UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA

Otávio de Souza Martins Gomes

Uma arquitetura híbrida de algoritmos criptográficos, utilizando *hardware* reconfigurável, com foco em vulnerabilidades baseadas em análises do tipo *side-channel*

Itajubá/MG, Dezembro de 2016.

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA

Otávio de Souza Martins Gomes

Uma arquitetura híbrida de algoritmos criptográficos, utilizando *hardware* reconfigurável, com foco em vulnerabilidades baseadas em análises do tipo *side-channel*

Texto submetido ao Programa de Pós-Graduação em Engenharia Elétrica para a defesa da tese, um dos requisitos para obtenção do Título de Doutor em Ciências em Engenharia Elétrica.

Área de Concentração: Microeletrônica

Orientador: Prof. Dr. Robson Luiz Moreno

Itajubá/MG, Dezembro de 2016.

633 Gomes, Otávio de Souza Martins
Uma arquitetura híbrida de algoritmos criptográficos, utilizando hardware reconfigurável, com foco em vulnerabilidades baseadas em análises do tipo side-chann / Otávio de Souza Martins Gomes. -- Itajubá, 2016.
149 f.

Orientador: Robson Luiz Moreno.
Tese (Doutorado - Programa de Pós-Graduação em Engenharia Elétrica) -- Universidade Federal de Itajubá, 2016.

1. criptografia. 2. fpga. 3. aes. 4. twofish. 5. segurança. I. Moreno, Robson Luiz. II. Título.

Dedico este trabalho à minha amada família:
Vandir, Regina, Matheus e Áthila.

"Porque o SENHOR dá a sabedoria; da sua boca é que vem o conhecimento e o entendimento." (Pv 2:6)

"Riquezas e glória vêm de diante de Ti, e Tu dominas sobre tudo, e na Tua mão há força e poder;
e na Tua mão está o engrandecer e o dar força a tudo. (...)
Como a sombra são os nossos dias sobre a terra, e sem Ti não há esperança."
(I Cr 29:11-16)

"Para que os seus corações sejam consolados, e estejam unidos em amor, e enriquecidos da plenitude da inteligência, para conhecimento do mistério de Deus e Pai, e de Cristo,
Em quem estão escondidos todos os tesouros da sabedoria e da ciência.
E digo isto, para que ninguém vos engane com palavras persuasivas".
(Colossenses 2:2-4)

"Não suceda que, depois de teres comido à saciedade, de teres construído e habitado formosas casas, de teres visto multiplicar teus bois e tuas ovelhas, e aumentar a tua prata, o teu ouro e o teu bem, o teu coração se eleve, e te esqueças do Senhor, teu Deus (...). Foi ele o teu guia neste vasto e terrível deserto, cheio de serpentes ardentes e escorpiões, terra árida e sem água, onde fez jorrar para ti água do rochedo duríssimo; foi ele quem te alimentou no deserto com um maná desconhecido de teus pais, para humilhar-te e provar-te, a fim de te fazer o bem depois disso.
Não digas no teu coração: a minha força e o vigor do meu braço adquiriram-me todos esses bens.
Lembra-te de que é o Senhor, teu Deus, quem te dá a força para adquiri-los, a fim de confirmar, como o faz hoje, a aliança que jurou a teus pais". (Deuteronômio 8:11-18)

Agradecimentos

Agradeço, primeiramente, a Deus por me capacitar a concluir este trabalho com êxito. *"Porque o SENHOR dá a sabedoria; da sua boca é que vem o conhecimento e o entendimento."* (Pv 2:6)

À minha família por ser meu suporte e auxílio em todo o tempo.

À minha namorada Fernanda, por seu carinho, apoio e compreensão durante a fase final desta jornada.

Agradeço a meu professor e orientador, Robson Luiz Moreno, pelo companheirismo, ajuda e esclarecimentos sem os quais a realização deste trabalho não seria possível.

Aos colegas do Grupo de Microeletrônica da UNIFEI e aos colegas do IFMG campus Formiga pelo companheirismo, sugestões e contribuições que permitiram o aperfeiçoamento deste.

Meus mais sinceros agradecimentos.

Resumo

Este trabalho apresenta um dispositivo de criptografia, desenvolvido em *hardware* reconfigurável, que utiliza características complementares de algoritmos de criptografia simétrica (AES-Rijndael e Twofish) para a construção de um dispositivo com chave de 256 bits. Foi proposta uma melhoria no *S-Box* do algoritmo Twofish que permitiu uma menor ocupação de área no dispositivo. A combinação dos algoritmos eliminou algumas limitações e vulnerabilidades conhecidas destes algoritmos quando implementados em *hardware* e poderá ser utilizada na área de redes (automotivas, V2V e V2I), dispositivos médicos, medição inteligente de energia elétrica, sistemas SCADA, entre outros.

Abstract

This work presents a cryptographic device, developed in reconfigurable hardware, which uses complementary characteristics of symmetric encryption algorithms (AES-Rijndael and Twofish) to construct a device with a 256-bit key. The proposal of an improvement on S-Box of the Twofish algorithm allowed a smaller area occupation in hardware. The combination of algorithms has eliminated some limitations and vulnerabilities known when implemented in hardware and can be used in the networking area (V2V and V2I), medical devices, smart metering, SCADA systems, among others.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

3DES	*Triple Data Encryption Standard*
AES	*Advanced Encryption Standard*
ASCII	*American Standard Code for Information Interchange*
ASIC	*Application-specific Integrated Circuit*
CBC	*Cipher Block Chaining*
CM	*Cryptographic Module*
CPLD	*Complex Programmable Logic Device*
DES	*Data Encryption Standard*
FPGA	*Field Programmable Gate Array*
HCD	*Hardware Cryptographic Device*
HSM	*Hardware Security Module* ou *Host Security Module*
IEEE	*Institute of Electrical and Electronics Engineers*
ITRC	*Identity Theft Resource Center*
LUT	*Look-Up Table*
NIST	*National Institute of Standards and Technology*
PCSM	*Personal Computer Security Module*
PGP	*Pretty Good Privacy*
PLD	*Programmable Logic Device*
RSA	*Rivest-Shamir-Adleman*
SAM	*Secure Application Module*
SCADA	*Supervisory Control and Data Acquisition*
SCD	*Secure Cryptographic Device*
SoC	*System on Chip*
SSCD	*Secure Signature Creation Device*
TRSM	*Tamper Resistant Security Module*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
XOR	*Exclusive Or*

Capítulo 1

Introdução

Antigamente, a privacidade e a segurança na comunicação entre pessoas podiam ser garantidas pela realização de encontros reservados. Para a comunicação à distância seria necessário enviar mensagens através de intermediários e, para manter a segurança, foram desenvolvidos códigos e cifras para esconder o conteúdo das mensagens daqueles que as interceptassem sem permissão. As telecomunicações aumentaram a rapidez da comunicação remota e, no século 20, o uso da criptografia foi automatizado, para tornar mais rápida e eficaz sua aplicação [1].

A criptologia é a ciência de desenvolver e descobrir tais cifras, principalmente nas áreas diplomáticas e militares dos governos. A criptografia é importante no âmbito da tecnologia de informação para que se possa garantir a segurança em todo o ambiente computacional que necessita de sigilo em relação às informações. Atualmente, a criptografia é usada como uma técnica de transformação de dados, segundo um código ou algoritmo, para que eles se tornem ininteligíveis, a não ser para quem possui a chave para a tradução do código. Há diversos órgãos governamentais e instituições que verificam, normatizam e controlam os padrões de segurança de dados [2].

Em 1977, foi desenvolvido pela IBM o algoritmo DES (*Data Encryption Standard*), utilizando criptografia de chave com tamanho de 56 bits. Seus pacotes de dados seguros foram quebrados pela primeira vez em 1997 em um desafio feito a alguns grupos de desenvolvedores. As empresas *Electronic Frontier Foundation* (EFF) e *Distributed.com*, utilizando a técnica de força bruta, conseguiram quebrar em pouco tempo este algoritmo que era dito como indecifrável [3].

Em 1997, o instituto norte-americano NIST (*National Institute of Standards and Technology*) iniciou um concurso para adotar um novo algoritmo que substituiria o DES e que passaria a se chamar AES (*Advanced Encryption Standard*). O objetivo deste concurso foi estabelecer um novo algoritmo de criptografia simétrica que seria utilizado para proteger os dados confidenciais das agências norte-americanas. Este algoritmo deveria atender a alguns requisitos como: direitos autorais livres; maior rapidez em relação ao 3DES (*Triple DES* - uma melhoria do

DES); cifrar em blocos de 128 bits com chaves de 128, 192 e 256 bits; além da possibilidade de utilização em *software* e *hardware* [4].

É importante ressaltar que um dos princípios da criptografia moderna é que o nível de segurança está na força da chave e não no algoritmo. Os métodos e algoritmos são amplamente divulgados, enquanto as chaves devem permanecer secretas [1,2,5-7]. Por este motivo, durante a competição do algoritmo de criptografia simétrica AES foram realizados vários testes para verificar a eficiência dos algoritmos em diversos aspectos computacionais: rendimento em *software* e em *hardware*, desempenho em diversas arquiteturas de microprocessadores, entre outros. Os cinco algoritmos finalistas da competição foram amplamente testados por especialistas do mundo todo e possuem características distintas que podem ser utilizadas para aplicações específicas. A partir da publicação do resultado final do concurso realizado pelo NIST foram apresentados vários questionamentos acerca da eficácia do vencedor da competição, o algoritmo Rijndael [8].

Em 2000, após uma filtragem de alguns candidatos e análises de especialistas na área de criptografia, foram escolhidas 15 propostas de algoritmos que possuíam mérito para a competição. Os cinco finalistas deste concurso foram os algoritmos MARS, RC6, Rijndael, *Serpent* e *Twofish*. O vencedor foi o algoritmo Rijndael, que foi desenvolvido pelos belgas Vincent Rijmen e Joan Daemen [4].

O algoritmo AES-Rijndael, quando desenvolvido em *hardware*, apresenta diferenças entre os processos de cifragem e de decifragem que influenciam em seu desempenho. Outro finalista, o algoritmo Twofish, possui algumas características que o diferenciam do Rijndael tanto na matemática utilizada para a realização da cifragem, quanto em sua modularização em *hardware*, além de apresentar um nível de segurança maior do que o Rijndael durante a competição [8]. Estas características podem ser utilizadas para complementar o funcionamento do AES. Cada um dos algoritmos finalistas possui características que podem ser utilizadas de modo a eliminar deficiências e fragilidades conhecidas em seus adversários na competição.

As fragilidades ou vulnerabilidades estão presentes em diversas áreas durante o desenvolvimento de sistemas e dispositivos. Em *software*, elas podem ser caracterizadas pela instalação de programas maliciosos ou pelo não preenchimento de alguns requisitos de segurança durante o desenvolvimento de sistemas. Com relação às redes de comunicação de dados, a transmissão das informações pode ser interceptada (*sniffing*) ou alterada (*spoofing*) por intrusos, o que violaria a privacidade da comunicação ou a integridade da mensagem, respectivamente [9].

As vulnerabilidades relacionadas ao ambiente físico relacionam-se às emissões sonoras ou eletromagnéticas realizadas pelos dispositivos durante o processamento das informações. Os

ataques relacionados ao ambiente físico tem recebido um grande foco na área acadêmica e industrial devido ao grande número de dispositivos que apresentam essa vulnerabilidade [10-11].

Os ataques que utilizam vulnerabilidades relativas a emissões do dispositivo podem ser classificados em dois grupos: aqueles que podem ser realizados à distância ou que necessitam de acesso físico ao sistema. Neste último cenário estão presentes as análises do tipo *side-channel* que interceptam as emissões eletromagnéticas dos dispositivos com a finalidade de descobrir a informação que está sendo processada. Este cenário se apresenta como um ponto de vulnerabilidade para os algoritmos de criptografia simétrica por permitirem a obtenção da chave que está sendo utilizada [9-13].

1.1 Proposta da Tese

Baseado nestas informações, a proposta desta tese é: a combinação das características de algoritmos finalistas da competição AES pode apresentar maior segurança e escalabilidade quando desenvolvido em *hardware*. Com isto, pretende-se aumentar a segurança do processo de criptografia, minimizar atrasos inerentes aos algoritmos e, através do processamento destes sinais, que não possuem dependência entre si, descaracterizar a emissão da informação que está sendo processada.

Merece destaque o ataque do tipo *side-channel*, que foi publicado antes da competição e hoje se apresenta como uma vulnerabilidade do AES-Rijndael. Essa vulnerabilidade, apesar de ser conhecida na época, não foi utilizada como critério durante a avaliação dos competidores [12].

Com base nestas informações e na proposta de tese, este trabalho foi intitulado: “Uma arquitetura híbrida de algoritmos criptográficos, utilizando *hardware* reconfigurável, com foco em vulnerabilidades baseadas em análises do tipo *side-channel*”. A escolha de um *hardware* reconfigurável para o desenvolvimento deste trabalho é justificada por sua flexibilidade quanto às alterações e seu custo e tempo de desenvolvimento reduzidos se comparado a um circuito integrado de aplicação específica (ASIC) [5].

Este trabalho foi desenvolvido tendo em vista as diversas aplicações que podem ser beneficiadas com uma interface de criptografia em *hardware* como, por exemplo:

- Monitoramento de tráfego e controle das vias de trânsito [14];
- Carros conectados e sistemas de auxílio à direção [15];
- Comunicação segura utilizando o barramento automotivo [16-18];
- Privacidade, segurança e navegação anônima [19].

1.2 Estrutura do trabalho

O Capítulo 2 apresentará o referencial teórico, que consta de uma breve revisão histórica da área de criptografia e da apresentação do funcionamento dos algoritmos Rijndael e *Twofish*. Além destes tópicos será apresentada uma análise de algumas vulnerabilidades atuais, das vantagens e desvantagens do desenvolvimento de criptografia em *hardware* e *software*; e peculiaridades dos algoritmos utilizados pelo NIST. Finalizando este capítulo serão apresentadas algumas informações sobre FPGA e linguagens de descrição de *hardware*.

O Capítulo 3 apresentará a metodologia e o desenvolvimento deste trabalho. O Capítulo 4 discutirá os resultados obtidos. O Capítulo 5 apresentará as conclusões e o Capítulo 6 as propostas de trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo será apresentado um breve resumo a respeito de segurança da informação, seguido da descrição do funcionamento dos algoritmos AES-Rijndael e Twofish. Serão apresentadas algumas aplicações de algoritmos de criptografia, assim como os padrões industriais e algumas vulnerabilidades conhecidas. Merece destaque a apresentação da análise do tipo *side-channel* e as possíveis medidas de contenção que podem ser utilizadas para esta vulnerabilidade. Ao final serão apresentados alguns conceitos a respeito de *hardware* reconfigurável e linguagens de descrição de *hardware*.

2.1 Segurança da Informação: Criptografia

A segurança da informação digital é uma questão de preocupação no mundo atual. Por isso foram desenvolvidas várias técnicas para dificultar a aquisição de dados confidenciais por aqueles que não deveriam ter acesso a eles. Uma dessas técnicas é a criptografia digital, que consiste em transformar informações em códigos para evitar o acesso de pessoas não autorizadas ao conteúdo da mensagem. Algumas destas técnicas são utilizadas desde a antiguidade [1].

Com o surgimento dos computadores, vários tipos de algoritmos criptográficos foram desenvolvidos. Os algoritmos considerados seguros necessitam de grande quantidade de processamento e algumas vezes o tamanho da mensagem processada aumenta significativamente. Diversos modelos de criptografia se tornaram ultrapassados devido à velocidade com que sua mensagem pode ser violada, por possuírem um baixo nível de segurança. Para o aumento da segurança é necessário que sejam utilizadas chaves maiores, ocorrendo um aumento significativo no tamanho da informação a ser transferida, o que pode tornar inviável a transmissão dos dados. Novos modelos de criptografia têm sido desenvolvidos ou modificados com o objetivo de aumentar a segurança e garantir uma velocidade adequada na transmissão de dados [2,5].

2.1.1 Histórico

Criptologia é o estudo de códigos e cifras. Deriva das palavras gregas *kryptos* (oculto) e *logos* (palavra ou verbo) e envolve as áreas de criptografia e criptoanálise. A criptografia, concatenação do termo grego *kryptos* e da palavra *grapho* (escrita), apresenta-se como a ciência capaz de prover meios através dos quais seja possível transformar um texto inteligível em um texto ininteligível (criptografado). A criptoanálise é o estudo de como "quebrar" os mecanismos criptográficos. A palavra cifra se origina do hebraico *saphar*, que significa "dar número" e representa uma combinação de caracteres devidamente estabelecida para transformar uma mensagem. A maioria das técnicas de cifragem são intrinsicamente sistemáticas e, frequentemente, baseadas em sistemas numéricos [1,2,6].

A criptografia é tão antiga quanto a própria escrita, visto que já estava presente no sistema de escrita hieroglífica dos egípcios. Os romanos utilizavam códigos secretos para comunicar planos de batalha. Com as guerras mundiais e a invenção do computador, a criptografia cresceu incorporando complexos algoritmos matemáticos [1].

A criptologia faz parte da história humana porque sempre houve fórmulas secretas e informações confidenciais que não deveriam cair no domínio público ou na mão de inimigos. O primeiro exemplo documentado da escrita cifrada relaciona-se aproximadamente ao ano de 1900 a.C., quando o escriba de *Khnumhotep II* (Egito antigo) teve a ideia de substituir algumas palavras ou trechos de texto. Caso o documento fosse roubado, o ladrão não encontraria o caminho que o levaria ao tesouro e morreria de fome perdido nas catacumbas da pirâmide [1].

Em 50 a.C., Júlio César usou sua famosa cifra de substituição (figura 2.1) para cifrar (criptografar) comunicações governamentais [20]. Para compor seu texto cifrado, César alterou letras deslocando-as de três posições; A se tornava D, B se tornava E etc.

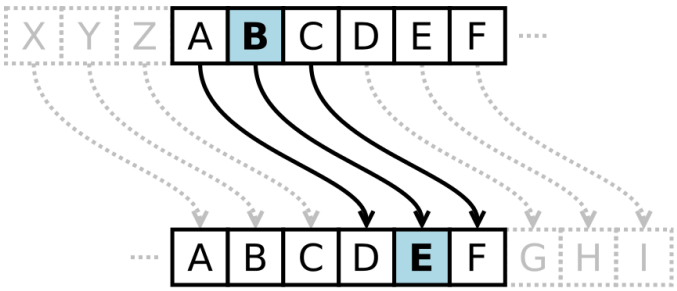


Figura 2.1: Cifra de césar [20].

César reforçava seu método de criptografar mensagens substituindo letras latinas por gregas. O código de César é o único da antiguidade que é usado até hoje. Cifras baseadas na

substituição cíclica do alfabeto são denominadas código de César. Essa cifra foi utilizada pelos oficiais sulistas na Guerra de Secessão americana e pelo exército russo em 1915 [1,6].

No início de 1900, o padre brasileiro Roberto Landell de Moura realizou a primeira transmissão sem fio de sons e sinais telegráficos. Ele foi pioneiro na transmissão da voz humana sem fio, antecedendo alguns inventores como o canadense Reginald Fessenden e o italiano Guglielmo Marconi. Por seu pioneirismo, o Padre Landell é o patrono dos radioamadores do Brasil sendo, também, homenageado pelo o CPqD (Centro de Pesquisas e Desenvolvimento), através da denominação de seu Centro de Pesquisa e Desenvolvimento [21-24].

Apesar da vantagem de uma comunicação de longa distância sem o uso de fios ou cabos, o sistema é aberto e aumentou o desafio da criptologia. Em 1921, Edward Hugh Hebern fundou a *Hebern Electric Code*, uma empresa produtora de máquinas de cifragem eletromecânicas baseadas em rotores com o objetivo de suprir a necessidade de segurança na transmissão destas informações [2,5].

Entre 1933 e 1945, a máquina Enigma, que havia sido criada por Arthur Scherbius, foi aperfeiçoada até se transformar na ferramenta criptográfica mais importante da Alemanha nazista. Uma das quebras desta máquina foi realizada pelo matemático polonês Marian Rejewski que se baseou apenas em textos cifrados interceptados e numa lista de chaves obtidas por um espião. Quando os poloneses quebraram a Enigma, a cifra era alterada apenas uma vez a cada poucos meses. Com o advento da guerra, a cifra era trocada pelo menos uma vez por dia, dando 159 milhões de milhões de configurações possíveis para escolher. Os poloneses decidiram informar os britânicos em julho de 1939, uma vez que precisavam de ajuda para quebrar a Enigma devido à iminente invasão da Polônia [25].

A primeira quebra operacional da máquina Enigma (figura 2.2) veio por volta de 23 de Janeiro de 1940, quando a equipe que trabalhava sob a coordenação de Dilly Knox, que continha os matemáticos John Jeffreys, Peter Twinn e Alan Turing, desvendou a chave principal do exército alemão. Ficou conhecido em *Bletchley Park* como *"The Green"*. Encorajado por este sucesso, os quebradores de código conseguiram quebrar a chave *'Red'* usada pelos agentes da *Luftwaffe* [26].

Figura 2.2: Enigma [26].

2.1.2 Tipos de Criptografia

Os algoritmos de criptografia podem ser classificados por meio do tratamento dado às informações que serão processadas; assim, têm-se os algoritmos de bloco e os algoritmos de fluxo.

Um algoritmo de blocos opera sobre um conjunto de dados. O texto antes de ser cifrado é dividido em blocos que serão cifrados ou decifrados. Quando o texto não completa o número de *bytes* de um bloco, este é preenchido com dados conhecidos (geralmente valor zero “0”) até completar o número de *bytes* necessário, cujo tamanho já é predefinido pelo algoritmo que está sendo usado.

Um problema na cifra de bloco é que se o mesmo bloco de texto simples aparecer mais de uma vez, a cifra gerada será a mesma, facilitando o ataque ao texto cifrado. Para resolver esse problema são utilizados os modos de realimentação [5].

O modo mais comum de realimentação é a cifragem de blocos por encadeamento CBC (*Cipher Block Chaining*). Neste modo é realizada uma operação de XOR (Ou-Exclusivo) do bloco atual de texto simples com o bloco anterior de texto cifrado. Para o primeiro bloco, não há bloco anterior de texto cifrado; assim, faz-se uma XOR com um vetor de inicialização. Este modo não adiciona nenhuma segurança extra, apenas evita o problema citado anteriormente [1].

Os algoritmos de blocos processam os dados como um conjunto de bits, sendo os mais rápidos e seguros para a comunicação digital. Outra vantagem é que os blocos podem ser codificados fora de ordem. Deste modo, o processo se torna resistente a erros, uma vez que um bloco não depende de outro. Como desvantagem, se a mensagem possuir padrões repetitivos nos blocos, o texto cifrado também os apresentará, o que facilitará o serviço do criptoanalista [5].

Os algoritmos de fluxo cifram a mensagem bit a bit, em um fluxo contínuo, sem esperar que se tenha um bloco completo de bits. É também chamado de criptografia em fluxo de dados, onde a cifragem se dá mediante uma operação XOR entre o bit de dados e o bit gerado pela chave.

Com relação ao número de chaves, pode-se classificar a criptografia como simétrica ou assimétrica. Na criptografia de chave simétrica, os processos de cifragem e decifragem são feitos com uma única chave, ou seja, tanto o remetente quanto o destinatário usam a mesma chave. Em algoritmos simétricos, como, por exemplo, o DES, ocorre o chamado "problema de distribuição de chaves". A chave tem de ser enviada para todos os usuários autorizados antes que as mensagens possam ser trocadas. Essa ação resulta num atraso de tempo e possibilita que a chave chegue a pessoas não autorizadas [5,7].

A criptografia assimétrica contorna o problema da distribuição de chaves mediante o uso de chaves públicas. A criptografia de chaves públicas foi desenvolvida em 1976 por Whitfield Diffie e Martin Hellman, a fim de resolver o problema da distribuição de chaves [1,2]. Neste novo sistema, cada pessoa tem um par de chaves denominado chave pública e chave privada. A chave pública é divulgada, enquanto a chave privada é mantida em segredo. Para mandar uma mensagem privada, o transmissor cifra a mensagem usando a chave pública do destinatário pretendido, que deverá usar a sua respectiva chave privada para conseguir recuperar a mensagem original. Um exemplo de criptossistema de chave pública é o RSA (Rivest-Shamir-Adleman), cuja maior desvantagem é a sua capacidade de canal limitada, ou seja, o número de bits de mensagem que pode transmitir por segundo [5,27].

