MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

EDUARDO MARSOLA DO NASCIMENTO

ALGORITMO DE CRIPTOGRAFIA LEVE COM UTILIZAÇÃO DE AUTENTICAÇÃO

Rio de Janeiro
2017

INSTITUTO MILITAR DE ENGENHARIA

EDUARDO MARSOLA DO NASCIMENTO

ALGORITMO DE CRIPTOGRAFIA LEVE COM UTILIZAÇÃO DE AUTENTICAÇÃO

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

Orientador: Prof. José Antônio Moreira Xexéo – D.Sc.

Rio de Janeiro
2017

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

005.82 Nascimento, Eduardo Marsola do
N244a Algoritmo de criptografia leve com utilização de autenticação / Eduardo Marsola do Nascimento, orientado por José Antônio Moreira Xexéo – Rio de Janeiro: Instituto Militar de Engenharia, 2017 .

114p.: il.

Dissertação (Mestrado) - Instituto Militar de Engenharia, Rio de Janeiro, 2017 .

1. Curso de Sistemas e Computação - teses e dissertações. 2. Criptografia. I. Xexéo, José Antônio Moreira. II. Título. III. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

EDUARDO MARSOLA DO NASCIMENTO

ALGORITMO DE CRIPTOGRAFIA LEVE COM UTILIZAÇÃO DE AUTENTICAÇÃO

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

Orientador: Prof. José Antônio Moreira Xexéo – D.Sc.

Aprovado em 28 de Abril de 2017 pela seguinte Banca Examinadora:

Prof. José Antônio Moreira Xexéo - D.Sc. do IME – Presidente

Prof. Anderson Fernandes Pereira dos Santos - D.Sc. do IME

Prof. Flávio Luis de Mello - D.Sc. da UFRJ

William Augusto Rodrigues de Souza - Ph.D. do Royal Holloway University of London

Rio de Janeiro
2017

À Laura que ilumina os meus dias com seu lindo sorriso

AGRADECIMENTOS

Agradeço primeiramente a Deus, sua imensa graça permitiu que meu caminho fosse repleto de realizações.

À minha esposa Camila que sempre me apoiou nessa caminhada, principalmente quando as dificuldades pareciam intransponíveis. À minha filha Laura que, apesar da pouca idade, sempre compreendeu quando o papai dizia que não poderia brincar naquele momento porque precisava estudar.

Ao meu pai Jorge que desembarcou do trem da vida antes de ver esta realização, mas, onde quer que esteja, está feliz com o resultado. A minha mãe Cida que sempre acreditou em mim, mesmo quando era um menino que não acreditava em si mesmo. As minhas irmãs Luciana e Cristiane, que me ensinaram que a vida é melhor quando podemos compartilhar tudo com quem amamos.

Ao prof. Xexéo, pela orientação e comentários diretos, que tanto ajudaram que a pesquisa não desviasse do rumo. A Petrobras por permitir que eu dedicasse o precioso tempo a esta pesquisa e ao IME, seus professores e funcionários que direta ou indiretamente contribuíram a este trabalho.

“How can the net amount of entropy of the universe be massively decresed ? Multivac fell dead and silent. The slow flashing lights ceased, the distant sounds of clicking relays ended. Then, just as frigthened technicians felt they could hold their breath no longer, there was a sudden springing to life of the teletype attached to that portion of Multivac. Five words were printed:
INSUFFICIENT DATA FOR MEANINGFUL ANSWER.”
ISAAC ASIMOV

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SÍMBOLOS

ABREVIATURAS

D.Sc.	Doutor
Ex.	Exemplo
M.Sc.	Mestre
Ph.D.	Doutor
Prof.	Professor

SÍMBOLOS

ϵ	Tendência de uma expressão
$\emptyset$	Conjunto vazio
$i_1 \% i_2$	Operação algébrica que retorna o resto da divisão ds números inteiros i_1, i_2
$b_1 \oplus b_2$	Operação lógica OU exclusivo (XOR) entre b_1, b_2
$s_1 \mid s_2$	Retorna a concatenação das sequências (strings) s_1, s_2
$\prod_{i=1}^{n} X_i$	Produtório da expressão X_i: $(X_0) \times (X_1) \times \ldots \times (X_n)$
$CIPH_K$	Função de cifrar utilizando a chave K
$CIPH_K^{-1}$	Função de decifrar utilizando a chave K
P	Bloco de texto claro
PF	Função de permutação
C	Bloco de texto cifrado
K	Chave de criptografia
tag	Etiqueta de validação

LISTA DE SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
AD	Associate Data
AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associate Data
AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
CBC-MAC	Cipher Block Chaining – Message Authentication Code
CCA	Chosen Ciphertext Attack
CCM	Couter with CBC-MAC
CFB	Cipher Feedback
CTR	Counter
DEA	Data Encryption Algorithm
DES	Data Encryption Standard
ECB	Eletronic Code Book
FDK	Factory Defined Key
GCM	Galois/Counter Mode
HMAC	Hash Based MAC
HTTPS	Hypertext Transfer Protocol Secure
IAPM	Integrity Aware Parellized Mode
IV	Initialization Vector
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IME	Instituto Militar de Engenharia

IoT	Internet of Things (Internet das Coisas)
IPSEC	Internet Protocol Security
ISO	International Standards Organization
KDS	Key Distribution Server
LTK	Long Term Key
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
NMAC	Nested MAC
OFB	Output Feedback
PKI	Public Key Infrastruture
SCADA	Supervisory Control and Data Acquisition (controle de supervisão e aquisição de dados)
SK	Session Key
SSH	Secure Shell
SSL	Secure Sockets Layer
TDEA	Triple Data Encryption Algorithm
TLS	Transport Layer Security
WPA2	Wi-Fi Protected Access 2

RESUMO

O ambiente da Internet das coisas (IoT) requer o uso de técnicas criptográficas que sejam leves, com baixo custo computacional. Os algoritmos existentes além de serem pouco flexíveis, normalmente fornecem autenticação por técnicas externas ao algoritmo, como os modos de operação. Este trabalho apresenta a cifra FlexAE, um novo algoritmo leve, flexível e autenticado para ser utilizado no ambiente de IoT. Ele trabalha com tamanhos variáveis de bloco (64×2^x bits, onde $x \geq 0$) e chave (128×2^x bits, onde $x \geq 0$). O algoritmo foi construído visando consumir poucos recursos computacionais. A base da cifra é a construção Even-Mansour (EVEN; MANSOUR, 1997), que utiliza duas operações XOR e uma permutação, e o modo paralelizável que considera integridade (IAPM) (JUTLA, 2001). A função de permutação também foi definida tomando o cuidado para permitir o uso de diversos tamanhos de blocos e chave, tornando a cifra flexível. A cifra foi validada quanto a capacidade de gerar sequência pseudo-aleatórias, leveza e resistência de ataques criptanálise diferencial e linear. Os testes estatísticos utilizados para validar a aleatoriedade das sequencias geradas foram o conjunto de testes estatísticos do NIST(RUKHIN *et al.*, 2010) e a ferramenta DieHarder (BROWN; EDDELBUETTEL; BAUER, 2016). Para confirmar que a cifra é leve, ela foi comparada com outras cifras utilizando as métricas da ferramenta FELICS (DINU *et al.*, 2015) - uso de memória RAM, tempo de processamento e tamanho de código. A dificuldade dos ataques de criptanálise diferencial e linear da cifra indica que a cifra é segura quando utiliza blocos de 64 e 128 bits. Para blocos maiores que 128 bits, a cifra é segura para trabalhar com chaves de até 256 bits. O trabalho também apresentou um sistema que permite a comunicação segura entre dispositivos em um ambiente IoT, utilizando a cifra proposta.

ABSTRACT

The Internet of Things (IoT) environment requires lightweight cryptographic techniques, with low computational costs. The existent algorithms have little flexibility and normally provide authentication through external techniques, like the modes of operation. This work presents the FlexAE cipher, a new algorithm that is lightweight, flexible and authenticated. It works with variable size for blocks (64×2^x bits, where $x \geq 0$) and keys (128×2^x bits, where $x \geq 0$). The algorithm was designed aiming to require low computational resources. The cipher is bases on Even-Mansour construction (EVEN; MANSOUR, 1997), that uses only two XOR operations and a permutation, and the Integrated Aware Parallelizable Mode (IAPM) (JUTLA, 2001). The permutation function was also defined taking care to allow variable size blocks and keys, making the cipher flexible. The cipher was validated in relation to its capacity to generate pseudo-random sequences, lightweight and resistance against differential and linear cryptanalises attacks. The statistical tests used to validate the sequences randomness generate were the NIST Statistical Tests Suite (RUKHIN *et al.*, 2010) and the DieHarder tool (BROWN; EDDELBUETTEL; BAUER, 2016). To assure the cipher is lightweight, it was compared to other ciphers using the FELICS (DINU *et al.*, 2015) framework metrics – RAM use, process time and code size. The differential and linear complexity calc indicated that the cipher is secure when using 64 and 128 block sizes. For larger than 128 bits block sizes, the cipher is secure to work with keys up to 256 bits. This work also presented a system that allows secure communication among devices on a IoT environment, using the proposed cipher.

1 INTRODUÇÃO

A criptografia tem como uma das suas principais funções proteger a comunicação entre dispositivos. Hoje, espera-se que as pessoas, que utilizam serviços bancários pela Internet, verifiquem se o cadeado do navegador está fechado antes de colocar suas senhas. Ele é um indicativo que a senha e os dados bancários passarão pela rede mundial de computadores de forma protegida. A senha estará cifrada de uma maneira que dificulta a leitura por terceiros, aumentando a confidencialidade.

Além disso, os protocolos e algoritmos criptográficos utilizados atualmente evitam que a mensagem possa ser falsificada. Um algoritmo que visa proteger apenas a confidencialidade não evita a adulteração da mensagem. Se o atacante sabe em que posição da mensagem cifrada está uma determinada informação, como o número de conta de um destinatário válido de uma transferência bancária, o atacante pode modificar o criptograma para que a mensagem decifrada contenha o número da sua conta. Apenas calcular um dígito verificador ou '*hash'* não evitaria o ataque. Se o algoritmo de '*hash*' utilizado é conhecido, é possível capturar uma mensagem, alterar o seu conteúdo e recalcular o '*hash*'. Para evitar o ataque, torna-se necessária uma técnica que, além da integridade, garanta a autenticidade da mensagem.

Para prover essa proteção, existe um grande custo computacional. Para computadores pessoais modernos, o impacto desse processamento é mínimo. Para celulares inteligentes (*smartphones*) e *tablets*, o processamento extra implica em um maior consumo de bateria e, em alguns casos, lentidão. O processamento gasto com a criptografia pode impactar significativamente a funcionalidade de um equipamento, se o hardware for muito limitado, como, por exemplo: em um sensor de presença de um sistema de alarme; em um atuador que liga uma mangueira no jardim; em uma bomba injetora de combustível de um automóvel; em um medidor de temperatura de um aquário; em uma enorme quantidade de dispositivos simples; entre outros.

Este trabalho busca especificar uma cifra que garanta confidencialidade, integridade e autenticidade de uma mensagem com um custo computacional baixo, se comparado a técnicas existentes.

1.1 MOTIVAÇÕES

No passado, não havia preocupação em cifrar a comunicação de dispositivos como atuadores e sensores. Eles operavam, na maioria das vezes, de forma independente. Mesmo quando operavam em rede, como nos sistemas SCADA de automação, a rede de automação era segregada da Internet e da rede corporativa por firewalls e a criptografia não era utilizada pelo custo computacional (IGURE; LAUGHTER; WILLIAMS, 2006).

Atualmente, a tendência é conectar esses dispositivos diretamente à Internet, permitindo que eles interajam entre si e que sejam acessados e controlados pelos seus proprietários de forma integrada. Isso está causando uma transformação na Internet e nos próprios dispositivos. Espera-se que sejam aproximadamente 24 bilhões de equipamentos conectados a Internet (GUBBI *et al.*, 2013). Essa é a Internet das Coisas (*IoT – Internet of Things*), na qual a maior parte das comunicações ocorrerá entre dispositivos inteligentes.

A confidencialidade da comunicação fim-a-fim é vital. Dados lidos de medidores de energia inteligentes (*smartmeters*) podem determinar a ocupação de uma casa (CHEN *et al.*, 2013), uma vez que a informação coletada pode ser utilizada por um ladrão para determinar o melhor horário para invadir a residência, por exemplo.

A maioria dos sensores e atuadores ainda possui hardware limitado. Por isso, nos anos recentes, vários trabalhos acadêmicos propondo criptografia leve foram apresentados à comunidade. Segundo a definição da ISO (ISO/IEC 29192-2:2012, 2012), *criptografia leve é aquela que foi ajustada para implementações em ambientes com restrição. Essas restrições podem ser características como área do chip, consumo de energia, tamanho da memória ou banda de comunicação.*

Também pode ser observado que organizações padronizadoras também estão interessadas nesse tipo de criptografia. Em 2012, a ISO publicou um padrão sobre cifras de bloco utilizando criptografia leve (ISO/IEC 29192-2:2012, 2012). O NIST também iniciou um projeto nessa área de estudos em 2013 (MCKAY *et al.*, 2016). Desde então, dois workshops sobre o tópico foram realizados e o próprio NIST está planejando um concurso de primitivas criptográficas leves com chamada de propostas para 2017.

A autenticação também é importante para validar as mensagens quando os comandos são enviados para os atuadores. Ligar ou desligar uma lâmpada pode não ser nocivo, porém, desligar um sistema de suporte à vida pode ter graves consequências. A proteção da comunicação também deve funcionar contra ataques de falsificação de mensagem. Caso um

atacante consiga interceptar alguma mensagem, ele não deve ser capaz de modificá-la pelo fato de ser autenticada. Vale destacar que a pesquisa feita para o desenvolvimento deste trabalho não encontrou esforços para integrar a autenticação nas cifras leves. Nem mesmo estudos sobre o impacto computacional do uso de autenticação foram encontrados.

Um terceiro ponto importante no ambiente de IoT é a variedade de dispositivos – existem dispositivos simples como sensor de porta ou outros mais complexos como uma câmera de reconhecimento facial. Alguns enviam informações na ordem de bits e outros na ordem de gigabytes. Enquanto um tamanho de bloco grande é mais eficiente na transmissão de muitos dados, ele pode ser inadequado para um elemento que envia poucos dados e tem uma restrição grande de hardware. O mesmo vale para o tamanho da chave, para controle de iluminação de uma casa, uma chave de 128 bits é aceitável. Para controle de componentes industriais de uma refinaria, pelo risco de graves acidentes em caso de invasão, pode-se optar por chaves maiores. Logo, a flexibilidade para trabalhar com tamanhos variáveis de bloco e chave é muito interessante nesse ambiente.

Em resumo, a motivação para este trabalho é propor um algoritmo de criptografia para ser utilizado no ambiente de IoT. Pelos motivos apresentados anteriormente, ele deve ser leve, flexível e autenticado.

O trabalho vai além, apresentando um método que permite utilizá-lo em um ambiente IoT ou mesmo em redes SCADA tradicionais.

1.2 CARACTERIZAÇÃO DO PROBLEMA

Pode-se dividir este trabalho em dois problemas. O primeiro problema a ser tratado é se é possível definir um algoritmo de criptografia leve autenticado e com tamanho de bloco variável para ser utilizado no ambiente IoT. O segundo problema é como aplicar um algoritmo de criptografia flexível, leve e autenticado no ambiente IoT.

Existem várias pesquisas que abordam a criptografia leve. Em Crypto 2016, as cifras SKINNY e MANTIS foram apresentadas (BEIERLE *et al.*, 2016). A MANTIS é uma variante de baixa-latência da primeira, com um menor atraso causado pelo processamento. A cifra

SKINNY é uma cifra leve com tamanho flexível de bloco, chave ou *tweak* [1]. Essa nova cifra foi criada para competir com a cifra SIMON (BEAULIEU *et al.*, 2015), projetada pela NSA. Essa última foi desenvolvida para atender os requisitos do ambiente IoT. Nenhuma dessas cifras prove autenticação.

A PRIDE (ALBRECHT; DRIESSEN, 2014) é uma cifra leve que teve seu projeto focado na camada linear[2]. A cifra PRINCE (BORGHOFF *et al.*, 2012) compete com a MANTIS em termos de latência. Outro exemplo de cifra leve é a LED (GUO *et al.*, 2011). Existem muitas outras cifras leves definidas e estudadas pela comunidade. O grupo de pesquisa CryptoLUX mantém uma lista atualizada de cifras leves (CRYPTOLUX RESEARCH GROUP - UNIVERSITY OF LUXEMBOURG, 2016).

Nenhum dos trabalhos citados anteriormente considerou o impacto da autenticação. Uma cifra de bloco simétrica pode prover autenticação pelo seu modo de operação. Provavelmente, os autores citados acima esperam que isso ocorra. Porém, nem sempre é possível, eles precisariam confirmar se o modo de operação é compatível com a cifra proposta. Por exemplo, as cifras com blocos de 64 bits são incompatíveis com os modos de operação NIST GCM (DWORKIN, 2007) e CCM (DWORKIN, 2004) que proveem autenticação. Esses modos de operação requerem uma cifra com tamanho de bloco de 128 bits. Criar uma cifra com a autenticação integrada pode evitar essa armadilha.

No ambiente IoT, que existem uma grande variedade de tipos de dispositivos, trabalhar apenas com cifras de blocos de 64 bits ou 128 bits pode não ser a solução mais eficiente. Seria interessante utilizar cifras flexíveis que trabalhem com tamanhos de blocos variáveis e que permitam tamanhos maiores como 256 bits e 512 bits. O tamanho da chave variável também é importante. Vale ressaltar que o NIST recomenda que o tamanho mínimo de uma chave seja 112 bits (BARKER; ROGINSKY, 2011).

Para tratar o primeiro problema, este trabalho projetou uma cifra leve autenticada com tamanho flexível de bloco e chave. Esta cifra foi baseada na construção Even-Mansour (EVEN; MANSOUR, 1997) e na IAPM (JUTLA, 2001) para autenticação integrada.

---

[1] *Tweak* é um dado utilizado para mudar o criptograma resultante do processo de cifragem. Ele é utilizado para cifragem de dados que podem possuir valores iguais em seus blocos e que precisam ser modificados de forma independente. Como exemplo de aplicação temos a cifragem de um disco rígido, neste caso o índice de cada bloco de dados do disco pode ser utilizado como *tweak* garantindo valores diferentes mesmo quando cifrando blocos com os mesmos dados.

[2] A camada linear pode ser representada por uma função linear dos bits de entrada gerando os bits de saída. Ela mistura ou reorganiza os bits no estado intermediário da cifra.

A propriedade de aleatoriedade da cifra proposta foi verificada com os mesmos experimentos utilizados para validar os candidatos do concurso AES (SOTO, 1999). Para verificar a segurança, foi feita a criptanálise diferencial e linear da cifra proposta. Para determinar se ela pode ser considerada uma cifra leve, foi utilizada a ferramenta de avaliação de sistemas criptográficos leves FELICS (DINU *et al.*, 2015).

Em relação a aplicar de maneira prática o algoritmo proposto, que é o segundo problema, o trabalho, com base em propostas feitas anteriormente para utilizar criptografia para proteger redes SCADA, detalha como proteger a comunicação fim-a-fim utilizando apenas criptografia simétrica.

A solução proposta poderá ser utilizada em redes tradicionais SCADA, que são segregadas por firewalls, e no ambiente IoT. Se a comunicação fosse apenas orientada a conexão, o transporte TLS/SSL poderia ser utilizado. Mas em vários cenários, como o de uma rede de sensores com alta taxa de perdas e erros, a comunicação sem conexão é mais adequada.

Para alguns casos é ainda mais eficiente se for utilizada comunicação *multicast*[3]. Em um andar que tenha uma centena de lâmpadas, enviar apenas um comando via *multicast* para apagar todas as lâmpadas é melhor que enviar o comando individualmente. Mesmo que seja necessário o reenvio do comando para algumas lâmpadas que não confirmaram o comando, o uso da rede seria significativamente menor.

Em resumo, o segundo problema é resolvido descrevendo um método que provê autenticação e confidencialidade sobre canais *multicast* e sem conexão, utilizando o algoritmo de criptografia leve, flexível e autenticado proposto.

1.3 OBJETIVOS DA DISSERTAÇÃO

Essa dissertação busca resolver os dois problemas citados anteriormente. Logo o objetivo dela é definir um algoritmo leve, flexível e autenticado e descrever uma aplicação prática para este algoritmo.

[3] *Multicast* é o tipo de comunicação que tem um originador e vários receptores. Pode ser comparada com a *unicast* com apenas um originador e um receptor ou a *broadcast* onde um originador para todos os receptores da rede.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

O capítulo 2 apresenta as cifras simétricas de bloco, como exemplo descreve brevemente as cifras DES, baseada numa construção Feistel e em uma função não-inversível, e AES, baseada em cálculos algébricos no corpo de Galois. Em seguida, o conceito de criptografia autenticada também é apresentado. Essa técnica, além de proteger a confidencialidade da mensagem, descarta a mesma se detectar uma alteração ou se sua origem não é de um participante válido da conversação. Para finalizar o capítulo, alguns modos de operação de uma cifra de bloco são apresentados. O entendimento do funcionamento interno desses modos é importante para aproveitar partes de suas estruturas na definição de uma nova cifra autenticada e leve.

O capítulo 3 trata do objeto da dissertação, descrevendo a cifra proposta FlexAE. Ela é baseada na construção Even-Mansour e em uma permutação que trabalha com tamanho de bloco variável. A cifra utiliza três caixas de substituição (SBox) de tamanho $8x8^4$ geradas pela inversa multiplicativa no corpo de Galois, definido por diferentes polinômios irredutíveis. A primeira SBox é a mesma utilizada pelo AES e as outras duas são calculadas da mesma maneira, apenas variando o polinômio irredutível e a transformação afim.

O capítulo 4 apresenta a avaliação de desempenho da cifra em comparação a outras cifras leves.

O capítulo 5 contém os testes estatísticos utilizados para validar a aleatoriedade da cifra e como foram seus resultados.

O capítulo 5 trata da verificação da resistência do algoritmo proposto em relação a ataques de criptanálise linear e diferencial.

O capítulo 7 mostra uma proposta de aplicação prática da cifra. Ele descreve um sistema que permite a comunicação de dispositivo no ambiente IoT de maneira segura.

O capítulo 8 traz a conclusão, contribuições e um breve comentário sobre trabalhos futuros.

[4] Neste caso possui entrada e saída com tamanho de oito bits totalizando 256 entradas e saídas possíveis.

2 CONCEITOS TEÓRICOS

2.1 CIFRAS DE BLOCO SIMÉTRICAS

As cifras de bloco simétricas são os blocos construtores de várias técnicas utilizadas para garantir a confidencialidade, integridade e autenticidade de uma mensagem. As técnicas simétricas possuem um custo computacional menor e são mais rápidas que as técnicas assimétricas. Esta seção descreve o que é um cifra de bloco simétrica e exemplifica com as cifras clássicas DES e AES. A figura FIG. 2.1 contém um diagrama básico das operações de cifragem e decifragem de uma cifra simétrica.

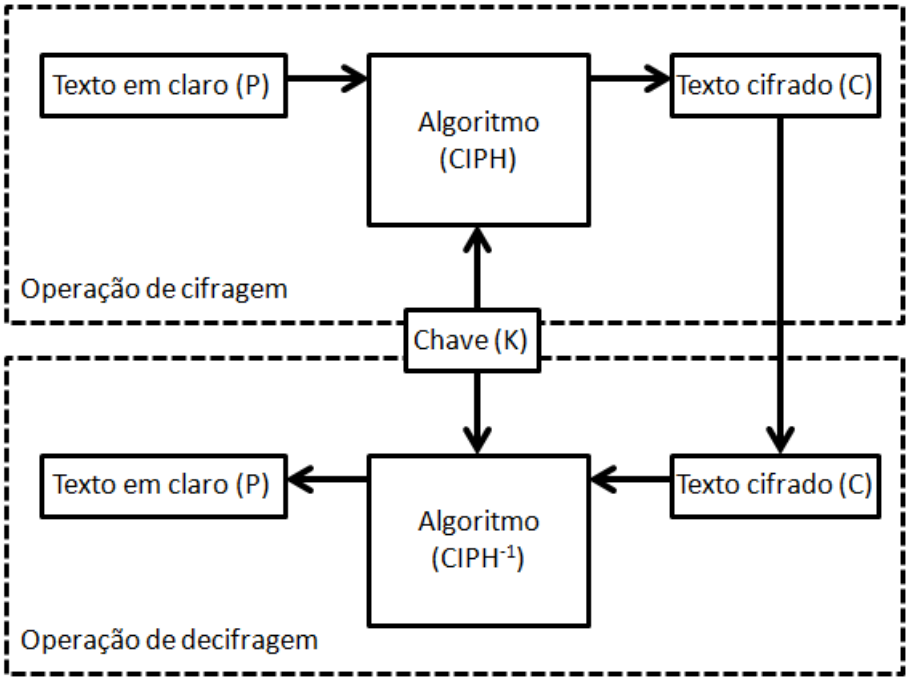

FIG. 2.1 diagrama das operações de cifragem e decifragem de cifra simétrica

Um algoritmo de criptografia é uma função que, tendo como entrada uma chave e um texto claro, gera como saída um texto cifrado ou criptograma. Se esse algoritmo utiliza a mesma chave (ou que possa ser derivada facilmente) para decifrar a mensagem, ele é um algoritmo simétrico. Se ele utilizar chaves diferentes para cifrar e decifrar a mensagem, ele é um algoritmo assimétrico.

Um algoritmo também pode ser classificado como de fluxo, se o texto for cifrado bit a bit,

ou de bloco. No caso de algoritmos de criptografia de blocos, a mensagem é dividida em blocos de tamanho igual para serem cifrados.

Uma cifra de bloco simétrica é um algoritmo determinístico $(CIPH)$, que gera um bloco de texto cifrado $(C \in \{0,1\}^n)$ a partir de um bloco de texto em claro $(P \in \{0,1\}^n)$ e uma chave $(K \in \{0,1\}^k)$.

$$CIPH_K(P) \rightarrow C$$

Para decifrar a mensagem, existe um algoritmo determinístico inverso $(CIPH^{-1})$ que recupera o bloco de texto em claro $(P \in \{0,1\}^n)$ a partir do bloco de texto cifrado $(C \in \{0,1\}^n)$ e uma chave $(K \in \{0,1\}^k)$.

$$CIPH_K^{-1}(C) \rightarrow P$$

Os algoritmos normalmente são padronizados por órgãos como o NIST, nos Estados Unidos, ou internacionalmente, pela ISO. Esses órgãos também padronizam métodos de utilização desses algoritmos para cifrar e decifrar uma mensagem. Esses são os modos de operação de uma cifra e atuam de forma externa ao algoritmo.

2.1.1 CIFRA DES

O DES é um algoritmo simétrico com bloco de tamanho 64 bits e chave de 56 bits. Ele é baseado numa rede Feistel de 16 rodadas. Essa estrutura divide o bloco de dados em duas partes iguais $P = L|R$. Para cada rodada $L_{i+1} = R_i, R_{i+1} = L_i \oplus F_{k(i+1)}(R_i)$.

A função $F_{K(i+1)}$ é composta por uma função E que expande a metade direita da sequência de 32 bits para 48 bits. Faz uma operação XOR desta sequência com a subchave K_{i+1}. O resutado do XOR passa por 8 caixas de substituição (SBox) que esperam como entrada 6 bits e devolvem 4 bits. O resultado das SBoxes passa por uma função de permutação P. As permutações inicial, incial invertida e P apenas mudam de posições os bits da sequência. A figura FIG. 2.2 contém um diagrama do algoritmo DES.

O DES foi inicialmente especificado como padrão pelo documento FIPS 46 do NIST, em 1977. Apesar de ser considerado obsoleto, o DES (ou DEA como é referido atualmente nos documentos do NIST) é o componente primário do algoritmo TDEA (BARKER; BARKER, 2012). Os detalhes de implementação do DES podem ser encontrados no documento que descreve o TDEA.

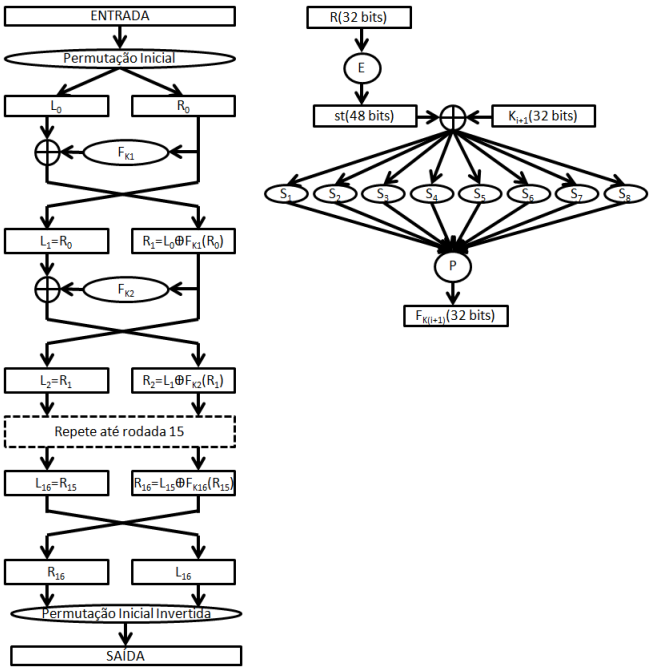

FIG. 2.2 diagrama de blocos da cifra DES

Várias cifras como Lucifer, FEAL, Blowfish e tantas outras também utilizam redes Feistel.

Uma construção clássica utilizando uma rede Feistel é a Lucky-Rackoff (LUBY; RACKOFF, 1988), que é utilizada para transformar uma função pseudo-aleatória[5] em uma permutação pseudo-aleatória. Ela indica a possibilidade de criar uma permutação pseudo aleatória com apenas 4 rodadas em uma rede Feistel.

2.1.2 CIFRA AES

A cifra AES foi escolhida a partir de um concurso lançado em 1997 para escolha de um substituto do DES pelo NIST. Em 2001, o algoritmo Rijndael, dos pesquisadores belgas Joan Daemen e Vicent Rijmen, foi escolhido como o vencedor do concurso (DAEMEN; RIJMEN, 2001). O diagrama básico da cifra AES está na figura FIG. 2.2.

O AES trabalha com uma matriz de bytes como seu estado interno. Esta matriz é organizado em 4 linhas e 4 colunas, totalizando os 128 bits de tamanho. Ele possui quatro funções internas AddRoundKey, SubBytes, ShiftRow e MixColumns. A cifra AES utiliza

[5] Função pseudo-aleatória gera uma sequência de bits que é indistinguível de uma sequência realmente aleatória.

álgebra no corpo de Galois. A função AddRoundKey faz um XOR do array de estado interno com a subchave de rodada.

A SubBytes substitui cada byte do estado interno por outro por meio de uma caixa de substituição ou SBox. O cálculo de sua SBox é baseado na álgebra no corpo de Galois. Ela é calculada pela inversa multiplicativa no corpo de Galois $GF(2^8)$ definido pelo polinômio irredutível $x^8 + x^3 + x^2 + x + 1$. O elemento 0 é mapeado para ele mesmo. Depois, cada número desta tabela é multiplicada por 0x1F $mod(x^8 + 1)$ e, então, é adicionada a constante 0x63. A SBox do AES direta e inversa do AES pode ser encontrada na Apêndice II.

A ShiftRows (FIG. 2.3) rotaciona as linhas da tabela de estado: a linha 0 não sofre mudanças; a linha 1 rotaciona um byte para a esquerda; a linha 2 rotaciona dois bytes; e a linha 3 rotaciona três bytes.

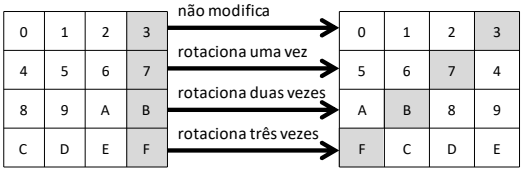

FIG. 2.3 função ShiftRows do AES

A MixColumns é uma multiplicação que trata cada coluna do estado interno da cifra como um polinômio de 4 termos. Este polinômio é multiplicado pelo polinômio fixo $a(x) = \{0x03\}x^3 + \{0x01\}x^2 + \{0x01\}x + \{0x02\}$ no corpo de Galois $GF(2^8)$ modulo $x^4 + 1$.

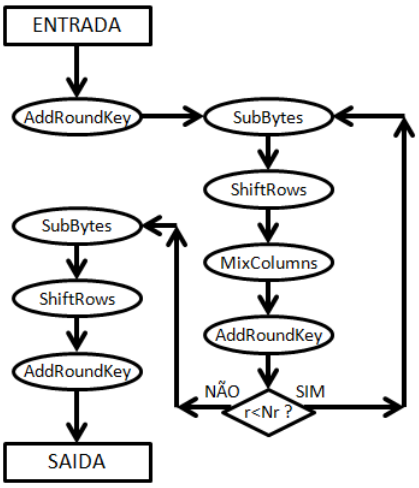

FIG. 2.4 diagrama de blocos da cifra AES

No algoritmo AES, o número de rodadas varia conforme o tamanho da chave. O número de rodadas são Nr=10 (AES128), Nr=12 (AES192) e Nr=14 (AES256).

2.2 CRIPTOGRAFIA AUTENTICADA

Criptografia Autenticada (AE) é o método que fornece confidencialidade, integridade e autenticidade. Ele é utilizado para proteger várias aplicações como a comunicação de uma rede sem fio (WPA2), a navegação segura na Internet (HTTPS), acesso remoto seguro a consoles (SSH), redes virtuais privativas (IPSEC), apenas para citar alguns exemplos.

A cifra proposta se diferencia dos outros métodos de criptografia autenticada por integrar a autenticação no seu modo básico de operação. Isso traz vantagens tanto no peso da cifra quanto na facilidade de uso.

Segundo Rogway (2011), existem duas estratégias para alcançar a criptografia autenticada. A primeira, e mais utilizada, é a composição de modos existentes de confidencialidade e de autenticidade. O modo GCM (DWORKIN, 2007) padronizado pelo NIST é um exemplo que cifra os dados utilizando uma variação do modo CTR (DWORKIN, 2001) e uma função hash[6] com chave. A segunda estratégia é o modo integrado que combina mecanismos de autenticidade e confidencialidade de maneira intrínseca. Essa estratégia também é chamada de criptografia autenticada de passo único.

