x max{c, 2r +4};
        for s = = 1 to v do  {
           A = S[i] = (S[i] + A + B)<<<3;
           B = L[j] = (L[j] + A + B) <<< (A + B);
           i = (i + 1) mod (2r + 4);
           j = (j + 1) mod c;
         }
```

4.3 Rijndael

Apresentado ao AES por dois pesquisadores belgas, o Rijndael é um algoritmo de blocos iterativos com tamanho de bloco e de chave variáveis, podendo ser especificados independentemente para 128, 192 ou 256 bits.

O algoritmo Rijndael diferencia-se da maioria dos outros algoritmos que são usados atualmente pois não usa uma estrutura do tipo *Feistel* na sua fase de rotação. Numa estrutura *Feistel,*os bits de um estado intermediário são transpostos em uma outra posição sem serem alterados; no Rijndael, a fase de rotação é composta de transformações uniformes inversíveis distintas chamadas de *layers*.

O resultado das diferentes operações realizadas no estágio intermediário é chamado de Estado. Um estado pode ser denotado por uma matriz retangular de bytes; esta matriz possui 4 linhas e o número de colunas (*Nb*) é igual ao tamanho do bloco dividido por 32.

A cifra da chave também pode ser denotada por uma matriz retangular de bits, também com 4 linhas e o número de colunas (*Nk*) sendo igual ao tamanho da chave divido por 32.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Figura 4.6. Matriz de Estado e Matriz de chave para *Nb* = 6 e *Nk* = 4

Em alguns casos precisamos demonstrar estas matrizes como vetores de 4 bytes, cujo comprimento se refere ao tamanho da linha da matriz retangular. Como a quantidade de colunas da matriz é obtida dividindo-se o tamanho do bloco (128, 192 ou 256) por 32, as colunas da matriz somente podem assumir o tamanho de 4, 6 ou 8.

Quando necessitamos especificar cada byte dentro do vetor usamos as letras *a, b, c* e *d* como índice.

Tanto a entrada como a saída do Rijndael são tratadas como um vetor de 8-bits numerados de *4*Nb – 1* (estes blocos podem ser de 16, 24 ou 32 bytes).

A cifra da chave também é tratada da mesma forma com a diferença de que o vetor é numerado de *4*Nk-1*.

O número de voltas é denotado por *Nr* e este número depende tanto de *Nb* como de *Nk*. A Tabela 4.1 [DAE 99] mostra o número de voltas para *Nb*=*Nk*=4, 6, 8.

Nr	Nb=4	Nb=6	Nb=8
Nk=4	10	12	14
Nk=6	12	12	14
Nk=8	14	14	14

Tabela 4.1. Número de voltas em função do tamanho do bloco e da chave

O algoritmo transforma os dados através do número de voltas usando quatro “funções” diferentes, são elas: *ByteSub*, *ShiftRow*, *MixColumn* e *AddRoundKey*. A última volta é um pouco diferente, ela apresenta apenas as funções: *ByteSub*, *ShiftRow* e *AddRoundKey*. Agora iremos descrever as quatro transformações que ocorrem a cada volta do algoritmo.

4.3.1 Funções Criptográficas usadas pelo Rijndael

4.3.1.1 *ByteSub*

A transformação *ByteSub* é uma substituição não linear que opera em cada Estado independentemente, a *S-Box* usada nesta transformação é inversível e é construída pela composição de duas transformações:

Executando uma multiplicação inversa em GF(2^8).

Aplicando uma transformação definida por:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

4.3.1.2 *ShiftRow*

Nesta transformação as linhas dos Estados são alteradas ciclicamente, a linha 1 não é alterada, a linha 2 é alterada em *C1* bytes, a linha 3 é alterada em *C2* bytes e a linha 4 é alterada em *C3* bytes. Estes parâmetros *C1*, *C2* e *C3* dependem de *Nb*. Estes valores estão especificados na Tabela 4.2 [DAE 99].

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Tabela 4.2. Parâmetros para a quantidade de bytes deslocados no *ShiftRow*

A figura 4.7 mostra o efeito da transformação ShiftRow em um Estado.

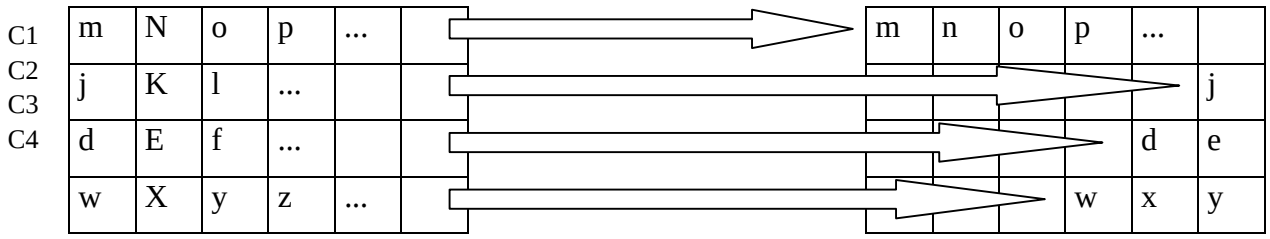

Figura 4.7.Transformação *ShiftRow* em um Estado.

4.3.1.3 *MixColumn*

As colunas de um Estado são consideradas polinômios do tipo $GF(2^8)$ denotadas por $a(x)$ e são multiplicadas em módulo $x^4 + 1$ por um polinômio fixo

$$c(x) = \text{‘03’}x^3 + \text{‘01’}x^2 + \text{‘01’}x + \text{’02’}$$

O resultado da multiplicação pode ser denotado por *b(x)* e a multiplicação por

$$b(x) = c(x) \otimes a(x).$$

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 01\ 02 \\ 03\ 01\ 01\ 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

4.3.1.4 *AddRoundKey*

Esta transformação consiste somente em aplicar um operação XOR num Estado usando uma matriz proveniente do algoritmo da chave. O tamanho da matriz da chave é igual ao tamanho da matriz de Estado.

A Figura 4.8 [DAE 99] mostra a transformação *AddRoundKey*.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$
$a_{2,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$
$a_{3,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$	$a_{0,0}$

$\oplus$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$	$k_{0,4}$	$k_{0,5}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$	$k_{1,4}$	$k_{1,5}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$	$k_{2,4}$	$k_{2,5}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$	$k_{3,4}$	$k_{3,5}$

=

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$	$b_{0,4}$	$b_{0,5}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$	$b_{1,4}$	$b_{1,5}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$	$b_{2,4}$	$b_{2,5}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$	$b_{3,4}$	$b_{3,5}$

Figura 4.8. Transformação *AddRoundKey*.

4.3.2 Tratamento da Chave

As inserções da chave no meio do algoritmo são feitas através das *RoundKeys* que são provenientes da cifra da chave. Esta cifra da chave é tratada por um escalonamento que possui dois componentes: Expansão da Chave e Seleção da *RoundKey*. O parágrafo seguinte descreve os princípios seguidos por este processo.