O termo autenticação se refere ao uso de assinaturas digitais. A assinatura é um conjunto de dados que não pode ser forjado, assegurando o nome do autor. Funciona como uma assinatura de documentos, ou seja, representa que determinada pessoa concordou com o que estava escrito. Tal procedimento também evita que a pessoa que assinou a mensagem possa alegar que a mensagem foi forjada [1,2].

Neste trabalho são utilizados algoritmos que trabalham com cifragem de blocos e criptografia simétrica.

2.2 AES (*Advanced Encryption Standard*)

Os algoritmos apresentados respeitam a metodologia matemática utilizada originalmente em cada uma das funções, de modo que a segurança intrínseca aos procedimentos não é alterada [4, 8, 28-33].

O atual padrão de criptografia dos EUA se originou de um concurso iniciado em 1997 pelo NIST. Houve a necessidade de escolher um algoritmo mais seguro e eficiente para substituir o DES, que apresentou fragilidades. Em 1998, na Primeira Conferência dos Candidatos AES, apresentaram-se 15 candidatos, como pode ser visto na tabela 2.1. A primeira rodada ocorreu em Junho de 1998, com os 15 candidatos apresentados. Esta rodada avaliou o nível de eficiência e segurança em *software*, isto é, verificou, basicamente, velocidade de acesso à memória e ocupação da mesma.

Em Agosto de 1999 ocorreu a segunda rodada que avaliou o nível de eficiência e segurança em *hardware*, para esta etapa só foram classificados cinco algoritmos: MARS, RC6, Rijndael, *Serpent* e *Twofish*. Esta segunda análise levou em conta velocidade de processamento, área ocupada pela implementação em *hardware* e consumo de potência.

Tabela 2.1: Os 15 candidatos aceitos para a 1ª rodada de avaliação do AES

ALGORITMO	RESPONSÁVEL	TIPO DE SUBMISSÃO	TIPO DE ALGORITMO
CAST-256	Entrust (CA)	Empresa	Rede Feistel Modificada
Crypton	Future Systems (KR)	Empresa	Rede de Substituição Linear
DEAL	Outerbridge, Knudsen (USA-DK)	Pesquisador	Rede Feistel
DFC	ENS-CNRS (FR)	Pesquisador	Rede Feistel
E2	NTT (JP)	Empresa	Rede Feistel
Frog	TecApro (CR)	Empresa	Outro tipo
HPC	Schroeppel (USA)	Pesquisador	Outro tipo
LOKI97	Brown et al. (AU)	Pesquisador	Rede Feistel
Magenta	Deutsche Telekom (DE)	Empresa	Rede Feistel
Mars	**IBM (USA)**	**Empresa**	**Rede Feistel Modificada**
RC6	**RSA (USA)**	**Empresa**	**Rede Feistel Modificada**
Rijndael	**Daemen and Rijmen (BE)**	**Pesquisador**	**Rede de Substituição Linear**
SAFER+	Cylink (USA)	Empresa	Rede de Substituição Linear
Serpent	**Anderson, Biham, Knudsen (UK-IL-DK)**	**Pesquisador**	**Rede de Substituição Linear**
Twofish	**Counterpane (USA)**	**Empresa**	**Rede Feistel**

O novo algoritmo deveria atender a uma lista de critérios de avaliação [4,29]:

- Com relação à segurança (o fator mais importante):
  - Maior segurança quando comparado aos outros competidores;
  - O bloco de saída do algoritmo não pode apresentar uma vulnerabilidade devido a uma permutação aleatória no bloco de entrada;
  - Uma base matemática sólida para a segurança do algoritmo; e

- Outros fatores de segurança apresentados pelo público durante o processo de avaliação, incluindo quaisquer ataques que demonstrem que a força do algoritmo foi prejudicada.

- Custo de desenvolvimento, processamento e divulgação:
  - Não possuir patentes. Licença disponível mundialmente, de maneira gratuita;
  - Ser mais rápido que o 3DES (uma variação recursiva do antigo padrão DES);
  - Eficiência computacional; e
  - Requisitos de ocupação de memória.

- Características do algoritmo:
  - Flexibilidade com relação a tamanho de chaves e blocos adicionais, cifra de fluxo, ampla variedade de plataformas para execução, entre outras características;
  - Adequação à execução em *hardware* e *software*;
  - Cifrar em blocos de 128 bits usando chaves de 128, 192 e 256 bits; e
  - Simplicidade.

Com relação ao critério segurança, é interessante notar que os ataques do tipo *side-channel* não foram apresentados pelo público durante a competição e não foram utilizados como critérios de vulnerabilidade mesmo existindo publicações que os demonstravam [4, 29, 34]. Atualmente foi demonstrado que estes ataques prejudicam a força do algoritmo [13].

Em 2000, foi conhecido o vencedor: Rijndael. O algoritmo, criado pelos belgas Vincent Rijmen e Joan Daemen, foi escolhido por ter obtido a melhor pontuação na soma dos critérios: segurança, bom desempenho em *software* e *hardware*, entre outros atributos [4].

2.2.1 Algoritmo AES-Rijndael

Dentre os critérios apresentados pelo NIST está a utilização de um bloco de entrada de 128 bits. Neste caso, é utilizado um bloco de palavra de quatro linhas e quatro colunas, com 8 bits em cada célula. No caso do algoritmo vencedor, para que o processo de cifragem com uma chave de 128 bits esteja completo é necessário que sejam realizadas 10 rodadas. A tabela 2.2 mostra as possíveis configurações para que se utilize o algoritmo AES. Para o AES 192 e 256 mantém-se a

palavra de 128 bits, alterando o tamanho da chave para 192 e 256 bits [29]. As seções seguintes irão apresentar o processamento realizado em cada uma das rodadas para o processamento da palavra e da chave. Uma apresentação do funcionamento completo de uma rodada deste algoritmo é fornecida no Apêndice A.

Tabela 2.2: Configurações do AES

AES	Palavra	Chave	Rodadas
128	4 x 4	4 x 4	10
192	4 x 4	4 x 6	12
256	4 x 4	4 x 8	14

a) Matemática utilizada no AES

O algoritmo de criptografia AES, trabalha sobre o Campo de Galois (GF), ou seja, todas as operações matemáticas utilizadas são realizadas sobre este campo, e usa-se o polinômio irredutível $m(x) = x^8 + x^4 + x^3 + x + 1$ [29].

A utilização de campo de Galois, ou campos finitos, em criptografia, é a garantia da existência de uma operação inversa para cada etapa, que é fundamental no processo de decifragem. Ao se utilizar $GF(2^n)$, com n pertencente aos números naturais, têm-se outro aspecto interessante: a soma coincide com a operação Ou-Exclusivo, operação muito rápida computacionalmente. Para a criptografia, em particular, o uso de $GF(2^8)$ é bastante adequado, visto que esse campo tem 2^8 elementos, o mesmo número de caracteres da tabela ASCII (*American Standard Code for Information Interchange*) estendida. Assim, é possível cifrar e decifrar qualquer mensagem. Todas as etapas de execução do AES, com exceção da função *ShiftRows* (que realiza somente um deslocamento de bits), utilizam como fundamento para a construção das tabelas e matrizes o campo $GF(2^8)$ [29].

b) Expansão da Chave

Para a realização da expansão da chave utiliza-se uma matriz de constantes que é dada pela matemática polinomial de Galois. A matriz de 4 linhas e 15 colunas está representada na tabela 2.3. Sua utilização varia de acordo com o tamanho da chave a ser utilizada (128, 192 ou 256 bits). Para a expansão de uma chave de 128 bits são utilizadas as dez colunas mais à esquerda (de 01 a 36). Para chaves de 192 bits são utilizadas as doze primeiras colunas. Finalmente, para chaves de 256 bits são utilizadas as 14 primeiras colunas [4].

01	02	04	08	10	20	40	80	1B	36	6C	D8	AB	4D
00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00

Tabela 2.3: Constante das Rodadas

A geração das sub-chaves inicia-se com o processamento da primeira coluna (à esquerda) da tabela 2.3, dando origem à primeira sub-chave. Esta primeira sub-chave é utilizada, juntamente com a segunda coluna (02 00 00 00), para a geração da segunda sub-chave e, assim, sucessivamente. O processo de cifragem utiliza as sub-chaves na ordem em que são geradas, iniciando na primeira sub-chave até a sub-chave correspondente à segurança do algoritmo, isto é, até a sub-chave 10 para 128 bits, até a sub-chave 12 para 192 bits e até a sub-chave 14 para 256 bits. Devido a esta característica o processo de cifragem pode ser executado de maneira simultânea ao processo de expansão das chaves.

O processo de decifragem utiliza as sub-chaves no sentido inverso, da última para a primeira. Esta característica causa um atraso inicial no processo de decifragem, pois é necessário realizar toda a expansão das chaves antes que o processo seja iniciado. No caso de uma decifragem de 256 bits, é necessário que sejam calculadas as 14 sub-chaves e todas precisam ser armazenadas para posterior utilização. Esta diferença entre os processos causa um atraso inicial no processo de decifragem e a necessidade de um dispositivo de armazenamento para as sub-chaves previamente calculadas [28,29].

c) Cifrando a Mensagem

A cifragem da palavra segue o fluxo mostrado na figura 2.3, onde é inserida uma mensagem (texto claro) que passa por várias operações até que fique cifrada. Este processo faz com que uma mensagem legível seja criptografada e não consiga ser lida por ninguém além daquele que possui a chave para decifrá-la.

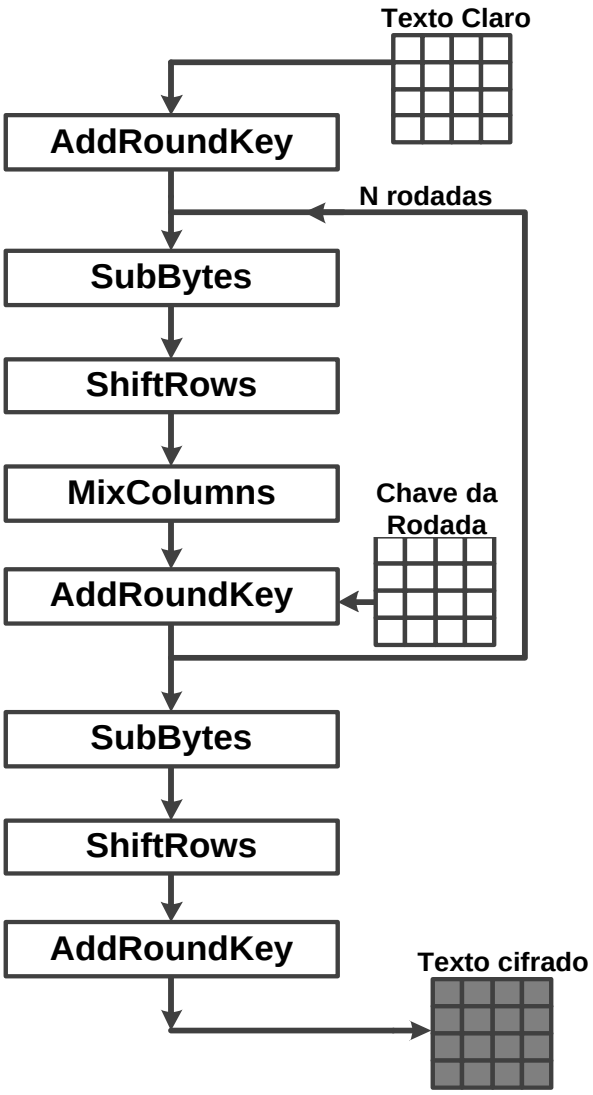


Figura 2.3: Processo de Cifragem do algoritmo AES

O texto será processado N vezes, sendo que N é o número de rodadas, que define também a segurança do algoritmo: 10, 12 ou 14 rodadas de acordo com o tamanho da chave 128, 192 ou 256, respectivamente. As chaves para cada rodada são calculadas na operação de Expansão da Chave e utilizadas nas operações *AddRoundKey* [29].

d) *SubBytes*

Consiste em uma substituição dos *bytes* do estado por outros contidos em uma caixa de substituição (S-Box). Essa caixa de substituição é resultado de operações polinomiais realizadas no campo de Galois. A matriz é chamada de S-Box e está representada na tabela 2.4.

Tabela 2.4: S-Box [29]

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

A cada rodada são realizadas 16 substituições a partir da matriz S-Box, que utiliza alguns critérios para sua construção em GF(2^8): o balanceamento dos dados, um alto grau algébrico na construção da tabela e a não-linearidade [35-38].

e) *ShiftRows*

A função *ShiftRows* realiza um deslocamento das células que estão à esquerda. O número de células deslocadas obedece ao número da linha que sofrerá a alteração. A figura 2.4 ilustra o bloco antes (2.4a) e depois (2.4b) da operação *ShiftRows* [33]:

1) A primeira linha não sofre alteração;
2) A segunda linha sofre apenas um deslocamento;
3) A terceira linha sofre dois deslocamentos; e
4) A quarta linha sofre três deslocamentos.

5F	7D	23	6A
27	B9	CE	D2
E6	2D	A9	F0
E8	A0	F3	50

(a)

5F	7D	23	6A
B9	CE	D2	27
A9	F0	E6	2D
50	E8	A0	F3

(b)

Figura 2.4: Execução da Função ShiftRows

f) *MixColmuns*

Esta função representa uma multiplicação de matrizes, onde os elementos do estado são considerados polinômios sobre GF(2^8). Para a realização da função *MixColumns*, utiliza-se outra matriz calculada a partir da matemática de Galois, que é formada por 4 linhas e 4 colunas. A tabela 2.5 apresenta os operandos, onde o bloco (a) é a palavra e o bloco (b) é a matriz utilizada nesta função. Esta função é utilizada como um dos mecanismos de difusão no algoritmo [29,33].

5F	7D	23	6A
27	B9	CE	D2
E6	2D	A9	F0
E8	A0	F3	50

(a)

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

(b)

Tabela 2.5: Palavra e Constante utilizada para Função MixColumns

O resultado da primeira célula da primeira coluna será: (5F · 02) ⊕ (27 · 03) ⊕ (E6 · 01) ⊕ (E8 · 01)
O resultado da segunda célula da primeira coluna será: (5F · 01) ⊕ (27 · 02) ⊕ (E6 · 03) ⊕ (E8 · 01)
O resultado da terceira célula da primeira coluna será: (5F · 01) ⊕ (27 · 01) ⊕ (E6 · 02) ⊕ (E8 · 03)
O resultado da quarta célula da primeira coluna será: (5F · 03) ⊕ (27 · 01) ⊕ (E6 · 01) ⊕ (E8 · 02)

Deste modo a primeira coluna do resultado estará completa.

g) *AddRoundKey*

A função de adição das chaves recebe como entradas a palavra e a chave da rodada e realiza uma operação XOR *byte* a *byte*. Isto quer dizer que se $s_{x,y}$ é um *byte* do estado S e $k_{x,y}$ um *byte* da chave, o *byte* $s'_{x,y}$ do novo estado S′ será igual a $s_{x,y} \oplus k_{x,y}$. A transformação inversa à *AddRoundKey* também consiste em um XOR entre o estado e a chave da rodada, ou seja, *AddRoundKey* é sua própria inversa.

A função *AddRoundKey* é exemplificada na figura 2.5. A matriz W representa a chave correspondente à rodada que está sendo calculada [33].

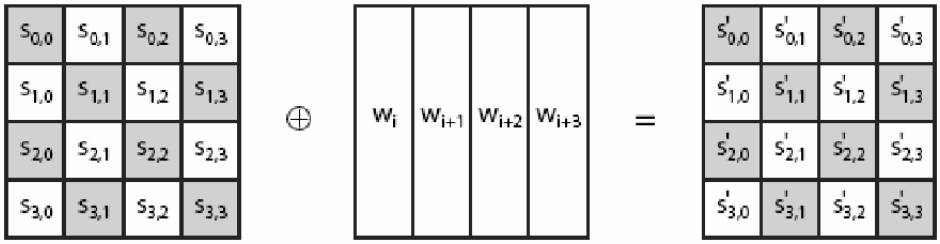


Figura 2.5: Função AddRoundKey[33]

h) Decifrando a Mensagem

Um texto cifrado é inserido no algoritmo e passa por N transformações, sendo N o número de rodadas. Todas as chaves para cada rodada precisam estar calculadas previamente para utilização nas operações *AddRoundKey*, como foi citado no item expansão das chaves. A decifragem da palavra segue o fluxo mostrado na figura 2.6 [4].

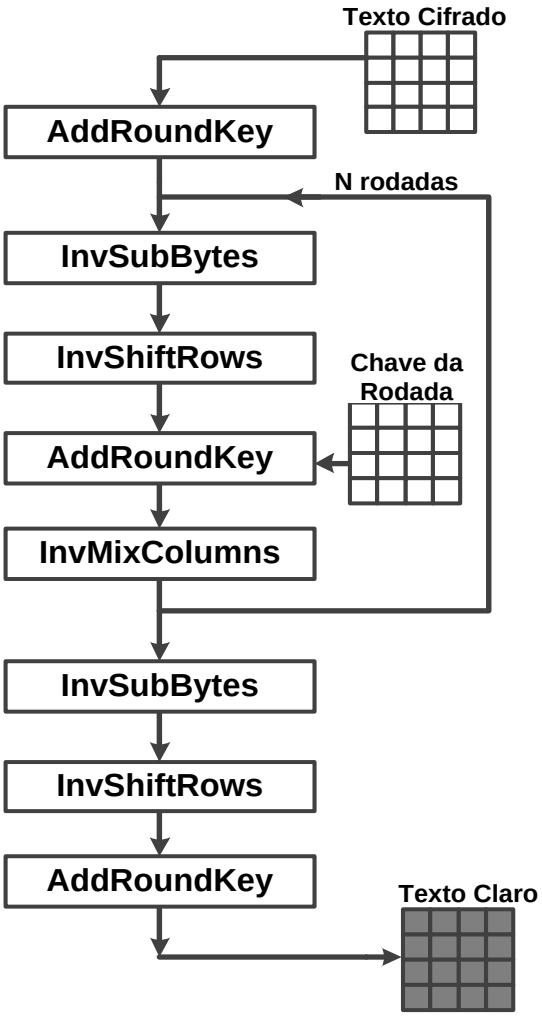


Figura 2.6: Processo de Decifragem do algoritmo AES

i) *InvSubBytes*

A função *InvSubBytes* realiza uma operação similar à função *SubBytes*. É importante ressaltar que a matriz utilizada para as substituições é diferente, ela é chamada *InvS-Box* e apresentada na tabela 2.6 [33].

Tabela 2.6: InvS-Box[33]

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

j) *InvShiftRows*

A função *InvShiftRows* é similar à função *ShiftRows*, realizando um deslocamento das células que estão à direita. O deslocamento das células tem como objetivo fazer com que o bloco fique embaralhado. As funções de decifragem visam reverter as mudanças realizadas pela cifragem para que mensagem volte estar legível ao usuário [33].

k) *InvMixColumns*

A função *InvMixColumns* realiza uma operação similar à função *MixColumns*, com a mudança na matriz utilizada para realizar a operação, que está representada na tabela 2.7.

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

Tabela 2.7: Constante para a Função InvMixColumns

l) Expansão das chaves para a decifragem no AES

A expansão das chaves é um processo que depende da chave da rodada anterior [28,29,33]. Isto significa que para gerar a chave K03, é necessário gerar todas as chaves que a

antecedem (K01 e K02). Esta característica não afeta a eficiência do processo de cifragem, pois as chaves são utilizadas, em cada rodada, na mesma ordem em que são geradas.

Como mencionado anteriormente, o processo de decifragem necessita das chaves na ordem inversa. Com isso, o processo de decifragem necessita de um tempo adicional antes do seu início para o cálculo de cada uma das chaves e de um local de armazenamento para as mesmas. A tabela 2.8 apresenta a ordem de utilização das chaves para um algoritmo de 128 bits (10 rodadas).

Tabela 2.8: Utilização das chaves na cifragem e decifragem do AES

Rodada	**1ª**	**2ª**	**3ª**	**4ª**	**5ª**	**6ª**	**7ª**	**8ª**	**9ª**	**10ª**
Cifragem	K01	K02	K03	K04	K05	K06	K07	K08	K09	K10
Decifragem	K10	K09	K08	K07	K06	K05	K04	K03	K02	K01

m) Diferença entre o Rijndael e o AES

O Rijndael foi desenvolvido para suportar tamanhos de chave e bloco variando entre 128, 160, 192, 224 e 256 bits. Os requisitos apresentados pela competição AES previam suporte somente blocos de 128 e chaves de 128, 192 e 256. Deste modo, o algoritmo Rijndael se adaptou aos requisitos solicitados na competição [28,29,33].

2.2.2 Algoritmo Twofish

O *Twofish* é um algoritmo que utiliza cifra de bloco de 128-bits e aceita chaves até o tamanho de 256 bits, obedecendo aos critérios da competição AES. Este algoritmo utiliza 16 rodadas de processamento em uma rede Feistel para qualquer tamanho de chave.

Uma rede Feistel contém uma função bijetora F formada por quatro S-Boxes, em grupos de 8 bits, dependentes de chaves; uma matriz sobre GF (2^8), de 4 por 4 *bytes* chamada *Maximum Distance Separable* (MDS); uma transformação *pseudo-Hadamard*; rotações de bits e um cálculo de chaves que foi cuidadosamente projetado e que possui algumas peculiaridades com relação aos outros finalistas do concurso realizado pelo NIST. Uma apresentação do funcionamento de uma rodada deste algoritmo é fornecida no Apêndice B.

a) Matemática utilizada no *Twofish*

Conforme apresentado na tabela 2.1, o *Twofish* é o único dos cinco finalistas que utiliza a rede de Feistel sem modificações. Sua estrutura básica de funcionamento pode ser vista na figura 2.7 [8].

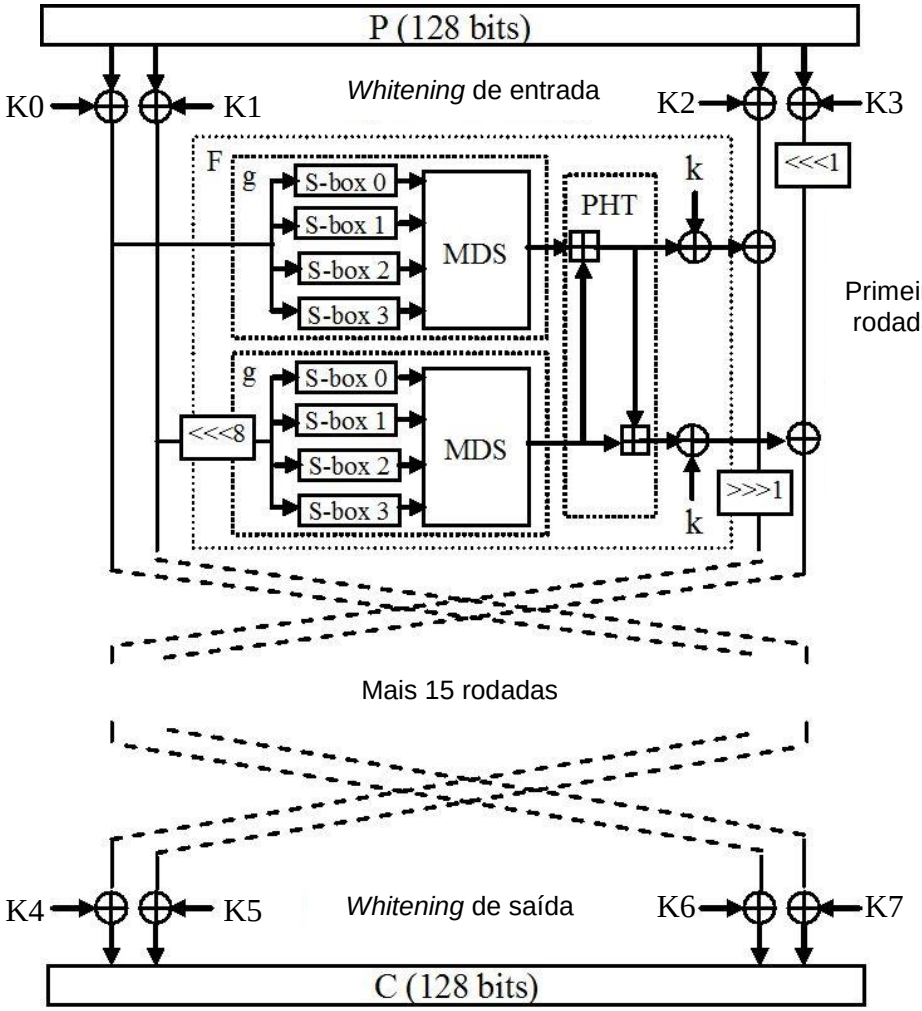


Figura 2.7: Processo de cifragem do algoritmo Twofish [8]

Diferentemente do algoritmo AES-Rijndael, o Twofish não utiliza funções inversas para o processo de decifragem. As funções utilizadas na cifragem são idênticas à decifragem, existe somente uma multiplexação em alguns caminhos de dados para que os processos funcionem. Este algoritmo recebe como entrada uma palavra de 128 bits, que é dividida em quatro blocos de 32 bits para que seja iniciado o processamento criptográfico. Em alguns pontos do algoritmo é realizada uma rotação de bits que tem como objetivo quebrar o alinhamento natural dos bytes devido às operações realizadas anteriormente.

b) Expansão das chaves

O algoritmo utiliza quarenta chaves durante as 16 rodadas, sendo que estas recebem a nomenclatura de K_0 até K_{39}. Estas sub-chaves são obtidas a partir da chave de entrada do sistema, que pode ser de 128, 192 ou 256 bits. O processamento das chaves é feito fora do processo apresentado na figura 2.7 e será apresentado posteriormente. São utilizadas duas chaves em cada rodada.