A operação de cifragem de uma função criptográfica autenticada, normalmente, requer como entrada uma mensagem em texto claro (M), um vetor de inicialização (IV) ou uma sequência de uso único (N) e a chave criptográfica. Como saída, ela retorna a mensagem cifrada (C) e uma etiqueta (tag).

$$CIPH_K(IV, M) \rightarrow (C, tag)$$

Para decifrar a função tem como entrada o (IV) ou (N), a mensagem cifrada (C) e a etiqueta (tag). Se a mensagem cifrada (C) estiver válida, ela retorna a mensagem em texto claro (M). Se ela detectar alguma alteração na mensagem cifrada (C), ela retorna uma mensagem vazia.

$$CIPH_K^{-1}(IV, C, tag) \rightarrow (M) \cup (\emptyset)$$

A AE pode incluir algum dado associado (*Associate Data* - AD) mantido em texto claro. Este AD pode, por exemplo, ser o cabeçalho de um pacote de rede. Qualquer alteração no cabeçalho pode indicar uma falsificação e o pacote deve ser descartado. Nesse caso, o método é chamado de criptografia autenticada com dados associados (AEAD).

---

[6] Segundo (MENEZES; VAN OORSCHOT; VANSTONE, 1996), hash é uma função computacionalmente eficiente que mapeia cadeias binárias de tamanho arbitrário em uma de tamanho fixo.

O *tag*, utilizado na criptografia autenticada, é muito semelhante a um *hash* ou *checksum*, utilizado para validar a integridade de uma mensagem. A principal diferença é que o *tag* não pode ser calculado por um atacante caso ele não conheça a chave de autenticação. Algumas vezes, para conseguir a autenticidade de uma mensagem, primeiro calcula um hash utilizando algum algoritmo não dependente de chave, como MD5 e SHA-1, e, em seguida, cifra o mesmo.

A importância da criptografia autenticada vem crescendo na comunidade científica. A competição CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness (Cryptographic Competitions, 2016) ocorreu no mesmo período que este trabalho era escrito. O objetivo dessa competição é selecionar um portfólio de cifras autenticadas.

2.3 MODOS DE OPERAÇÃO DE UMA CIFRA DE BLOCO SIMÉTRICA

O estudo dos modos de operação é importante para aproveitar parte de suas estruturas para integrar a cifra proposta. Esta seção do trabalho é resultado de um survey sobre os modos de operação existentes. Os métodos são descritos de maneira resumida, mais detalhes de cada modo pode ser conseguido na bibliografia utilizada. Vale ressaltar que a opção foi por descrever apenas a operação de cifragem. A omissão proposital da operação de decifragem, que pode ser deduzida da análise do processo de cifragem, não afeta o entendimento das estruturas internas dos métodos. Existem muitos outros modos de operação e o leitor poderá notar a falta deles, como exemplo o XTS, no trabalho. Isso foi feito porque uma lista exaustiva de modos não traria significativa contribuição, uma vez que as estruturas possuem relação direta com a cifra proposta nesta dissertação.

Os modos de operação de uma cifra podem ser separados em três categorias. Na primeira, os modos apenas focam em garantir a confidencialidade de uma mensagem. São métodos que tem como saída mensagens que não podem ser entendidas por terceiros que não conheçam a chave. Na segunda, vemos métodos para garantir a autenticidade da mensagem. Esse pode ser tanto em texto claro quanto cifradas. A terceira categoria, de criptografia autenticada, são métodos que garantem a confidencialidade, integridade e autenticidade das mensagens.

2.3.1 MODOS DE OPERAÇÃO PARA CONFIDENCIALIDADE

Para confidencialidade, tanto o NIST (DWORKIN, 2001) quanto a ISO (ISO/IEC 10116, 2006), recomendam 5 modos de operação para as cifras de blocos simétricas. Os modos recomendados são: ECB – Eletronic Code Book (livro de códigos eletrônico), CBC – Cipher Block Chaining (encadeamento de blocos de cifra), CFB – Cipher Feedback (realimentação de cifra), OFB – Output Feedback (realimentação de saída) e CTR – Counter (contador). O modo de operação mais básico de uma cifra de bloco é o ECB.

2.3.1.1 MODO ECB

O modo ECB (FIG. 2.5) ou livro eletrônico de códigos é a aplicação da cifra de bloco utilizando uma chave K sobre o texto em claro.

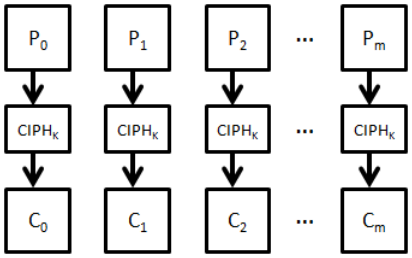

FIG. 2.5 cifragem no modo de operação ECB

O problema desse modo é que todos os blocos de texto em claro iguais irão gerar o mesmo bloco de texto cifrado. Isso pode ser utilizado para inferir alguma informação da mensagem. Se a mesma chave for utilizada para enviar várias mensagens, pode haver uma colisão de blocos de texto em claro e o atacante pode criar uma tabela de relação de alguns blocos com o texto em claro. Essa característica indica que o modo ECB é semanticamente inseguro.

De todas as maneira, esse modo de operação é utilizado como bloco de construir dos outros modos, conforme pode ser visto nos próximos modos, tornando ele muito importante.

2.3.1.2 MODO CBC

O modo CBC (FIG. 2.6), ou encadeamento de blocos de cifra, utiliza o bloco cifrado

anterior e faz um XOR com o bloco de texto em claro. Para o primeiro bloco, um vetor de inicialização aleatório é utilizado no lugar do bloco cifrado. Assim, o texto cifrado será diferente mesmo se os blocos de texto em claro e a chave forem iguais.

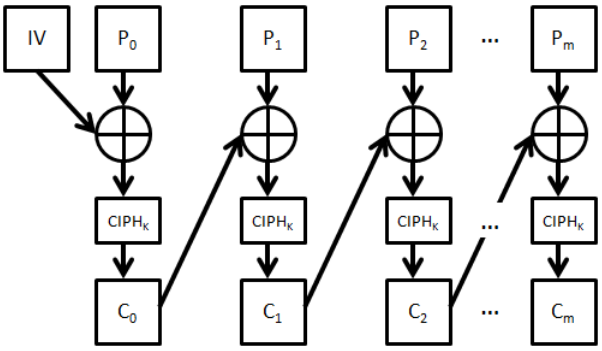

FIG. 2.6 cifragem no modo de operação CBC

Uma desvantagem deste modo é o fato de que as operações de cifragem dos blocos não podem ocorrer em paralelo. Elas dependem do resultado da cifragem anterior.

2.3.1.3 MODO CFB

O modo CFB (FIG. 2.7) ou realimentação de cifra utiliza o bloco cifrado anterior como parte da entrada do cifrador. Este modo tem, como uma de suas características, trabalhar com blocos de texto em claro e blocos cifrados menores que o tamanho de bloco da cifra.

Para o primeiro bloco, toma-se o IV de tamanho b bits como entrada I_0. Este bloco I_0 é cifrado utilizando a chave K gerando um bloco de estado intermediário. Desse bloco, os s bits mais significativos são separados e faz-se um XOR como bloco de texto claro P_0 de tamanho s bits. O resultado é o bloco cifrado C_0. Esse bloco é concatenado a direita do I_0. Os b bits menos significativos são utilizados como entrada da cifra no próximo bloco.

A operação de cifragem dos blocos do modo CFB não pode ser feita de forma paralela porque a cifragem de um bloco depende do da cifragem do bloco anterior.

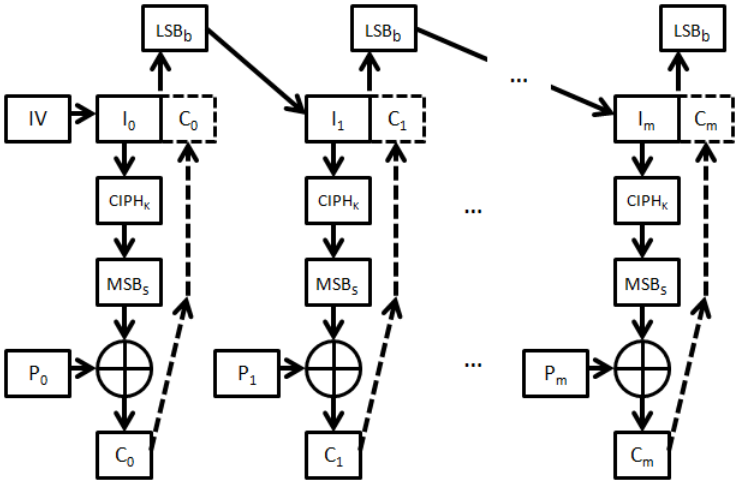

FIG. 2.7 cifragem no modo de operação CFB

2.3.1.4 MODO OFB

O modo OFB, ou realimentação de saída, utiliza a cifra para gerar uma sequência de bits que faz um XOR com a mensagem em texto claro. Inicialmente, toma o IV e submete a cifra sob a chave K. O resultado S_0 desta cifragem é submetido novamente à cifra sob a chave K até criar uma sequência do tamanho da mensagem. Então, faz-se um XOR com a mensagem em texto claro para gerar a mensagem cifrada.

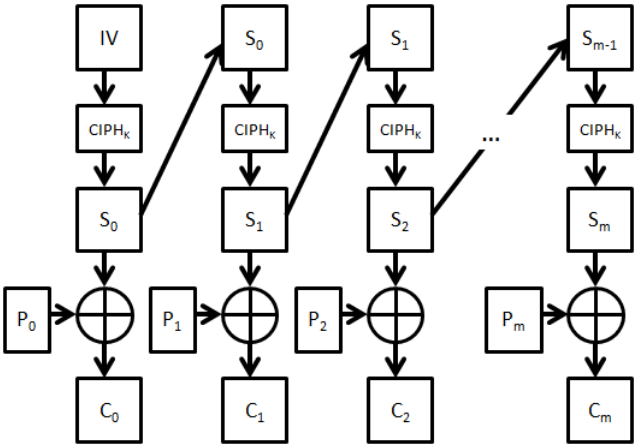

FIG. 2.8 cifragem no modo OFB

2.3.1.5 MODO CTR

O modo contador (CTR) é um método que a partir de um método de contagem específico gera uma sequência de bits utilizada para fazer XOR com a mensagem em texto claro. Para exemplificar considere que o método de contagem é a simples soma de +1 a cada bloco. Nesse caso, toma-se um bloco contador CB_0 com um valor inicial, que não poderá ser repetido para a mesma chave. Se isso ocorrer, a segurança da cifra é perdida. Os b bits menos signficativos do contator são incrementados gerando os blocos seguintes $CB_1 CB_2 \ldots CB_m$. Cada bloco contador é cifrado utilizando a chave K para gerar uma sequência de bits $S_0 S_1 \ldots S_m$. Faz-se então um XOR com a mensagem em texto claro para gerar a mensagem cifrada.

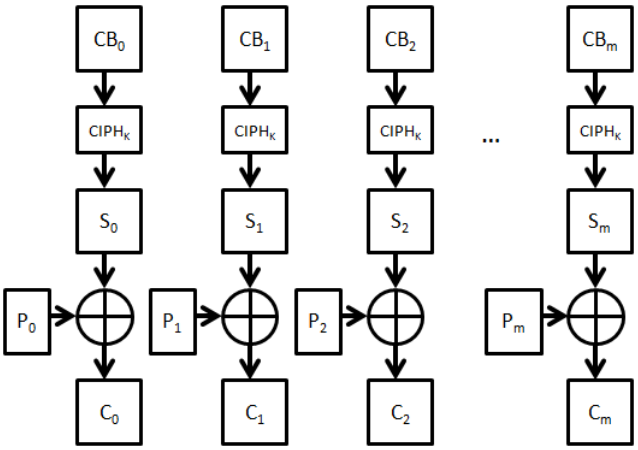

FIG. 2.9 cifragem no modo contador

2.3.2 MODOS DE OPERAÇÃO PARA VERIFICAÇÃO DE AUTENTICIDADE

Para verificação de autenticidade destaca-se os modos de operação HMAC, NMAC, CBC-MAC e o CMAC.

2.3.2.1 MODOS HMAC E NMAC

Os modos HMAC e NMAC foram propostos por BELLARE et al. (1996). Nesse material, primeiro descreve-se a construção NMAC que pode ser utilizada com uma função. O modo NMAC é apenas a aplicação de uma função hash dependente de chave de forma aninhada.

$$NMAC_K(x) = H_{K1}\big(H_{K2}(x)\big)$$

Onde K=K1|K2 e x é uma sequência de tamanho arbitrário. A função H pode ser conseguida pela aplicação de uma construção Merkle-Damgard, conforme explicado no artigo, em cima de uma cifra de bloco ou uma função não reversível dependente de chave.

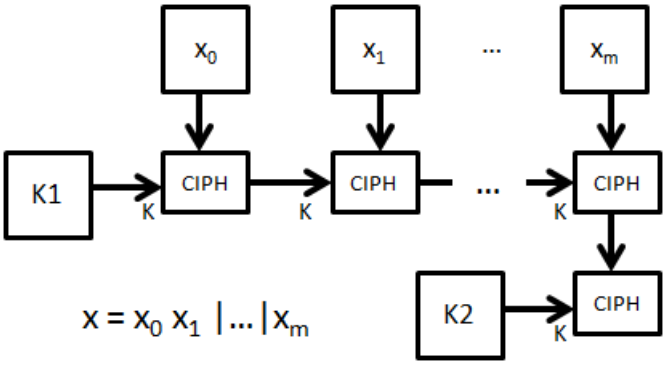

FIG. 2.10 NMAC utilizando construção Merkle-Damgard e uma cifra de bloco

Funções *hash* padrões, como MD5 e SHA-1, utilizam uma construção similar, porém, ao invés de uma chave inicial variável como $K1$ da figura FIG. 2.10, elas utilizam um IV fixo.

Logo o NMAC não poderia ser utilizado com funções *hash* padrão. Para contornar esta situação, os autores também propõem o HMAC que utiliza uma função *hash* sem a necessidade de chave. A construção HMAC é:

$$\mathrm{HMAC_k(x)} = \mathrm{F}(\,(\,\bar{\mathrm{k}} \oplus \mathrm{opad})||\mathrm{F}\,(\,(\bar{\mathrm{k}} \oplus \mathrm{ipad}\,)||\mathrm{x})\,)$$

Onde x é uma sequência de entrada de tamanho arbitrário, $\bar{\mathrm{k}}$ é a chave HMAC k completada com 0s até o tamanho do bloco da função *hash*, opad é composto pelo byte 0x36 repetido até o tamanho da chave $\bar{\mathrm{k}}$, ipad é composto pelo byte 0x5c repetido até o tamanho da chave $\bar{\mathrm{k}}$ e F é uma função hash como MD5 or SHA1.

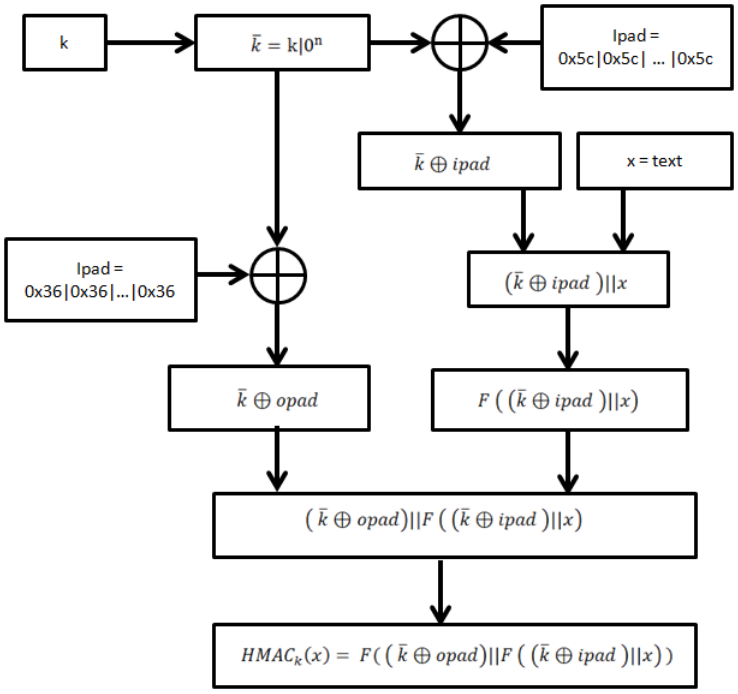

FIG. 2.11 esquema HMAC

2.3.2.2 MODO CBC-MAC

O CBC-MAC utiliza uma cifra de bloco para autenticar a mensagem. O *tag* ou o MAC é o último bloco da cifragem pelo modo de operação CBC (DWORKIN, 2001), usado para confidencialidade.

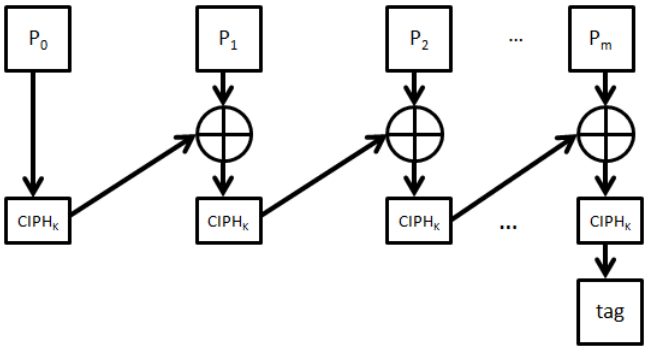

FIG. 2.12 modo CBC-MAC

Essa construção não considera completar a mensagem para chegar ao tamanho do bloco, por isso a mensagem precisa ter um tamanho múltiplo do tamanho do bloco. Como

demonstrado por ROGAWAY (2011), ele é suscetível aos ataques do tipo copiar-e-colar[7].

2.3.2.3 MODO CMAC

O CMAC (DWORKIN, 2005) é basicamente algoritmo OMAC com as melhorias do algoritmo XCBC (NIST, 2016).

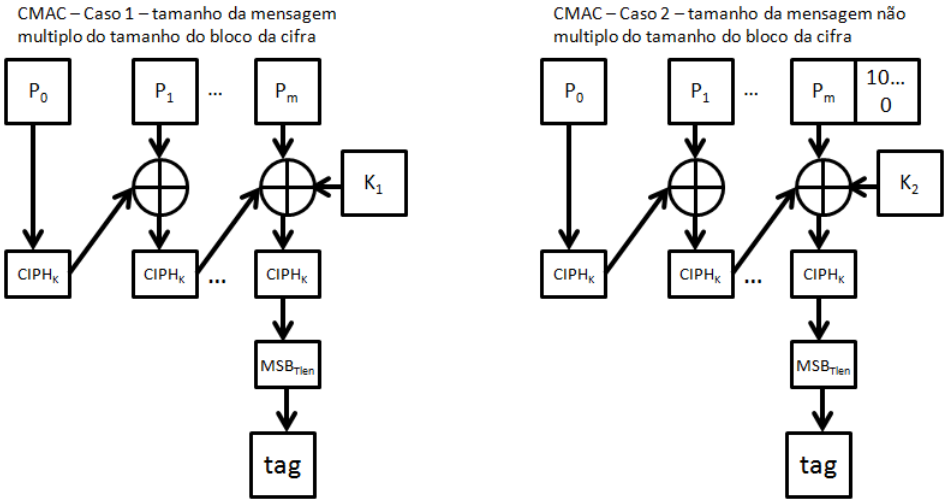

FIG. 2.13 modo CMAC do NIST

As subchaves K_1 e K_2 são derivadas uma chave mestre K: $K_1 = CIPH_K(0^b) \times 2$ e $K_2 = K_1 \times 2$, no $GF(2^b)$ usando o polinomio irredutível $1||R_b$. Onde $R_{128} = 0^{120}10000111$ e $R_{64} = 0^{59}11011$.

2.3.3 MODOS DE OPERAÇÃO DE CRIPTOGRAFIA AUTENTICADA

Entre os modos de operação que atingem criptografia autenticada, pode-se destacar o CCM, o GCM e o IAPM.

2.3.3.1 CCM

O CCM (DWORKIN, 2004) é um modo de criptografia autenticada composto. Ele utiliza

[7] A mensagem de um único bloco M e a mensagem $M||(M \oplus T)$ tem a mesma tag T.

o modo CTR para confidencialidade e o CBC-MAC para garantir a autenticidade. Nesse modo, primeiro é feita a autenticação e, depois, a mensagem é cifrada (autenticar então cifrar).

O modo CCM, representado na FIG. 2.14, também suporta a autenticação do dado associado (AD). Os parâmetros de entrada para o modo é uma sequência de uso único (N), o dado associado (A), a mensagem em texto claro (P) e a chave secreta (K). Na primeira parte, (N|A|P) são separados em blocos (B_0 to B_n). Esses blocos são submetidos ao CBC-MAC para gerar o tag (T). Na segunda parte, a mensagem em texto claro (P) e o tag (T) são cifrados usando o modo CTR.

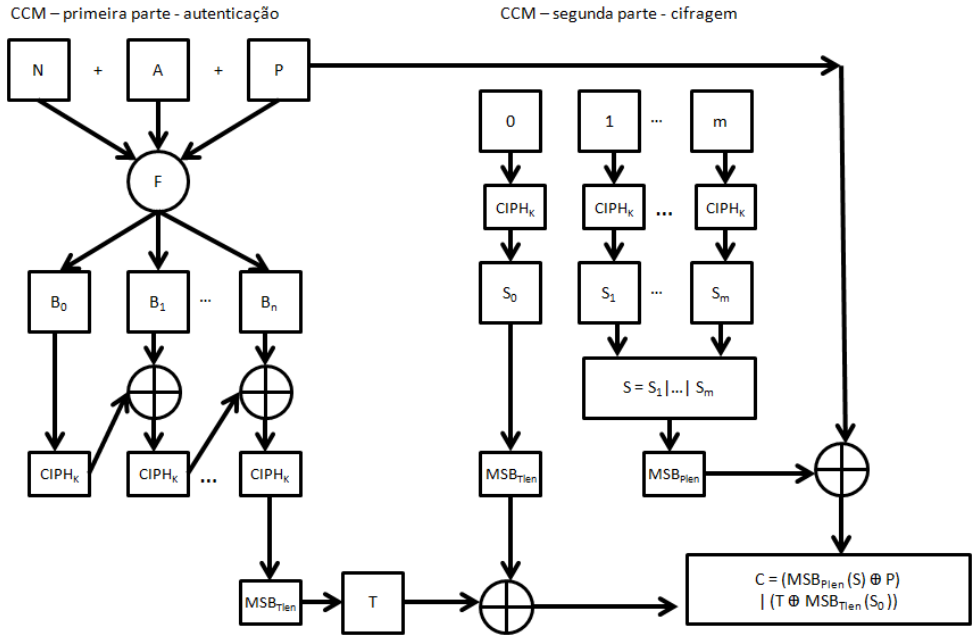

FIG. 2.14 modo CCM

2.3.3.2 MODO GCM

O GCM (DWORKIN, 2007) é outro modo composto, ele utiliza a ordem "cifrar-então-autenticar" (EtA). O modo CTR é utilizado para confidencialidade e o MAC é calculado pela função GHASH. Essa função é baseada na multiplicação no $GF(2^{128})$ usando o polinômio irredutível $11100001(0^{120})1$. Ele é o mesmo polinômio utilizado para calcular as subchaves do modo CCM quando trabalhando com uma cifra de 128 bits. Nesse padrão, ele aparenta ser diferente porque está representado usando o formato "*little endian*" – com o bit menos significativo na frente. A função GHASH basicamente multiplica o bloco pelo número H no

(2^{128}), onde $H = CIPH_K(0^{128})$ e, depois, é feita a operação XOR com o próximo bloco.

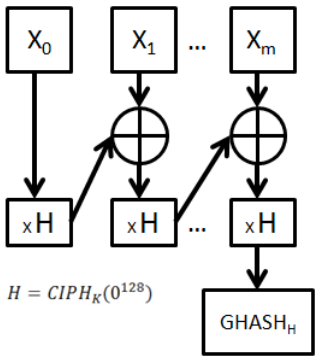

FIG. 2.15 função GHASH

No padrão, a cifragem é representada pela função $\mathrm{GCTR_K}$. O contador é somente incrementado nos últimos 32 bits utilizando a função INC32.

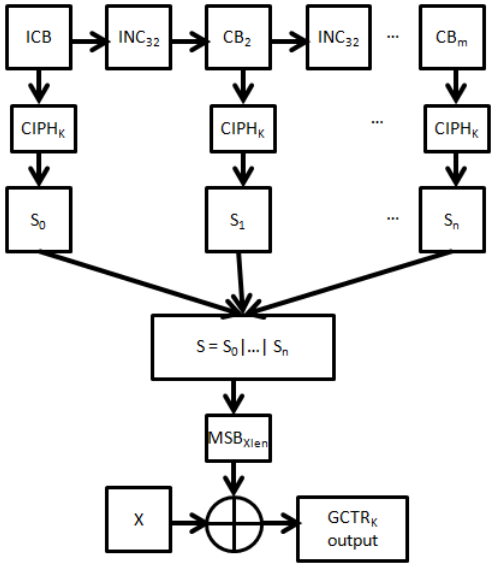

FIG. 2.16 função $\mathrm{GCTR_K}$

O valor inicial do contador (ICB) é J_0. Se o tamanho do IV for exatamente 96 bits, $J_0 = IV|(0^{31})|1$. Se o tamanho for diferente, J_0 será calculado usando a função GHASH sobre o IV e o tamanho do IV.

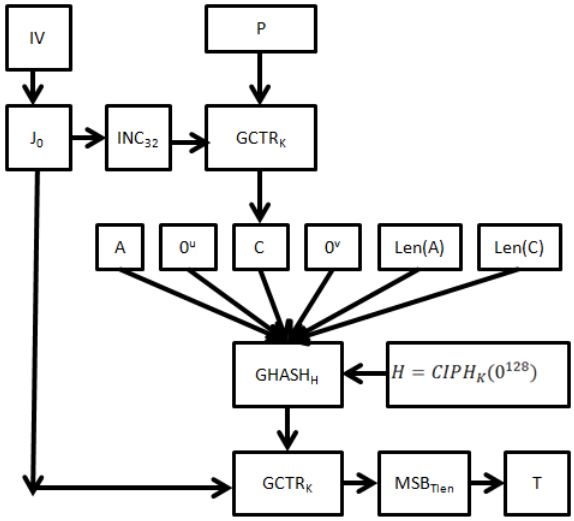

FIG. 2.17 diagrama do modo GCM

2.3.3.3 MODO IAPM

Diferente do CCM e GCM, que são modos compostos, o IAPM (JUTLA, 2001), ou modo paralelizável que considera integridade, é um modo criptografia autenticada integrado ou de passo único.

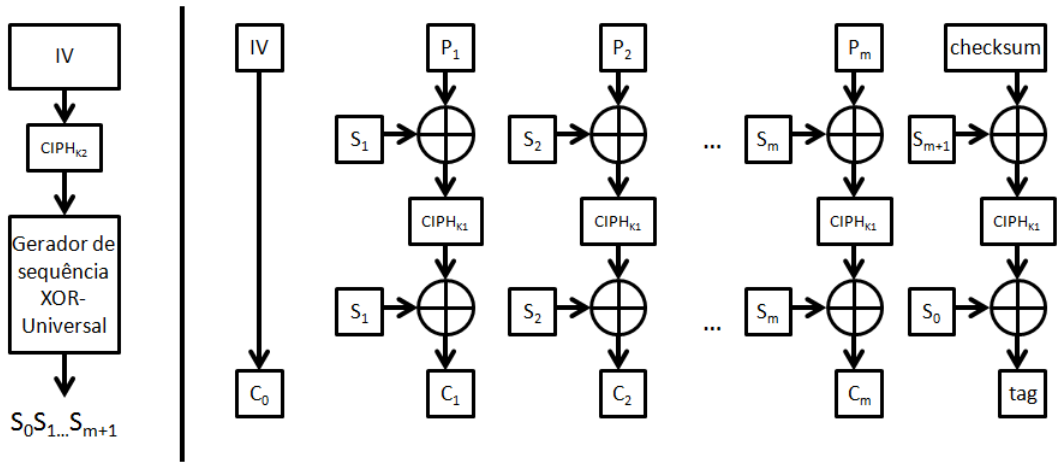

FIG. 2.18 modo IAPM

Nesse modo, uma sequência pseudo-aleatória $S_0S_1 \dots S_{m+1}$ do tipo XOR-Universal é gerada a partir da cifragem do IV utilizando a chave $K2$ e uma construção no corpo de Galois. Tal sequência é utilizada para fazer um XOR com o texto em claro antes e depois de submeter a cifra utilizando a chave $K1$. O *checksum* é calculado fazendo uma operação XOR de todos

os blocos de texto claro. Para calcular o *tag*, é feito um XOR do *checksum* com S_{m+1}, submete a cifra utilizando a chave K1 e, depois, é feito um XOR com S_0.

3 CONSTRUÇÃO DE UM ALGORITMO LEVE E AUTENTICADO

Para ter um algoritmo leve, com baixo custo computacional , deve-se manter sua estrutura simples e com poucas operações.

Para cifras de fluxos, a estrutura mais simples é o *One-Time Pad*, que é composto de apenas uma operação XOR entre o texto em claro e uma sequência aleatória ou pseudo-aleatória.

Para cifras de bloco, Even-Mansour (EVEN; MANSOUR, 1997) é uma cifra minimalista, com poucas operações, e uma grande candidata para cifras leves. Ela utiliza duas operações XOR e uma função de permutação. A figura FIG. 3.1 contém o diagrama da cifra Even-Mansour.

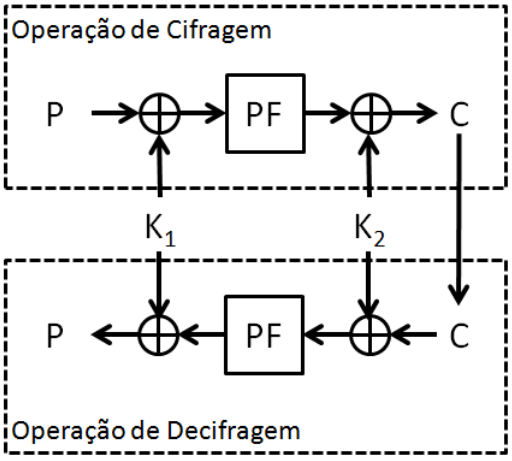

FIG. 3.1 cifra Even-Mansour (EVEN; MANSOUR, 1997)

O objetivo do autor foi propor a construção mais simples de cifra simétrica com alguma prova formal de segurança. No artigo original, os próprios autores indicam que não é possível utilizar uma única operação XOR porque tornaria a cifra insegura.

Apesar de ter se passado mais de 25 anos da publicação desse artigo, não foi localizada outra construção de cifra de bloco mais simples. Em (DUNKELMAN; KELLER; SHAMIR, 2012) a construção foi reanalisada, neste trabalho os autores argumentam que a cifra Even-Mansour de uma chave única seria a mais simples cifra concebível com alguma segurança provável.

A construção Even-Mansour também pode ser descrita pelas equações:

$$\mathrm{CIPH}_{\mathrm{K1,K2}} = \mathrm{PF}(\mathrm{P} \oplus \mathrm{K}_1) \oplus \mathrm{K}_2 \rightarrow \mathrm{C}$$

$$\mathrm{CIPH}^{-1}_{\mathrm{K1,K2}} = \mathrm{PF}(\mathrm{C} \oplus \mathrm{K}_2) \oplus \mathrm{K}_1 \rightarrow \mathrm{P}$$

Onde $\mathrm{CIPH}_{\mathrm{K1,K2}}$ é a operação de cifragem, $\mathrm{CIPH}^{-1}_{\mathrm{K1,K2}}$ é a operação de decifragem, C é o bloco de texto cifrado, P é o bloco de texto em claro, $(\mathrm{K}_1, \mathrm{K}_2)$ são as chaves e PF() é uma função de permutação conhecida.

Para melhorar a segurança, é possível utilizar uma cifra Even-Mansour de r-rodadas. Nesse caso, a construção inclui mais permutação e chaves.

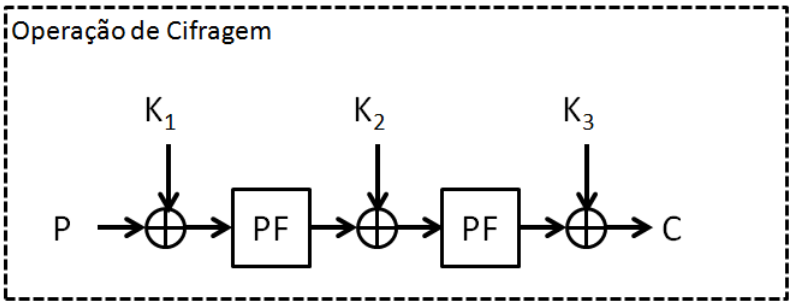

FIG. 3.2 cifra Even-Mansour de duas rodadas

Para uma implementação prática de uma cifra leve baseada na Even-Mansour, o primeiro passo é definir a função de permutação (PF). A opção mais segura e imediata seria utilizar uma função geradora de sequência pseudo-aleatória existente e aplicar a construção Luby-Rackoff (NAOR; REINGOLD, 1999), que é baseada em uma permutação Feistel. O problema dessa construção seria a complexidade da função geradora tornando a permutação pesada.

A segunda opção seria a utilização de uma tabela de substituição fixa gerada a partir de uma distribuição aleatória. Essa estratégia não é adequada para blocos grandes por causa dos requerimentos de memória. Enquanto para uma caixa de substituição de 8x8 são necessários 256 bytes de memória, para uma caixa de substituição de 16x16 são necessários 131.072 bytes memória e para 32x32 são necessários 16 GBytes de memória. Também deve ser considerado que o tamanho do bloco não pode ser mudado facilmente. Nesse caso, a cifra proposta não seria flexível.

Uma terceira opção foi utilizada para essa implementação. Ela utiliza uma função de expansão da permutação. Uma permutação básica de n bits pode ser expandida para ser uma função de permutação de $\mathrm{n} \times 2^{\mathrm{x}}$ bits, onde $x \geq 1$. Essa opção é um meio termo entre o uso de

CPU e o uso de memória.