O número total de *RoundKeys* necessários é igual ao tamanho do bloco multiplicado pelo número de voltas mais um. Por exemplo: um bloco de 128 bits e 10 voltas necessita de 1408 RoundKeys.

A cifra da chave é expandida dentro de uma *Expanded Key*.

As *RoundKeys* são selecionadas da *Expanded Key* para serem usadas no algoritmo da seguinte maneira: a primeira *RoundKey* consiste das primeiras *Nb* palavras da *Expanded Key*, a segunda *RoundKey* consiste das seguintes *Nb* palavras da *Expanded Key* e assim sucessivamente.

4.3.2.1 Expansão da Chave

A *Expanded Key* é um vetor linear de palavras de 4-bytes e é denotado por *W[Nb*(Nr+1)]*. As primeiras *Nk* palavras possuem a cifra da chave enquanto as outras palavras são definidas recursivamente em palavras com índices menores. A função que calcula a *Expanded Key* depende de *Nk*, existindo uma versão da função para *Nk* < 6 e outra para *Nk* > 6.

Para *Nk* < 6 temos (em Pseudo - C):

```
KeyExpansion(byte Key[4*Nk], word W[Nb*(Nr+1)])  {

  for (i = 0; i <Nk; i++)
     W[i]= (Key[4 * i], Key[4 * i + 1], Key[4 * i + 2], Key[4 * i + 3]);
  for (i = Nk; i < Nb * (Nr + 1); i++)  {
    temp= W[i - 1];
    if (i % Nk == 0)
      temp= SubByte(RotByte(temp)) ^Rcon[i / Nk];
    W[i]= W[i - Nk] ^ temp;
    }
  }
```

Para *Nk* > 6 temos (em Pseudo - C):

```
KeyExpansion(byte Key[4*Nk], word W[Nb*(Nr+1)])  {

  for (i = 0; i <Nk; i++)
     W[i]= (Key[4 * i], Key[4 * i + 1], Key[4 * i + 2], Key[4 * i + 3]);
  for (i = Nk; i < Nb * (Nr + 1); i++)  {
    temp= W[i - 1];
    if (i % Nk == 0)
      temp= SubByte(RotByte(temp)) ^Rcon[i / Nk];
    W[i]= W[i - Nk] ^ temp;
    }
  }
```

4.3.3 O algoritmo Rijndael

O algoritmo de criptografia Rijndael é uma associação das funções descritas acima. O algoritmo se baseia em três passos:

- Uma adição inicial de uma *Round Key;*
- Nr-1 voltas;
- Uma volta final.

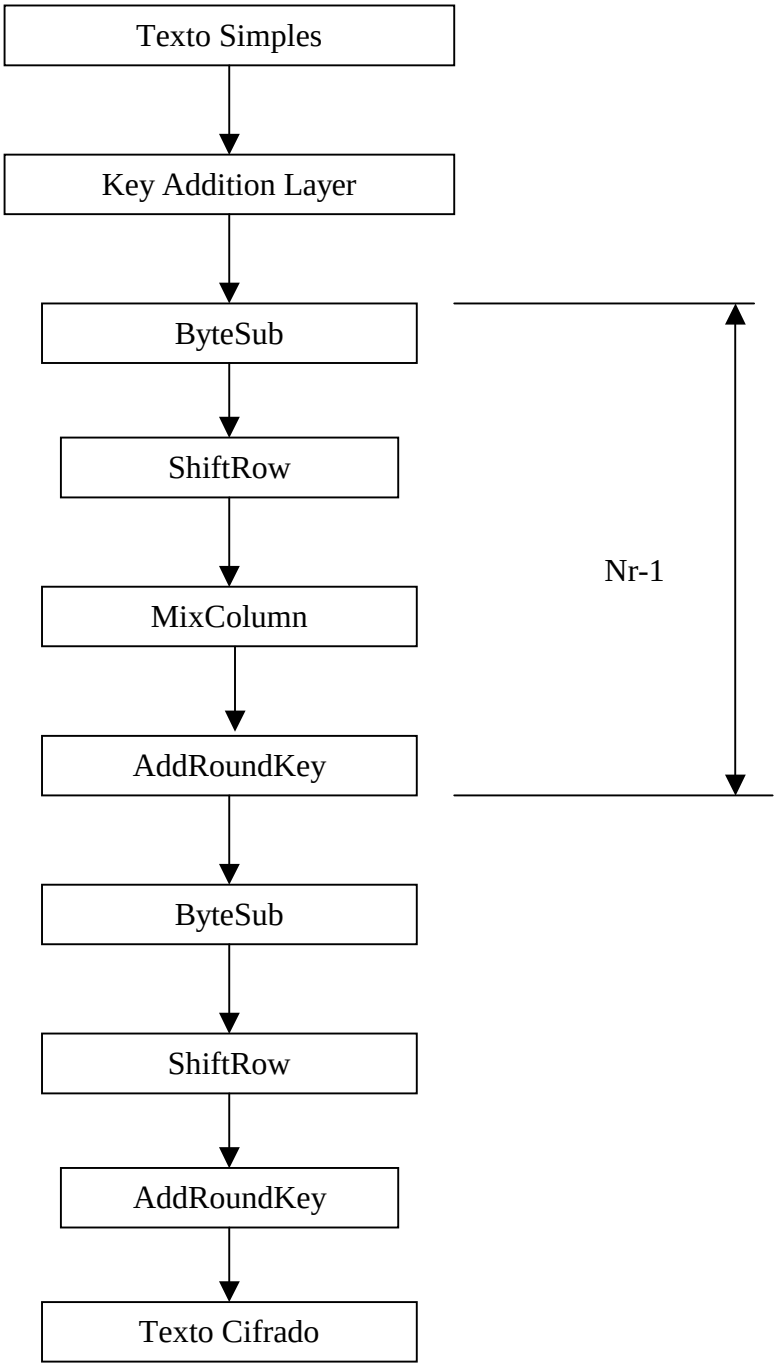

Figura 4.9. As etapas do Rijndael

4.4 Serpent

Serpent é um algoritmo apresentado ao AES por três pesquisadores, são eles: Ross Anderson da Inglaterra, Eli Biham de Israel e Lars Knudsen da Noruega.

O algoritmo Serpent possui um bloco de 128 bits dividido em 4 palavras de 32 bits cada e trabalha com um tamanho de chave de 128, 192 ou 256 bits.

Basicamente o algoritmo Serpent constitui-se de:

- uma permutação *IP*;
- 32 voltas onde se aplicam as funções criptográficas;
- uma permutação *FP*.

4.4.1 Descrição do Algoritmo

As funções criptográficas que são executadas a cada uma das 32 voltas são:

- uma inserção da chave;
- uma passagem pelas *S-Boxes*;
- uma transformação linear, que é descartada na última volta.

4.4.1.1 Permutação *IP*

A Permutação *IP* é uma permutação simples onde os 128 bits do texto são trocados de lugar conforme a tabela seguinte.