A utilização das chaves nos processos de cifragem e decifragem difere-se do AES-Rijndael, pois cada uma delas pode ser calculada conforme a demanda. Não existe dependência entre as chaves produzidas pelo processo de expansão, não havendo a necessidade de um dispositivo para armazená-las. As chaves podem ser calculadas durante o processamento das palavras, simultaneamente.

c) Whitening

O *Whitening* é uma técnica que têm como objetivo aumentar a segurança de uma cifra de bloco, através da combinação de partes da informação com partes da chave. Ela é utilizada nos processos inicial e final do Twofish. No processo inicial são utilizadas as chaves de K_0 a K_3 e no processo final são utilizadas as chaves de K_4 a K_7. Menores índices representam os 32 bits menos significativos da palavra. A figura 2.8 apresenta o primeiro tratamento da palavra com as chaves, chamado de *Whitening* de entrada. A figura 2.9 apresenta o último tratamento recebido dentro do processo de cifragem ou decifragem, chamado de *Whitening* de saída [7,8].

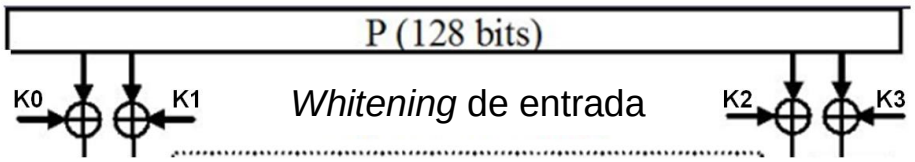


Figura 2.8: Processo Whitening de entrada [8]

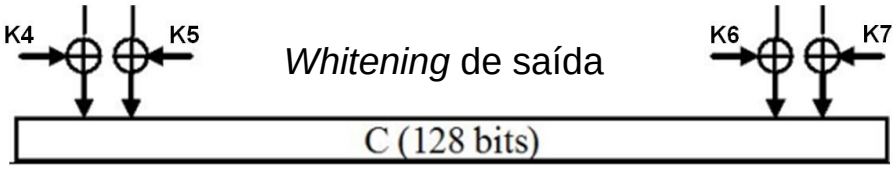


Figura 2.9: Processo Whitening de saída [8]

d) Rede de *Feistel*

A rede de Feistel é uma estrutura simétrica usada na construção de cifras de bloco. Tem seu nome em homenagem ao físico e criptografista alemão Horst Feistel, que foi um dos pioneiros nesta área. A figura 2.10 apresenta os sub-blocos das funções utilizadas nesta rede. Na parte superior de cada uma das linhas verticais são inseridos 32 bits da palavra de entrada ou da rodada anterior. Existem várias funções dentro desta rede que participam do processo de cifragem e/ou

decifragem do texto, sendo elas duas funções g, uma transformação *Pseudo-Hadamard* (PHT), adições das chaves, rotações e deslocamentos binários [6-8].

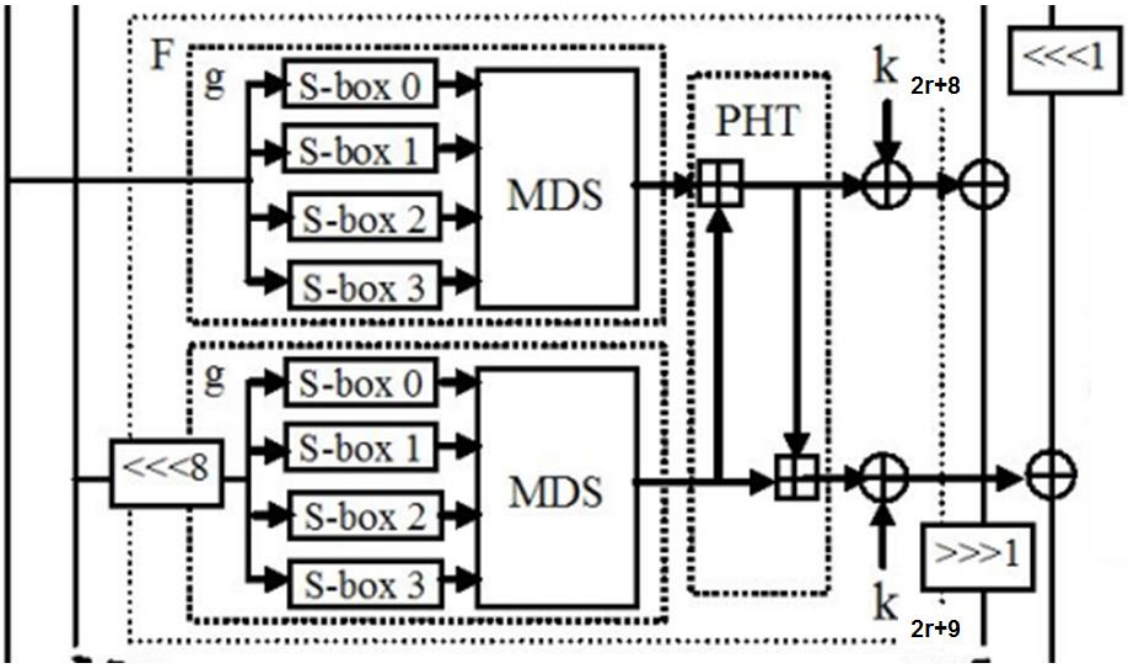


Figura 2.10: Rede de Feistel [8]

As funções g e h são os maiores blocos funcionais presentes no desenvolvimento do algoritmo Twofish. O funcionamento destas funções é similar, facilitando a construção das mesmas, e ocorre de maneira simultânea, porém com valores de entradas diferentes para cada uma delas. As funções g podem ser vistas na figura 2.10. As funções h são utilizadas no processo de expansão de chaves. Cada uma destas funções recebem como entrada um número de 32 bits, que é processado por quatro S-Boxes dependentes de chave e seu resultado é processado pela função *Maximum Distance Separable* (MDS). Após o processamento das funções g e h, as informações ainda são submetidas a uma transformação Pseudo-Hadamard e a um cálculo com as chaves da rodada.

e) Funções S-Box

A figura 2.11 apresenta a rede de funções S-Box utilizada para o processamento dos dados. Os 32 bits recebidos são divididos entre os quatro S-Boxes existentes.

Cada um dos quadrados apresentados na figura 2.11 representa uma função *q*, que pode apresentar dois tipos: q_0 e q_1. A rede de funções S-Box apresentada na figura 2.11 permite o processamento criptográfico em todos os níveis de chave: 128 bits (k=2), 192 (k=3) e 256 (k=4).

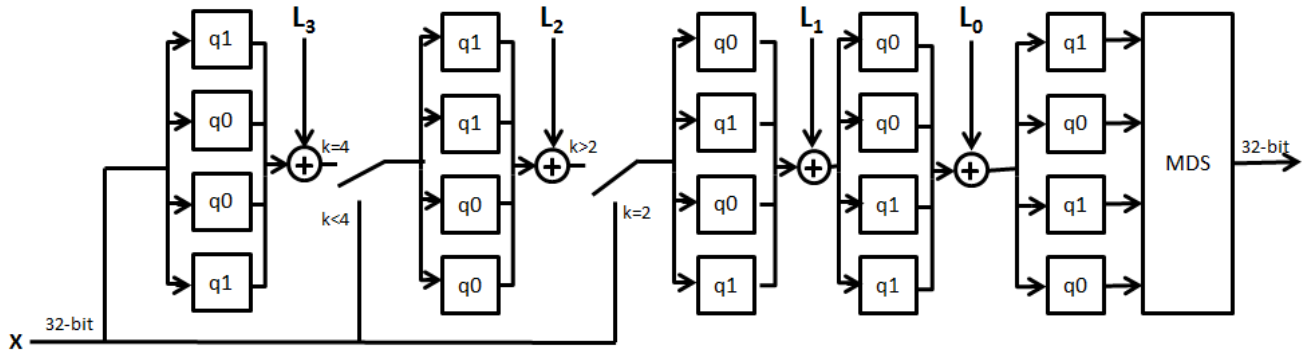


Figura 2.11: Funções S-Box

Cada S-box representa uma função que lê informações de 8 bits e fornece uma saída de mesma dimensão (8 bits). Para uma chave de 128 bits são utilizadas as sub-chaves S_0 e S_1 nas operações Ou-Exclusivo intermediárias. Cada S-Box possui uma estrutura de permutação entre os valores q_0 e q_1, conforme segue:

a_0; b_0 = [x/16]; x mod 16

$a_1 = a_0 \oplus b_0$

$b_1 = a_0 \oplus \text{ROR}(b_0; 1) \oplus 8a_0 \bmod 16$

a_2; $b_2 = t_0[a_1]; t_1[b_1]$

$a_3 = a_2 \oplus b_2$

$b_3 = a_2 \oplus \text{ROR}(b_2; 1) \oplus 8a_2 \bmod 16$

a_4; $b_4 = t_2[a_3]; t_3[b_3]$

$y = 16\, b_4 + a_4$

Cada uma destas estruturas funcionará de acordo com o tamanho da chave utilizada (128, 192 ou 256-bit). Utilizando uma chave de 128-bit, temos:

$y_0 = q_1[q_0[q_0[y_{2;0}] \oplus l_{1;0}] \oplus l_{0;0}]$

$y_1 = q_0[q_0[q_1[y_{2;1}] \oplus l_{1;1}] \oplus l_{0;1}]$

$y_2 = q_1[q_1[q_0[y_{2;2}] \oplus l_{1;2}] \oplus l_{0;2}]$

$y_3 = q_0[q_1[q_1[y_{2;3}] \oplus l_{1;3}] \oplus l_{0;3}]$

Existem algumas vantagens em se utilizar S-Boxes dependentes de chaves, pois a realização dos cálculos gera a mudança das tabelas e o intruso não conhecerá o seu conteúdo, pois são dinâmicas. Outra vantagem é que a complexidade desta estrutura está ligada ao tamanho da

chave, que irá gerar sinais de construção das tabelas durante o processamento das informações, podendo apresentar uma medida de contenção contra ataques do tipo *Side-Channel* [10,12].

f) Função *q*

A função *q* recebe oito bits de entrada e realiza transformações sobre essa informação, conforme apresentado na figura 2.12. Para o processamento destas informações é utilizada a tabela 2.9, de valores constantes, previamente estabelecidos e que obedece ao tipo de bloco utilizado: q_0 ou q_1.

Tabela 2.9: Parâmetros da função q [8]

Entradas		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
q_0	t_0	8	1	7	D	6	F	3	2	0	B	5	9	E	C	A	4
	t_1	E	C	B	8	1	2	3	5	F	4	A	6	7	0	9	D
	t_2	B	A	5	E	6	D	9	0	C	8	F	3	2	4	7	1
	t_3	D	7	F	4	1	2	6	E	9	B	3	0	8	5	C	A
q_1	t_0	2	8	B	D	F	7	6	E	3	1	9	4	0	A	C	5
	t_1	1	E	2	B	4	C	3	7	6	D	A	5	F	9	0	8
	t_2	4	C	7	5	1	6	9	A	0	E	D	8	2	B	3	F
	t_3	B	9	5	1	C	3	D	E	6	4	7	F	2	0	8	A

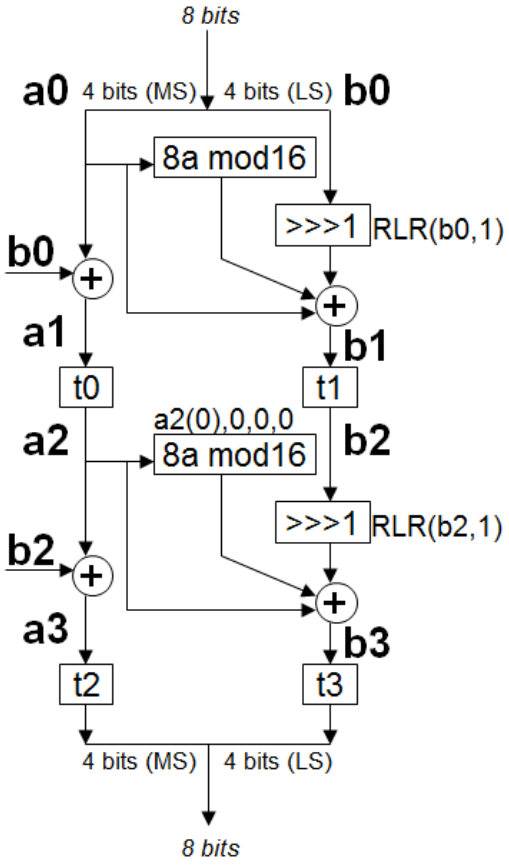


Figura 2.12: Função q

g) MDS – *Maximum Distance Separable*

O cálculo da função MDS é semelhante ao da função *MixColumns* no algoritmo AES. A figura 2.13 apresenta a matriz MDS e a figura 2.14 apresenta o cálculo a ser realizado. Esta função é utilizada como o principal mecanismo de difusão no algoritmo Twofish [8].

$$\text{MDS} = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix}$$

Figura 2.13: Matriz MDS

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \text{MDS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$Z = \sum_{i=0}^{3} z_i \cdot 2^{8i}$$

Figura 2.14: Cálculo da função MDS [8]

h) PHT – *Pseudo-Hadamard Transformation*

A transformação *Pseudo-Hadamard* utiliza os 32 bits de saída de cada uma das funções g (ou h) e realiza o cálculo apresentado na figura 2.15. Esta transformação visa aumentar a segurança através da combinação entre partes da informação [8].

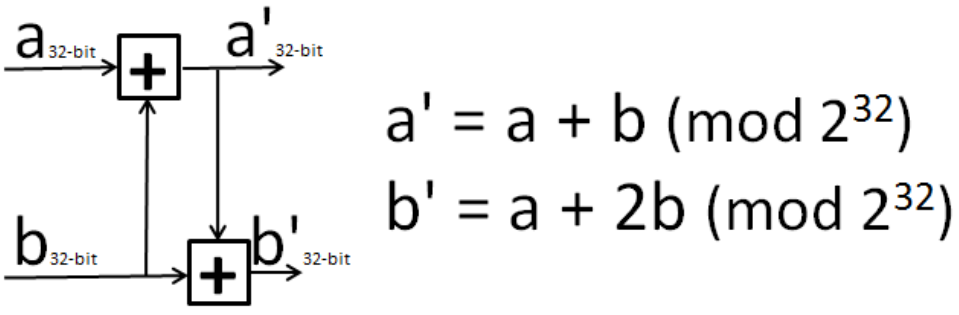
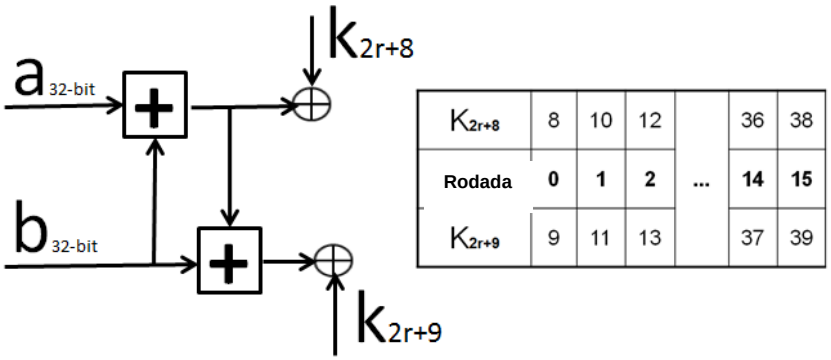


Figura 2.15: Cálculo da função PHT

j) Adição das chaves

A adição das chaves é realizada após a função PHT, de acordo com a figura 2.16. As chaves de 0 a 7 são utilizadas nas operações de *Whitening*. Por este motivo a adição das chaves

utiliza a regra *2r+8* e *2r+9* para verificar qual a chave necessária para a rodada em questão [8]. A variável *r* representa o número da rodada, que vai de 0 a 15.



K_{2r+8}	8	10	12	...	36	38
Rodada	**0**	**1**	**2**	**...**	**14**	**15**
K_{2r+9}	9	11	13		37	39

Figura 2.16: Adição das chaves

k) Expansão das chaves

A expansão das chaves utiliza as mesmas primitivas da rede de *Feistel*, com pequenas modificações, fornecendo dificuldades ao atacante que tiver como objetivo conseguir extrair o conteúdo das sub-chaves. Os S-Boxes utilizados também são dependentes de chaves e utilizam a chave fornecida pelo usuário para o cálculo das demais. Uma grande vantagem deste método de expansão das chaves é que ele pode ser utilizado de duas maneiras: as chaves podem ser calculadas antes do processamento da mensagem (pré-processamento) e necessitam de um dispositivo de armazenamento para as mesmas, ou podem ser calculadas durante o processamento da mensagem (*on-the-fly*). O esquemático do processo de expansão das chaves é apresentado na figura 2.17 [8, 30,31].

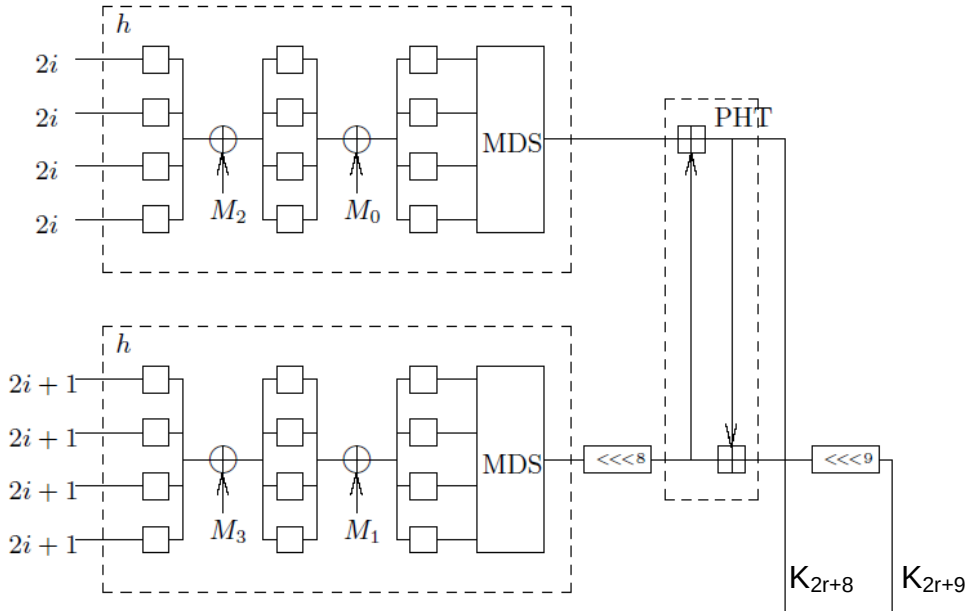


Figura 2.17: Expansão das chaves [5]

l) Processos de Cifragem e Decifragem

A diferença entre os processos de cifragem e decifragem pode ser visto na figura 2.18. Enquanto o algoritmo AES necessita de duas estruturas diferentes em *hardware* para realizar cada um dos processos, o algoritmo Twofish permite que o processo inverso seja realizado utilizando apenas um multiplexador, que controla qual dos processos está sendo executado [8,29,30,33]. A diferença entre os processos está nas rotações que ocorrem com os 64 bits mais significativos da palavra a ser processada.

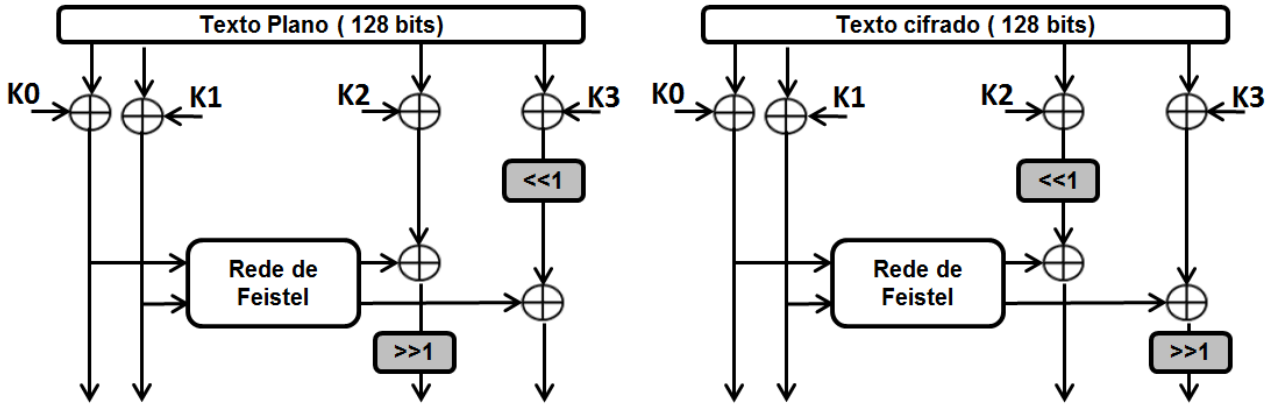


Figura 2.18: Processos de Cifragem e Decifragem

2.3 Utilização de algoritmos de criptografia

Algoritmos de criptografia podem ser aplicados e desenvolvidos tanto em *software* quanto em *hardware*. Uma vantagem da utilização de algoritmos de criptografia em software é que qualquer linguagem poderá ser utilizada de acordo com a velocidade computacional que se espera obter para o processamento das informações. Um ponto que merece atenção no desenvolvimento em *software* é que os dados e as chaves serão, na maioria das vezes, armazenados em uma memória compartilhada, criando assim um ponto de vulnerabilidade na aplicação. Deste modo, o acesso aos dados na memória (*data leakage* ou *memory dump*) apresenta uma vulnerabilidade que pode vir a diminuir a força do algoritmo que está sendo utilizado [39-41].

Sistemas operacionais de uso geral (como Windows e Linux) são desenvolvidos para permitir a execução de códigos de aplicações carregados em seus sistemas. A utilização destes equipamentos sem critérios rígidos de segurança pode permitir que ferramentas maliciosas sejam instaladas, comprometendo o sistema [42].

O instituto norte-americano SANS (*SysAdmin, Audit, Networking, and Security*) é um órgão privado especializado em segurança da informação, fundado em 1989 [43]. Este instituto realiza treinamentos e auditorias relacionados à área de segurança, além de publicar diversos

trabalhos científicos nesta área. Um dos mais recentes trabalhos do SANS relaciona-se a uma análise da implementação de algoritmos de criptografia em hardware e software, apresentando as vantagens e desvantagens de cada uma destas abordagens [9].