O trabalho verificou duas formas de fazer essa expansão: uma recursiva e outra iterativa. A função implementada expande uma ou mais caixas de substituição (SBox) de 8 bit transformando-as em uma função de permutação maior de 8×2^x-bits, onde $x \geq 1$. O tempo de execução desta expansão é quasilinear $O(n \log n)$ em relação ao tamanho do bloco.

Depois de selecionar a função de permutação flexível, foi necessário definir como implementar a autenticação. Para conseguir uma proposta leve, a autenticação foi integrada com base no modo IAPM (JUTLA, 2001). A principal diferença com o IAPM é que o *checksum* é calculado com os blocos já cifrados. Isso é feito para ter a ordem "cifrar-então-autenticar", que segundo KRAWCZYK (2001) é genericamente segura. Em seu artigo KRAWCZYK prova, por meio de exemplos, que implementações "cifrar-e-autenticar" e "autenticar-então-cifrar" podem ser completamente quebradas mesmo tendo uma cifra e um algoritmo de MAC perfeitos.

3.1 FUNÇÃO DE PERMUTAÇÃO EXPANSÍVEL

Uma tentativa de expandir, que dobra o tamanho de uma permutação, seria dividir a entrada em dois e aplicar cada metade à função de permutação. Esta tentativa está ilustrada na figura FIG. 3.3. Pode-se verificar facilmente que esse tipo de construção não funciona porque a primeira metade da saída somente depende da primeira metade da entrada e a segunda metade da saída somente depende da segunda metade da entrada.

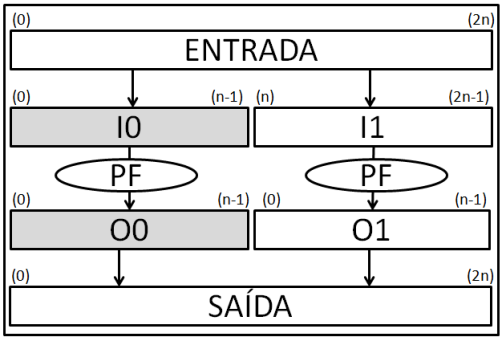

FIG. 3.3 tentativa de expandir uma permutação (PF) de n-bits para uma de 2n-bits

Uma solução para esta expansão é dividir ao meio as saídas da função de permutação PF e trocar metade da primeira função com metade da segunda função. Em seguida, é preciso submeter esses novos estados intermediários à função de permutação. Essa solução pode ser

visualizada na figura FIG. 3.4.

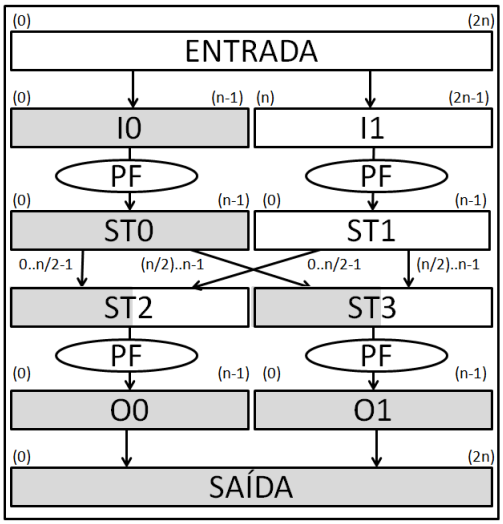

FIG. 3.4 expandindo uma função permutação (PF) de n-bits para 2n-bits

Essa construção somente dobra o tamanho da função de permutação. Se ela for utilizada de maneira recursiva, pode-se ter uma função de permutação com tamanho $\mathrm{n} \times 2^{\mathrm{x}}$. O tempo de execução dessa construção é quasilinear[8] $O(x\,log(x))$. Como a utilização de um algoritmo recursivo pode não ser adequado para implementações em hardware, substituí-se a troca de metade do sub-bloco de saída por uma camada de embaralhamento para permitir uma solução iterativa. A construção Even-Mansour necessita de funções de permutação reversíveis. Tanto a opção iterativa quanto a opção recursiva são reversíveis.

Na solução iterativa, a entrada de $2^{\mathrm{x}} \times \mathrm{n}$ é subdividida em $2^{\mathrm{x+1}}$ blocos de tamanho $\mathrm{n}/2$ $b[0], b[1], \ldots, b[2^{x-1}]$. Os blocos são reordenados: $\mathrm{b}[0], \mathrm{b}[2^{\mathrm{x}}], \mathrm{b}[1], \mathrm{b}[2^{\mathrm{x}}+1], \ldots, \mathrm{b}[2^{\mathrm{x}}-1]$, $\mathrm{b}[2^{\mathrm{x+1}}-1]$. Cada par de sub-blocos consecutivos é submetido à função de permutação original. A saída é novamente dividida em blocos de tamanho n/2 bits. Depois de x+2 iterações, todos os bits de saída dependerão, com a mesma probabilidade, de cada bit de entrada. Assim, como na solução recursiva, o tempo de execução desta solução é quasilinear. A figura FIG. 3.5 tem o exemplo do processo para uma entrada de tamanho 4n-bits.

A função precisa de quatro (4) iterações para que a mudança em um bit de entrada afete todos os bits de saída. Pode-se verificar que uma mudança de um bit no sub-bloco b0, em cinza, atinge toda a saída após três (3) iterações. Uma última rodada é necessária para

[8] No modo recursivo cada chamada da função permutação tem tempo de execução linear $O(x)$ e como a função é chamada $\log(x)$ vezes, o tempo de execução será $O(x \log(x))$.

assegurar que a saída dependa dos blocos de entrada com probabilidade igual. Sem ela a saída da última rodada dependeria apenas de metade dos bits de entrada porque a outra metade seria fixa.

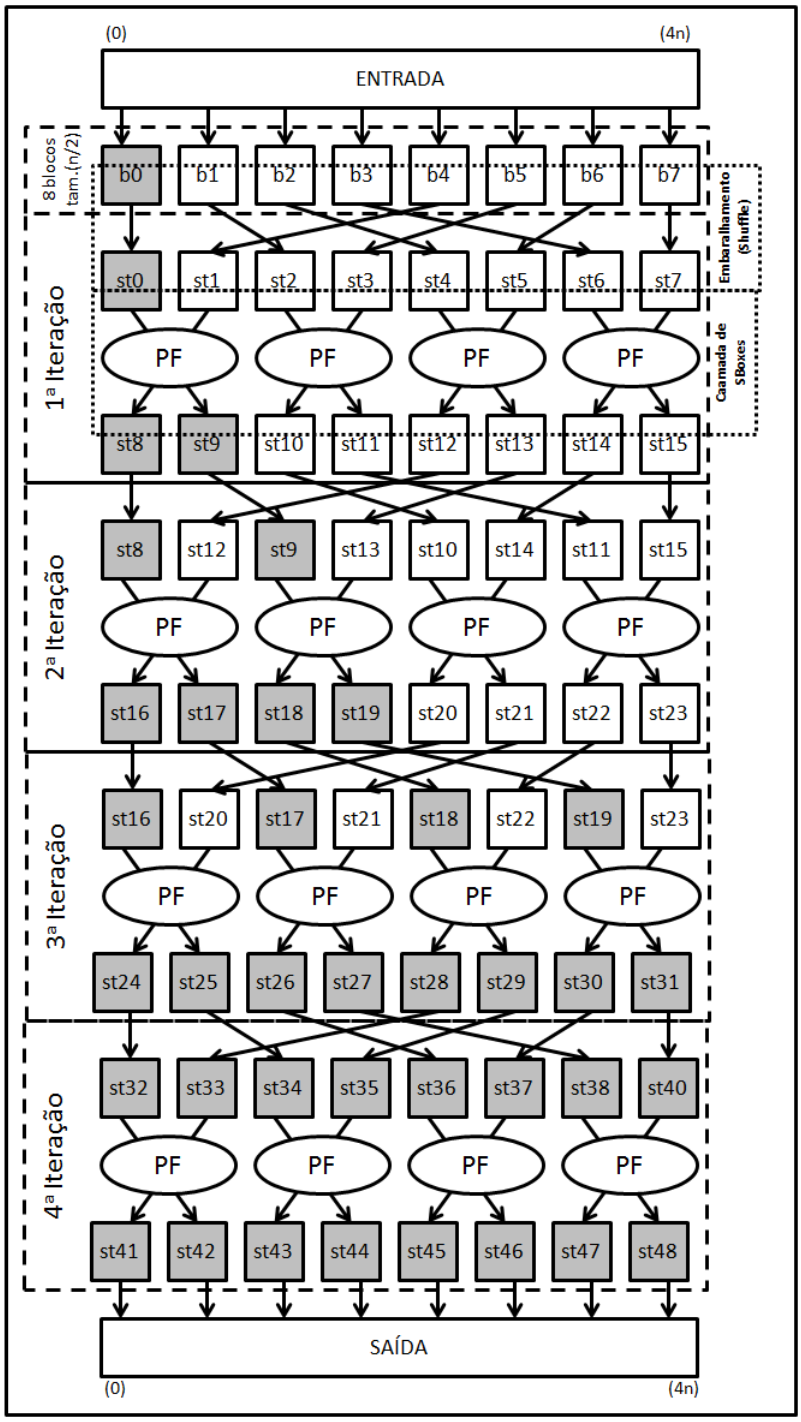

FIG. 3.5 expansão iterativa de uma função de permutação (PF) de n-bits para 4n-bits

O próximo passo foi selecionar uma permutação de n-bits para ser a base da expansão. Preferencialmente, ela deveria ter características não lineares fortes. A forma mais comum de

uma permutação pequena são as caixas de substituição utilizadas por várias cifras de bloco simétricas. Algumas, como a PRESENT (BOGDANOV *et al.*, 2007) e a LED (GUO *et al.*, 2011) utilizam SBoxes de tamanho 4x4; outras, como o AES, utilizam SBoxes de 8x8. Para ter um melhor balanço entre tamanho e desempenho, o tamanho de 8x8 foi selecionado para as SBoxes a serem utilizadas.

Uma opção para a SBox é utilizar a mesma do AES. Ela é calculada criando uma tabela com a inversa multiplicativa dos números 0x1 até 0xFF no corpo de Galois $GF(2^8)$ definido pelo polinômio irredutível $p = x^8 + x^4 + x^3 + x^1 + 1$. O zero é mapeado para ele mesmo porque ele não possui inversa. O um também é mapeado para ele mesmo porque ele é a sua própria inversa multiplicativa. A tabela, então, passa por uma transformação afim. Cada elemento é multiplicado pela constante 0x1F $mod(x^8 + 1)$ e, então, é adicionada a constante 0x63.

Para tratar o problema da camada SBox da função expansível ser muito simétrica, a cifra utilizará 3 Sbox ao invés de apena uma SBox. Na construção proposta, o primeiro byte do bloco utilizará a primeira SBox, o segundo byte utilizará a segunda SBox, o terceiro byte a terceira SBox, o quarto byte utilizará novamente a primeira e assim por diante.

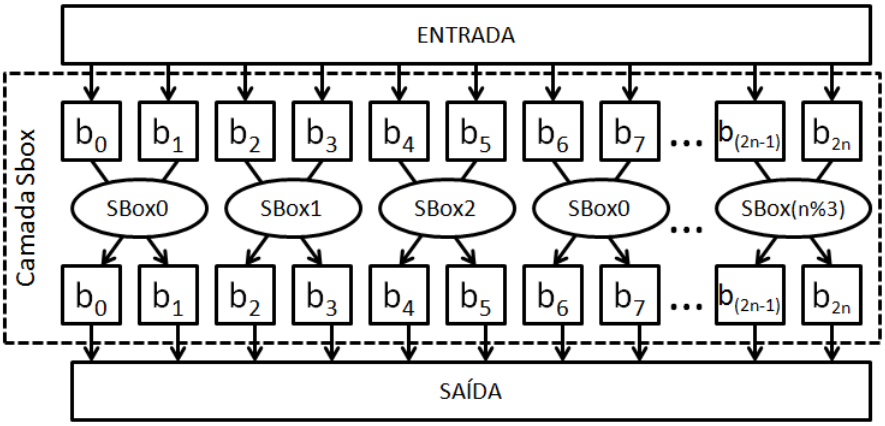

FIG. 3.6 a camada SBox com 3 SBoxes

As outras duas SBox são calculadas da mesma forma da primeira, porém, utilizando outros polinômios irredutíveis. Em (LIDL; NIEDERREITER, 1994), existe uma lista com 30 polinômios irredutíveis de grau 8 que podem ser utilizados. A SBox do AES utiliza o primeiro da lista, para as outras duas SBox os seguintes polinômios irredutíveis são utilizados.

Se a mesma transformação afim fosse utilizada, os dois primeiros bytes seriam iguais nas 3 SBox. Isso ocorre por o 0 e o 1 serem mapeados para eles mesmos na tabela de inversas multiplicativas. Assim, para evitar esse problema, a constante aditiva da transformada afim é

alterada. Como no caso da SBox original do AES, as constantes aditivas selecionadas possuem 2 bits ligados e 2 bits desligados em cada metade do byte. As constantes multiplicativas das transformadas não são alteradas.

TAB. 3.1 parâmetros utilizados para criar as SBox

# SBox	Polinômio Irredutível	Constante Multiplicativa	Constante Aditiva
0	$x^8 + x^4 + x^3 + x^1 + 1$	0x1F	0x63
1	$x^8 + x^4 + x^3 + x^2 + 1$	0x1F	0x95
2	$x^8 + x^5 + x^3 + x^1 + 1$	0x1F	0xA6

As SBox utilizadas pela cifra FlexAE estão disponíveis no Apêndice II. O código fonte para gerar as mesmas está disponível no Apêndice III.

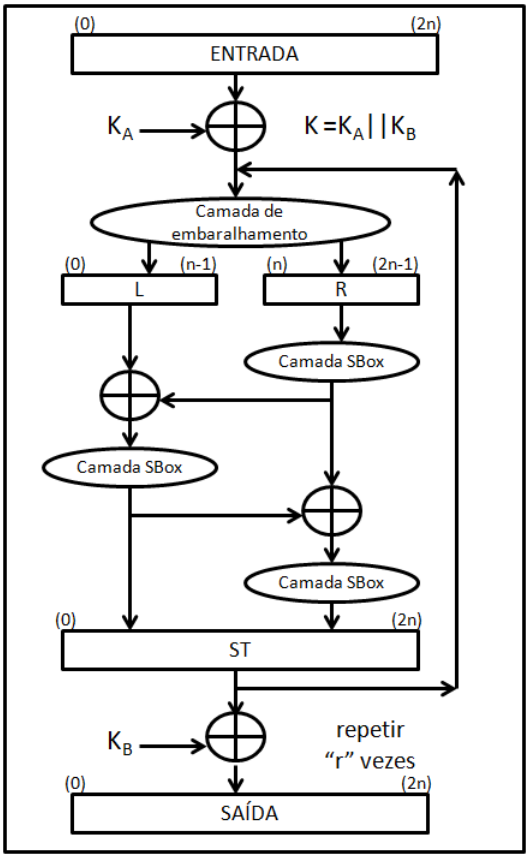

FIG. 3.7 função de permutação expansível dependente de chave PF_K

A construção da figura FIG. 3.5 era a candidata principal para a função de permutação,

porém, ela apresenta uma fraqueza em sua criptanálise diferencial[9]. Para melhorar sua resistência, apesar de aumentar um pouco o custo com processamento, uma modificação foi feita tornando a construção parecida com uma rede Feistel.

A dependência de chave da função de permutação expansível é conseguida com a construção Even-Mansour. Um XOR com as chaves criptográficas é feito antes e depois da função expansível básica – que se torna uma função de permutação expansível dependente de chave PF_K.

As chaves são aplicadas apenas duas vezes, uma no texto claro de entrada e outra pouco antes da saída. Não é necessário criar uma subchave para cada rodada. De qualquer maneira, um processo de expansão da chave mestre é utilizado para gerar as subchaves para a cifra completa.

A figura FIG. 3.7 contém a função de permutação proposta. O número de rodadas da função é $r = (\log_n nb + 2)$, onde nb é o tamanho do bloco em bytes.

3.2 PROCESSO DE GERAÇÃO DE SUBCHAVES

Para a cifra proposta, um total de 3 subchaves são utilizadas: (K_0, K_1, K_2). Cada subchave tem o dobro do tamanho do bloco. Elas são geradas a partir da chave mestra. Para isso, a função de permutação dependente de chave é aplicada duas vezes a uma sequência de 0^n.

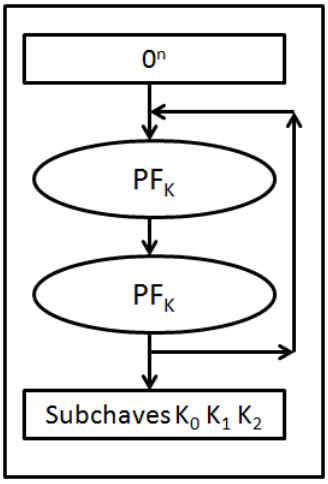

FIG. 3.8 processo de geração das subchaves

[9] A fraqueza consiste no fato que esta construção teria apenas uma SBox ativa por rodada, gerando uma dificuldade de ataque menor que um ataque de força bruta.

O processo é repetido utilizando as saídas como as entradas subsequentes até uma sequência do tamanho das três subchaves ser gerada. A chave master deve ter um tamanho 64×2^{x}, onde $x \geq 1$. O tamanho mínimo da chave mestre é 128 bits e o tamanho máximo é de duas vezes o tamanho do bloco.

3.3 INTEGRANDO AUTENTICAÇÃO À CIFRA

Segundo ROGAWAY (2011), existem duas estratégias para alcançar criptografia autenticada. O primeiro, e mais utilizado, é compor com métodos de confidencialidade e autenticidade existentes. Um exemplo é o modo de operação NIST CCM (DWORKIN, 2004) para autenticação e confidencialidade. Ele cifra os dados utilizando o modo CTR e utiliza o modo CBC-MAC para autenticidade.

O segundo modo é o de Criptografia Autenticada (AE) integrada. Ele combina intrinsecamente mecanismos de autenticidade e confidencialidade. Ele também é chamado de criptografia autenticada de uma passagem (one-pass AE).

A construção integrada é mais adequada para um algoritmo leve. O algoritmo proposto utiliza uma variação do modo paralelizável que considera integridade IAPM – *Integrity Aware Parallelized Mode* (JUTLA, 2001).

O *checksum* original do modo IAPM é calculado diretamente a partir do texto em claro. No algoritmo proposto, antes de calcular o *checksum*, a função de permutação dependente de chave é aplicada. Isso é feito porque, segundo KRAWCZYK (2001), existem três ordens possíveis para criptografia autenticada: (a) EtA (*Encrypt-then-Authenticate*) – cifrar-então-autenticar; (b) AtE (*Authenticate-then-Encrypt*) – autenticar-então-cifrar; e (c) E&A (*Encrypt-and-Authenticate*) – cifrar-e-autenticar. O artigo provou, por exemplos, que as ordens (b) e (c) não são genericamente seguras. A única genericamente segura é a ordem (c).

O método proposto, assim como o IAPM, usa uma sequência $S_0 S_1 S_2 \ldots S_m$ criada a partir de um vetor de inicialização (IV). Essa sequência é utilizada para executar uma operação XOR com os blocos de texto em claro. Ela possui duas funções, a primeira é para deixar a cifragem segura contra ataques de texto cifrado escolhido (CCA) e a segunda é para garantir que o *checksum* seja diferente se a ordem de dois blocos de texto cifrada for modificada. O numero de blocos da sequência $(m + 1)$ deve ser igual ao número de blocos de texto em claro.

No algoritmo proposto, a sequência é gerada da seguinte maneira: a) o IV é submetido à

função de permutação dependente de chave utilizando a chave K_2 (PF_{K_2}); b) o resultado gerado será um bloco contador (CB), que deve ser incrementado pela função $INC32$ para cada bloco; c) o bloco contador é submetido novamente a (PF_{K_2}).

A função $INC32$ divide o bloco contador em sub-blocos de 32 bits. Cada sub-bloco é incrementado em 1. A geração dos blocos da sequência $S_0, S_1, .., S_m$ pode ser paralelizada.

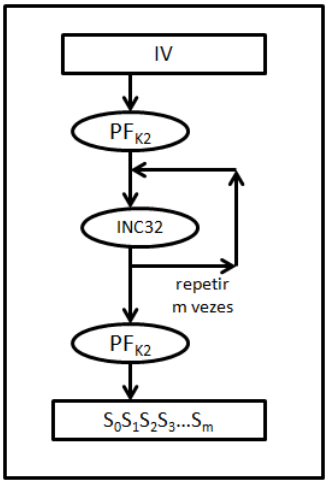

FIG. 3.9 processo gerador da sequência $S_0S_1S_2 \dots S_m$

Para gerar os blocos de texto cifrado, executa-se um XOR entre cada bloco de texto em claro $P_0P_1P_2 \dots P_m$ e os blocos da sequencia $S_0S_1S_2 \dots S_m$. O resultado é aplicado à função PF_{K1} e gera blocos de estado intermediário $st_0st_1st_2 \dots st_m$. Uma operação XOR de todos os blocos intermediários é utilizada para calcular o *checksum* da cifra. Os blocos de estado intermediário $st_0st_1st_2 \dots st_m$ são submetidos à função PF_{K1} para gerar os blocos de texto cifrado $C_1C_2C_3 \dots C_{m+1}$.

A cifra também considera a necessidade de completar o último bloco com um preenchimento (*padding*). O preenchimento tem o formato (10^{ps-1}), onde ps é tamanho do preenchimento em bits. Se o preenchimento for utilizado ($ps \geq 1$), uma operação XOR entre o *checksum* e a constante $(10)^{\frac{n}{2}}$ é executada. Se o preenchimento não for utilizado ($ps = 0$), uma operação XOR entre o checksum e a constante $(01)^{\frac{n}{2}}$ é executada. O resultado desta operação XOR é submitida à função PF_{K1}. O resultado é submetido à função MSB_{Tlen}, que retorna os $Tlen$ bits mais significativos para gerar o tag de tamanho $Tlen$. O tamanho mínimo do tag depende do número máximo de blocos da mensagem (m). A razão $\frac{m}{2^{Tlen}}$ deve ser pequena suficiente para que a probabilidade de colisão seja desprezível para a aplicação da cifra.

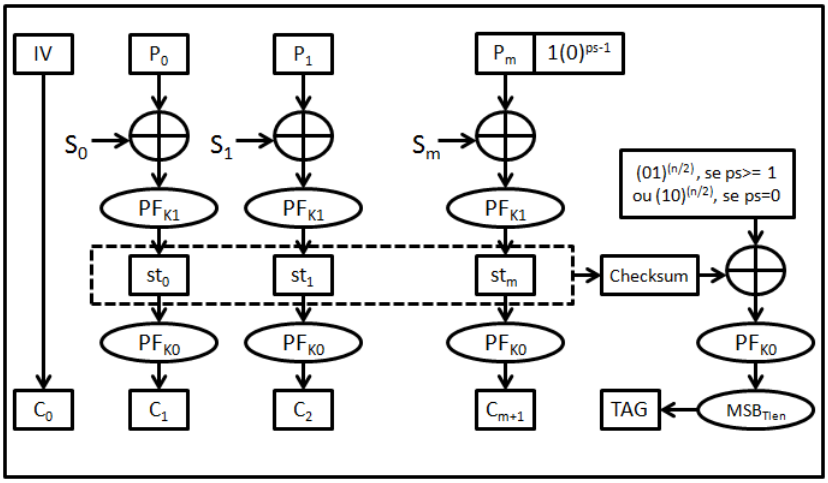

FIG. 3.10 FlexAE - cifra flexível, leve e autenticada proposta

Para garantir a autenticação das mensagens, um processo que impede o reuso do IV deve ser implementado. A maneira mais simples de fazer isso é diminuir o tamanho do IV em 24 bits. O IV seria então concatenado a um contador que seria incrementado para cada mensagem enviada. O número máximo de mensagens enviadas com a mesma chave seria $2^{24} - 1$.

No lugar do IV, pode ser utilizado um número de uso único (NONCE). Isso é vantajoso quando a cifra trabalhar com hardware limitado. Normalmente, esses dispositivos não possuem um gerador de aleatoriedade capaz de gerar um IV confiável. Outra vantagem de utilizar um NONCE é que se o receptor e o transmissor puderem gerar o mesmo número, o primeiro bloco pode ser omitido.

Se para algum uso específico, a autenticação não for necessária, o TAG pode ser omitido. Apesar de não recomendado, a cifra também pode emular o modo de operação ECB. Nesse caso, o IV sempre será 0 e a mensagem precisa ser do tamanho de um bloco e o TAG não deve ser gerado.

A principal vantagem da cifra proposta é seu modo de operação básico, que é flexível leve e prove criptografia autenticada (AE). Por essas razões, ela foi chamada de cifra FlexAE.

4 ANÁLISE COMPARATIVA DE DESEMPENHO DO ALGORITMO

A comparação de desempenho do algoritmo foi feita por meio da ferramenta FELICS (DINU *et al.*, 2015). Essa ferramenta foi proposta pelo grupo de estudos Cryptolux, da Universidade de Luxemburgo para comparar de forma justa cifras leves.

Em (DINU *et al.*, 2015), os autores indicam que um dos motivos para o desenvolvimento da ferramenta foi a dificuldade para comparar os diversos projetos de cifra pois cada um utiliza dados de desempenho utilizando diferentes plataformas e condições de comparação. Ainda segundo os autores, a ferramenta foi projetada para ter as seguintes características chaves:

- Comparação justa – uma metodologia clara indicando como extrair as métricas e quais requerimentos seguir permite comparar de forma igual as implementações.
- Comparação consistente – para manter a consistência, a mesma metodologia é utilizada para medir o desempenho entre as arquiteturas permitidas.
- Medições precisas – a precisão das simulações é medida em ciclos de CPU.
- Grátis *(free)* e código aberto – tanto a ferramenta quando o seu código fonte podem ser distribuídos, utilizados e modificados livremente.
- Transparência – a ferramenta deve permitir total transparência de como as métricas são coletadas.
- Resultados abrangentes – o detalhe das várias métricas extraídas permitem o desenvolvedor da cifra entender qual parte do algoritmo está afetando o desempenho (ex. expansão de subchaves, processo de cifragem, processo de decifragem etc).
- Flexível – utilizar uma arquitetura modular, permitindo o desenvolvimento futuro de outros módulos.
- Avaliação automática – a ferramenta avalia automaticamente os vetores de testes fornecidos pelo implementador da cifra, facilitando o processo de validação dos testes.

A ferramenta FELICS gera três métricas para comparação: tamanho do código em bytes (code); uso de memória RAM em bytes (RAM); e tempo de processamento em ciclos do

processador (time). As métricas podem ser geradas para três cenários: o Cenário0, que gera as métricas para cifrar e decifrar um vetor de teste; o Cenário1, que gera as métricas para cifrar e decifrar 128 bytes de dados utilizando o modo de operação CBC; e o Cenário2, que gera as métricas para cifrar 128 bytes de dados utilizando o modo de operação CTR.

As métricas podem ser geradas para três microcontroladores:

- AVR ATmega128 fabricado pela Atmel que utiliza uma arquitetura RISC de 8-bit.
- MSP430F1611 fabricado pela Texas Instruments que utiliza uma arquitetura RISC de 16-bit.
- ARM Cortex-M3 (SAM3X8) fabricado pela Atmel que utiliza uma arquitetura RISC de 32-bit.

Para gerar as métricas do microcontrolador ARM Cortex-M3 é necessário possuir uma placa Arduino Due Board.

Quatro implementações em linguagem C da cifra proposta foram submetidas à ferramenta: FlexAE_64_128 – bloco de 64 bits e chave de 128 bits, FlexAE_128_256 – bloco de 128 bits e chave de 256 bits, FlexAE_256_512 – bloco de 256 bits e chave de 512 bits e FlexAE_512_1024 – bloco de 512 bits e chave de 1024 bits.

As métricas foram extraídas para dispositivos AVR de 8 bits e MSP de 16 bits. Não foi possível coletar as métricas para dispositivos ARM de 32 bits porque a placa Arduino Due Board que não estava disponível para uso no presente trabalho. O código escrito em linguagem C não foi submetido a técnicas específicas para melhorar o desempenho. Nas implementações, para manter compatibilidade com a ferramenta, a cifra proposta trabalhou no modo de emulação ECB. O *tag* utilizado para autenticação era gerada e descartada para todos os blocos.

Para comparação, os resultados das quatro implementações próprias para o Cenário 1 mais outras 23 implementações de cifras leves, totalizando 27 implementações, foram compilados na tabela TAB. 4.1. As métricas para comparação foram recuperadas da página da ferramenta (FELICS Block Ciphers Brief Results, 2016).

É importante destacar que as implementações disponíveis são otimizadas para as arquiteturas de teste. Inclusive algumas das implementações, como a do AES, foram escritas em linguagem Assembly.

Em relação às métricas de uso de memória, considerando as 27 implementações (23 da página do FELICS e as 4 da cifra proposta), a implementação FlexAE_64_128 é a que utiliza

menos memória de todas as implementações nos dispositivos AVR e MSP. A FlexAE_128_256 está na posição 7/27 para o dispositivo AVR e posição 11/27 para o dispositivo MSP. A FlexAE_256_512 está na posição para dispositivo AVR e 23/27 para dispositivo MSP. A FlexAE_512_1024 está na posição 27/27 para ambos dispositivos.

TAB. 4.1 métricas da ferramenta FELICS no Cenário1

Implementação	AVR			MSP		
	Code	RAM	Time	Code	RAM	Time
AES	3.010	408	58.246	2.684	408	86.506
Chaskey	1.510	229	22.142	1.136	244	23.402
Chaskey-LTS	1.510	229	34.814	1.140	244	37.626
Fantomas	3.520	227	141.838	2.918	222	85.911
FlexAE_64_128	4.728	208	56.338	7.108	208	55.532
FlexAE_128_256	8.026	288	99.790	13.568	288	70.430
FlexAE_256_512	3.450	448	212.372	8.374	448	94.338
FlexAE_512_1024	3.358	768	237.661	3.670	768	374.456
HIGHT	1.414	333	94.557	1.238	328	120.716
LBlock	2.954	494	183.324	1.632	324	263.778
LEA	1.684	631	61.020	1.130	626	47.339
LED	5.156	574	2.221.555	7.004	252	2.065.695
Piccolo	1.992	314	407.269	1.354	310	324.221
PRESENT	2.160	448	245.232	1.818	448	202.050
PRIDE	1.402	369	146.742	2.566	212	242.784
PRINCE	1.954	369	299.322	2.028	236	386.781
RC5-20	2.782	372	275.730	1.240	378	386.026
RECTANGLE	1.152	352	66.722	818	396	45.688
RECTANGLE	1.118	353	64.813	844	402	46.196
RoadRunneR	2.316	209	125.635	3.218	218	222.032
RoadRunneR	2.504	330	144.071	3.088	338	235.317
Robin	2.474	229	184.622	1.900	224	110.527
Simon	1.084	363	63.649	738	360	47.767
Simon	1.122	375	66.613	760	372	49.829
Speck	966	294	39.875	556	288	31.360
Speck	874	302	44.895	572	296	32.333
TWINE	4.236	646	297.265	3.796	564	387.562

Fonte: (FELICS Block Ciphers Brief Results, 2016).

Em relação às métricas de tempo, vale destacar que as implementações FlexAE_64_128 e FlexAE_128_256 são mais rápidas que o AES nos dispositivos MSP de 16 bits. As suas

métricas são 56.338 e 70.430, respectivamente contra 86.506 da implementação em assembler do AES. A implementação FlexAE_64_128 também é mais rápida que o AES no dispositivo AVR de 8 bits. A sua métrica é 55.532 contra 58.246.

O tamanho do código, o uso de memória e o tempo de execução foram comparados com as outras implementações disponíveis na página do FELICS para criar uma tabela de classificação.

TAB. 4.2 tabela com a posição comparativa das implementações FlexAE

Cipher Implementation	**AVR**			**MSP**		
	Code	**RAM**	**Time**	**Code**	**RAM**	**Time**
FlexAE_64_128	25	1	5	25	1	10
FlexAE_128_256	27	7	13	27	10	11
FlexAE_256_512	22	22	20	26	23	14
FlexAE_512_1024	21	27	21	22	27	23

A classificação das métricas permite observar que as implementações da cifra proposta, sem um código fonte otimizado, possuem métricas similares a outras cifras leves. Esse fato permite concluir que a cifra proposta tem características para ser considerada uma cifra leve.

5 VALIDAÇÃO ESTATÍSTICA DO ALGORITMO PROPOSTO

O concurso do NIST, que escolheu o algoritmo para ser AES, definia que a cifra concorrente deveria mostrar que era adequada para a geração de sequências pseudo-aleatórias (SOTO, 1999). Essa característica também é desejada para a cifra proposta.

Uma sequência pseudo-aleatória, como o próprio nome diz, não é realmente aleatória. Ela apenas parece aleatória. A partir de uma sequência de poucos bits de entropia[10], um algoritmo expande em uma sequência de bits muitas vezes maiores. Apesar de seu tamanho ter sido aumentado, a sua entropia continua de poucos bits.

O ideal seria utilizar sempre sequências aleatórias verdadeiras, porém, nem sempre é possível. Conforme definido em (BARKER; KELSEY, 2015):

> Em uma sequência aleatória ideal, cada bit é imprevisível e imparcial, com um valor que é independente dos valores dos outros bits da sequência. Antes da observação da sequência, o valor de cada bit é igualmente provável de ser 0 ou 1, e a probabilidade de um bit ter um valor particular não é afetada pelo conhecimento dos valores de qualquer ou todos os outros bits. Uma sequência aleatória ideal de n bits contém n bits de entropia.

Uma sequência realmente aleatória depende de fenômenos físicos não determinísticos como ruído térmico, ruído atmosférico, tempo de velocidade de acesso de um disco rígido, radiação do decaimento nuclear etc.

Alguns fabricantes de circuitos integrados já incluem componentes específicos para a geração de sequência aleatória. A Intel, por exemplo, em algumas linhas de processadores, fornece um gerador de bits aleatórios baseado em hardware. Para acessá-lo, deve-se utilizar a instrução RDSEED (INTEL CORPORATION, 2014). Mesmo assim, a taxa de geração da sequência pode não ser adequada para as aplicações e esses geradores não estão disponíveis para hardwares mais simples. Por causa dessa complexidade, o mais usual é a utilização sequências pseudo-aleatórias.