0	32	64	96	1	33	65	97	2	34	66	98	3	35	67	99
4	36	68	100	5	37	69	101	6	38	70	102	7	39	71	103
8	40	72	104	9	41	73	105	10	42	74	106	11	43	75	107
12	44	76	108	13	45	77	109	14	46	78	110	15	47	79	111
16	48	80	112	17	49	81	113	18	50	82	114	19	51	83	115
20	52	84	116	21	53	85	117	22	54	86	118	23	55	87	119
24	56	88	120	25	57	89	121	26	58	90	122	27	59	91	123
28	60	92	124	29	61	93	125	30	62	94	126	31	63	95	127

Tabela 4.3. A Permutação IP [AND 99]

Por exemplo, o bit de número 1 do bloco de texto (o segundo bit) vai para a posição do bit número 4 (o quinto bit).

A figura seguinte mostra o funcionamento do Serpent.

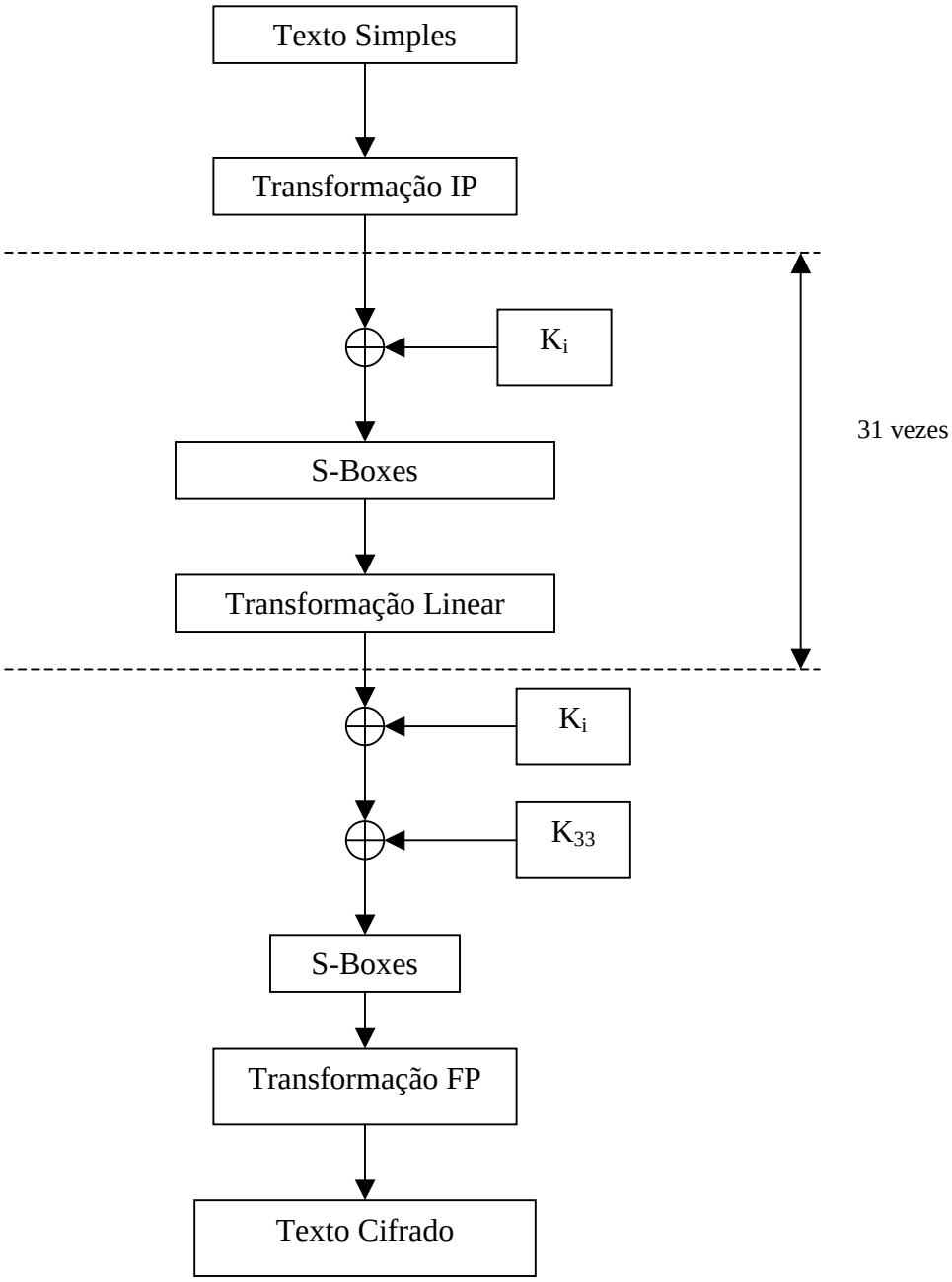

Figura 4.10. O Funcionamento do Algoritmo Serpent

4.4.1.2 Funções Criptográficas

4.4.1.2.1 Inserção da Chave

A cada volta é inserida no texto uma sub-chave que faz parte de um conjunto de 33 sub-chaves de 128, 192 ou 256 bits produzidas a partir de uma chave de tamanho variável, fornecida pelo usuário. Chaves menores do que o tamanho determinado são formatadas adicionando um bit 1 ao final da chave mais tantos bits 0 quantos forem necessário.

Uma sub-chave *Ki* é inserida no bloco de texto *Bi* através de uma operação XOR.

4.4.1.2.2 *S-Boxes*

O Serpent trabalha com 8 *S-Boxes* que são utilizadas 4 vezes cada uma. Uma *S-Box* é utilizada de cada vez, então a *S-Box S0* é utilizada na volta 0, a *S-Box S1* na volta 2 e assim por adiante até que na volta 8 a *S-Box S1* é utilizada novamente.

As *S-Boxes* do Serpent são caixas de substituição de 4 bits, então para alcançar os 32 bits de cada bloco de texto, a cada volta são criadas 8 réplicas da *S-Box* que está sendo usada. Os 32 bits do bloco de texto são distribuídos em ordem pelas 8 réplicas da *S-Box*, ou seja, os primeiros 4 bits do bloco de texto vão para a primeira réplica da *S-Box*, os próximos 4 bits vão para a segunda réplica e assim por diante.

4.4.1.2.3 Transformação Linear

Os 32 bits de cada bloco de texto são transformados linearmente da seguinte maneira:

$$
\begin{aligned}
x_0, x_1, x_2, x_3 &:= S_1 (B_i \oplus K_i) \\
x_0 &:= x_0 <<< 13 \\
x_2 &:= x_2 <<< 3 \\
x_1 &:= x_1 \oplus x_0 \oplus x_2 \\
x_3 &:= x_3 \oplus x_2 \oplus (x_0 << 3) \\
x_1 &:= x_1 <<< 1 \\
x_3 &:= x_3 <<< 7 \\
x_0 &:= x_0 \oplus x_1 \oplus x_3 \\
x_2 &:= x_2 \oplus x_3 \oplus (x_1 << 7) \\
x_0 &:= x_0 <<< 5 \\
x_2 &:= x_2 <<< 22 \\
B_{i+1} &:= x_0, x_1, x_2, x_3
\end{aligned}
$$

Onde “<<<” denota rotação e “<<” denota deslocamento.