2.4 Padrões industriais

O vazamento de informações comerciais e industriais pode causar prejuízos para seus responsáveis. Com o objetivo de assegurar a confidencialidade e integridade das informações, foram desenvolvidos alguns padrões para a criação de dispositivos de segurança, chamados de HSM's (*Hardware Security Module* ou *Host Security Module*). Os HSMs são equivalentes a caixas-pretas (*black boxes*) que combinam *hardware* e *software* e podem ser conectados a um computador para fornecer funções criptográficas e segurança [44-47].

Os HSM's tem como propósito fornecer criptografia, armazenar e realizar o backup de arquivos e prover um ambiente seguro para a execução dos algoritmos de criptografia. Outras nomenclaturas podem ser utilizadas para estes dispositivos, são elas: PCSM (*Personal Computer Security Module*), SAM (*Secure Application Module*), SCD (*Secure Cryptographic Device*), SSCD (*Secure Signature Creation Device*), TRSM (*Tamper Resistant Security Module*), HCD (*Hardware Cryptographic Device*) e CM (*Cryptographic Module*) [48].

Aplicações em diversas áreas já utilizam os recursos dos HSM's, por exemplo: sistemas de pagamento de cartões; verificação de autenticação e integridade de mensagens; verificação online de PIN; *e-commerce; Home banking; Trusted Platform Modules* (TPM) e aplicações militares. Existem certificações para os diversos tipos de aplicação destes módulos, alguns exemplos são:

- ISO-13491-1:2007 *Banking – Secure Cryptographic Devices;*
- ISO-9564: *Banking – Personal Identification Number;*
- ISO-16609: *Banking – Requirements for Message Authentication;*
- ISO-11568: *Banking – Key Management;*
- *Protection Profile – Secure Signature Creation Device:*
    - BSI-PP-0004-2002T 03.04.2002 – Type1;
    - BSI-PP-0005-2002T 03.04.2002 – Type2;
    - BSI-PP-0006-2002T 03.04.2002 – Type3;
- Certificações:
    - FIPS 140-2; FIPS 140-3;
    - *Common Criteria* (CC);
    - PCI HSM - PCI SSC (*Payment Card Industry Security Standards Council*).

2.5 Vulnerabilidades

A necessidade de se desenvolver dispositivos que garantam a segurança dos dados decorre da existência de vulnerabilidades e falhas que possam causar o vazamento de informações confidenciais e restritas. Uma vulnerabilidade pode ser descrita como uma falha ou fraqueza em uma estrutura computacional que permite a um atacante invadir o sistema. Ela pode ser caracterizada como a combinação de três itens: uma suscetibilidade ou falha do sistema, o acesso do invasor à falha e a capacidade do invasor de explorar a falha [49,50].

Em 1999, foi criado em San Diego, Califórnia, Estados Unidos, um centro de recursos contra o roubo de identidades (ITRC - *Identity Theft Resource Center*) que tem como objetivo fornecer recursos sobre questões de consumo relacionadas com a segurança cibernética, violações de dados, mídia social, fraude, golpes e outras questões. O ITRC relatou que 436 vulnerabilidades encontradas até Junho de 2015 permitiram a exposição de mais de 135 milhões de registros de dados. Muitos ataques ocorreram pelo fato de senhas de acesso não terem sido trocadas com regularidade, atualizações de *software* não realizadas ou excesso de privilégios fornecidos para acesso ao sistema. Segundo o ITRC, a forma de armazenamento de dados que oferece maior dificuldade para ser acessada é aquela em que os dados estão encriptados através da utilização de sistemas desenvolvidos especificamente para este fim, como os módulos de segurança em *hardware* (HSMs) [42, 51-54].

O armazenamento de informações encriptadas fornece um alto de nível de segurança contra o vazamento de dados e o acesso ao mapa de memória. Entretanto, se o sistema operacional for comprometido, ou se a chave de encriptação não for devidamente armazenada, os dados encriptados estarão fornecendo uma falsa sensação de segurança. Para combater de maneira efetiva os contínuos avanços destas ameaças e as demandas comerciais e industriais da comunicação segura de dados, é necessário que sejam utilizados dispositivos ou métodos para proteger e acelerar o processamento das funções críticas de segurança [55].

2.6 Algoritmos utilizados pelo NIST

O Instituto Nacional de Padrões e Tecnologia (NIST) é um órgão governamental norte-americano que tem a tarefa de criar e manter os padrões legais em diversas áreas. Ele é o responsável pela elaboração de algumas normas americanas como a *Federal Information Processing Standards* (FIPS), que incluem o *Digital Signature Standard* e o *Advanced Encryption Standard* [4].

Em novembro de 2013, o NIST informou sobre o vazamento de documentos sigilosos que questionavam a integridade do processo de desenvolvimento dos padrões recomendados por ele. A partir deste momento houve uma grande preocupação por parte das empresas e usuários dos padrões de criptografia aprovados pelo NIST quanto à segurança de suas informações perante a Agência Nacional de Segurança (NSA) [56-58]. O NIST também sofreu críticas de criptólogos a respeito do algoritmo *Keccak* [59], vencedor do concurso para escolher um novo algoritmo de *hash*. De acordo com os críticos, o algoritmo havia sido enfraquecido pelo NIST durante sua padronização [60].

Com a ocorrência destes eventos e com diversas dúvidas acerca da segurança dos algoritmos aprovados pelo NIST, diversos grupos e empresas começaram a desenvolver modificações para os algoritmos existentes e padronizados, de modo que a segurança dos mesmos fosse aumentada. Um dos grupos que apresenta estes algoritmos modificados é o *TrueCrypt* [61,62].

O *TrueCrypt* disponibiliza um aplicativo de código aberto, usado para criar discos criptografados. Este aplicativo realiza a encriptação em tempo real de forma transparente, possibilitando dois níveis de proteção. Em 28 de maio de 2014, o website oficial do *TrueCrypt* anunciou que o projeto não seria mais mantido pelos seus desenvolvedores. Este aplicativo utiliza os seguintes algoritmos para a encriptação dos dados: AES, *Serpent* e *Twofish*. Adicionalmente, ele disponibiliza cinco diferentes combinações de algoritmos cascata: *AES-Twofish, AES-Twofish-Serpent, Serpent-AES, Serpent-Twofish-AES e Twofish-Serpent* [63].

Em 18 de Fevereiro de 2014, o NIST publicou um documento descrevendo algumas alterações propostas nos algoritmos e métodos que ele adota com o objetivo de reduzir a repercussão negativa do vazamento e possível quebra na segurança das informações [56-60,64].

2.7 Análise *Side-Channel*

Nesta seção serão apresentados alguns conceitos a respeito das análises do tipo *side-channel* e de como elas tem relação com os dispositivos de criptografia.

Uma análise do tipo *side-channel* é caracterizada pela obtenção de informações através de acesso ao ambiente do criptossistema. Ao invés da utilização de ataques de força bruta ou da utilização de outras vulnerabilidades, a obtenção de informações ocorre através de emissões do dispositivo. Existem diversas análises denominadas do tipo *side-channel*, são elas: acesso a informações de temporização, consumo de potência (*power consumption*), vazamento eletromagnético (*electromagnetic leaks*) ou até mesmo sonoro. As informações provenientes destas emissões podem ser utilizadas para a quebra do sistema [13].

Existem duas classes de análises do tipo *side-channel*: simples e diferencial. Em cada uma dessas classes, os atacantes podem usar uma série de propriedades do dispositivo, como: calor gerado, consumo de energia ou tempo de execução. Para os sistemas em que o atacante tem acesso ao *hardware*, calor e energia representam as fontes mais importantes de vazamentos de informação. Os ataques baseados em temporização são mais utilizados em sistemas multitarefa e multiprocessador onde o invasor é capaz de carregar seu próprio código ou usar interações entre aplicativos existentes para controlar o comportamento. Algumas destas análises necessitam de conhecimento técnico a respeito de como são realizadas as operações internas no *hardware*. Outras técnicas de ataque, como a análise diferencial de potência (*differential power analysis*) funcionam como uma análise de caixa-preta (*black-box*). Muitos métodos eficientes na área de análises do tipo *side-channel* são baseados em métodos estatísticos [13].

Para sistemas em rede, os ataques baseados no tempo são os mais utilizados. Sistemas que usam caches de memória são particularmente vulneráveis a ataques baseados em tempo devido à diferença significativa no desempenho de uma determinada seção de código com base em acessos acertados ou perdidos. Eles forçam uma leitura ou gravação mais lenta na memória principal. Semelhante aos sistemas com caches, microcontroladores sem circuitos de criptografia dedicados muitas vezes levam vários ciclos de *clock* para executar os cálculos necessários para criptografar ou descriptografar dados. Os sistemas de criptografia comumente usados empregam uma mistura de exponenciação e multiplicação, processando um bit de cada vez. Como a exponenciação pode ser alcançada em um sistema binário usando apenas operações de deslocamento, esta operação é mais rápida do que o algoritmo de multiplicação em série *shift-and-add* que geralmente é utilizado. Deste modo o atacante pode verificar o tempo de processamento de cada instrução. Caso o microcontrolador possua um multiplicador dedicado, isso consumirá mais energia do que a operação de exponenciação, consumindo mais corrente e gerando mais calor e emissões eletromagnéticas [10,13,58].

A seguir, é apresentada uma breve descrição de alguns destes ataques:

- *Timing Analysis*: são baseados na temporização entre as operações realizadas pelo HSM;
- *Power Consumption Analysis*: são baseados no consumo de potência do HSM durante as operações de encriptação:
  - SPA (*Single Power Analysis*): utilizam uma representação visual do consumo de potência;
  - DPA (*Differential Power Analysis*): utilizam uma análise estatística do consumo de potência;

- *Fault Analysis*: são baseados na investigação das cifras e na extração das chaves através da geração de falhas.

A análise de potência simples funciona para circuitos integrados de baixa integração, onde há pouca atividade no chip para mascarar o comportamento do circuito. Por esta razão, a análise de energia simples geralmente não é muito útil, embora tenha servido para descobrir as chaves de criptografia processadas por microcontroladores *low-end*. Análise de potência diferencial (DPA) é um método estatístico que provou ser eficaz em descobrir informações sensíveis sobre os circuitos alvo, mesmo quando outras portas estão ativamente envolvidas em comutação. O DPA permite que o atacante desenvolva uma hipótese sobre o comportamento ou estado do circuito de destino (a suposição de uma parte da chave de criptografia completa, por exemplo). Se a suposição estiver correta, as emissões associadas à atividade elétrica dentro do chip serão correlacionadas. Se não, a atividade não será correlacionada. Em um grande número de suposições e medições, os resultados correlacionados se separarão, fornecendo ao invasor pistas sobre o valor, que é a chave para a quebra do sistema. À medida que mais medições são tomadas, qualquer ruído não correlacionado é reduzido [30,57,65].

As análises do tipo *side-channel* são baseadas na relação entre o sinal emitido e o dado secreto [65,66]. Medidas de contenção para os ataques que utilizam estas técnicas baseiam-se em duas linhas principais:

- Eliminar ou reduzir a emissão destas informações;
- Eliminar a relação entre o sinal emitido e o dado secreto, através da injeção de algum tipo de sinal aleatório (ruído) que prejudique a comparação entre o dado processado e o sinal emitido.

A primeira alternativa de contenção envolve características ligadas à fabricação dos dispositivos, o que torna a proposição de uma solução mais complexa. Com relação à segunda alternativa, a geração de um sinal aleatório pode dificultar a realização de uma análise do tipo *side-channel*, pelo fato de que irá descaracterizar o sinal, evitando o rastreamento por parte do invasor, e servindo como uma medida de proteção ao dispositivo.

Este tipo de análise não é recente e sua utilização como técnica de ataque a dispositivos eletrônicos é anterior à competição AES. A partir destas informações, serão apresentadas algumas informações a respeito do ataque do tipo *side-channel* realizado por Paul Kocher, em 1996, onde são apresentadas as vulnerabilidades do algoritmo DES [12,34]. Estes artigos confirmam que durante a realização da competição AES essa vulnerabilidade nos dispositivos já era conhecida, mas não foi utilizada para a verificação dos níveis de segurança dos competidores. Após a apresentação

de alguns resultados do DES será feita a apresentação do mesmo tipo de análise realizada em seu sucessor, o AES.

O algoritmo DES utiliza uma rede de substituição e permutação com 16 rodadas para a cifragem do texto. A figura 2.19 apresenta o esquemático do funcionamento do DES. É importante ressaltar que, dentro deste processo de encriptação, a sub-palavra R_{15} é encontrada na 15ª rodada. Na 16ª rodada esta palavra é utilizada novamente (L_{16}=R_{15}) e representa a parte mais signifcativa do texto final. Deste modo são obtidos os dados para a análise, que são: as informações emitidas pelo dispositivo e o texto final, obtido através de um acesso ao barramento. Com estas informações é possível realizar um ataque para que se obtenha a sub-chave da 16ª rodada.

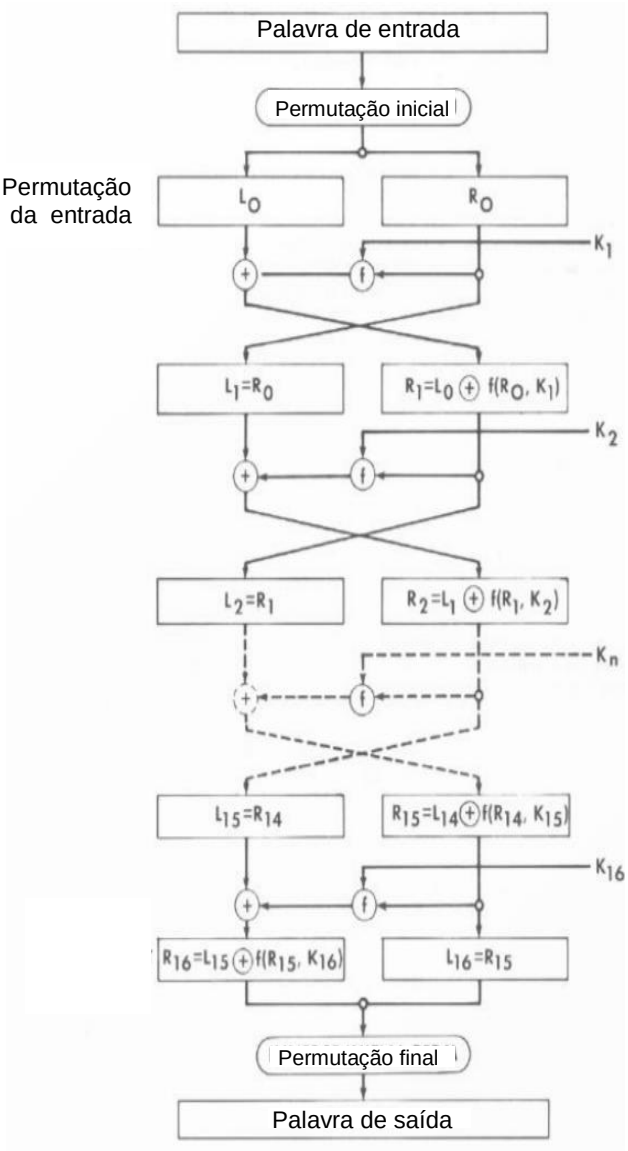


Figura 2.19: Esquemático do algoritmo DES [34]

A figura 2.20 apresenta os sinais externos do processamento criptográfico oriundos do DES. É possível verificar 16 formas de onda semelhantes (ciclos), que representam as 16 rodadas do algoritmo. Para a realização da análise do tipo *side-channel* é utilizado um sinal de referência e diversas amostras. A chave da rodada é descoberta quando é encontrada uma alta correlação entre a amostra e o sinal.

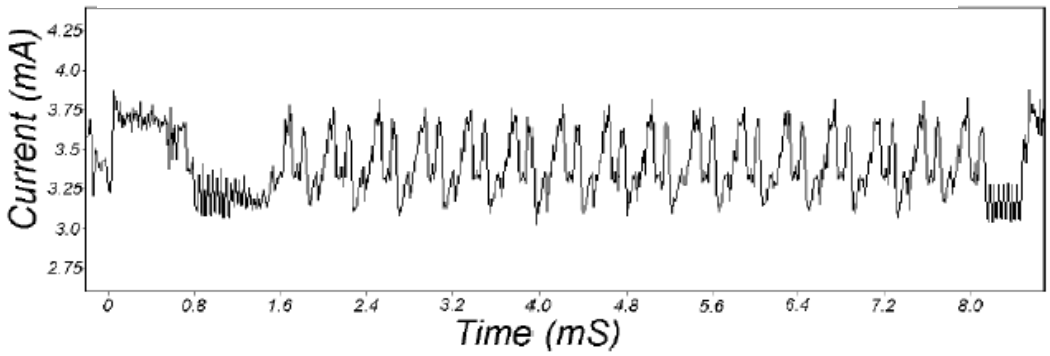


Figura 2.20: Análise side-channel do algoritmo DES [34]

Após uma breve apresentação do funcionamento da análise do tipo *side-channel* em um dispositivo de criptografia que utiliza o DES, será apresentado o mesmo tipo de análise realizada com um dispositivo AES.

Em 2011, Paul Kocher publicou um artigo com as vulnerabilidades do AES, apresentando a fragilidade de alguns algoritmos que utilizam a rede de permutação e substituição quando submetidos à análise do tipo *side*-channel. De maneira similar ao que foi realizado na análise do DES, é possível verificar na figura 2.21 dez formas de onda semelhantes (picos), que representam as dez rodadas do algoritmo utilizando chave de 128 bits. A figura 2.21 apresenta os sinais externos do processamento criptográfico oriundos do AES e o momento em que uma alta correlação é encontrada entre o sinal emitido e a amostra gerada [12].

Diversas empresas e grupos de pesquisa começaram a desenvolver mecanismos de verificação para esse tipo de vulnerabilidade a partir da confirmação de que ela está presente na construção clássica do AES. Um equipamento que merece destaque para a realização de análises do tipo *side-channel* é o *ChipWhisperer*, da fabricante NewAE [67].

ChipWhisperer [68] é a primeira ferramenta (com licença GPL) para verificação e validação de segurança em *hardware* embarcado, fornecendo informações de *side-channel power analysis* e *glitching*. Várias publicações acadêmicas e comerciais analisaram a eficácia desta ferramenta. O objetivo deste equipamento é fornecer aos projetistas de dispositivos de segurança a capacidade de realizar uma análise do tipo *side*-channel, entender as vulnerabilidades do dispositivo e projetar medidas de contenção [10,67,68].

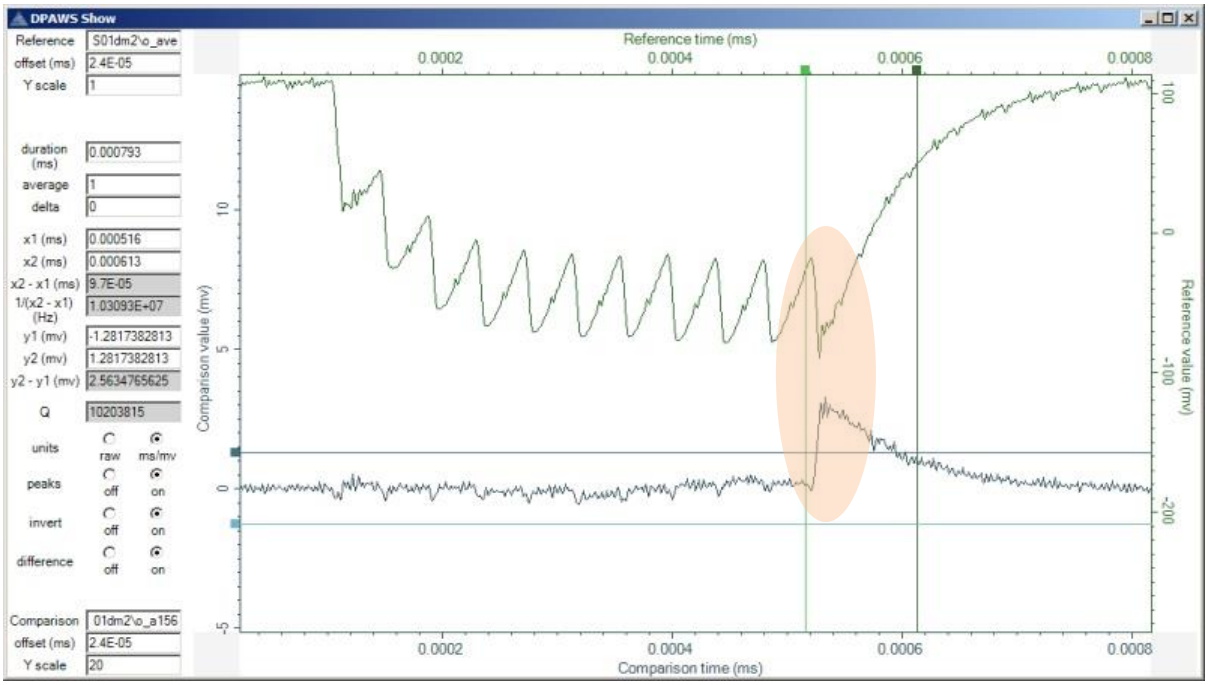


Figura 2.21: Análise side-channel do algoritmo AES [12]

O *ChipWhisperer* recebeu o segundo prêmio na competição *Hackaday Prize* 2014 e apresentou a quebra de algoritmos seguros como AES-128 e o AES-256. Na página *web* do projeto é possível assistir a um exemplo de aplicação e utilização deste dispositivo quebrando um dispositivo AES-128 em 120 segundos. Ele está à venda somente para a utilização com fins acadêmicos [10, 67-69].

Apesar da existência desta vulnerabilidade e de sua ampla divulgação, ainda existem muitos dispositivos comerciais que utilizam os padrões de criptografia DES e AES sem as devidas proteções ou medidas de contenção. Como foi apresentado anteriormente, a geração de ruídos pode prejudicar a caracterização da informação que está sendo processada, servindo como uma medida de contenção. Na próxima seção será apresentada uma breve revisão de alguns métodos de geração de números aleatórios em *hardware*, com o foco em criptografia. Estes números aleatórios são utilizados como sementes na geração dos ruídos [65-66].

2.7.1 Geração de ruído

Números aleatórios são utilizados em diversos sistemas presentes no nosso cotidiano. Eles fazem parte desde simulações de fenômenos físicos, tomada de decisões, geração de ruídos, jogos e entretenimento [70].

Devido a sua grande utilização torna-se necessário o estudo do comportamento desses números, bem como métodos eficientes e seguros para sua geração. Qualquer simulação em computador de um sistema físico que envolve aleatoriedade deve incluir um método para geração de sequências de números aleatórios. Esses números aleatórios devem satisfazer as propriedades dos processos físicos que eles estão simulando [71].

Geralmente números aleatórios não são descritos em representação decimal, mas em representação binária. Dessa forma um gerador aleatório para esse tipo de saída produz uma sequência de *bits* aleatórios, tal gerador é chamado de *Random Bit Generator* (RBG) [72]. *Bits* aleatórios podem ser gerados tanto por meio de *hardware* como por *software*. São exemplos de geradores baseados em *hardware* [72]:

- O tempo decorrido entre a emissão de partículas durante um declínio radioativo;
- O ruído térmico de um diodo semicondutor ou um resistor;
- A quantidade de carga em um capacitor dentro de um intervalo fixo;
- A entrada de vídeo de uma câmera;
- A entrada de som de um microfone;
- A turbulência dentro de um disco rígido selado.

Os geradores baseados em *hardware* são susceptíveis a eventos e fatores externos e ao mau funcionamento. Além disso, devem ser protegidos de um adversário ou intruso para que este não possa observar seu funcionamento ou interferir no mesmo. Alguns geradores necessitam ser construídos do lado de fora do sistema que processa a informação (externamente), o que dificulta ainda mais essa restrição de acesso. Já geradores baseados em *software* podem ser obtidos a partir de:

- Captura da hora do sistema;
- Estatísticas da rede;
- A frequência da digitação ou movimentação do *mouse* pelo usuário;
- Sobrecarga do sistema.

