O CRIPTO-SISTEMA RIJNDAEL

Sérgio Augusto Prazin de Barros.

Recife, 29 de julho de 2003

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

O CRITPO-SISTEMA RIJNDAEL

por

Sérgio Augusto Prazin de Barros

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da UFPE
como um dos requisitos à obtenção do título de Mestre

Orientador: Prof. Ricardo Menezes Campello de Souza, PhD.

Recife, 29 de julho de 2003

Barros, Sérgio Augusto Prazin de
O Cripto-Sistema Rijndael / Sérgio Augusto Prazin de Barros. – Recife : O Autor, 2003.
v, 97 folhas : il., fig.,tab.

Dissertação (mestrado) - Universidade Federal de Pernambuco. CTG. Engenharia Elétrica, 2003.

Inclui bibliografia, apêndices e anexos.

1. Engenharia elétrica – Telecomunicações. 2. Telecomunicações – Transmissão de dados – Segurança. 3. Criptografia – Cifras de chave secreta (Cripto-Sistema) – Análise e implementação. 4. Criptoanálise (Função *hash*) – Análise de desempenho. I. Título.

621.391(083.73) CDU(2.ed.) UFPE
621.3820285 CDD (21.ed.) BC2004-396

Universidade Federal de Pernambuco

Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE DISSERTAÇÃO DE MESTRADO DE

SÉRGIO AUGUSTO PRAZIN DE BARROS

TÍTULO

"O Cripto –Sistema Rijndael"

A comissão examinadora composta pelos professores: RICARDO MENEZES CAMPELLO DE SOUZA, DES/UFPE, VALDEMAR CARDOSO DA ROCHA JÚNIOR, DES/UFPE e MANOEL JOSÉ MACHADO SOARES LEMOS, CCEN/DM/UFPE, sob a presidência do primeiro, consideram o candidato **SÉRGIO AUGUSTO PRAZIN DE BARROS APROVADO**.

Recife, 29 de julho de 2003.

RICARDO MENEZES CAMPELLO DE SOUZA

VALDEMAR CARDOSO DA ROCHA JÚNIOR

MANOEL JOSÉ MACHADO SOARES LEMOS

Dedico este trabalho a todas as pessoas que não se acovardam diante das dificuldades e enfrentam-nas frontalmente em busca do conhecimento intelectual

Agradecimentos

Gostaria de agradecer, inicialmente, a Deus pelo dom da vida e da inteligência.

Agradeço ao Professor Ricardo Campello pelas valiosas orientações, ensinamentos, paciência e acima de tudo pela amizade que demonstrou durante este período de formação, além da paixão alvirrubra compartilhada. Também aos professores Valdemar Cardoso, Cecilio Pimentel, Marcia Mahom e Hélio Magalhães pelas importantes lições transmitidas e a todos os membros do Grupo de Pesquisa em Comunicações – CODEC – pela união e coesão vivenciada. À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo suporte financeiro.

Agradeço de forma muito especial à minha esposa Marliene e à minha filha Marina pelas noites e finais de semana sacrificados, por amor, em prol da minha formação.

Minha gratidão aos amigos Émerson Aguiar e Rodrigo Távora, ex-alunos do programa, que me incentivaram de forma decisiva durante a seleção antes do início do curso e a todos aqueles que me ajudaram ao longo desta conquista.

ÍNDICE

Resumo

Com a evolução da velocidade dos processadores modernos, usar cifras de bloco de comprimento 56 bits para a chave, como no Padrão de Cifragem de Dados (Data Encryption Standard – DES), tornou-se menos seguro. Pensando neste sentido, foi feito um concurso, iniciado em 1998, pelo Instituto Nacional de Padrões e Tecnologia (National Institute of Standards and Technology – NIST) do governo dos EUA para escolher um novo algoritmo criptográfico para substituir o antigo padrão. Em outubro de 2000 foi divulgado o resultado, o Rijndael tornou-se o Padrão Avançado de Cifragem (Advanced Encryption Standard – AES).

O AES é uma cifra de bloco iterativa cujo comprimento da chave é variável: 128, 192 ou 256 bits, operando sobre um bloco de 128 bits. Sua estrutura é baseada em transformações que utilizam a álgebra de corpos finitos, mais especificamente GF(256). Suas rodadas são seqüências de transformações provedoras dos requisitos básicos para uma cifra segura do ponto de vista criptográfico: confusão e difusão. Suas implementações nas mais diversas plataformas são velozes e ocupam pouca memória. Pode ser considerada uma cifra segura contra ataques de criptoanálise pois não existe nenhum ataque conhecido cujo desempenho seja melhor do que a busca exaustiva da chave.

Esta dissertação é o resultado de um trabalho de pesquisa minucioso que revela de forma clara, mas sem excesso de rigores matemáticos, os detalhes do funcionamento, implementação, segurança e uso do Rijndael, servindo como instrumento para interessados na área de criptografia. Ele apresenta também uma implementação prática e didática da cifra, além da proposta e análise de uma função hash baseada no algoritmo.

Abstract

With the evolution of modern processors, the use of block ciphers with key length 56 bits, as in the Data Encryption Standard – DES, has become less and less secure. Considering this state of affairs, the US National Institute of Standards and Technology (NIST) decided, in 1997, to start a process that would lead to the creation of a new encryption standard, the so-called Advanced Encryption Standard (AES). The new standard was chosen through an open contest organized and run by NIST, which announced, in October 2000, Rijndael as its winner. Thus, the old data encryption standard was replaced by the Rijndael cryptosystem, which became the Advanced Encryption Standard -AES.

The AES is an iterative block cipher with variable key length (128, 192 or 256 bits) which operates over 128 bits message blocks. Its structure is based on Galois fields algebra, particularly the field GF(256). The implementation of AES in several different processors is very eficient, in terms of both speed and memory requirements. As far as security is concerned, up to this moment in time, the AES may be considered secure against the known cryptanalytic attacks, since that attacks that are more efficient than exhaustive search key, are yet to be found.

This dissertation is the result of a particular research work which reveals, in a clear way, but without extreme mathematical rigor, details of operation, implementation, security and use of Rijndael, serving as a useful instrument to people interested in cryptography. It presents a didactic and practical implementation of the Rijndael cryptosystem and its use as a hash function generator is proposed.

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

Neste início de século XXI vemos um mundo cada vez mais dependente dos sistemas de comunicações. Esta dependência gera uma crescente demanda na qualidade do tráfego de informações. Um dos aspectos mais relevantes à qualidade dos sistemas de comunicações é a segurança. Atualmente, bilhões de dólares circulam diariamente pelo mundo através de transações financeiras realizadas em ambiente computacional. Bytes circulam o planeta carregando informações requerentes de um alto grau de sigilo e autenticidade. Como prover a segurança necessária para ocorrência destas operações sem nenhum tipo de prejuízo?

A criptografia moderna é uma ciência capaz de prover segredo, autenticidade e integridade dos dados armazenados ou transmitidos através de sistemas computacionais. As origens da criptografia clássica remontam à antiguidade, onde comandantes militares e reis precisavam proteger a informação da interceptação por seus inimigos, ao trocar mensagens entre si. A criptografia estuda o segredo da escrita. Na própria mensagem (texto claro) é aplicado um método reversível de modificação do texto (cifragem) de modo que a mensagem modificada (texto cifrado) possa trafegar num ambiente inseguro até chegar ao seu destinatário. O receptor é possuidor de um conhecimento prévio para desfazer o método de modificação da mensagem (decifragem) e recuperar o texto original. Deste modo, a criptografia é capaz de prover a segurança para o tráfego de informações num ambiente inseguro. Para a ocorrência disto pode ser necessária uma troca de um segredo entre os usuários por um meio de comunicação totalmente seguro, o qual será utilizado na cifragem e na decifragem da mensagem: a chave. A este tipo de sistema criptográfico chamamos de cripto-sistema de chave secreta ou simétrico.

Em meados dos anos 70, o Instituto Nacional de Padrões e Tecnologia norte-americano (NIST-National Institute of Standards and Technology), na época Bureau Nacional de Padrões (NBS-National Bureau of Standards), aprovou o uso de um algoritmo

denominado Padrão de Cifragem de Dados (DES-Data Encryption Standard), para cifrar comunicações governamentais não classificadas. Sua escolha foi baseada em um algoritmo chamado Lucifer, criado por alguns dos mais famosos pesquisadores da área, na época, incluindo Host Feistel [Feis73]. Não existiam outras cifras capazes de ocupar tal posto. O algoritmo adotado não foi integralmente o Lucifer, algumas alterações foram feitas principalmente em relação ao comprimento da chave que, na cifra original, tinha 128 bits e no DES apenas 56. Na época, foi comentada na comunidade de pesquisadores em criptografia a existência de uma possível armadilha colocada na cifra pela Agência de Segurança Nacional norte-americana (NSA), para permitir sua manipulação. Apesar disto, este padrão rapidamente tornou-se o padrão mundial para comunicações do gênero.

Com a evolução tecnológica tornou-se visível a possibilidade do DES ser ineficiente e no começo de 1997 o NIST declarou aberta uma iniciativa para desenvolver um novo padrão. Três anos mais tarde, em outubro de 2000, o Rijndael foi escolhido como o novo padrão de cifragem de dados [Nech99].

O Rijndael recebeu o nome de Padrão Avançado de Cifragem (AES-Advanced Encryption Standard), e foi publicado como um Padrão Federal de Processamento de Dados (FIPS-Federal Institute Processing Standard) e, assim como o DES, foi rapidamente adotado mundialmente como padrão criptográfico, principalmente pelo fato de ser livre de direitos autorais e pela facilidade de implementação em um grande número de plataformas distintas [FIPS197].

A seleção do AES foi baseada em critérios muito bem definidos e a eliminação dos outros 14 candidatos foi fundamentada individualmente. Esta eliminação ocorreu em etapas preliminares até restarem apenas 5 candidatos. Durante este processo pôde-se observar, principalmente nos finalistas, a apresentação de um alto nível de qualidade. Isto demonstrou o amadurecimento da criptografia como ciência, ao contrário da época da escolha do DES. O Rijndael tem uma estrutura algébrica muito bem definida e baseada em sólidos fundamentos teóricos parecendo não dar margem a ocultação de armadilhas criptográficas.

Após a escolha do AES muitos pesquisadores o tem estudado. Uma série de artigos foi publicada principalmente no tocante à segurança da cifra. Até agora, nada de grande relevância pôde afetar o status de cifra segura obtida no processo de seleção. Surge então

uma tendência de ataques algébricos contra a cifra buscando mostrar as suas fragilidades [CoPi02].

Este trabalho está estruturado em seis capítulos, dentre os quais, o primeiro é esta introdução.

O segundo capítulo expõe algumas premissas necessárias à compreensão do restante da dissertação. Ele inicia contando a história do processo de seleção do Rijndael como o AES e como foi a evolução deste. Descreevem-se ainda aspectos básicos de Criptografia, tais como od conceitos de cifra de bloco e cifra iterativa, classes de cifras nas quais o Rijndael se enquadra. Finalmente, são introduzidas algumas premissas algébricas, descrevendo conceitos de entidades matemáticas relevantes para o entendimento das operações durante o processo de funcionamento do algoritmo. Lá encontram-se conceitos de grupos, anéis e corpos finitos, sendo esta última estrutura muito relevante para o trabalho, uma vez que o Rijndael usa a aritmética do corpo finito GF(256).

O terceiro capítulo mostra todos os detalhes de funcionamento da cifragem e da decifragem incluindo a estrutura de entrada\saída de dados da cifra e as funções que compõem a parte iterativa (rodada) da cifra. Ainda é exposto como ocorre o processo de expansão da chave do usuário e como se dá a escolha da chave das rodadas. Aspectos relevantes de implementação para processadores de 8 e 32 bits ou mais são ressaltados visando uma otimização do desempenho da cifra.

No quarto capítulo é mostrado um programa em linguagem C desenvolvido com a intenção de facilitar a compreensão do funcionamento da cifragem e da decifragem pois mostra todas as etapas intermediárias do processo. Ainda neste capítulo é vista uma implementação útil do Rijndael como uma função hash unidirecional. Esta implementação e sua avaliação de desempenho resultou em um artigo a ser publicado no 26º Congresso Nacional de Matemática Computacional Aplicada - CNMAC.

No quinto capítulo é vista a criptoanálise do Rijndael. No início se encontram as justificativas para a resistência contra a criptoanálise linear e diferencial, como critério de projeto, obtida através da estratégia do trilho largo. Mais adiante são abordados os ataques por diferenciais truncados. O ataque Square, ou de saturação, tem especial destaque por ter sido desenvolvido pelos autores da cifra. O ataque de Gilbert-Minier, por interpolação e com chaves relacionadas, é também abordado. Os menos convencionais ataques de

implementação têm uma seção reservada para comentá-los. Os recentes ataques com sistemas de equações superdefinidas são também comentados, sendo também indicada a não existência de chaves fracas como no DES ou IDEA.

O capítulo 6 apresenta as conclusões e sugestões para futuros trabalhos relacionados a esta dissertação.

Existem ainda 4 apêndices. O apêndice A contêm as tabelas das funções SubByte( ) e InvSubByte( ). O apêndice B contém exemplos de expansão da chave com 128, 192 e 256 bits. O apêndice C contém o código-fonte da implementação didática em linguagem C descrita no capítulo 3 e o apêndice D exemplifica uma cifragem do Rijndael passo a passo com Nb=Nk=4.

Após as referências bibliográficas existe uma breve biografia do matemático francês Evariste Galois.

Esta dissertação é fruto de um trabalho de pesquisa minucioso que revela de forma clara e detalhada, mas sem excesso de rigores matemáticos, os detalhes do funcionamento, implementação, segurança e uso do Rijndael. Pode servir como instrumento para pesquisadores em criptografia e segurança de dados devido ao seu caráter acadêmico, também para alunos de engenharia e informática em busca de uma descrição sucinta e acessível ao AES e a profissionais da área de segurança da informação buscando conhecimento sobre a cifra além de sua implementação prática.

A implementação prática existente neste documento é totalmente utilizável, isto é, toda parte de entrada/saída de dados está codificada. Seu código-fonte usa comandos do ANSI-C ou seja é compilável na maioria dos compiladores disponíveis e não precisa nada mais do que as bibliotecas padrão do C <stdio.h> e <stdlib.h> e tem uso irrestrito.

CAPÍTULO 2

Premissas

Neste capítulo veremos inicialmente como ocorreu o processo de seleção do AES. Quais foram os critérios de projeto determinados pelo NIST e como eles influenciaram as escolhas e eliminações das cifras nas diversas etapas da análise das candidatas.

Alguns conceitos básicos em criptografia são também apresentados e, visando uma melhor compreensão do material apresentado nos capítulos seguintes, é dada uma visão introdutória do que são estruturas algébricas como grupos, anéis e corpos finitos. Representaremos polinômios sobre o corpo finito GF(256) e realizaremos operações sobre este corpo e assim estaremos preparados para entender como operam as funções utilizadas pelo Rijndael.

2.1. Processo de seleção do AES

Em janeiro de 1997, o NIST anunciou o início de uma iniciativa para desenvolver um novo padrão de cifragem para substituir o DES. Em setembro do mesmo ano foram publicados os requisitos necessários à cifra. Ela deveria ser uma cifra de bloco simétrica capaz de trabalhar com blocos de comprimento de 128 bits e chaves de comprimento 128, 196 e 256 bits. Foi declarada a necessidade de uma cifra "tão segura quanto o DES-triplo, porém muito mais eficiente".

Foram aceitas como candidatas 15 cifras. A tabela 1 mostra em ordem alfabética as 15 cifras aceitas até 15 de maio de 1998.

O processo foi dividido em alguns estágios com conferências públicas no final de cada um deles. A Primeira Conferência de Candidatas a Padrão de Cifragem Avançado ocorreu de 20 a 22 de agosto de 1998, onde foram apresentadas as candidatas para a apreciação da comunidade internacional. As cifras foram avaliadas por quatro principais

características: segurança, custo, características de implementação e algoritmo. O NIST convidou a comunidade criptográfica a montar ataques contra as cifras. Estas contribuições foram utilizadas nos critérios de seleção das cinco finalistas.

Candidata	Submetedor	Tipo de submetedor
CAST-256	Entrust (CA)	Empresa
Crypton	Future Systems (KR)	Empresa
DEAL	Outerbridge, Knudsen (EUA-DIN)	Pesquisadores
DFC	ENS-CNRS (FR)	Pesquisadores
E2	NTT (JP)	Empresa
Frog	TecApro (CR)	Empresa
HPC	Schroeppel (EUA)	Pesquisador
LOKI97	Brown et al. (AU)	Pesquisadores
Magenta	Deutsche Telekom (DIN)	Empresa
Mars	IBM (EUA)	Empresa
RC6	RSA (EUA)	Empresa
Rijndael	Daemen e Rijman (BEL)	Pesquisadores
SAFER+	Cylink (EUA)	Empresa
Serpent	Anderson, Biham, Knudsen (RU-IRL-DIN)	Pesquisadores
Twofish	Counterpane (EUA)	Empresa

Tabela 1.1 . *As 15 candidatas inicialmente aceitas*

Em março de 1999 aconteceu o *workshop* anual da *Fast Software Encryption* em Roma, Itália [Baud99]. Neste evento ocorreu a Segunda Conferência de Candidatas a Padrão de Cifragem Avançado e foi divulgada uma lista com o nome de cinco desclassificadas. As cifras Frog, LOKI97, Magenta, DEAL foram eliminadas por não satisfazer aos requisitos de segurança impostos pelo NIST. Um artigo anterior já havia mostrado fraqueza no HPC o que o eliminou também. Restavam então 10 candidatas.

Em agosto de 1999, foi divulgada pelo NIST a lista dos cinco finalistas e o motivo da desclassificação das outras cinco. Os finalistas foram: MARS, RC6, Rijndael, Serpent e Twofish. As justificativas da eliminação foram: [NBBD99]

CAST-256: comparável ao Serpent mas com um alto custo de implementação.

Crypton: comparável ao Rijndael e Twofish mas com uma margem de segurança pequena.

DFC: margem de segurança pequena e má performance em processadores diferentes de 64 bits.

E2: estruturalmente comparável ao Rijndael e Twofish mas com uma margem de segurança pequena e alto custo de implementação.

SAFER+: margem de segurança alta em relação ao Serpent mas comparavelmente mais lento.

Após este anúncio, foram solicitadas novas contribuições enfocando os cinco finalistas. Em abril de 2000, ocorreu a Terceira Conferência de Candidatas a Padrão de Cifragem Avançado em Nova York [AES00], durante o *Workshop* da *Fast Encryption Software,* como tinha ocorrido na conferência anterior. Nas sessões sobre ataque criptográfico foi confirmada a conclusão da segunda conferência. Todas as cifras eram seguras e todos os ataques montados tiveram relevância apenas de caráter acadêmico, pois foram no máximo um pouco melhores do que a busca exaustiva da chave. Nas sessões de implementação em *hardware* dedicado notou-se um excelente desempenho do Serpent. O Rijndael e o Twofish foram considerados satisfatórios. O RC6 foi avaliado como caro devido as suas multiplicações em 32 bits e a implementação do MARS pareceu cara também. Porém, na avaliação geral dos organizadores o Rijndael foi o mais votado.

Em 2 de outubro de 2000, através do [FIPS-197], o NIST anunciou o Rijndael como o novo padrão a ser adotado [Nech99]. Os motivos da escolha foram:

"O Rijndael aparenta ter consistentemente um bom desempenho, tanto em *hardware* quanto em *software* em uma larga faixa de ambientes computacionais, sem levar em conta o seu uso nos modos de realimentação ou não realimentação. Seu tempo de expansão da chave é excelente, e a agilidade da sua chave é boa. Seu baixo requerimento de memória o faz bem apropriado para ambientes de espaço restrito, nos quais ele também apresenta excelente desempenho. As operações do Rijndael estão entre as mais fáceis de defender contra ataques de tempo e de análise de potência. Adicionalmente, ele aparenta permitir implementações de defesa contra esses ataques sem impacto significante sobre seu desempenho.

Finalmente, a estrutura interna da rodada do Rijndael aparenta ter bom potencial para se beneficiar dos níveis de instrução de paralelismo."

2.2. Cifras de Bloco

Quando imaginamos inicialmente uma cifra criptográfica, como foi descrito anteriormente, pensamos em algo semelhante à figura 2.1. A chave utilizada na cifragem é a mesma da decifragem, disto vem o nome de criptografia de chave simétrica. A chave é distribuída de uma forma segura e não pode ser conhecida de um possível bisbilhoteiro do canal, daí o nome de criptografia de chave secreta.

Uma outra forma de aplicação da criptografia consiste em utilizar duas chaves, uma para cifragem (chave pública) e outra para a decifragem (chave privada). A este tipo de operação chamamos de criptografia de chave pública. Como o Rijndael é um modelo de cifra de chave secreta, este assunto não será alvo da nossa abordagem.

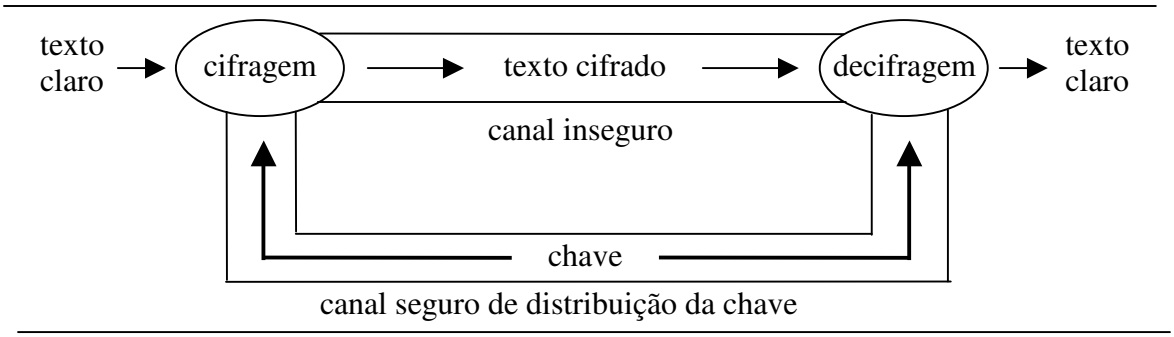

Figura 2.1: *cripto-sistema de chave secreta*

Dentre os tipos de cifras de chave secreta, uma categoria é particularmente interessante: as cifras de bloco. Este tipo de cifra opera cifrando blocos de texto claro de comprimento fixo. Por exemplo, o Rijndael cifra blocos de 128 bits de texto claro em blocos de 128 bits de texto cifrado.

Cifrar em blocos é uma grande vantagem pois os textos claros normalmente são bytes computacionais e tem comprimento fixo. Deste modo, uma cifra de bloco é mais facilmente implementada computacionalmente. Por outro lado, esta padronização de comprimento se reflete no texto claro que terá o mesmo comprimento do bloco. Um mesmo bloco cifrado com a mesma chave produzirá um mesmo texto cifrado. Para resolver este problema são utilizados diferentes modos de operação. Os modos de operação mais comuns

são os de Encadeamento de Bloco da Cifra (CBC – Cipher Blcok Chaning), Realimentação da Cifra (CFB – Cipher Feedback) e Realimentação da Saída da Cifra (OFB – Output Feedback). Estes modos operam sobre um comprimento fixo de entrada. Quando o texto de entrada não tem o comprimento necessário deve ser completado por um "enchimento" de bytes ou bits de valor pré-determinado.

O modo direto de cifragem é chamado de dicionário eletrônico (ECB – Eletronic Codebook) e permite a passagem de padrões entre o texto claro e o texto cifrado. Dependendo da aplicação deve ser evitado.

2.3. Cifras Iterativas

Existe um grupo de cifras, chamadas iterativas, cuja composição inclui basicamente um conjunto de transformações que são aplicadas repetidamente aos resultados intermediários. Cada um destes conjuntos não é necessariamente forte do ponto de vista criptográfico mas a repetição deles provê a segurança necessária à cifra. As transformações que se repetem são chamadas de rodada e pode ser precedida de uma rodada inicial e sucedida por uma rodada final. Podemos visualizar a idéia na figura 2.2.

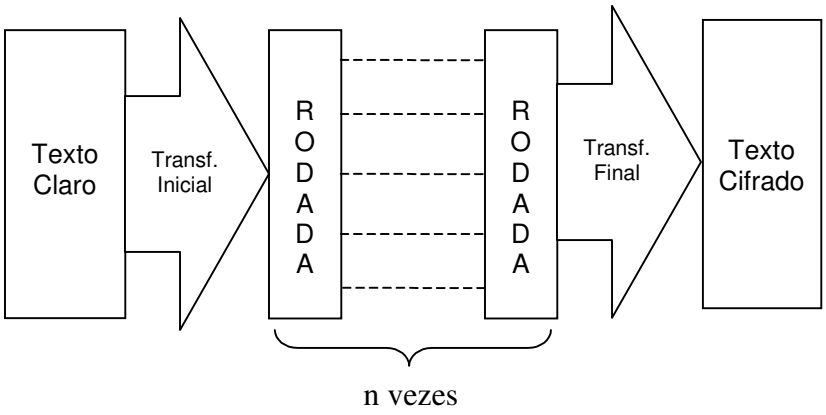

Figura 2.2: *Composição de uma cifra iterativa*

Quando uma cifra de bloco iterativa sofre uma adição da chave da rodada antes da primeira rodada e após a última rodada sempre através de uma operação simples de adição módulo 2 (ou-exclusivo bit a bit) ela é classificada como uma cifra de bloco com chaves

alternantes. Esta classe de cifras de bloco na qual o Rijndael está inserido permite uma análise de sua resistência contra ataques de criptoanálise linear e diferencial que será vista no capítulo 5.