4.4.1.3 Permutação *FP*

A Permutação *FP* segue o mesmo princípio da Permutação *IP*, entretanto as permutações são realizadas seguindo a tabela seguinte.

0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
65	69	73	77	81	85	89	93	97	101	105	109	113	117	121	125
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
66	70	74	78	82	86	90	94	98	102	106	110	114	118	122	126
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
67	71	75	79	83	87	91	95	99	103	107	111	115	119	123	127

Tabela 4.4. A Permutação FP [AND 99]

4.4.1.2 Escalonamento da chave

A chave fornecida pelo usuário é formatada para 256 bits se necessário conforme descrito no início desta seção, então esta chave é dividida em 8 pré-chaves de 32 bits

4.5 Twofish

Twofish é um algoritmo apresentado ao AES por um grupo de pesquisadores da *Counterpane Systems*. Twofish possui um bloco de 128 bits e chaves que podem ter um tamanho de 128, 192 ou 256 bits.

Algumas estruturas matemáticas e técnicas de manipulação de dados são de vital importância na implementação do Twofish, entre elas podemos destacar: *Feistel Network, S-Boxes*, Matrizes MDS, *Pseudo-Hadamard Transforms* (PHT) e *Whitening*. As estruturas que não apareceram ainda neste trabalho merecerão agora um atenção especial, enquanto que para as já descritas faremos apenas um descrição de como elas são implementadas no Twofish.

4.5.1 Funções Criptográficas usadas pelo Twofish

4.5.1.1 *Feistel Network*

Como apresentado no início deste trabalho, é usado na maioria dos algoritmos criptográficos atuais.

Twofish é um algoritmo que implementa 16 voltas de uma função *F* bijetora do tipo *Feistel Network.*

4.5.1.2 *S-Boxes*

Twofish usa quatro *S-Boxes* diferentes de 8 bits cada uma, estas *S-Boxes* são bijetoras e dependentes da chave.

4.5.1.3 Matrizes MDS

Uma matriz MDS é um mapeamento linear de elementos de um campo a para os elementos de um campo b, produzindo um vetor composto de elementos de $a+b$, com a propriedade que o número mínimo de elementos “não zero” em qualquer vetor que possua elementos não zero seja maior ou igual a $b+1$. Colocando de outra forma, a “distância”, isto é o número de elementos que diferem, entre qualquer dois vetores produzidos pelo mapeamento MDS é pelo menos $b+1$.

4.5.1.4 *Pseudo-Hamard Transforms* (PHT)

PHT é uma operação simples de mistura de dados que funciona muito bem tanto em hardware quanto em software.

Dadas duas entradas, a e b, uma PHT de 32 bits é definida como:

$$a' = a + b \bmod 2^{32}$$

Twofish usa um PHT de 32 bits para misturar a saída de duas funções *g* (que será descrita mais tarde) de 32 bits paralelas.

4.5.1.5 *Whitening*

Whitening é uma técnica de inserir a chave no texto através de um XOR antes da primeira volta e depois da última volta.

Twofish aplica um XOR com 128 bits de uma sub-chave e o texto fonte antes da primeira volta *Feistel* e outros 128 bits da sub-chave depois da última volta *Feistel*. Estas sub-chaves são calculadas da mesma maneira que as sub-chaves das voltas *Feistel*.

4.5.2 Descrição do algoritmo

Twofish usa 16 voltas tipo *Feistel* com um *whitening* adicional na entrada e na saída, as rotações de 1 bit existentes no Twofish não são consideradas parte da estrutura *Feistel*.

Na entrada do algoritmo, o texto simples é repartido em quatro partes iguais de 32 bits. É aplicado, então, um XOR em cada entrada com quatro palavras chave, esta operação é o primeiro *whitening*, após a chave ser introduzida no texto seguem as 16 voltas do tipo *Feistel*. Nestas voltas, uma das palavras da esquerda é primeiro rotacionada em 8 bits para a esquerda então junto com a outra palavra da esquerda são usadas como entrada de uma função *g*. Esta função *g* é composta de 4 *S-Boxes*, seguidas por uma mistura linear baseada numa matriz MDS. O resultado das duas funções *g* é combinado usando um PHT, após duas palavras chave são adicionadas. Para cada uma das saídas é aplicado um XOR com uma das duas palavras da direita. Em uma das saídas, é feita uma rotação de 1 bit para a esquerda antes do XOR e na outra saída é feita uma rotação de 1 bit para a direita após o XOR. Ao final de cada volta, as duas palavras da esquerda são trocadas pelas duas palavras da direita. Após as 16 voltas, as duas palavras da direita são trocadas novamente de posição pelas duas da direita e é aplicado um XOR novamente em cada palavra com mais quatro palavras chave.

A algoritmo Twofish tem a sua parte mais importante concentrada em duas funções, a função *F* (*Feistel Network*) e a função *g* que na verdade é quem faz a maior transformação dos dados.

4.5.2.1 A função F

A função *F* é uma permutação dependente de chave em valores de 64 bits. Na sua entrada são passados três parâmetros, as duas palavras de entrada (*R0* e *R1*) e também o número da volta que vai ser executada. Este último parâmetro serve para selecionar a sub-chave correta que vai ajudar na transformação dos dados.

O parâmetro *R0* é passado através da saída da função *g;* o parâmetro *R1* é rotacionado 8 bits a esquerda e então é passado pela saída de uma outra função *g*. Estes dois parâmetros *R0* e *R1*, que passarão a se chamar a partir de agora *T0* e *T1* respectivamente, são combinados usando um PHT; depois, duas palavras de chave expandida são inseridas.

4.5.1.3 A função *g*

Dentro desta função é que acontece a parte mais importante da cifragem no Twofish. Na entrada da função, a palavra de 4 bytes é dividida em 4 partes iguais de 1 byte cada uma. Então cada byte passa por uma *S-Box*. Os 4 bytes da saída das 4 *S-Box* são interpretados como um vetor de 4 posições e multiplicados numa matriz MDS de 4x4. A saída desta função é novamente uma palavra de 4 bytes.

4.5.3 O tratamento da chave

A partir de uma chave inicial, são criadas 40 chaves expandidas (*k0, k1, ..., k39*) e *4 S-Boxes* (usadas na função *g*). O Twofish trabalha com chaves de tamanho menor do que os definidos (128, 192, 256 bits) sem perder a segurança, pois o tratamento recebido pela chave torna qualquer chave fraca em chave forte. Para chaves de tamanho menor do que o exigido o tratamento da chave completa o tamanho mínimo com zeros.