Para que a substituição de uma sequência aleatória por uma pseudo-aleatória não afete a aplicação, elas devem ser indistinguíveis. Segundo GOLDREICH (2010), a aleatoriedade não

[10] Entropia é uma medida de aleatoriedade. Quando falamos que uma sequência possui n bits de entropia significa que desta sequência n bits são realmente aleatórios.

é uma propriedade inerente do objeto. Ela depende da capacidade do observador. Quando uma moeda é jogada, pode-se considerar o resultado um evento aleatório. Porém, se o observador possui capacidade de avaliar todas as forças aplicadas a moeda e consegue calcular seu resultado antes dela chegar ao chão, o evento deixará de ser aleatório. Logo, as sequências serão indistinguíveis quando não existir um algum algoritmo eficiente para diferenciá-las. Conforme a teoria da complexidade, um algoritmo eficiente é aquele que pode ser executado em tempo polinomial.

Na prática, para determinar se uma sequência é indistinguível de uma aleatória aplicam-se testes estatísticos na tentativa de identificação de algum padrão. O conjunto de testes estatísticos do NIST (RUKHIN *et al.*, 2010) é um dos mais utilizados para verificação de aleatoriedade de uma sequência.

O teste estatístico é aplicado para provar a hipótese de que a sequência é aleatória. Essa é chamada de hipótese nula (H0) e é associada a uma hipótese alternativa (Ha), que significa que a sequência não é aleatória. Caso a sequência passe no teste, a hipótese nula seria considerada verdadeira.

Mas se uma sequência não passar no teste, não quer dizer que ela não é aleatória ou pseudo-aleatória. Um gerador aleatório ideal, de tempos em tempos, gera sequências que não parecem aleatórias. Nesse caso, ocorre um erro do tipo I – quando a sequência é realmente aleatória, mas o teste estatístico indica que ela não seja. Também existe o erro tipo II que é o inverso, quando a sequência não é aleatória e o teste indica que ela seja.

A probabilidade de um erro tipo I ocorrer é chamada de nível de significância e é representado por α. Normalmente, o valor $\alpha = 0{,}01$ é utilizado em criptografia. Se as sequências de gerador pseudo-aleatória falharem nos testes estatísticos numa taxa maior que $P > \alpha$, na maioria dos casos, deve-se considerar que o gerador é falho. Porém, testes em geradores de sequência aleatória ideal algumas vezes falham numa proporção maior que a esperada. As causas destas falhas são anomalias estatísticas. Logo, antes de descartar um bom gerador incorretamente deve-se tentar identificar se a falha é uma anomalia estatística ou se é evidência de não-aleatoriedade. Para isso o teste deve ser repetido com outra amostragem.

Dois métodos foram utilizados para validar a aleatoriedade da cifra proposta. O primeiro foi o mesmo usado para validar a aleatoriedade dos candidatos ao AES no concurso de seleção do NIST (SOTO, 1999); o segundo foi a aplicação direta da ferramenta Dieharder (BROWN; EDDELBUETTEL; BAUER, 2016) a uma sequência gerada pela cifragem de um contador de 64 bits pelo algoritmo.

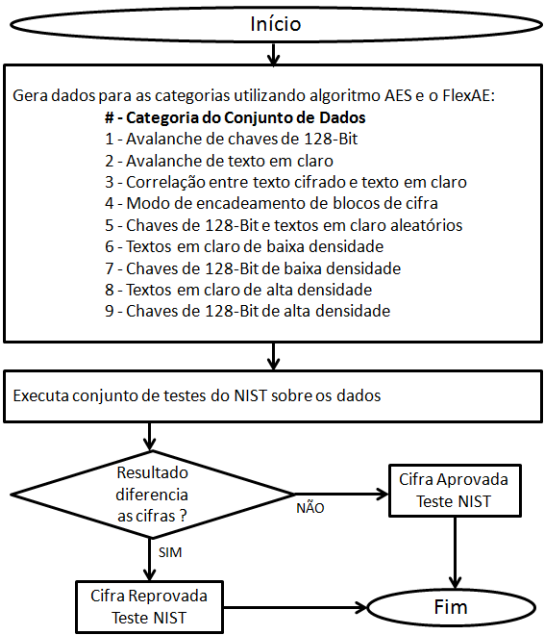

FIG. 5.1 processo de validação da cifra utilizando testes NIST

O método utilizado na validação do AES consiste em gerar nove conjuntos de dados com diferentes categorias de dados. Eles são submetidos ao conjunto de testes estatísticos do NIST (NIST, 2014). Os tipos de testes automatizados pela ferramenta são: teste de frequência (*The Frequency Monobit Test*); teste de frequência dentro de um bloco (*Frequency Test within a Block*); teste de sequências seguidas (*The Runs Test*); teste da maior sequência seguida de uns em um bloco (*Tests for the Longest-Run-of-Ones in a Block*); teste de classificação de matriz binária (*The Binary Matriz Rank Test*); teste da transformada discreta de Fourier (*The Discrete Fourier Transform Spectral Test*); teste de coincidência de padrões não-sobrepostos (*The Non-overlapping Template Matching Test*); teste de coincidência de padrões sobrepostos (*The Overlapping Template Matching Test*); teste estatístico universal de Maurer (*Maurer's "Universal Statistical" Test*); teste de complexidade linear (*The Linear Complexity Test*); teste serial (*The Serial Test*); teste de entropia aproximada (*The Approximate Entropy Test*); teste de somas cumulativas (*The Cummulative Sums Test*); teste de excursões aleatórias (*The Random Excursions Test*); e variação do teste de excursões aleatórias (*The Random Excursions Variant Test*).

As nove categorias dos conjuntos de dados gerados são as seguintes:

- Avalanche de chave de 128-Bit (*128-Bit Key Avalanche*) – esta categoria serve para avaliar o efeito da mudança de cada bit da chave. Para originar os conjuntos de dados dessa categoria, uma chave aleatória é gerada para cifra um bloco de

somente zeros. Então, cada bit da chave é alterado, gerando outras 128 chaves. O bloco de somente zeros é cifrado com estas chaves e é feito um XOR dos resultados com o bloco cifrado com a chave original. O resultado destas operações XOR é guardado na sequência. O processo total é feito com 24.576 chaves aleatórias de forma a gerar 384 sequências de 1.048.576 bits.

- Avalanche de texto em claro (*Plaintext Avalanche*) – essa categoria avalia o efeito da mudança de cada um dos bits do texto em claro. A geração do conjunto de dados é muito similar à geração para a categoria de avalanche da chave, somente que agora a chave é composta de um bloco de somente zeros e o texto em claro é um bloco aleatório. O texto aleatório é gerado e cifrado como uma chave de somente zeros. Cada um dos 128 bits do texto aleatório é alterado independentemente e esses novos blocos são cifrados com a chave de somente zeros. Faz-se um XOR dos blocos cifrados gerados com o bloco cifrado original. O resultado é colocado na sequência. O processo é repetido 24.576 vezes para gerar 384 sequências de 1.048.576 bits.
- Correlação entre texto cifrado e texto em claro (*Plaintext/Ciphertext Correlation*) – nessa categoria, um bloco de texto aleatório e uma chave aleatória são gerados. O texto em claro é cifrado com a chave. Faz-se um XOR do texto em claro com o bloco cifrado. O resultado é gravado na sequência. O processo é repetido até ter um total de 128 sequências de 1.048.576 bits.
- Modo de encadeamento de blocos de cifra (*CBC mode*) – nessa categoria, uma chave aleatória é gerada. Esta chave aleatória é utilizada para cifrar 8.192 blocos de texto em claro contento somente zeros utilizando a cifra no modo operação CBC e como um IV de somente zeros. Os blocos cifrados são guardados na sequência a ser analisada. O processo é executado até gerar 300 sequências de 1.048.576 bits.
- Chaves de 128-Bit e textos em claro aleatórios (*Random Plaintext/Random 128-Bit Keys*) – nessa categoria, uma chave aleatória é gerada para cifrar 8.192 blocos de texto em claro aleatórios. Os blocos cifrados são guardados na sequência. O processo é executado até gerar 128 sequências de 1.048.576 bits. A utilidade dessa categoria pode ser questionada e ela poderia ser eliminada. Porém, para manter o mesmo método utilizado no concurso do AES, ela foi mantida.
- Textos em claro de baixa densidade (Low Density Plaintext) – nessa categoria,

8.257 blocos de baixa densidade são gerados. O primeiro bloco tem apenas zeros, os 128 seguintes apenas um 1-bit e os 8.128 finais apenas dois 1-bit. Esses blocos são cifrados por 128 chaves aleatórias diferentes, produzindo 128 sequências de 1.056.896 bits.

- Chaves de 128-Bit de baixa densidade (*Low Density 128-Bit Keys*) – nessa categoria, o mesmo conjunto de blocos de baixa densidade da categoria anterior é utilizado como chave para cifra um bloco de texto em claro aleatório. O processo é repetido com outros blocos de texto em claro aleatórios até 128 sequências de 1.056.896 bits.
- Textos em claro de alta densidade (*High Density Plaintext*) – essa categoria utiliza o mesmo processo da categoria de textos em claro de baixa densidade. A diferença é que os blocos utilizados são de alta densidade (possuem apenas dois 0-bit, um 0-bit e nenhum 0-bit). Também são geradas 128 sequências de 1.056.896 bits.
- Chaves de 128-Bit de alta densidade (*High Density 128-Bit Keys*) – essa categoria utiliza o mesmo processo da categoria de chaves de 128-Bit textos de baixa densidade, porém, com blocos de alta densidade. Ela é composta de 128 sequências de 1.056.896 bits.

A geração dos conjuntos de dados foi feita por um programa escrito em Java. Apenas com a troca da função que faz a cifragem, ele pode gerar o conjunto de dados para qualquer algoritmo de criptografia de 128 bits de chave e bloco.

TAB. 5.1 número de sequências e tamanho por categoria de dados

#	**Categoria do Conjunto de Dados**	**Número de Sequências**	**Tamanho das Sequências (em bits)**
1	Avalanche de chaves de 128-Bit	384	1.048.576
2	Avalanche de texto em claro	384	1.048.576
3	Correlação entre texto cifrado e texto em claro	128	1.048.576
4	Modo de encadeamento de blocos de cifra	300	1.048.576
5	Chaves de 128-Bit e textos em claro aleatórios	128	1.048.576
6	Textos em claro de baixa densidade	128	1.056.896
7	Chaves de 128-Bit de baixa densidade	128	1.056.896
8	Textos em claro de alta densidade	128	1.056.896
9	Chaves de 128-Bit de alta densidade	128	1.056.896

Os conjuntos de dados gerados para cada categoria foram submetidos pelo conjunto de testes do NIST. Para manter a compatibilidade do método utilizado, apenas a implementação

do FlexAE com bloco de 128 bits e chave de 128 bits foi utilizada. Os resultados mostraram que os conjuntos de dados não puderam ser distinguidos de sequências aleatórias. Para confirmar a validade do experimento, conjuntos de dados equivalentes foram gerados com a cifra AES e submetidos aos testes do NIST. Os resultados foram similares e com isso pode-se concluir que as cifras não podem ser diferenciadas entre elas utilizando este método.

A aleatoriedade da cifra proposta também foi checada com um experimento utilizando a ferramenta de testes estatísticos DieHarder (BROWN; EDDELBUETTEL; BAUER, 2016). Essa ferramenta executa 32 tipos de testes estatísticos. O resultado de cada teste pode ser: "*PASSED*", se a sequência avaliada apresentar boa aleatoriedade; "*WEAK*", se a sequência avaliada apresentar alguma fraqueza; e "*FAILURE*", se a sequência avaliada indicar ser não aleatória.

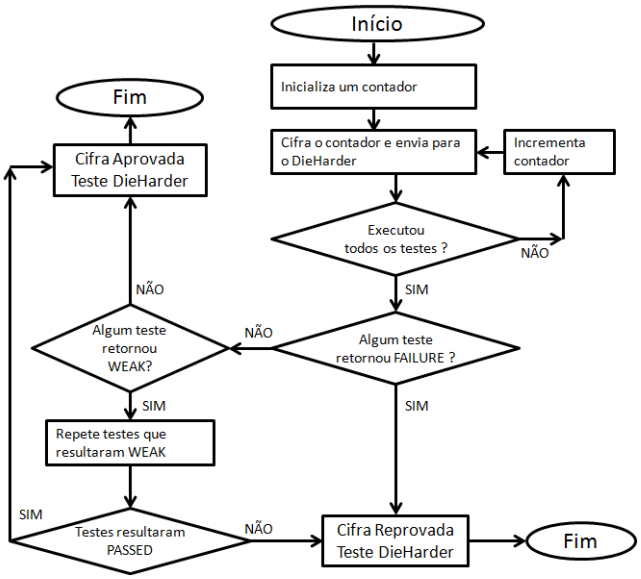

FIG. 5.2 processo de validação da cifra utilizando DieHarder

Um ponto controverso da ferramenta DieHarder é que ele indica fraqueza (*weak*) ou falha (*failure*) caso a sequência falhe nos testes menos que o esperado (p>99,5% e p>99,95 respectivamente). Ou seja, se o gerador produzir mais sequências que parecem aleatórias, o teste pode falhar.

Outro ponto importante é a indicação para refazer o teste caso tenha recebido resultado "*WEAK*". Isso ocorre porque a sequência pode apresentar alguma anomalia estatística. Segundo a própria documentação da ferramenta, o gerador de pseudo-aleatório não deve ser condenado por apresentar alguns resultados "*WEAK*".

Para o experimento, um bloco contador com o tamanho da cifra de bloco é utilizado. A

sequência era alimentada com o bloco contador cifrado. A cada vez que era utilizado, o bloco contador era incrementado em um. A geração continua até que todos os testes fossem executados. Para executar todos os testes, a ferramenta espera como entrada uma sequência de aproximadamente de 300 gigabytes de dados.

O experimento inicialmente foi executado com a cifra AES para servir de comparativo. Ele recebeu "*PASSED*" em todos os testes executados. Depois, o experimento foi repetido com as implementações da cifra proposta de tamanhos de bloco de 64,128, 256 e 512 bits. Em sete testes, a cifra proposta apresentou o resultado "*WEAK*". Quando rodado novamente os testes que falharam, o resultado alterava para "*PASSED*". Como essas falhas são em pequeno número e não se repetem, pode-se considerar que, em partes da sequencia gerada, existem anomalias que a faz parecer ter um pouco mais ou um pouco menos aleatoriedade que o esperado em uma sequência realmente aleatória. Esse tipo falha pode acontecer mesmo em uma sequência realmente aleatória de boa qualidade.

Esses resultados permitem concluir que, apesar de apresentar algumas anomalias estatísticas, a cifra apresenta-se como um gerador de sequência pseudo-aleatória de boa qualidade.

6 RESISTÊNCIA À CRIPTANÁLISE LINEAR E DIFERENCIAL

Os métodos de criptanálise linear e diferencial são poderosas ferramentas de avaliação da segurança de uma cifra. Muitos ataques são baseados nas características lineares e diferenciais do algoritmo. A resistência de um algoritmo depende, em grande parte, da resistência das suas SBox. Neste ponto, a escolha da SBox do AES e de duas variações também baseadas na álgebra no corpo de Galois traz grande vantagem.

Na seção atual, descrevem-se as duas técnicas e os resultados de dificuldade de ataques baseados em criptánalises lineares e diferenciais da cifra FlexAE.

6.1 CRIPTANÁLISE DIFERENCIAL

A técnica de criptanálise diferencial foi publicada inicialmente como artigo em (BIHAM; SHAMIR, 1991). Depois, foi detalhada em livro pelos mesmos autores (BIHAM; SHAMIR, 1993). A técnica consiste em analisar as probabilidades das diferenças das entradas e saídas das funções utilizadas para permutação.

A permutação utilizada pela FlexAE utiliza três SBox de 8 bits x 8 bits. Elas são geradas pelo cálculo da inversa multiplicativa no corpo de Galois, conforme descrito no item 3.1. As três SBox utilizadas encontram-se no Apêndice II.

Para fazer a análise, deve-se considerar um par qualquer de entrada (X, X^*) e suas saídas (Y, Y^*), as diferenças $\Delta X = X \oplus X^*$ e $\Delta Y = Y \oplus Y^*$ são utilizadas para construir a tabela de distribuição das diferenças. Na tabela, os valores de ΔX são as linhas e os valores de ΔY são as colunas. As entradas são preenchidas com a contagem dos possíveis pares $(\Delta X, \Delta Y)$. Para a resistência a um ataque baseado em criptanálise diferencial é maior se as distribuições dos pares $(\Delta X, \Delta Y)$ forem ser uniformes.

Uma tabela de tamanho 256x256 será gerada para cada SBox da cifra FlexAE. As tabelas das SBox da cifra proposta geram distribuições uniformes. Na primeira linha, para qualquer $\Delta X = 0$, todos os ΔY são iguais 0. Isso é esperado porque quando $\Delta X = 0$ significa que $X = X^*$. Neste caso específico, pode-se dizer que $\Delta X = 0 \rightarrow \Delta Y = 0,\ com\ probabilidade\ ou\ p_{\Delta X=0 \rightarrow \Delta Y=0} = 1$. Exceto para a primeira

linha/coluna da tabela, os valores máximos em todos as linhas/colunas é 4, significando que a probabilidade máxima de um par $(\Delta X, \Delta Y)$ qualquer é igual $\frac{4}{256}$ ou $p = 2^{-6}$. O valor é o mesmo para todas as três SBox. Essas probabilidade são importantes para definir a característica da n-rodada de uma cifra.

O código utilizado para gerar as tabelas de distribuição das diferenças das SBox da cifra FlexAE e está disponível no Apêndice III.

TAB. 6.1 tabela de distribuição das diferenças da SBox0 ou SBox do AES (parcial)

Input XOR (ΔX)	Output XOR (ΔY)														
	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0xFA	0xFB	0xFC	0xFD	0xFE	0xFF
0x0	256	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1	0	2	0	0	2	0	2	0	2	2	2	0	0	0	2
0x2	0	0	0	2	2	2	2	2	0	2	0	0	2	2	2
0x3	0	0	2	0	2	2	0	0	2	2	0	0	2	0	2
0x4	0	0	0	0	0	0	0	0	0	2	0	0	0	0	2
0x5	0	0	0	0	2	0	0	0	4	0	0	2	0	2	2
0x6	0	2	0	0	2	2	0	0	0	2	0	2	2	0	2
...															
0xF9	0	0	0	2	2	2	0	0	0	4	0	2	0	0	0
0xFA	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2
0xFB	0	0	0	0	0	2	0	2	2	2	0	0	0	2	0
0xFC	0	2	0	0	2	2	0	0	0	0	0	2	0	2	2
0xFD	0	0	2	2	2	2	0	2	0	0	0	0	2	0	2
0xFE	0	0	0	2	2	0	0	2	2	0	0	2	2	0	2
0xFF	0	2	2	2	0	0	0	2	0	2	0	2	2	2	2

As diferenças ΔX e ΔY formam as características de n-rodadas de uma cifra, onde n é o número de rodadas. Elas possuem uma probabilidade p^{Ω} de ocorrer. As características de texto em claro são expressas pelo símbolo ΩP e as de texto cifrado por ΩT.

Exemplo 6.1: Uma característica de 1-rodada da cifra FlexAE de 64 bits com probabilidade $p^{\Omega} = 1$ é ilustrada na figura FIG. 6.1.

No exemplo 6.1, quando $\Delta X = 0$ nas entradas da SBox0 ou SBox1 ou SBox2, ΔY será sempre 0 ($p_{\Delta X=0 \rightarrow \Delta Y=0} = 1$). O exemplo tem a probabilidade máxima para uma característica de 1-rodada que é $p^{\Omega} = 1$.

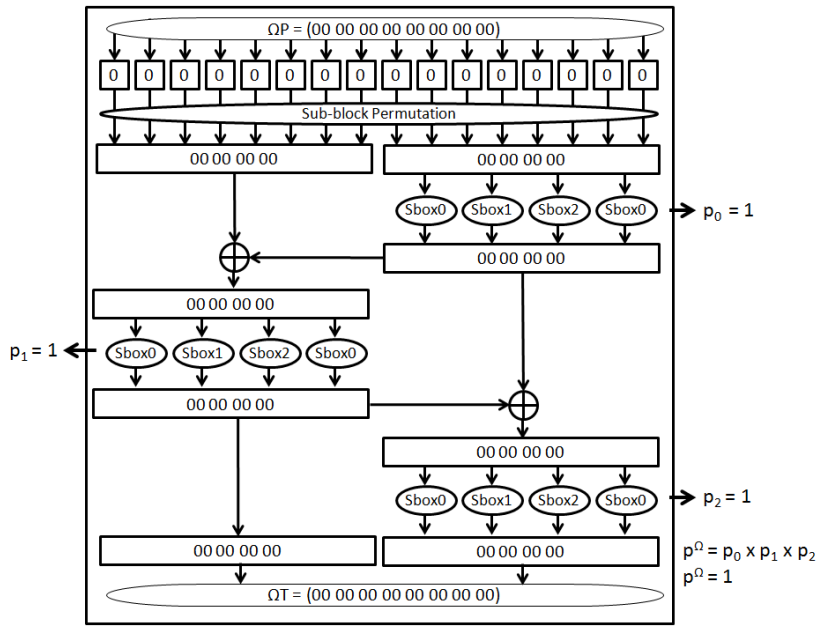

FIG. 6.1 característica da cifra FlexAE de 64bit com $p^{\Omega} = 1$

Exemplo 6.2: Uma característica de 1-rodada da cifra FlexAE de 64 bits com probabilidade $p^{\Omega} = 2^{-12}$ é ilustrada na figura FIG. 6.2.

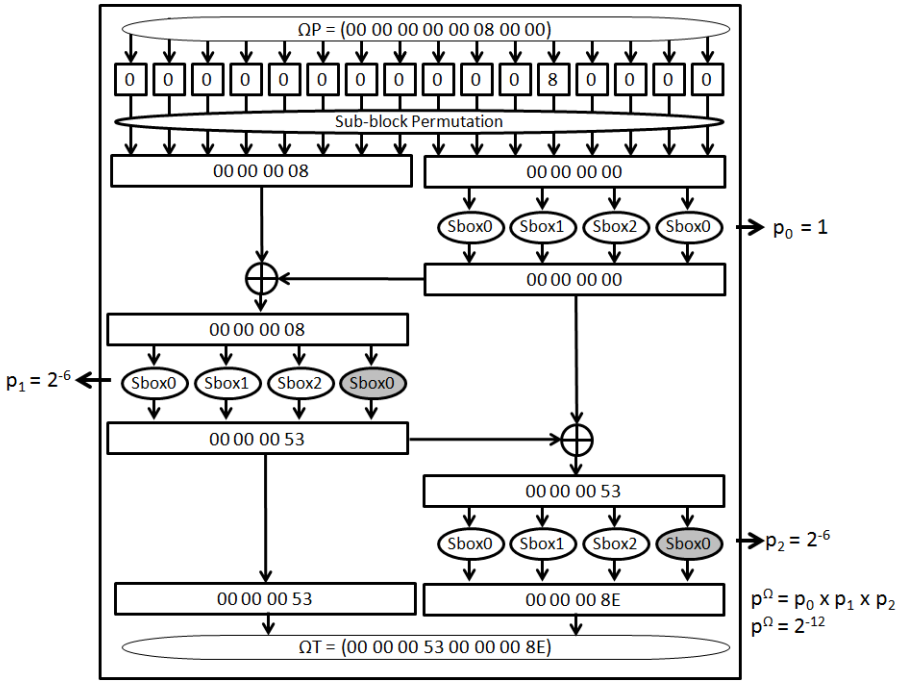

FIG. 6.2 característica da cifra FlexAE de 64 bit com $p^{\Omega} = 2^{-12}$

Para calcular a probabilidade do exemplo 6.2 foi utilizada a definição feita por (BIHAM; SHAMIR, 1991) para a cifra DES.

Definição 6.1: A probabilidade p de $\Delta X \rightarrow \Delta Y$ em 1-rodada da função de permutação é o produto de p_i em qual $\Delta X_i \rightarrow \Delta Y_i$ pelas SBox S_i $(i \in (0,1,..,nb))$ onde $\Delta X = \Delta X_0 \Delta X_1 \ldots \Delta X_{nb}$, $\Delta Y = \Delta Y_0 \Delta Y_1 \ldots \Delta Y_{nb}$ e $nb = tamanho\ do\ bloco\ em\ bytes$.

A partir dos valores das tabelas de distribuição das diferenças das SBox da cifra FlexAE, calcula-se:

$p_0 = 1 \times 1 \times 1 \times 1$, pois $p_{\Delta X=0 \rightarrow \Delta Y=0} = 1$ para qualquer SBox.

$p_1 = 1 \times 1 \times 1 \times \frac{4}{256} = 2^{-6}$, pois $p_{\Delta X=0 \rightarrow \Delta Y=0} = 1$ para qualquer SBox e $p_{\Delta X=0x8 \rightarrow \Delta Y=0x53} = \frac{4}{256}$ para SBox0.

$p_2 = 1 \times 1 \times 1 \times \frac{4}{256} = 2^{-6}$, pois $p_{\Delta X=0 \rightarrow \Delta Y=0} = 1$ para qualquer SBox e $p_{\Delta X=0x53 \rightarrow \Delta Y=0x8E} = \frac{4}{256}$ para SBox0.

$p^{\Omega} = 1 \times 2^{-6} \times 2^{-6} = 2^{-12}$

A probabilidade do exemplo 6.2 é a maior possível (abaixo de $p = 1$) para uma característica de 1-rodada. Assim, como na criptanálise linear, quanto maior o número de SBox ativas menor a probabilidade de um ataque. Em uma rodada, a cifra FlexAE tem um mínimo de duas SBox ativas, que estão destacas em cinza na figura FIG. 6.2.

Para calcular a probabilidade de mais de uma rodada a partir das probabilidades das rodadas independentes, deve-se utilizar a definição a seguir, que foi feita em (BIHAM; SHAMIR, 1993) para a cifra DES e é válida na criptanálise diferencial de outras cifras:

Definição 6.2: para uma característica de n-rodadas Ω, a probabilidade p^{Ω}é o produto das probabilidades de cada rodada $p_{r=i}, onde\ \ i = 1, \ldots, n$:

$$p^{\Omega} = \prod_{\{i=0\}}^{n} p_{r=i}$$

Exemplo 6.3: Uma característica de 2-rodada da cifra FlexAE de 64 bits com probabilidade $p^{\Omega} = 2^{-36}$ é apresentada na figura FIG. 6.3.

Na figura FIG. 6.3 pode-se observar que o número de SBox ativas na primeira rodada é dois. Na segunda rodada o número de SBox ativas aumenta para quatro. As duas rodadas possuem um total de seis SBox ativas.

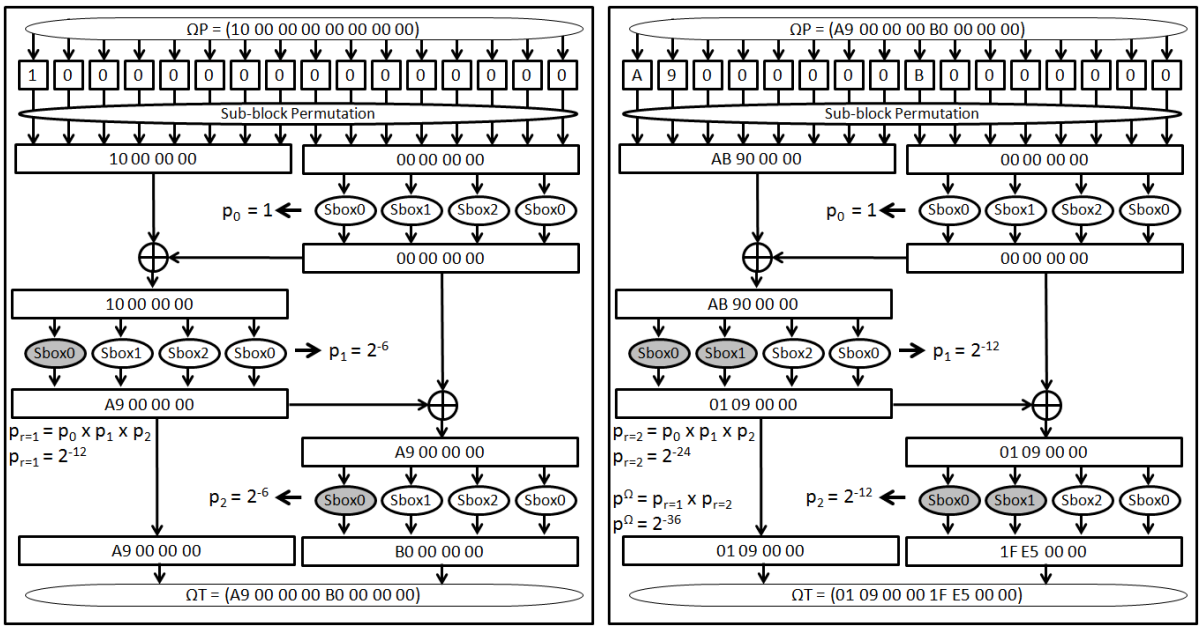

FIG. 6.3 característica de 2-rodada da cifra FlexAE de 64 bits com $p^{\Omega} = 2^{-36}$

O cálculo da probabilidade da primeira rodada é:

$p_0 = 1 \times 1 \times 1 \times 1$, pois $p_{\Delta X=0 \to \Delta Y=0} = 1$ para qualquer SBox.

$p_1 = \frac{4}{256} \times 1 \times 1 \times 1 = 2^{-6}$, pois $p_{\Delta X=0 \to \Delta Y=0} = 1$ para qualquer SBox e $p_{\Delta X=0x10 \to \Delta Y=0xA9} = \frac{4}{256}$ para SBox0.

$p_2 = \frac{4}{256} \times 1 \times 1 \times 1 = 2^{-6}$, pois $p_{\Delta X=0 \to \Delta Y=0} = 1$ para qualquer SBox e $p_{\Delta X=0xA9 \to \Delta Y=0xB0} = \frac{4}{256}$ para SBox0.

$p_{r=1} = 1 \times 2^{-6} \times 2^{-6} = 2^{-12}$

O cálculo da probabilidade da segunda rodada é:

$p_0 = 1 \times 1 \times 1 \times 1$, pois $p_{\Delta X=0 \to \Delta Y=0} = 1$ para qualquer SBox.

$p_1 = \frac{4}{256} \times \frac{4}{256} \times 1 \times 1 = 2^{-12}$, pois $p_{\Delta X=0 \to \Delta Y=0} = 1$ para qualquer SBox, $p_{\Delta X=0xAB \to \Delta Y=0x01} = \frac{4}{256}$ para SBox0 e $p_{\Delta X=0x90 \to \Delta Y=0x09} = \frac{4}{256}$ para SBox1.

$p_2 = \frac{4}{256} \times \frac{4}{256} \times 1 \times 1 = 2^{-12}$, pois $p_{\Delta X=0 \to \Delta Y=0} = 1$ para qualquer SBox, $p_{\Delta X=0x01 \to \Delta Y=0x1F} = \frac{4}{256}$ para SBox0 e $p_{\Delta X=0x09 \to \Delta Y=0xE5} = \frac{4}{256}$ para SBox1.

$p_{r=2} = 1 \times 2^{-12} \times 2^{-12} = 2^{-24}$

A probabilidade das duas rodadas é:

$p^{\Omega} = p_{r=1} \times p_{r=2} = 2^{-12} \times 2^{-24} = 2^{-36}$

Exemplo 6.4: Uma característica de 4-rodada da cifra FlexAE de 64 bits com probabilidade $p^{\Omega} = 2^{-198}$ é apresentada na figura FIG. 6.4.

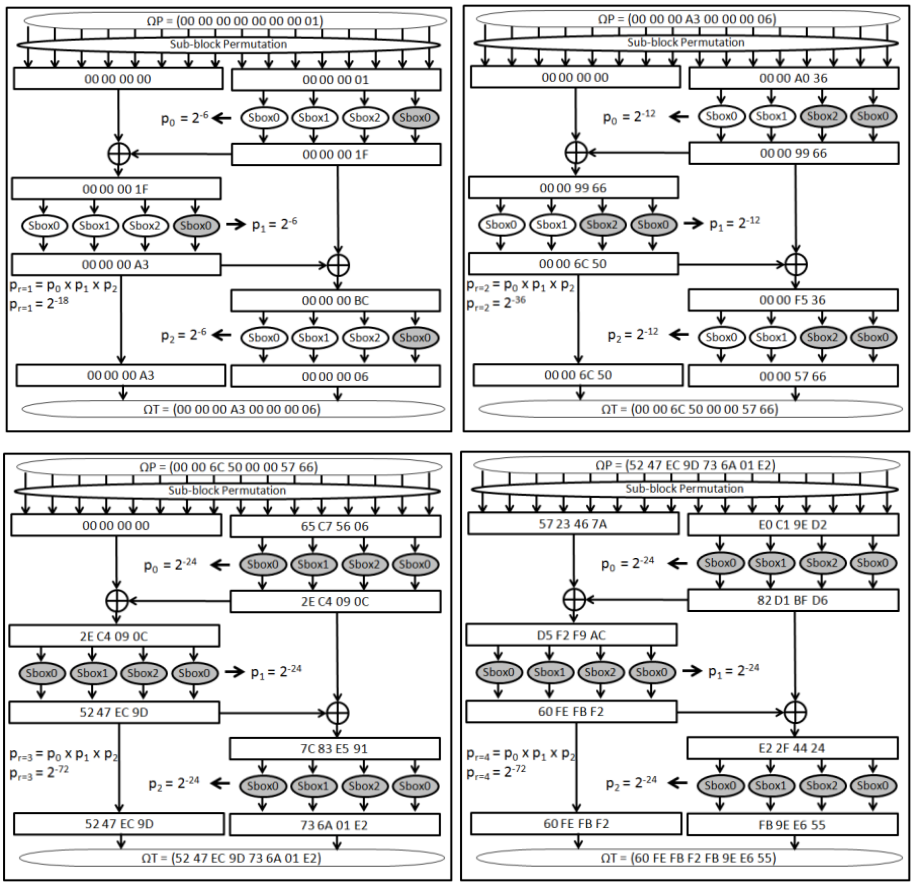

FIG. 6.4 característica de 4-rodada da cifra FlexAE de 64 bits com $p^{\Omega} = 2^{-198}$

Pode-se observar que p^{Ω} pode ser calculado diretamente a partir do número de SBox ativas e a probabilidade máxima (abaixo de 1) de cada SBox. Segundo (HEYS, 2001):

$$p^{\Omega} = \prod_{i=1}^{a} p_i$$

Onde a é o número de SBox ativas e p_i a probabilidade máxima de cada SBox ativa. Considerando que no exemplo 6.4 existem 33 SBox ativas e que a probabilidade máxima de todas as SBox é $p = 2^{-6}$, $p^{\Omega} = (2^{-6})^{33} = 2^{-198}$. O valor é o mesmo que o calculado por rodada. Logo, para calcular a probabilidade de uma característica deve-se apenas saber o número de SBox ativas daquela característica.