2.4. Bases Matemáticas

Muitas cifras utilizam-se de ferramentas matemáticas para permitir a sua implementação computacional e prover a segurança necessária contra ataques criptoanalíticos. Uma estrutura algébrica muito usada no meio criptográfico são os corpos finitos. Eles, também chamados campos de Galois (GF-Galois Field), tem particular importância para o Rijndael. Antes de estudá-los veremos o que são as abrangentes estruturas chamadas grupos, passaremos pelos anéis e chegaremos aos corpos, onde veremos a representação polinomial de GF(256) e as operações de adição e multiplicação que o definem.

2.4.1. Grupos, anéis e corpos

Grupos:

Def: Um grupo é uma estrutura algébrica $< G , * >$ formada por um conjunto G e uma operação binária $*$ sobre G, satisfazendo as propriedades que seguem:

1. Fechamento: $\forall$ a,b $\in$ G, a $*$ b $\in$ G .
2. Associatividade: $*$ é associativa, isto é, $\forall$ a,b,c $\in$ G,

$$a * b * c = a * (b * c) = (a * b) * c.$$

3. Elemento identidade: existe um elemento *e* em G de forma que para todo a $\in$ G,

$$a * e = e * a = a.$$

4. Elemento inverso: para cada a $\in$ G, existe apenas um elemento $a^{-1} \in$ G onde,

$$a * a^{-1} = a^{-1} * a = e.$$

Se o grupo também satisfizer:

5. Comutatividade: para todo a, b $\in$ G,

$$a * b = b * a,$$

então o mesmo é chamado abeliano ou comutativo.

Como exemplo de um grupo podemos tomar o conjunto dos números inteiros e a operação de adição denotado por $< Z , + >$. A propriedade da associatividade é satisfeita, o elemento identidade é 0 e o elemento inverso de a é –a. Como a propriedade 5 também é satisfeita podemos dizer que este é um grupo abeliano.

Uma estrutura que atende apenas às duas primeiras propriedades da definição anterior (fechamento e associatividade) é chamada semigrupo.

Anéis:

Def: Um anel $< R , + , \cdot >$ é um conjunto R , associado a duas operações, denotadas por + e $\cdot$ **,** onde:

1. A estrutura $< R , + >$ é um grupo abeliano.
2. A estrutura $< R , \cdot >$ é um semigrupo.
3. As leis distributivas são satisfeitas; isto é, $\forall$ a,b e c $\in$ R temos:

   $a \cdot (b + c) = a \cdot b + a \cdot c$ e $(b + c) \cdot a = b \cdot a + c \cdot a$.

Se a operação $\cdot$ for comutativa, isto é, se $\forall$ a,b $\in$ R, $a \cdot b = b \cdot a$, então $< R , + , \cdot >$ é dito ser um anel comutativo.

Se há um elemento identidade associado à operação $\cdot$, então $< R , + , \cdot >$ é dito ser um anel com identidade.

Se a estrutura $< R - \{0\}, \cdot >$ é fechada, então o anel é dito não ter divisores de zero.

Um exemplo de anel é o conjunto dos números inteiros munido das operações de adição e multiplicação $<Z, + , \cdot >$. Além de exemplificar um anel simples, esta estrutura também é um anel comutativo, com identidade e sem divisores de zero.

Corpos:

Def: Uma estrutura $< F , + , \cdot >$ é um corpo se satisfizer as propriedades que seguem:

1. $< F , + , \cdot >$ é um anel comutativo, com identidade e sem divisores de zero.
2. Para todo elemento de F, existe um elemento inverso em F com respeito a operação $\cdot$, exceto para o elemento *e* , que é o elemento identidade de $< F , + >$.

Um exemplo de corpo é o conjunto dos números reais juntamente com as operações de adição e multiplicação.

Corpos finitos:

São corpos onde a quantidade de elementos é finita. A ordem de um corpo finito, ou seja a quantidade de elementos deste, é $q = p^m$, para m inteiro e p primo [Lidl98]. Então, um corpo de ordem q existe, se e somente se q é uma potência de um primo. Chamamos p de característica do corpo e para o Rijndael p é igual a 2.

Corpos de mesma ordem são isomórficos, isto é, existe apenas um corpo para cada potência de um primo. As maneiras de representá-lo podem ser diferentes, mas sempre se referindo a uma mesma estrutura algébrica.

Quando m=1, GF(p) é formado pelos inteiros módulo p, juntamente com as operações de soma e multiplicação módulo p. GF(p) é composto então por dois grupos, $< Z_p , +_p >$ e $< Z_p^* , \cdot_p >$.

Quando temos um corpo de ordem $q = p^m$ onde $m \neq 1$, os elementos de GF(q) não são mais inteiros módulo q, e sim polinômios. Essa representação polinomial não é a única representação para $GF(p^m)$, mas é a utilizada pelo Rijndael.

2.4.2. Polinômios sobre um corpo

Para as aplicações computacionais onde o Rijndael pode ser utilizado, existem as unidades de armazenamento bit e byte. Como um byte tem 8 bits, uma representação prática é através do corpo finito $GF(2^8)$. Podemos então visualizar o polinômio:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

como um conjunto de bits $b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$, onde $b_i \in \{0,1\}$, formando um byte.

Podemos tomar como exemplo o byte ‘FC’ em hexadecimal (‘11111100’ em binário) como o polinômio:

$$x^7 + x^6 + x^5 + x^4 + x^3 + x^2 .$$

2.4.3. Operações polinomiais sobre GF(256)

Adição:

A adição polinomial pode ser imaginada como uma adição de coeficientes inteiros num corpo de característica 2, ou seja os coeficientes são adicionados módulo 2. Tomemos como exemplo a adição que segue:

$$\begin{array}{rl} & x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 \\ + & \quad x^6 + \quad x^4 + \quad x^2 + \quad 1 \\ \hline & x^7 + \quad x^5 + \quad x^3 \quad\quad x \end{array}$$

A implementação válida computacionalmente é operar uma soma em ou-exclusivo entre as representações dos polinômios bit-a-bit. ‘FF’⊕ ‘55’= ‘CC’

$$\begin{array}{rl} & 11111111\ (\text{‘FF’}) \\ + & 01010101\ (\text{‘55’}) \\ \hline & 10101010\ (\text{‘CC’}) \end{array}$$

A operação de ou-exclusivo (⊕) bit a bit representa a adição polinomial e será algumas vezes simplesmente denotada por +.

Multiplicação:

A multiplicação na representação polinomial, representada por •, é o produto entre dois polinômios reduzidos módulo um terceiro polinômio. Este último polinômio tem fundamental importância na execução desta operação. Num corpo $GF(p^m)$ devemos escolher um polinômio, com coeficientes em GF(p), de grau m e irredutível em GF(p) para possibilitar a redução polinomial. Um polinômio m(x) é irredutível sobre GF(p) se, e somente se, não existam dois polinômios a(x) e b(x) com coeficientes em GF(p) de forma que m(x)=a(x) • b(x), onde a(x) e b(x) tem grau maior que zero. No nosso caso, o polinômio de grau 8 escolhido para montar $GF(2^8)$ será denotado por m(x):

$$m(x) = x^8 + x^4 + x^3 + x + 1 \qquad (1)$$

ou (‘11B’) em notação hexadecimal.

Um polinômio, na representação no nível de byte, pode ter grau máximo 7. Como a operação de multiplicação simples entre dois polinômios permite grau máximo de 14 para o resultado, não podemos representá-lo no nível de byte. A redução módulo m(x) do produto, reduz seu grau, permitindo um grau máximo de 7. Vejamos o exemplo abaixo:

$$(x^6 + x^5 + x^4 + 1) \bullet (x^7 + x^3 + x) = x^{13} + x^{12} + x^{11} + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + x ,$$

$$(x^{13} + x^{12} + x^{11} + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + x) \text{ módulo } (x^8 + x^4 + x^3 + x + 1)$$
$$= x^7 + x^6 + x^5 + x^4 + x^3 + 1.$$

Infelizmente, não temos uma operação algébrica simples para descrever a multiplicação como temos o ou-exclusivo para adição. Podemos então executar computacionalmente a multiplicação com o uso de mais uma operação, o deslocamento de n bits à esquerda, que funcionará como uma multiplicação por x^n. Em seguida, basta aplicar ou-exclusivo entre os coeficientes das potências de mesmo expoente. Do resultado reduzimos sucessivamente m(x) até que não existam potências superiores a x^7. Veja a figura 2.3 ilustrando a operação.

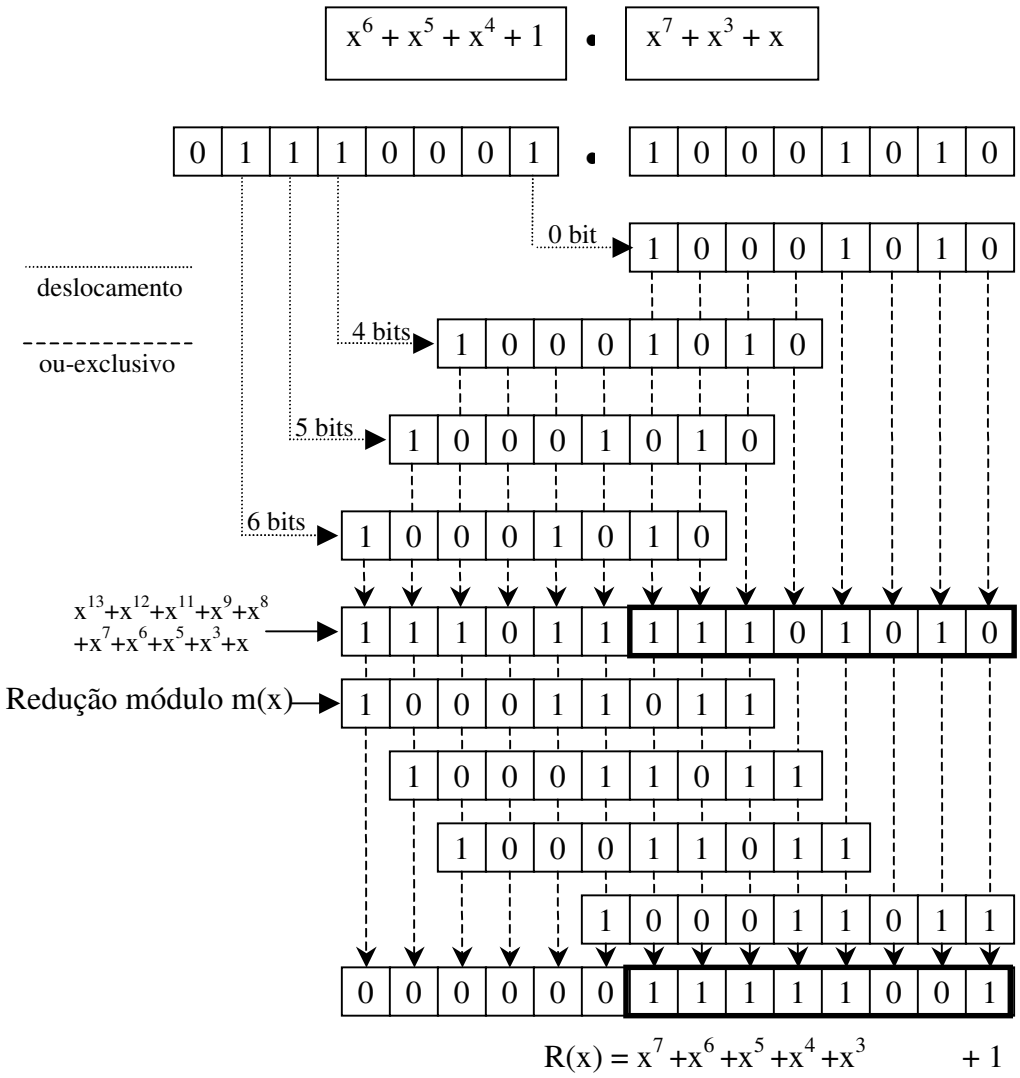

Figura 2.3: *Operação de multiplicação em GF(256)*

Esta estrutura, por ter na sua composição apenas instruções simples, permite a utilização desta operação algébrica usada pelo Rijndael em microprocessadores de pequeno porte ou microcontroladores.

Existência de GF(256):

Definimos as duas operações sobre GF(256): adição e multiplicação módulo m(x). Em relação à adição temos um grupo abeliano. Quanto à multiplicação, é associativa e tem como elemento identidade (‘01’). O polinômio A(x) tem um elemento inverso $A^{-1}(x)$ com

grau máximo 7, de modo que $A(x) \bullet A^{-1}(x) \equiv$ '01'(mod m(x)). Podemos encontrar o inverso de A(x) usando o algoritmo da divisão de Euclides para polinômios [Bigg89]:

$$A(x) \bullet B(x) + C(x) \bullet m(x) = (\text{'01'})$$

Logo, $A(x) \bullet B(x) \bmod m(x) = (\text{'01'})$, ou seja

$$A^{-1}(x) = B(x) \bmod m(x).$$

Como a distributividade:

$$A(x) \bullet (B(x) + C(x)) = A(x) \bullet B(x) + A(x) \bullet C(x),$$

é valida, então as operações de adição, representada por +, e multiplicação, representada por • e definida como uma multiplicação polinomial modular, sobre os polinômios de grau menor ou igual a sete, definem GF(256).

Polinômios com coeficientes em GF(256):

Na estrutura do Rijndael, polinômios com coeficientes em GF(256) de grau menor ou igual a três são considerados vetores colunas de quatro bytes. Vejamos o exemplo abaixo:

$$\begin{bmatrix} \text{'05'} \\ \text{'8b'} \\ \text{'54'} \\ \text{'ca'} \end{bmatrix} = \text{'ca'}x^3 + \text{'54'}x^2 + \text{'8b'}x + \text{'05'} \qquad (2)$$

Esta estrutura tem fundamental importância na execução de uma das funções utilizadas pelo Rijndael. Vamos então definir a adição e a multiplicação sobre este tipo de polinômio.

A adição simplesmente adiciona os coeficientes de cada uma das potências através da operação de soma em GF(256). Dados dois polinômios $A(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ e $B(x) = b_3x^3 + b_2x^2 + b_1x + b_0$, a soma entre eles é:

$$C(x) = A(x) + B(x) = (a_3 + b_3)x^3 + (a_2 + b_2)x^2 + (a_1 + b_1)x + (a_0 + b_0).$$

A multiplicação requer um pouco mais de cálculos. Assim como na definição da multiplicação em GF(256), temos que usar uma redução modular. Tomando os polinômios A(x) e B(x) usados na adição, D(x) será o produto entre A(x) e B(x), denotado como:

$$D(x) = A(x) \otimes B(x) = (a_3x^3 + a_2x^2 + a_1x + a_0)(b_3x^3 + b_2x^2 + b_1x + b_0) =$$

$$= (a_3 \bullet b_3)x^6 + (a_3 \bullet b_2 + a_2 \bullet b_3)x^5 + (a_3 \bullet b_1 + a_2 \bullet b_2 + a_1 \bullet b_3)x^4 + (a_3 \bullet b_0 + a_2 \bullet b_1 + a_1 \bullet b_2 + a_0 \bullet b_3)x^3 + (a_2 \bullet b_0 + a_1 \bullet b_1 + a_0 \bullet b_2)x^2 + (a_1 \bullet b_0 + a_0 \bullet b_1)x^4 + (a_0 \bullet b_0) \quad (3)$$

Este produto fere o formato definido em (2) e D(x) deve sofrer uma redução modular através de um polinômio de grau 4, de modo que o resultado não ultrapasse grau 3. O polinômio escolhido para esta função foi $x^4 + 1$. Ele simplesmente opera sobre o expoente do polinômio reduzindo-o, pois, na aritmética módulo $(x^4 + 1)$, $x^4 = 1$. Logo, $x^k \pmod{x^4 + 1} = x^{k \bmod 4}$.

Aplicando a redução sobre o resultado obtido em (3), obtemos:

$$D(x) = (a_3 \bullet b_0 + a_2 \bullet b_1 + a_1 \bullet b_2 + a_0 \bullet b_3)x^3 + (a_3 \bullet b_3 + a_2 \bullet b_0 + a_1 \bullet b_1 + a_0 \bullet b_2)x^2 + (a_3 \bullet b_2 + a_2 \bullet b_3 + a_1 \bullet b_0 + a_0 \bullet b_1)x + (a_3 \bullet b_1 + a_2 \bullet b_2 + a_1 \bullet b_3 + a_0 \bullet b_0) \quad (4)$$

Considerando $D(x) = d_3x^3 + d_2x^2 + d_1x + d_0$, podemos representar o resultado obtido em (4) matricialmente:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Considerando:

$$d_0 = (a_3 \bullet b_1) + (a_2 \bullet b_2) + (a_1 \bullet b_3) + (a_0 \bullet b_0)$$

$$d_1 = (a_3 \bullet b_2) + (a_2 \bullet b_3) + (a_1 \bullet b_0) + (a_0 \bullet b_1)$$

$$d_2 = (a_3 \bullet b_3) + (a_2 \bullet b_0) + (a_1 \bullet b_1) + (a_0 \bullet b_2)$$

$$d_3 = (a_3 \bullet b_0) + (a_2 \bullet b_1) + (a_1 \bullet b_2) + (a_0 \bullet b_3) \quad (5)$$

Vejamos um exemplo numérico. Tomemos A(x) = (‘03’)x^3+ (‘01’)x^2 + (‘01’)x + (‘02’) e B(x) = (‘53’)x^3 + (‘fb’)x^2 + (‘4f’)x + (‘97’). Após efetuarmos os cálculos dos coeficientes em (5) chegaremos ao resultado D(x) = (‘b0’)x^3 + (‘c0’)x^2 + (‘4c’)x + (‘4c’).

O A(x) do exemplo é usado na função *MixColumn( )* durante o processo de cifragem. Veremos mais detalhes sobre esta função no próximo capítulo.

CAPÍTULO 3

Especificação do Rijndael

Neste capítulo veremos como funciona detalhadamente o processo da cifragem e da decifragem e todas as funções que compõem uma rodada do Rijndael. Por se tratar de uma cifra iterativa, definiremos a estrutura que receberá o texto claro e realizará alterações sobre o mesmo ao longo de cada uma das rodadas. Veremos também como são geradas as subchaves utilizadas na cifragem e quais são as diferenças entre o AES e o Rijndael.

3.1. Estrutura de entrada e saída e número de rodadas

Existe uma estrutura semelhante a uma matriz chamada *estado*, cujos elementos são bytes representando polinômios em GF(256). Sua função é armazenar os dados de entrada da cifra (texto claro), os resultados intermediários e a saída da cifra após a última rodada (texto cifrado).

O estado tem 4 linhas e Nb colunas. Nb é o comprimento do bloco de texto claro em bits, dividido por 32 (número de bits por coluna). O comprimento da chave em bits divido por 32 é chamado Nk. As chaves usadas nas rodadas são chamadas chaves da rodada e a sua estrutura é semelhante a do estado. O Rijndael pode ser implementado com diferentes combinações de Nk (4,5,...,8) e Nb (4,5,...,8) o que resulta em número de bits de 128, 160, 190, 222, e 256 bits tanto para o bloco de texto claro quanto para a chave. Os bytes do texto claro e da chave, durante a entrada de dados, vão sendo acrescentados coluna por coluna de cima para baixo. Logo, segundo a figura 3.1, o primeiro byte de texto claro seria o byte $a_{0,0}$. Em seguida entraria o que está abaixo e assim sucessivamente. Se tivéssemos um texto claro concatenado num bloco teríamos:

$a_{0,0}\ a_{1,0}\ a_{2,0}\ a_{3,0}\ a_{0,1}\ a_{1,1}\ a_{2,1}\ a_{3,1}\ a_{0,2}\ a_{1,2}\ a_{2,2}\ a_{3,2}\ \ldots\ a_{0,7}\ a_{1,7}\ a_{2,7}\ a_{3,7}$

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,30}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$

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

Figura 3.1: Estado com Nb=8 e Chave da rodada com Nk=4

Os valores de Nb e Nk influenciarão no número de rodadas da cifra. É de se esperar um aumento do número de rodadas proporcionalmente ao aumento do comprimento do bloco de texto claro, possibilitando uma maior iteração entre os estados intermediários e as chaves das rodadas. Esta relação entre Nb, Nk e o número de rodadas está apresentada na tabela 3.1. O número de rodadas será denotado por Nr.

	Nb				
Nk	**4**	**5**	**6**	**7**	**8**
4	10	11	12	13	14
5	11	11	12	13	14
6	12	12	12	13	14
7	13	13	13	13	14
8	14	14	14	14	14

Tabela 3.1: Número de rodadas (Nr) em função de Nb e Nk

Para o AES, o comprimento do bloco é de 128 bits (Nb=4) e o comprimento da chave pode ser de 128,196 ou 256 bits (Nk=4;6 ou 8). Esta é a única diferença entre o AES e o Rijndael, ou seja, o AES é um Rijndael particularizado.

3.2. As transformações da rodada

A cifra será descrita neste tópico como um algoritmo, representando a evolução do estado durante o processo de cifragem. Inicialmente, definiremos duas funções: a função rodada (*round*) e a função rodada final (*final round*). O algoritmo mostra entre parêntesis as variáveis passadas e recebidas pelas funções.

Cifragem:

```
Rijndael( State, CipherKey)
        {
        KeyExpansion(CipherKey,ExpandedKey);
        AddRoundKey(State,ExpandedKey[0]);
        for ( i=1 ; i < Nr ; i++ )  { Round(State, ExpandedKey[i]); }
        FinalRound(State, ExpandenKey[Nr]);
        }                                                                    (6)
```

Inicialmente recebemos do usuário a chave (*CipherKey*) e o texto claro, cuja armazenagem será feita pela variável *state* (estado). O primeiro passo efetivo do processo é a expansão da chave que irá gerar as subchaves usadas nas rodadas (*ExpandedKey[i]*). Após este passo, é feita uma adição da 1ª chave expandida, que é a própria *CipherKey* como veremos adiante, ao 1º estado, que é o próprio texto claro. O ciclo de execução da cifragem inicia então uma série de Nr-1 rodadas cujos resultados intermediários são armazenados no estado. Estes resultados são frutos das aplicações das funções da rodada sobre o estado e sua iteração com a fração da chave expandida correspondente através da função *AddRoundKey()*. A rodada é executada da seguinte forma:

```
Round( State, ExpandedKey[i])
        {
        SubByte(State);
        ShiftRow(State);
        MixColumn(State);
        AddRoundKey(State,ExpandedKey[i]);
        }
```

Cada uma dessas funções será explicada a seguir. A rodada final é levemente diferente, não tendo a função *mixcolumn*().

```
FinalRound( State, ExpandedKey[Nr])
        {
        SubByte(State);
        ShiftRow(State);
        AddRoundKey(State,ExpandedKey[Nr]);
        }
```

3.2.1. A transformação SubByte( )

Esta é a transformação responsável por executar a parte não linear da cifra e prover a confusão [Shan49] necessária para a segurança da cifra. É a caixa de substituição do

algoritmo (S-Box) atuando sobre cada um dos bytes do estado independentemente dos demais e armazenando o resultado na mesma posição como podemos ver pela figura 3.2 . Ela consiste em dois passos independentes:

1. Calcular o inverso em GF(256) do polinômio representado pelo byte x.

$$y = (x)^{-1}$$

2. Aplicar uma transformação afim sobre o resultado do passo 1.

$$z = f(k)$$

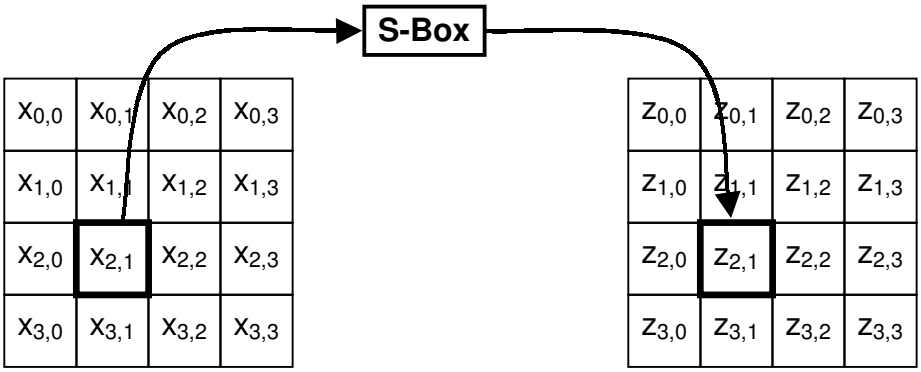

Figura 3.2: A transformação SubByte( )

Inicialmente temos que calcular o inverso de um polinômio. Como o polinômio nulo (‘00’) não tem inverso ele é mapeado sobre ele mesmo. Concluído este cálculo, aplicaremos a transformação afim que se segue, ao nível de bit (assumiremos as variáveis a seguir como participantes de uma operação matricial).

$$\begin{bmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

O byte $y = y_7y_6y_5y_4y_3y_2y_1y_0$ onde y_i são os bits posicionando o mais significativo à esquerda. Da mesma forma $z = z_7z_6z_5z_4z_3z_2z_1z_0$.