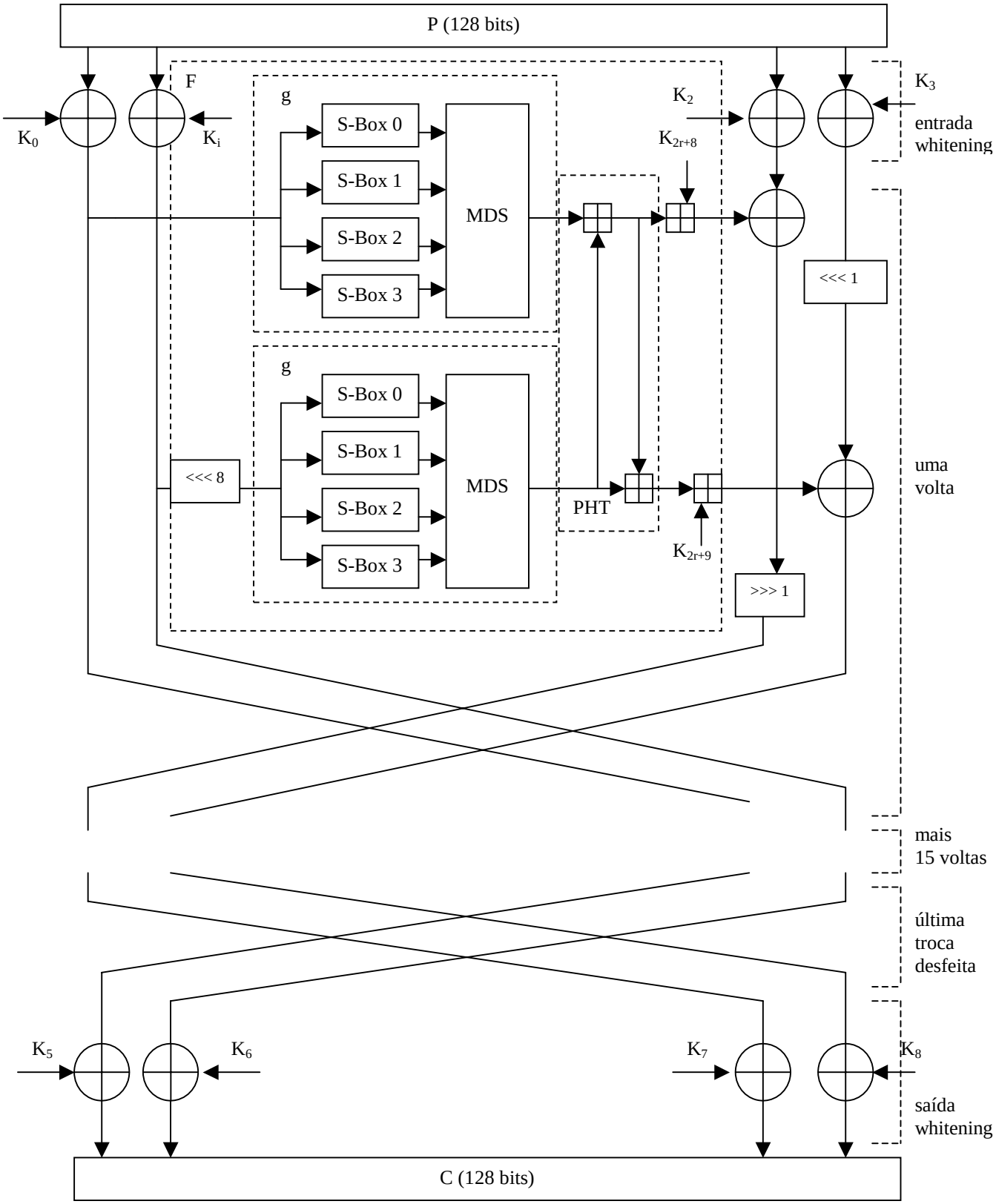

Figura 4.11 – Funcionamento do Twofish [SCH 98]

5 COMPARAÇÃO DOS ALGORITMOS

Neste capítulo se realiza a parte mais importante do trabalho, onde os algoritmos são comparados de modo que possamos analisá-los quanto aos mais relevantes fatores associados a algoritmos de criptografia, como desempenho, segurança e flexibilidade.

5.1 Quanto a complexidade computacional

Devido às exigências do NIST para a submissão de algoritmos ao AES imporem que os mesmos devem ser aplicados tanto em software quanto em hardware, os algoritmos criptográficos não são muito complexos computacionalmente.

Por complexidade computacional se entende todos os fatores que caracterizam a implementação de um algoritmo, como por exemplo: memória necessária, capacidade de ser aplicado em diversas plataformas, etc.

Uma comparação dos algoritmos quanto a complexidade computacional é uma comparação baseada em fatores subjetivos, pois depende de diversos parâmetros para se chegar a um resultado. Então, esta comparação será feita nas decorrer das seções seguintes com mais detalhamento.

5.2 Quanto aos requisitos de memória

A capacidade de um algoritmo criptografar ou decriptografar usando pouca memória é um requisito bastante valorizado na sua avaliação, visto que este algoritmo pode ser aplicado em máquinas com muita memória ou em pequenos *smart cards* que dispõem de menos de 1KB de memória.

Testes onde possa se comprovar a quantidade de memória necessária para a utilização dos algoritmos não foram feitos devido ao escopo deste trabalho. Utilizaram-se, então os dados obtidos em [SCH 00].

A implementação de um algoritmo em hardware talvez seja onde mais se deseja uma economia de memória. Por exemplo, os *smart cards*, onde são implementados algoritmos de criptografia, possuem no máximo 256 bytes de memória RAM [SCH 00].

A tabela 5.1, conforme [SCH 00], mostra a quantidade de memória utilizada pelos algoritmos em sua implementação em *smart cards*.

Algoritmo	Memória Mínima Requerida (bytes)
MARS	100
RC6	210
Rijndael	52
Serpent	50
Twofish	60

Tabela 5.1. Memória mínima requerida para implementação em *smart cards*.

Mesmo um *Smart Card* de 256 bytes não comporta todos os tipos de algoritmos aqui apresentados, pois, além do bloco de encriptação (ou decriptação), o algoritmo necessita de mais memória para outras estruturas que são utilizadas durante a aplicação do algoritmo. Teoricamente, todos os algoritmos apresentados seriam implementáveis em *Smart Cards,* pois o algoritmo que mais necessita de memória é o RC6 com 210 bytes, cerca de 56 bytes a menos que o máximo utilizável em um *Smart Card.* Entretanto, qualquer algoritmo que utiliza mais de 64 bytes não tem a sua implementação em *Smart Card* garantida.

Conclui-se, então, que somente o Rijndael, Serpent e Twofish têm sua implementação garantida em *Smart Cards*.

A implementação dos algoritmos em Software não apresenta problemas de memória, visto que a disponibilidade de memória das máquinas que implementam estes algoritmos em software é muito maior do que a necessária para a sua implementação.

Os requisitos mínimos de memória necessários para a implementação destes algoritmos em software está melhor descrito em [SCH 98, RIV 98, DAE 99, AND 99 e KAU 95].

5.3 Quanto a velocidade de processamento

Apesar de segurança ser o requisito mais importante a ser analisado em um algoritmo criptográfico, a velocidade em que se consegue processá-lo é um fator determinante para a sua utilização, pois de nada adianta a segurança de um algoritmo estar diretamente relacionada a uma quantidade de processamento impraticável na tecnologia atual.