Pelos exemplos apresentados, até agora, o mais esperado era que o número de SBox ativas dobre a cada rodada até atingir o máximo de SBox por rodadas. Isso realmente ocorre para um grande número de características, como a do exemplo 6.4. Porém, existem exceções

nas quais o número de SBox ativas se mantém rodada à rodada.

Para o número de SBox ser mantido, as diferenças devem ocorrer apenas nos quatros bits mais significativos dos primeiros bytes das metades do bloco, ou nos quatro bits menos significativos dos últimos bytes das metades do bloco. Isso ocorre porque esses bits nunca são movimentados pela função permutação.

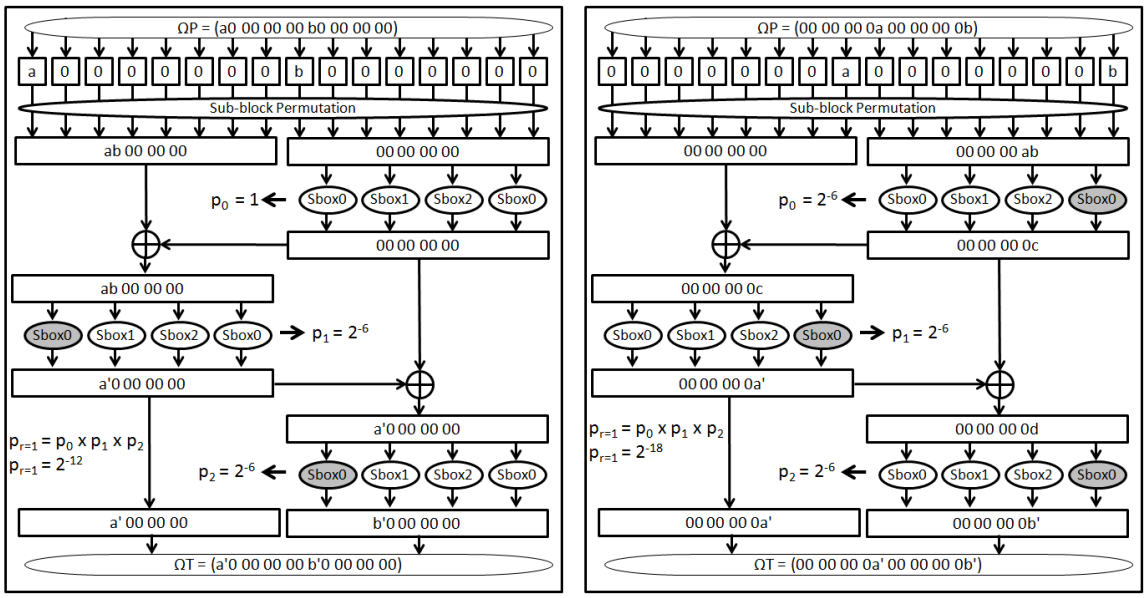

FIG. 6.5 características que mantém o mínimo SBox ativas rodada à rodada

Segundo (HEYS, 2001), a dificuldade de um ataque diferencial depende da probabilidade p_D de uma característica (n-1)-rodada de uma cifra, onde n é o número total de rodadas da cifra. Essa dificuldade pode ser aproximada ao número de blocos de texto conhecidos N_D. Esse número é calculado por $N_D \approx \frac{c}{p_D}$, onde c é uma constante pequena (no pior caso c=1). Para a cifra FlexAE com bloco de 64 bits, a função permutação com 5 rodadas é executada duas vezes, logo, essas características são repetidas um mínimo de 10 rodadas. Como o cálculo deve considerar o número de (n-1) rodadas e que o número mínimo de SBox ativas por rodada é 2, temos: $p_D = \prod_{i=1}^{(2\times 9)} 2^{-6} = (2^{-6})^{18} = 2^{-108}$.

Considerando o pior caso onde $c = 1$, a dificuldade do ataque é $N_D = 2^{108}$. Como N_D é maior que o número possível de blocos (2^{64}), pode-se concluir que esse ataque é impossível.

Para tamanhos de bloco maiores como 256 e 512 bits, o ataque pode ser possível. Considerando que apenas duas SBox estão ativas na penúltima rodada, essas características podem ser utilizadas para descobrir até 16 bits da chave (tamanho da saída das 2 SBox ativas).

A tabela TAB. 6.2 contém a dificuldade do ataque para descobrir até 16 bits da chave

com os vários tamanhos de bloco de cifra FlexAE.

TAB. 6.2 dificuldade de ataque para descobrir até 16 bits da chave

Tamanho do Bloco	Número de Blocos	Rodadas (r-1)	SBox Ativas	Dificuldade do Ataque
64	2^{64}	9	18	2^{108}
128	2^{128}	11	22	2^{132}
256	2^{256}	13	26	2^{156}
512	2^{512}	15	30	2^{180}

Apenas a partir de blocos com tamanho de 256 bits o ataque é possível, porém, em nenhum caso ele é prático porque a dificuldade para descobrir até 16 bits da chave é muito maior que um ataque de força bruta.

Se um ataque para descobrir mais bits da chave for tentado, o número de SBox ativas também aumenta. Por exemplo, para a cifra FlexAE e qualquer outra com tamanho de bloco de 64 bits, um ataque para 64 bits de chave precisa ter 8 SBox 8x8 ativas na (n-1)-rodada (penúltima). Por causa da estrutura da cifra FlexAE, se a (n-1)-rodada tem 8 SBox ativas, a (n-2)-rodada precisa ter metade das SBox ativas da (n-1)-rodada ou 4 SBox ativas. E assim sucessivamente, até chegar ao mínimo de 2 SBox ativas. Isso pode ser visto graficamente na FIG. 6.6.

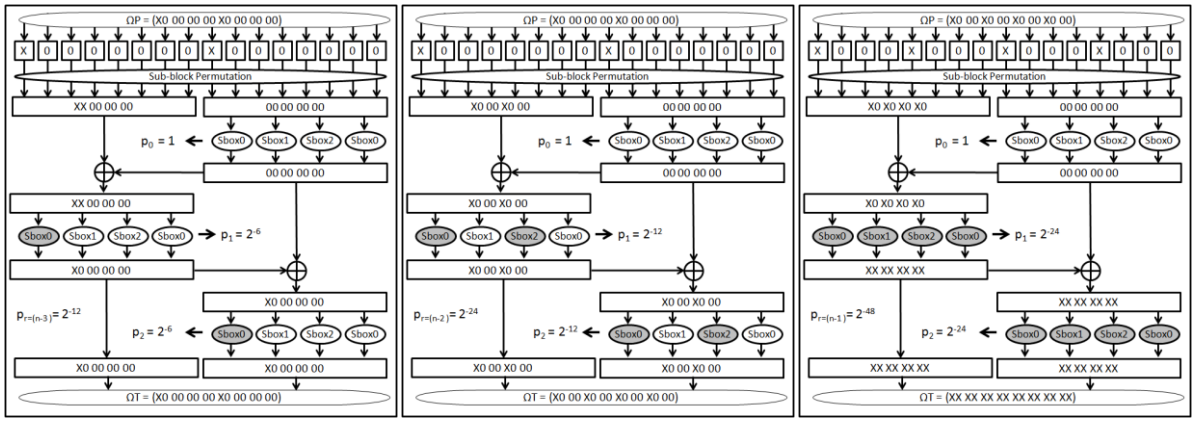

FIG. 6.6 característica de 3-rodada da cifra FlexAE de 64 bits com $p^{\Omega} = 2^{-84}$

A tabela TAB. 6.3 resume a dificuldade dos ataques para descobrir todos os bits da chave utilizados na última rodada da cifra FlexAE com tamanhos de bloco de 64, 128, 256 e 512.

Pode-se verificar pela tabela que a dificuldade de um ataque é sempre maior que a busca exaustiva da chave. Logo, a cifra pode ser considerada resistente a ataques de criptanálise diferencial.

TAB. 6.3 dificuldade de ataque de criptanálise diferencial

Tamanho do Bloco	Número de Blocos	Número de Bits da Chave Atacados	Rodadas	SBox Ativas	Dificuldade do Ataque
64	2^{64}	64	9	26	2^{156}
128	2^{128}	128	11	44	2^{264}
256	2^{256}	256	13	78	2^{468}
512	2^{512}	512	15	144	2^{864}

6.2 CRIPTANÁLISE LINEAR

A criptanálise linear foi proposta em (MATSUI, 1993). Ela consiste em avaliar a tendência de possíveis equações lineares que são utilizadas como aproximações lineares das permutações. Quanto maior a tendência de uma determinada equação linear ser verdadeira, mais provável e fácil será calcular os bits da chave.

Antes da criptanálise linear do algoritmo proposto, o processo será apresentado com uma versão reduzida da função de permutação proposta com bloco de 16 bits. Essa versão reduzida utiliza a SBox da cifra PRESENT(BOGDANOV *et al.*, 2007) que possui tamanho 4x4.

TAB. 6.4 SBox da cifra PRESENT em binário

X_1X_0	X_3X_2			
	00	01	10	11
00	1100	1001	0011	0100
01	0101	0000	1110	0111
10	0110	1010	1111	0001
11	1011	1101	1000	0010

O primeiro passo da criptanálise linear é construir a tabela de aproximação linear da SBox. Isso é feito verificando quais expressões lineares, que utilizam apenas operações XOR, são verdadeiras. Elas são avaliadas contra todas as possíveis entradas e saídas da SBox.

Considerando que a SBox possui entrada $(X_3X_2X_1X_0)$ e saída $(Y_3Y_2Y_1Y_0)$ de 4 bits, um exemplo de expressão linear seria: $X_3 \oplus X_1 \oplus Y_0 = 0$ ou sua equivalente $X_3 \oplus X_1 = Y_0$.

O lado esquerdo da equação linear possui apenas expressões utilizando os bits (X_3, X_2, X_1, X_0), existem ao todo 15 possibilidades $(X_0, X_1, X_0 \oplus X_1, \dots, X_3 \oplus X_2 \oplus X_1, X_3 \oplus X_2 \oplus X_1 \oplus X_0)$ do lado esquerdo. O mesmo é válido para o lado direito da equação na qual existem 15 possibilidades de expressões $(Y_0, Y_1, Y_0 \oplus Y_1, \dots, Y_3 \oplus Y_2 \oplus Y_1, Y_3 \oplus Y_2 \oplus Y_1 \oplus Y_0)$.

Se (a, b) são dois números hexadecimais onde $(a = a_3a_2a_1a_0\ e\ b = b_3b_2b_1b_0)$, as expressões podem ser representadas:

$$(a_3 \times X_3) \oplus (a_2 \times X_2) \oplus (a_1 \times X_1) \oplus (a_0 \times X_0), onde\ a = \{0x1, \dots, 0xF\}$$

e

$$(b_3 \times Y_3) \oplus (b_2 \times Y_2) \oplus (b_1 \times Y_1) \oplus (b_0 \times Y_0)\ , onde\ b = \{0x1, \dots, 0xF\ \}$$

Para montar a tabela de aproximação linear, os resultados das expressões lineares são calculados. Conforme a tabela TAB. 6.5.

TAB. 6.5 resultado das expressões lineares

X	Y	$(a_3 \times X_3) \oplus (a_2 \times X_2) \oplus (a_1 \times X_1) \oplus (a_0 \times X_0)$ a =															$(b_3 \times Y_3) \oplus (b_2 \times Y_2) \oplus (b_1 \times Y_1) \oplus (b_0 \times Y_0)$ b =														
		1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	1100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
0001	0101	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0
0010	0110	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
0011	1011	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1
0100	1001	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0
0101	0000	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0110	1010	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	1	1	0	0	1	1	1	1	0	0	1	1	0	0
0111	1101	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	1	0	1	0	1	0	0	1	0	1
1000	0011	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
1001	1110	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	0	1	1	1	1	0	0	1	1	0	0	0	0	1	1
1010	1111	0	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
1011	1000	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1100	0100	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1101	0111	1	0	1	1	0	1	0	1	0	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1
1110	0001	0	1	1	1	1	0	0	1	1	0	0	0	0	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1111	0010	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1

Com o resultado das expressões lineares, conta-se o número de vezes que a expressão $a \times X = b \times Y$ é verdadeira, para $a = \{0x1,\ 0x2,\ \dots, 0xF\}$, $b = \{0x1,\ 0x2, \dots, 0xF\}$, $X = \{0x0, 0x1, \dots, 0xF\}$. Se a equação linear não apresenta tendência, o resultado da soma para determinados (a, b) fixos deveria ser exatamente metade do número de entradas possíveis na SBox Por isso, deve-se subtrair da soma metade do tamanho da entrada da SBox (nesse exemplo, subtrai-se 8) para obter o resultado das células da tabela de aproximação linear. A tabela de aproximação linear da SBox da cifra PRESENT está representada pela tabela TAB. 6.6.

Os valores de $a = \{0x1, 0x2, \dots\ ,\ 0xF\}$ são representados em cada linha da tabela e os valores $b = \{0x1, 0x2, \dots, 0xF\}$ são representados em cada coluna da tabela. O código utilizado para gerar as tabelas de aproximação linear das SBox da cifra FlexAE e da cifra PRESENT está disponível no Apêndice III.

Uma SBox 4x4 gera uma tabela com 225 expressões lineares possíveis. Já na cifra FlexAE completa, que utiliza três SBox 8x8, o número de expressões lineares é 65025 por

SBox.

TAB. 6.6 tabela de aproximação linear da SBox da cifra PRESENT

a =	b =														
	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	0	0	0	0	-4	0	-4	0	0	0	0	0	-4	0	4
2	0	2	2	-2	-2	0	0	2	-2	0	4	0	4	-2	2
3	0	2	2	2	-2	-4	0	-2	2	-4	0	0	0	-2	-2
4	0	-2	2	-2	-2	0	4	-2	-2	0	-4	0	0	-2	2
5	0	-2	2	-2	2	0	0	2	2	-4	0	4	0	2	2
6	0	0	-4	0	0	-4	0	0	-4	0	0	4	0	0	0
7	0	0	4	4	0	0	0	0	-4	0	0	0	0	4	0
8	0	2	-2	0	0	-2	2	-2	2	0	0	-2	2	4	4
9	4	-2	-2	0	0	2	-2	-2	-2	-4	0	-2	2	0	0
A	0	4	0	2	2	2	-2	0	0	0	-4	2	2	-2	2
B	-4	0	0	-2	-2	2	-2	-4	0	0	0	2	2	2	-2
C	0	0	0	-2	-2	-2	-2	4	0	0	-4	-2	2	2	-2
D	4	4	0	-2	-2	2	2	0	0	0	0	2	-2	2	-2
E	0	2	2	-4	4	-2	-2	-2	-2	0	0	-2	-2	0	0
F	4	-2	2	0	0	-2	-2	-2	2	4	0	2	2	0	0

O valor 0 em uma determinada célula significa que a expressão representada pela expressão $a \times X = b \times Y$ não apresenta tendência e é verdadeira para metade das entradas do SBox. Como exemplo, a célula especificada pelo par $(a = 0x7, b = 0xC)$ é 0, logo a expressão linear $(X_2 \oplus X_1 \oplus X_0 = Y_3 \oplus Y_2)$ é verdadeira para metade das entradas possíveis da SBox. Pode-se dizer que a probabilidade da expressão ser verdadeira é $p = 0{,}5$.

Um valor positivo significa que a expressão linear possui uma tendência de ser verdadeira. Essa tendência é igual ao número contido na célula divido pelo número de entradas possíveis. Na tabela, o valor (4) aparece na célula determinada pelo par $(a = 0xC, b = 0x8)$. Como o número de entradas possíveis da SBox é 16, esse número significa que a expressão linear $X_3 \oplus X_2 = Y_3$ é verdadeira com uma tendência de $\epsilon = \frac{4}{16}$ ou $\epsilon = 0{,}25$ ou $\epsilon = 2^{-2}$. Também pode-se dizer que a probabilidade da expressão ser verdadeira é $p = 0{,}75$.

Um valor negativo significa que a expressão linear possui uma tendência de ser falsa. Essa tendência é igual ao número contido na célula divido pelo número de entradas possíveis. Na tabela, o valor (-4) aparece na célula determinada pelo par $(a = 0x1, b = 0x5)$. Isso significa que a expressão linear $X_0 = Y_2 \oplus Y_0$ é falsa com uma tendência de $\epsilon = \frac{4}{16}$ ou $\epsilon = 0{,}25$ ou $\epsilon = 2^{-2}$. Nesse caso, a probabilidade da expressão ser falsa é $0{,}75$ e a probabilidade de ela ser verdadeira é $p = 0{,}25$.

Nas SBox da cifra FlexAE, a tendência máxima de uma expressão linear ser verdadeira ou falsa por SBox é $\epsilon = \frac{16}{256}$ ou $\epsilon = 0{,}0625$ ou $\epsilon = 2^{-4}$. Apesar de as SBox da cifra proposta serem diferentes, elas apresentam o mesmo número de expressões lineares por tendência. A

tabela TAB. 6.7 possui o número de expressões lineares por tendência para as três SBox da cifra FlexAE e para a SBox da cifra PRESENT.

TAB. 6.7 número de expressões lineares por tendência

Tendência (ϵ)	Número de Expressões Lineares			
	AES SBox ou SBox0	SBox1	SBox2	PRESENT SBox
-4/16 ou -64/256				17
-2/16 ou -32/256				52
-16/256	640	640	640	
-14/256	2040	2040	2040	
-12/256	4592	4592	4592	
-10/256	3064	3064	3064	
-8/256	4334	4334	4334	
-6/256	5096	5096	5096	
-4/256	4592	4592	4592	
-2/256	6112	6112	6112	
0	4080	4080	4080	93
2/256	6128	6128	6128	
4/256	4588	4588	4588	
6/256	5104	5104	5104	
8/256	4336	4336	4336	
10/256	3056	3056	3056	
12/256	4588	4588	4588	
14/256	2040	2040	2040	
16/256	635	635	635	
2/16 ou 32/256				44
4/16 ou 64/256				19
Total de Expressões:	65025	65025	65025	225

O próximo passo é verificar quais são as melhores expressões lineares por rodada. Para fazer isso, verifica-se o efeito da estrutura da cifras nas expressões lineares da SBox. Para exemplificar, será utilizada uma rodada da função de permutação reduzida de 16 bits.

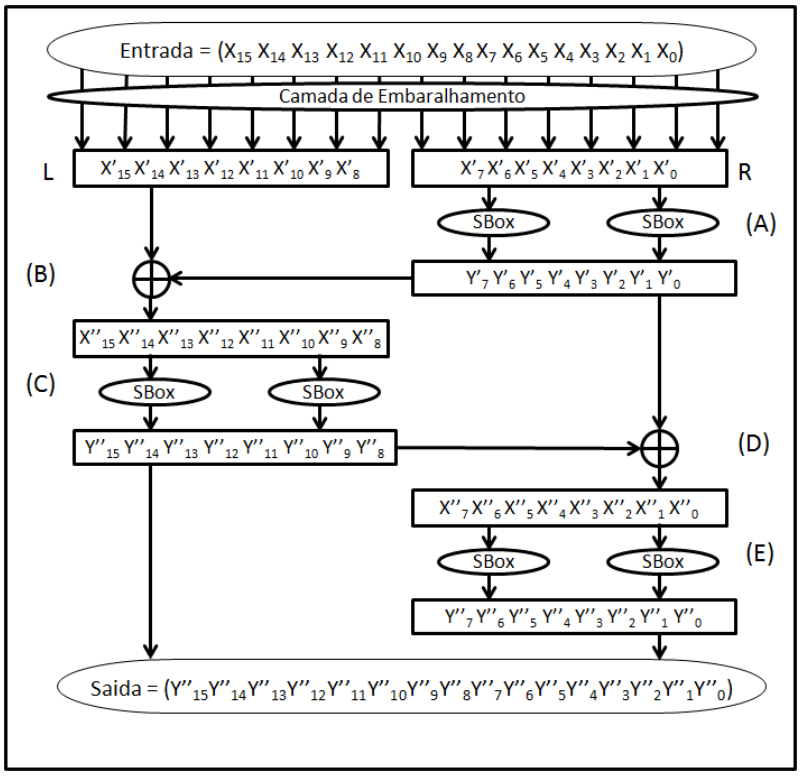

FIG. 6.7 diagrama da versão reduzida de 16 bits da função de permutação

O efeito da camada de embaralhamento pode ser ignorado na análise de uma rodada. Ele deve ser considerado quando for fazer a análise de mais de uma rodada.

Para cada rodada, deve-se verificar quais expressões lineares possuem maior tendência de serem verdadeiras ou falsas.

Na análise, a entrada e os estados intermediários foram separados em duas partes: (L) para esquerda e (R) para direita. Em uma rodada existem 5 operações: (A) – submeter a parte direita (R) às SBox, (B) – fazer $L = L \oplus R$, (C) – submeter a parte (L) às SBox, (D) – fazer $R = L \oplus R$ e (E) – submeter a parte direita (R) às SBox.

Como o número de expressões, mesmo para uma SBox 4x4 são muito grandes, a busca da melhor expressão linear deve ser automatizada. Para exemplificar será calculada a tendência da expressão linear $(X_1 \oplus X_0 = Y_1 \oplus Y_0)$ da PRESENT SBox.

Considerando $X'_{15}, X'_{14}, \dots, X'_0$ como variáveis independentes, a estrutura da cifra reduzida na FIG. 6.7, e $\epsilon = \frac{2}{16}$ ou $\epsilon = 2^{-3}$ da aproximação linear $(X_1 \oplus X_0 = Y_1 \oplus Y_0)$ da SBox da cifra PRESENT, é possível afirmar que, após a operação (A), a tendência da aproximação linear $\left(X'_1 \oplus X'_0 = Y'_1 \oplus Y'_0\right)$ (1) ser verdadeira ou falsa é $\epsilon = 2^{-3}$.

Após a operação (B), $X''_9 = X'_9 \oplus Y'_1$ e $X''_8 = X'_8 \oplus Y'_0$, logo $X''_9 \oplus X''_8 = X'_9 \oplus X'_8 \oplus Y'_1 \oplus Y'_0$ (2). Considerando que $X'_9 \oplus X'_8$ são variáveis independentes, e portanto, não apresentam tendência de serem 0 ou 1, a tendência da expressão ser verdadeira depende de $Y'_1 \oplus Y'_0$, logo $\epsilon = 2^{-3}$.

Combinando (1) e (2): $X''_9 \oplus X''_8 \oplus Y'_1 \oplus Y'_0 = X'_9 \oplus X'_8 \oplus X'_1 \oplus X'_0$ (3). A tendência de (3) ou da sub-rodada composta pelas operações (A) e (B) é $\epsilon_{(A),(B)} = 2^{-3}$. Um fato importante a ser observado é que caso o lado direito da entrada seja mantido fixo, a tendência da rodada não será afetada pela tendência desta sub-rodada.

Aplicando a operação (C) e seguindo a estrutura pelo lado esquerdo: $Y''_9 \oplus Y''_8 = X''_9 \oplus X''_8$ (4), com $\epsilon = 2^{-3}$. Substituindo (3) em (4): $Y''_9 \oplus Y''_8 = X'_9 \oplus X'_8 \oplus X'_1 \oplus X'_0$ (5).

Aplicando a operação (D) e seguindo a estrutura pelo lado direito: $X''_1 = Y''_9 \oplus Y'_1$ e $X''_0 = Y''_8 \oplus Y'_0$, logo $X''_1 \oplus X''_0 = Y''_9 \oplus Y''_8 \oplus Y'_1 \oplus Y'_0$.

A tendência das operações (C) e (D) é calculada de maneira similar à tendência das operações (A) e (B). Nesse caso, o resultado numérico é igual: $\epsilon_{(C),(D)} = \epsilon_{(A),(B)} = 2^{-3}$. O valor da tendência desta sub-rodada como da anterior depende apenas da tendência da SBox

que está ativa. Diz-se que uma SBox está ativa[11] se a entrada de uma aproximação linear passar pela SBox.

Na operação (E), a expressão $X_1'' \oplus X_0''$ é entrada de uma SBox, logo na saída $Y_1'' \oplus Y_0'' = Y_9'' \oplus Y_8'' \oplus Y_1' \oplus Y_0'$ (6), com $\epsilon_{(E)} = 2^{-3}$. No caso dessa sub-rodada, existe apenas a aplicação da SBox para a metade direita do estado interno. A tendência é apenas da SBox ativa.

A combinação de (5) e (6): $Y_9'' \oplus Y_8'' \oplus Y_1'' \oplus Y_0'' = X_9' \oplus X_8' \oplus X_1' \oplus X_0'$ (7) é uma das muitas aproximações lineares de uma rodada da cifra.

Para determinar a tendência de (7) ser verdadeira ou falsa, deve-se utilizar o Piling-up Lemma[12]:

$$p = \frac{1}{2} + 2^{3-1} \times \epsilon_{(A)(B)} \times \epsilon_{(C)(D)} \times \epsilon_{(E)} = \frac{1}{2} + 2^2 \times (2^{-3})^3 = \frac{1}{2} + 2^{-7}, \text{ logo } \epsilon = 2^{-7}$$

Em várias rodadas, deve-se contar o número de SBox ativas pelas expressões lineares de cada rodada, suas tendências e, então, deve-se aplicar o Piling-up Lemma. Sabendo a tendência máxima de uma SBox e o número mínimo de SBox ativas para as rodadas, pode-se definir o limite inferior da tendência de uma cifra, sem precisar verificar todas as aproximações lineares da cifra. Esse método normalmente é utilizado devido à inviabilidade de verificar todas as possibilidades de expressões lineares.

Uma aproximação linear também pode ser representada graficamente. Como a cifra reduzida possui 16 bits, as sua entrada, saída e estados intermediários é representado por 16 quadrados. Os bits que fazem parte da expressão linear são pintados de cinza.

Para facilitar o entendimento, o processo do exemplo anterior pode ser representado graficamente conforme a figura FIG. 6.8.

---

[11] O conceito de SBox ativa é similar na criptanálise linear e na criptanálise diferencial mas as SBoxes ativas e as probabilidades não são as mesmas.

[12] O Piling-up Lemma (MATSUI, 1993) define a probabilidade de uma equação no formato $X_1 \oplus X_2 \oplus \ldots \oplus X_n = 0$ ser verdadeira, considerando todas as variáveis independentes. Segundo o Piling-up Lemma a probabilidade da equação ser verdadeira é:

$$p = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \left(p_i - \frac{1}{2}\right)$$

A partir da probabilidade deve-se subtrair 0.5 para encontrar a tendência(ϵ).

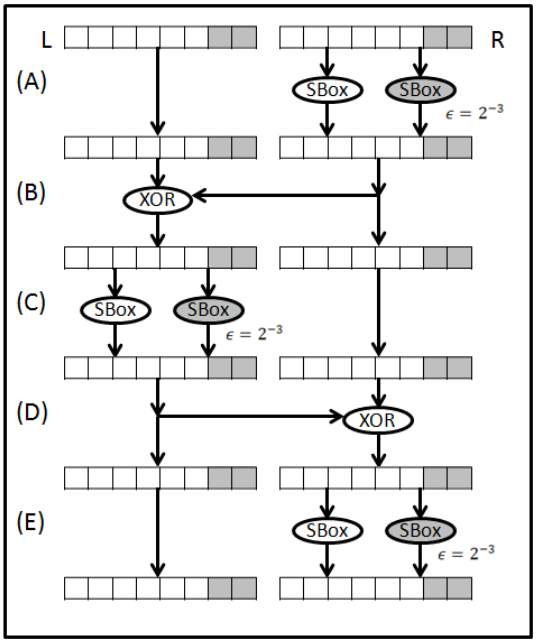

FIG. 6.8 representação de $Y_9'' \oplus Y_8'' \oplus Y_1'' \oplus Y_0'' = X_9' \oplus X_8' \oplus X_1' \oplus X_0'$

A figura FIG. 6.8 pode ser simplificada ainda mais se cada SBox ativa for representada com uma seta. Neste caso fica mais legível se os estados forem divididos pela metade e colocados em duas linhas. Essa simplificação está ilustrada na figura FIG. 6.9.

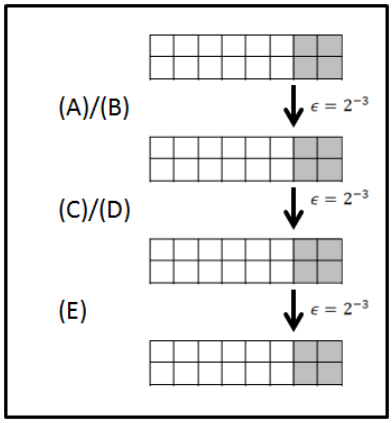

FIG. 6.9 outra representação de $Y_9'' \oplus Y_8'' \oplus Y_1'' \oplus Y_0'' = X_9' \oplus X_8' \oplus X_1' \oplus X_0'$

As aproximações lineares da FlexAE, com seus vários tamanhos de bloco, também podem ser representadas graficamente. A aproximação linear da SBox0 $Y_2 \oplus Y_0 = X_2 \oplus X_0$ com $\epsilon = \frac{12}{256}$ da cifra proposta com bloco de 64 bits é representada pela figura FIG. 6.10.

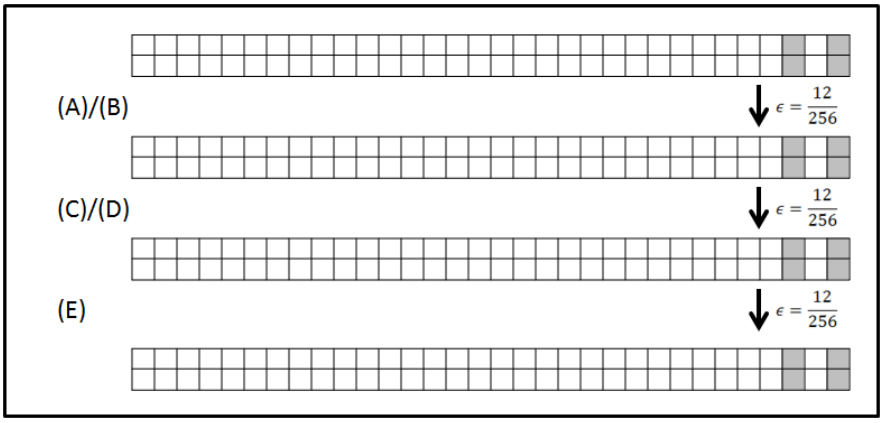

FIG. 6.10 representação gráfica de $Y_2 \oplus Y_0 = X_2 \oplus X_0$ na FlexAE com bloco de 64 bits

A tendência da expressão ser verdadeira ou falsa pode ser calculada utilizando o Piling-up Lemma:

$$p = \frac{1}{2} + 2^2 \times \left(\frac{12}{256}\right)^3 = \frac{1}{2} + 2^2 \times \frac{2^6 \times 3^3}{2^{24}} = \frac{1}{2} + \frac{27}{2^{16}} \approx \frac{1}{2} + 2^{-11}, logo\ \epsilon \approx 2^{-11}$$

Considerando que as SBox da cifra proposta apresentam uma tendência máxima de $\epsilon = \frac{16}{256} = 2^{-4}$ e o número mínimo de SBox ativas por rodada é 3. O Piling-up Lemma também permite calcular o máximo de tendência de uma rodada da função de permutação da cifra proposta. Nesse caso $\epsilon = 2^2 \times (2^{-4})^3 = 2^{-10}$.

Pela dificuldade do processo de criptanálise linear e a quantidade de expressões que podem ser geradas quando a cifra possui um tamanho de bloco grande e utiliza SBox 8x8, pode tornar inviável a tentativa de calcular exatamente a resistência da cifra um ataque baseado em criptanálise linear. Uma possibilidade para fazer o cálculo é determinar a tendência máxima da cifra completa. Se esse número for suficientemente pequeno para inviabilizar um ataque, pode-se considerar a cifra segura. Para fazer este cálculo, assim como no cálculo da tendência máxima de uma rodada, utiliza-se o Piling-up Lemma e o número de mínimo de SBox ativas da cifra com todas as suas rodadas.

Para determinar o número mínimo de SBox ativas em mais de uma rodada, deve-se avaliar o efeito da camada de embaralhamento no total de rodadas necessárias para cifrar um texto em claro. Na cifra FlexAE, o número de rodadas que a função permutação é executada depende do tamanho do bloco utilizado, por exemplo com blocos de 64 bits são necessária 5 rodadas. Logo, considerando que a tendência de cada rodada é $\epsilon_r = 2^{-10}$, a tendência

máxima da função permutação para cinco rodadas da FlexAE com blocos de 64 bits é $\epsilon = 2^4 \times (2^{-10})^5 = 46$

Como a camada de embaralhamento divide o byte de saída da SBox em dois, as expressões lineares que afetam apenas os quatro bits mais significativos ou apenas os quatro bits menos significativos do byte são as que geram o menor número de SBox ativas. Nessa situação o número de SBox ativas por rodada se mantém no mínimo. Isso pode ser visto na FIG. 6.11.