Usaremos então o algoritmo da divisão de Euclides e encontremos o MDC (máximo divisor comum) entre m(x), o polinômio sobre o qual o corpo é construído. Como m(x) é irredutível, o MDC é 1 e conseqüentemente podemos encontrar fatores da forma :

$a(x).b(x) \pmod{m(x)} \equiv 1$

$a(x).b(x) + c(x)m(x) = 1$

podemos então dizer que b(x) é o inverso multiplicativo de a(x).

Podemos tomar como exemplo a transformação do S-Box sobre o polinômio $a(x) = x^6$ (‘40’).

Então, $y(x^6) = x^4 + x^3 + x^2 + 1$ (‘1d ’)

Vem então a segunda parte, a transformação afim sobre y=(00011101)

$$\begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Encontramos então z=(00001001). Então o byte na saída da função é (‘09’). A conversão do byte(‘40’) no byte (‘09’) usando o S -Box do Rijndael parte do trabalho da função SubByte( ) que opera simultaneamente sobre todos os bytes do estado, como mostrado na figura 3.2 .

Na decifragem, a função recebe o nome de **InvSubByte( )** e consiste em realizar byte a byte a transformação afim inversa seguida do cálculo do inverso multiplicativo do resultado da primeira parte. A transformação afim inversa é:

$$\begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Uma implementação computacionalmente eficiente é através do uso de tabelas para as funções SubByte( ) e InvSubByte( ). Essas tabelas estão no apêndice A.

3.2.2. A transformação ShiftRow( )

Consiste em uma transposição simples sobre as linhas do estado e provê difusão à cifra[Shan49]. Cada uma das i linhas será deslocada ciclicamente à esquerda por C_i posições neste passo. A tabela 3.2 expressa a quantidade de posições a ser deslocada para cada uma das linhas. Um byte inicialmente na coluna j passará a ocupar a coluna j - C_i mod Nb na mesma linha. Podemos tomar como exemplo a figura 3.3.

Nb	C_0	C_1	C_2	C_3
4	0	1	2	3
5	0	1	2	3
6	0	1	2	3
7	0	1	2	4
8	0	1	3	4

Tabela 3.2: Número de posições deslocadas à esquerda pelas i linhas do estado de acordo com Nb

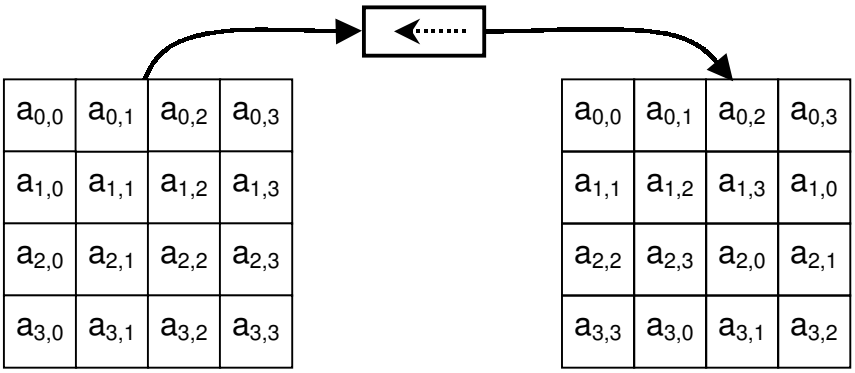

Figura 3.3: Função ShiftRow( )

A transformação inversa é executada pela função **InvShiftRow(**). Ela consiste em realizar os deslocamentos definidos pela tabela 3.2 à direita. As linhas sofrerão um deslocamento de Nb - C_i posições. Um byte na coluna j passará a ocupar a coluna j + C_i mod Nb.

3.2.3. A transformação MixColumn( )

Este passo tem a finalidade de complementar a difusão inicialmente provida pela função ShiftRow( ). Enquanto ShiftRow( ) troca a posição dos bytes pelas linhas, esta MixColumn( ) espalhará a influência de cada um dos bytes sobre os outros da mesma coluna. Consiste em realizar um produto modular de polinômios com coeficientes em GF(256) como definido no tópico correspondente na seção 2.3.3. Cada coluna do estado será tratada como um polinômio com coeficientes em GF(256), e será multiplica pelo polinômio $A(x) = (\text{‘03’})x^3 + (\text{‘01’})x^2 + (\text{‘01’})x + (\text{‘02’})$. Deste modo teremos como resultado o polinômio D(x). Tomando uma dada coluna c:

$D(x) = A(x) \otimes B(x) \pmod{x^4 - 1}$,

representada como um produto matricial:

$$\begin{bmatrix} d_{0,c} \\ d_{1,c} \\ d_{2,c} \\ d_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} b_{0,c} \\ b_{1,c} \\ b_{2,c} \\ b_{3,c} \end{bmatrix}$$

Conseqüentemente:

$d_{0,c} = (\text{‘02’}).b_{0,c} + (\text{‘03’}).b_{1,c} + (\text{‘01’}).b_{2,c} + (\text{‘01’}).b_{3,c}$

$d_{1,c} = (\text{‘01’}).b_{0,c} + (\text{‘02’}).b_{1,c} + (\text{‘03’}).b_{2,c} + (\text{‘01’}).b_{3,c}$

$d_{2,c} = (\text{‘01’}).b_{0,c} + (\text{‘01’}).b_{1,c} + (\text{‘02’}).b_{2,c} + (\text{‘03’}).b_{3,c}$

$d_{3,c} = (\text{‘03’}).b_{0,c} + (\text{‘01’}).b_{1,c} + (\text{‘01’}).b_{2,c} + (\text{‘02’}).b_{3,c}$

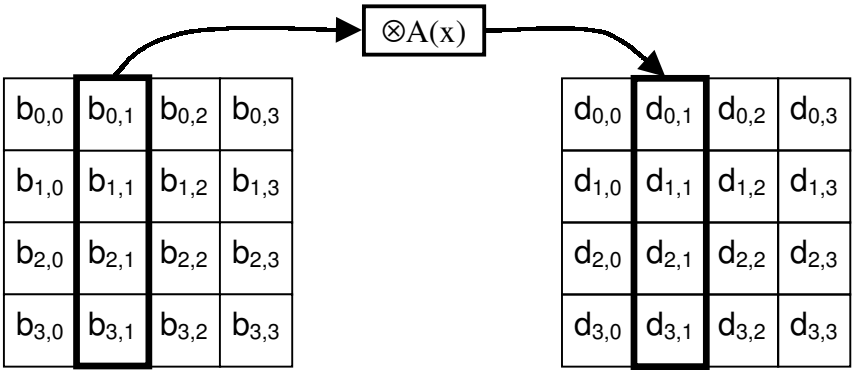

Figura 3.4: Função MixColumn( )

Tomando o exemplo da seção 2.3.3., sendo $B(x) = (\text{‘53’})x^3 + (\text{‘fb’})x^2 + (\text{‘4f’})x + (\text{‘97’})$, temos:

$$\begin{bmatrix} d_{0,c} \\ d_{1,c} \\ d_{2,c} \\ d_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} 97 \\ 4f \\ fb \\ 53 \end{bmatrix},$$

$d_{0,c} = (\text{‘02’}).(\text{‘97’}) + (\text{‘03’}).(\text{‘4f’}) + (\text{‘01’}).(\text{‘fb’}) + (\text{‘01’}).(\text{‘53’}) =$

$= (\text{‘35’}) + (\text{‘d1’}) + (\text{‘fb’}) + (\text{‘53’}) = (\text{‘4c’})$

$d_{1,c} = (\text{‘01’}).(\text{‘97’}) + (\text{‘02’}).(\text{‘4f’}) + (\text{‘03’}). \quad (\text{‘fb’}) + (\text{‘01’}).(\text{‘53’}) =$

$= (\text{‘97’}) + (\text{‘9e’}) + (\text{‘16’}) + (\text{‘53’}) = (\text{‘4c’})$

$d_{2,c}$ = (‘01’).(‘97’) + (‘01’).(‘4f’) + (‘02’).(‘fb’) + (‘03’).(‘53’) =

= (‘97’) + (‘4f’) + (‘ed’) + (‘f5’) = (‘c0’)

$d_{3,c}$ = (‘03’).(‘97’) + (‘01’).(‘4f’) + (‘01’).(‘fb’) + (‘02’).(‘53’) =

= (‘a2’) + (‘4f’) + (‘fb’) + (‘a6’) = (‘b0’)

Então, se tivéssemos um estado onde todas as colunas fossem iguais à representação do polinômio (‘53’)x^3 + (‘fb’)x^2 + (‘4f’)x + (‘97’), após a aplicação de MixColumn teríamos o resultado mostrado na figura 3.5.

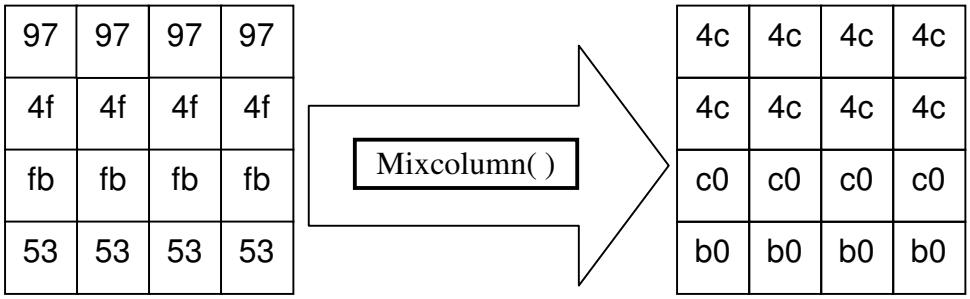

Figura 3.5: Exemplo da aplicação da Função MixColumn( ) sobre o estado

A operação inversa **InvMixColumn( )** consiste em multiplicar cada coluna do estado pelo inverso multiplicativo do polinômio A(x) módulo x^4+1. Denotaremos este polinômio por E(x). Ou seja

$$[(\text{‘03’})x^3 + (\text{‘01’})x^2 + (\text{‘01’})x + (\text{‘02’})].E(x) \equiv (\text{‘01’}) \pmod{x^4+1}$$

Com alguns cálculos, baseando-se no algoritmo de Euclides, encontramos

E(x) = (‘0b’)x^3 + (‘0d’)x^2 + (‘09’)x + (‘0e’)

Matricialmente,

$$\begin{bmatrix} b_{0,c} \\ b_{1,c} \\ b_{2,c} \\ b_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \times \begin{bmatrix} d_{0,c} \\ d_{1,c} \\ d_{2,c} \\ d_{3,c} \end{bmatrix}$$

3.2.4. A transformação AddRoundKey( )

Este é o passo de adição da chave que implementará o segredo criptográfico. É uma operação simples de adição por ou-exclusivo entre o estado e a chave da rodada correspondente. O cálculo ocorre byte a byte colocando os resultados em suas respectivas posições originais, de modo que $b_{,j} = a_{i,j} \oplus k_{i,j}$. Cada chave da rodada, denotada por ExpandedKey[i], terá seu tamanho igual ao comprimento do bloco de texto claro, independente do comprimento da chave. Como a operação de ou-exclusivo é auto inversível, a função AddRoundKey( ) tem como inversa ela mesma.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$\oplus$	$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$	=	$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$		$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$		$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$		$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$		$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$		$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

Figura 3.6: Aplicação da Função AddRoundKey( ) sobre o estado

3.3. O esquema de geração das chaves da rodada

Durante o processo de cifragem ou decifragem utilizado pelo Rijndael, o primeiro processo a ser executado é o esquema de geração das chaves através da função KeyExpansion( ). Este processo consiste basicamente em expandir a chave e definir quem são as subchaves das i rodadas chamadas ExpandedKey[i].

A explicação deste esquema será feita através de um algoritmo. Como a chave será adicionada ao estado através da função AddRoundKey( ) Nr +1 vezes, o comprimento do vetor ExpandedKey será (Nr+1) x Nb. Este vetor será denotado por W[Nb*(Nr+1)] e armazenará, em cada posição, vetores de 4 bytes que representam colunas a serem adicionadas ao estado por AddRoundKey( ). As primeiras Nk posições do vetor armazenarão a chave da cifra.

As chaves da rodada serão armazenadas em Nr +1 vetores com 4 bytes cada. As primeiras Nk posições serão ocupadas pela própria chave do usuário. O modo de expansão depende de Nk. Para Nk $\leq$ 6:

```
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 mod Nk == 0 )
                        temp = SubWord(RotWord(temp)) ⊕ Rcon[ i / Nk ];
                W[i] = W[i - Nk] ⊕ temp;
        }                                                                    (7)
}
```

A função KeyExpansion( ) recebe duas variáveis, Key contendo a chave do usuário e W que receberá a chave expandida. No início, a variável Key é carregada em W na mesma seqüência em que está disposta na chave do usuário. Em seguida, ocorrem Nb*(Nr+1) - Nk repetições onde a chave na posição i em W será o resultado da aplicação de ou-exclusivo bit-a-bit entre a chaves da posição anterior e a chave de Nk posições anteriores, exceto quando a posição é um múltiplo de Nk. Neste caso, a chave da posição anterior sofre algumas alterações antes de ser adicionada. Nesta alteração podemos notar o uso de duas funções: SubWord( ) e RotWord( ) além de um vetor Rcon[i] cujo valor ainda não foi declarado.

A primeira destas funções, SubWord( ) consiste em aplicar o S-Box do Rijndael sobre W, ou seja, os bytes que compõem uma coluna da chave expandida. A segunda, desloca os bytes da palavra ciclicamente à esquerda. Se tivermos W[j] = {‘01’,‘02’, ‘03’, ‘04’}, a aplicação da função resulta em RotWord(W[j]) = {‘02’,‘03’, ‘04’, ‘01’}.

O vetor Rcon[i] é carregado da seguinte forma:

Rcon[i] = { RC[i] , ‘00’ , ‘00’ , ‘00’ }

onde RC[i] é um elemento em GF(256) com valor $x^{(i-1)}$, de modo que:

RC[1] = 1

RC[i] = x • RC[i-1]

Logo para i = 5, por exemplo,

Rcon[5] ={ RC[i] , ‘00’ , ‘00’ , ‘00’ }= { ‘10’ , ‘00’ , ‘00’ , ‘00’ }

Quando o valor de Nk > 6, o esquema é levemente diferente:

```
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 mod Nk == 0 )
                        temp = SubWord(RotWord(temp)) ⊕ Rcon[ i / Nk ];
                else if ( i mod Nk == 4 )
                        temp = SubWord(temp);
                W[i] = W[i - Nk] ⊕ temp;
        }
}
```

A diferença em relação ao esquema para Nk $\leq$ 6 é que a chave da posição anterior sofre alterações também se o resto da divisão da posição atual com Nk for 4, onde é aplicada SubWord( ) sobre ela antes de ser adicionada a chave de Nk posições anteriores.

As chaves das rodadas são escolhidas baseando-se no tamanho do bloco de texto claro Nb. A chave da i-ésima rodada será a composição de W[i*Nr] a W[i*(Nr+1)-1]. Estas posições do vetor expandido serão dispostas como colunas para ficar no mesmo formato do estado e assim permitir a execução da função AddRoundKey( ). Na figura 3.7 vemos um exemplo de seleção das chaves da rodada para Nb = 4 e 6, os a_i são os bytes que compõem o vetor W. Exemplos de expansão de chaves do usuário são encontrados no apêndice B.

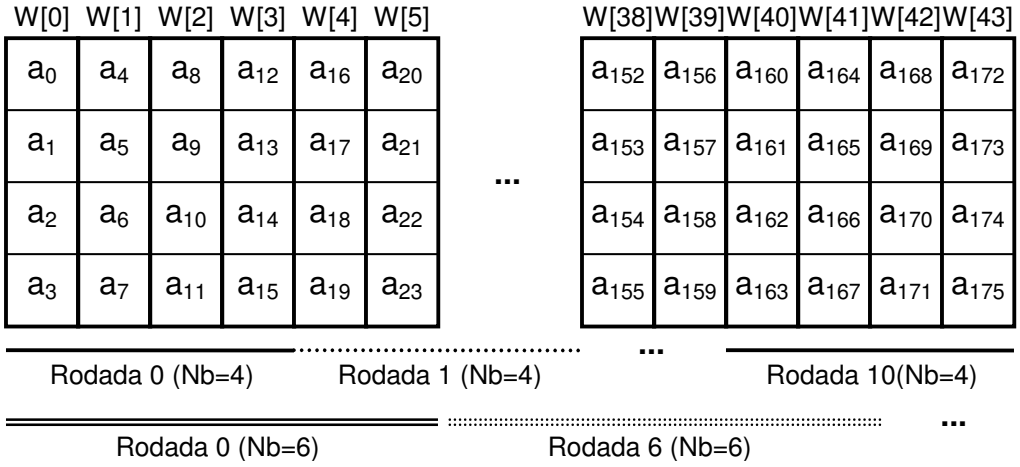

Figura 3.7: Exemplo de seleção das chaves da rodada para Nb=4 e Nb=6.

Ao final da expansão, ExpandedKey[i]=W[Nb*i] || W[Nb*i +1)] ||...|| W[Nb*(i + 1)-1], onde || representa a concatenação entre os vetores W.

3.4. A Decifragem

Este é o processo em que o Rijndael leva alguma desvantagem em relação às cifras auto-inversíveis como o IDEA [Lai92]. A decifragem do Rijndael não é igual a cifragem. Logo, em qualquer implementação, precisaremos usar funções inversas e seqüências levemente diferentes da cifragem. Nem sempre a decifragem é usada na implementação de um algoritmo criptográfico. Existem aplicações como usar o algoritmo em códigos de autenticação de mensagem (Message Autentication Code-MAC) [Schn96] onde a decifragem não é usada.

A decifragem é processo inverso da cifragem. Logo, a decifragem direta consiste em desfazer todas as aplicações das funções da rodada final e das rodadas na seqüência inversa e usando as inversas das funções. É necessário observar que antes do inicio deste processo precisaremos realizar a expansão da chave do mesmo modo da cifragem. O modelo é o que se segue:

```
Rijndael-1( State, CipherKey)
        {
        KeyExpansion(CipherKey,ExpandedKey);
        AddRoundKey(State,ExpandedKey[Nr]);
        for ( i=Nr-1 ; i > 0 ; i--)  { Round(State, ExpandenKey[i]); }
        FinalRound(State, ExpandenKey[0]);
        }

Round( State, ExpandedKey[i])
        {
        InvShiftRow(State);
        InvSubByte(State);
        AddRoundKey(State,ExpandedKey[i]);
        InvMixColumn(State);
        }

FinalRound( State, ExpandedKey[0] )
        {
        InvShiftRow(State);
        InvSubByte(State);
```

```
    AddRoundKey(State,ExpandedKey[0]);
    }
```

Esta implementação é intuitiva mas apresenta alguns aspectos negativos em relação ao desempenho. Veremos detalhes de implementação visando um melhor desempenho na seção 3.5. onde algumas alterações são propostas.

3.5. Implementações

Extrair do Rijndael o melhor desempenho possível é altamente desejável, principalmente em aplicações críticas, tais como em transmissão em redes de dados de alta velocidade. Dependendo da plataforma onde a cifra é aplicada existem aspectos da implementação que devem ser ressaltados para obter a máxima velocidade na cifragem ou decifragem.

3.5.1. Processadores de 8 bits

Como no Rijndael não existe multiplicação em GF(256) entre duas variáveis, existe uma alternativa de implementação para esta operação. Considere b o byte composto pelos bits $b_7b_6....b_1b_0$ e '02' como a representação para o polinômio x, o produto entre eles é:

$$b.x = b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \ (\text{mod } m(x)) =$$

$$= b_6x^7 + b_5x^6 + b_4x^5 + (b_3 \oplus b_7)x^4 + (b_2 \oplus b_7)x^3 + b_1x^2 + (b_0 \oplus b_7)x$$

Esta multiplicação por x é realizada pela função xtime( ). Como todos os elementos de GF(256) podem ser escritos como soma de potências de '02', podemos realizar qualquer multiplicação desejada entre uma variável polinomial e um polinômio constante.

Tomemos como exemplo o produto de b por '07':

$$b.\text{'07'} = b.(\text{'01'} \oplus \text{'02'} \oplus \text{'04'}) = b.(\text{'01'} \oplus \text{'02'} \oplus \text{'02'}^2) =$$

$$= b \oplus \text{xtime}(b) \oplus \text{xtime}(\text{xtime}(b))$$

$$= b \oplus \text{xtime}(b \oplus \text{xtime}(b))$$

Podemos então aplicar esta alteração na função MixColumn( ) aproveitando a execução apenas de instruções simples como o ou-exclusivo. Os comandos que se seguem são as transformações sofridas por uma coluna.

```
t = a[0] ⊕ a[1] ⊕ a[2] ⊕ a[3]; /* a é uma coluna do estado */
u = a[0];
v = a[0] ⊕ a[1]; v = xtime(v); a[0] = a[0] ⊕ v ⊕ t;
v = a[1] ⊕ a[2]; v = xtime(v); a[1] = a[1] ⊕ v ⊕ t;
v = a[2] ⊕ a[3]; v = xtime(v); a[2] = a[2] ⊕ v ⊕ t;
v = a[3] ⊕ u;     v = xtime(v); a[3] = a[3] ⊕ v ⊕ t;
```

A função InvMixColumn( ) que é a inversa de MixColumn( ) também sofre uma otimização para ser utilizada na decifragem. Foi notado que o polinômio usado na transformação inversa pode ser decomposto em função do polinômio usado na transformação direta.

$$E(x) = (\text{‘04’}x^2 + \text{‘05’})A(x) \pmod{x^4 + 1} \qquad (8)$$

Em notação matricial:

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} 05 & 00 & 04 & 00 \\ 00 & 05 & 00 & 04 \\ 04 & 00 & 05 & 00 \\ 00 & 04 & 00 & 05 \end{bmatrix}$$

Logo, podemos implementar a função como um simples pré-processamento seguido da aplicação de MixColumn( ). Este pré-processamento é :

```
u = xtime(xtime(a[0] ⊕ a[1])); /* a é uma coluna do estado */
v = xtime(xtime(a[1] ⊕ a[3]));
a[0] = a[0] ⊕ u;
a[1] = a[1] ⊕ v;
a[2] = a[2] ⊕ u;
a[3] = a[3] ⊕ v;
```

As outras funções não sofrem alteração de código visando otimização para processadores de 8 bits e são usadas tais quais se apresentam nas seções precedentes deste capítulo.

3.5.2. Processadores de 32 bits

As funções que compõem a rodada da cifra podem ser combinadas visando um desempenho ótimo para processadores de palavra de 32 bits ou superior.

Sendo $a_{i,j}$ o byte da i-ésima linha e j-ésima coluna na entrada da transformação da rodada e $b_{i,j}$ o byte da posição correspondente após a aplicação de SubByte( ), temos:

$$b_{i,j} = S_{BOX}[a_{i,j}],\ 0 \le i < 4;\ 0 \le j < Nb. \tag{9}$$

Sendo c um byte na entrada de MixColumn( ) e d o byte após a sua aplicação temos:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j+C_0} \\ b_{1,j+C_1} \\ b_{2,j+C_2} \\ b_{3,j+C_3} \end{bmatrix},\ 0 \le j < Nb \tag{10}$$

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix},\ 0 \le j < Nb \tag{11}$$

A adição dos índices em (10) deve ser efetuada módulo Nb. Combinando as operações de (9) a (11) obtemos:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{BOX}[a_{0,j+C_0}] \\ S_{BOX}[a_{1,j+C_1}] \\ S_{BOX}[a_{2,j+C_2}] \\ S_{BOX}[a_{3,j+C_3}] \end{bmatrix},\ 0 \le j < Nb \tag{12}$$

A multiplicação matricial pode ser vista como uma combinação linear de quatro vetores coluna:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} S_{BOX}[a_{0,j+C_0}] \oplus \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} S_{BOX}[a_{1,j+C_1}] \oplus \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} S_{BOX}[a_{2,j+C_2}] \oplus \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} S_{BOX}[a_{3,j+C_3}]$$

Pode-se então definir quatro tabelas T: T_0, T_1, T_2, T_3:

$$T_0[a] = \begin{bmatrix} 02.S_{BOX}[a] \\ 01.S_{BOX}[a] \\ 01.S_{BOX}[a] \\ 03.S_{BOX}[a] \end{bmatrix}, T_1[a] = \begin{bmatrix} 03.S_{BOX}[a] \\ 02.S_{BOX}[a] \\ 01.S_{BOX}[a] \\ 01.S_{BOX}[a] \end{bmatrix}, T_2[a] = \begin{bmatrix} 01.S_{BOX}[a] \\ 03.S_{BOX}[a] \\ 02.S_{BOX}[a] \\ 01.S_{BOX}[a] \end{bmatrix}, T_3[a] = \begin{bmatrix} 01.S_{BOX}[a] \\ 01.S_{BOX}[a] \\ 03.S_{BOX}[a] \\ 02.S_{BOX}[a] \end{bmatrix},$$

Cada uma dessas tabelas tem 256×4 bytes entradas e requerem 4kB de espaço para armazenagem. Usando-as, (12) pode ser reescrita como:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = T_0[a_{0,j+C_0}] \oplus T_1[a_{1,j+C_1}] \oplus T_2[a_{2,j+C_2}] \oplus T_3[a_{3,j+C_3}] \quad , 0 \leq \text{j} < \text{Nb} \qquad (13)$$

Levando em conta que AddRoundKey( ) pode ser implementada com um ou-exclusivo adicional de 32-bits por coluna, o custo de armazenagem é mantido no mesmo patamar. Podemos observar que as demais colunas serão apenas rotações da primeira, reduzindo o custo de armazenagem para apenas 1kB.