Algumas desvantagens estão presentes no uso desse tipo de gerador. Por exemplo, de acordo com a plataforma e sistema utilizados o comportamento desses geradores pode ser alterado. Fontes de bits verdadeiramente aleatórios têm como origem meios físicos que, em sua maioria, são lentos e ineficientes. Ainda, pode ser impraticável armazenar com segurança tais fontes de dados e transmitir uma grande quantidade de bits [72].

Um modo de amenizar os problemas encontrados em geradores de *bits* aleatórios seria utilizar os geradores pseudoaleatórios (PRBG – *pseudorandom bit generator*). Geradores pseudoaleatórios utilizam algoritmos determinísticos. Desta forma, para uma mesma semente, o gerador sempre irá gerar a mesma sequência de *bits*. Uma forma de manter um nível de segurança confortável é utilizar uma semente k suficientemente grande, de modo a tornar impraticável que o adversário compute os 2^k possíveis elementos [72].

As características determinísticas dos números pseudoaleatórios podem ser verificadas por um método de geração que utiliza uma função de recorrência dada por $x_n = (5x_{n-1} + 1)\ mod\ 16$ [73], que fixa como semente x_0 = 5 e obtém os 32 primeiros números gerados, que são: 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, **10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5**.

É possível observar que após o 16º valor, a sequência começa a repetir, assim, o comprimento de ciclo do gerador utilizado para essa sequência é igual a 16 valores. Dessa forma, conhecendo a função de geração e a semente, é possível reproduzir a sequência gerada. Existem geradores que não repetem a parte inicial de sua sequência, que é chamada de cauda, como mostrado na figura 2.22, quando isso ocorre o comprimento do seu período é dado pela soma do comprimento de seu ciclo e o comprimento da cauda [73].

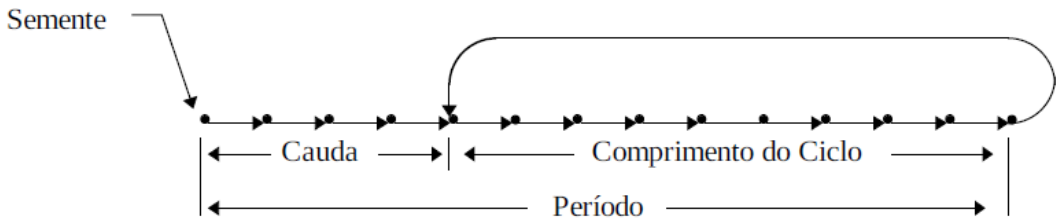


Figura 2.22 : Ciclo, cauda e período de gerador de números pseudoaleatórios [73].

Um gerador de números aleatórios deve possuir algumas propriedades [73], são elas:

- **Deve ser computacionalmente eficiente:** o tempo de processamento deve ser mínimo, para que não influencie na complexidade do algoritmo que o utilize.
- **O período deve ser muito longo:** a existência de um período curto infere em uma ciclagem rápida da sequência de números, limitando o período utilizável em uma rodada de simulação.
- **Os valores devem ser independentes e uniformemente distribuídos:** a dependência entre os valores gerados em uma sequência deve ser mínima.

2.7.2 Geradores de números pseudoaleatórios

Nessa seção serão apresentados alguns dos principais algoritmos para geração de números pseudoaleatórios [72,73].

2.7.2.1 *Linear Congruential Generators* (LCG)

São os algoritmos mais conhecidos e utilizados para a geração de números aleatórios [74]. O algoritmo foi desenvolvido pelo Prof. D. H. Lehmer em 1951 para utilização no ENIAC [73]. É o algoritmo utilizado para geração de números pseudoaleatórios na linguagem de programação ANSI-C [75]. Segundo Lehmer, "os restos de sucessivas potências de um número possuíam boas características de aleatoriedade" [73]. Originalmente utilizava a Equação 1 como base para gerar o n-ésimo número de uma sequência.

$$x_n = a^n \ mod \ m \tag{1}$$

O parâmetro a é chamado de multiplicador, e o parâmetro m chamado de módulo. Lehmer escolheu como valores para esses parâmetros, os valores $a = 23$ e $m = 10^8 + 1$ [73].

Outra forma de encontrar números pseudoaleatórios por meio do algoritmo LCG é por meio de versões atuais e generalizadas da fórmula original de Lehmer, como mostrado na Equação 2. Os valores de x_n são números inteiros pertencentes ao intervalo entre 0 e $m - 1$ [73].

$$x_n = a x_{n-1} \ mod \ m \tag{2}$$

Na Tabela 2.10, é mostrado um exemplo de sequências de um gerador com $a = 13$, $b = 0$, $m = 2^6$ e a semente assumindo os múltiplos valores $x_0 = 1, 2, 3, 4$. Fica claro que o valor da semente está intimamente relacionado com o período que cada sequência irá possuir.

Tabela 2.10: Sequências geradas com múltiplas sementes [73].

i	x_i	x_i	x_i	x_i
0	1	2	3	4
1	13	26	39	52
2	41	18	59	36
3	21	42	63	20
4	17	34	51	4
5	29	58	23	
6	57	50	43	
7	37	10	47	
8	33	2	35	
9	45		7	
10	9		27	
11	53		31	
12	49		19	
13	61		55	
14	25		11	
15	5		15	
16	1		3	

a) Relações Entre os Componentes da Recorrência

A relação que os componentes das equações do LCG exercem sobre o valor resultante pode ser vista nos itens a seguir [73]:

- O valor de m deve ser grande, pois todos os valores gerados estarão contidos entre 0 e $m-1$;
- É desejável que m seja uma potência de 2, dessa forma sua computação é dada de modo mais eficiente;
- Somente é possível configurar o gerador com o período máximo se b for diferente de zero e
    - m e b forem primos entre si;
    - Todo primo que é um fator de m, é também um fator de $a-1$;
    - $a-1$ é múltiplo de 4, se m é também múltiplo de 4.
- Se $b=0$, e m é uma potência de 2, o maior período possível será $P=\frac{m}{4}$, considerando que: x_0 seja ímpar e o valor de a seja dado por $a=8k+3$ ou $a=8k+5$ para algum $k=0,1,2,\cdots$

b) Vantagens e Desvantagens

O LCG possui como vantagem o fato de ser de fácil desenvolvimento. Além disso, passa em testes estatísticos, ou seja, nenhum algoritmo de tempo polinomial é capaz de distinguir a saída do

gerador de uma sequência verdadeiramente aleatória de mesmo tamanho; acertando com uma probabilidade significativamente maior que $\frac{1}{2}$ [72].

Porém, sua saída é previsível, pois mesmo sem conhecer os parâmetros a, b e m é possível calcular o próximo número de sua sequência tendo acesso apenas a uma parcela de sua saída. Essa previsibilidade o torna inseguro para fins criptográficos [72].

2.7.2.2 Mersenne Twister

Proposto por Makoto Matsumoto e Takuji Nishimura, esse gerador possui a grande vantagem de ter um período extremamente longo ($p = 2^{19.937} - 1$). Seu nome é dado devido ao expoente de seu período ser um número de Mersenne [75]. Foi desenvolvido no final da década de 90 com o objetivo de ser utilizado em simulações de Monte Carlo. Seu desenvolvimento tem como base fundamentos teóricos de matemática discreta e da álgebra abstrata [76].

Existem duas versões do Mersenne Twister, a primeira, MT19937, é um gerador de palavras de 32 *bits*, já a segunda, MT19937-64, é um gerador de palavras de 64 *bits*. Sua fórmula de recorrência linear é mostrada na Equação 3 [76].

$$x_{k+n} = x_{k+m} \oplus (x_k^u \,|x_{k+1}^l)A, (k = 0, 1, \cdots) \tag{3}$$

Onde:

- $x_0, x_1, \cdots, x_{n-1}$ são as sementes iniciais.
- Dada a semente inicial, a Equação 3 gera x_n, com $k = 0$.
- Os valores de $x_{n+1}, x_{n+2}, \cdots$ são gerados fazendo-se $k = 1, 2, \cdots$.
- Vetor $x = (x_{w-1}, x_{w-2}, \cdots, x_0)$ é uma palavra de w *bits*.
- $\oplus$ é o operador ou exclusivo *bit* a *bit.*
- $|$ é a operação de concatenação.
- x_k^u corresponde aos $w - r$ *bits* superiores de x_k.
- x_{k+1}^l corresponde aos $w - r$ *bits* inferiores de $x_k + 1$.
- Assim se $x = (x_{w-1}, x_{w-2}, \cdots, x_0)$, $x^u = (x_{w-1}, x_{w-2}, \cdots, x_r)$ e $x^l = (x_{r-1}, x_{r-2}, \cdots, x_0)$.
- $(x_k^u \,|x_{k+1}^l)$ é o vetor obtido pela concatenação dos $w - r$ *bits* superiores de x_k com os r *bits* inferiores de x_{k+1}.
- Multiplica-se a matriz A pelo vetor concatenado e executa-se a operação $\oplus$ com x_{k+m}, gerando o próximo vetor x_{k+n}.

A matriz A é escolhida de tal forma que a multiplicação de um vetor por ela seja feita rapidamente [75]. A matriz A possui a seguinte forma:

$$A = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ & \vdots & \vdots & & & \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & \cdots & a_1 & a_0 \end{pmatrix}$$

Utilizando essa matriz, o cálculo de xA pode ser realizado apenas com operações *bit* a *bit*, como mostrado na Equação 4:

$$xA = \begin{cases} deslocamento_{direita(x)} & se\ x_0 = 0 \\ deslocamento_{direita(x)} \oplus a & se\ x_0 = 1 \end{cases} \quad (4)$$

onde $a = (a_{w-1}, a_{w-2}, \cdots, a_0)$. Os valores x_k^u e x_{k+1}^l também podem ser calculados com operações *bit* a *bit*, dessa forma toda a equação 4 pode ser calculada através de operações binárias básicas, como deslocamentos, e, ou e ou exclusivo.

a) Vantagens e Desvantagens

Algumas vantagens e desvantagens do algoritmo Mersenne Twister são mostradas a seguir. São vantagens do gerador [76]:

- Sem restrições quanto à inicialização do vetor de sementes, somente é necessário que esse seja não nulo;
- Utiliza menos memória que a versão anterior de seu algoritmo (GFSR)
- A sequência gerada é estatisticamente equidistribuída;
- O gerador passou por uma bateria rigorosa de testes estatísticos.

São desvantagens do gerador [76]:

- Seleção de sementes é crítica e influencia profundamente a aleatoriedade da sequência;
- O período real da sequência é muito menor que o teórico;
- Requer um grande espaço de memória se vários geradores forem utilizados simultaneamente;

Além dessas desvantagens, [77] complementa:

- O gerador não é ideal para usos criptográficos. Conhecida suas saídas, é fácil prever o próximo valor da sequência;
- Para amenizar esse problema, os criadores do Mersenne Twister recomendam que as saídas desse gerador passem por uma função *hash*, tal como a SHA-1. Essa alteração dá origem ao algoritmo chamado de *"CryptMT"*.

2.7.2.3 *Linear Feedback Shift Register* (LFSR)

Consiste em um registrador de deslocamento capaz de gerar uma sequência de $2^n - 1$ valores pseudoaleatórios. Seu funcionamento é baseado em um polinômio primitivo de grau n como mostrado na Equação 5 [78].

$$p(x) = 1 + C_1x + C_2x^2 + C_3x^3 + \cdots + x^n \quad (5)$$

onde:

- Os termos 1 e x^n estão sempre presentes;
- Os coeficientes $C_1, C_2, C_3, \cdots, C_{n-1}$ assumem valor 0 ou 1;
- Os polinômios devem ser irredutíveis, ou seja, não podem ser divisíveis por outros de maior ordem.

A tabela 2.11 apresenta alguns exemplos de polinômios primitivos com grau variando de 1 a 5 [78].

n	Polinômios primitivos
1	$1+x$
2	$1+x+x^2$
3	$1+x^2+x^3$, $1+x+x^3$
4	$1+x+x^4$, $1+x^3+x^4$
5	$1+x^2+x^5$, $1+x+x^2+x^3+x^5$, $1+x^3+x^5$, $1+x+x^3+x^4+x^5$, $1+x^2+x^3+x^4+x^5$, $1+x+x^2+x^4+x^5$

Tabela 2.11: Exemplo de polinômios primitivos [78].

Sua montagem é realizada por meio de n *flip-flops*. Entre a saída de um registrador (*flip-flop*) e a entrada de uma porta Ou-Exclusivo existe uma ligação, chamada de *tap*. O período da sequência de um LFSR depende tanto do comprimento do registrador de deslocamento, ou seja, a

quantidade de *flip-flops*, como também do número e da posição dos *taps* [78]. Uma tabela com os polinômios primitivos mais utilizados pode ser encontrada em [74].

A figura 2.23 representa um LFSR genérico com configuração de Fibonacci. Existem dois tipos de possíveis montagens para geradores LFSR, são elas: Fibonacci e Galois [78].

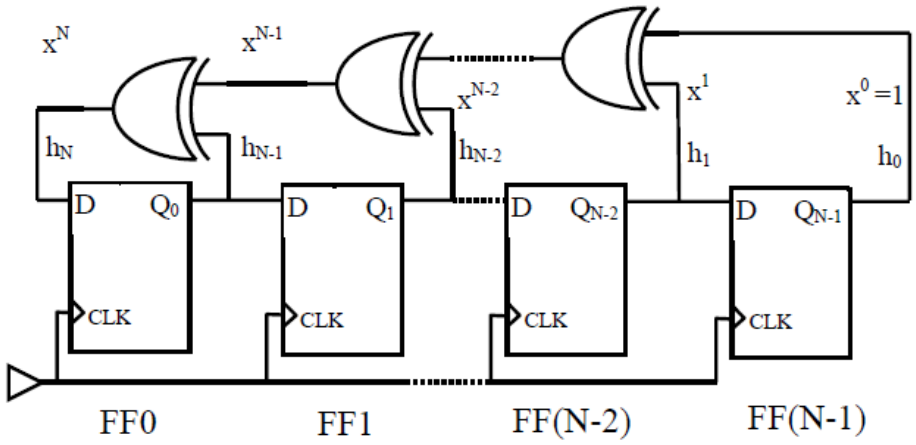


Figura 2.23: Gerador LFSR genérico com configuração Fibonacci [64].

a) Configuração de Fibonacci

Na configuração de Fibonacci, algumas saídas de registradores do LFSR são ligadas em uma porta Ou-Exclusivo e em seguida realimentadas na entrada do primeiro registro. Tal realimentação ocorre na direção oposta à direção de deslocamento e os *taps* são incrementados na direção do deslocamento.

Em um LFSR com poucos *taps*, essa configuração geralmente possui um *clock* mais rápido do que a de Galois. Porém, com o aumento de *taps*, seu desempenho se degrada [78]. Na figura 2.24 é possível observar o esquemático de um gerador em configuração Fibonnaci com o polinômio primitivo $f(x) = x^3 + x + 1$.

Após inicializar os valores do registrador de deslocamentos em 111 o sinal de *clock* é ativado e assim uma sequência pseudoaleatória de 0's e 1's flui da saída do *flip-flop* Q2 [78].

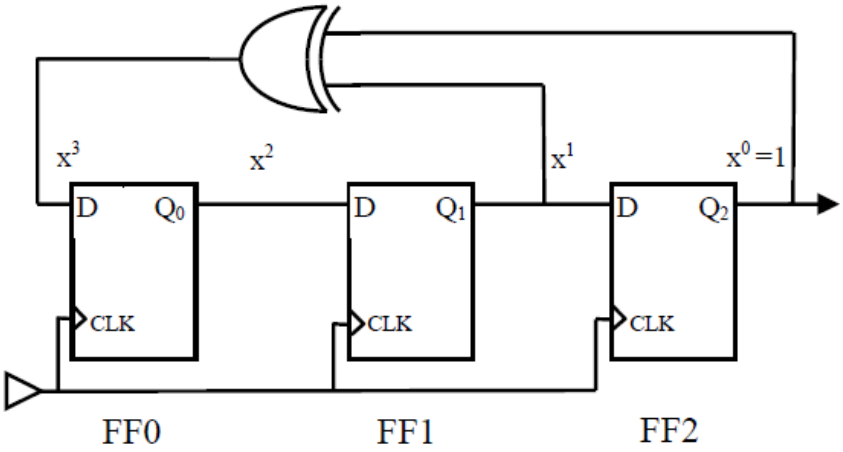


Figura 2.24: Gerador LFSR com configuração de Fibonacci [78].

Na tabela 2.12 são mostrados os valores resultantes da saída do LFSR mostrado na figura 2.24, é importante observar que somente são gerados sete valores aleatórios antes que a sequência repita, ou seja, seu período é igual a $2^n - 1$.

Clk	Q2	Q1	Q0
0	1	1	1
1	1	1	0
2	1	0	0
3	0	0	1
4	0	1	0
5	1	0	1
6	0	1	1
7	1	1	1

Tabela 2.12: Sequência pseudoaleatória gerada 1 [78].

b) Configuração de Galois

Na configuração de Galois, são posicionadas portas Ou-Exclusivo entre os registros, ao contrário da configuração de Fibonacci, onde tais portas eram conectadas as saídas desses registros. A realimentação ocorre na mesma direção dos registradores, e os *taps* são decrementados na direção de deslocamento [78].

Na figura 2.25 é possível observar um LFSR com mesma configuração do que o mostrado na figura 2.24, porém em configuração de Galois. A tabela 2.13 mostra qual a sequência gerada pelo LFSR mostrado na figura 2.25.

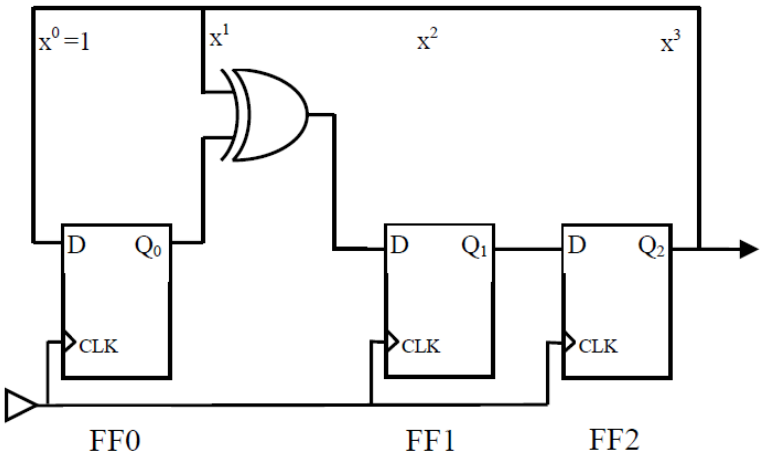


Figura 2.25: Contador LFSR com configuração de Galois [78].

Clk	Q2	Q1	Q0
0	1	1	1
1	1	0	1
2	0	0	1
3	0	1	0
4	1	0	0
5	0	1	1
6	1	1	0
7	1	1	1

Tabela 2.13: Sequência pseudoaleatória gerada 2 [78].

c) Vantagens e Desvantagens

O gerador de números pseudoaleatórios LFSR possui como uma grande vantagem a facilidade de entendimento e desenvolvimento de seu modelo, que é naturalmente em *hardware*.

Sua estrutura minimiza a complexidade de roteamento e assim reduz a quantidade de lógica requerida. Devido ao uso de componentes lógicos em sua construção, possui uma velocidade aprimorada na geração de resultados. Apresenta outra utilidade além da geração de números pseudoaleatórios, pois o LFSR pode ser utilizado para conversão de dados serial para paralelo e vice-versa [78].

LFSRs individuais geram uma saída que é altamente previsível. Caso se conheça $2m$ bits de saída de um LFSR de grau m, é possível calcular o valor dos coeficientes de realimentação (taps). A solução para dificultar essa previsibilidade é a utilização de combinações de LFSRs. O

algoritmo de Berlekamp-Massey é capaz de calcular um polinômio característico para uma sequência binária de n dígitos informada [7,68].

Nesta seção foram apresentados alguns geradores pseudo-aleatórios, suas vantagens e desvantagens. É importante ressaltar que o gerador pseudo-aleatório forneceria a semente para a geração de ruídos e seria utilizado somente para uma contenção aos ataques do tipo *side-channel*, sem aumentar a segurança do processo de criptografia.

Nos próximos capítulos, este trabalho apresenta uma combinação dos algoritmos AES e Twofish de modo que a segurança do processo de criptografia seja aumentada e que o processamento de informações (que não possuem dependência entre si) venha a gerar um sinal que descaracterize a informação que está sendo processada.

2.8 *Hardware* reconfigurável

A Computação Reconfigurável é uma solução intermediária na resolução de problemas complexos, possibilitando combinar a velocidade do *hardware* com a flexibilidade do *software*. Dentre os vários segmentos em relação às arquiteturas reconfiguráveis, destacam-se os processadores reconfiguráveis. Estes processadores combinam as funções de um microprocessador com uma lógica reconfigurável e podem ser adaptados depois do processo de desenvolvimento. É representada pelos PLDs (*Programmable Logic Device*), CPLDs (*Complex Programmable Logic Device*), FPGAs (*Field Programmable Gate Array*) e SoCs (*System on Chip*) [79].

Os primeiros dispositivos foram desenvolvidos pela empresa Xilinx Inc., com o seu lançamento no ano de 1985. Nesta ocasião foi apresentado como um dispositivo que poderia ser programado de acordo com as aplicações do usuário. A promessa de flexibilidade de programação, associada a potentes ferramentas de desenvolvimento e modelagem, possibilitaria ao usuário acesso a projetos de circuitos integrados complexos sem os altos custos de engenharia associados aos ASICs (*Aplication-specific Integrated Circuit*) [80].

Neste trabalho são utilizados FPGAs. Estes dispositivos podem apresentar milhares de unidades lógicas. Neste aspecto, estas unidades lógicas são vistas como componentes que podem ser configurados independentemente e que são interconectados a partir de uma matriz de trilhas condutoras e chaves programáveis. A estrutura básica de um FPGA é formada pelo vetor de unidades lógicas e pela matriz de interconexão que podem ser programados pelo usuário [79].

Para configurar o FPGA, é utilizado um arquivo binário que contém as informações necessárias para especificar a função de cada unidade lógica e fechar as chaves da matriz de

interconexão necessárias. Para gerar o arquivo binário, podem ser utilizadas ferramentas de software seguindo um fluxo de projeto previamente determinado [81].

Os recursos adicionais como *flip-flops*, multiplexadores, lógica de transporte (*carry*) dedicado e portas lógicas podem ser utilizados em conjunto com as LUT (*Look-Up Tables*) para desenvolver diversas funções booleanas, multiplicadores e somadores, contadores, conversores serial-paralelo e paralelo-serial, e memórias com praticamente qualquer comprimento de palavra, fornecendo assim flexibilidade ao desenvolvimento [79].

2.9 Linguagens de Descrição de *Hardware*

Para que se estabeleça a maneira como um FPGA irá funcionar é necessário utilizar uma linguagem que descreva o *hardware* a ser inserido no dispositivo. Neste trabalho foi utilizado VHDL.

VHDL é uma linguagem para descrição de *hardware*, é a abreviação de ***VHSIC H**ardware **D**escription **L**anguage.* A sigla VHSIC quer dizer *Very High Speed Integrated Circuits* (Circuitos integrados de altíssima velocidade) e foi criada pelo DoD (Departamento de Defesa Norte Americano) nos anos 80. A primeira versão do VHDL foi chamada de VHDL'87. Foi a primeira linguagem de descrição de *hardware* padronizada pelo IEEE (*Institute of Electrical and Electronics Engineers*) através do padrão IEEE 1076. Depois surgiu o padrão IEEE 1164, que possui algumas modificações [82].

As aplicações básicas de VHDL são em FPGA's, CPLD's e o processo de desenvolvimento de ASIC's (*Aplication Specified Integrated Circuits*). O VHDL é uma linguagem que trabalha de forma paralela por descrever o funcionamento do *hardware* [80].