A relação segurança-velocidade também passa pelo ponto que envolve flexibilidade, ou seja, um algoritmo necessita ser implementável nas mais diversas plataformas.

Uma precisa e relevante comparação dos algoritmos quanto a velocidade com que são processados é uma comparação onde são realizados testes nas mais diversas plataformas. Devido a escassez de tempo e de recursos tecnológicos necessários para a realização dos testes, resolveu-se utilizar os testes realizados em [SCH 00].

Os gráficos a seguir mostram, em ciclos de *clock*, o desempenho dos algoritmos criptografando em Assembler e na linguagem C, usando os tamanhos de chave exigidos pelo NIST – 128, 192 e 256 bits.

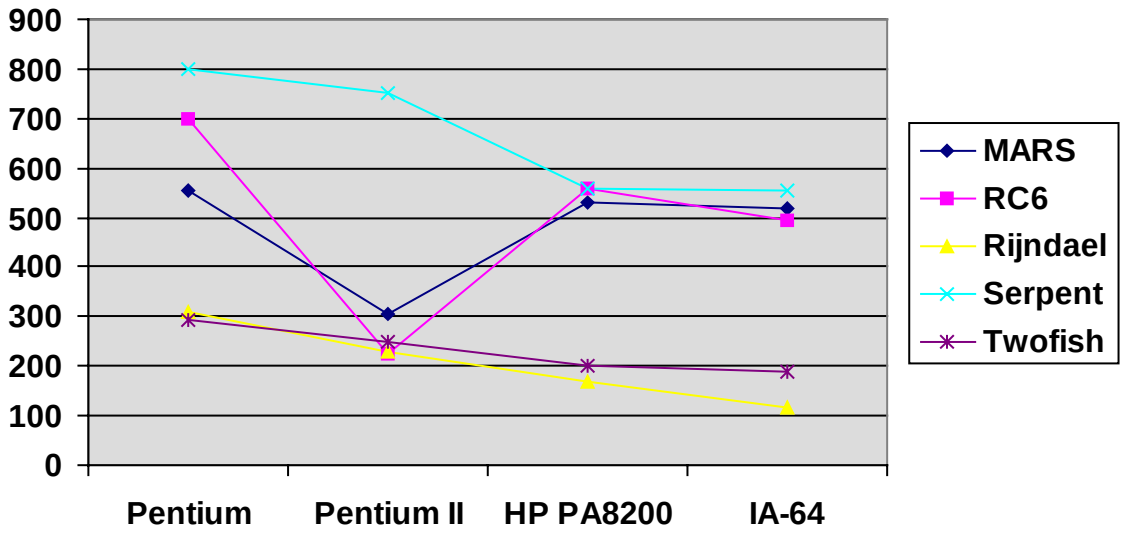

Figura 5.1 – Velocidade de Encriptação para uma chave de 128 bits em Assembler

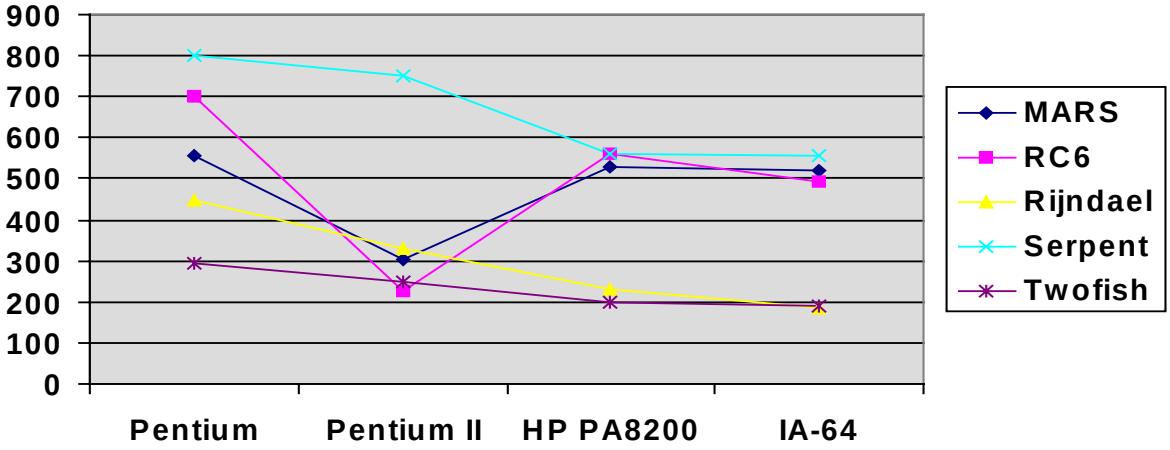