Na última rodada não ocorre MixColumn( ), então os bytes sofrem apenas a aplicação de S_{BOX}[ ]. Não é necessária a construção de outra tabela para o S-Box, basta executar um mascaramento sobre as tabelas T.

Na expansão da chave existem apenas aplicações de ou-exclusivo 32 bits e um deslocamento cíclico que pode ser implementado com muita eficiência.

Na decifragem existe uma forma equivalente para o algoritmo. Imaginemos um Rijndael de duas rodadas. A decifragem direta consiste na inversão da rodada final seguida pela inversão da rodada seguida por uma adição da chave, como segue a codificação abaixo:

```
AddRoundKey( State, ExpandedKey[2] );
IInvShiftRow(State);
InvSubByte(State);
AddRoundKey( State, ExpandedKey[1] );
InvMixColumn(State);
InvShiftRow(State);
InvSubByte(State);
AddRoundKey( State, ExpandedKey[0] );
```

(14)

Existem duas propriedades algébricas que serão usadas com o objetivo de obter o algoritmo equivalente:

1. A ordem de InvShiftRow( ) e InvSubByte( ) é indiferente.
2. A ordem de AddRoundKey( ) e InvMixColumn pode ser invertida se a chave da rodada for feita uma adpatação na chave, usando uma chave equivalente.

A primeira dessas propriedades é de fácil visualização pois InvShiftRow( ) apenas desloca os bytes sem alterar seu valor e InvSubByte( ) atua sobre cada um dos bytes da mesma forma sem levar em conta a posição. Logo, as duas são comutativas.

A segunda é mais sutil. Para qualquer transformação linear A: $x \rightarrow y = A(x)$, é assegurado por definição que:

$A(x \oplus k) = A(x) \oplus A(k)$.

Como AddRoundKey( ) simplesmente adiciona uma constante à sua entrada e InvMixColumn( ) é uma operação linear, a seqüência :

```
AddRoundKey( State, ExpandedKey[i] );
InvMixColumn(State);
```

Pode ser substituída por:

```
InvMixColumn(State);
AddRoundKey( State, EqExpandedKey[i] );
```

onde EqExpandedKey[i] é a aplicação de InvMixColumn( ) sobre ExpandedKey[i].

Logo, a seqüência de decifragem em (14) pode ser substituída por :

```
AddRoundKey( State, ExpandedKey[2] );
InvSubByte(State);
InvShiftRow(State);
InvMixColumn(State);
AddRoundKey( State, EqExpandedKey[1] );
InvSubByte(State);
InvShiftRow(State);
AddRoundKey( State, ExpandedKey[0] );                                    (15)
```

Estendendo a idéia para a cifra completa temos :

```
EqRound( State, ExpandedKey[i] )
{
InvSubByte(State);
InvShiftRow(State);
InvMixColumn(State);
AddRoundKey( State, EqExpandedKey[i] );
}
```

```
EqFinalRound( State, ExpandedKey[i] )
{
InvSubByte(State);
InvShiftRow(State);
AddRoundKey( State, ExpandedKey[0] );
}
```

Então Rinjdael^{-1} será:

```
InvRijndael( State, CipherKey )
{
EqKeyExpansion( CipherKey , EqCipherKey[i] );
AddRoundKey( State , EqExpandedKey[i] );
for ( i = Nr - 1 ; i > 0 ; i– – ) EqRound( State, EqExpandedKey[i] );
EqFinalRound( State , ExpandedKey[0] );
}
```

A função EqKeyExpansion( ) usada em conjunto com a decifragem equivalente é definida da seguinte forma:

1. Aplicar a expansão da chave KeyExpansion( ).
2. Aplicar InvMixColumn( ) a todas as chaves exceto à primeira e à última.

Em pseudocódigo:

```
EqKeyExpansion( CipherKey , EqExpandedKey )
{
KeyExpansion( CipherKey , EqCipherKey[i] );
for ( i = 1 ; i < Nr ; i++ )
        InvMixColumn( ExpandedKey[i] );
}
```

Esta é a forma otimizada da decifragem. É possível então utilizar tabelas T da mesma forma que foram usadas na cifragem, uma vez que a seqüência de operações da decifragem tornou-se igual à da cifragem. As tabelas T devem ser refeitas baseando-se na propriedade vista em (8).

CAPÍTULO 4

Aplicações

Duas aplicações desenvolvidas serão apresentadas neste capítulo. A primeira delas é uma implementação do Rijndael em C que ajuda a compreensão sobre a evolução da cifragem e da decifragem passo a passo. A segunda é a implementação de uma função unidirecional *hash* baseada em cifras de bloco onde o Rijndael é a função de rodada. Neste ponto será vista uma breve introdução sobre funções hash e sumários de mensagem.

4.1. Uma Implementação didática em C

Visualizar a evolução do estado do Rijndael tem um grande valor no estudo da cifra, uma vez que possibilita acompanhar todas as transformações dos bytes passo a passo. Não existia nenhuma implementação cuja função fosse esta. Partindo desta necessidade, foi desenvolvido um pequeno programa em linguagem C mostrando todas as etapas, função por função, das transformações sofridas pelo estado ao longo da cifragem e da decifragem.

A implementação foi realizada visando o AES, logo, Rijndael com Nb = 4. O programa permite a cifragem ou a decifragem com o uso de chaves de 128, 192 e 256 bits usando o modo dicionário eletrônico. Nenhuma otimização de desempenho que atrapalhasse a visualização correta dos estados intermediários partindo da descrição original da cifra foi feita. Inicialmente é escolhido o tipo de operação: cifragem ou decifragem. Em seguida se dá a entrada do texto claro e da chave da cifra. A partir deste ponto o programa vai apresentando os estados intermediários após a aplicação de cada uma das transformações. A chave da rodada é mostrada antes da aplicação de AddRoundKey( ). A evolução termina com o texto cifrado ou claro de acordo com o tipo de operação.

Existem algumas funções usadas no programa cuja descrição vem a seguir:

s_box: retorna, através da consulta da tabela da função SubByte( ) que está no apêndice A, o valor da aplicação do S-Box do Rijndael sobre um byte.

inv_s_box: retorna, através da consulta da tabela da função InvSubByte( ) que está no apêndice A, o valor da aplicação do S-Box inverso do Rijndael sobre um byte.

SubWord: executa o mesmo papel desta função no processo de expansão da chave. Aplica o S-Box sobre um conjunto de 4 bytes que são as colunas da estrutura matricial da chave. O byte que é armazenado mais acima na coluna é o mais à esquerda no vetor.

RotWord: executa o mesmo papel desta função no processo de expansão da chave. Rotaciona ciclicamente à esquerda os bytes dentro de um vetor de 4 bytes que são as colunas da estrutura matricial da chave.

conv_char_hex: converte os caracteres armazenados em duas posições de um string em dois algarismos hexadecimais referentes aos caracteres e que representam um polinômio em GF(256).

Input_Data: é a rotina de entrada de dados do programa. Pergunta se o programa será usado como cifrador ou decifrador, recebe a chave da cifra e o texto claro ou cifrado. Testa se os dados de entrada estão entre os valores possíveis de entrada. Em caso afirmativo é armazenado o primeiro estado com o texto claro ou cifrado e a chave da cifra. A passagem de referência entre funções ocorre por ponteiros visando um melhor desempenho do programa, uma vez que nenhuma otimização que altere a estrutura original da cifra foi feita.

KeyExpansion: Realiza a expansão da chave. Pega a chave da cifra (CipherKey) e armazena na chave expandida (ExpandedKey).

AddRoundKey_e: Realiza a adição da chave da rodada por ou-exclusivo como descrito no capítulo anterior. Esta é a versão a ser utilizada na cifragem por uma questão funcional do código.

AddRoundKey_d: Esta é a versão a ser utilizada na decifragem da função anterior.

print_state: imprime o estado na tela.

print_key_e: imprime a chave da rodada durante a cifragem.

print_key_d: imprime a chave da rodada durante a decifragem.

SubByte: realiza a transformação de mesmo nome sobre o estado. Aplica o S-Box sobre todos os bytes do estado. Utiliza a função s_box.

InvSubByte: realiza a transformação inversa de mesmo nome sobre o estado. Aplica o S-Box inverso sobre todos os bytes do estado. Utiliza a função inv_s_box.

ShiftRow: realiza a transformação de mesmo nome que desloca ciclicamente à esquerda as linhas do estado de acordo com a tabela 3.2.

InvShiftRow: realiza a transformação inversa da função anterior deslocando ciclicamente à direita as linhas do estado de acordo com a tabela 3.2.

mult: realiza a multiplicação polinomial em GF(256) módulo m(x) como visto no capítulo 2. A operação ocorre entre dois bytes (polinômios).

MixColumn: realiza a transformação denotada produto modular por um polinômio fixo sobre todas as colunas do estado utilizando a função mult para efetuar a multiplicação entre os bytes das colunas e do polinômio fixo.

InvMixColum: realiza a transformação inversa da função anterior multiplicando as colunas por um polinômio fixo que é o inverso multiplicativo módulo $x^4 + 1$ em relação ao usado na função direta.

Round_e: é o conjunto de transformações da rodada usado na cifragem segundo a descrição da seção 3.2. Consiste basicamente do uso de SubByte, ShiftRow, MixColumn e AddRoundKey_e.

Round_d: é o conjunto de transformações da rodada usado na decifragem segundo a descrição da seção 3.4. Consiste basicamente do uso de InvSubByte, InvShiftRow, InvMixColumn e AddRoundKey_d.

FinalRound_e: é o conjunto de transformações da rodada final da cifragem. Se assemelha a Round_e porém não faz uso de MixColumn. Ao final desta função obtemos o texto cifrado armazenado no estado.

FinalRound_d: é o conjunto de transformações da rodada final da decifragem. Se assemelha a Round_d porém não faz uso de InvMixColumn. Ao final desta função obtemos o texto claro armazenado no estado.

Como em todo código em linguagem C a função principal, main( ), realiza a chamada de todas as demais funções utilizadas pelo programa. Ela começa executando a entrada de dados através de InputData e recebe o estado, a chave da cifra, o número de colunas da chave Nk, o número de rodadas Nr e o modo de operação (cifrador/decifrador).

Em seguida é efetuada a expansão da chave através de KeyExpansion. Uma estrutura condicional é iniciada de acordo com o tipo de operação.

Se estivermos usando o programa como cifrador, é executada a adição da chave da rodada através de AddRoundKey. Em seguida iniciam-se Nr-1 rodadas usando Round_e. Ao final é executada FinalRound_e e está concluída a cifragem sendo exibido por fim o texto cifrado. Após cada transformação em uma rodada é exibido o estado, o que dá o caráter didático desta implementação.

Se estivermos no modo decifrador é executada uma seqüência de operações que se assemelha a da cifragem. Inicialmente há uma adição da chave da rodada através de AddRoundKey_d. A parte iterativa ocorre Nr -1 vezes através de Round_d. Finalmente ocorre a rodada final, FinalRound_d, cujo resultado é o texto claro.

Uma versão completa do código está no apêndice C. Foi usado o ambiente Bloodshed Dev-C++ que está sobre a licença pública da GNU para compilar o código. A versão final do programa tem 15.360 bytes de arquivo executável. O código fonte tem 465 linhas.

4.2. Uma função unidirecional hash usando o Rijndael

Relembrando o que foi afirmado na introdução, a criptografia provê, dentre outras coisas, integridade. Como ter a certeza de que uma mensagem é integra, exatamente igual a original enviada pelo emissor? O problema, do ponto de vista da criptografia, se resolve de maneira simplificada calculando um valor de comprimento em bits fixo chamado sumário de mensagem, que pode ser comparado a uma espécie de impressão digital da mensagem. Ao enviar a mensagem é possível incluir o sumário em um ponto pré-estabelecido. Caso a mensagem tenha sido violada, ao calcular o sumário não haverá uma igualdade entre o calculado e o existente na mensagem, indicando a violação.

Uma função hash unidirecional, H, opera sobre uma mensagem, denotada por pré-imagem (M), de comprimento arbitrário e retorna um sumário de mensagem h chamado de hash da mensagem, de modo que:

$h = H(M)$.

A unidirecionalidade de uma função deste tipo é assegurada pelas seguintes características:

Dado M é fácil calcular H(M).

Dado H(M) é computacionalmente inviável calcular M.

Funções hash são utilizadas em protocolos de autenticação de senhas, testes de integridade de dados, assinatura digital, códigos de autenticação de mensagens (Message Authentication Codes – MACs) dentre outras aplicações.

Uma função hash unidirecional normalmente utiliza uma função de rodada iterativa dependente da mensagem de entrada e do hash da rodada anterior. Existem vários algoritmos que possuem suas próprias funções de rodada como o MD5 [Rive92] e o SHA [FIPS-180-2].

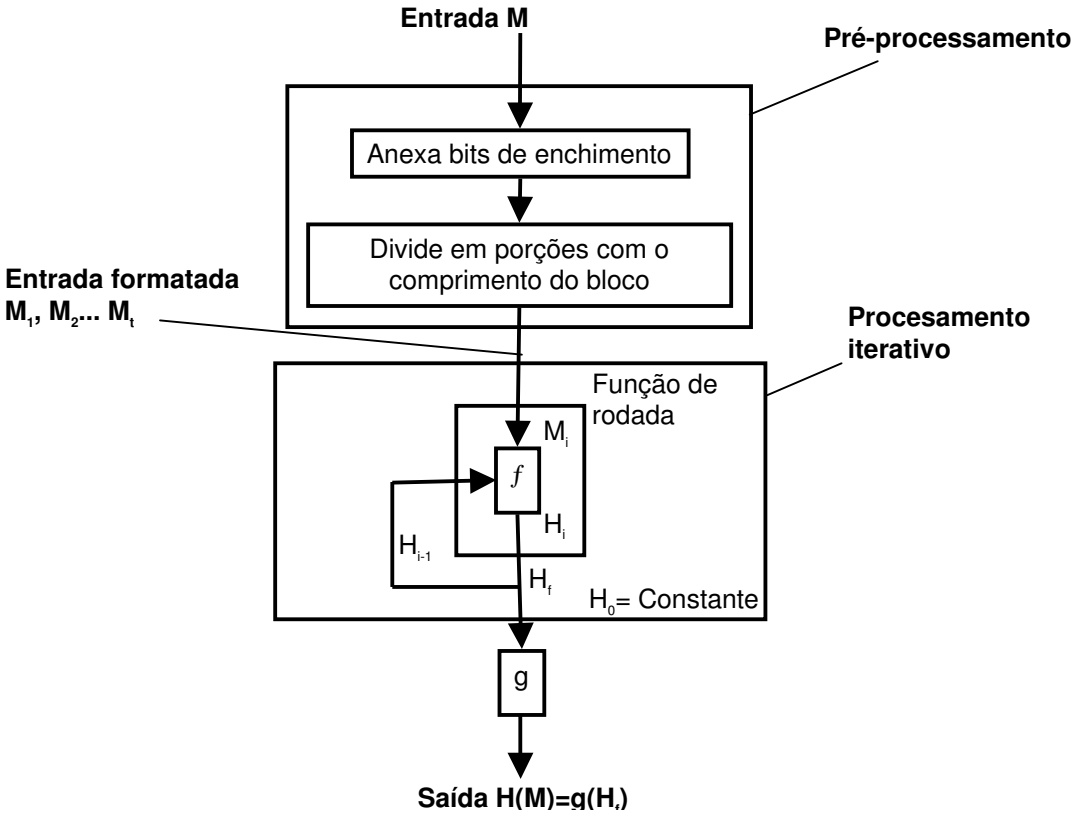

Figura 4.1: Estrutura básica de uma função hash.

A Figura 4.1 expõe a composição básica de uma função hash. Antes de começar o processamento propriamente dito sobre a mensagem, são anexados bits ao final dela, de modo que ela tenha como comprimento final um número de bits múltiplo da quantidade de bits usada na operação de f, a função de rodada. Em seguida, a mensagem é dividida em blocos a serem operados por f formando i+1 mensagens, M_0 ... M_i. O processamento da mensagem ocorre de forma iterativa. O hash calculado é fruto de uma operação entre a mensagem da i-ésima rodada e o hash da rodada anterior. É comum adotar uma constante como valor do hash inicial H_0. Ao final do processamento, o hash final H_f é submetido a uma transformação final g. Denotamos por H(M), ou simplesmente h, a saída do processo.

Existem duas categorias principais de funções hash: chaveadas e não chaveadas. As chaveadas utilizam uma chave que restringe a verificação de integridade da mensagem apenas aos usuários autorizados. As não chaveadas dividem-se em unidirecionais e resistentes a colisão. De acordo com a aplicação, podemos avaliá-las pela dificuldade ou resistência à colisão (encontrar duas mensagens com o mesmo sumário), à pré-imagem (não encontrarmos nenhuma mensagem para determinado sumário) ou à segunda pré-imagem (para uma dada mensagem, não exista outra cujo sumário seja igual ao dela). As funções unidirecionais são resistentes a ambos os tipos de pré-imagem enquanto as resistentes à colisão já expressam sua funcionalidade pelo nome e além dessa resistência são também à segunda pré-imagem, podendo ser resistentes a pré-imagem.

Existe uma classe de funções hash baseadas em cifras de bloco. A cifra executa o papel da função de rodada f. Existem esquemas combinando o uso de M_i, H_{i-1}, $M_i \oplus H_{i-1}$ e constantes nas posições A, B e C de acordo com a figura 4.2.

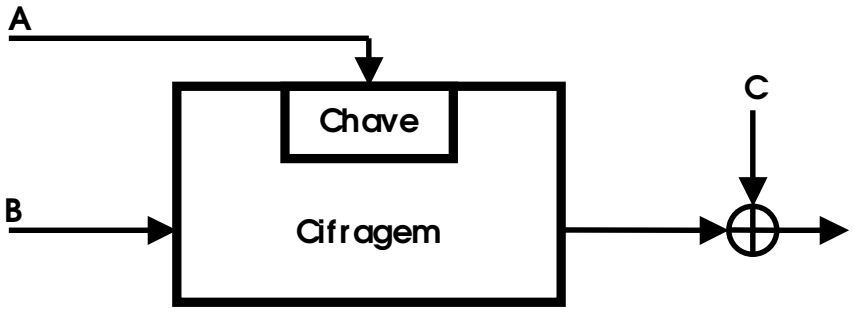

Figura 4.2: Esquema de uma função de rodada para geração de hash baseado em cifras de bloco

Três desses esquemas não tem nenhum ataque conhecido melhor do que a busca exaustiva. São eles Davies-Meyer ($A = M_i$, $B = H_{i-1}$, $C = H_{i-1}$) [Wint84], Matyas-Meyer-Oseas ($A = H_{i-1}$, $B = M_i$,$C = M_i$) [MMO85] e Miyaguchi-Preneel ($A = H_{i-1}$, $B = M_i$, $C = H_{i-1} \oplus M_i$) [Pren93]. A figura 4.3 mostra como funcionam os esquemas.

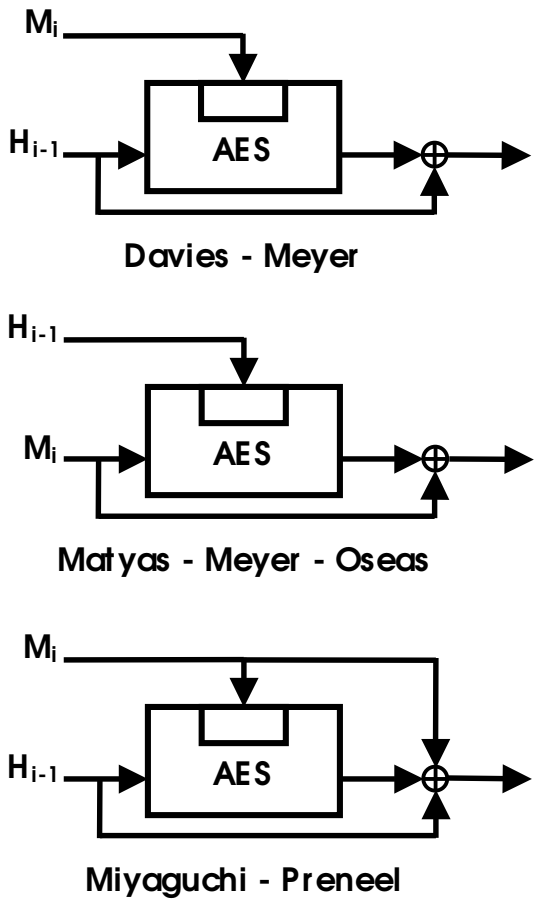

Figura 4.3: Esquemas considerados seguros usando o AES como a cifra de bloco

Dos esquemas acima, o Davies-Meyer apresenta uma vantagem do ponto de vista do desempenho pois permite a realização da expansão da chave em paralelo com a cifragem. A chave, que passará pelo processo de expansão, não depende do hash da rodada anterior e

sim do bloco proveniente da mensagem de entrada. A expansão pode ser executada a qualquer momento.

Existe uma grandeza chamada taxa de hash para funções baseadas em cifras de bloco, que mede o número de n-bits blocos de mensagem, onde n é o comprimento do bloco do algoritmo processado por cifragem. Quanto maior a taxa, mais veloz o algoritmo. Os esquemas Matyas-Meyer-Oseas e Miyaguchi-Preneel tem taxa de hash unitária, enquanto o Davies-Meyer tem taxa k/n, onde k é o comprimento da chave e n é o comprimento do bloco da cifra.

Pelos motivos expostos, foi usado o esquema de Davies-Meyer para implementação de uma função unidirecional hash usando o Rijndael. É importante lembrar que uma função hash unidirecional é resistente apenas às duas variações de pré-imagem enquanto a função hash mais segura é aquela resistente à colisão. As versões do Rijndael onde k = n são as de melhor desempenho.

A aplicação desse tipo de função é usual quando já temos um módulo pronto que faz a cifragem como visto na figura 4.3. Este tipo de hash dispensa o projeto de outro módulo com a finalidade de prover integridade.

Podemos explicar por analogia os dois tipos básicos de ataque por bruta força. Imaginemos um auditório repleto de pessoas. Qual a quantidade mínima de pessoas para que a probabilidade de ao menos uma delas compartilhar a mesma data de aniversário (dia e mês) que a sua seja 50% ? Ou seja, Quantas pessoas são necessárias na sala para que se torne provável achar um aniversariante igual a você ?

Proposição 1:

Ao tirarmos uma quantidade r de unidades de um conjunto com um total de N unidades com reposição, a probabilidade de um dado elemento ser tirado converge para:

$$p = 1 - \exp\left(-\frac{r}{N}\right)$$

Então, ao assumir p = 0,5 e N = 365, calcula-se valor de r. Logo, para uma sala com 253 pessoas é provável que encontremos alguém cuja data de aniversário seja igual a sua.

Vendo a situação por outro ângulo, qual a quantidade de pessoas necessárias na sala para que se torne provável encontrarmos duas pessoas com o mesmo aniversário (mês e ano) ?

Proposição 2:

Ao tirarmos uma quantidade r de unidades de um conjunto com um total de N unidades com reposição, a probabilidade de que exista ao menos uma coincidência é dada por:

$$p = 1 - \exp\left(-\frac{r^2}{2N}\right)$$

Então, assumindo p = 0,5 e N = 365, calculamos o valor de r. Logo, para uma sala com apenas 23 pessoas é provável que encontremos ao menos duas pessoas cujas datas de aniversário sejam iguais. Isto ocorre porque podemos formar 253 pares diferentes (23!/21!2!). Esta probabilidade da proposição 2 é a probabilidade de ocorrência de uma colisão.