Em VHDL, assim como nas outras linguagens, os projetos são organizados em hierarquias, isto é, interpretando componentes básicos como blocos que virão a formar um bloco maior, que no caso é o projeto em si. Isso torna o código reutilizável, permitindo fazer alterações e reutilizar as "peças" já desenvolvidas. De modo geral, VHDL é uma linguagem adequada para se trabalhar com grandes projetos, que possuem um alto nível de abstração [82]. Desta forma, escolheu-se realizar o desenvolvimento deste trabalho utilizando a linguagem VHDL.

Neste capítulo foi apresentado um breve resumo a respeito de segurança da informação e as principais características dos algoritmos AES-Rijndael e Twofish, além de conceitos de padronização de HSMs.

O desenvolvimento de um dispositivo de segurança deve ter como foco as vulnerabilidades a que ele está sujeito, por este motivo foi apresentada a análise do tipo *side-channel*. Foram apresentados alguns métodos de geração de sinais pseudo-aleatórios, que poderiam ser utilizados como medida de contenção a este tipo de ataque, mas também apresentam vulnerabilidades. Novamente, é importante ressaltar que a utilização destes métodos aumenta o tamanho do dispositivo, sem garantir o aumento de segurança do mesmo. Este trabalho pretende aumentar a segurança do processo de criptografia, minimizar atrasos inerentes aos algoritmos e, através do processamento destes sinais que não possuem dependência entre si, descaracterize a informação que está sendo processada.

O próximo capítulo apresentará os materiais e métodos utilizados no desenvolvimento deste trabalho e alguns resultados preliminares.

Capítulo 3

Desenvolvimento

Este capítulo tem como objetivo apresentar os métodos e materiais utilizados neste trabalho e as etapas de desenvolvimento do mesmo.

O desenvolvimento deste trabalho foi dividido em três fases, sendo que a primeira delas foi utilizada como fundamentação para as fases seguintes.

A primeira fase foi o estudo do algoritmo Twofish, suas características, modos de desenvolvimento e possíveis aplicações. A segunda fase foi o desenvolvimento do algoritmo Twofish em VHDL, utilizando FPGA para que se verificasse seu funcionamento. Esta fase será apresentada neste capítulo.

Os resultados apresentados utilizam o Twofish 128-bit, com valores representados em hexadecimal. O algoritmo AES apresentado [86] será utilizado na terceira fase. Os códigos de descrição de *hardware* desenvolvidos e utilizados são apresentados nos Apêndices C e D.

Na terceira fase foi realizada a integração em *hardware* dos módulos desenvolvidos, o ajuste e o aprimoramento de algumas funcionalidades. Esta fase será apresentada no próximo capítulo.

3.1 Materiais e Métodos

Durante a competição para a escolha do AES, abordada no capítulo anterior, foi publicada uma documentação a respeito do funcionamento dos algoritmos finalistas. Com relação aos dois algoritmos utilizados neste trabalho, a documentação da competição e alguns livros estão disponíveis para consulta [4, 8, 28-30, 33, 60, 83]. Esta documentação foi utilizada no desenvolvimento e verificação deste trabalho.

O desenvolvimento de cada uma das partes do projeto foi realizado em módulos, utilizando a metodologia *bottom-up*. Nesta metodologia, blocos funcionais mais simples são construídos, testados e validados para que possam fazer parte de blocos funcionais mais complexos.

Um exemplo deste desenvolvimento é o bloco somador de 32 bits, que foi utilizado na transformada Pseudo-Hadamard e no bloco de expansão de chaves.

Outro exemplo é a construção dos S-Boxes, que foi realizada a partir das funções q. Após testes e validações, as funções q foram utilizadas na construção dos S-Boxes, que passaram por seus próprios testes e validação. As funções g e h são mais complexas e utilizaram os S-Boxes em sua construção. Deste modo os blocos funcionais mais simples podem ser utilizados em outros projetos, pois já foram testados e funcionam de acordo com as especificações. A figura 3.1 apresenta a utilização dos blocos q dentro dos S-Boxes, que se encontram inseridos nas funções g.

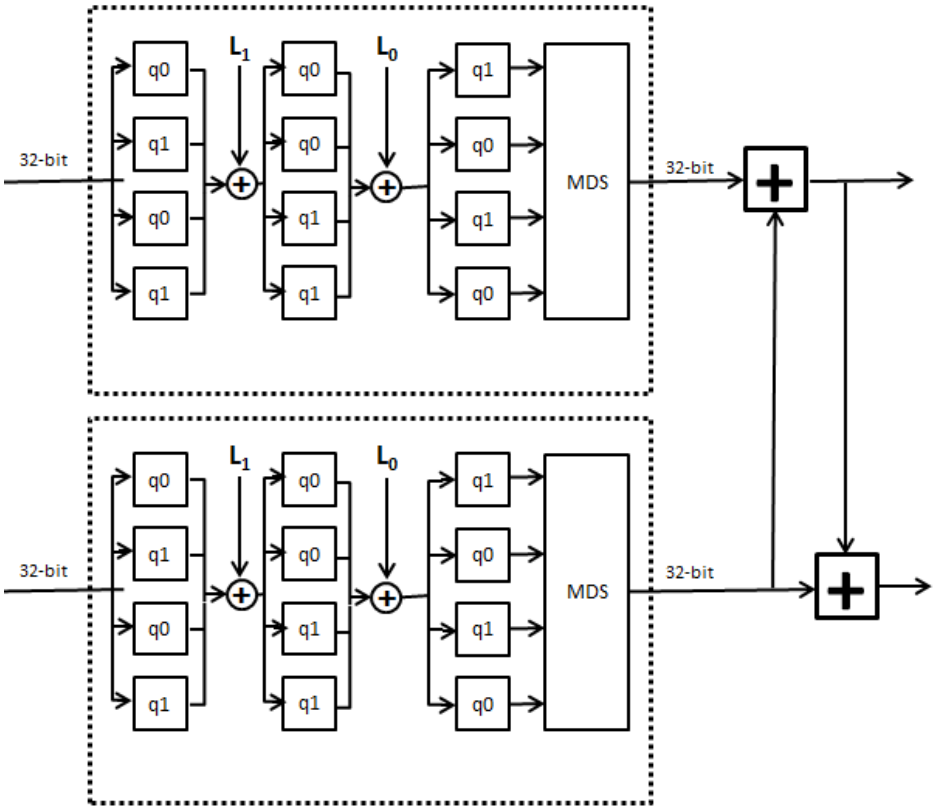


Figura 3.1: Montagem modularizada das funções g

Na construção do código de descrição de *hardware* de cada um dos blocos foram utilizadas as primitivas da linguagem VHDL, de modo que o mesmo possa ser sintetizado nas principais famílias de dispositivos reconfiguráveis como, por exemplo, Xilinx e Altera (agora Intel). O objetivo de desenvolver uma descrição HDL portável é o fato de que este dispositivo não fique vinculado a somente um fabricante de FPGAs.

Os blocos foram desenvolvidos e testados de três modos: ferramentas de simulação (ModelSim), comunicação entre os dispositivos (com *hardware* adicional de controle) e com a utilização de analisadores lógicos.

Foram utilizadas as seguintes ferramentas de síntese e simulação:

- Altera Quartus versões 11 e 13;
- Xilinx ISE versão 14.7;
- Altera ModelSim versão 10.

Para os testes de comunicação entre os dispositivos foram desenvolvidas descrições de *hardware* e *testbenchs* que realizaram a inserção e leitura dos dados de cada uma das funcionalidades. Cada módulo desenvolvido foi testado individualmente.

Após a verificação de funcionamento, foram propostas algumas melhorias relacionadas à ocupação de área no *hardware* reconfigurável. A comparação foi realizada e enviada, através de artigos, a congressos para a validação por especialistas externos. Estes tópicos serão apresentados no próximo capítulo.

No desenvolvimento deste trabalho foram utilizados os seguintes equipamentos:

- Altera EP2C5T144 – Cyclone;
- Altera EP4CE115F29C7 – Cyclone IV;
- Xilinx ZYBO – Zynq Board.
- Xilinx Nexys2 (Digilent) - Spartan 3E-500;
- Gerador de funções Tektronix AFG3021C;
- Analisador Lógico *Logic16* - Saleae;
- Analisador Lógico *Analog Discovery*– Analog Devices and Xilinx.

A figura 3.2 apresenta uma das montagens utilizadas para a realização dos testes de bancada. Nesta montagem foram utilizadas duas placas Altera EP4CE115F29C7 (Cyclone IV), o analisador lógico e o gerador de funções. Foi realizada a cifragem em uma placa e a decifragem na outra. Os dados foram enviados às placas pelo gerador de funções e por um microcontrolador.

Para a realização dos testes e validação de cada um dos blocos foram utilizados os documentos oficiais disponibilizados por cada um dos autores, assim como *testbenchs* e vetores de teste disponibilizados durante a competição para a escolha do AES [8, 28-30, 33, 83]. Alguns vetores de teste são apresentados nos Anexos I e II.

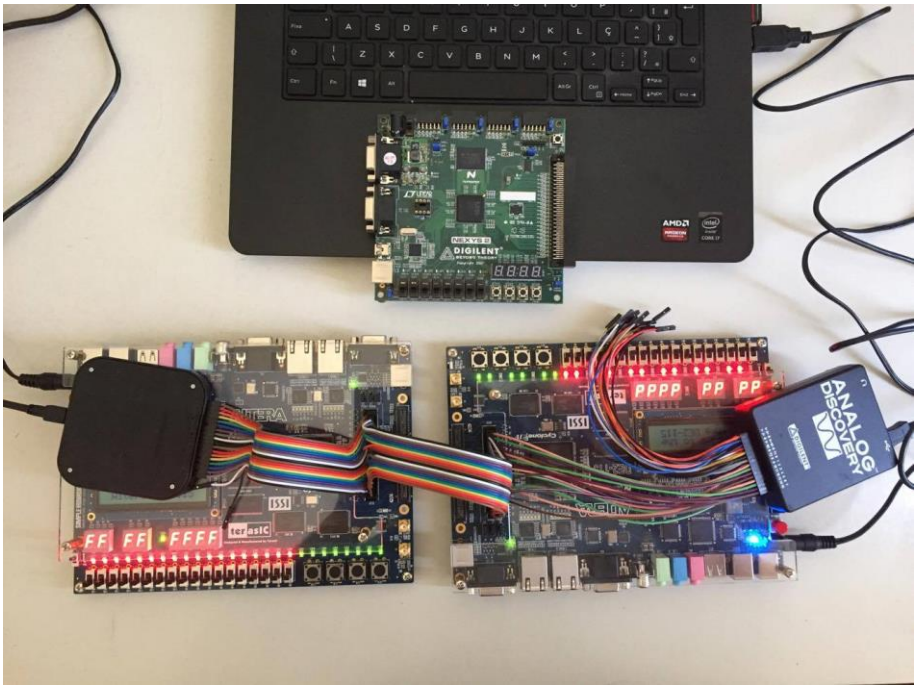


Figura 3.2: Montagem com os equipamentos utilizados para os testes do dispositivo.

3.2 Twofish

O dispositivo foi desenvolvido de modo a fornecer o resultado de cada um dos cálculos intermediários durante o processamento da informação, além de ser modular e permitir que várias configurações possam ser montadas. Nas próximas seções será apresentado o desenvolvimento do *hardware* em FPGA, utilizando VHDL. Um conjunto de valores intermediários que validam o funcionamento do algoritmo pode ser visto na tabela 3.1 [8, 30].

Tabela 3.1: Vetor de testes Twofish128

Texto plano	00000000_00000000_00000000_00000000
Chave inicial	00000000_00000000_00000000_00000000
Chaves do tipo S	00000000_00000000
Input whithening	11F0626D_52C54DDE_4D1B4AAA_7CAC9D4A
Output whitening	1E7D0BEB_B7B83A10_CFE14BE4_EE9C341F
Rodada 0	9C5B3C17_F98FFEF9
Rodada 1	342A4D81_15A48310
Rodada 2	C14724A7_424D89FE
Rodada 3	FDE87320_311B834C
Rodada 4	26CD67B4_3302778F
Rodada 5	C2BAF60E_7A6C6362
Rodada 6	D972C87F_3411B994
Rodada 7	A7DEE434_84ADB1EA
Rodada 8	A2F7CAA8_54D2960F
Rodada 9	8014C425_A6B8FF8C
Rodada 10	EDBAF720_6A748D1C
Rodada 11	0338EE13_928EF78C
Rodada 12	C8314176_9949D6BE
Rodada 13	ECAE7EA7_07C07D68
Rodada 14	85C05C89_1FE71844
Rodada 15	696EA672_F298311E
Texto cifrado	9F589F5C_F6122C32_B6BFEC2F_2AE8C35A

a) Função *q*

O desenvolvimento da função *q* segue o diagrama apresentado na figura 2.12. Alguns valores são calculados, outros são lidos de uma memória ROM que obedece à tabela 2.6. Foi desenvolvida uma ROM de sete bits de largura de endereçamento, com quatro bits de largura de dados, onde cada uma das informações de endereço faz referência a uma célula da tabela 2.6. Uma parte do desenvolvimento desta ROM pode ser vista na figura 3.3. O valor de entrada da ROM, representado por `0XXXXXXX`, contém os valores de q_n e t_n e o valor de entrada da tabela.

```
entity q_unit is
        port(
                clk             : IN std_logic;
                in_q            : IN std_logic;
                load            : IN std_logic;
                rst             : IN std_logic;
                in_value : IN std_logic_vector(7 DOWNTO 0);
                out_ready: OUT std_logic;
                out_q     : OUT std_logic_vector(7 DOWNTO 0)        );
end q_unit;

architecture q_unit_behav of q_unit is
-- (...)
begin
--  q0 t0 0-15
-- 0X  XX XXXX
-- 00000000 a 00001111
        with a1 select
                a2_q0 <= "1000" when "0000", -- 8
                         "0001" when "0001", -- 1
                         "0111" when "0010", -- 7
                         "1101" when "0011", -- D
                         "0110" when "0100", -- 6
                         "1111" when "0101", -- F
                         "0011" when "0110", -- 3
                         "0010" when "0111", -- 2
                         "0000" when "1000", -- 0
                         "1011" when "1001", -- B
                         "0101" when "1010", -- 5
                         "1001" when "1011", -- 9
                         "1110" when "1100", -- E
                         "1100" when "1101", -- C
                         "1010" when "1110", -- A
                         "0100" when others; -- 4
-- (...)
```

Figura 3.3: Parte da função q utilizando uma ROM

Para a validação do funcionamento da função *q*, foram utilizados sinais de controle apresentados na figura 3.4. O sinal interno de controle *sig_out_cont* apresenta o estado em que o processo está. Quando o sinal *sig_out_cont* é igual a 0, a máquina está no estado *reset*. Sendo igual a 1, é a fase de leitura do dado de entrada. Quando este sinal é igual a 2, temos o valor de entrada lido. Os outros estados representam a transformação do valor até seu resultado final, conforme o diagrama da figura 2.12. Considerando o valor de entrada igual a `0xB6` e utilizando uma função q_0, temos como valor de saída `0x50`, confirmando o funcionamento do módulo.

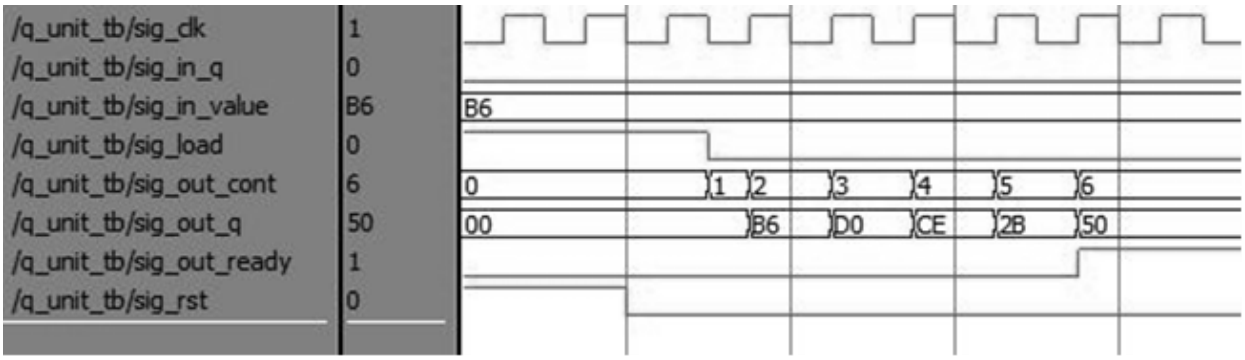


Figura 3.4: Forma de onda da função q

b) Função *S-Box (128 bits)*

O desenvolvimento da função S-Box de 128 bits segue o diagrama apresentado na figura 2.11. Neste caso, deve ser considerada somente a utilização das chaves *S1* e *S0*, isto é, as chaves para 128 bits. A figura 3.5 apresenta parte da descrição de *hardware* da função S-Box_128. É importante ressaltar que a função S-Box utiliza quatro instâncias da função *q*. Por este motivo a figura apresenta a utilização do *componente port* relativo à função *q*.

A figura 3.6 apresenta a forma de onda da função S-Box_128, a partir das entradas de duas chaves de 32 bits (*0x*`BA987654` e *0x*`A9876543`) e de uma palavra de 32 bits (*0x*`8550B161`), tendo como saída o valor *0x*`3B6D231B`. A saída obedece aos critérios de saída do algoritmo, confirmando seu funcionamento correto.

```
entity sBox is -- g function without MDS
    port(
            clk       : IN std_logic;
            rst       : IN std_logic;

            in_load   : IN std_logic;
            in_data   : IN std_logic_vector (31 DOWNTO 0);
            in_L0     : IN std_logic_vector(31 DOWNTO 0); -- keySchedule S0
            in_L1     : IN std_logic_vector(31 DOWNTO 0); -- keySchedule S1

            out_cont  : OUT std_logic_vector(3 DOWNTO 0);
            out_data  : OUT std_logic_vector(31 DOWNTO 0);
            out_ready : OUT std_logic
    );
end sBox;

architecture sBox_behav of sBox is

    component q_unit port(
            clk             : IN std_logic;
            rst             : IN std_logic;
            load            : IN std_logic;
            in_q            : IN std_logic;
            in_value        : IN std_logic_vector(7 DOWNTO 0);
            out_ready       : OUT std_logic;
            out_q           : OUT std_logic_vector(7 DOWNTO 0)
    );
    end component;
-- (...)
```

Figura 3.5: Parte da implementação da função S-Box_128

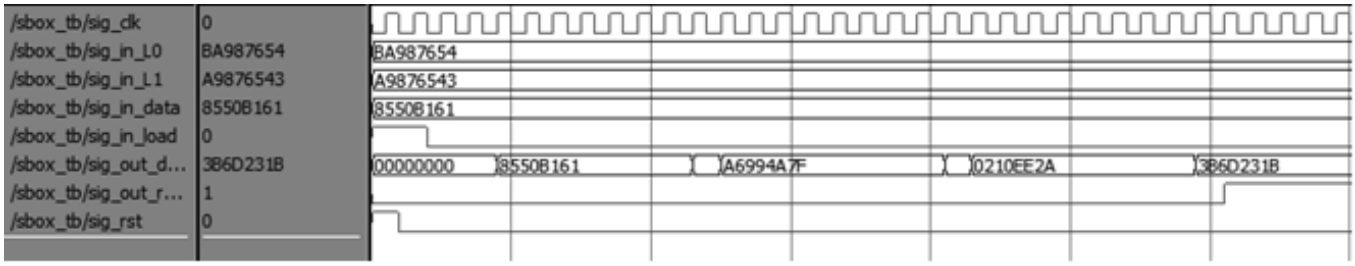


Figura 3.6: Forma de onda da função S-Box_128

c) Função *MDS*

Para o desenvolvimento do bloco MDS, foram utilizados multiplicadores que obedecem às funções apresentadas nas figuras 2.13 e 2.14. São necessárias multiplicações pelas constantes que formam a matriz utilizada nesta função, que são: `0x01`, `0x5B` e `0xEF`. Os resultados destes produtos polinomiais são fornecidos através dos blocos multiplicativos.

A construção do campo de Galois GF(2^8) requer o polinômio primitivo p(x) de grau 8 e o elemento β que é a raiz de p(x). Todos os elementos do campo podem ser expressos em função do elemento primitivo [8, 30, 84]. A partir daí, temos as representações de p(x), 5B e EF, que são:

- $p(x) = x^8 + x^6 + x^5 + x^3 + x^1$
- $5B = 1 + \beta^{-2}$
- $EF = 1 + \beta^{-1} + \beta^{-2}$

Estas representações são utilizadas nas operações da função MDS, tendo suas representações na forma binária: $5B_{16} = 1011011_2$ e $EF_{16} = 11101111_2$.

A partir daí, podemos chegar à seguinte representação para 5B·x:

$$5B \cdot x = (2^6 + 2^4 + 2^3 + 2^1 + 2^0) \cdot x$$
$$= 2^0 \cdot x + (2^6 + 2^4 + 2^3 + 2^1) \cdot x$$
$$= 2^0 \cdot x + 2^{-2} \cdot x + (2^6 + 2^4 + 2^3 + 2^1 + 2^{-2}) \cdot x$$
$$= x + (x>>2) + (2^6 + 2^4 + 2^3 + 2^1 + 2^{-2}) \cdot x$$

Trabalhando nas equações anteriores, chegamos à equação 6.

$$2^6+2^4+2^3+2^1+2^{(-2)} = \frac{2^8+2^6+2^5+2^3+2^0}{2^2} = \frac{169_{16}}{4} \quad (6)$$

A partir daí, temos como resultado a equação 7 [84]:

$$5B \cdot x = x + (x \gg 2) + \frac{169_{16}}{4}[x_0\,2^0 + x_1\,2^1 + x_2\,2^2 + \ldots + x_6\,2^6 + x_7\,2^7] \quad (7)$$

$\frac{169_{16}}{4}$ é um número de 7 bits e todas as suas multiplicações por 2^n, com $n \geq 2$, irão produzir um valor maior que 8 bits. Como este produto é limitado por mod p(x), isto é mod 169_{16}, todos os termos nesta condição serão iguais a zero. Resultando na equação final:

$$5B \cdot x = x + (x \gg 2) + \frac{169_{16}}{4}\,x_0 + \frac{169_{16}}{2}\,x_1 \quad (8)$$

A figura 3.7 apresenta a código em linguagem de descrição de *hardware* que realiza a multiplicação por `0x5B`.

```
entity mult_5B is
    port(
            in_5b     : in std_logic_vector(7 downto 0);
            out_5b    : out std_logic_vector(7 downto 0)
    );
end mult_5B;

architecture mult_5B_arch of mult_5B is
begin
    out_5b(0) <= in_5b(2) XOR in_5b(0);
    out_5b(1) <= in_5b(3) XOR in_5b(1) XOR in_5b(0);
    out_5b(2) <= in_5b(4) XOR in_5b(2) XOR in_5b(1);
    out_5b(3) <= in_5b(5) XOR in_5b(3) XOR in_5b(0);
    out_5b(4) <= in_5b(6) XOR in_5b(4) XOR in_5b(1) XOR in_5b(0);
    out_5b(5) <= in_5b(7) XOR in_5b(5) XOR in_5b(1);
    out_5b(6) <= in_5b(6) XOR in_5b(0);
    out_5b(7) <= in_5b(7) XOR in_5b(1);
end mult_5B_arch;
```

Figura 3.7: Implementação em VHDL da multiplicação por 0x5B.

A figura 3.8 apresenta a forma de onda da função MDS desenvolvida, tendo como entrada o valor *0x*3B6D231B e como saída o valor *0x*1C90068C.