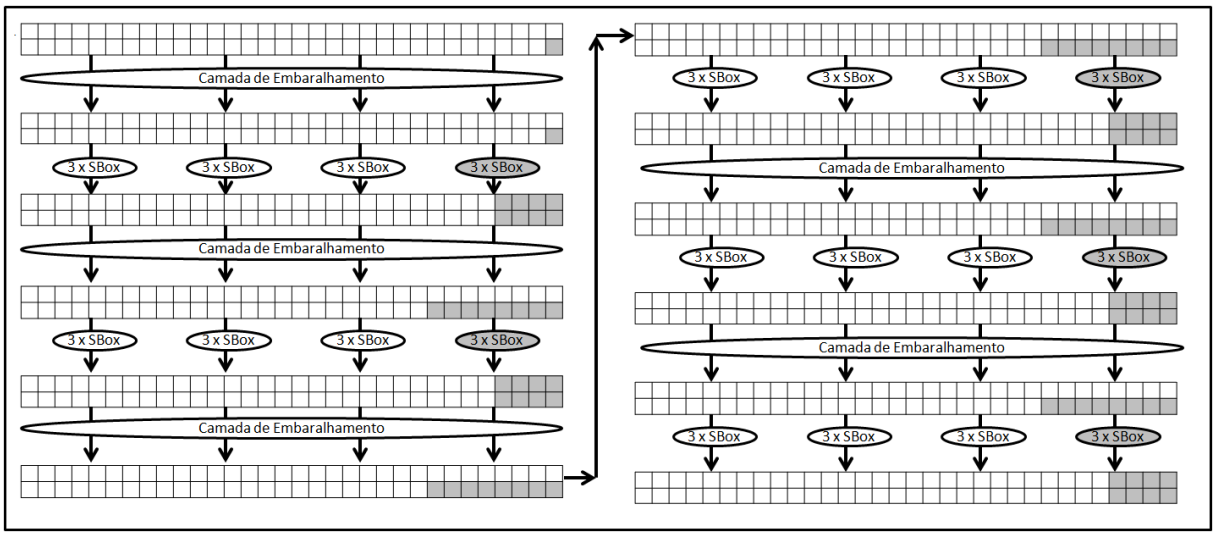

FIG. 6.11 SBox ativas para 5 rodadas da cifra FlexAE com bloco de 64 bits

O número de SBox ativas para a função de permutação da cifra FlexAE é três vezes o número de rodadas da função permutação. O número de rodadas depende do tamanho do bloco da cifra. A partir destes números foi possível calcular a tendência máxima da aproximação linear da cifra utilizando o Piling-up Lemma. O resultado do cálculo da tendência máxima para os vários tamanhos de bloco está disponível na tabela TAB. 6.8.

TAB. 6.8 tendência máxima para a função de permutação

Tamanho do Bloco	Rodadas	SBox Ativas	Tendência Máxima
64	5	15	$\epsilon = 2^{-46}$
128	6	18	$\epsilon = 2^{-55}$
256	7	21	$\epsilon = 2^{-64}$
512	8	24	$\epsilon = 2^{-73}$

Na cifra FlexAE, a função permutação é executada duas vezes para cifrar cada bloco de texto em claro (veja na FIG. 3.10 que P0 passa por PF_{K1} e PF_{K0} para gerar C1). Logo, para calcular tendência máxima da cifra deve-se considerar a função da permutação com o dobro

de rodadas.

Segundo (HEYS, 2001), a dificuldade de um ataque baseado em criptanálise linear pode ser determinada pelo número de textos em claro conhecido. Ainda segundo Heys, este número é $N_L = \frac{1}{\epsilon^2}$.

TAB. 6.9 blocos necessários para um ataque de criptanálise linear contra a cifra FlexAE

Tamanho do Bloco	Número de Blocos	Tamanho Máximo da Chave	Rodadas	SBox Ativas	Tendência Máxima	$N_L = \frac{1}{\epsilon^2}$
64	2^{64}	128	10	30	$\epsilon = 2^{-91}$	$N_L = 2^{182}$
128	2^{128}	256	12	36	$\epsilon = 2^{-109}$	$N_L = 2^{218}$
256	2^{256}	512	14	42	$\epsilon = 2^{-127}$	$N_L = 2^{254}$
512	2^{512}	1024	16	48	$\epsilon = 2^{-145}$	$N_L = 2^{290}$

Pela tabela TAB. 6.9, pode-se ver que o número de pares de blocos necessários para o ataque é maior que o número pares possíveis para os tamanhos de bloco de 64 e 128 bits. Logo, o ataque por criptanálise linear não é aplicável.

Quando a cifra trabalha com tamanhos de bloco maiores que 128, o ataque é aplicável e a dificuldade de um ataque é menor que o tamanho máximo da chave. A dificuldade apresentada na tabela é para encontrar 16 bits da chave de sessão da última rodada. Para os blocos de 256 bits e 512 bits é necessário repetir o ataque 16 vezes ou 32 vezes. A dificuldade para achar todos os bits de sessão de última rodada final para bloco de 256 bits é 2^{258} e para bloco de 512 bits é 2^{295}. Essas dificuldades devem ser consideradas como os limites de segurança para os tamanhos das chaves.

7 USO DO ALGORITMO FlEXAE NO AMBIENTE IoT

Este capítulo propõe um sistema que utiliza o algoritmo FlexAE para proteger comunicações em um ambiente IoT. Além do ambiente de IoT, o algoritmo também pode ser utilizado em redes SCADA. A cifra pode resolver o problema de utilização de criptografia pelos equipamentos industriais, que apesar de muito discutido ainda não é aplicado por causa do custo computacional. O sistema também permite simplificação das redes industriais eliminando a necessidade de segregação.

A proposta de aplicação considera que a comunicação não terá conexão na camada de transporte de rede. Como o ambiente de IoT, os dispositivos podem ser móveis e trocar de endereço requereria refazer a conexão. A confiabilidade das redes utilizadas pelos dispositivos pode ser baixa. Se houver perda de pacotes ou erros, um protocolo de transporte TLS/SSL irá requerer retransmissões, tornando ineficiente a comunicação.

O sistema proposto também considerou a possibilidade de utilização de comunicação *multicast*. Com isso o envio de comandos para grupos vários sensores torna-se mais eficiente que o envio para cada sensor de maneira independente.

Deve-se observar que em um ambiente IoT, os atuadores e sensores podem receber comandos de mais de um dispositivo de controle. Outro ponto que não deve ser esquecido é o fato dos dispositivos sensores/atuadores poderem ter hardware com restrições. Nesse caso é prudente assumir que eles não podem gerar boas sequências pseudo-aleatórias para serem utilizadas em chaves ou como vetores de inicialização (IV). Os cálculos complexos para utilização de chaves públicas e privadas também podem ser inadequados para esses equipamentos.

Tomando um exemplo da rede SCADA, segundo (PIÈTRE-CAMBACÉDÈS; SITBON, 2008) existem quatro métodos típicos para troca de chaves: a) troca de chaves baseada em servidor; b) arquitetura ponto-a-ponto; c) PKI padrão (infraestrutura de chave publica); e d) PKI customizada. Apesar dos autores resaltarem que nenhuma das opções serve para tudo, suas conclusões apontam que as opções baseadas que podem utilizar somente criptografia simétrica (a e b) apresentam uma pequena vantagem sobre as outras.

O SKMA (DAWSON *et al.*, 2006) é uma arquitetura de distribuição de chaves baseada em servidor que utiliza somente criptografia simétrica. A proposta é utilizar uma variação da

arquitetura. A modificação foi necessária para suportar comunicação multicast.

No esquema modificado, existe um servidor de distribuição de chaves (KDS) responsável por controlar o processo de distribuição de chaves para a comunicação entre os dispositivos. O tipo de comunicação pode ser: a) unicast – entre o KDS e cada dispositivo; b) unicast – entre os dispositivos; c) multicast – do KDS para um grupo de dispositivos; ou d) multicast – entre um dispositivo e um grupo de dispositivos:

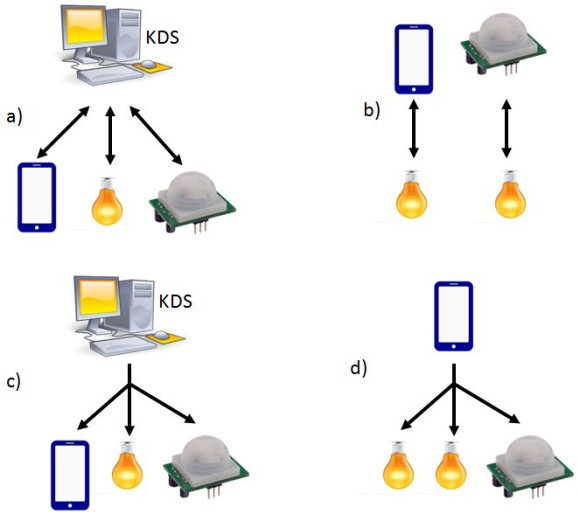

FIG. 7.1 tipos de comunicação entre dispositivos e o KDS

A mensagem utilizada na comunicação será cifrada utilizando o algoritmo FlexAE. Para cada origem e destino existirá chave $K_{origem/destino}$ e um vetor de inicialização ($IV_{origem/destino}$). O KDS gerará o $K_{origem/destino}$ e o $IV_{origem/destino}$ pela necessidade de terem boa aleatoriedade. O IV para cifrar a mensagem será a combinação do $IV_{origem/destino}$com o ID da mensagem. O processo de cifragem está na FIG. 7.2.

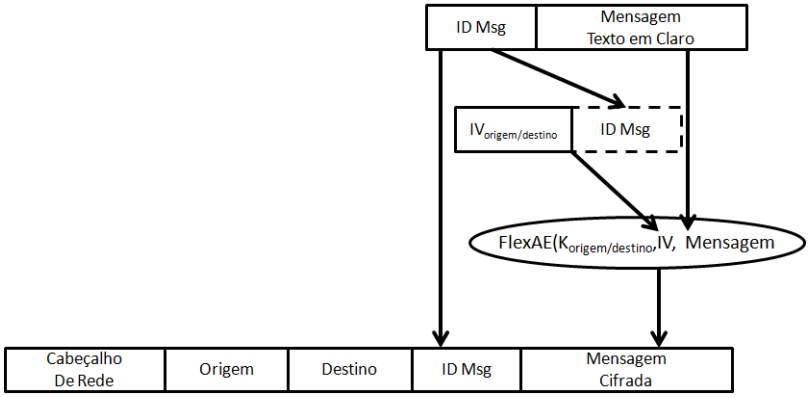

FIG. 7.2 processo básico de cifragem da mensagem

Para evitar ataques de reenvio de mensagem (replay attacks), o identificador da mensagem (ID Msg) é utilizado para validar que a mensagem é única. Este ID Msg pode ser um carimbo de tempo (timestamp) – se a mensagem fosse muito velha significa que alguém está tentando reutilizar a mensagem. Essa estratégia exige que todos os dispositivos tivessem sincronismo de relógio, resultando em um sistema mais complexo e que necessitaria de mais recursos. Além disso, nem todos os dispositivos podem ter um relógio interno em seu hardware.

Outra solução mais simples é o uso de um contador como identificador da mensagem (ID Msg). Se o número repetir, a mensagem deve ser descartada. O tamanho do contador é importante pois toda vez que ele repetir, a chave deve ser trocada toda vez . O tamanho proposto para utilização é 24 bits porque permite o envio de mais de 16 milhões de mensagens sem a necessidade de troca de chave. Cada mensagem pode conter mais de um bloco.

O controle do contador é feito na recepção da mensagem. Ele nunca deve ser menor que o da mensagem anterior (considerando que a comunicação é interativa, as mensagens sempre devem chegar em ordem). Como pode haver perda de mensagens na rede, por ser uma comunicação sem conexão, deve-se ter uma tolerância para receber mensagens com ID maior. Essa tolerância deve ser algo entre 32 e 256 mensagens. Como exemplo, assumirá o valor de 64.

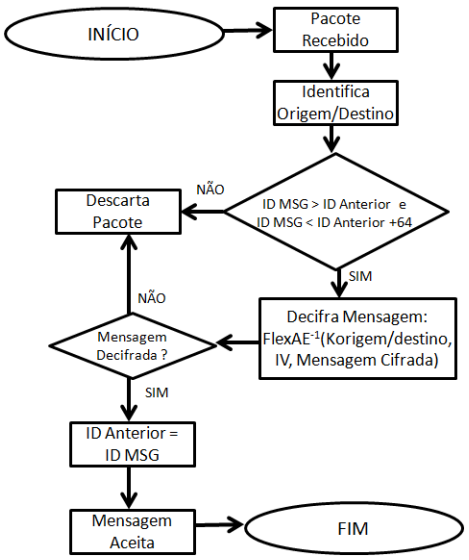

FIG. 7.3 processo de aceite de mensagens recebidas

O descarte de mensagens com o contador fora do aceitável pode ser feito de imediato, porém, o contador interno de controle de recepção somente deve ser atualizado após receber

uma mensagem que foi decifrada corretamente. Somente mensagens válidas são decifradas porque a cifra FlexAE integra a autenticação. Se a mensagem for decifrada, ela é aceita. O diagrama com o processo de aceite de uma mensagem está na FIG. 7.3.

Antes de iniciar o processo de comunicação entre os dispositivos, deve haver uma configuração inicial do dispositivo com o KDS (FIG. 7.4). O sistema pré-supõe que cada dispositivo possui uma chave definida em fábrica (FDK). A cópia desta FDK é carregada no KDS pelo administrador do sistema. Essa FDK é disponibilizada pelo fabricante por meio eletrônico em um canal secundário seguro. Considerando a utilização da cifra FlexAE com bloco de 64 bits e chave de 128 bits, a FDK é composta por uma chave simétrica K de 128 bits mais um IV de 40 bits. A FDK é simétrica e é feita para ser utilizada apenas na configuração do dispositivo. Durante o processo de configuração, o KDS gera uma chave de longa duração para comunicação com o dispositivo (LTK). A LTK é composta da chave $K_{origem/destino}$ de 128 bits e o $IV_{origem/destino}$ de 40 bits. Antes de iniciar o processo de configuração, o dispositivo deve encontrar o endereço do KDS. Se eles estiverem em uma rede IP, o endereço do KDS pode ser perguntado por *broadcast* na rede ou informado ao momento que o dispositivo adquire um IP pelo protocolo DHCP (usando a opção 43 – *vendor specific*).

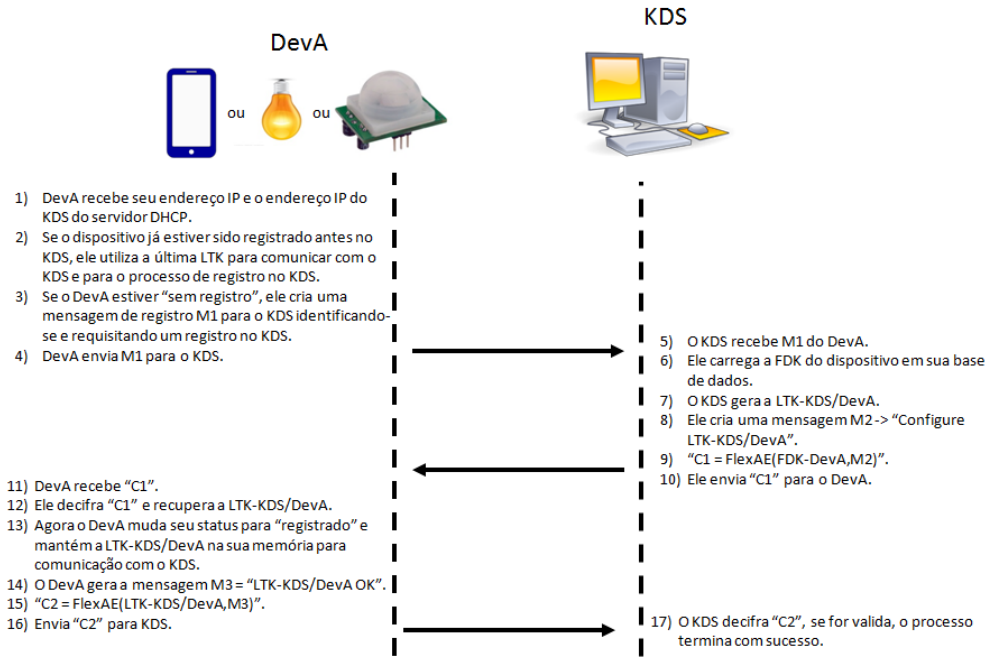

FIG. 7.4 processo de configuração inicial do dispositivo

A LTK gerada no processo de registro pode ser utilizada para configurar (FIG. 7.5) ou revogar uma chave de sessão SK entre o KDS e o dispositivo. Assim, como a FDK e LTK, chave SK é composta por $K_{KDS/DevA}$ de 128 bits e $IV_{\frac{KDS}{DevA}}$ de 40 bits.

A LTK também pode ser utilizada para desconfigurar um dispositivo, permitindo que ele utilize novamente sua FDK no processo de registro automático em outro KDS. Após a configuração da LTK entre o KDS e o dispositivo, a chave FDK do dispositivo pode ser descartada do KDS.

O KDS também pode também se comunicar com entidades externas. Estas entidades externas podem ser a companhia de energia elétrica ou a polícia. O KDS gera uma LTK para cada entidade externa. A LTK é enviada para as entidades externa por um canal secundário seguro, como um email cifrado.

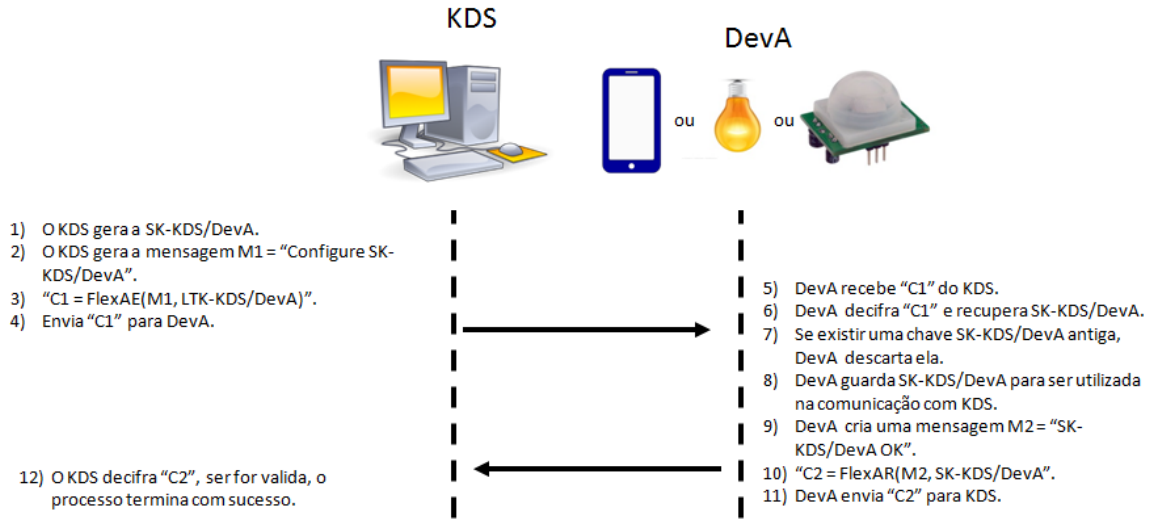

FIG. 7.5 processo de configuração da chave de sessão SK entre o KDS e um dispositivo

A chave de sessão SK é utilizada para todas as outras comunicações entre o KDS e o dispositivo. Ainda haverá uma chave SK para cada fluxo de comunicação ponto-a-ponto e para cada grupo *multicast*. O processo de configuração de chave SK entre dois dispositivos está na FIG. 7.6.

O KDS não precisa guardar as chaves de sessão SK das comunicações ponto-a-ponto. Ele somente precisa gerá-las e após isso ele deve descartá-las. Apesar do KDS não guardar as chaves de sessão SK ponto-a-ponto, ele deve conhecer o fluxo de comunicação de dados para gerar e informar os participantes da comunicação.

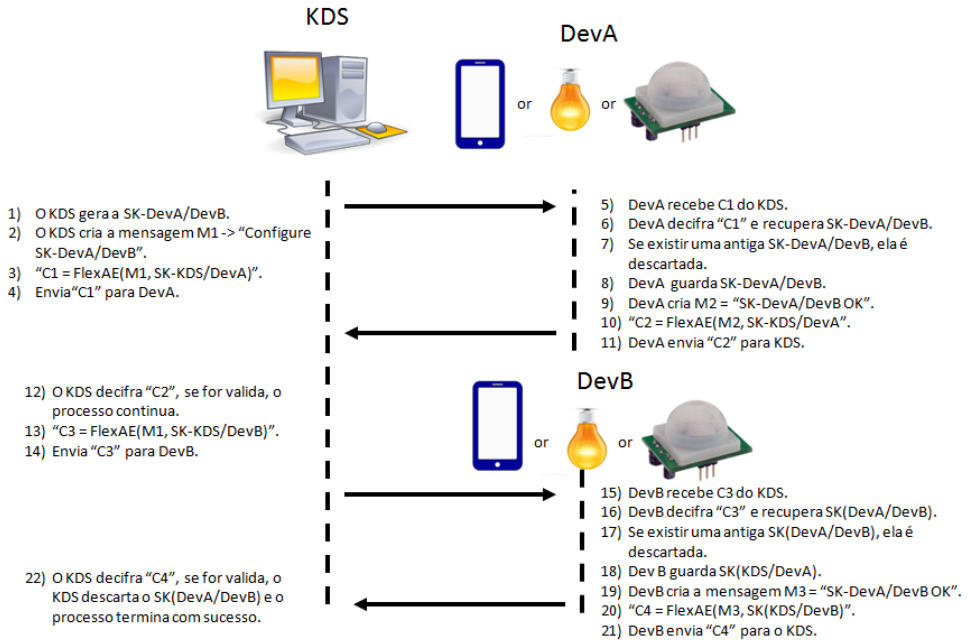

FIG. 7.6 processo de configuração da chave de sessão entre dois dispositivos.

No caso dos grupos *multicast*, ele deve ter uma cópia das chaves para enviar aos novos membros do grupo assim que ele se conectar ao sistema. A geração e o processo de configuração da chave do grupo *multicast* é similar ao processo da geração de chave de sessão para comunicação ponto-a-ponto, conforme a FIG. 7.7.

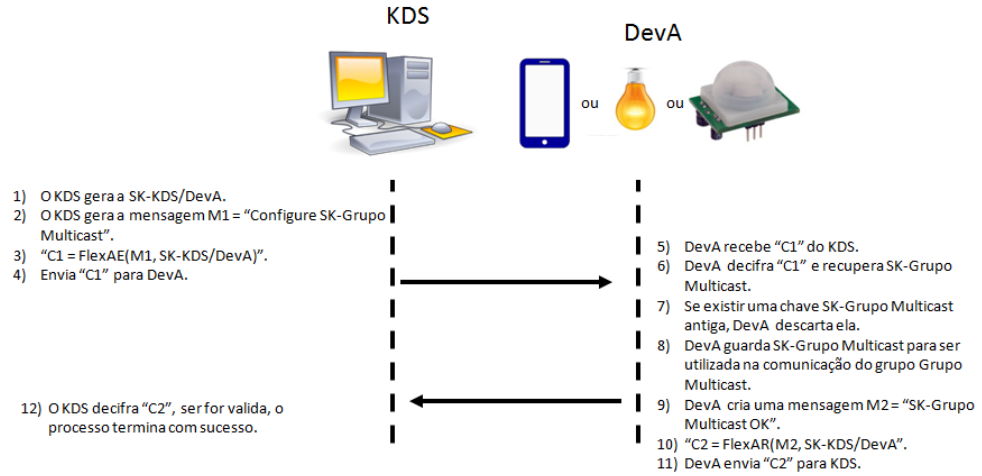

FIG. 7.7 processo de configuração da chave de sessão de um grupo multicast

Um exemplo de grupo *multicast* é "Luzes Externas Acessíveis à Polícia", esse grupo pode ser utilizado pela polícia para acender as luzes externas das casas durante uma ronda noturna na vizinhança. Outro exemplo de grupo é "Dispositivos Não-Essenciais", esse grupo

pode ser utilizado pela companhia de energia para desligar todos os dispositivos não essenciais em caso de sobrecarga do sistema visando evitar um apagão massivo.

Em relação aos grupos *multicast* internos, uma empresa pode ter um grupo "Luzes do 7º andar". Esse grupo permite a equipe de segurança apagar ou acender todas as luzes do sétimo andar.

Outra utilização para grupos multicast é a transmissão de canais de rádio ou TV por Internet ou ainda a transmissão de fluxos de câmeras de segurança para vários monitores.

Um dispositivo pode pertencer a vários grupos *multicast*. Para cada grupo o dispositivo deve-se manter uma chave de sessão SK.

A tabela TAB. 7.1 resume as mensagens utilizadas entre os vários componentes do sistema.

TAB. 7.1 mensagens utilizadas no sistema

Origem	Destino	Mensagem	Chave de Criptografia
Dispositivo	KDS	Pedido de registro de dispositivo	Nenhuma (texto em claro)
KDS	Dispositivo	Configuração de LTK	FDK
Dispositivo	KDS	Confirmação de LTK	LTK-KDS/Dispositivo
KDS	Dispositivo	Pedido de desconfiguração de dispositivo	LTK-KDS/Dispositivo
Dispositivo	KDS	Confirmação de desconfiguração	Nenhuma (texto em claro)
KDS	Dispositivo	Configuração de SK-KDS/Dispositivo	LTK-KDS/Dispositivo
Dispositivo	KDS	Confirmação de SK-KDS/Dispositivo	LTK-KDS/Dispositivo
KDS	Dispositivo	Revogação de SK-KDS/Dispositivo	LTK-KDS/Dispositivo
Dispositivo	KDS	Confirmação de revogação	LTK-KDS/Dispositivo
KDS	Dispositivo	Configuração de SK-Dispositivo/Dispositivo	SK-KDS/Dispositivo
Dispositivo	KDS	Confirmação de SK-Dispositivo/Dispositivo	SK-KDS/Dispositivo
KDS	Dispositivo	Revogação de SK-Dispositivo/Dispositivo	SK-KDS/Dispositivo
Dispositivo	KDS	Confirmação de revogação	SK-KDS/Dispositivo
KDS	Dispositivo	Configuração de SK-Grupo Multicast	SK-KDS/Dispositivo
Dispositivo	KDS	Confirmação de SK-Grupo Multicast	SK-KDS/Dispositivo
Dispositivo	Dispositivo	Mensagens de Comando/Controle	SK-Dispositivo/Dispositivo
Dispositivo	Grupo De Dispositivos	Mensagens de Comando/Controle	SK-GrupoMulticast

Após a troca das chaves de sessão SK, mesmo com o servidor KDS indisponível, os sensores/atuadores ainda podem ser controlados.

Na figura FIG. 7.8, é possível ver onde as chaves ficam guardadas nos vários componentes do sistema. O número de chaves que o servidor KDS precisa guardar é igual ao dobro do número de dispositivos mais o número de grupos *multicast*. Cada dispositivo de controle deve guardar um número de chaves igual ao número de dispositivos de controlado

mais dois (LTK e SK com o KDS). Cada dispositivo controlado deve guardar duas chaves mais o número de comunicações ponto-a-ponto e *multicast* que ele participa.

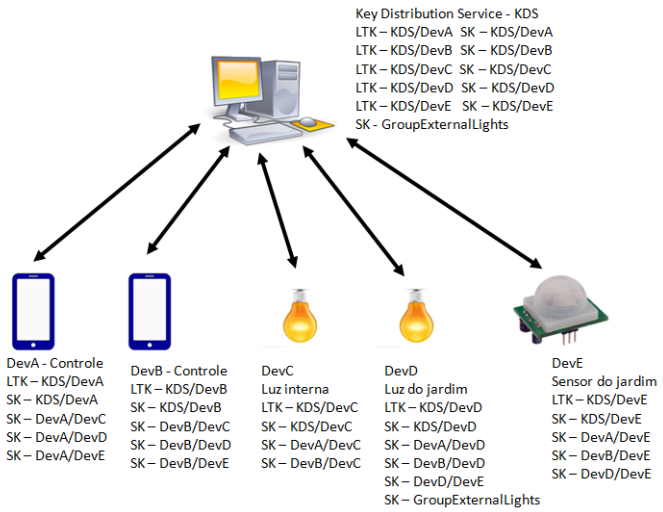

FIG. 7.8 localização das chaves em cada componente do sistema

Como toda a comunicação crítica do sistema proposto neste capítulo é autenticada pelo algoritmo FlexAE, ele pode ser utilizado por dispositivos diretamente conectados à Internet. O mesmo sistema pode ser utilizado para conectar equipamentos industriais diretamente à rede interna ou à Internet evitando a necessidade de criar redes convencionais SCADA ou segregar redes por *firewall*. Isso simplifica as arquiteturas de rede, resultando indiretamente em sistemas mais robustos e baratos. O fato de não exigir conexão na camada de transporte e permitir comunicação *multicast* resulta em maior eficiência de comunicação em diversas situações como para controlar um grupo de lâmpadas de um andar.

O sistema fica simples pelo fato de utilizar uma cifra que dispõe da autenticação integrada à cifra.

8 CONCLUSÃO

O objetivo principal deste trabalho foi construir uma cifra leve, flexível e autenticada. A criptografia leve trabalha em ambientes restritivos onde o custo computacional e uso de memória da cifra devem ser minimizados. A cifra FlexAE proposta utiliza uma função permutação que também foi projetada para ser flexível, permitindo vários tamanhos de bloco e chaves. A autenticação foi integrada à cifra, com a utilização de uma técnica baseada em vários modos de operação existentes, mantendo-a leve. Como qualquer cifra, a FlexAE pode ser utilizada como um bom gerador de sequências pseudo-aleatórias. Em comparação com outras cifras, além de ser mais leve que várias implementações, inclusive uma otimizada em Assembly da cifra AES, ela tem a grande vantagem de integrar autenticação no seu modo básico de operação e permitir tamanhos de blocos. A cifra também apresentou boa resistência contra ataques de criptanálises diferencial e linear. Como objetivo secundário, um sistema para proteger a comunicação de dispositivos no ambiente IoT com a utilização da cifra FlexAE foi proposto.

O primeiro ponto a destacar da função de permutação projetada para a cifra foi o processo utilizado para embaralhar os sub-blocos antes de submetê-los à camada de SBox. O desafio foi mantê-lo o mais simples possível de maneira a não impactar no custo computacional da cifra. É possível misturar todos os bits do bloco de entrada com o mínimo de rodadas possíveis. Este processo também permite aumentar o tamanho do bloco, dando flexibilidade à cifra.

A camada de SBox utilizou três SBox, sendo a primeira idêntica a utilizada pela cifra AES. As outras duas foram geradas por processos muito similares, também utilizando a álgebra no corpo de Galois. A escolha dessas SBox e a utilização de uma rede similar a Feistel nesta camada ajudaram a garantir a segurança da cifra contra ataques baseados em criptánalise diferencial e linear.

Para deixar a função de permutação dependente de senha, a construção Even-Mansour foi utilizada. Nessa construção, para aplicar a chave, utiliza-se apenas a operação XOR somente duas vezes. Além de ser uma solução leve, ela ainda traz a vantagem de requerer poucas subchaves. Isso minimiza o custo do processo de geração de subchaves e a necessidade de memória, em comparação a outras cifras como a PRESENT e o AES.

O processo de geração das subchaves utiliza a própria função de permutação flexível. Isso gera um reuso de código, minimizando o tamanho do código. A geração de subchaves também é flexível, permitindo que a cifra trabalhe com vários tamanhos de chave, sendo o tamanho mínimo de 128 bits e o tamanho máximo duas vezes o tamanho do bloco. Conseguir essa flexibilidade com poucos recursos de memória e processamento pode ser considerado uma inovação.

A integração da autenticação utilizou elementos de vários modos de operação. A geração da sequência $S_0, S_1, \dots, S_m$ é similar à maneira utilizada pelo modo CTR, porém com um processo mais leve. A principal diferença é que a cifra utiliza um contador simples, enquanto o CTR utiliza um processo mais complexo, como um contador baseado na álgebra no corpo de Galois.

Selecionar uma autenticação integrada ao invés de uma autenticação composta traz vantagem no custo computacional da cifra porque requer um número menor de operações. Além disso, a autenticação integrada também utilizou técnicas para manter-se com o menor custo possível. A geração do "*checksum*" com um XOR entre todos os blocos, é muito leve do ponto de vista computacional. Ele estaria sujeito a um ataque de troca de ordem dos blocos da mensagem, mas isso foi evitado com o XOR com a sequência $S_0, S_1, \dots, S_m$. Ele também estaria sujeito a ataques de texto-conhecido, mas, como antes do XOR o bloco é cifrado, o ataque fica inviável. O fato de cifrar o "*checksum*" para gerar o tag evita ataques do tipo copiar-e-colar como os que afetam o modo CBC-MAC. O método de *padding* utilizado é similar ao do CMAC, tendo a principal diferença a utilização de chaves fixas $(01)^{\frac{n}{2}}$ e $(10)^{\frac{n}{2}}$, que diminui a complexidade do processo.

Para verificar se a cifra FlexAE era realmente leve, quatro implementações da cifra em linguagem C, com tamanhos de bloco de 64, 128, 256 e 512 bits, foram comparadas à 27 implementações de outras cifras. Para esta comparação, a ferramenta FELICS do grupo de pesquisa CRYPTOLUX foi utilizada. Um ponto a destacar foi o uso mínimo de memória RAM. A implementação da FlexAE com 64 bits de bloco utilizou menos memória que todas as outras. Talvez o principal ponto a destacar desta comparação seja o em relação ao tempo de processamento, as implementações em C da FlexAE com blocos de 64 e 128 bits, compiladas para a plataforma AVR de 16 bits, foram melhores inclusive que uma implementação do AES escrita em Assembler e otimizada para a plataforma em questão.

Foram feitas avaliações para verificar a capacidade da cifra de gerar sequências pseudo-aleatórias. A primeira avaliação foi reproduzir um dos experimentos que o NIST utilizou para

avaliar os candidatos ao AES. Esse experimento gerou várias sequências categorizadas de dados cifrados com a FlexAE e submeteu ao conjunto de testes estatísticos do NIST. Para comparação, o experimento foi repetido com a cifra AES e os resultados verificados foram similares. A cifra passou nesta primeira avaliação.

A segunda avaliação utilizou a ferramenta de validação de geradores pseudo-aleatórios DieHarder. Os resultados da FlexAE indicaram que ela pode ser utilizada como um gerador de pseudo-aleatório de boa qualidade. Novamente, os resultados da cifra AES e da FlexAE foram comparados. Foi observado que os resultados do AES são marginalmente melhores.