Conseqüentemente, se temos uma função hash que calcula um sumário de comprimento m-bits, encontrar uma mensagem cujo hash é um determinado valor deve requerer 2^m mensagens aleatórias. Se for desejado encontrar duas mensagens que tenham o mesmo sumário você deve ter em mãos $2^{m/2}$ mensagens aleatórias, número significantemente menor em relação ao primeiro.

As implementações de melhor desempenho são as do Rijndael-Hash (RH) de 128 e 256 bits de comprimento do bloco e da chave – RH(128,128) e RH(256,256). Existe uma recomendação dos autores do Rijndael de que seja usado o RH(128,128) para aplicações onde for necessária resistência aos ataques de pré-imagem. Quando o objetivo for a resistência a ataques de colisão, deve ser usado o RH(256,256).

Para o primeiro, podemos calcular que a quantidade de mensagens para que se torne provável encontramos uma colisão é de aproximadamente 10^{19} para RH(128,128). Já esta quantidade para o RH(256,256) é bastante superior, aproximadamente 10^{38}. Para o RH(128,128) este número não é suficientemente alto e inviabiliza um ataque de pré-

imagem por força bruta. Como cada aplicação da função de rodada que é o Rijndael é uma cifragem e assumindo que a implementação consegue executar uma cifragem completa em 1290 ciclos considerando adicionados de mais 4 ciclos para executar uma operação de ou-exclusivo. Ao usar um processador veloz de 3 GHz de 64 bits o tempo necessário para realizar uma cifragem é de $4{,}3.10^{-7}$ s. Logo, um ataque de força bruta visando uma colisão seria provável, a partir de uma mensagem de apenas 1 bloco (16 bytes para o RH(128,128)) após o cálculo de 10^{19} sumários, usando um conjunto de 10^4 computadores trabalhando em paralelo, após aproximadamente 166 meses ou 13,8 anos. Ainda assim, o Rijndael não foi considerado seguro contra este tipo de ataque.

Segue uma tabela comparativa entre os desempenhos de funções hash baseadas nas cifras de bloco mais conhecidas sobre um processador da classe Pentium.

Cifra	Comprimento do hash (bits)	Velocidade (ciclos)	Quantidade de textos para uma colisão
RH(128,128)	128	1294	10^{19}
RH(256,256)	256	2110	10^{38}
DES	64	458	5.10^{9}
Serpent	128	2534	10^{19}
RC6	128	1964	10^{19}
TwoFish	128	9004	10^{19}
MARS	128	4794	10^{19}

Tabela 4.1: Comparativo entre funções hash baseadas em cifras de bloco

CAPÍTULO 5

Criptoanálise

Criptoanálise é a ciência que estuda métodos para quebrar cifras. Uma cifra é quebrável se é possível determinar o texto claro ou a chave a partir do texto cifrado, ou de pares de texto claro-texto cifrado. Existem 3 métodos básicos de ataque: somente texto cifrado, texto claro conhecido e texto claro escolhido.

Ataques apenas de texto cifrado são difíceis de implementar pois se baseiam apenas no conhecimento do texto cifrado interceptado, que apesar de ter sofrido a cifragem, revela alguns padrões provenientes da linguagem, do assunto sobre o qual está tratando a mensagem e palavras prováveis de serem encontradas no contexto.

Ataques com texto claro conhecido ocorrem a partir do conhecimento de alguns pares de texto claro-texto cifrado pelo criptoanalista. Quanto maior a quantidade de pares interceptados, maior a possibilidade de encontrar padrões propagados do texto claro para o texto cifrado.

Ataques de texto claro escolhido são os mais favoráveis à criptoanálise e dependem de ter em mãos pares de texto claro-texto cifrado, onde o texto claro foi deliberadamente escolhido pelo criptoanalista. Os ataques implementados atualmente contra cifras modernas baseiam-se neste último tipo de ataque.

Um dos critérios de submissão de cifras à seleção do AES foi a resistência contra ataques conhecidos. Baseando-se nessa premissa os criadores do Rijndael projetaram-na resistente contra estes ataques. Os principais ataques algébricos de criptoanálise surgidos no início da década de 90 foram de criptoanálise diferencial [BiSh91] e linear[Mats94] . Estes ataque foram projetados visando atacar a estrutura do DES.

O ataque de criptoanálise diferencial tem como princípio para a descoberta de partes da chave a alta probabilidade de ocorrência de diferenças entre os bytes de texto claro e de diferenças no texto cifrado. Já o ataque de criptoanálise linear consiste em explorar a

ocorrência de expressões lineares de alta probabilidade relacionando os bits de entrada e de saída da cifra.

5.1. A Estratégia do trilho largo

É uma estrutura usada para projetar transformações da rodada de cifras de bloco com chaves alternantes que combina eficiência e resistência contra criptoanálise diferencial e linear.

Basicamente a transformação da rodada é composta de dois passos inversíveis:

1. γ . Uma transformação local não linear. Local significa que qualquer bit da saída depende somente de um número limitado de bits da entrada e que a vizinhança dos bits de saída depende da vizinhança dos bits de entrada.

2. λ . Uma mistura linear provendo alta difusão.

Logo, a transformação da rodada ρ é denotada por:

$$\rho = \gamma \circ \lambda$$

e será referida como a transformação de rodada $\gamma\lambda$.

A Estrutura de Rodada $\gamma\lambda$ em Cifras de Bloco:

No projeto de cifras de bloco γ é uma permutação *camada de tijolos* consistindo de uma certa quantidade de S-Boxes. Uma função *camada de tijolos* é uma função que pode ser decomposta em um número de funções booleanas operando independentemente sobre subconjuntos de bits do vetor de entrada. Deste modo:

$\gamma : b = \gamma(a) \Leftrightarrow b_i = S_\gamma(a_i)$,

onde S_γ é um caixa de substituição de m bits inversível.

O passo λ combina pacotes linearmente: cada pacote tem na sua saída o resultado de uma aplicação de uma função linear sobre pacotes obtidos na sua entrada. λ pode ser especificado ao nível de bit por uma matriz nb $\times$ nb M. De modo que:

$\lambda : b = \lambda(a) \Leftrightarrow b = Ma$.

Visando atingir um alto nível de difusão, sem a implementação de grandes S-Boxes, a transformação λ pode ser construída em dois passos:

1. θ. Um passo que provê alta difusão local
2. π. Um passo que provê alta dispersão.

Em projetos de cifras de bloco θ, é uma permutação camada de tijolos. Os componentes de cada permutação operam sobre um número limitado de pacotes. O passo π preocupa-se com a dispersão. Entende-se por dispersão a operação pela qual os bits que estão próximos uns dos outros no contexto de θ são movidos para posições distantes.

A Estrutura da Rodada do Rijndael:

A aplicação da estratégia do trilho largo ao Rijndael assegura sua resistência contra ataques de criptoanálise diferencial ou linear e foi desenvolvido como segue:

1. SubByte: O estágio não linear γ, operando sobre o estado em paralelo.

2. ShiftRow: O estágio de transposição π.

3. MixColumn: O estágio da mistura θ, operando sobre colunas de quatro bytes cada.

Os coeficientes de MixColumn foram escolhidos para que, em conjunto com ShiftRow, consigam atingir a máxima difusão.

É possível obter valores numéricos como indicativo da resistência do Rijndael consultando o capítulo 5 de [Daem95], que é um apêndice de [DaRi99], ou os capítulos de 7 a 9 de [DaRi02].

5.2. Diferenciais truncados

Este conceito foi descrito por Lars Knudsen em [Knud95]. Estes ataques baseiam-se na tendência de algumas cifras ao agrupamento dos trilhos diferenciais [Daem95, Cap. 5]. Este agrupamento desperta a atenção quando para certos conjuntos de padrões de diferenças de entrada e saída, o número de trilhos diferenciais é excessivamente alto. A probabilidade

de que um trilho diferencial fique dentro dos limites do agrupamento pode ser calculada independentemente das probabilidades dos demais trilhos diferenciais. Cifras nas quais todos os passos operam sobre um estado em camadas tendem a ser suscetíveis a este tipo de ataque. Como este é o caso do Rijndael, com todos os passos operando sobre bytes mais especificamente do que sobre bits, foi investigada pelos autores do Rijndael a resistência da cifra contra os ataques por diferenciais truncados. Para seis rodadas ou mais, não foram encontrados ataques mais rápidos do que por busca exaustiva.

5.3. Ataque de saturação

Este tipo de ataque foi proposto por Daemen, Knudsen e Rijmen sobre a cifra Square [DKR97], a predecessora do Rijndael. Este ataque é também conhecido como ataque Square e explora a estrutura orientada a byte do Square, sendo aplicável a versões reduzidas do Rijndael. S. Lucks propôs o nome de "ataque de saturação" [Luck01] enquanto A. Biryukov e A. Shamir o chamaram "ataque estrutural" [BiSh01] mais recentemente. Uma melhoria foi proposta por N. Ferguson e equipe em [FKSS01], reduzindo o fator de trabalho do ataque.

O ataque de saturação é um ataque de texto claro escolhido sobre cifras com a estrutura de rodada usada no Rijndael. Aplicado a ele, o ataque é mais rápido que a busca exaustiva para versões reduzidas de até seis rodadas da cifra. Será mostrada a versão básica do ataque para a cifra com quatro rodadas e a estendermos para cinco e seis rodadas.

5.3.1. Premissas

Dado um conjunto Λ de 256 estados onde todos são diferentes em alguns dos bytes do estado (bytes ativos) e iguais em todos os outros bytes do estado (bytes passivos), temos:

$$\forall x, y \in \Lambda : \begin{cases} x_{i,j} \neq y_{i,j} & se\ (i = j)\ ativo \\ x_{i,j} = y_{i,j} & caso\ contrário \end{cases}.$$

Assim todos os bytes são constantes ou variam assumindo todos os valores possíveis (de 00 a FF), temos:

$$\bigoplus_{x \in \Lambda} x_{i,j} = 0, \ \forall\, i, j\ .$$

Aplicações das transformações SubByte e AddRoundKey resultam num outro conjunto Λ, porém, as posições dos bytes ativos são mantidas. A aplicação de ShiftRow resulta num novo conjunto Λ onde os bytes ativos são deslocados ciclicamente nas linhas dos estados pela função ShiftRow( ). A aplicação de MixColumn a um conjunto Λ não necessariamente resulta num novo conjunto Λ. Apesar disto, como cada byte de saída de MixColumn é uma combinação linear com coeficientes inversíveis de quatro bytes de entrada na mesma coluna do estado, uma coluna com apenas um byte ativo resultará, após a aplicação de MixColumn, em uma coluna com quatro bytes ativos. Deste modo, a saída desta transformação continuará sendo um conjunto Λ desde que na entrada exista no máximo um byte ativo por coluna.

5.3.2. O Ataque básico

Tomemos um conjunto Λ no qual apenas 1 byte está ativo. Vamos observar a evolução das posições dos bytes ativos através de 3 rodadas. Inicialmente o conjunto sofre uma aplicação de AddRoundKey antes da primeira rodada que não altera a posição do byte ativo. Na primeira rodada SubByte não altera a posição do byte ativo e ShiftRow apenas desloca-o mas não altera a quantidade de bytes ativos. Quando vem a aplicação de MixColumn ocorre uma ativação de toda coluna onde se encontrava o byte ativo na entrada da transformação. Na seqüência vem AddRoundKey da primeira rodada e SubByte da segunda que não alteram as posições dos bytes ativos. Na aplicação de ShiftRow na segunda rodada a coluna de bytes ativos é espalhada em diferentes colunas. A subseqüente aplicação de MixColumn ativa as quatro colunas onde estavam os bytes ativos. O conjunto permanecerá sendo um conjunto Λ até a nova aplicação de MixColumn na terceira rodada. A figura 5.1 demonstra esta evolução num estado com Nb = 4.

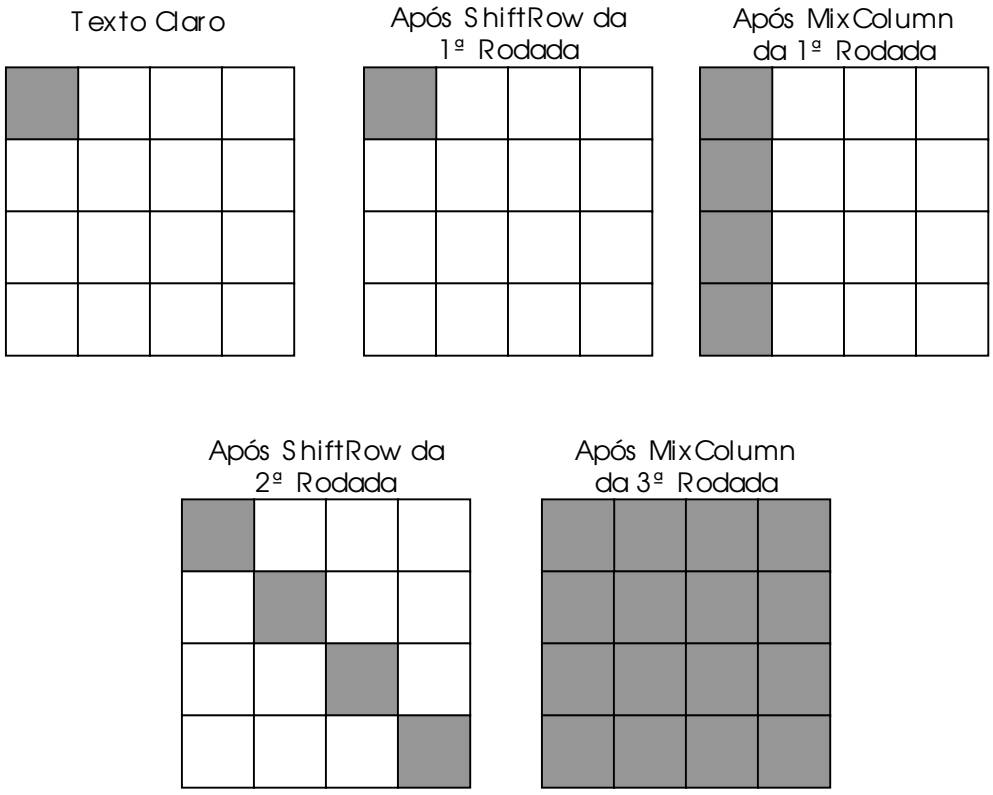

Figura 5.1: Evolução das posições dos bytes ativos num conjunto Λ com Nb = 4

Denotando as entradas de MixColumn da terceira rodada por b^{λ} , temos para todos os valores de i, j que:

$$\begin{aligned}
\bigoplus_{\lambda} b^{\lambda}_{i,j} &= \bigoplus_{\lambda} MixColumn\,(a^{\lambda}_{i,j}) \\
&= \bigoplus_{\lambda} (02.a^{\lambda}_{i,j} \oplus 03.a^{\lambda}_{i+1,j} \oplus a^{\lambda}_{i+2,j} \oplus a^{\lambda}_{i+3,j} \\
&= 02.\bigoplus_{\lambda} a^{\lambda}_{i,j} \oplus 03.\bigoplus_{\lambda} a^{\lambda}_{i+1,j} \oplus \bigoplus_{\lambda} a^{\lambda}_{i+2,j} \oplus \bigoplus_{\lambda} a^{\lambda}_{i+3,j} \\
&= 0 \oplus 0 \oplus 0 \oplus 0 = 0
\end{aligned}$$

Conseqüentemente, todos os bytes na entrada da quarta rodada tem soma nula. Esta propriedade é geralmente destruída após a subseqüente aplicação de SubByte.

Assumindo a quarta rodada como a rodada final, a operação MixColumn não ocorrerá. Cada byte na saída da quarta rodada depende exclusivamente de apenas um byte

na entrada da rodada. Denotando a entrada da quarta rodada por **c** , a saída por **d** e a chave da quarta rodada por k, temos:

$$\mathbf{d} = \text{AddRoundKey(ShiftRow(SubByte(c)),k)}$$

$$d_{i,j} = S_{Box}[c_{i,j+C_i}] \oplus k_{i,j} , \ \forall\ i , j$$

$$c_{i,j} = S_{Box}^{-1}[d_{i,j-C_i} \oplus k_{i,j-C_i}], \ \forall\ i , j \qquad (15)$$

onde as operações sobre os índices das colunas são efetuadas módulo Nb. Usando a última expressão acima, o valor de $c_{i,j}$ pode ser calculado do texto cifrado para todos os elementos do conjunto Λ a partir de valores assumidos para os bytes da chave $k_{i,j-C_i}$. Se o valor assumido para $k_{i,j-C_i}$ é igual ao valor correto do byte da chave da rodada, a seguinte equação deve ser satisfeita:

$$\bigoplus_{\lambda} c_{i,j}^{\lambda} = 0 , \ \forall\ i, j . \qquad (16)$$

Se (15) não for satisfeita, o valor assumido para o byte da chave deve estar errado. Com este procedimento é esperado encontrar-se um byte da chave. Podemos repeti-lo com os demais bytes da chave. Uma vez que verificar (16) para um único conjunto Λ deixa apenas 1/256 das chaves erradas assumidas como possíveis candidatas, a chave da cifra pode ser encontrada com alta probabilidade usando apenas 2 conjuntos Λ. O fator de trabalho do ataque é determinado pelo processamento do primeiro conjunto de 2^8 textos claros. Para todos os valores possíveis de um byte da chave da rodada, de 00 a FF, a equação (16) deve ser testada. Isto significa 2^{16} operações de ou-exclusivo e consultas ao S-Box correspondendo a 2^{10} execuções da cifra de 4 rodadas. Um insignificante número de valores possíveis para o byte da chave da rodada serão verificados usando o segundo conjunto Λ. Com o objetivo de recuperar completamente a chave de rodada, o ataque deve ser repetido 16 vezes, levando a complexidade total do ataque para 2^{14} execuções da cifra.

5.3.3. Influência da rodada final

O fato de MixColumn não estar presente na rodada final não enfraquece a cifra contra ataques de saturação. Será mostrado a seguir que a adição da transformação MixColumn ao final da cifra não aumenta a sua resistência. Dada a entrada da quarta

rodada ainda denotada por **c**, e a saída de uma quarta rodada completa (incluindo MixColumn) denotada por **e,** temos:

e = AddRoundKey(MixColumn(ShiftRow(SubByte(c))),k)

$$e_{i,j} = 02.S_{Box}[c_{i,j+C_i}] \oplus 03.S_{Box}[c_{i+1,j+C_{i+1}}] \oplus S_{Box}[c_{i+2,j+C_{i+2}}] \oplus S_{Box}[c_{i+3,j+C_{i+3}}] \oplus k_{i,j}, \forall i, j \quad (17)$$

Existem 4.Nb equações (16) : uma para cada valor de i , j . As equações podem ser resolvidas para os bytes de c, por exemplo, para $c_{0,0}$:

$$c_{0,0} = S_{Box}^{-1}[0E.(e_{0,0} \oplus k_{0,0}) \oplus 0B.(e_{1,-C_1} \oplus k_{1,-C_1}) \oplus 0D(e_{2,-C_2} \oplus k_{2,-C_2}) \oplus 09.(e_{3,-C_3} \oplus k_{3,-C_3})]$$

$$c_{0,0} = S_{Box}^{-1}[0E.e_{0,0} \oplus 0B.e_{1,-C_1} \oplus 0D.e_{2,-C_2} \oplus 09.e_{3,-C_3} \oplus k'_{0,0}]$$

onde a chave equivalente k' é definida como

k' = InvMixColumn(InvShiftRow(k)). (18)

Equações similares são obtidas para os demais bytes de **c**. O valor de $c_{0,0}$ em todos os elementos do conjunto Λ podem ser calculados a partir do texto cifrado assumindo um valor para um byte da chave equivalente k'. O mesmo ataque visto em 5.3.2 pode ser executado com o objetivo de recuperar os bytes da chave equivalente k'. Quando obtidos todos os bytes de k' é usada a equação (18) para calcular os bytes de k.

Concluímos então que a remoção de MixColumn da rodada final não enfraquece o Rijndael com respeito ao ataque de saturação em 4 rodadas.

5.3.4. Extensão no fim

Se uma rodada é adicionada, é necessário calcular o valor de $c_{i,j+C_i}$, partindo de (15) e usando a saída da quinta rodada ao invés da quarta. Isto pode ser feito assumindo o valor para um conjunto de 4 bytes da quinta chave da rodada. Como no caso do ataque de quatro rodadas, as chaves erradas serão descartadas usando a condição de teste (16).

Neste ataque de cinco rodadas, 2^{40} valores de chaves serão testados, e este processo deve ser repetido quatro vezes. A chave da cifra pode ser encontrada com alta probabilidade usando cinco conjuntos Λ. A complexidade pode ser estimada em quatro

execuções × 2^{40} possíveis valores para cinco bytes de chave × 2^8 textos cifrados × 5 consultas ao S-Box, ou 2^{46} execuções da cifra de 5 rodadas.

5.3.5. Extensão no começo

A idéia básica desta extensão é trabalhar com conjuntos de textos claros que resultem em um conjuntos Λ com apenas um byte ativo na saída da primeira rodada.

Considerando um conjunto de 2^{32} textos claros, cada coluna de bytes na entrada de MixColumn na primeira rodada assume todos os valores possíveis e todos os outros bytes permanecem constantes. Como MixColumn e AddRoundKey são inversíveis e trabalham independentemente sobre as quatro colunas esta propriedade será conservada: na saída da primeira rodada os estados terão valores constantes para três colunas, e o valor da quarta coluna assumirá todos os 2^{32} valores possíveis. Este conjunto de 2^{32} textos claros pode ser considerado como uma união de 2^{24} conjuntos Λ, onde cada um deles possui apenas um byte ativo na saída da primeira rodada. É impossível separar os textos claros dos diferentes conjuntos Λ, mas desde que a condição (16) seja satisfeita para cada conjunto Λ individualmente, deve satisfazer também quando somarmos os 2^{32} valores. Conseqüentemente, a chave da rodada final pode ser recuperada byte a byte, da mesma forma que no ataque para cifra de quatro rodadas. Este ataque de cinco rodadas requer duas estruturas de 2^{32} textos claros escolhidos. O fator de trabalho do ataque pode ser estimado em 16 execuções × 2^{32} textos cifrados no conjunto × 2^8 valores possíveis para cada byte da chave × uma consulta ao S-Box por teste, ou 2^{38} execuções da cifra de quatro rodadas.

5.3.6. Ataque sobre 6 rodadas

Combinando os dois tipos de ataque com extensão no começo e no fim temos o ataque de 6 rodadas. O fator de trabalho pode ser estimado para quatro execuções × 2^{32} textos em um conjunto × 240 valores possíveis para 5 bytes da chave × 5 consultas ao S-box por teste, ou 2^{70} execuções da cifra de seis rodadas. Existe uma apresentação mais

eficiente para este ataque proposta em [FKSS01] onde o fator de trabalho pode ser reduzido para 2^{46} execuções da cifra através do uso da técnica de somas parciais. A tabela 5.1 expressa a quantidade de memória necessária e o fator de trabalho dos ataques. S. Lucks observa que para o Rijndael com comprimento de chave de 192 e 256 bits, o ataque de seis rodadas pode ser estendido em uma rodada a mais estimando uma chave da rodada adicional [Luck00].

Tipo de Ataque	Número de textos claros	Número de execuções da cifra	Memória (bits)
Básico (4 rodadas)	2^9	2^{14}	pequena
Extensão no fim	2^{11}	2^{46}	pequena
Extensão no começo	2^{33}	2^{38}	2^{32}
Ambas as extensões	2^{35}	2^{46}	2^{32}

Tabela 5.1: Comparativo entre os tipo de ataque de saturação.

5.3.7. Ataque dos rebanhos

Um rebanho é um conjunto de 2^{120} textos cifrados onde o quinto byte após a aplicação de MixColumn na primeira rodada é fixo como descrito em [FKSS01]. Um ataque de sete rodadas necessita de $2^{128} – 2^{119}$ textos claros escolhidos e 2^{64} bits de memória. O fator de trabalho é comparável a 2^{120} cifragens. Já o ataque com oito rodadas da cifra requer a mesma quantidade de textos claros escolhidos usando 2^{104} bits de memória. O fator de trabalho é excessivamente alto para o Rijndael de 128 bits de chave. Para a cifra com 192 bits de chave o fator de trabalho é de 2^{188} cifragens enquanto para a versão de 256 bits de chave tem um fator de 2^{204} cifragens.

5.4. Ataque de Gilbert-Minier

Este ataque foi publicado por H. Gilbert e M. Minier na 3ª Conferência de Candidatas a AES [GiMi00] com o nome de ataque de colisão. O melhor ataque existente à

época da publicação deste ataque era o ataque de saturação básico de seis rodadas. Este ataque propõe-se a quebrar uma versão reduzida de sete rodadas do Rijndael. O ataque de saturação básico é baseado no fato que as três primeiras rodadas da cifra podem ser diferenciadas a partir de uma permutação aleatória. Gilbert e Minier desenvolveram o ataque de colisão a partir de uma estrutura de distinção baseada em quatro rodadas da cifra, que permite um ataque de até sete rodadas. Devido ao fator de trabalho o ataque só é melhor do que a busca exaustiva para determinados comprimentos da chave.