Figura 5.2 – Velocidade de Encriptação para uma chave de 192 bits em Assembler

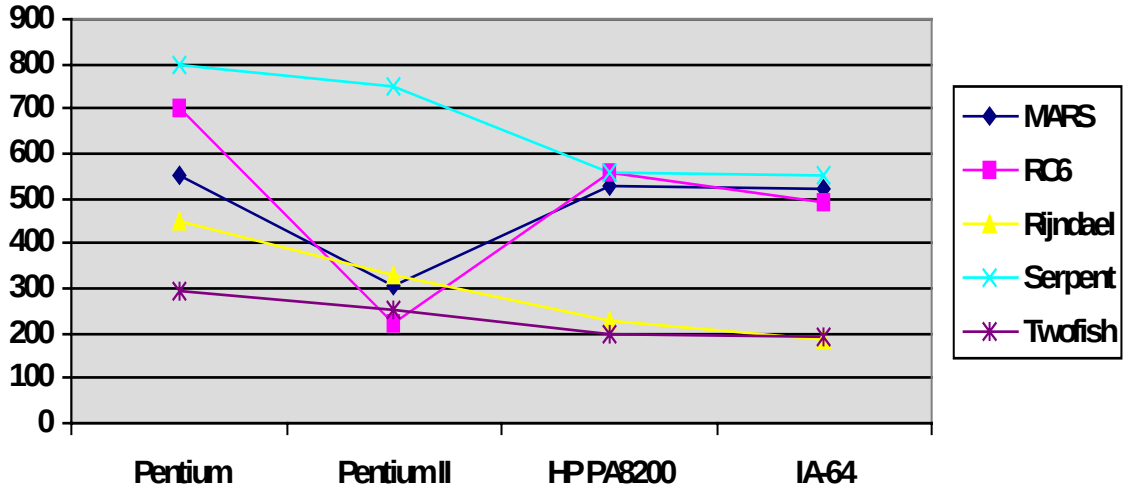

Figura 5.3 – Velocidade de Encriptação para uma chave de 256 bits em Assembler

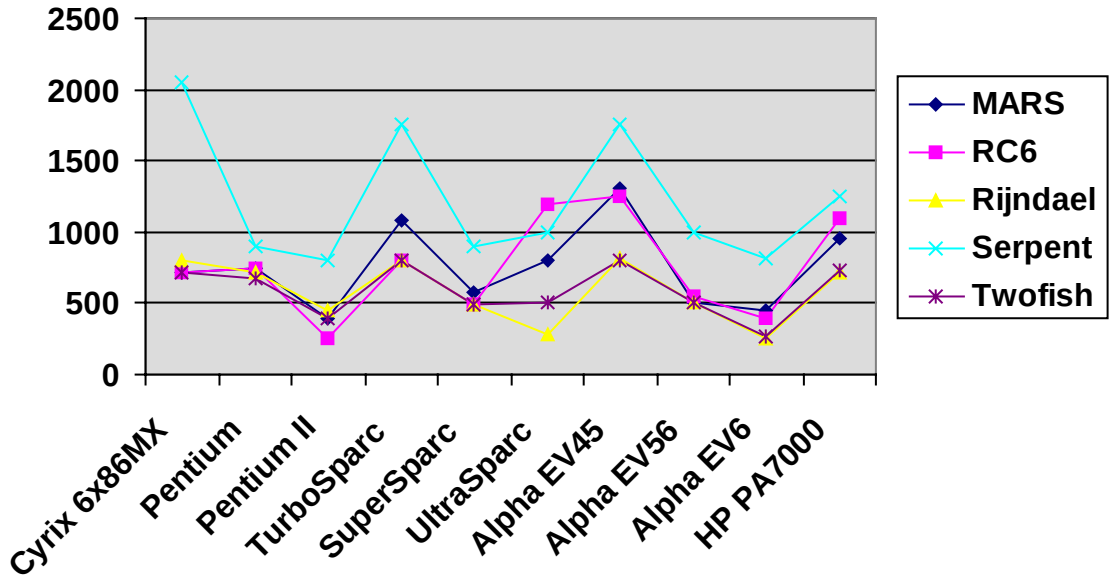

Figura 5.4 – Velocidade de Encriptação para uma chave de 128 bits em C

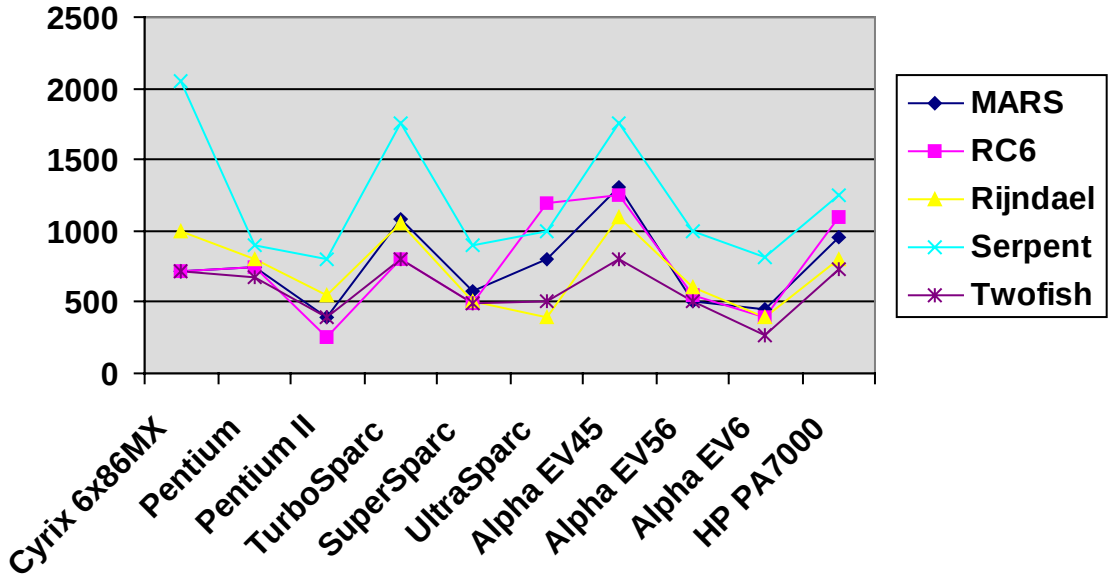

Figura 5.5 – Velocidade de Encriptação para uma chave de 192 bits em C

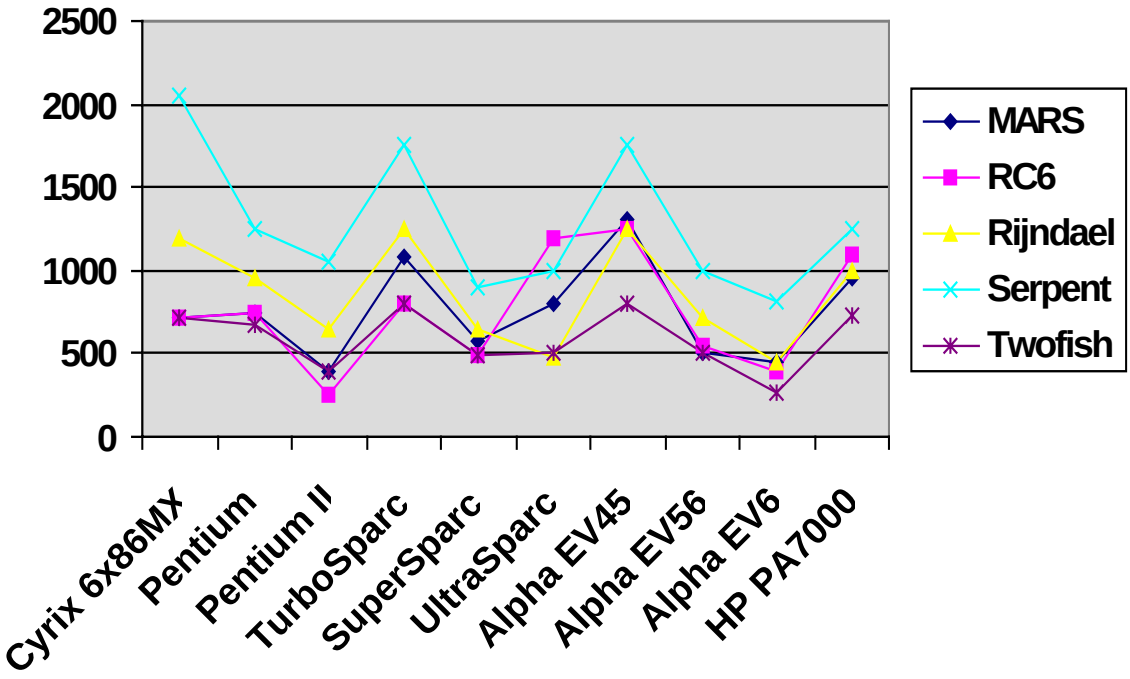

Figura 5.6 – Velocidade de Encriptação para uma chave de 256 bits em C.

Em Assembler, nota-se que a velocidade de processamento dos algoritmos é praticamente constante para os três tamanhos de chave (128, 192 e 256 bits), com a exceção do Rijndael que tem uma queda na velocidade de processamento a medida que a chave se torna maior.