Também foi avaliada a segurança da cifra contra ataques baseados em criptanálises diferencial e linear. No capítulo 6, a dissertação explicou como aplicar as duas técnicas de criptanálise na cifra. Este capítulo pode ser utilizado como material didático sobre o tema

A cifra apresentou resistência a ataques de criptanálise diferencial. A dificuldade de um ataque utilizando esta técnica contra as versões da FlexAE que possuem tamanho máximo de chave de 128, 256, 512 e 1024 bits foram, respectivamente, $2^{156}, 2^{264}, 2^{468}$ *e* 2^{864}. Esses valores são maiores que um ataque de força bruta e que a quantidade de pares de blocos de texto em claro/texto cifrado tornando os ataques inviáveis.

Em relação à criptanálise linear, os ataques à cifra trabalhando com blocos de 64 e 128 bits também são inviáveis pela quantidade de pares de blocos necessários. Eles necessitariam respectivamente de 2^{182} *e* 2^{218}. Quanto a cifra FlexAE trabalha com blocos de tamanho maiores que 128 bits, um ataque pode ser viável e sua dificuldade pode ser menor que o tamanho máximo permitido da chave pela definição da cifra. Logo, nesses casos, o tamanho máximo da chave deve ser limitado à dificuldade do ataque para garantir que ela seja maior que um ataque de força bruta. No caso de blocos com tamanho de 256 e 512 bits, a dificuldade de um ataque para recuperar todos os bits da última subchave aplicada é, respectivamente, $2^{256}e\ 2^{295}$. A cifra continuará segura se uma chave de 256 bits for utilizada para esses casos.

Um sistema que aplica a utilização da cifra FlexAE em um ambiente IoT foi descrito no capítulo 7. Essa descrição é teórica e uma implementação ainda precisa ser feita para validar o sistema. De qualquer maneira, pode-se observar que, pelo fato de utilizar uma cifra com autenticação integrada, se um pacote foi recebido, ele é válido e de uma fonte autorizada. Esse fato traz uma enorme vantagem para o sistema. Tudo pode ser definido de maneira mais simples. Se a troca de chave for segura, toda a comunicação estará segura. Este capítulo atendeu ao objetivo de apresentar uma aplicação à cifra proposta.

8.1 CONTRIBUIÇÕES DO TRABALHO

A primeira contribuição a destacar do trabalho foi a integração da autenticação à uma cifra leve no seu modo básico de operação. Tal construção foi submetida como artigo e foi apresentada no simpósio IEEE Internacional Conference of Communications em maio de 2017, em Paris (IEEE ICC2017 Technical Symposia Program, 2017).

A segunda contribuição é criar a possibilidade da mesma cifra trabalhar com tamanhos variáveis de bloco e chave, mantendo as características de uma cifra leve. Isso permite ajustar a chamada da cifra conforme a aplicação. Trabalhar com blocos de 256bits pode ser adequado para transmissão de vídeo em alta definição, mas para comandos de controle de um atuador um bloco de 64 bits é melhor.

A terceira contribuição é o sistema de proteção da comunicação de dispositivos no ambiente IoT proposto no capítulo 7 para aplicação da cifra. Ele pode servir de base para uma nova geração de dispositivos que já sairá preparada para o ambiente IoT. Ele tem como vantagem simplificar, de forma segura, a integração de dispositivos industriais à rede, tornando desnecessária a solução de segregação de redes utilizada normalmente em sistemas SCADA. Diferente de soluções baseadas em TLS/SSL, ele permite a cifragem de comunicações sem conexão e multicast.

A quarta contribuição é a descrição dos processos de criptanálise diferencial e linear. A comunidade científica espera que elas sejam aplicadas a qualquer novo algoritmo definido. O capítulo pode ser utilizado como material didático em português para explicar o processo a estudantes de criptologia. Durante a pesquisa realizada para este trabalho, somente encontramos referências em inglês.

8.2 TRABALHOS FUTUROS

Esse trabalho pode ser a base para vários trabalhos futuros como:

- melhoria do algoritmo contra ataques de criptanálise linear;
- otimização das implementações de software do algoritmo;
- implementação em hardware do algoritmo proposto;
- participação do concurso do NIST para algoritmos leves;
- comparar o desempenho da cifras com o portifólio final de cifras autenticadas do

concurso CAESAR (previsto para dezembro/2017);

- implementação do sistema de proteção da comunicação para dispositivos no ambiente IoT.

9 REFERÊNCIAS BIBLIOGRÁFICAS

ALBRECHT, M. R.; DRIESSEN, B. Block Ciphers - Focus on the Linear Layer (feat. PRIDE). **Advances in Cryptology - CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I**, Berlin, Heidelberg, 2014. 57-76.

BARKER, E.; KELSEY, J. **NIST Special Publication 800-90A Revision 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators**. NIST. Gaithersburg, p. 109. 2015. (doi:10.6028/NIST.SP.800-90Ar1).

BARKER, E.; ROGINSKY, A. **NIST Special Publication 800-131A - Transitions:** Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. [S.l.]: US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2011.

BARKER, W. C.; BARKER, E. **NIST Special Publication 800-67 Revision 1: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher**. NIST. Gaithersburg, p. 34. 2012. (doi:10.6028/NIST.SP.800-67r1).

BEAULIEU, R. et al. SIMON and SPECK: Block Ciphers for the Internet of Things, 12 jun. 2015. Disponivel em: <http://eprint.iacr.org/2015/585>. Acesso em: 1 out. 2016.

BEIERLE, C. et al. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. **Advances in Cryptology -- CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II**, Berlin, Heidelberg, 2016. 123-153.

BELLARE, M.; CANETTI, R.; KRAWCZYK, H. Keying hash functions for message authentication. **Advances in Cryptology - CRYPTO 96**, 1996.

BIHAM, E.; DUNKELMAN, O.; KELLER, N. **Linear cryptanalysis of reduced round Serpent**. Fast Software Encryption 2001. Yokohama: Springer-Verlag Berlin Heidelberg. 2002. p. 16-27.

BIHAM, E.; SHAMIR, A. Differential cryptanalysis of DES-like cryptosystems. **Journal of CRYPTOLOGY**, 4, n. 1, 1991. 3-72.

BIHAM, E.; SHAMIR, A. **Differential Cryptanalysis of the Data Encryption Standard**. London, UK, UK: Springer-Verlag, 1993. ISBN 0-387-97930-1.

BOGDANOV, A. et al. **PRESENT:** An Ultra-Lightweight Block Cipher. Cryptographic Hardware and Embedded Systems - CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg. 2007. p. 450-466.

BORGHOFF, J. et al. PRINCE-a low-latency block cipher for pervasive computing applications. **Advances in Cryptology-ASIACRYPT 2012**, 2012. 208-225.

BROWN, R. G.; EDDELBUETTEL, D.; BAUER, D. Dieharder: A Random Number Test Suite, 2016. Disponivel em: <http://phy.duke.edu/~rgb/General/dieharder.php>. Acesso em: 13 maio 2016.

CHEN, D. et al. Non-Intrusive Occupancy Monitoring Using Smart Meters. **Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings**, 2013. 9:1-9:8.

CRYPTOGRAPHIC Competitions. **CAESAR:** Competition for Authenticated Encryption: Security, Applicability, and Robustness, 2016. Disponivel em: <https://competitions.cr.yp.to/caesar.html>. Acesso em: 22 mar. 2017.

CRYPTOLUX RESEARCH GROUP - UNIVERSITY OF LUXEMBOURG. Lightweight Block Ciphers, 2016. Disponivel em: <https://www.cryptolux.org/index.php/Lightweight_Block_Ciphers>. Acesso em: 1 out. 2016.

DAEMEN, J.; RIJMEN, V. Specification for the advanced encryption standard (AES). **Federal Information Processing Standards Publication**, 2001.

DAWSON, R. et al. SKMA: a key management architecture for SCADA systems. **Proceedings of the 2006 Australasian workshops on Grid computing and e-research**, 2006. 183-192.

DINU, D. et al. FELICS – Fair Evaluation of Lightweight Cryptographic Systems, jul. 2015. Disponivel em: <http://csrc.nist.gov/groups/ST/lwc-workshop2015/papers/session7-dinu-paper.pdf>. Acesso em: 12 out. 2016.

DUNKELMAN, O.; KELLER, N.; SHAMIR, A. Minimalism in cryptography: The Even-Mansour scheme revisited. **Advances in Cryptology-EUROCRYPT 2012**, 2012. 336-354.

DWORKIN, M. **NIST Special Publication 800-38A: Recommendation for block cipher modes of operation. methods and techniques**. Gaithersburg. 2001.

DWORKIN, M. **NIST Special Publication 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality**. Gaithersburg. 2004.

DWORKIN, M. **NIST Special Publication 800-38B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication**. Gaithersburg. 2005.

DWORKIN, M. **NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC**. Gaithersburg. 2007.

EVEN, S.; MANSOUR, Y. A construction of a cipher from a single pseudorandom permutation. **Journal of Cryptology**, 10, 1997. 151-161.

FELICS Block Ciphers Brief Results, 2016. Disponivel em: <https://www.cryptolux.org/index.php/FELICS_Block_Ciphers_Brief_Results>. Acesso em: 12 out. 2016.

GOLDREICH, O. **A primer on pseudorandom generators**. Rehovot, Israel: Weizmann Institute of Science, 2010.

GUBBI, J. et al. Internet of Things (IoT): A vision, architectural elements, and future directions. **Future Generation Computer Systems**, 2013. 1645-1660.

GUO, J. et al. The LED block cipher. **Cryptographic Hardware and Embedded Systems-CHES 2011**, 2011. 326-341.

HEYS, H. M. **A tutorial on linear and differential cryptanalysis**. Centre for Applied Cryptographic Research. Waterloo. 2001.

IEEE ICC2017 Technical Symposia Program. **IEEE International Conference on Communications 2017**, 2017. Disponivel em: <http://icc2017.ieee-icc.org/program/symposia>. Acesso em: 25 abr. 2017.

IGURE, V. M.; LAUGHTER, S. A.; WILLIAMS, R. D. Security issues in SCADA networks. **Computers & Security**, 25, 2006. 498-506.

INTEL CORPORATION. Intel Digital Random Number - Software Implementation Guide, 2014. Disponivel em: <https://software.intel.com/sites/default/files/managed/4d/91/DRNG_Software_Implementation_Guide_2.0.pdf>. Acesso em: 16 jan. 2016.

ISO/IEC 10116. **Information technology - Security techniques - Modes of operation for an n-bit block cipher**. Geneva: ISO, 2006.

ISO/IEC 29192-2:2012. **Information technology - Security techniques - Lightweight cryptography - Part 2:** Block ciphers. Geneva: ISO, 2012.

JUTLA, C. S. Encryption modes with almost free message integrity. **International Conference on the Theory and Applications of Cryptographic Techniques**, 2001. 529-544.

KRAWCZYK, H. The order of encryption and authentication for protecting communications (or: How secure is SSL?). **Advances in Cryptology—CRYPTO 2001**, 2001. 310-331.

LIDL, R.; NIEDERREITER, H. **Introduction to finite fields and their applications**. First Edition. ed. Cambridge: Cambridge university press, 1994.

LUBY, M.; RACKOFF, C. How to construct pseudorandom permutations from pseudorandom functions. **SIAM Journal on Computing**, Philadelphia, v. 17, n. 2, p. 373-386, 1988. ISSN DOI:10.1137/0217022.

MATSUI, M. Linear cryptanalysis method for DES cipher. **Workshop on the Theory and Application of of Cryptographic Techniques**, 1993. 386-397.

MCKAY, K. A. et al. DRAFT NISTIR 8114 - Report on Lightweight Cryptography, Gaithersburg, MD, 11 Aug 2016. Disponivel em: <http://csrc.nist.gov/publications/drafts/nistir-8114/nistir_8114_draft.pdf>. Acesso em: 1 out. 2016.

MENEZES, A. J.; VAN OORSCHOT, P. C.; VANSTONE, S. A. **Handbook of applied cryptography**. [S.l.]: CRC press, 1996.

NAOR, M.; REINGOLD, O. On the Construction of Pseudorandom Permutations: Luby-Rackoff Revisited. **Journal of Cryptology**, 12, n. 1, 1999. 29-66.

NIST. Random Number Generation - Download Documentation and Software, 2014. Disponivel em:

<http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html>. Acesso em: 06 fev. 2016.

NIST. CSRC HOME > GROUPS > ST > CRYPTOGRAPHIC TOOLKIT > BLOCK CIPHER MODES, 2016. Disponivel em: <http://csrc.nist.gov/groups/ST/toolkit/BCM/current_modes.html>. Acesso em: 08 ago. 2016.

PIÈTRE-CAMBACÉDÈS, L.; SITBON, P. Cryptographic key management for SCADA systems-issues and perspectives. **International Conference on Information Security and Assurance, 2008. ISA 2008.** , 2008. 156-161.

ROGAWAY, P. Evaluation of some blockcipher modes of operation. **Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan**, 2011.

RUKHIN, A. et al. **NIST Special Publication 800-22:A statistical test suite for random and pseudorandom number generators for cryptographic applications**. [S.l.]. 2010.

SCHNEIER, B. **Applied Cryptography:** Protocols, Algorithms, and Source Code in C. New York, NY, USA: John Wiley & Sons, Inc., 1996.

SOTO, J. Randomness testing of the AES candidate algorithms, 1999. Disponivel em: <http://csrc.nist.gov/archive/aes/round1/r1-rand.pdf>. Acesso em: 2 fev. 2016.

10 APÊNDICES

10.1 APÊNDICE I: RESULTADOS DA FERRAMENTA DIEHARDER

10.1.1 CIFRA AES

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  4.26e+05  |2783966867|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.14519236|  PASSED
      diehard_operm5|   0|   1000000|     100|0.75155091|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.74791322|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.80133963|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.97533256|  PASSED
        diehard_opso|   0|   2097152|     100|0.85212490|  PASSED
        diehard_oqso|   0|   2097152|     100|0.28622170|  PASSED
         diehard_dna|   0|   2097152|     100|0.19888468|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.93668549|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.71046774|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.86288622|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.10648657|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.36504934|  PASSED
     diehard_squeeze|   0|    100000|     100|0.37947328|  PASSED
        diehard_sums|   0|       100|     100|0.26298169|  PASSED
        diehard_runs|   0|    100000|     100|0.96698411|  PASSED
        diehard_runs|   0|    100000|     100|0.76885777|  PASSED
       diehard_craps|   0|    200000|     100|0.67993016|  PASSED
       diehard_craps|   0|    200000|     100|0.53708260|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.80673873|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.66049277|  PASSED
         sts_monobit|   1|    100000|     100|0.50087998|  PASSED
            sts_runs|   2|    100000|     100|0.65049771|  PASSED
          sts_serial|   1|    100000|     100|0.62576530|  PASSED
          sts_serial|   2|    100000|     100|0.54455099|  PASSED
          sts_serial|   3|    100000|     100|0.22967450|  PASSED
          sts_serial|   3|    100000|     100|0.23069701|  PASSED
          sts_serial|   4|    100000|     100|0.14644948|  PASSED
          sts_serial|   4|    100000|     100|0.55627389|  PASSED
          sts_serial|   5|    100000|     100|0.47312909|  PASSED
          sts_serial|   5|    100000|     100|0.57885092|  PASSED
          sts_serial|   6|    100000|     100|0.08632527|  PASSED
          sts_serial|   6|    100000|     100|0.86914234|  PASSED
          sts_serial|   7|    100000|     100|0.45671900|  PASSED
          sts_serial|   7|    100000|     100|0.69350600|  PASSED
          sts_serial|   8|    100000|     100|0.06557136|  PASSED
          sts_serial|   8|    100000|     100|0.03977494|  PASSED
          sts_serial|   9|    100000|     100|0.30997012|  PASSED
          sts_serial|   9|    100000|     100|0.94257230|  PASSED
          sts_serial|  10|    100000|     100|0.55257338|  PASSED
          sts_serial|  10|    100000|     100|0.95100627|  PASSED
          sts_serial|  11|    100000|     100|0.17380740|  PASSED
          sts_serial|  11|    100000|     100|0.54620461|  PASSED
          sts_serial|  12|    100000|     100|0.22446206|  PASSED
          sts_serial|  12|    100000|     100|0.95556613|  PASSED
          sts_serial|  13|    100000|     100|0.21699969|  PASSED
          sts_serial|  13|    100000|     100|0.74602764|  PASSED
          sts_serial|  14|    100000|     100|0.23269320|  PASSED
          sts_serial|  14|    100000|     100|0.22632480|  PASSED
          sts_serial|  15|    100000|     100|0.16875123|  PASSED
          sts_serial|  15|    100000|     100|0.40469065|  PASSED
          sts_serial|  16|    100000|     100|0.98625751|  PASSED
          sts_serial|  16|    100000|     100|0.06623148|  PASSED
         rgb_bitdist|   1|    100000|     100|0.09645146|  PASSED
         rgb_bitdist|   2|    100000|     100|0.78869300|  PASSED
         rgb_bitdist|   3|    100000|     100|0.14956573|  PASSED
         rgb_bitdist|   4|    100000|     100|0.74574567|  PASSED
         rgb_bitdist|   5|    100000|     100|0.86192138|  PASSED
         rgb_bitdist|   6|    100000|     100|0.25295539|  PASSED
         rgb_bitdist|   7|    100000|     100|0.62189710|  PASSED
         rgb_bitdist|   8|    100000|     100|0.99432466|  PASSED
         rgb_bitdist|   9|    100000|     100|0.69587898|  PASSED
         rgb_bitdist|  10|    100000|     100|0.68697493|  PASSED
         rgb_bitdist|  11|    100000|     100|0.77180699|  PASSED
         rgb_bitdist|  12|    100000|     100|0.97664665|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.32329813|  PASSED
```

```
rgb_minimum_distance|   3|     10000|    1000|0.97215852|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.29396846|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.03828786|  PASSED
    rgb_permutations|   2|    100000|     100|0.58796560|  PASSED
    rgb_permutations|   3|    100000|     100|0.39859220|  PASSED
    rgb_permutations|   4|    100000|     100|0.77496143|  PASSED
    rgb_permutations|   5|    100000|     100|0.02661811|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.84627122|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.69644844|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.98169613|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.56912075|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.75671626|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.58073159|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.67160951|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.38480899|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.34717610|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.81388775|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.09096244|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.87322730|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.92520570|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.65751482|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.08336855|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.93122687|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.68266798|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.22511983|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.10747554|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.67824582|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.52455600|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.76616827|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.98337066|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.80158824|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.54847157|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.52310244|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.90709377|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.71077131|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.99200264|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.57812715|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.32979999|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.80387481|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.75643989|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.61879238|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.06004248|  PASSED
             dab_dct| 256|     50000|       1|0.53896064|  PASSED
Preparing to run test 207.  ntuple = 0
        dab_filltree|  32|  15000000|       1|0.34105124|  PASSED
        dab_filltree|  32|  15000000|       1|0.20911966|  PASSED
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.76199507|  PASSED
       dab_filltree2|   1|   5000000|       1|0.32097858|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.78426382|  PASSED
```

10.1.2 CIFRA FLEXAE_64_128

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  1.52e+05  |1513952858|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.14596724|  PASSED
      diehard_operm5|   0|   1000000|     100|0.61257044|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.36031960|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.30377886|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.96089312|  PASSED
        diehard_opso|   0|   2097152|     100|0.57094564|  PASSED
        diehard_oqso|   0|   2097152|     100|0.79347543|  PASSED
         diehard_dna|   0|   2097152|     100|0.38527759|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.96056004|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.22578115|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.82824986|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.03529780|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.98266896|  PASSED
     diehard_squeeze|   0|    100000|     100|0.16297483|  PASSED
        diehard_sums|   0|       100|     100|0.04630711|  PASSED
        diehard_runs|   0|    100000|     100|0.70551570|  PASSED
        diehard_runs|   0|    100000|     100|0.45732195|  PASSED
       diehard_craps|   0|    200000|     100|0.51126592|  PASSED
```

diehard_craps	0	200000	100	0.90308635	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.28810589	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.72985058	PASSED
sts_monobit	1	100000	100	0.73433420	PASSED
sts_runs	2	100000	100	0.33523708	PASSED
sts_serial	1	100000	100	0.38425652	PASSED
sts_serial	2	100000	100	0.50983150	PASSED
sts_serial	3	100000	100	0.24793749	PASSED
sts_serial	3	100000	100	0.22813683	PASSED
sts_serial	4	100000	100	0.31416018	PASSED
sts_serial	4	100000	100	0.92705157	PASSED
sts_serial	5	100000	100	0.41871675	PASSED
sts_serial	5	100000	100	0.52723406	PASSED
sts_serial	6	100000	100	0.65722154	PASSED
sts_serial	6	100000	100	0.96348151	PASSED
sts_serial	7	100000	100	0.61282542	PASSED
sts_serial	7	100000	100	0.29220005	PASSED
sts_serial	8	100000	100	0.34646194	PASSED
sts_serial	8	100000	100	0.89706211	PASSED
sts_serial	9	100000	100	0.38219375	PASSED
sts_serial	9	100000	100	0.69469849	PASSED
sts_serial	10	100000	100	0.98788400	PASSED
sts_serial	10	100000	100	0.34174114	PASSED
sts_serial	11	100000	100	0.95012962	PASSED
sts_serial	11	100000	100	0.90236567	PASSED
sts_serial	12	100000	100	0.75629342	PASSED
sts_serial	12	100000	100	0.61416598	PASSED
sts_serial	13	100000	100	0.98795680	PASSED
sts_serial	13	100000	100	0.93545254	PASSED
sts_serial	14	100000	100	0.96772495	PASSED
sts_serial	14	100000	100	0.42093961	PASSED
sts_serial	15	100000	100	0.90048461	PASSED
sts_serial	15	100000	100	0.52991307	PASSED
sts_serial	16	100000	100	0.54140292	PASSED
sts_serial	16	100000	100	0.91098849	PASSED
rgb_bitdist	1	100000	100	0.47783519	PASSED
rgb_bitdist	2	100000	100	0.03625330	PASSED
rgb_bitdist	3	100000	100	0.34779055	PASSED
rgb_bitdist	4	100000	100	0.82637443	PASSED
rgb_bitdist	5	100000	100	0.91842215	PASSED
rgb_bitdist	6	100000	100	0.89423505	PASSED
rgb_bitdist	7	100000	100	0.93589344	PASSED
rgb_bitdist	8	100000	100	0.72242700	PASSED
rgb_bitdist	9	100000	100	0.08455114	PASSED
rgb_bitdist	10	100000	100	0.65425409	PASSED
rgb_bitdist	11	100000	100	0.06963085	PASSED
rgb_bitdist	12	100000	100	0.50589574	PASSED
rgb_minimum_distance	2	10000	1000	0.31003777	PASSED
rgb_minimum_distance	3	10000	1000	0.39803412	PASSED
rgb_minimum_distance	4	10000	1000	0.89906191	PASSED
rgb_minimum_distance	5	10000	1000	0.46211586	PASSED
rgb_permutations	2	100000	100	0.99054960	PASSED
rgb_permutations	3	100000	100	0.08149175	PASSED
rgb_permutations	4	100000	100	0.94064273	PASSED
rgb_permutations	5	100000	100	0.57949633	PASSED
rgb_lagged_sum	0	1000000	100	0.28029943	PASSED
rgb_lagged_sum	1	1000000	100	0.07613508	PASSED
rgb_lagged_sum	2	1000000	100	0.10279959	PASSED
rgb_lagged_sum	3	1000000	100	0.02866857	PASSED
rgb_lagged_sum	4	1000000	100	0.86832288	PASSED
rgb_lagged_sum	5	1000000	100	0.70052719	PASSED
rgb_lagged_sum	6	1000000	100	0.24043638	PASSED
rgb_lagged_sum	7	1000000	100	0.91121565	PASSED
rgb_lagged_sum	8	1000000	100	0.94009047	PASSED
rgb_lagged_sum	9	1000000	100	0.76229263	PASSED
rgb_lagged_sum	10	1000000	100	0.21339133	PASSED
rgb_lagged_sum	11	1000000	100	0.18533214	PASSED
rgb_lagged_sum	12	1000000	100	0.66201114	PASSED
rgb_lagged_sum	13	1000000	100	0.82805542	PASSED
rgb_lagged_sum	14	1000000	100	0.55295030	PASSED
rgb_lagged_sum	15	1000000	100	0.19454189	PASSED
rgb_lagged_sum	16	1000000	100	0.91084025	PASSED
rgb_lagged_sum	17	1000000	100	0.94211069	PASSED
rgb_lagged_sum	18	1000000	100	0.36054703	PASSED
rgb_lagged_sum	19	1000000	100	0.96858477	PASSED
rgb_lagged_sum	20	1000000	100	0.90843814	PASSED
rgb_lagged_sum	21	1000000	100	0.65498923	PASSED
rgb_lagged_sum	22	1000000	100	0.41555870	PASSED
rgb_lagged_sum	**23**	**1000000**	**100**	**0.99782910**	**WEAK**
rgb_lagged_sum	24	1000000	100	0.83899437	PASSED
rgb_lagged_sum	25	1000000	100	0.88449361	PASSED
rgb_lagged_sum	26	1000000	100	0.09895277	PASSED
rgb_lagged_sum	27	1000000	100	0.98969824	PASSED
rgb_lagged_sum	28	1000000	100	0.45705719	PASSED
rgb_lagged_sum	29	1000000	100	0.70526740	PASSED

```
      rgb_lagged_sum|  30|   1000000|     100|0.15223417|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.93736519|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.71676103|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.70295881|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.61900490|  PASSED
             dab_dct| 256|     50000|       1|0.42070206|  PASSED
Preparing to run test 207.  ntuple = 0
        dab_filltree|  32|  15000000|       1|0.57431202|  PASSED
        dab_filltree|  32|  15000000|       1|0.96906839|  PASSED
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.92651076|  PASSED
       dab_filltree2|   1|   5000000|       1|0.05640034|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.29790695|  PASSED
```

rgb_lagged_sum - 23 - rerun:

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  7.22e+04  |1267141021|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
      rgb_lagged_sum|  23|   1000000|     100|0.52439340|  PASSED
```

10.1.3 CIFRA FLEXAE_128_256

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  1.51e+05  |1179247939|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.90677051|  PASSED
      diehard_operm5|   0|   1000000|     100|0.17073966|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.90973348|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.20429733|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.48649196|  PASSED
        diehard_opso|   0|   2097152|     100|0.25916906|  PASSED
        diehard_oqso|   0|   2097152|     100|0.46168811|  PASSED
         diehard_dna|   0|   2097152|     100|0.05569302|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.16167106|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.89719217|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.38931075|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.35251138|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.52743078|  PASSED
     diehard_squeeze|   0|    100000|     100|0.40474842|  PASSED
        diehard_sums|   0|       100|     100|0.09626758|  PASSED
        diehard_runs|   0|    100000|     100|0.44655248|  PASSED
        diehard_runs|   0|    100000|     100|0.75962497|  PASSED
       diehard_craps|   0|    200000|     100|0.70614614|  PASSED
       diehard_craps|   0|    200000|     100|0.07738727|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.88495934|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.88539789|  PASSED
         sts_monobit|   1|    100000|     100|0.59316651|  PASSED
            sts_runs|   2|    100000|     100|0.21658845|  PASSED
          sts_serial|   1|    100000|     100|0.53392890|  PASSED
          sts_serial|   2|    100000|     100|0.86162909|  PASSED
          sts_serial|   3|    100000|     100|0.96410530|  PASSED
          sts_serial|   3|    100000|     100|0.55344948|  PASSED
          sts_serial|   4|    100000|     100|0.97220553|  PASSED
          sts_serial|   4|    100000|     100|0.64282316|  PASSED
          sts_serial|   5|    100000|     100|0.44869012|  PASSED
          sts_serial|   5|    100000|     100|0.21702205|  PASSED
          sts_serial|   6|    100000|     100|0.46320654|  PASSED
          sts_serial|   6|    100000|     100|0.77670738|  PASSED
          sts_serial|   7|    100000|     100|0.04699450|  PASSED
          sts_serial|   7|    100000|     100|0.25369171|  PASSED
          sts_serial|   8|    100000|     100|0.32069407|  PASSED
          sts_serial|   8|    100000|     100|0.95162200|  PASSED
          sts_serial|   9|    100000|     100|0.04790029|  PASSED
          sts_serial|   9|    100000|     100|0.20683787|  PASSED
          sts_serial|  10|    100000|     100|0.34695646|  PASSED
          sts_serial|  10|    100000|     100|0.86305325|  PASSED
          sts_serial|  11|    100000|     100|0.15350556|  PASSED
          sts_serial|  11|    100000|     100|0.08306128|  PASSED
```

```
          sts_serial|  12|    100000|     100|0.04736172|  PASSED
          sts_serial|  12|    100000|     100|0.10670224|  PASSED
          sts_serial|  13|    100000|     100|0.14481541|  PASSED
          sts_serial|  13|    100000|     100|0.57989969|  PASSED
          sts_serial|  14|    100000|     100|0.71869182|  PASSED
          sts_serial|  14|    100000|     100|0.32354790|  PASSED
          sts_serial|  15|    100000|     100|0.25714466|  PASSED
          sts_serial|  15|    100000|     100|0.88313758|  PASSED
          sts_serial|  16|    100000|     100|0.72628851|  PASSED
          sts_serial|  16|    100000|     100|0.32247101|  PASSED
         rgb_bitdist|   1|    100000|     100|0.82563602|  PASSED
         rgb_bitdist|   2|    100000|     100|0.55512820|  PASSED
         rgb_bitdist|   3|    100000|     100|0.65086272|  PASSED
         rgb_bitdist|   4|    100000|     100|0.67335144|  PASSED
         rgb_bitdist|   5|    100000|     100|0.78379301|  PASSED
         rgb_bitdist|   6|    100000|     100|0.92335933|  PASSED
         rgb_bitdist|   7|    100000|     100|0.56729329|  PASSED
         rgb_bitdist|   8|    100000|     100|0.94565374|  PASSED
         rgb_bitdist|   9|    100000|     100|0.86057390|  PASSED
         rgb_bitdist|  10|    100000|     100|0.29230116|  PASSED
         rgb_bitdist|  11|    100000|     100|0.32145912|  PASSED
         rgb_bitdist|  12|    100000|     100|0.98843203|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.92402260|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.83656311|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.42153040|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.01554910|  PASSED
    rgb_permutations|   2|    100000|     100|0.90422140|  PASSED
    rgb_permutations|   3|    100000|     100|0.34150528|  PASSED
    rgb_permutations|   4|    100000|     100|0.17608199|  PASSED
    rgb_permutations|   5|    100000|     100|0.81265263|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.99434139|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.99856447|   WEAK
      rgb_lagged_sum|   2|   1000000|     100|0.15765025|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.23449223|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.99095066|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.85991930|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.26743502|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.40442148|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.86055821|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.48624262|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.32226229|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.76911117|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.07770938|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.79597941|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.60356464|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.90438545|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.35026185|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.98689204|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.99890739|   WEAK
      rgb_lagged_sum|  19|   1000000|     100|0.83634740|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.83549437|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.84233906|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.84807003|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.29816751|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.92429918|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.45482185|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.34477998|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.49605927|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.10874616|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.65787322|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.89547526|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.35852054|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.32087504|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.93407626|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.95630957|  PASSED
             dab_dct| 256|     50000|       1|0.06315936|  PASSED
Preparing to run test 207.  ntuple = 0
        dab_filltree|  32|  15000000|       1|0.86745507|  PASSED
        dab_filltree|  32|  15000000|       1|0.96020096|  PASSED
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.14722377|  PASSED
       dab_filltree2|   1|   5000000|       1|0.23670584|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.10017173|  PASSED
```

rgb_lagged_sum - 1 - rerun:

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  8.44e+04  |3972439611|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
```