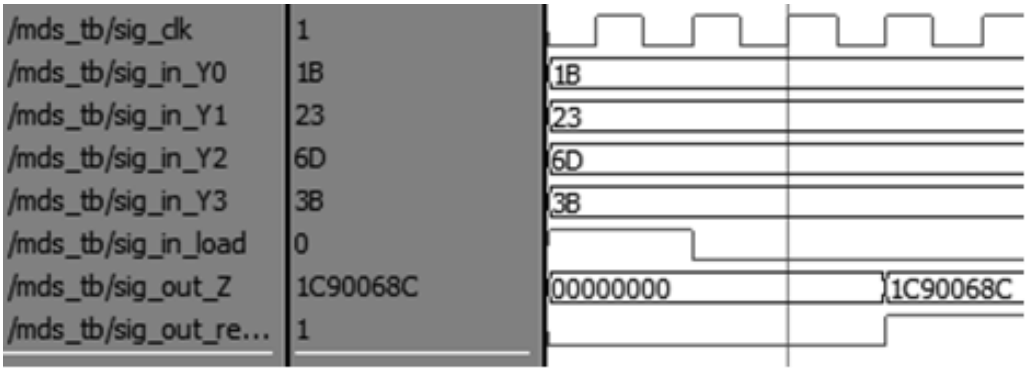


Figura 3.8: Forma de onda da função MDS

A figura 3.9 apresenta o código da instância principal deste bloco. Neste código é possível ver a importação dos blocos multiplicativos `mult_EF` e `mult_5B` através dos comandos *component port*.

```
entity mds is
    port(
            clk         : IN std_logic;
            in_load     : IN std_logic;
            in_Y0 : IN std_logic_vector(7 DOWNTO 0); --LSB
            in_Y1 : IN std_logic_vector(7 DOWNTO 0);
            in_Y2 : IN std_logic_vector(7 DOWNTO 0);
            in_Y3 : IN std_logic_vector(7 DOWNTO 0); --MSB
            out_Z : OUT std_logic_vector(31 DOWNTO 0);
            out_ready           : OUT std_logic
    );
end mds;

architecture mds_behav of mds is
    component mult_EF
    port    (
            in_ef : in std_logic_vector(7 downto 0);
            out_ef : out std_logic_vector(7 downto 0)
    );
    end component;

    component mult_5B
    port    (
            in_5b : in std_logic_vector(7 downto 0);
            out_5b : out std_logic_vector(7 downto 0)
    );
    end component;
-- (...)
```

Figura 3.9: Implementação em VHDL da função MDS

d) Função *g*

A figura 3.10 apresenta a estrutura do código em linguagem de descrição de *hardware* utilizado para o desenvolvimento da função *g*. Neste código são apresentadas as entradas da função e suas saídas, que irão fornecer os dados para a transformada *Pseudo-Hadamard*. Pode ser vista, também, a importação do componente sBox_MDS, que realiza as funções apresentadas anteriormente.

```
library ieee;
use ieee.std_logic_1164.all;

entity gFunc is
    port      (
              clk        : IN std_logic;
              rst        : IN std_logic;
              in_load    : IN std_logic;
              in_data    : IN std_logic_vector(31 downto 0);
              out_data : OUT std_logic_vector(31 downto 0);

              in_S0      : IN std_logic_vector(31 DOWNTO 0); -- keySchedule S0
              in_S1      : IN std_logic_vector(31 DOWNTO 0); -- keySchedule S1
              out_ready: OUT std_logic  );
end gFunc;

architecture gFunc_arch of gFunc  is
    component sBox_MDS
              port      (
                        clk        : IN std_logic;
                        rst        : IN std_logic;
                        in_load    : IN std_logic;
                        in_data    : IN std_logic_vector(31 downto 0);
                        out_data : OUT std_logic_vector(31 downto 0);
                        in_L0      : IN std_logic_vector(31 DOWNTO 0);
                        in_L1      : IN std_logic_vector(31 DOWNTO 0);
                        out_ready: OUT std_logic  );
    end component;
```

Figura 3.10: Implementação em VHDL da função g

A função g é a combinação das funções S-Box e MDS. Esta função é utilizada duas vezes, em paralelo, de modo que seu resultado forneça os dados de entrada para a transformação *Pseudo-Hadamard* (PHT). Foram utilizados como valores de teste para esta função os valores das entradas de duas chaves de 32 bits (*0x*BA987654 e *0x*A9876543) e de uma palavra de 32 bits (*0x*8550B161). Como esperado, o valor de saída é *0x*1C90068C (figura 3.11).

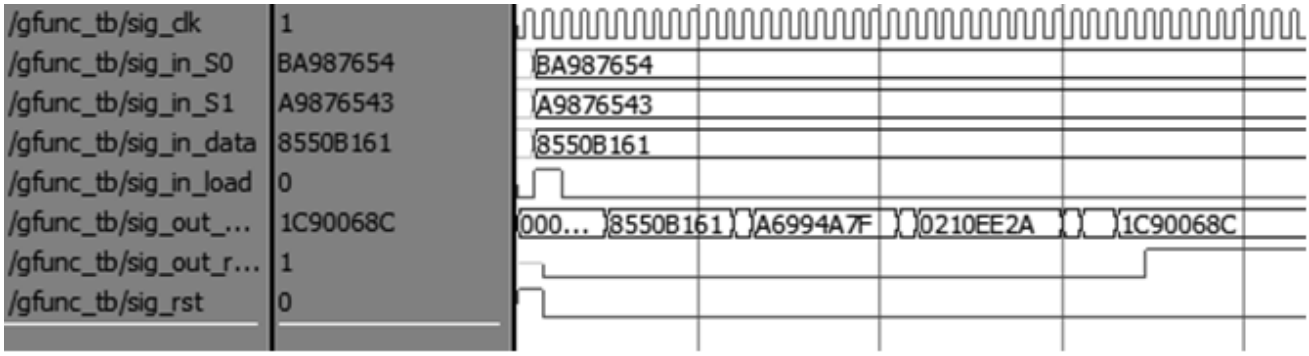


Figura 3.11: Forma de onda da função g

A figura 3.12 apresenta um esquemático da função *g*, com seus blocos, tendo como entrada e saída valores de 32 bits.

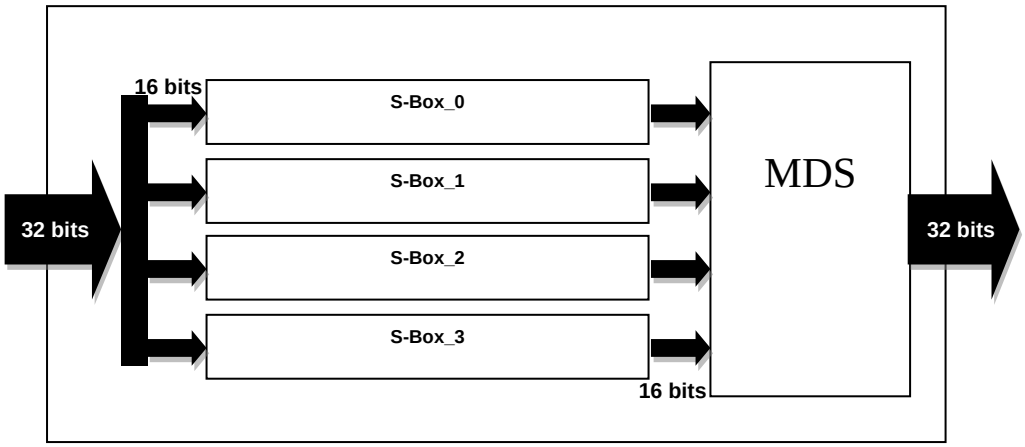


Figura 3.12: Diagrama de blocos da função g

e) Transformação *Pseudo-Hadamard*

A figura 2.15 apresentou como são realizados os cálculos desta transformação. São lidos 64 bits de entrada, divididos em dois barramentos de 32 bits e, após a realização dos cálculos, são fornecidos 64 bits, também apresentados em dois barramentos de 32 bits cada.

Para o desenvolvimento deste bloco foi utilizado um somador de 32 bits, de modo a facilitar o tratamento dos dados e proporcionar a disponibilização de um somador para reutilização durante o projeto.

A figura 3.13 apresenta parte da descrição do somador mencionado.

```
architecture bit_adder32_arch of bit_adder32  is
    signal    sig_intermediate_carry, sig_adder_out  : std_logic_vector(31 downto 0);
    signal    sig_zero                               : std_logic;
    component bit_adder
    port      (
              inA, inB, in_carry    : in std_logic;
              out_Sum, out_carry: out std_logic);
    end component;
begin
    sig_zero <= '0';
    adder32bits: for i in 0 to 31 generate
              adder_LSb: if (i=0) generate  -- LSAdder
                        adder0: bit_adder
                        port map          (
                                  inA => in_dataA(0),
                                  inB => in_dataB(0),
                                  in_carry => sig_zero,
                                  out_Sum => sig_adder_out(0),
                                  out_carry => sig_intermediate_carry(0) );
              end generate adder_LSb;
              others_add: if (i>0) generate  -- 1 to 31 bits Adders
                        next_adder: bit_adder
                        port map          (
                                  inA => in_dataA(i),
                                  inB => in_dataB(i),
                                  in_carry => sig_intermediate_carry(i-1),
                                  out_Sum => sig_adder_out(i),
                                  out_carry => sig_intermediate_carry(i)  );
              end generate others_add;
    end generate adder32bits;
    sig_intermediate_carry(31) <= '0';
    out_data <= sig_adder_out;
end bit_adder32_arch;
```

Figura 3.13: Descrição do somador de 32 bits

A figura 3.14 apresenta a descrição do bloco que realiza a transformação *Pseudo-Hadamard*. Nesta descrição pode ser visto o componente somador de 32 bits descrito anteriormente. Os barramentos *in_T0* e *in_T1* recebem os resultados das funções g, da parte menos significativa e da parte mais significativa, respectivamente.

Para realizar a validação do funcionamento do bloco PHT foram utilizados os seguintes valores de entrada *in_T0* = *0x*1C90068C e *in_T1* = *0x*71417427. Estes valores são o resultado dos cálculos das funções *g* para as entradas *in_g0* = *0x*8550B161 e *in_g1* = 0xA858B0C2, com as chaves S0 = *0x*BA987654 e S1 = *0x*A9876543. A figura 3.15 apresenta os resultados.

```
entity pht is
    port    (
            clk, rst, in_load      : IN std_logic;
            in_T0, in_T1           : IN std_logic_vector(31 downto 0);
            out_P0, out_P1 : OUT std_logic_vector(31 downto 0);
            out_ready              : OUT std_logic );
end pht;

architecture pht_arch of pht  is
    component bit_adder32 port          (
            in_dataA, in_dataB  : IN std_logic_vector(31 downto 0);
            out_data : OUT std_logic_vector(31 downto 0)
            );
    end component;
-- (...)
```

Figura 3.14: Descrição da transformação Pseudo-Hadamard

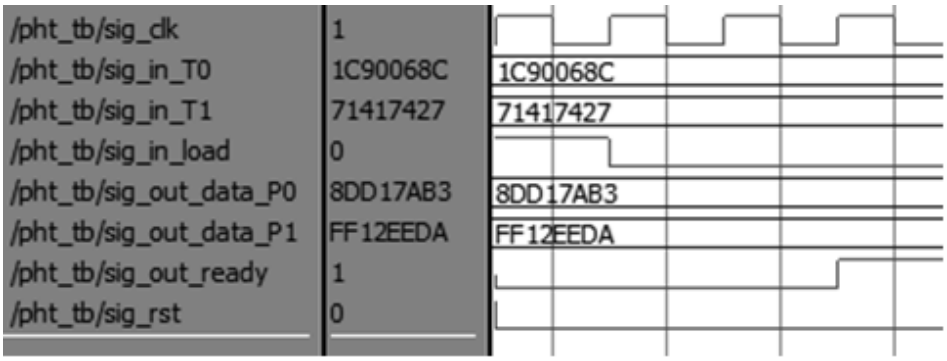


Figura 3.15: Forma de onda da transformação Pseudo-Hadamard

f) Adição das chaves da rodada

Para a adição das chaves da rodada foi utilizado o somador de 32 bits desenvolvido anteriormente. A figura 3.16 apresenta como está disposto o arranjo de blocos até este ponto. Os blocos de expansão das chaves serão mostrados posteriormente.

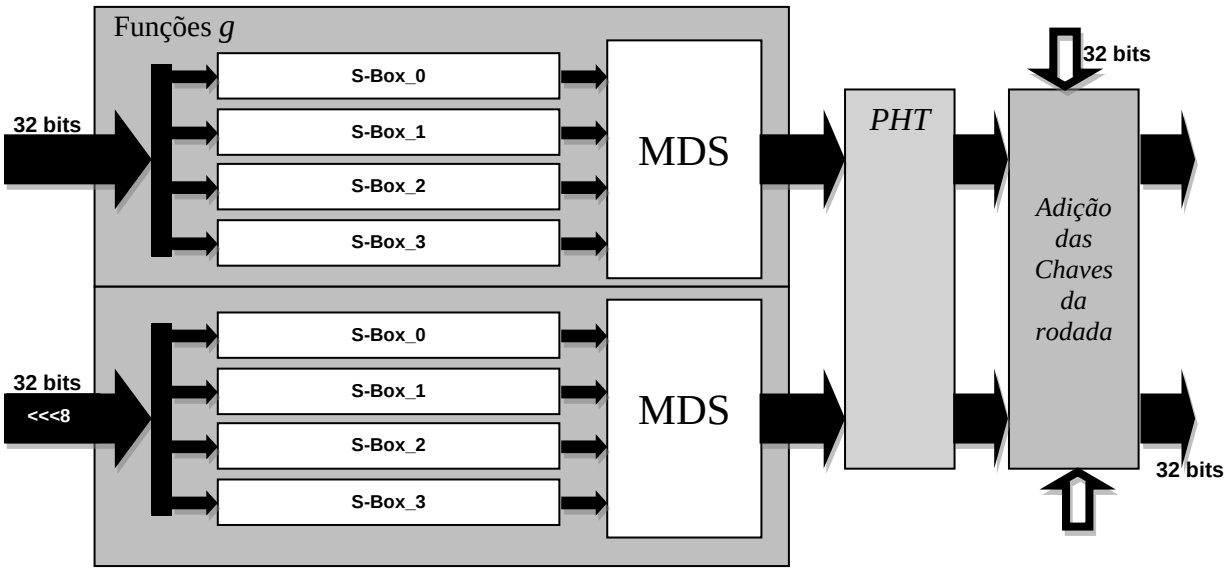


Figura 3.16: Diagrama da comunicação entre os blocos desenvolvidos

Após a adição das chaves da rodada, os dados serão calculados junto com o resultado das transformações realizadas na parte mais significativa da palavra (levando-se em conta a primeira rodada).

Até o momento foi apresentado o desenvolvimento da rede de *Feistel* do algoritmo Twofish, conforme apresentado na figura 2.10.

g) Expansão das chaves

O processo de expansão das chaves é semelhante ao processo de cifragem, com algumas modificações que serão apresentadas nesta seção. Por este motivo a figura 2.17 está sendo apresentada novamente, agora com nova numeração (figura 3.17).

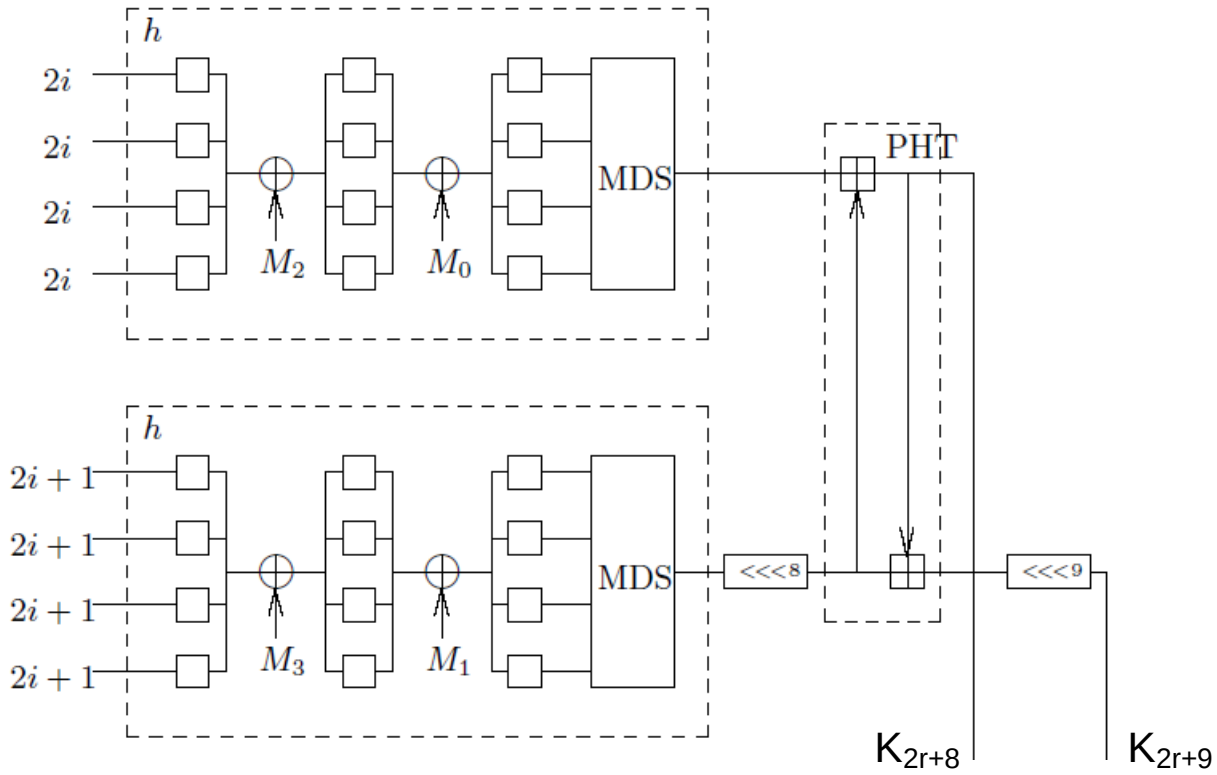


Figura 3.17: Expansão das chaves (128 bits)[8]

Para a expansão das chaves, na rodada de número zero os índices *i* e *r* serão iguais a 0, seguindo a ordem crescente de numeração das rodadas. Deste modo, na rodada zero as entradas das funções *h* são `0x00000000` e `0x01010101`, respectivamente, para a função menos significativa e para a mais significativa.

Há uma diferença importante entre os processos de encriptação e de expansão de chaves. Durante o processo de encriptação, quando se utiliza a função *g*, os valores de chaves S0 e S1 são extraídos através da função *S*, tendo 32 bits cada. O cálculo destes valores é realizado de

maneira semelhante ao da função MDS. A matriz *S* e a operação realizada pela função *S* são apresentadas na figura 3.18.

$$\mathrm{RS} = \begin{pmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{pmatrix} \qquad \begin{pmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \vdots & & \mathrm{RS} & \vdots \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \cdot \begin{pmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{pmatrix}$$

(a) (b)

Figura 3.18: Matriz para a função S (a) e Função S (b)

No caso da expansão das chaves, as entradas M3, M2, M1 e M0 são partes da chave principal do processo, isto é, da chave fornecida pelo usuário. Cada uma delas contendo 32 bits, sendo que o maior índice representa o valor mais significativo. Estas quatro chaves são utilizadas no processo de cifragem com chave de 128 bits.

Outra característica que diferencia a expansão das chaves do processo de encriptação é a saída mais significativa da função *h*, que sofre algumas transformações antes e após a transformação *Pseudo-Hadamard*. Estas transformações são rotações à esquerda de oito e de nove bits, respectivamente. Após estas transformações, as chaves da rodada estão calculadas.

```
entity keyExpansion is
    port    (
            clk             : IN std_logic;
            rst             : IN std_logic;
            in_load         : IN std_logic;

            in_data         : IN std_logic_vector(5 downto 0);

            out_data_0      : OUT std_logic_vector(31 downto 0);
            out_data_1      : OUT std_logic_vector(31 downto 0);

            in_M0           : IN std_logic_vector(31 DOWNTO 0);
            in_M1           : IN std_logic_vector(31 DOWNTO 0);
            in_M2           : IN std_logic_vector(31 DOWNTO 0);
            in_M3           : IN std_logic_vector(31 DOWNTO 0);

            out_ready       : OUT std_logic
    );
end keyExpansion;
```

Figura 3.19: Descrição do bloco de expansão das chaves

A descrição de *hardware* na figura 3.19 apresenta como foi desenvolvida a entidade principal do bloco de expansão das chaves. Este bloco importa os componentes S-Box, MDS e

PHT para utilizar suas funcionalidades.

A figura 3.20 apresenta as formas de onda do processo de expansão das chaves para as operações de *whitening*. Utilizando a entrada *in_data* igual a zero, são calculadas as chaves K_0 e K_1, seguindo até o índice seis, que calcula as chaves K_6 e K_7. Deste modo são calculadas as chaves para os processos de *whitening*.

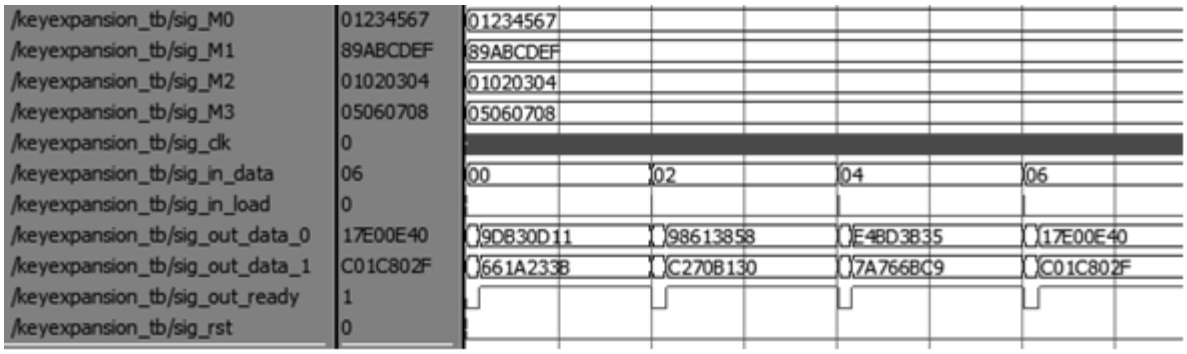


Figura 3.20: Forma de onda da expansão das chaves para a operação de Whitening

A figura 3.21 apresenta as formas de onda do processo de expansão das chaves para as primeiras rodadas. Utilizando os índices de 8 a 38 são calculadas as chaves das rodadas do algoritmo.

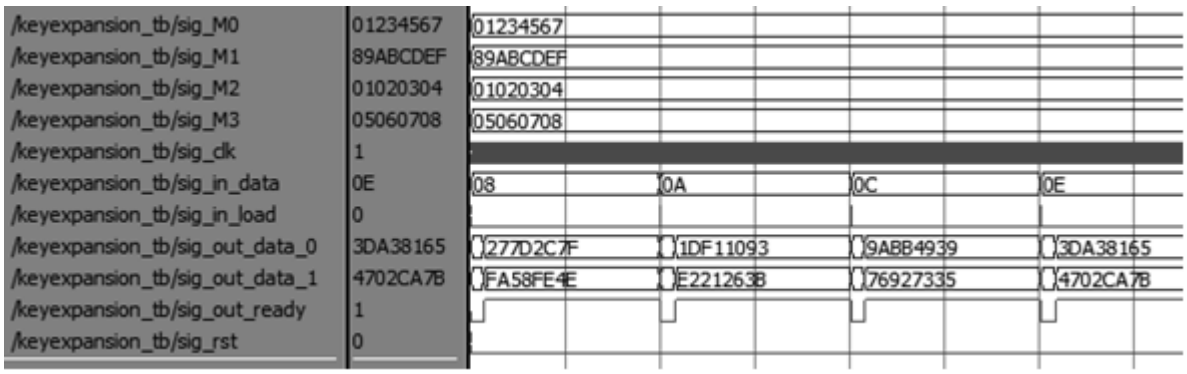


Figura 3.21: Forma de onda da expansão das chaves

h) Cifragem e Decifragem

A entidade principal desta arquitetura realiza o controle das operações, o funcionamento dos processos, a inserção e a leitura de novos dados. Ao final do processamento, sinaliza aos outros dispositivos que os valores estão disponíveis para a leitura. O bloco de controle irá definir as operações de cifragem ou de decifragem, conforme os sinais enviados pelo usuário. Como foi apresentado na figura 2.18, a mudança no contexto de cálculo pode ser realizada através da utilização de multiplexadores que definirão o caminho de dados. Mais informações a respeito dos processos de cifragem e decifragem serão apresentadas no próximo capítulo.