5.4.1. A estrutura de distinção de 4 rodadas

Denotando por *a* a entrada da primeira rodada, *b* a entrada da segunda, *c* a entrada da terceira, *d* a entrada da quarta e *e* a saída da quarta, é assumindo que $R4_k$ denota a cifragem executada por uma versão reduzida do Rijndael de quatro rodadas, sobre a desconhecida chave k. Foi investigado o comportamento de uma família de funções $f_{uvw,k}$ que são parametrizadas por três bytes u,v,w e a chave k. As funções operam sobre o valor do byte x e resultam no valor do byte y como saída das funções. As funções $f_{uvw,k}$ são definidas como:

$$f_{uvw,k}(x) = y \Leftrightarrow \begin{cases} a_{0,0} = x, a_{1,0} = u, a_{2,0} = v, a_{3,0} = w, \\ os\ outros\ a_{i,j}\ são\ desconhecidos, mas\ constantes, \\ R4_k(a) = e, \\ \quad y = 0Ee_{0,0} \oplus 0Be_{1,0} \oplus 0De_{2,0} \oplus 09e_{3,0} \end{cases}.$$

Pode ser mostrado que a herança da estrutura da transformação da rodada impõe restrições sobre a família de funções $f_{uvw,k}$: se 2^{16} valores diferentes para os parâmetros u, v e w são selecionados, com alta probabilidade ao menos dois conjuntos de parâmetros resultarão na mesma função f. Esta propriedade vale para todos os valores da chave **k** e pode ser usada para distinguir $R4_k$ de uma permutação aleatória. Vale observar que a estrutura de distinção não opera com probabilidade 1. Detalhes sobre a montagem da estrutura podem ser encontrados em [GiMi00].

5.4.2. Ataque sobre 7 rodadas

Da mesma forma que o ataque de saturação de seis rodadas, este ataque é montado adicionando uma rodada antes da estrutura de distinção e duas depois dela.

Assumindo um valor para quatro bytes da chave da primeira rodada, é possível determinar um conjunto de textos claros de forma que as entradas da segunda rodada são constantes em três colunas. Este conjunto é dividido em subconjuntos com valores constantes para os parâmetros u,v e w na entrada da segunda rodada. Deverão existir 2^{16} subconjuntos, com 16 textos claros em cada um deles. 16 valores para x são suficientes para determinar como os dois conjuntos de parâmetros resultam em funções idênticas, com probabilidade de erro desprezível. Os textos claros requeridos para todos os 2^{32} valores dos quatro bytes da primeira chave da rodada podem ser obtidos a partir de um conjunto de 2^{32} textos claros.

Cada um dos bytes de $e_{i,j}$ pode ser expresso como uma função de 4 bytes de texto cifrado e 5 bytes da chave. Portanto, os y valores dependem de 20 bytes da chave. Estes bytes devem ser sugeridos para proceder ao ataque. A complexidade do ataque pode ser estimada em 2^{192} execuções da transformação da rodada, menor que a busca exaustiva para a chave de 256 bits e aproximadamente igual a da busca exaustiva para a chave de 192 bits.

Existe uma variante deste ataque que ataca a versão de 128 bits da versão reduzida de sete rodadas do Rijndael. Este modo de ataque é levemente mais rápido que a busca exaustiva para chave de 128 bits.

5.5. Ataque por Interpolação

Este novo tipo de ataque foi criado por T. Jakobsen e L. Knudsen e publicado em [JaKn97]. Nele, o criptoanalista constrói polinômios usando pares texto claro / texto cifrado. Este ataque é praticável se os componentes da cifra tem um expressão algébrica compacta que pode ser combinada com determinadas expressões sem atingir uma alta complexidade. Se um polinômio construído tem grau pequeno, apenas poucos pares texto

claro / texto cifrado são necessários para encontrar os (dependentes da chave) coeficientes polinomiais.

A expressão polinomial que descreve a transformação em $GF(2^8)$ executada por S_{Box} no Rijndael, pode ser encontrada através da técnica da interpolação de Lagrange. S_{Box} em função do byte de entrada x, é dada por:

$$S_{Box}[x]=63 \oplus 8Fx^{127} \oplus B5x^{191} \oplus 01x^{223} \oplus F4x^{239} \oplus 25x^{247} \oplus F9x^{251} \oplus 09x^{253} \oplus 05x^{254}.$$

Esta complicada expressão de S_{Box} em $GF(2^8)$, combinada ao efeito da transformações provedoras da mistura e das transposições, impede o ataque para versões da cifra com mais do que poucas rodadas.

Ferguson, Schroeppel, e Whiting mostram uma forma de encontrar uma expressão algébrica para o Rijndael de dez rodadas [FSW01]. A expressão deve conter 2^{50} termos. Esta interessante descoberta é citada no mais novo ataque ao Rijndael "Criptoanálise de Cifras de Bloco com Sistemas de Equações Superdefinadas" [CoPi02], que será comentado na seção 5.8.

5.6. Ataque com Chaves Relacionadas

Este tipo de ataque foi introduzido por E. Biham [Biha94]. Foi demonstrado depois por Kelsey que muitas cifras tem fraquezas com chaves relacionadas [KeSc96]. Em ataques com chaves relacionadas o criptoanalista deve ter acesso a textos cifrados que resultem de cifragens usando diferentes (desconhecidas ou parcialmente desconhecidas) chaves com uma relação escolhida. O esquema de expansão da chave usado pelo Rijndael tem alta difusão e não linearidade, dificultando assim a montagem de um ataque desse tipo. N. Ferguson e equipe descrevem um ataque usando chaves relacionadas reduzido para nove rodadas da cifra [FKSS01]. O ataque trabalha sobre versões do Rijndael com 128 bits de comprimento do bloco e 256 bits da chave. São necessários 2^{72} textos claros escolhidos e tem um fator de trabalho de 2^{227} cifragens, que é mais rápido do que a busca exaustiva da chave.

5.7. Ataques de implementação

Ataques de implementação são baseados não apenas em propriedades matemáticas da cifra, mas também nas características físicas de implementação. Ataques de implementação típicos são os ataques pelo tempo [Koch96], introduzido por P. Kocher, e ataques por análise de potência[KJJ99], introduzido por P. Kocher, J. Jaffe e B. Jun. Nos ataques pelo tempo, a medida do tempo de execução do algoritmo de cifragem é usada para obter informações sobre a chave. Nos ataque por análise de potência, medições do consumo de potência do dispositivo que executa a cifragem são usadas para extrair informações sobre a chave. Ataques por análise de potência podem ser generalizados para medida de outras grandezas como radiação emitida ou quantidade de calor emanada do dispositivo.

5.7.1. Ataque pelo tempo

Um ataque pelo tempo pode ser montado contra cifras cujo tempo de execução do algoritmo depende do valor da chave. Por exemplo, se tivermos num determinado momento de uma cifragem diante de uma cláusula condicional que dependa de algum valor da chave para ser executada e o valor corresponder a não execução desse processo o tempo total de cifragem pode ser reduzido. Logo, uma medida detalhada do tempo de execução da chave poderá identificar a que universo de chaves possíveis pertence a chave usada.

Para o Rijndael a única fraqueza possível é devido ao uso da função xtime( ) que executa a multiplicação em $GF(2^8)$ na transformação MixColumn em processadores de 8 bits (veja seção 3.5.1.). Todas as outras operações do Rijndael são naturalmente executadas de modo que durem o mesmo tempo independente da chave. Esta fraqueza pode ser eliminada através do uso de tabelas de consulta para implementar xtime( ).

5.7.2. Ataque por análise de potência

Análise de Potência Simples (*Simple Power Analysis – SPA*) é um tipo de ataque onde o analista obtém informações sobre o consumo de potência durante a execução de

uma cifra. Este ataque é especialmente aplicável a dispositivos que dependam de fonte de potência externa como, por exemplo, smart cards. Se a seqüência de instruções executadas influenciam no consumo de potência, um atacante pode obter informações ao avaliar os dados coletados sobre o consumo durante a cifragem. O Rijndael pode ser facilmente implementado com uma seqüência fixa de instruções de modo a prevenir este tipo de ataque.

Na maior parte dos processadores, o padrão de consumo de potência depende do valor dos operandos. Para um dado bit num registrador o fato de ele estar ativo (1) pode levar o processador a um consumo mais elevado do que se ele estivesse inativo (0). Esta variação normalmente está mergulhada no próprio ruído de operação do dispositivo e não aparece em poucas medições. Porém, combinando a medida de muitas cifragens, o criptoanalista pode remover da medida a média de ruído e obter a informação sobre o valor do operando. Esta classe de ataque é chamada de Análise Diferencial de Potência (*Differential Power Analysis-DPA*). Proteger contra estes ataques é muito mais difícil do que contra ataques por analise de potência simples. Os ataques DPA são divididos em três categorias.

Proteções individuais de instruções

Cada instrução pode ser protegida individualmente contra análise de potência. Uma idéia, a do balanceamento de carga, é mostrada pelos autores do Rijndael em [Daem99]. A idéia consiste em eliminar a dependência entre o consumo de potência e o valor dos operandos através de um redesenho do hardware. Outra técnica é a de mascaramento dos operandos. Nela, as instruções sobre o operando x são substituídas por instruções sobre o operando x´ ou x´´, de modo que x´ e x´´ sejam não previsíveis pelo atacante. Apenas o conhecimento conjunto de x´e x´´ revela a informação sobre o valor de x. Alguns aproximações são propostas em [AkGi01,DPA01,Mess01]. Para o Rijndael, o fato de poder implementar a cifra com tabelas de consulta e ou-exclusivos é um grande trunfo a favor.

Desincronização

Baseia-se em trocar a seqüência de instruções a cada cifragem completa ou parte dela, dificultando a obtenção de uma padronização significativa. O paralelismo da rodada do Rijndael permite algumas variações na seqüência de instruções, contudo, a quantidade de variações é limitada.

Complexidade na expansão da chave

S. Chari e equipe defendem a idéia de que um esquema de expansão da chave complexo ajuda a aumentar a resistência contra ataques de análise de potência [CJR99]. Se o conhecimento de uma chave da rodada não permite a reconstrução da chave da cifra, um criptoanalista terá que recuperar mais – ou todas – as chaves das rodadas para ter seu objetivo atingido. Apesar disto, da mesma forma que foi obtida a chave de uma rodada, as outras podem ser obtidas a um custo semelhante. Somando todos os custos é que se obtém a segurança necessária e as justificativas para o uso de um esquema complexo.

5.8. Ataques com sistemas de equações superdefinidas

Este ataque recente, publicado na AsiaCrypt 2002 no mês de dezembro, foi criado por N. Courtois e J. Pieprzyk [CoPi02] e baseia-se em resolver um sistema multivariável quadrático para encontrar a chave, partindo de uma estrutura chamada XSL (X = ou exclusivo, S = S-Box, L = camada linear) existente em determinadas cifras de bloco. Quando este sistema torna-se superdefinido, isto é, com mais equações do que incógnitas, a complexidade da sua resolução cai substancialmente. Para o caso do Rijndael as equações são extraídas do S-Box da cifra partindo da expressão usada na seção 5.5. A partir daí uma série de melhoramentos são feitos no ataque. A melhor aproximação para o ataque, usando equações cúbicas, estima ser capaz de quebrar o Rijndael de 256 bits de chave com uma complexidade de 2^{203}, sendo melhor do que a busca exaustiva da chave.

Após este ataque, muitos comentários surgiram sobre o assunto. Alguns foram mais cautelosos em dizer que o ataque poderia ter sucesso enquanto outros partiram contra o ataque afirmando haver incorreções na aproximações utilizadas. Courtois se defende

afirmando que os críticos não conhecem o suficiente o ataque para criticá-lo. O fato é que o ataque é muito complexo e não existe uma simulação implementada contra o Rijndael nem contra o Serpent [ABK98], a outra cifra de bloco finalista do concurso de submissão ao AES que o ataque também afirma poder quebrar.

Um comentário interessante de B. Schneier pode ser encontrado na edição de 15 de outubro de 2002 do jornal "Crypto-Gram Newsletter" da Counterpane, disponível no endereço eletrônico <http://www.counterpane.com/crypto-gram.html>.

5.9. Chaves fracas

Apesar de haver muita simetria na estrutura da cifra, os autores foram cuidadosos em eliminá-las no comportamento da cifragem. A possibilidade de chaves fracas ou semifracas como no DES, proposta por D. Davies em [Davi83], é improvável devido ao fato da cifragem ser diferente da decifragem. A possibilidade de chaves equivalentes também pode ser considerada remota pois o processo de expansão da chave é não linear.

Chaves fracas como no IDEA [Lai92], são chaves que resultam em uma transformação com fraqueza detectada, ou seja, quando a parte não linear da cifra depende fortemente da aplicação da chave. Este não é o caso do Rijndael pois os S-boxes, responsáveis pela parte não linear da cifra, são independentes da chave.

Capítulo 6: Conclusões e Sugestões para Futuras Pesquisas

Pelo fato do AES ser um padrão do governo dos Estados Unidos, talvez fosse esperado, como resultado do processo de seleção para o novo padrão, a escolha de uma cifra americana como o RC6 da RSA [RRSY98], o Twofish da Counterpane [SKWW98] ou o MARS [BCDG98], da IBM. O critério de seleção não foi parcial e escolheu, sem barreiras de nacionalidade, o Rijndael. Ele mostrou numa série de critérios estabelecidos pelo NIST grandes vantagens em relação aos demais algoritmos, resultando em reconhecimento por parte de toda a comunidade internacional aos pesquisadores belgas Joan Daemen e Vincent Rijmen e seu algoritmo.

O Rijndael é a terceira de uma série de cifras criadas pela dupla de pesquisadores, que começou com o SHARK [RDP96], contando com a participação de Bart Preneel, e passou pelo Square [DKR97], com a participação de Lars Knudsen. O processo, iniciado em meados de 1995, sofreu diversos aperfeiçoamentos até chegar a versão submetida ao processo de seleção.

Talvez o principal motivo da sua escolha tenha sido o desempenho. Um padrão deve ter um alto índice de desempenho nas mais diversas plataformas. Além disso, sua concepção já implementa níveis elevados de segurança contra ataques conhecidos como podemos ver no capítulo 5. Porém, é necessário ressaltar o fato do Rijndael ser o padrão utilizado para documentos de grau de sigilo "não classificado", ou seja, dados não sensíveis. É obvio que um governo de um país hegemônico como os EUA não admitiriam abertamente o uso de um padrão para informações sensíveis. No mundo da criptografia os padrões são indesejáveis pois reduzem os níveis de segurança. É verdade também que a segurança de uma cifra de bloco não pode se basear em detalhes obscuros do algoritmo. O fato é que o governo americano passou um período de mais de três anos investindo num processo para selecionar o melhor candidato para ser o padrão e não escolheria um algoritmo com margem de segurança reduzida. O Rijndael pode ser considerada a melhor cifra de bloco existente para ser o AES. Não se pode desmerecer nem desprezar as outras

finalistas. A criptografia ganhou um conjunto de cifras de bloco muito bem projetadas e com capacidade de ocupar espaço em diversos projetos de segurança mundo afora.

Finalmente, deve ser lembrado o fato da segurança ser baseada em ataques conhecidos. A criatividade associada à alta velocidade dos processadores modernos e à capacidade de processamento paralelo impulsiona a criptoanálise, este misto de ciência e arte. O Rijndael já foi projetado sobre a estratégia do trilho largo, protegendo-o contra os clássicos ataques de criptoanálise linear e diferencial. Desenvolver novas linhas de ataque é o principal caminho para evolução dos níveis de segurança existentes atualmente.

Foi apresentada no capítulo 4 uma implementação didática em linguagem C cuja função é demonstrar aos interessados como ocorre a evolução da cifragem ou da decifragem passo a passo, mostrando as trasformações sofridas pelo estado ao longo do processo. Não é uma implementação veloz pois não foram usadas as técnicas de implementação veloz expostas na seção 3.5. Seu objetivo é o de ensinar como ocorrem os efeitos de difusão e confusão propostas pelas camadas linear e não-linear da cifra sobre os bytes. Caso fossem usados os melhoramentos, as rodadas se transformariam em consultas a tabelas e não atingiriam o objetivo proposto. Encorajo aos conhecerdores de linguagem de baixo nível como assembler a tentar implementações usando estas ferramentas. Com certeza alcançarão um desempenho bem mais elevado. Não foi feita também a possibilidade de alterar o Nb, é uma outra sugestão porposta aos conhecedores da linguagem C. Mesmo não sendo um conhecedor da linguagem C é possível estudar o apêndice C e encarar a tarefa como um desafio.

Ainda naquele mesmo capítulo foi proposta uma implemenção de uma função unidirecional hash usando o Rijndael. Foi feito um comparativo dela com as finalistas do concurso do AES e com o DES. De forma notória podemos observar a vantagem de desempenho do Rijndael frente às outras cifras. Não foram feitas implementações de 256 bits de comprimento do bloco de texto para as outras cifras. Encorajo aos conhecedores da linguagem C a completar a tabela criando implementações dos algoritmos das demais finalistas para blocos de texto de 256 bits.

No capítulo 5 foram vistos os principais ataques conhecidos contra o Rijndael. Foi mostrada a ineficiência desses ataques contra a cifra. Na maioria deles, umas poucas rodadas seriam suficientes para inviabilizar o ataque. Exceto no último ataque, com

sistemas de equações superdefinidas, os demais ataques atestam total segurança da cifra. Quanto a este último ataque, não podemos afirmar nada conclusivo, nem os próprios autores. No texto do artigo que publica o ataque são usadas palavras como "parece" e "deve" durante as conclusões acerca do funcionamento contra o Rijndael e o Serpent. De qualquer modo serve como sugestão para um trabalho futuro a implementação prática deste ataque, apesar dele não ter sido feito nem pelos autores. Além disso, a compreensão dele é uma façanha alcançada por poucos. Aqueles críticos mais ofensivos sobre a eficácia do ataque são acusados por Courtois de não o terem compreendido.

Como lição final para todos que não se fazem entender, é possível invocar a frase de René Descartes: "Quando questões transcendentais são discutidas, seja transcendentalmente claro".

Apêndice A – Tabelas das Funções SubByte( ) e Inv SubByte

Dado o byte xy em hexadecimal procure o valor x entre as linhas e y entre as colunas. Na posição de encontro linha × coluna estará o resultado aplicação da transformação SubByte( ) ou InvSubByte( ).

		Y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	F
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	Cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	F	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tabela A1: Transformação SubByte sobre o byte xy (em hexadecimal)

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7ª	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	F	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Tabela A2: Transformação InvSubByte sobre o byte xy (em hexadecimal)

Apêndice B – Exemplos de Expansões de Chaves

Este apêndice mostra exemplos da expansão da chave de 128, 192 e 256 bits de comprimento, ao longo do processo. Cada vetor W[i] de quatro bytes mostra os valores armazenados como visto na seção 3.3.

Chave de 128 bits:

Chave da Cifra: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

W[0]=00112233 W[1]=44556677 W[2]=8899aabb W[3]=ccddeeff

i (dec)	temp	Após RotWord()	Após SubWord()	Rcon [i/Nk]	Após XOR com Rcom	W[i-Nk]	W[i]= temp XOR W[i-Nk]
4	ccddeeff	28164bc0	3447b3ba	01000000	c028164b	00112233	c0393478
5	c0393478					44556677	846c520f
6	846c520f					8899aabb	0cf5f8b4
7	0cf5f8b4					ccddeeff	c028164b
8	c028164b	47b3ba36	a06df405	02000000	3647b3ba	c0393478	f67e87c2
9	f67e87c2					846c520f	7212d5cd
10	7212d5cd					0cf5f8b4	7ee72d79
11	7ee72d79					c028164b	becf3b32
12	becf3b32	e223ae8e	9826e419	04000000	8ee223ae	f67e87c2	789ca46c
13	789ca46c					7212d5cd	0a8e71a1
14	0a8e71a1					7ee72d79	74695cd8
15	74695cd8					becf3b32	caa667ea
16	caa667ea	8587742c	97179271	08000000	2c858774	789ca46c	54192318
17	54192318					0a8e71a1	5e9752b9
18	5e9752b9					74695cd8	2afe0e61
19	2afe0e61					caa667ea	e058698b
20	e058698b	f93de17a	9927f8da	10000000	7af93de1	54192318	2ee01ef9
21	2ee01ef9					5e9752b9	70774c40
22	70774c40					2afe0e61	5a894221
23	5a894221					e058698b	bad12baa
24	bad12baa	f1acf41e	a191bf72	20000000	1ef1acf4	2ee01ef9	3011b20d
25	3011b20d					70774c40	4066fe4d

26	4066fe4d					5a894221	1aefbc6c
27	1aefbc6c					bad12baa	a03e97c6
28	a03e97c6	88b4e0f2	c48de189	40000000	f288b4e0	3011b20d	c29906ed
29	c29906ed					4066fe4d	82fff8a0
30	82fff8a0					1aefbc6c	981044cc
31	981044cc					a03e97c6	382ed30a
32	382ed30a	666707b1	3385c5c8	80000000	b1666707	c29906ed	73ff61ea
33	73ff61ea					82fff8a0	f100994a
34	f100994a					981044cc	6910dd86
35	6910dd86					382ed30a	513e0e8c
36	513e0e8c	ab64d1a9	62433ed3	1b000000	a9ab64d1	73ff61ea	da54053b
37	da54053b					f100994a	2b549c71
38	2b549c71					6910dd86	424441f7
39	424441f7					513e0e8c	137a4f7b
40	137a4f7b	84217dec	5ffdffce	36000000	ec84217d	da54053b	36d02446
41	36d02446					2b549c71	1d84b837
42	1d84b837					424441f7	5fc0f9c0
43	5fc0f9c0					137a4f7b	4cbab6bb

Chave de 192 bits:

Chave da Cifra: ff ee dd cc bb aa 99 88 77 66 55 44
33 22 11 00 0f 0e 0d 0c 0b 0a 09 08

W[0]=ffeeddcc W[1]=bbaa9988 W[2]=77665544 W[3]=33221100
W[4]=0f0e0d0c W[5]=0b0a0908

i (dec)	temp	Após RotWord()	Após SubWord()	Rcon [i/Nk]	Após XOR com Rcom	W[i-Nk]	W[i]= temp XOR W[i-Nk]
6	0b0a0908	01302b66	7c04f133	01000000	6601302b	ffeeddcc	99efede7
7	99efede7					bbaa9988	2245746f
8	2245746f					77665544	5523212b
9	5523212b					33221100	6601302b
10	6601302b					0f0e0d0c	690f3d27
11	690f3d27					0b0a0908	6205342f
12	6205342f	1815aa69	ad59acf9	02000000	691815aa	99efede7	f0f7f84d
13	f0f7f84d					2245746f	d2b28c22
14	d2b28c22					5523212b	8791ad09
15	8791ad09					6601302b	e1909d22

16	e1909d22					690f3d27	889fa005
17	889fa005					6205342f	ea9a942a
18	ea9a942a	22e587bc	93d91765	04000000	bc22e587	f0f7f84d	4cd51dca
19	4cd51dca					d2b28c22	9e6791e8
20	9e6791e8					8791ad09	19f63ce1
21	19f63ce1					e1909d22	f866a1c3
22	f866a1c3					889fa005	70f901c6
23	70f901c6					ea9a942a	9a6395ec
24	9a6395ec	2aceb8f3	e58b6c0d	08000000	f32aceb8	4cd51dca	bfffd372
25	bfffd372					9e6791e8	2198429a
26	2198429a					19f63ce1	386e7e7b
27	386e7e7b					f866a1c3	c008dfb8
28	c008dfb8					70f901c6	b0f1de7e
29	b0f1de7e					9a6395ec	2a924b92
30	2a924b92	b34fe55f	6d84d9cf	10000000	5fb34fe5	bfffd372	e04c9c97
31	e04c9c97					2198429a	c1d4de0d
32	c1d4de0d					386e7e7b	f9baa076
33	f9baa076					c008dfb8	39b27fce
34	39b27fce					b0f1de7e	8943a1b0
35	8943a1b0					2a924b92	a3d1ea22
36	a3d1ea22	87930a1e	17dc6772	20000000	1e87930a	e04c9c97	fecb0f9d
37	fecb0f9d					c1d4de0d	3f1fd190
38	3f1fd190					f9baa076	c6a571e6
39	c6a571e6					39b27fce	ff170e28
40	ff170e28					8943a1b0	7654af98
41	7654af98					a3d1ea22	d58545ba
42	d58545ba	6ef403d7	9fbf7b0e	40000000	d76ef403	fecb0f9d	29a5fb9e
43	29a5fb9e					3f1fd190	16ba2a0e
44	16ba2a0e					c6a571e6	d01f5be8
45	d01f5be8					ff170e28	2f0855c0
46	2f0855c0					7654af98	595cfa58
47	595cfa58					d58545ba	8cd9bfe2
48	8cd9bfe2	89864b5	304643d5	80000000	b5089864	29a5fb9e	9cad63fa
49	9cad63fa					16ba2a0e	8a1749f4
50	8a1749f4					d01f5be8	5a08121c
51	5a08121c					2f0855c0	750047dc