Neste tipo de implementação, Rijndael e Twofish são os dois algoritmos mais rápidos, Serpent é em todos os testes o mais lento e, RC6 e MARS apresentam uma característica interessante: com exceção do Processador Pentium II eles (RC6 e MARS) são mais lentos que o Rijndael e o Twofish, entretanto apresentam a mesma velocidade de processamento neste processador. Isso se deve a dependência que estes algoritmos têm de rotações de dados e multiplicações em 32 bits, operações estas que fazem parte do conjunto de instruções do Pentium II.

Em C, repete-se a constância de velocidade de processamento para os três tamanhos de chave, e novamente o Rijndael se torna mais lento a medida que a chave se torna maior.

Com a exceção dos testes realizados num Pentium II, o Rijndael e o Twofish se apresentaram mais rápidos que os demais algoritmos.

Conclui-se que, dos dados obtidos dos testes realizados, Rijndael e Twofish são na média os algoritmos mais rápidos, tanto implementados em Assembler quanto em C.

5.4 Quanto as funções criptográficas

Os algoritmos criptográficos usam um conjunto de estruturas e funções criptográficas em comum, algumas delas apresentadas no início deste trabalho. Essas estruturas podem aparecer um pouco modificadas ou com outro nome, todavia possuem a mesma função.

Além dessas funções criptográficas os algoritmos aqui estudados também podem utilizar outras funções novas.

O estudo dessas funções criptográficas e a comparação delas entre os algoritmos nos mostra entre outras coisas a sua flexibilidade.

A tabela seguinte enumera as funções criptográficas mais comuns e quais são usadas pelos 5 algoritmos estudados.

	XOR	ADD	S-Box	Feistel	Deslocamento	Multiplicação
MARS	■	■	■	■	■	■
RC6	■	■		■	■	■
Rijndael	■	■	■		■	■
Serpent	■		■		■	
Twofish	■	■	■	■	■	■

Tabela 5.2 – Funções Criptográficas

Ao analisarmos esta tabela concluímos que MARS e Twofish são os algoritmos que utilizam um número maior de estruturas diferentes em sua implementação, esta característica pode proporcionar uma segurança maior mas diminui sua flexibilidade visto que algumas estruturas são mais fáceis de implementar em software e outras em hardware.

Teoricamente, quanto menos estruturas forem utilizadas, maior a flexibilidade do algoritmo, entretanto o Serpent, que é o algoritmo que utiliza menos estruturas mostrou-se igualmente lento tanto em C quanto em Assembler (veja seção 5.3).

6 CONCLUSÃO

O esforço do NIST em promover o AES é fundamental, pois garante a continuidade no desenvolvimento de novos algoritmos criptográficos que possam aumentar a segurança nas áreas onde eles são empregados.

Os 5 algoritmos finalistas: MARS, RC6, Rijndael, Serpent e Twofish, a que esse trabalho se propôs a descrever e comparar, podem ser considerados como o atual estado da arte em algoritmos criptográficos.

Com o resultado das comparações entre os algoritmos realizadas neste trabalho, podemos afirmar que, mesmo com a escolha de um algoritmo como o novo padrão de criptografia, os demais também têm a sua importância, pois em última análise, a sua eficiência e segurança apresentaram resultados semelhantes.

Os testes realizados neste trabalho foram realizados em uma única plataforma, uma única máquina e num único compilador. É desejável que este trabalho tenha continuidade para que testes mais aprofundados e mais diversos sejam realizados.

A escolha de um algoritmo para substituir o DES é muito difícil, pois além de apresentar maior segurança este algoritmo precisa se adequar a todas as implementações que o DES se adequava. Entretanto, com o resultado dos testes apresentados pode se tirar algumas conclusões:

- O algoritmo Serpent apesar de ser um algoritmo simples (pouco complexo) possui uma velocidade de processamento comprometedora já que, tanto para aplicações em Assembler ou na linguagem C ele se apresentou como o algoritmo mais lento.
- Os algoritmos MARS e RC6 apresentam boa velocidade de processamento somente em algumas plataformas, pois apresentam estruturas que são próprias do conjunto de instruções destas plataformas. Isso mostra que estes algoritmos são pouco flexíveis.
- Os algoritmos Rijndael e Twofish foram os algoritmos que apresentaram melhores resultados nos testes e nas comparações realizadas, mostrando que são forte candidatos a substituir o DES e ter a sua implementação difundida.

Durante o desenvolvimento deste trabalho, o NIST, após quase 4 anos de análises, chegou ao resultado final escolhendo o Rijndael como o novo modelo padrão de algoritmo criptográfico.

A partir desta escolha, o algoritmo Rijndael será mais profundamente estudado. Sugere-se que em trabalhos posteriores se realize uma comparação mais detalhada entre o

Rijndael e o DES para melhor explicitar quais são as diferenças entre os dois e as vantagens do Rijndael sobre o DES.

A proposta inicial deste trabalho previa uma comparação dos 5 algoritmos quanto a manipulação de arquivos. Contudo, essa comparação não foi realizada devido à escassez de tempo, uma vez que para uma completa descrição dos algoritmos foi utilizado mais tempo do que o previsto. A comparação dos algoritmos quanto a manipulação de arquivos é um aspecto muito relevante, e sugere-se que esta comparação seja feita em um outro trabalho acadêmico.

Finalmente, como contribuição acadêmica, espera-se que este trabalho desperte o interesse de outros acadêmicos pela pesquisa sobre segurança na informática, mais precisamente sobre criptografia.

7 BIBLIOGRAFIA

[AND 99] ANDERSON, R.; BIHAM, E.; KNUDSEN, L**.. *Serpent: A Proposal for the Advanced Encryption Standard***. 1999.

[DAE 99] DAEMEM, J.; RIJMEN, V.. ***AES Proposal: Rijndael***. 1999.

[KAU 95] KAUFMAN, C.; PERLMAN, R.; SPECINER, M***.; Network Security: Private Comunication in a Public World***; Ed. Prentice-Hall, 1995.

[IBM 99] IBM Corporation. ***MARS – a candidate cipher for AES***. 1999.

[RIV 98] RIVEST, R.L.; ROBSHAW, M.J.B.; SIDNEY, R.; YIN, Y.L**.. *The RC6 Block Cipher*. RSA Laboratories**. 1998.

[SCH 97] SCHNEIER, B.; ***Applied Cryptography (2nd edition)***. 1997.

[SCH 98] SCHNEIER, B.; KELSEY, J.; WHITING, D.; WAGNER, D.; HALL, C.; FERGUSON, N.. ***Twofish: A 128-Bit Block Cipher***. 1998.

[SCH 00] SCHNEIER, B.; WHITING, D**.; *A Performance Comparison of the Five AES Finalists***. 2000. via Web www.nist.gov/aes

[TAN 97] TANENBAUM, A.; **Redes de Computadores**. 1997.

[WEB 95] WEBER, R. F**.; Criptografia Contemporânea**. Congresso Brasileiro de Computação. Canela 1995.

[WWW 1] www.nist.gov/aes; ***Advanced Effort Standard Oficial Site***