```
      rgb_lagged_sum|   1|   1000000|     100|0.65094885|  PASSED

### rgb_lagged_sum - 18 - rerun:
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  7.82e+04  | 811123811|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
      rgb_lagged_sum|  18|   1000000|     100|0.33377476|  PASSED
```

10.1.4 CIFRA FLEXAE_256_512

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  1.41e+05  | 155218752|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.52885163|  PASSED
      diehard_operm5|   0|   1000000|     100|0.67674908|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.72012414|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.49825320|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.86204685|  PASSED
        diehard_opso|   0|   2097152|     100|0.72674184|  PASSED
        diehard_oqso|   0|   2097152|     100|0.26617282|  PASSED
         diehard_dna|   0|   2097152|     100|0.64055183|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.98736914|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.17260900|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.06636655|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.35581164|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.92188209|  PASSED
     diehard_squeeze|   0|    100000|     100|0.61144361|  PASSED
        diehard_sums|   0|       100|     100|0.19877099|  PASSED
        diehard_runs|   0|    100000|     100|0.96782268|  PASSED
        diehard_runs|   0|    100000|     100|0.53798736|  PASSED
       diehard_craps|   0|    200000|     100|0.28006062|  PASSED
       diehard_craps|   0|    200000|     100|0.45403570|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.30724436|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.76899511|  PASSED
         sts_monobit|   1|    100000|     100|0.66009392|  PASSED
            sts_runs|   2|    100000|     100|0.01298030|  PASSED
          sts_serial|   1|    100000|     100|0.23630334|  PASSED
          sts_serial|   2|    100000|     100|0.49123729|  PASSED
          sts_serial|   3|    100000|     100|0.72803844|  PASSED
          sts_serial|   3|    100000|     100|0.53333392|  PASSED
          sts_serial|   4|    100000|     100|0.03515221|  PASSED
          sts_serial|   4|    100000|     100|0.31240933|  PASSED
          sts_serial|   5|    100000|     100|0.08726640|  PASSED
          sts_serial|   5|    100000|     100|0.85064523|  PASSED
          sts_serial|   6|    100000|     100|0.73952284|  PASSED
          sts_serial|   6|    100000|     100|0.23320831|  PASSED
          sts_serial|   7|    100000|     100|0.84144615|  PASSED
          sts_serial|   7|    100000|     100|0.05848022|  PASSED
          sts_serial|   8|    100000|     100|0.55505912|  PASSED
          sts_serial|   8|    100000|     100|0.81965439|  PASSED
          sts_serial|   9|    100000|     100|0.84944909|  PASSED
          sts_serial|   9|    100000|     100|0.51767807|  PASSED
          sts_serial|  10|    100000|     100|0.25190290|  PASSED
          sts_serial|  10|    100000|     100|0.24650955|  PASSED
          sts_serial|  11|    100000|     100|0.05533647|  PASSED
          sts_serial|  11|    100000|     100|0.42354408|  PASSED
          sts_serial|  12|    100000|     100|0.71362760|  PASSED
          sts_serial|  12|    100000|     100|0.86163224|  PASSED
          sts_serial|  13|    100000|     100|0.51413915|  PASSED
          sts_serial|  13|    100000|     100|0.89970880|  PASSED
          sts_serial|  14|    100000|     100|0.73147431|  PASSED
          sts_serial|  14|    100000|     100|0.04784240|  PASSED
          sts_serial|  15|    100000|     100|0.99921989|   WEAK
          sts_serial|  15|    100000|     100|0.17245883|  PASSED
          sts_serial|  16|    100000|     100|0.33934113|  PASSED
          sts_serial|  16|    100000|     100|0.80960625|  PASSED
         rgb_bitdist|   1|    100000|     100|0.36986170|  PASSED
         rgb_bitdist|   2|    100000|     100|0.99878840|   WEAK
         rgb_bitdist|   3|    100000|     100|0.81542452|  PASSED
```

```
         rgb_bitdist|   4|    100000|     100|0.77384286|  PASSED
         rgb_bitdist|   5|    100000|     100|0.38708787|  PASSED
         rgb_bitdist|   6|    100000|     100|0.93848607|  PASSED
         rgb_bitdist|   7|    100000|     100|0.88407389|  PASSED
         rgb_bitdist|   8|    100000|     100|0.20891414|  PASSED
         rgb_bitdist|   9|    100000|     100|0.21283708|  PASSED
         rgb_bitdist|  10|    100000|     100|0.93700669|  PASSED
         rgb_bitdist|  11|    100000|     100|0.75303503|  PASSED
         rgb_bitdist|  12|    100000|     100|0.10150262|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.19424840|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.97611708|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.06308878|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.39983445|  PASSED
    rgb_permutations|   2|    100000|     100|0.41486057|  PASSED
    rgb_permutations|   3|    100000|     100|0.24381108|  PASSED
    rgb_permutations|   4|    100000|     100|0.74630887|  PASSED
    rgb_permutations|   5|    100000|     100|0.11448023|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.67714033|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.70127552|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.60671840|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.22178148|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.09638610|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.95410965|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.34966111|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.78133447|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.90818384|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.88711343|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.32472016|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.65945916|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.78524523|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.35865828|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.11285704|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.85681728|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.53661007|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.68511812|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.99306049|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.65411874|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.24267080|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.76589717|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.90128227|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.06132701|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.52885646|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.50476908|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.29842360|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.24291823|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.74452070|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.44341834|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.96810920|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.97269003|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.57406086|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.46067459|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.84725124|  PASSED
             dab_dct| 256|     50000|       1|0.30329826|  PASSED
        dab_filltree|  32|  15000000|       1|0.33674577|  PASSED
        dab_filltree|  32|  15000000|       1|0.71987025|  PASSED
       dab_filltree2|   0|   5000000|       1|0.66443634|  PASSED
       dab_filltree2|   1|   5000000|       1|0.01547037|  PASSED
        dab_monobit2|  12|  65000000|       1|0.86875170|  PASSED
```

rgb_bitdist - 2 - rerun:

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  3.89e+04  |3900752562|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
         rgb_bitdist|   2|    100000|     100|0.92726151|  PASSED
```

sts-serial - rerun:

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  4.08e+04  | 951014233|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
          sts_serial|   1|    100000|     100|0.25087241|  PASSED
          sts_serial|   2|    100000|     100|0.97641121|  PASSED
          sts_serial|   3|    100000|     100|0.22542666|  PASSED
```

```
          sts_serial|   3|    100000|     100|0.59592318|  PASSED
          sts_serial|   4|    100000|     100|0.49377708|  PASSED
          sts_serial|   4|    100000|     100|0.82453710|  PASSED
          sts_serial|   5|    100000|     100|0.20152335|  PASSED
          sts_serial|   5|    100000|     100|0.55835794|  PASSED
          sts_serial|   6|    100000|     100|0.94470749|  PASSED
          sts_serial|   6|    100000|     100|0.04274300|  PASSED
          sts_serial|   7|    100000|     100|0.95348043|  PASSED
          sts_serial|   7|    100000|     100|0.09716613|  PASSED
          sts_serial|   8|    100000|     100|0.61296056|  PASSED
          sts_serial|   8|    100000|     100|0.71406711|  PASSED
          sts_serial|   9|    100000|     100|0.93997758|  PASSED
          sts_serial|   9|    100000|     100|0.78700734|  PASSED
          sts_serial|  10|    100000|     100|0.94567525|  PASSED
          sts_serial|  10|    100000|     100|0.98860472|  PASSED
          sts_serial|  11|    100000|     100|0.76833601|  PASSED
          sts_serial|  11|    100000|     100|0.43238881|  PASSED
          sts_serial|  12|    100000|     100|0.68006330|  PASSED
          sts_serial|  12|    100000|     100|0.37215765|  PASSED
          sts_serial|  13|    100000|     100|0.40954932|  PASSED
          sts_serial|  13|    100000|     100|0.70281905|  PASSED
          sts_serial|  14|    100000|     100|0.24290245|  PASSED
          sts_serial|  14|    100000|     100|0.91016199|  PASSED
          sts_serial|  15|    100000|     100|0.65117089|  PASSED
          sts_serial|  15|    100000|     100|0.45237434|  PASSED
          sts_serial|  16|    100000|     100|0.88998874|  PASSED
          sts_serial|  16|    100000|     100|0.52799056|  PASSED
```

10.1.5 CIFRA FLEXAE_512_1024

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  1.26e+05  |2356046250|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.05611581|  PASSED
      diehard_operm5|   0|   1000000|     100|0.94045036|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.73156153|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.95673882|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.75113873|  PASSED
        diehard_opso|   0|   2097152|     100|0.93534450|  PASSED
        diehard_oqso|   0|   2097152|     100|0.00105990|   WEAK
         diehard_dna|   0|   2097152|     100|0.59590051|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.94876395|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.65620723|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.15579947|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.92138400|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.66829792|  PASSED
     diehard_squeeze|   0|    100000|     100|0.77723320|  PASSED
        diehard_sums|   0|       100|     100|0.24817000|  PASSED
        diehard_runs|   0|    100000|     100|0.23497554|  PASSED
        diehard_runs|   0|    100000|     100|0.90316177|  PASSED
       diehard_craps|   0|    200000|     100|0.77972009|  PASSED
       diehard_craps|   0|    200000|     100|0.80681333|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.30946007|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.35995326|  PASSED
         sts_monobit|   1|    100000|     100|0.67583307|  PASSED
            sts_runs|   2|    100000|     100|0.46219303|  PASSED
          sts_serial|   1|    100000|     100|0.79071764|  PASSED
          sts_serial|   2|    100000|     100|0.59914047|  PASSED
          sts_serial|   3|    100000|     100|0.10662572|  PASSED
          sts_serial|   3|    100000|     100|0.72980568|  PASSED
          sts_serial|   4|    100000|     100|0.03536364|  PASSED
          sts_serial|   4|    100000|     100|0.22496672|  PASSED
          sts_serial|   5|    100000|     100|0.76883299|  PASSED
          sts_serial|   5|    100000|     100|0.52933019|  PASSED
          sts_serial|   6|    100000|     100|0.94547469|  PASSED
          sts_serial|   6|    100000|     100|0.17109904|  PASSED
          sts_serial|   7|    100000|     100|0.86113594|  PASSED
          sts_serial|   7|    100000|     100|0.37300933|  PASSED
          sts_serial|   8|    100000|     100|0.92056193|  PASSED
          sts_serial|   8|    100000|     100|0.11077371|  PASSED
          sts_serial|   9|    100000|     100|0.96201498|  PASSED
          sts_serial|   9|    100000|     100|0.45018303|  PASSED
          sts_serial|  10|    100000|     100|0.76843935|  PASSED
          sts_serial|  10|    100000|     100|0.70498134|  PASSED
```

```
          sts_serial|  11|    100000|     100|0.14068043|  PASSED
          sts_serial|  11|    100000|     100|0.30942644|  PASSED
          sts_serial|  12|    100000|     100|0.19134407|  PASSED
          sts_serial|  12|    100000|     100|0.39441272|  PASSED
          sts_serial|  13|    100000|     100|0.02570766|  PASSED
          sts_serial|  13|    100000|     100|0.10990414|  PASSED
          sts_serial|  14|    100000|     100|0.80614339|  PASSED
          sts_serial|  14|    100000|     100|0.66806795|  PASSED
          sts_serial|  15|    100000|     100|0.46104284|  PASSED
          sts_serial|  15|    100000|     100|0.97190412|  PASSED
          sts_serial|  16|    100000|     100|0.19189979|  PASSED
          sts_serial|  16|    100000|     100|0.28669244|  PASSED
         rgb_bitdist|   1|    100000|     100|0.72081814|  PASSED
         rgb_bitdist|   2|    100000|     100|0.66020369|  PASSED
         rgb_bitdist|   3|    100000|     100|0.11878929|  PASSED
         rgb_bitdist|   4|    100000|     100|0.96104510|  PASSED
         rgb_bitdist|   5|    100000|     100|0.47264412|  PASSED
         rgb_bitdist|   6|    100000|     100|0.20403465|  PASSED
         rgb_bitdist|   7|    100000|     100|0.96690220|  PASSED
         rgb_bitdist|   8|    100000|     100|0.63287057|  PASSED
         rgb_bitdist|   9|    100000|     100|0.79472904|  PASSED
         rgb_bitdist|  10|    100000|     100|0.68421807|  PASSED
         rgb_bitdist|  11|    100000|     100|0.29977752|  PASSED
         rgb_bitdist|  12|    100000|     100|0.66789735|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.38984678|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.29038440|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.89786322|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.77445736|  PASSED
    rgb_permutations|   2|    100000|     100|0.29974211|  PASSED
    rgb_permutations|   3|    100000|     100|0.70358639|  PASSED
    rgb_permutations|   4|    100000|     100|0.61992820|  PASSED
    rgb_permutations|   5|    100000|     100|0.91366542|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.49768981|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.96763575|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.58965104|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.97698257|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.79421140|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.73650904|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.46751511|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.82218446|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.36778262|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.00392904|   WEAK
      rgb_lagged_sum|  10|   1000000|     100|0.32232569|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.56225251|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.66025189|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.66574170|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.11274244|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.94152549|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.83970468|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.48210246|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.87554825|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.98242316|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.70446811|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.47729410|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.87818799|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.77941913|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.96590593|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.89564657|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.77177237|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.35072325|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.05527480|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.85747796|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.06543526|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.95650442|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.31251007|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.31884886|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.02714357|  PASSED
             dab_dct| 256|     50000|       1|0.89345213|  PASSED
        dab_filltree|  32|  15000000|       1|0.12817532|  PASSED
        dab_filltree|  32|  15000000|       1|0.76799723|  PASSED
       dab_filltree2|   0|   5000000|       1|0.57804496|  PASSED
       dab_filltree2|   1|   5000000|       1|0.27153546|  PASSED
        dab_monobit2|  12|  65000000|       1|0.95910929|  PASSED
```

diehard_oqso - 0 - rerun:

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  6.46e+04  |2965852834|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
        diehard_opso|   0|   2097152|     100|0.73028998|  PASSED
```

rgb_lagged_sum - 9 - rerun:

```
#==============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown           #
#==============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  1.16e+05  |2321088326|
#==============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#==============================================================================#
      rgb_lagged_sum|   9|   1000000|     100|0.90671792|  PASSED
```

10.2 APÊNDICE II: SBOX DO AES E CIFRA PROPOSTA

10.2.1 SBOX0 OU SBOX AES – DIRETA

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0x63	0x7C	0x77	0x7B	0xF2	0x6B	0x6F	0xC5	0x30	0x01	0x67	0x2B	0xFE	0xD7	0xAB	0x76
0x1	0xCA	0x82	0xC9	0x7D	0xFA	0x59	0x47	0xF0	0xAD	0xD4	0xA2	0xAF	0x9C	0xA4	0x72	0xC0
0x2	0xB7	0xFD	0x93	0x26	0x36	0x3F	0xF7	0xCC	0x34	0xA5	0xE5	0xF1	0x71	0xD8	0x31	0x15
0x3	0x04	0xC7	0x23	0xC3	0x18	0x96	0x05	0x9A	0x07	0x12	0x80	0xE2	0xEB	0x27	0xB2	0x75
0x4	0x09	0x83	0x2C	0x1A	0x1B	0x6E	0x5A	0xA0	0x52	0x3B	0xD6	0xB3	0x29	0xE3	0x2F	0x84
0x5	0x53	0xD1	0x00	0xED	0x20	0xFC	0xB1	0x5B	0x6A	0xCB	0xBE	0x39	0x4A	0x4C	0x58	0xCF
0x6	0xD0	0xEF	0xAA	0xFB	0x43	0x4D	0x33	0x85	0x45	0xF9	0x02	0x7F	0x50	0x3C	0x9F	0xA8
0x7	0x51	0xA3	0x40	0x8F	0x92	0x9D	0x38	0xF5	0xBC	0xB6	0xDA	0x21	0x10	0xFF	0xF3	0xD2
0x8	0xCD	0x0C	0x13	0xEC	0x5F	0x97	0x44	0x17	0xC4	0xA7	0x7E	0x3D	0x64	0x5D	0x19	0x73
0x9	0x60	0x81	0x4F	0xDC	0x22	0x2A	0x90	0x88	0x46	0xEE	0xB8	0x14	0xDE	0x5E	0x0B	0xDB
0xA	0xE0	0x32	0x3A	0x0A	0x49	0x06	0x24	0x5C	0xC2	0xD3	0xAC	0x62	0x91	0x95	0xE4	0x79
0xB	0xE7	0xC8	0x37	0x6D	0x8D	0xD5	0x4E	0xA9	0x6C	0x56	0xF4	0xEA	0x65	0x7A	0xAE	0x08
0xC	0xBA	0x78	0x25	0x2E	0x1C	0xA6	0xB4	0xC6	0xE8	0xDD	0x74	0x1F	0x4B	0xBD	0x8B	0x8A
0xD	0x70	0x3E	0xB5	0x66	0x48	0x03	0xF6	0x0E	0x61	0x35	0x57	0xB9	0x86	0xC1	0x1D	0x9E
0xE	0xE1	0xF8	0x98	0x11	0x69	0xD9	0x8E	0x94	0x9B	0x1E	0x87	0xE9	0xCE	0x55	0x28	0xDF
0xF	0x8C	0xA1	0x89	0x0D	0xBF	0xE6	0x42	0x68	0x41	0x99	0x2D	0x0F	0xB0	0x54	0xBB	0x16

10.2.2 SBOX0 OU SBOX AES – INVERSA

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0x52	0x09	0x6A	0xD5	0x30	0x36	0xA5	0x38	0xBF	0x40	0xA3	0x9E	0x81	0xF3	0xD7	0xFB
0x1	0x7C	0xE3	0x39	0x82	0x9B	0x2F	0xFF	0x87	0x34	0x8E	0x43	0x44	0xC4	0xDE	0xE9	0xCB
0x2	0x54	0x7B	0x94	0x32	0xA6	0xC2	0x23	0x3D	0xEE	0x4C	0x95	0x0B	0x42	0xFA	0xC3	0x4E
0x3	0x08	0x2E	0xA1	0x66	0x28	0xD9	0x24	0xB2	0x76	0x5B	0xA2	0x49	0x6D	0x8B	0xD1	0x25
0x4	0x72	0xF8	0xF6	0x64	0x86	0x68	0x98	0x16	0xD4	0xA4	0x5C	0xCC	0x5D	0x65	0xB6	0x92
0x5	0x6C	0x70	0x48	0x50	0xFD	0xED	0xB9	0xDA	0x5E	0x15	0x46	0x57	0xA7	0x8D	0x9D	0x84
0x6	0x90	0xD8	0xAB	0x00	0x8C	0xBC	0xD3	0x0A	0xF7	0xE4	0x58	0x05	0xB8	0xB3	0x45	0x06
0x7	0xD0	0x2C	0x1E	0x8F	0xCA	0x3F	0x0F	0x02	0xC1	0xAF	0xBD	0x03	0x01	0x13	0x8A	0x6B
0x8	0x3A	0x91	0x11	0x41	0x4F	0x67	0xDC	0xEA	0x97	0xF2	0xCF	0xCE	0xF0	0xB4	0xE6	0x73
0x9	0x96	0xAC	0x74	0x22	0xE7	0xAD	0x35	0x85	0xE2	0xF9	0x37	0xE8	0x1C	0x75	0xDF	0x6E
0xA	0x47	0xF1	0x1A	0x71	0x1D	0x29	0xC5	0x89	0x6F	0xB7	0x62	0x0E	0xAA	0x18	0xBE	0x1B
0xB	0xFC	0x56	0x3E	0x4B	0xC6	0xD2	0x79	0x20	0x9A	0xDB	0xC0	0xFE	0x78	0xCD	0x5A	0xF4
0xC	0x1F	0xDD	0xA8	0x33	0x88	0x07	0xC7	0x31	0xB1	0x12	0x10	0x59	0x27	0x80	0xEC	0x5F
0xD	0x60	0x51	0x7F	0xA9	0x19	0xB5	0x4A	0x0D	0x2D	0xE5	0x7A	0x9F	0x93	0xC9	0x9C	0xEF
0xE	0xA0	0xE0	0x3B	0x4D	0xAE	0x2A	0xF5	0xB0	0xC8	0xEB	0xBB	0x3C	0x83	0x53	0x99	0x61
0xF	0x17	0x2B	0x04	0x7E	0xBA	0x77	0xD6	0x26	0xE1	0x69	0x14	0x63	0x55	0x21	0x0C	0x7D

10.2.3 SBOX1 – DIRETA

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0x95	0x8A	0xA0	0xB3	0x0F	0xA4	0x86	0xCE	0x62	0x70	0xB7	0x13	0x1C	0x3F	0x38	0xA9
0x1	0xD4	0x7E	0xDD	0x5B	0x3E	0x3D	0xD6	0xF3	0xEB	0x96	0xC0	0x1A	0xF9	0x3B	0x8B	0xB0
0x2	0x35	0xA5	0x60	0xBC	0xB1	0xE1	0xF2	0xDC	0x40	0x2C	0xC1	0x94	0x34	0xC3	0xA6	0x0B
0x3	0xAA	0xFD	0x14	0xCC	0x85	0xFC	0x52	0x19	0xA3	0x48	0x78	0x1E	0x20	0x11	0x07	0x21
0x4	0xC5	0x06	0x37	0x4F	0xD5	0xE8	0xBB	0xE9	0x87	0xE2	0xAF	0xDE	0x26	0xEC	0x31	0xCA
0x5	0x7F	0x57	0x49	0x9E	0x05	0xA2	0x15	0xAE	0x45	0xC2	0x04	0x90	0xB6	0x2F	0xDA	0x4D
0x6	0x0A	0xFE	0x1B	0x7A	0xEF	0xA1	0x39	0xC9	0x9D	0x81	0x9B	0x29	0x76	0xBA	0xD3	0xB9
0x7	0x8E	0x33	0x7B	0xCB	0xD9	0x16	0xEA	0xA7	0xF5	0x1D	0xD7	0xD1	0x66	0x46	0x75	0x4B
0x8	0xBD	0xDF	0xE6	0xFF	0xC4	0x59	0x42	0x08	0xB5	0x98	0x2B	0xED	0x82	0x56	0xAB	0xAC
0x9	0x9C	0x89	0x2E	0xF7	0x32	0x8C	0x30	0x27	0x4C	0xA8	0x93	0x97	0x7D	0x72	0x80	0xC7
0xA	0xE0	0x24	0x4E	0x9F	0xFB	0xE3	0x10	0xF6	0x67	0x5E	0x0E	0x03	0x6F	0x6D	0xB2	0x55
0xB	0x47	0x5D	0x84	0x68	0xE7	0xF1	0xAD	0x5C	0xBE	0xCD	0xC8	0x99	0x88	0x2D	0x43	0x4A
0xC	0x5A	0x44	0x9A	0xE4	0xD2	0xF0	0xD8	0x51	0x12	0x92	0x8F	0x65	0x79	0xDB	0x01	0x41
0xD	0x91	0x5F	0x25	0x8D	0x28	0x73	0x71	0x6E	0x64	0x61	0x02	0xB8	0x0C	0x53	0x83	0xF4
0xE	0x18	0x22	0x7C	0xCF	0x58	0x17	0x00	0xD0	0x09	0x3C	0xEE	0x74	0x2A	0x69	0x36	0x77
0xF	0x1F	0x63	0x6B	0xF8	0xB4	0xBF	0x0D	0x3A	0x6C	0x50	0xC6	0x23	0xE5	0x6A	0xFA	0x54

10.2.4 SBOX1 – INVERSA

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0xE6	0xCE	0xDA	0xAB	0x5A	0x54	0x41	0x3E	0x87	0xE8	0x60	0x2F	0xDC	0xF6	0xAA	0x04
0x1	0xA6	0x3D	0xC8	0x0B	0x32	0x56	0x75	0xE5	0xE0	0x37	0x1B	0x62	0x0C	0x79	0x3B	0xF0
0x2	0x3C	0x3F	0xE1	0xFB	0xA1	0xD2	0x4C	0x97	0xD4	0x6B	0xEC	0x8A	0x29	0xBD	0x92	0x5D
0x3	0x96	0x4E	0x94	0x71	0x2C	0x20	0xEE	0x42	0x0E	0x66	0xF7	0x1D	0xE9	0x15	0x14	0x0D
0x4	0x28	0xCF	0x86	0xBE	0xC1	0x58	0x7D	0xB0	0x39	0x52	0xBF	0x7F	0x98	0x5F	0xA2	0x43
0x5	0xF9	0xC7	0x36	0xDD	0xFF	0xAF	0x8D	0x51	0xE4	0x85	0xC0	0x13	0xB7	0xB1	0xA9	0xD1
0x6	0x22	0xD9	0x08	0xF1	0xD8	0xCB	0x7C	0xA8	0xB3	0xED	0xFD	0xF2	0xF8	0xAD	0xD7	0xAC
0x7	0x09	0xD6	0x9D	0xD5	0xEB	0x7E	0x6C	0xEF	0x3A	0xCC	0x63	0x72	0xE2	0x9C	0x11	0x50
0x8	0x9E	0x69	0x8C	0xDE	0xB2	0x34	0x06	0x48	0xBC	0x91	0x01	0x1E	0x95	0xD3	0x70	0xCA
0x9	0x5B	0xD0	0xC9	0x9A	0x2B	0x00	0x19	0x9B	0x89	0xBB	0xC2	0x6A	0x90	0x68	0x53	0xA3
0xA	0x02	0x65	0x55	0x38	0x05	0x21	0x2E	0x77	0x99	0x0F	0x30	0x8E	0x8F	0xB6	0x57	0x4A
0xB	0x1F	0x24	0xAE	0x03	0xF4	0x88	0x5C	0x0A	0xDB	0x6F	0x6D	0x46	0x23	0x80	0xB8	0xF5
0xC	0x1A	0x2A	0x59	0x2D	0x84	0x40	0xFA	0x9F	0xBA	0x67	0x4F	0x73	0x33	0xB9	0x07	0xE3
0xD	0xE7	0x7B	0xC4	0x6E	0x10	0x44	0x16	0x7A	0xC6	0x74	0x5E	0xCD	0x27	0x12	0x4B	0x81
0xE	0xA0	0x25	0x49	0xA5	0xC3	0xFC	0x82	0xB4	0x45	0x47	0x76	0x18	0x4D	0x8B	0xEA	0x64
0xF	0xC5	0xB5	0x26	0x17	0xDF	0x78	0xA7	0x93	0xF3	0x1C	0xFE	0xA4	0x35	0x31	0x61	0x83

10.2.5 SBOX2 – DIRETA

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0xA6	0xB9	0xBB	0x4F	0xBA	0xE2	0x52	0xB6	0x3A	0x4A	0x16	0xF3	0x4E	0x54	0xAE	0x65
0x1	0xE8	0x18	0x42	0x04	0xFE	0x77	0x9E	0xEB	0xD2	0x90	0xDF	0x1E	0xA2	0x95	0xD5	0x20
0x2	0x13	0xC7	0x6B	0xF5	0x46	0x12	0xF7	0x48	0x8A	0xD3	0xDC	0xB4	0x28	0x31	0x92	0x2F
0x3	0x9C	0xC3	0xBD	0xCC	0x88	0x24	0xFA	0x11	0x36	0x9D	0xAD	0x56	0x8D	0xE0	0xE5	0xE9
0x4	0xEE	0x80	0x84	0x1D	0x40	0x9B	0x0F	0x78	0xD6	0x8C	0x6E	0xAC	0x0E	0xF6	0x43	0xD9
0x5	0xB0	0xCD	0x1C	0x7A	0x09	0x7F	0xAF	0x8F	0xE1	0x34	0x6D	0xB2	0xBC	0x85	0xF0	0x27
0x6	0x29	0x53	0x14	0x9A	0x2B	0x68	0x01	0x91	0xB1	0x03	0x75	0xF8	0x1A	0x62	0xEF	0x35
0x7	0x7C	0xC2	0xA9	0xE4	0x23	0x67	0xDE	0x37	0xA1	0x33	0x17	0x96	0x07	0x57	0x93	0x99
0x8	0x82	0x4C	0xB5	0x89	0x25	0x41	0x7B	0xEA	0x47	0xD0	0x38	0x2A	0x72	0x49	0xC9	0x86
0x9	0x0C	0x5A	0x21	0x66	0x50	0x98	0xA3	0x0B	0xF2	0xCA	0x8E	0xC0	0xC6	0xDB	0x19	0x05
0xA	0x3F	0xED	0x81	0x0A	0xFB	0x4D	0xC8	0x3B	0xE3	0xA8	0xD8	0x45	0x22	0x4B	0xA0	0x00
0xB	0x97	0xD7	0x7D	0x06	0xD1	0xCF	0x3E	0xF9	0xAB	0x83	0xA5	0xC5	0x1F	0x70	0xF4	0x15
0xC	0x61	0x9F	0xCE	0x58	0xFF	0x30	0xB8	0x71	0x60	0xEC	0xC1	0x2E	0xE7	0x44	0x3D	0x51
0xD	0xBF	0x6F	0xE6	0x79	0xDD	0x73	0x1B	0x69	0x6A	0x5B	0xC4	0x5C	0x02	0x5F	0xFD	0xDA
0xE	0x59	0x26	0x94	0xBE	0xB3	0xA7	0x87	0xA4	0x64	0x74	0xD4	0xCB	0x08	0xAA	0xFC	0x2D
0xF	0xB7	0x32	0x6C	0x63	0x7E	0xF1	0x2C	0x8B	0x76	0x5D	0x5E	0x55	0x3C	0x10	0x39	0x0D

10.2.6 SBOX2 – INVERSA

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0xAF	0x66	0xDC	0x69	0x13	0x9F	0xB3	0x7C	0xEC	0x54	0xA3	0x97	0x90	0xFF	0x4C	0x46
0x1	0xFD	0x37	0x25	0x20	0x62	0xBF	0x0A	0x7A	0x11	0x9E	0x6C	0xD6	0x52	0x43	0x1B	0xBC
0x2	0x1F	0x92	0xAC	0x74	0x35	0x84	0xE1	0x5F	0x2C	0x60	0x8B	0x64	0xF6	0xEF	0xCB	0x2F
0x3	0xC5	0x2D	0xF1	0x79	0x59	0x6F	0x38	0x77	0x8A	0xFE	0x08	0xA7	0xFC	0xCE	0xB6	0xA0
0x4	0x44	0x85	0x12	0x4E	0xCD	0xAB	0x24	0x88	0x27	0x8D	0x09	0xAD	0x81	0xA5	0x0C	0x03
0x5	0x94	0xCF	0x06	0x61	0x0D	0xFB	0x3B	0x7D	0xC3	0xE0	0x91	0xD9	0xDB	0xF9	0xFA	0xDD
0x6	0xC8	0xC0	0x6D	0xF3	0xE8	0x0F	0x93	0x75	0x65	0xD7	0xD8	0x22	0xF2	0x5A	0x4A	0xD1
0x7	0xBD	0xC7	0x8C	0xD5	0xE9	0x6A	0xF8	0x15	0x47	0xD3	0x53	0x86	0x70	0xB2	0xF4	0x55
0x8	0x41	0xA2	0x80	0xB9	0x42	0x5D	0x8F	0xE6	0x34	0x83	0x28	0xF7	0x49	0x3C	0x9A	0x57
0x9	0x19	0x67	0x2E	0x7E	0xE2	0x1D	0x7B	0xB0	0x95	0x7F	0x63	0x45	0x30	0x39	0x16	0xC1
0xA	0xAE	0x78	0x1C	0x96	0xE7	0xBA	0x00	0xE5	0xA9	0x72	0xED	0xB8	0x4B	0x3A	0x0E	0x56
0xB	0x50	0x68	0x5B	0xE4	0x2B	0x82	0x07	0xF0	0xC6	0x01	0x04	0x02	0x5C	0x32	0xE3	0xD0
0xC	0x9B	0xCA	0x71	0x31	0xDA	0xBB	0x9C	0x21	0xA6	0x8E	0x99	0xEB	0x33	0x51	0xC2	0xB5
0xD	0x89	0xB4	0x18	0x29	0xEA	0x1E	0x48	0xB1	0xAA	0x4F	0xDF	0x9D	0x2A	0xD4	0x76	0x1A
0xE	0x3D	0x58	0x05	0xA8	0x73	0x3E	0xD2	0xCC	0x10	0x3F	0x87	0x17	0xC9	0xA1	0x40	0x6E
0xF	0x5E	0xF5	0x98	0x0B	0xBE	0x23	0x4D	0x26	0x6B	0xB7	0x36	0xA4	0xEE	0xDE	0x14	0xC4

10.3APÊNDICE III: CÓDIGOS FONTES

10.3.1 GERAÇÃO DAS SBOX DA CIFRA FLEXAE EM JAVA

```
package br.eb.ime.genSBoxflexae;
import java.io.IOException;
import javax.xml.bind.DatatypeConverter;

public class GenSBoxFlexAE {

    public static byte multiGF8( byte input1, byte input2, byte modP8)
    {
        byte state1 = 0x0;
        byte state2;
        byte i1 = input1;

        state2=input2;
        for(int i=0;i<8;i++)
        {
            if((i1&0x1)==1)
                state1=(byte) (state1^state2);
            i1 = (byte) (i1>>>1);
            if(state2<0)
            {
                state2 = (byte) (state2 << 1);
                state2 = (byte) (state2 ^ modP8);
            }
            else
                state2 = (byte) (state2 << 1);
        }
        return state1;
    }

    public static byte invMultiGF8( byte input1, byte modP8 )
    {
        byte invmulti, state1;
        invmulti=1;
        while(invmulti != 0)
        {
            state1=multiGF8(invmulti,input1,modP8);
            if(state1==1) return invmulti;
            invmulti++;
        }
        return invmulti;
    }

    public static byte affTransf( byte input1, byte addConst, byte multConst )
    {
        byte afftransf, state1;
        state1=multiGF8((byte) multConst, input1, (byte) 1);
        afftransf = (byte) (state1^addConst);
        return afftransf;
    }


    public static byte[] genSBox( byte IP , byte MC, byte AC )
    {
        byte [] SBox = new byte [256];
        SBox[0]=(byte) AC;
        for(int i1=1; i1<256; i1++)
        {
            SBox[i1] = invMultiGF8( (byte) (i1 & 0xff) , IP);
            SBox[i1] = affTransf(SBox[i1], AC, MC );
        }
        return SBox;
    }

    public GenSBoxFlexAE()
    {
        byte IP0  = 0b00011011;  // irreductible polynomial x^8+x^4+x^3+x^1+1
        byte MC0  = (byte) 0x1F; // Multiplicative Constant = 0x1F
        byte AC0  = (byte) 0x63; // Additive Constant = 0x63
        byte [] SBox0 = genSBox(IP0,MC0,AC0);

        byte IP1  = 0b00011101;  // irreductible polynomial x^8+x^4+x^3+x^2+1
        byte MC1  = (byte) 0x1F; // Multiplicative Constant = 0x1F
        byte AC1  = (byte) 0x95; // Additive Constant = 0x95
        byte [] SBox1 = genSBox(IP1,MC1,AC1);
```

```
        byte IP2  = 0b00101011;  // irreductible polynomial x^8+x^5+x^3+x^1+1
        byte MC2  = (byte) 0x1F; // Multiplicative Constant = 0x1F
        byte AC2  = (byte) 0xA3; // Additive Constant = 0xA3
        byte [] SBox2 = genSBox(IP2,MC2,AC2);
    }

    public static void main(String[] args) throws IOException
    {
        GenSBoxFlexAE gf = new GenSBoxFlexAE();
    }
}
```

10.3.2 GERAÇÃO DAS TABELAS DE DISTRIBUIÇÃO DAS DIFERENÇAS

```
    public static int[][] genDifTable(int [] SBox)
    {
        int i1,i2,dx,dy;
        int difTable[][];

        difTable= new int[256][256];

        for(i1=0;i1<256;i1++)
        {
            for(i2=0;i2<256;i2++)
            {
                dx = i1^i2;
                dy = SBox[i1]^SBox[i2];
                difTable[dx][dy]++;
            }
        }
        return difTable;
    }
```

10.3.3 GERAÇÃO DAS TABELAS DE APROXIMAÇÃO LINEAR

```
public static int [][] linearTable(int []SBox)
    {
        int linTable[][];
        int ax,by,parityAX,parityBY;
        long steps=0;

        linTable = new int [SBox.length][SBox.length];

        for(int x=0,y=0;x<SBox.length;x++)
        {
            y=SBox[x];
            for(int a=0;a<SBox.length;a++)
            {
                for(int b=0;b<SBox.length;b++)
                {
                    // initialize the Linear App Table
                    if(x==0) linTable[a][b]=-(SBox.length/2);

                    ax = a&x;
                    by = b&y;
                    parityAX=0;
                    parityBY=0;
                    for(int e=1; e<(SBox.length); e*=2)
                    {
                        parityAX = parityAX^(ax&1);
                        parityBY = parityBY^(by&1);
                        ax=ax>>1;
                        by=by>>1;
                    }
                    if(parityAX==parityBY) linTable[a][b]++;
                    if((steps++%10000000)==0) System.out.println("steps="+steps);
                }
            }
        }
        return linTable;
    };
```