Chave de 256 bits:

Chave da Cifra: cf ce cd cc cb ca c9 c8 c7 c6 c5 c4 c3 c2 c1 c0
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f

W[0]=cfcecdcc W[1]=cbcac9c8 W[2]=c7c6c5c4 W[3]=c3c2c1c0

W[4]=70717273 W[5]=74757677 W[6]=78797a7b W[7]=7c7d7e7f

i (dec)	temp	Após RotWord()	Após SubWord()	Rcon [i/Nk]	Após XOR com Rcom	W[i-Nk]	W[i]= temp XOR W[i-Nk]
8	7c7d7e7f	f3d210fe	db5cabb	01000000	fef3d210	cfcecdcc	313d1fdc
9	313d1fdc					cbcac9c8	faf7d614
10	faf7d614					c7c6c5c4	3d3113d0
11	3d3113d0					c3c2c1c0	fef3d210
12	fef3d210		d7d574ea			70717273	cb7cc7b9
13	cb7cc7b9					74757677	bf09b1ce
14	bf09b1ce					78797a7b	c770cbb5
15	c770cbb5					7c7d7e7f	bb0db5ca
16	bb0db5ca	d574ead5	03928703	02000000	d5d574ea	313d1fdc	e4e86b36
17	e4e86b36					faf7d614	1e1fbd22
18	1e1fbd22					3d3113d0	232eaef2
19	232eaef2					fef3d210	dddd7ce2
20	dddd7ce2		78ca4678			cb7cc7b9	0abdd721
21	0abdd721					bf09b1ce	b5b466ef
22	b5b466ef					c770cbb5	72c4ad5a
23	72c4ad5a					bb0db5ca	c9c91890
24	c9c91890	ad60ddd9	95d0c135	04000000	d9ad60dd	e4e86b36	3d450beb
25	3d450beb					1e1fbd22	235ab6c9
26	235ab6c9					232eaef2	0074183b
27	0074183b					dddd7ce2	dda964d9
28	dda964d9		661a9678			0abdd721	cb6e9414
29	cb6e9414					b5b466ef	7edaf2fb
30	7edaf2fb					72c4ad5a	0c1e5fa1
31	0c1e5fa1					c9c91890	c5d74731
32	c5d74731	a0c7a606	e0c6246f	08000000	06a0c7a6	3d450beb	3be5cc4d
33	3be5cc4d					235ab6c9	18bf7a84
34	18bf7a84					0074183b	18cb62bf
35	18cb62bf					dda964d9	c5620666
36	c5620666		aca8c324			cb6e9414	6dc4fb27

37	6dc4fb27					7edaf2fb	131e09dc
38	131e09dc					0c1e5fa1	1f00567d
39	1f00567d					c5d74731	dad7114c
40	dad7114c	8229571e	8229571e	10000000	1e822957	3be5cc4d	2567e51a
41	2567e51a					18bf7a84	3dd89f9e
42	3dd89f9e					18cb62bf	2513fd21
43	2513fd21					c5620666	e071fb47
44	e071fb47		0a76e0f8			6dc4fb27	8c67f487
45	8c67f487					131e09dc	9f79fd5b
46	9f79fd5b					1f00567d	8079ab26
47	8079ab26					dad7114c	5aaeba6a
48	5aaeba6a	f402bec4	bf77ae1c	20000000	c4f402be	2567e51a	e193e7a4
49	e193e7a4					3dd89f9e	dc4b783a
50	dc4b783a					2513fd21	f958851b
51	f958851b					e071fb47	19297e5c
52	19297e5c		060dd648			8c67f487	58c207cd
53	58c207cd					9f79fd5b	c7bbfa96
54	c7bbfa96					8079ab26	47c251b0
55	47c251b0					5aaeba6a	1d6cebda
56	1d6cebda	e957a410	1e5b49ca	40000000	10e957a4	e193e7a4	f17ab000
57	f17ab000					dc4b783a	2d31c83a
58	2d31c83a					f958851b	d4694d21
59	d4694d21					19297e5c	cd40337d

Apêndice C – Código Fonte da Implementação Didática em C

```
#include <stdio.h>
#include <stdlib.h>
int s_box(int num){
int table[256]=
{0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
 0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
 0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
 0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
 0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
 0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
 0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
 0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
 0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
 0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
 0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
 0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
 0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
 0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
 0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16};
  num=table[num];
  return num;
}
int inv_s_box(int num){
  int table[256]=
{0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
 0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
 0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
 0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
 0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
 0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
 0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
 0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
 0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
 0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
 0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
 0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
 0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
 0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
 0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d};
  num=table[num];
  return num;
}
unsigned long int SubWord(unsigned long int num)
{
  int i;
  unsigned long int temp[4];
  temp[0]=(num>>24);
  temp[1]=(num>>16)&0xff;
  temp[2]=(num>>8)&0xff;
  temp[3]=num&0xff;
  for (i=0;i<4;i++) temp[i]=s_box(temp[i]);
```

```
  num=(temp[0]<<24)^(temp[1]<<16)^(temp[2]<<8)^temp[3];
  return num;
}
unsigned long int RotWord(unsigned long int num)
{
  unsigned long int temp;
  temp=(num&0xff000000)>>24;
  num=(num<<8)^temp;
  return num;
}
int conv_char_hex(char ch)
{
  int hex;
  switch(ch)
  {
    case '0':{hex=0x0;break;}
    case '1':{hex=0x1;break;}
    case '2':{hex=0x2;break;}
    case '3':{hex=0x3;break;}
    case '4':{hex=0x4;break;}
    case '5':{hex=0x5;break;}
    case '6':{hex=0x6;break;}
    case '7':{hex=0x7;break;}
    case '8':{hex=0x8;break;}
    case '9':{hex=0x9;break;}
    case 'A':{hex=0xa;break;}
    case 'a':{hex=0xa;break;}
    case 'B':{hex=0xb;break;}
    case 'b':{hex=0xb;break;}
    case 'C':{hex=0xc;break;}
    case 'c':{hex=0xc;break;}
    case 'D':{hex=0xd;break;}
    case 'd':{hex=0xd;break;}
    case 'E':{hex=0xe;break;}
    case 'e':{hex=0xe;break;}
    case 'F':{hex=0xf;break;}
    case 'f':{hex=0xf;break;}
    default:hex=0xff;
  }
return hex;
}
void InputData(int *Sta,int *Key,int *Nkey,int *Nrod, int *kind )
{
  int i,j,Nk,cont1,cont2,*Sta0,*Key0,ki;
  char *ind1,stat_char[100];
  printf("Cifrador/Decifrador AES(Rijndael Nb=4)\n");
  inicio: printf("Digite uma opcao (1)Cifrador  (2)Decifrador : ");
  scanf("%d",&ki);
  if ((ki!=1)&&(ki!=2)) {
    printf("Voce digitou uma opcao invalida !!!\n");
    goto inicio; }
  inicio1:printf("Digite o comprimento da chave [(Nk)/bits] (4)/128 (6)/196
(8)/256 : ");
  scanf("%d",&Nk);
  switch(Nk)
  {
    case 4:{*Nrod=10;break;}
    case 6:{*Nrod=12;break;}
    case 8:{*Nrod=14;break;}
    default:{printf("Comprimento da chave invalido!!!\n");goto inicio1;};
  }
  char key_char[100];
```

```
Sta0=Sta;
inicio2:if (ki==1)printf("Digite os 16 bytes de texto claro em hexadecimal: ");
else printf("Digite os 16 bytes de texto cifrado em hexadecimal: ");
scanf("%s",&stat_char);
ind1=stat_char;
Sta=Sta0;
for (cont1=0;cont1<4;cont1++)
  {
  for (cont2=0;cont2<4;cont2++)
    {
    if (conv_char_hex(*ind1)!=0xFF)
      {
      *(Sta+4*cont2+cont1)=conv_char_hex(*ind1);
      }
    else
      {
      printf("Voce entrou ao menos um digito nao hexadecimal !!!\n");
      goto inicio2;
      }
    ind1++;
    if (conv_char_hex(*ind1)!=0xFF)
      {
      *(Sta+4*cont2+cont1)=(*(Sta+4*cont2+cont1)<<4)^conv_char_hex(*ind1);
      ind1++;
      }
    else
      {
      printf("Voce entrou ao menos um digito nao hexadecimal !!!\n");
      goto inicio2;
      }
    }
  }
printf("O estado inicial do processo:\n");
Sta=Sta0;
for (i=0;i<16;i++)
  {
  printf("%3x",*Sta);
  if (!((i+1)%4)) printf("\n");
  Sta++;
  }
Key0=Key;
inicio3:printf("Digite os %d bytes da chave em hexadecimal: ",4*Nk);
scanf("%s",&key_char);
ind1=key_char;
Key=Key0;
for (cont1=0;cont1<Nk;cont1++)
  {
  for (cont2=0;cont2<4;cont2++)
    {
    if (conv_char_hex(*ind1)!=0xFF)
      {
      *(Key+Nk*cont2+cont1)=conv_char_hex(*ind1);
      ind1++;
      }
    else
      {
      printf("Voce entrou ao menos um digito nao hexadecimal !!!\n");
      goto inicio3;
      }
    if (conv_char_hex(*ind1)!=0xFF)
      {
      *(Key+Nk*cont2+cont1)=(*(Key+Nk*cont2+cont1)<<4)^conv_char_hex(*ind1);
```

```
        ind1++;
        }
      else
        {
        printf("Voce entrou ao menos um digito nao hexadecimal !!!\n");
        goto inicio3;
        }
      }
    }
  printf("Chave:\n");
  Key=Key0;
  for (i=0;i<4*Nk;i++)
    {
    printf("%3x",*Key);
    if (!((i+1)%Nk)) printf("\n");
    Key++;
    }
  *Nkey=Nk;
  *kind=ki;
}
void KeyExpansion(int *Key,unsigned long int *ExpKey,int Nk,int Nr)
{
  int i,j;
  unsigned long int temp,Rcon[11]={0,0x01000000,0x02000000,0x04000000,0x08000000,
0x10000000,0x20000000,0x40000000,0x80000000,0x1B000000,0x36000000};
  if (Nk<=6)
    {
    for (i=0;i<Nk;i++)

*(ExpKey+i)=(*(Key+i)<<24)^(*(Key+Nk+i)<<16)^(*(Key+2*Nk+i)<<8)^(*(Key+3*Nk+i));
    for(i=Nk;i<4*(Nr+1);i++)
      {
      temp=*(ExpKey+i-1);
      if (i%Nk==0) temp=SubWord(RotWord(temp))^Rcon[i/Nk];
      *(ExpKey+i)=*(ExpKey+i-Nk)^temp;
      }
    }
  else
    {
    for (i=0;i<Nk;i++)
*(ExpKey+i)=(*(Key+i)<<24)^(*(Key+Nk+i)<<16)^(*(Key+2*Nk+i)<<8)^(*(Key+3*Nk+i));
    for(i=Nk;i<4*(Nr+1);i++)
      {
      temp=*(ExpKey+i-1);
      if (i%Nk==0) temp=SubWord(RotWord(temp))^Rcon[i/Nk];
      else if ((i%Nk)==4) temp=SubWord(temp);
      *(ExpKey+i)=*(ExpKey+i-Nk)^temp;
      }
    }
}
void AddRoundKey_e(int *State,unsigned long int *ExpKey,int *nr)
{
  int i;
  for (i=0;i<4;i++)
    {
    *(State+i)=(*(State+i))^(*(ExpKey+i+*nr*4)>>24);
    *(State+i+4)=(*(State+i+4))^((*(ExpKey+i+*nr*4)>>16)&(0xff));
    *(State+i+8)=(*(State+i+8))^((*(ExpKey+i+*nr*4)>>8)&(0xff));
    *(State+i+12)=(*(State+i+12))^(*(ExpKey+i+*nr*4)&(0xff));
    }
  *nr=(*nr)++;
}
```

```
void AddRoundKey_d(int *State,unsigned long int *ExpKey,int *nr,int *Nr)
{
  int i;
  for (i=0;i<4;i++)
    {
    *(State+i)=(*(State+i))^(*(ExpKey+*Nr*4+i-*nr*4)>>24);
    *(State+i+4)=(*(State+i+4))^((*(ExpKey+*Nr*4+i-*nr*4)>>16)&(0xff));
    *(State+i+8)=(*(State+i+8))^((*(ExpKey+*Nr*4+i-*nr*4)>>8)&(0xff));
    *(State+i+12)=(*(State+i+12))^(*(ExpKey+*Nr*4+i-*nr*4)&(0xff));
    }
}
void print_state(int *Sta,int nr)
{
  int i;
  printf("Estado[%d]:\n",nr);
  for (i=1;i<=16;i++)
    {
    if (i%4==0)
      printf("%3x\n",*(Sta+i-1));
    else
      printf("%3x",*(Sta+i-1));
    }
  system("PAUSE");
}
void print_key_e(unsigned long int *ExpKey,int nr)
{
  int i;
  printf("Chave da Rodada[%d]:\n",nr);
  for (i=0;i<4;i++)
    {
    printf("%3x",(*(ExpKey+nr*4)>>((3-i)*8))&(0xff));
    printf("%3x",(*(ExpKey+1+nr*4)>>((3-i)*8))&(0xff));
    printf("%3x",(*(ExpKey+2+nr*4)>>((3-i)*8))&(0xff));
    printf("%3x\n",(*(ExpKey+3+nr*4)>>((3-i)*8))&(0xff));
    }
  system("PAUSE");
}
void print_key_d(unsigned long int *ExpKey,int nr, int Nr)
{
  int i;
  printf("Chave da Rodada[%d]:\n",Nr-nr);
  for (i=0;i<4;i++)
    {
    printf("%3x",(*(ExpKey+4*Nr-nr*4)>>((3-i)*8))&(0xff));
    printf("%3x",(*(ExpKey+1+4*Nr-nr*4)>>((3-i)*8))&(0xff));
    printf("%3x",(*(ExpKey+2+4*Nr-nr*4)>>((3-i)*8))&(0xff));
    printf("%3x\n",(*(ExpKey+3+4*Nr-nr*4)>>((3-i)*8))&(0xff));
    }
  system("PAUSE");
}
void SubByte(int *Sta)
{
  int i;
  for (i=0;i<16;i++) *(Sta+i)=s_box(*(Sta+i));
}
void InvSubByte(int *Sta)
{
  int i;
  for (i=0;i<16;i++) *(Sta+i)=inv_s_box(*(Sta+i));
}
```

```
void ShiftRow(int *Sta)
{
  int temp;
  temp=*(Sta+4);
  *(Sta+4)=*(Sta+5);
  *(Sta+5)=*(Sta+6);
  *(Sta+6)=*(Sta+7);
  *(Sta+7)=temp;
  temp=*(Sta+8);
  *(Sta+8)=*(Sta+10);
  *(Sta+10)=temp;
  temp=*(Sta+9);
  *(Sta+9)=*(Sta+11);
  *(Sta+11)=temp;
  temp=*(Sta+12);
  *(Sta+12)=*(Sta+15);
  *(Sta+15)=*(Sta+14);
  *(Sta+14)=*(Sta+13);
  *(Sta+13)=temp;
}
void InvShiftRow(int *Sta)
{
  int temp;
  temp=*(Sta+4);
  *(Sta+4)=*(Sta+7);
  *(Sta+7)=*(Sta+6);
  *(Sta+6)=*(Sta+5);
  *(Sta+5)=temp;
  temp=*(Sta+8);
  *(Sta+8)=*(Sta+10);
  *(Sta+10)=temp;
  temp=*(Sta+9);
  *(Sta+9)=*(Sta+11);
  *(Sta+11)=temp;
  temp=*(Sta+12);
  *(Sta+12)=*(Sta+13);
  *(Sta+13)=*(Sta+14);
  *(Sta+14)=*(Sta+15);
  *(Sta+15)=temp;
}
unsigned int mult(unsigned int num1, unsigned int num2)
{
  unsigned int result=0;
  int i;
  for (i=7;i>=0;i--)
    if ((num2>>i)&0x01)
      result=result^num1<<i;
  for (i=15;i>7;i--)
    if ((result>>i)&0x01)
      result=result^(0x11B<<(i-8));
  return result;
}
void MixColumn(int *Sta)
{
  int i,n_State[16];
  for (i=0;i<4;i++)
    {
n_State[i]=mult(*(Sta+i),0x2)^mult(*(Sta+i+4),0x3)^(*(Sta+i+8))^(*(Sta+i+12));
n_State[i+4]=(*(Sta+i))^mult(*(Sta+i+4),0x2)^mult(*(Sta+i+8),0x3)^(*(Sta+i+12));
n_State[i+8]=(*(Sta+i))^(*(Sta+i+4))^mult(*(Sta+i+8),0x2)^mult(*(Sta+i+12),0x3);
n_State[i+12]=mult(*(Sta+i),0x3)^(*(Sta+i+4))^(*(Sta+i+8))^mult(*(Sta+i+12),0x2);
    }
```

```
  for (i=0;i<16;i++)
    *(Sta+i)=n_State[i];
}
void InvMixColumn(int *Sta)
{
  int i,n_State[16];
  for (i=0;i<4;i++)
    {
n_State[i]=mult(*(Sta+i),0xe)^mult(*(Sta+i+4),0xb)^mult(*(Sta+i+8),0xd)^mult(*(St
a+i+12),0x9);

n_State[i+4]=mult(*(Sta+i),0x9)^mult(*(Sta+i+4),0xe)^mult(*(Sta+i+8),0xb)^mult(*(
Sta+i+12),0xd);

n_State[i+8]=mult(*(Sta+i),0xd)^mult(*(Sta+i+4),0x9)^mult(*(Sta+i+8),0xe)^mult(*(
Sta+i+12),0xb);

n_State[i+12]=mult(*(Sta+i),0xb)^mult(*(Sta+i+4),0xd)^mult(*(Sta+i+8),0x9)^mult(*
(Sta+i+12),0xe);
    }
  for (i=0;i<16;i++)
    *(Sta+i)=n_State[i];
}
void Round_e(int *State,unsigned long int *ExpKey,int *nr,int Nk )
{
  printf("RODADA (%d):\n",*nr);
  SubByte(State);
  printf("Apos SubByte:\n");
  print_state(State,*nr);
  ShiftRow(State);
  printf("Apos ShiftRow:\n");
  print_state(State,*nr);
  MixColumn(State);
  printf("Apos MixColumn:\n");
  print_state(State,*nr);
  print_key_e(ExpKey,*nr);
  AddRoundKey_e(State,ExpKey,nr);
  printf("Apos AddRoundKey:\n");
  print_state(State,*nr);
}
void Round_d(int *State,unsigned long int *ExpKey,int *nr,int Nk,int *Nr )
{
  InvShiftRow(State);
  printf("Antes de ShiftRow:\n");
  print_state(State,*Nr-*nr+1);
  InvSubByte(State);
  printf("Antes de SubByte:\n");
  print_state(State,*Nr-*nr+1);
  printf("RODADA (%d):\n",*Nr-*nr);
  print_key_d(ExpKey,*nr,*Nr);
  AddRoundKey_d(State,ExpKey,nr,Nr);
  printf("Antes de AddRoundKey:\n");
  print_state(State,*Nr-*nr);
  InvMixColumn(State);
  printf("Antes de MixColumn:\n");
  print_state(State,*Nr-*nr);
  *nr=(*nr)++;
}
```

```
void FinalRound_e(int *State,unsigned long int *ExpKey,int *nr,int Nk )
{
  printf("RODADA FINAL (%d):\n",*nr);
  SubByte(State);
  printf("Apos SubByte:\n");
  print_state(State,*nr);
  ShiftRow(State);
  printf("Apos ShiftRow:\n");
  print_state(State,*nr);
  print_key_e(ExpKey,*nr);
  AddRoundKey_e(State,ExpKey,nr);
  printf("Apos AddRoundKey: Estado Final (TEXTO CIFRADO): \n");
  print_state(State,*nr);
}
void FinalRound_d(int *State,unsigned long int *ExpKey,int *nr,int Nk , int *Nr)
  {
  InvShiftRow(State);
  printf("Antes de ShiftRow:\n");
  print_state(State,1);
  InvSubByte(State);
  printf("Antes de SubByte:\n");
  print_state(State,1);
  printf("RODADA FINAL (%d):\n",0);
  print_key_d(ExpKey,*nr,*Nr);
  AddRoundKey_d(State,ExpKey,nr,Nr);
  printf("Antes AddRoundKey: Estado Inicial(TEXTO CLARO):\n");
  print_state(State,0);
}

void main()  // Função Principal:
{
  int Nk,i,Nr,cont,num_rod,ki;
  int State[16], CipherKey[32];
  unsigned long int ExpandedKey[60];
  InputData(&State[0],&CipherKey[0],&Nk,&Nr,&ki);
  KeyExpansion(&CipherKey[0],&ExpandedKey[0],Nk,Nr);
  if (ki==1) {
    AddRoundKey_e(&State[0],&ExpandedKey[0],&num_rod);
    printf("Apos AddRoundKey inicial: \n");
    print_state(&State[0],num_rod);
    for (i=1;i<Nr;i++) Round_e(&State[0],&ExpandedKey[0],&num_rod,Nk);
    printf("\n");
    FinalRound_e(&State[0],&ExpandedKey[0],&num_rod,Nk);}
  else {
    AddRoundKey_d(&State[0],&ExpandedKey[0],&num_rod,&Nr);
    print_key_d(&ExpandedKey[0],num_rod,Nr);
    num_rod++;
    printf("Antes de MixColumn: \n");
    print_state(&State[0],10);
    for (i=Nr-1;i>0;i--) Round_d(&State[0],&ExpandedKey[0],&num_rod,Nk,&Nr);
    FinalRound_d(&State[0],&ExpandedKey[0],&num_rod,Nk,&Nr);}
}
```

Apêndice D – Exemplo de Operação da Cifragem

O diagrama abaixo mostra a evolução do estado ao longo das aplicações das transformações durante uma cifragem. Foi usado Nb=Nk=4.

Texto claro = `00010203 04050607 08090a0b 0c0d0e0f`

Chave da cifra = `00112233 44556677 8899aabb ccddeeff`

Nº da rodada	Começo da rodada	Após SubByte	Após ShiftRow	Após MixColumn		Chave da Rodada	
Texto Claro	00 04 08 0c 01 05 09 0d 02 06 0a 0e 03 07 0b 0f				⊕	00 44 88 cc 11 55 99 dd 22 66 aa ee 33 77 bb ff	=
1	00 40 80 c0 10 50 90 d0 20 60 a0 e0 30 70 b0 f0	63 09 cd ba ca 53 60 70 b7 d0 e0 e1 04 51 e7 8c	63 09 cd ba 53 60 70 ca e0 e1 b7 d0 8c 04 51 e7	5f 57 f7 1d 72 f5 be b9 64 bc 3b f9 15 92 29 1a	⊕	c0 84 0c c0 39 6c f5 28 34 52 f8 16 78 0f b4 4b	=
2	9f d3 fb dd 4b 99 4b 91 50 ee c3 ef 6d 9d 9d 51	db 66 0f c1 b3 ee b3 81 53 28 2e df 3c 5e 5e d1	db 66 0f c1 ee b3 81 b3 2e df 53 28 d1 3c 5e 5e	7b e1 8b 21 bf 5d bd 9a 01 34 ca c0 0f be 7f 7f	⊕	f6 72 7e be 7e 12 e7 cf 87 d5 2d 3b c2 cd 79 32	=
3	8d 93 f5 9f c1 4f 5a 55 86 e1 e7 fb cd 73 06 4d	5d dc e6 db 78 84 be fc 44 f8 94 0f bd 8f 6f e3	5d dc e6 db 84 be fc 78 94 0f 44 f8 e3 bd 8f 6f	5a c8 03 b2 0a 17 46 57 d4 a0 18 f9 2a af 8c 28	⊕	78 0a 74 ca 9c 8e 69 a6 a4 71 5c 67 6c a1 d8 ea	=
4	22 c2 77 78 96 99 2f f1 70 d1 44 9e 46 0e 54 c2	93 25 f5 bc 90 ee 15 a1 51 3e 1b 0b 5a ab 20 25	93 25 f5 bc ee 15 a1 90 1b 0b 51 3e 25 5a ab 20	2a 24 f3 d6 5c 48 f4 e5 24 c8 10 30 11 c5 b9 31	⊕	54 5e 2a e0 19 97 fe 58 23 52 0e 69 18 b9 61 8b	=
5	7e 7a d9 36 45 df 0a bd 07 9a 1e 59 09 7c d8 ba	f3 da 35 05 6e 9e 67 7a c5 b8 72 cb 01 10 61 f4	f3 da 35 05 9e 67 7a 6e 72 cb c5 b8 f4 01 10 61	c2 cc 31 61 b6 53 85 6b 8e 33 ee a3 11 db c0 1b	⊕	2e 70 5a ba e0 77 89 d1 1e 4c 42 2b f9 40 21 aa	=
6	ec bc 6b db 56 24 0c ba 90 7f ac 88 e8 9b e1 b1	ce 65 7f b9 b1 36 fe f4 60 d2 91 c4 9b 14 f8 c8	ce 65 7f b9 36 fe f4 b1 91 c4 60 d2 c8 9b 14 f8	84 8c 8d 8b c2 4e 38 55 82 be 77 a4 65 b8 3d 58	⊕	30 40 1a a0 11 66 ef 3e b2 fe bc 97 0d 4d 6c c6	=

Nº da rodada	Começo da rodada	Após SubByte	Após ShiftRow	Após MixColumn		Chave da Rodada	
7	b4 cc 97 2b d3 28 d7 6b 30 40 cb 33 68 f5 51 9e	8d 4b 88 f1 66 34 0e 7f 04 09 1f c3 45 e6 d1 0b	8d 4b 88 f1 34 0e 7f 66 1f c3 04 09 0b 45 e6 d1	49 02 68 8b cf 4c 9c f7 9a 17 ce ed b1 9a 2f de	⊕	c2 82 98 38 99 ff 10 2e 06 f8 44 d3 ed a0 cc 0a	=
8	8b 80 f0 b3 56 b3 8c d9 9c ef 8a 3e 5c 3a e3 d4	3d cd 8c 6d b1 6d 64 35 de df 7e b2 4a 80 11 48	3d cd 8c 6d 6d 64 35 b1 7e b2 de df 48 4a 80 11	fb d5 02 dc 2d b2 1f 7f 74 08 85 4a c4 0e 7f fb	⊕	73 f1 69 51 ff 00 10 3e 61 99 dd 0e ea 4a 86 8c	=
9	88 24 6b 8d d2 82 0f 41 15 91 58 44 2e 44 f9 77	c4 36 7f 5d b5 13 76 83 59 81 6a 1b 31 1b 99 f5	c4 36 7f 5d 13 76 83 b5 6a 1b 59 81 f5 31 1b 99	39 dc 22 66 a9 c6 92 2d 07 25 63 41 df 55 6d fa	⊕	da 2b 42 13 54 54 44 7a 05 9c 41 4f 3b 71 f7 7b	=
10	e3 f7 60 75 fd 92 d6 57 02 b9 22 0e e4 24 9a 81	11 68 d0 9d 54 4f f6 5b 77 56 93 ab 69 36 b8 0c	11 68 d0 9d 4f f6 5b 54 93 ab 77 56 0c 69 36 b8		⊕	36 1d 5f 4c d0 84 c0 ba 24 b8 f9 b6 46 37 c0 bb	=
Saída	27 75 8f d1 9f 72 9b ee b7 13 8e e0 4a 5e f6 03						

Bibliografia

[ABK98] R. A. Anderson, E. Biham, L. R. Knudsen, "Serpent", Proc. Of the 1st AES candidate conference, CD-1: Documentation, Agosto 1998, Ventura.

[AES00] 3ª conferência de candidatos à AES (AES3), abril de 2000, Nova York-EUA, disponível em < http://csrc.nist.gov/CryptoToolkit/aes/round2/conf3/aes3conf.htm>.

[AkGi01] M.-L. Akkar, C. Giraud, "An Implementation of DES and AES, secure against some attacks" , Cryptographic Hardware and Embedded Systems CHES 2001, LNCS 2162, Ç. K. Koç, D. Naccache, C. Paar, Eds., Springer-Verlag, 2001, pp. 315-324.

[BaSo03] S. A. P. de Barros, R. M. C. de Souza, "Uma Função Unidirecional Hash Usando o Rijndael", 26° CNMAC, setembro 2003, São José do Rio Preto – SP.

[Baud99] O. Baudron, H. Gilbert, L. Granboulan, H. Handshuh, A. Joux, P. Nguyen, F. Noilham, D. Pointcheval, T. Pornin, G. Poupard, J. Stern, S. Vaudenay, "Report on AES candidates," proc. 2ª conferência de candidatas à AES, março de 1999, Roma-Itália, pp. 53-67.

[BCDG98] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas , L. O'Connor, M. Peyravian, D. Safford, N. Zunic, "MARS – A Candidate Cipher for AES", NIST AES proposal, junho de 1998.

[Bigg89] Norman L. Biggs, "Discrete Mathmatics", revised edition, Oxford Science Publications, 1989.

[BiSh91] E. Biham, A.Shamir, "Differential Cryptanalysis of DES-like Cryptosystems", Journal of Cryptology, Vol. 4, nº 1, pp. 3-72,1991.

[Biha94] E. Biham, "New Types of Cryptanalytic Attacks Using Related Keys", Advances in Cryptology, Proc. Eurocrypt '93, LNCS 765, T. Helleseth, Ed. S´ringer-Verlag, 1994, pp 398-409.

[BiSh01] A. Biryukov, A. Shamir, "Structural cryptanalysis of SASAS", Advances in Cryptology, Proc. Eurocrypt'01, LNCS 2045, B. Pfitzmann, Ed. , Springer-Verlag, 2001, pp. 394-405.

[Burt98] D. M. Burton, "Elementary Number Theory",4ª Edição, McGraw-Hill, 1998.

[CJR99] S. Chari, C. Jutla, J. R. Rao, P. J. Rohatgi, "A Cautionary Note Regarding Evaluating of AES Candidates on Smart Cards", Proc. of the 2nd AES candidates conference, Março 1999, Roma, pp. 133-150.

[CoPi02] N. T. Courtois, J. Pieprzyk, "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations", apresentado na AsiaCrypt 2002, disponível em : <http://eprint.iacr.org/2002/044>.

[Daem95] J. Daemen, "Cipher and Hash Function Design Strategies Based on Linear and Differential Cryptanalysis", Doctoral Dissertation, Katholieke Universiteit Leuven, março 1995.

[Daem99] J. Daemen, V. Rijmen, "Resistence Against Implementions Attacks: a Comparative Study of AES proposals", Proc. of the 2nd AES candidate conference, março de 1999, Roma, pp 13-27.

[DaRi99] Joan Daemen, Vincent Rijmen, "AES proposal: The Rijndael Block Cipher", http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaeldocV2.zip, (1999).

[DaRi02] J. Daemen, V. Rijmen, "The Design of Rijndael:AES - The Advanced Encription Standard", Springer Verlag, Berlin, 2002.

[Davi83] D. W. Davies, "Some Regular Properties of the DES", Advances in Cryptology, Proc. Crypto '82, D. Chaum, R. Rivest e A. Sherman, Eds., Plenum Press, 1983, pp 89-96.

[Denn82] D.E. R. Denning, "Cryptography and Data Security", Addison-Wesley, 1982.

[DKR97] J. Daemen, L. R. Knudsen, V. Rijmen, " The Block Cipher Square", Fast Software Encryption '97, LNCS 1267, E. Biham, Ed. , Springer-Verlag, 1997, pp. 149-165.

[DPA01] J. Daemen, M. Peeters, G. Van Assche, "Bitslice ciphers and Implementations Attacks", Fast Software Encryption 2000, LNCS 1978, B. Schneier, Ed. , Springer-Verlag, 2001, pp 134-149.

[Feis73] H Feistel, "Cryptography and Computer Privacy", Scientific American v.228, n. 5, Maio 1973, pp 15-23.

[FIPS-180-2] NSA-NIST, "*Secure Hash Standard-SHS* ", Publicação 180-2, agosto 2002, EUA.

[FIPS-197] Advanced Encryption Standard (AES), Informação Federal de Processamento de Padrão (FIPS), Publicação 197, Instituto Nacional de Padrões e Tecnologia (NIST), novembro de 2001, EUA.

[FKSS01] N. Ferguson, J. Kelsey, B. Schneier, M. Stay, D. Wagner, D. Whiting, "Improved cryptanalysis of Rijndael", *Fast Software Encryption 2000*, Ed. Springer-Verlag, 2001.

[FSW01] N. Ferguson, R. Schroeppel, D. Whiting, "A Simple Algebraic Representation of Rijndael", Selected Areas in Cryptography '01, disponível em: <http://www.macfergus.com/niels/pubs/rdalgeq-draft.pdf >.

[GiMi00] H. Gilbert, M. Minier, " A Collision Attack on 7 Rounds of Rijndael ", Proc. of the 3rd AES candidates conference, abril de 2000, Nova York, pp 230-241.

[JaKn97] T. Jakobsen, L. R. Knudsen, "The Interpolation Attack on Block Ciphers", Fast Software Encryption '97, LNCS 1267, E. Biham, Ed. Springer-Verlag, 1997.

[KeSc96] J. Kelsey, B. Schneier, D. Wagner, "Key-schedule cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES", Advances in Cryptology, Proc. Crypto '96 , LNCS 1109, N. Koblitz, Ed. Springer-Verlag, 1996, pp 237-252.

[KJJ99] P. Kocher, J. Jaffe, B. Jun,"Differential Power Analysis", Advances in Cryptology, Proc. Crypto '99, LNCS 1666, M. Wiener, Ed. Springer-Verlag, 1999, pp 388-397.

[Knud95] L. R. Knudsen, "Truncated and Higher Order Diferentials", Faster Software Encryption '94, LCNS 1008, B. Preneel, Ed. , Springer-Verlag, 1995, pp. 196-211.

[Koch96] P. Kocher, "Timing Attacks in implementations of Diffie-Hellman, RSA, DSS, and other systems", Advances in Cryptology, Proc. Crypto '96, LNCS 1109, N. Koblitz, Ed. Springer-Verlag, 1996, pp 104-113.

[Lai92] Xuejia Lai, "On the Design and Security of Block Chiphers", Vol. 1, Doctoral Dissertation, ETH, 1992.

[Lidl98] Rudolf Lidl, Günter Pilz, "Applied Abtract Algebra", 2ª edição, Springer-Verlag, 1998, Nova York.

[Luck00] S. Lucks, "Attacking 7 Rounds of Rijndael under 192-bt and 256-bit keys", Proc. Of the 3rd Aes candidate conference, April 13-14, 2000, New York, pp 215-229.

[Luck01] S. Lucks, "The saturation attack – a bait for Twofish", Fast Software Encryption 2001, LNCS, M. Matsui, Ed. , Springer-Verlag, a ser publicado.

[Mats94] M. Matsui, "Linear Cryptanalysis Method for DES Cipher", Advances in Cryptology – EUROCRYPT '93 (Lecture Notes in Computer Science nº 765), Springer-Verlag, pp.386-397,1994.

[Mess01] T. S. Messerges, "Securing the AES Finalists Against Power Analysis Attacks", Fast Software Encryption 2000, LNCS 1978, B. Schneier, Ed., Springer-Verlag, 2001, pp 150-164.

[MMO85] S.M. Matyas, C. H. W. Meyer, B. Oseas, "Generating Strong Block Ciphers with Criptographic Algorithm", IBM Technical Disclosure Bulletin, vol.27, pp. 5658-5659, 1985.

[NBBD99] J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, E. Roback, "Status report on the first round of the development of the Advanced Encryption Standard", agosto de 1998,disponível em <http://www.nist.gov/aes>.

[Nech99] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, E. Roback, "Report on the Development of the Advanced Encryption Standard (AES)" disponível no endereço eletrônico <http://www.nist.gov/AES>.

[Pren93] B. Preneel, "Analysis and Design of Cryptographic Hash Functions", Katholieke Universiteit Leuven, janeiro 1993.

[RDP96] V. Rijmen, J. Daemen, B. Preneel, "The Cipher SHARK", Fast Software Encryption '96, LNCS 1039, D. Gollmann, Ed. Springer-Verlag, 1996, pp 99-111.

[Rive92] R.L. Rivest, "The MD5 Message Digest Algoritm," RFC 1321, abril 1992.

[RRSY98] R. L. Rivest, M. J. B. Robshaw, R. Sidney, W. L. Yin, "The RC6 Block Cipher", Proc. da 1ª conferência de candidatas a AES, CD-1: Documentation, agosto 1998, Ventura.

[Schn96] Bruce Schneier, "Applied Cryptography", 2ª Edição, John Wiley & Sons, 1996, seção 18.14 p.455-459, USA.

[Shan49] C. E. Shannon, "Communication Theory of Secrecy Systems", Bell Syst. Tech. Journal, Vol. 28, pp. 656-715, 1949.

[SKWW98] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, "Twofish: A 128 bits block cipher", NIST AES proposal, junho de 19 98.

[Wint84] R. S. Winternitz, "Producing One-Way Hash Functions from DES" Advances in Cryptology – Preceedings of Crypto 83, Plenum Press, 1984.

Biografia de Galois

Évariste Galois**,** foi um matemático francês que nasceu num vilarejo, ao sul de Paris, hoje chamado Bourg-la-Reine, no dia 25 de outubro de 1811.

Os pais de Galois eram Nicholas Gabriel Galois e Adelaide Marie Demante. Ele, um homem culto e politizado, foi Prefeito de Bourg-la-Reine, durante os reinados de Napoleão e Louís XVIII. Fora da política, seu maior interesse foi a composição de versos satíricos que ele lia nas reuniões da cidade. Ela, com formação em filosofia, literatura clássica e religião, em suas horas de descanso, se dedicava com exclusividade a ensinar grego, latim e religião a seu filho até quando ele completou doze anos.

Sua infância foi intensamente influenciada pelas idéias liberais de seu pai no campo da política e pelas tendências legítimas e cristãs de sua mãe. A França vivia um estado de agitação política. Napoleão Bonaparte havia fracassado em sua campanha na Rússia, e em 1814, foi exilado e abdicou em favor do Rei Louís XVIII. Em 1815 ele fugiu da ilha Elba e entrou em Paris retomando o poder. Todo este contexto despertou um sentimento ainda mais republicano em Galois.

Em 6 de outubro de 1823, Galois ingressa no internato Lycée de Louis-le-Grand na 4ª série. Em 1826, Galois apresentou uma deficiência de aprendizado chegando a repetir o ano. Ele não era um bom aluno do internato, no entanto, teve a oportunidade de ler alguns trabalhos dos matemáticos Lagrange e Legendre. Nestes trabalhos descobriu que a sua vocação era voltada para a matemática. A partir desse momento, manifestou o firme propósito de ingressar na Écòle Polytechnique, em que Cauchy ministrava o curso e onde nos ensinamentos reinava um ambiente revolucionário de tendências republicanas.

Em fevereiro de 1827, já com quase dezessete anos, Galois começou a fazer o seu primeiro curso de matemática e logo ficou entusiasmado, chegando a ponto de não só ser elogiado pelo seu professor como também pelo diretor da escola.

Em 1828, Galois se submeteu ao exame na célebre Écòle Polytechnique – fundada por Napoleão Bonaparte e responsável pela criação dos melhores engenheiros da França – porém não obtém bons resultados em face de não estar preparado nas outras matérias.

A sua reprovação não o impediu de continuar estudando porque o seu sonho era entrar na referida escola. A partir de então, passou a assistir às aulas, como aluno-ouvinte, na Écòle Polytechnique ministradas pelo matemático francês Louis Paul Émile Richard, que reconheceu o gênio precoce e lhe facilitou o acesso à leitura dos trabalhos contemporâneos de Abel, Cauchy, Gauss e Jacobi.

O desejo de Galois pela matemática era tão grande que trabalhava incessantemente em suas pesquisas, deixando um pouco de lado os trabalhos da escola e superando a capacidade do seu professor. Passou então a estudar diretamente dos livros escritos pelos gênios de sua época e rapidamente absorveu os conceitos mais modernos.

Em abril de 1829, Galois publica o seu primeiro trabalho sobre frações contínuas nos "Annales de Mathématiques". Naquele ano e le submeteu vários artigos relacionados com a solução de equações algébricas à "Académie des Sciences" e que Cauchy foi designado para julgar. Esses artigos continham soluções tão sofisticadas e inovadoras que seus professores não conseguiam julgá-las corretamente.

Durante esse período um novo sacerdote chegou ao vilarejo de Bourg-la-Reine, onde o pai de Galois ainda era prefeito. O sacerdote não gostando das poesias satíricas e conceituosas de Nicholas Gabriel Galois, como também, de suas tendências republicanas, começou a fazer uma campanha para depô-lo. Uma das táticas usadas pelo sacerdote era a de escrever versos vulgares ridicularizando membros da comunidade e assinando-os com o nome do prefeito. Nicholas não suportando a pressão de ter gerado um verdadeiro escândalo no seio da comunidade, cometeu suicídio no dia 2 de julho de 1829.

Algumas semanas após a morte do pai, Galois, um jovem de temperamento agressivo, ainda abalado com a tragédia se apresentou, mais uma vez, para exame na Écòle Polytechnique. Tendo certeza de sua reprovação na prova oral e sentindo uma frustração por sua

inteligência não estar sendo reconhecida, ele perdeu a calma e jogou um apagador no examinador Monsieur Dinet. Isto foi o suficiente para afastá-lo definitivamente das famosas salas do mais prestigiado colégio do país.

Apesar desses infelizes acontecimentos, Galois teve forças suficientes para continuar seus trabalhos e com suas pesquisas na busca de soluções para equações do quinto grau.

Em 29 de dezembro de 1829, Galois é admitido na Écòle Normale Supérieure (Escola Normal Superior). Tinha como idéia principal enfocar as noções de grupo e de corpo, estabelecendo uma correlação entre as teorias dos grupos e a das equações algébricas. Ele evidenciou a importância do conceito de grupo na resolução de equações. Demonstrou que a dificuldade da resolução de uma equação não depende do seu grau, mas do grupo correspondente. Sua teoria permitiu-lhe determinar as condições necessárias e suficientes para que uma equação tenha solução, e estabelecer a impossibilidade da resolução algébrica das equações gerais de grau superior a quatro.

Em fevereiro de 1830, Galois envia a Cauchy trabalhos adicionais sobre as teorias das equações. Cauchy ficou tão impressionado com os trabalhos de Galois, que julgou-o capaz para participar da competição do Grande Prêmio de Matemática da Academia, bastando, para isso, juntar os dois trabalhos e remetê-los em uma única memória. Assim sendo, Cauchy devolveu-os para Galois e ficou aguardando que ele se inscrevesse.

Galois, voltando para Paris, pois tinha ido assistir ao funeral do seu pai, imediatamente juntou os dois trabalhos em uma única memória e remeteu-os ao então secretário da Academia, Joseph Fourier que morrera algumas semanas antes da decisão dos juízes.

Em abril do mesmo ano, após pesquisar os trabalhos de Abel e de Jacobi sobre a teoria das funções elípticas e integrais abelianas, Galois publicou três artigos no “Bulletin de Férussac” com o apoio de Jacques Sturn. Em junho Galois tomou conhecimento que o seu trabalho fora premiado pela academia, juntamente com o de Abel, que falecera exatamente um ano antes.

Em dezembro de 1830, o Diretor da Écòle Normale Supérieure, Monsieur Guigniault, um monarquista, escreveu vários artigos nos jornais atacando os estudantes da escola, que na maioria eram radicais republicanos. Galois, em resposta aos seus artigos, redigiu um manifesto violento, acusando o diretor de reacionário. Sem causar surpresa, o diretor da escola expulsou Galois e com isso a sua carreira matemática chegou ao fim. Ele então se uniu à Artilharia da Guarda Nacional, um ramo da milícia conhecido também como "Amigos do Povo". Em 31 de dezembro daquele ano o Rei Louis-Phillipe I, por achar ser uma ameaça ao trono, aboliu a Artilharia da Guarda Nacional através de Decreto Real.

A partir da abolição da Guarda Nacional, começaram as perseguições aos ex-membros. Galois, apesar de estar bastante apreensivo com os acontecimentos, ainda teve tempo de publicar. Foi convidado por Poisson para encaminhar, numa terceira tentativa, a notável memória "Sur la resolution génerale des équations" (Sobre a resolução geral das equações) para a Academia e ele assim o fez no dia 17 de janeiro de 1831.

Em 09 de maio de 1831, dezenove oficiais da Artilharia da Guarda Nacional – que foram presos, numa tarde de 1830, por conspiração para derrubar o governo – comemoravam num jantar a absolvição juntamente com mais duzentos republicanos. Durante o jantar, Galois, um dos mais ardentes republicanos, ergueu o seu cálice e com um punhal na mão fazia ameaças contra o Rei Louis-Phillipe I.

Após alguns dias, Galois foi detido e levado para a prisão de Saint-Pélagie onde passou um mês. Durante esse período, foi acusado de ameaçar a vida do Rei e, em conseqüência, foi levado a julgamento. O júri, formado por jovens de sua idade, o absolveu porque devido ao barulho durante o jantar em que participara Galois, não se ouviu qualquer ameaça direta contra o Rei.

Em 14 de julho de 1831, dia da Bastilha, Galois, em gesto de desafio, usando um uniforme da proscrita Artilharia da Guarda Nacional marchou nas ruas de Paris. No dia seguinte foi preso e levado para Saint-Pélagie onde ficou recluso e, em 23 de outubro de 1831, foi condenado à prisão por seis meses. Durante esse período, recebeu a notícia da rejeição de sua memória, pois segundo informação de Poisson, a redação era demasiadamente concisa,

tornando a leitura incompreensível. No entanto, Poisson o encorajou no sentido de que ele publicasse uma memória mais completa.

Algumas semanas após a sua prisão um fato interessante aconteceu: um atirador que se encontrava do lado oposto da cadeia, disparou um tiro contra a sua cela e um outro prisioneiro que estava a seu lado foi atingido. A partir deste instante, Galois ficou com bastante temor, pois, concluíra que a bala que feriu o seu colega de prisão seria para ele, talvez até para intimidá-lo. A situação de Galois era a pior possível e ele ficou muito atemorizado diante do acontecimento. Isolado dos seus amigos e da família, com um sentimento de rejeição devido à notícia de que suas idéias matemáticas não tinham sido aceitas, Galois entra num estado de depressão. Ele passou a beber, chegando a ponto de tentar o suicídio com um punhal.

Em março de 1832, um mês antes do final da condenação, uma epidemia de cólera se alastrou por Paris e os prisioneiros, inclusive Galois, foram transferidos para a prisão de Sieur Faultrier. Na prisão conheceu Stéphanie-Félice Poterine du Motel, filha de um médico residente, e por ela se apaixonou. Depois da sua libertação, ocorrida no dia 29 de abril de 1832, começou a trocar cartas com Stéphanie. Ele usou este método porque ela estava comprometida com Pescheux d' Herinville que posteriormente descobriu a infidelidade de sua noiva.

D' Herinville não teve outra alternativa, a não ser desafiar Galois para um duelo que foi marcado para a manhã do dia seguinte. Na noite que precedeu ao duelo, endereçou um manifesto político ' À tous les républicains ' (a todos os republicanos) aos seus amigos e companheiros com o seguinte teor:

"... não me censurem se eu morrer por outro motivo que não pelo meu país. Será que eu vou morrer por uma coisa tão insignificante e vergonhosa? Peço a Deus que testemunhe, pois eu fui coagido e cedi à provocação, pois, fiz de tudo para evitar."

Em seguida, Galois escreveu uma carta ' Lettre à Auguste Chevalier ' (Carta a Auguste Chevalier), solicitando ao amigo que submetesse os seus trabalhos à apreciação de Gauss e Jacobi.

Os ensaios matemáticos foram um resumo não só sobre o que diz a carta, mas também, a teoria das equações algébricas, bem como conclusões sobre as integrais abelianas, sua classificação e seus períodos, resultados estes que foram estabelecidos vinte e cinco anos após, por Bernhard Riemann.

Como fora combinado, na manhã de uma quarta-feira do dia 30 de maio de 1832, Galois e D' Herbinville se enfrentaram armados com pistolas e, em alguns segundos, Galois, foi atingido no estômago por uma bala. Algumas horas após a tragédia, o irmão de Galois, Alfred, chega ao local e leva-o para o hospital Cochin.

No dia 31 de maio de 1832, Galois não resistindo às dores abdominais devido à peritonite, faleceu. O seu funeral só aconteceu no dia 02 de junho com um cerimonial reunindo cerca de dois mil republicanos e que terminou com vários tumultos, durando vários dias.

Após a sua morte, o seu irmão juntou-se a Auguste Chevalier e copiaram os ensaios matemáticos. Em setembro de 1832 enviaram-nos a Gauss, Jacobi e outros matemáticos através da "Revue Encyclopédique" (revista enciclopédica). Apesar disto, esses trabalhos não foram reconhecidos durante os dez primeiros anos após sua morte.

Em 1846, uma cópia dos trabalhos de Galois chegou às mãos do matemático francês Joseph Liouville que reconheceu esse magnífico trabalho, editou e publicou parte de seus manuscritos no "Journal de Mathématiques Pures et Appliquées" (Jornal de Matemáticas Puras e Aplicadas).

Galois tinha de fato formulado uma completa explicação de como se poderiam obter soluções para equações do quinto grau. Ademais, estabeleceu as condições sob as quais uma equação algébrica admite uma solução por radicais, e é por isso que são chamados grupos solúveis; examinou as equações de grau maior do que cinco, identificando as que tinham soluções.

Galois foi considerado o verdadeiro iniciador da teoria de grupos. Deve-se a ele o termo "grupo", o desenvolvimento da teoria dos corpos, como também a primeira idéia da representação linear dos grupos.