Neste capítulo foram apresentados os materiais e métodos utilizados no desenvolvimento deste trabalho. Foram apresentados os menores blocos funcionais utilizados na construção dos dispositivos de segurança, assim como alguns resultados preliminares de simulação e alguns códigos de descrição de *hardware*. O próximo capítulo irá apresentar a integração em *hardware* dos módulos desenvolvidos e os resultados obtidos.

Capítulo 4

Resultados

Este capítulo irá apresentar uma melhoria desenvolvida para o cálculo das funções g e h através da compactação das funções S-Box [85]. Quando comparada a uma implementação do AES-128 [86], a compactação dos S-Boxes reduziu a área ocupada em *hardware* a 70% com relação a chaves de 128 bits [87] e a 83% em relação a chaves de 256 bits [88].

O segundo resultado foi a comparação de um dispositivo AES-128 [86] com o dispositivo Twofish-128 [87]. Este resultado obteve uma ocupação menor de área que o AES.

O terceiro resultado foi o desenvolvimento de um dispositivo Twofish-256, com ocupação de área menor do que o AES-128 [86], fornecendo um nível muito maior de segurança (256 bits) em uma área menor [87].

O quarto resultado refere-se à proposta e desenvolvimento de um dispositivo de criptografia híbrida de 256 bits, que tem como objetivo utilizar as características dos algoritmos AES e Twofish e gerar um nível de ruído que dificulte a utilização do ataque do tipo *side-channel*.

4.1 – S-Box Compacto

Conforme apresentado na seção 2.2.2, dentro do capítulo Referencial Teórico, o S-Box é uma estrutura que compõe as funções g e h que, por sua vez, são blocos essenciais no funcionamento das redes de Feistel de processamento da palavra e de expansão das chaves. Sendo assim, o primeiro resultado que merece destaque é a redução da área ocupada pelo Twofish em *hardware* reconfigurável através do desenvolvimento de um S-Box compacto.

A documentação do algoritmo recomenda que o esquemático das funções g e h seja construído conforme a figura 4.1 [8, 30]. Utilizando este arranjo de blocos, é possível cifrar dados com chaves de 128, 192 ou 256 bits. No caso de chaves de 128 bits, somente os três conjuntos de funções *q* mais à direita seriam utilizados, ficando desabilitados os outros blocos. Utilizando chaves de 192 bits, seriam utilizados os quatro blocos mais à direita, sendo que o bloco à esquerda ficaria

desabilitado. Somente com a utilização de chaves de 256 bits todos os blocos seriam utilizados para o processamento da mensagem e para a expansão das chaves.

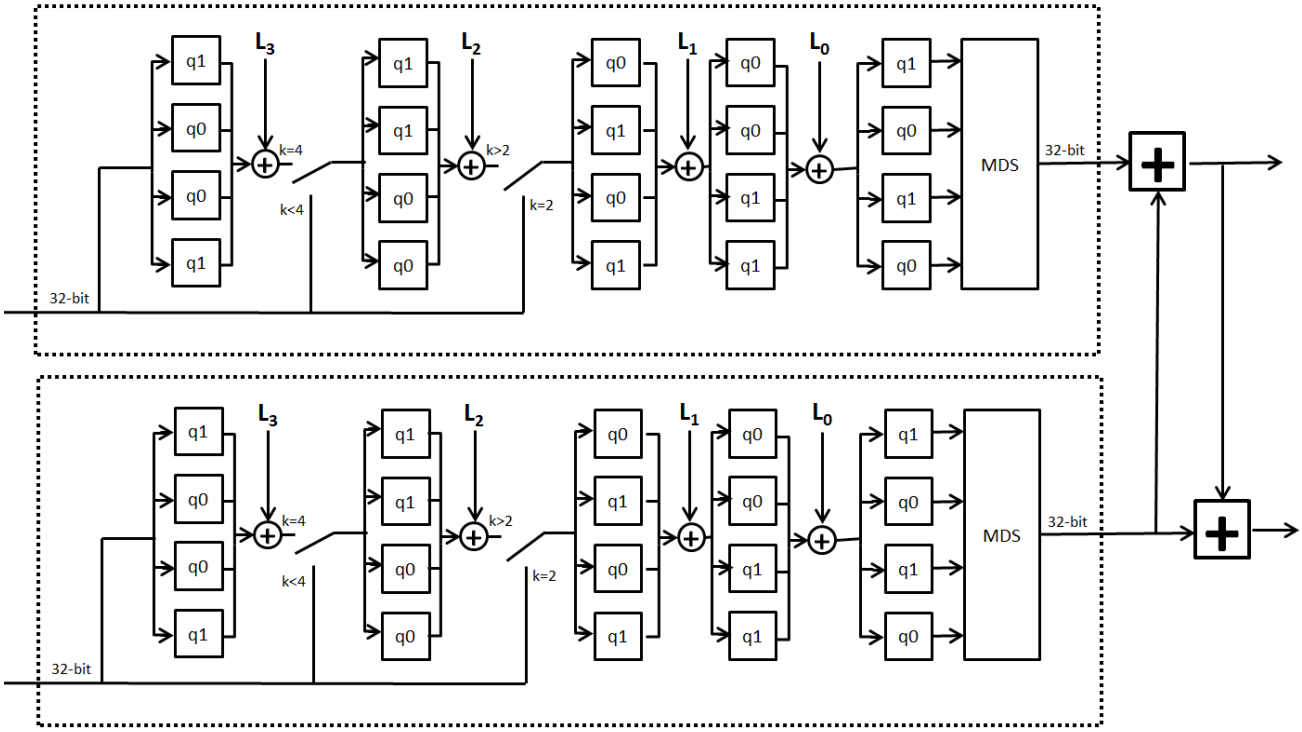


Figura 4.1: Esquemático das funções f e g

Ao utilizar chaves menores (128 ou 192 bits), este esquemático ocupa uma área significativa do dispositivo reconfigurável e que não será efetivamente utilizada enquanto o tamanho destas chaves for mantido.

A partir destas informações e da necessidade de compactação destes blocos, foi desenvolvida uma nova estrutura que possui somente um conjunto de funções q e uma máquina de estados que controla as entradas e saídas de dados. A figura 4.2 apresenta o esquemático do novo bloco desenvolvido.

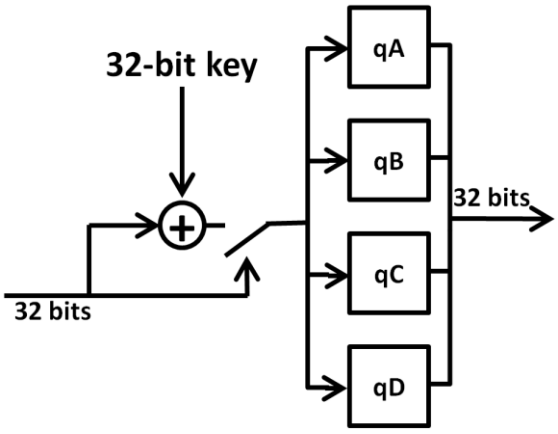


Figura 4.2: S-Box compacto

Este novo bloco apresenta o mesmo funcionamento para as funções *q*, que agora são preenchidas de maneira dinâmica, de acordo com os comandos da máquina de estado que controla as rodadas. A figura 4.3 apresenta a descrição de *hardware* da entidade da função *q* desenvolvida para este S-Box compacto.

```
entity q_unit is
      port(
          clk       : IN std_logic;
          rst       : IN std_logic;
          in_q      : IN std_logic;
          load      : IN std_logic;
          in_value : IN std_logic_vector(7 DOWNTO 0);
          out_ready: OUT std_logic;
          out_q    : OUT std_logic_vector(7 DOWNTO 0)
     );
end q_unit;
```

Figura 4.3: Código HDL da entidade do S-Box compacto

Com a utilização deste novo bloco, obteve-se uma redução do número de elementos lógicos utilizados pela FPGA (Altera *Cyclone* IVE - EP4CE115F29C7). Este novo bloco pode ser dinamicamente modificado para a utilização de qualquer tamanho de chave (128,192 ou 256), sem a modificação de sua área ocupada em *hardware*. De acordo com a tabela 4.1, é possível observar que o bloco proposto é menor que o bloco tradicional quando este utiliza o menor tamanho de chave.

Tabela 4.1: Ocupação do S-Box compacto

S-Boxes	**Nr. Elementos Lógicos**	**Tamanho da chave** (bits)
Compacto	1.064	128 / 192 / 256
Padronizado [8, 30]	1.937	128
	2.627	192
	3.290	256

A figura 4.4 apresenta os resultados de simulação do S-Box Compacto. O sinal *sig_out_ready* em nível alto representa que as informações apresentadas pelo sinal sig_out_data podem ser lidas como resultado do processamento (0x67F2FB00). O processamento desta estrutura utiliza o número máximo de sete ciclos de *clock* para cada iteração, e consiste na leitura da informação de 32 bits, seguida do cálculo com a chave de 32 bits (se necessário). A informação resultante é lida pelas funções *q* e processada conforme apresentado na seção que descreve o funcionamento deste bloco (figura 2.12).

Este primeiro resultado foi apresentado na *9th International Conference on Developments in eSystems Engineering* (DeSE2016), que ocorreu em Liverpool, Reino Unido [85].

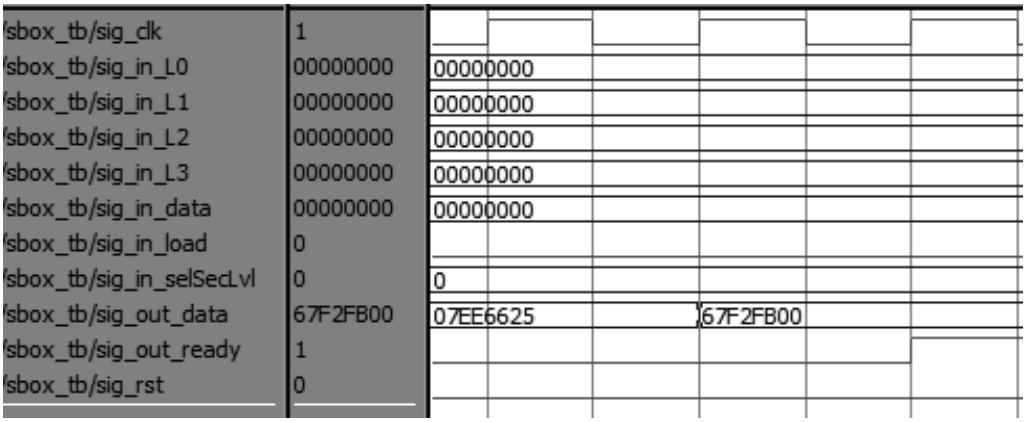


Figura 4.4: Forma de onda do funcionamento do S-Box Compacto

4.2 – Twofish 128

Após o desenvolvimento do S-Box Compacto, foi desenvolvido o Twofish 128. A descrição de *hardware* apresentada na figura 4.5 apresenta os sinais utilizados para o bloco de controle para a cifragem ou decifragem.

Dentre os sinais de controle, merece destaque o sinal *in_mode* que define se a operação realizada será a cifragem (igual a ‘0’) ou a decifragem (igual a ‘1’). O sinal *in_sel_input* permite que o usuário realize a leitura de uma informação, obedecendo à tabela 4.2. O sinal *in_load* em nível alto faz com que o comando fornecido por *in_sel_input* seja realizado. A tabela 4.3 apresenta os sinais de *status* do dispositivo, que permitem seu controle e acompanhamento das informações processadas.

```
entity twofish128 is
	port	(
		clk	: IN std_logic;
		rst	: IN std_logic;
		in_mode: IN std_logic; -- '0' = ciphering / '1' = deciphering
		in_load : IN std_logic;
		in_data : IN std_logic_vector(31 downto 0);
		in_sel_input: IN std_logic_vector(3 downto 0);

		out_data : OUT std_logic_vector(31 downto 0);
		out_sel_output: OUT std_logic_vector(3 downto 0);
		out_ready: OUT std_logic;
		out_error : OUT std_logic
		);
end twofish128;
```

Figura 4.5: Descrição do bloco de controle de cifragem/decifragem

A figura 4.6 apresenta a representação do bloco construído para o dispositivo Twofish128.

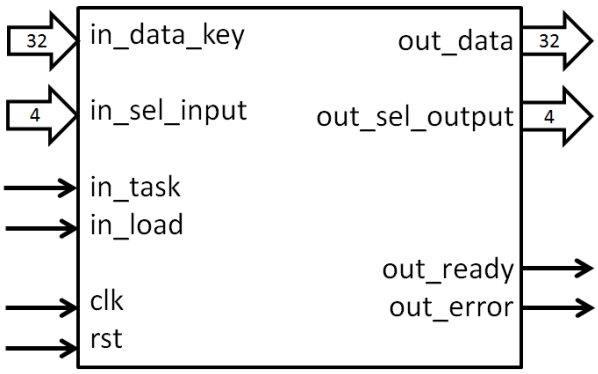


Figura 4.6: Representação do bloco do Twofish128

O dispositivo irá iniciar o processo de cifragem ou decifragem de acordo com o comando inserido no sinal *in_load signal*. Quando este sinal estiver em nível alto, os valores serão lidos de acordo com os comandos apresentados na tabela 4.2. O processo de carregamento das informações verifica todos os valores necessários para o processamento (*out_sel_output = "0100")* e inicia a encriptação quando recebe o comando para iniciar a cifragem ou decifragem (*out_sel_output = "0101"* e *in_sel_input = "1001").*

Tabela 4.2: Sinais de controle do dispositivo

in_sel_input	Ação
0000	Leitura da *LSWord* (Chave 128-bit)
0001	Leitura da 2ª *LSWord* (Chave 128-bit)
0010	Leitura da 3ª *LSWord* (Chave 128-bit)
0011	Leitura da *MSWord* (Chave 128-bit)
0100	Leitura da *LSWord* (Dado)
0101	Leitura da 2ª *LSWord* (Dado)
0110	Leitura da 3ª *LSWord* (Dado)
0111	Leitura da *MSWord* (Dado)
1000	Próximo valor
1001	Iniciar cifragem / decifragem
1010 - 1111	A ser definido

Tabela 4.3: Sinais de status do dispositivo

out_sel_output	Ação
0000	*LSWord* – Informação pronta
0001	2ª *LSWord* – Informação pronta
0010	3ª *LSWord* – Informação pronta
0011	*MSWord* – Informação pronta
0100	Aguardando valores de entrada
0101	Valores de entrada inseridos com sucesso
0110	Ocupado / trabalhando
0111 - 1111	A ser definido

Durante o processo de cifragem ou decifragem, o dispositivo apresenta o *status* ocupado (*out_sel_output = "0110"*). Quando o processamento da informação termina, o sinal *out_ready* ficará em nível alto e o usuário poderá realizar a leitura dos valores processados. Os resultados estarão disponíveis para a leitura no dispositivo até que seja executado um novo processo de carregamento ou que ocorra um comando de *reset*.

O resultado da síntese lógica do Twofish 128-bit é apresentado na tabela 4.4. Nesta tabela é apresentado o número de ciclos de *clock* utilizados para a execução de cada uma das funções, obtido através do teste e validação de cada uma delas.

Tabela 4.4: Resultados da síntese lógica

Funções	**Núm. Elementos Lógicos utilizados**	**Núm. de ciclos de *clock* para execução**	**Pinos I/O**
PHT	155	5	132
Função q	161	7	21
MDS	185	4	67
S-Boxes	1.947	30	200
Funções g / h	3.289	36	73
Expansão das chaves	3.378	42	345
Feistel Function	3.911	48	330

A figura 4.7 apresenta a forma de onda da função MDS (*Maximum Distance Separable*) usando como entrada o valor 0x67F2FB00. Tendo em vista a necessidade de verificar a eficiência deste dispositivo, foi realizada a comparação do mesmo com o AES 128-bit.

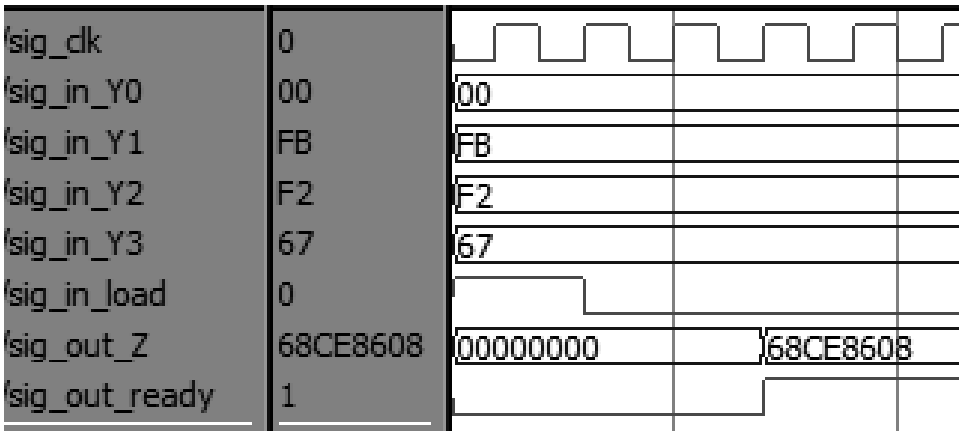


Figura 4.7: Forma de onda da função MDS

O AES-128 [86] utilizou 12,8 mil elementos lógicos na placa Altera Cyclone IV EP4CE115F29C7, enquanto o Twofish-128 [87] utilizou 8,9 mil elementos lógicos. Deste modo, o Twofish-128 ocupou, aproximadamente, 70% da área ocupada pelo AES. Utilizando o clock interno da placa (50 MHz), têm-se o tempo aproximado de processamento de uma palavra, sem *pipelining*, igual a 16 µs. O processamento do AES-128 é executado em 9 µs.

Os resultados desta comparação entre o Twofish desenvolvido e o AES-128 foram apresentados na *8th IEEE International Conference on Information Technology and Electrical Engineering* (ICITEE 2016), que ocorreu em Yogyakarta, na Indonésia [87].

4.3 – Twofish 128/192/256

Com o objetivo de aumentar a segurança do algoritmo de acordo com as diretrizes previstas na documentação da competição [4,8,30], foi desenvolvido o Twofish-128/192/256.

Como apresentado na figura 4.1, as funções g utilizam S-Boxes dependentes de chaves, onde cada S-Box contém permutações de q_0 e q_1, que são permutações fixas de valores de 8 bits. Cada S-Box possui sua configuração para o cálculo das funções *q*. Entre cada um dos cálculos é necessário que se realize uma operação XOR (Ou-Exclusivo) com a chave S correspondente. Para as chaves de 256 bits, o controle dos S-Boxes funciona conforme apresentado nas equações a seguir:

$$y_0 = s_0(x_0) = q_1\,[q_0[q_0[x] \oplus s_{0,0}] \oplus s_{1,0}]$$
$$y_1 = s_1(x_1) = q_0\,[q_0[q_1[x] \oplus s_{0,1}] \oplus s_{1,1}]$$
$$y_2 = s_2(x_2) = q_1\,[q_1[q_0[x] \oplus s_{0,2}] \oplus s_{1,2}]$$
$$y_3 = s_3(x_3) = q_0\,[q_1[q_1[x] \oplus s_{0,3}] \oplus s_{1,3}]$$

Cada S-Box funcionará de acordo com o tamanho da chave escolhida. Utilizando uma chave de 256-bit, temos k=4. Com uma chave de 192-bit, o valor de k é igual a 3. Utilizando 128-bit, k =2.

Para k = 4, têm-se:

$$y_{3;0} = q_1[y_{4;0}] \oplus l_{3;0}$$
$$y_{3;1} = q_0[y_{4;1}] \oplus l_{3;1}$$
$$y_{3;2} = q_0[y_{4;2}] \oplus l_{3;2}$$
$$y_{3;3} = q_1[y_{4;3}] \oplus l_{3;3}$$

Se $k \geq 3$, têm-se:

$$y_{2;0} = q_1[y_{3;0}] \oplus l_{2;0}$$
$$y_{2;1} = q_1[y_{3;1}] \oplus l_{2;1}$$
$$y_{2;2} = q_0[y_{3;2}] \oplus l_{2;2}$$
$$y_{2;3} = q_0[y_{3;3}] \oplus l_{2;3}$$

Em todos os casos, têm-se:

$$y_0 = q_1[q_0[q_0[y_{2;0}] \oplus l_{1;0}] \oplus l_{0;0}]$$
$$y_1 = q_0[q_0[q_1[y_{2;1}] \oplus l_{1;1}] \oplus l_{0;1}]$$
$$y_2 = q_1[q_1[q_0[y_{2;2}] \oplus l_{1;2}] \oplus l_{0;2}]$$
$$y_3 = q_0[q_1[q_1[y_{2;3}] \oplus l_{1;3}] \oplus l_{0;3}]$$

A figura 4.8 apresenta a descrição da entidade HDL para a representação da rede de Feistel utilizando todos os tamanhos de chave.

```
entity feistel is
port   (
       clk      : IN std_logic;
       rst      : IN std_logic;
       in_load : IN std_logic;
       in_data_LS      : IN std_logic_vector(31 downto 0);
       in_data_MS      : IN std_logic_vector(31 downto 0);
       in_key_2r8      : IN std_logic_vector(31 downto 0);
       in_key_2r9      : IN std_logic_vector(31 downto 0);
       in_load_key     : IN std_logic;
       in_L0   : IN std_logic_vector(31 downto 0);
       in_L1   : IN std_logic_vector(31 downto 0);
       in_L2   : IN std_logic_vector(31 downto 0);
       in_L3   : IN std_logic_vector(31 downto 0);
       in_selSecLvl     : IN std_logic_vector (1 downto 0);
       out_data_LS    : OUT std_logic_vector(31 downto 0);
       out_data_MS : OUT std_logic_vector(31 downto 0);
       out_ready       : OUT std_logic
);
end feistel;
```

Figura 4.8: Código HDL da rede de Feistel

O dispositivo foi testado e validado de acordo com as diretrizes e *testbenchs* fornecidos pelos autores/desenvolvedores dos algoritmos [4,8,28-30]. A tabela 4.5 apresenta os resultados da síntese realizada na placa Altera Cyclone IV E (EP4CE115F29C7), com as ferramentas Altera Quartus II 64-Bit Version 13.1.0 Build 162 e Altera ModelSim version 10.1d.

O AES-128 [86] utilizou 12,8 mil elementos lógicos na placa Altera Cyclone IV EP4CE115F29C7, enquanto o Twofish-128/192/256 [88] utilizou 10,6 mil elementos lógicos. Sendo assim, o Twofish-128/192/256 ocupou, aproximadamente, 83% da área ocupada pelo AES,

com um nível muito maior de segurança [4,7,8,28]. Utilizando clock interno da placa (50 MHz), têm-se o tempo aproximado de processamento de uma palavra, sem *pipelining*, igual a 20 µs.

Tabela 4.5: Resultados da síntese logica do Twofish-128/192/256

Funções	Núm. Elementos Lógicos utilizados	Núm. de ciclos de *clock* para execução	Pinos I/O
PHT	155	5	132
Função q	161	7	21
MDS	185	4	67
S-Boxes	1.947	45	200
Funções g / h	3.289	50	73
Expansão das chaves	3.378	56	345
Feistel Function	3.911	62	330

Este resultado foi apresentado na *Conference on Computational Interdisciplinary Sciences* (CCiS 2016), que ocorreu no Instituto Nacional de Pesquisas Espaciais (INPE), em São José dos Campos/SP, Brasil [88].

4.4 – Dispositivo híbrido de 256 bits

Tendo em vista que o ataque a um dispositivo de criptografia AES ocorre devido à caracterização de seus sinais emitidos [10,37,65], a proposta deste dispositivo é realizar uma combinação dos algoritmos AES e Twofish de modo que esta combinação possa reduzir algumas limitações dos dois algoritmos e dificultar o processo de análise do tipo *side-channel* através da emissão de sinais de cada um dos processos, não dependentes entre si. A chave utilizada é de 256 bits, tendo sido especificado que a parte mais significativa será utilizada pelo AES-128 e a parte menos significativa pelo Twofish-128.

A figura 4.9 é executada de maneira simultânea ao funcionamento do bloco que executa o Twofish. Esta é a configuração das funções executadas para todas as versões do AES (128, 192 e 256 bits). Neste caso, a expansão das chaves está sendo realizada durante o processamento da palavra.

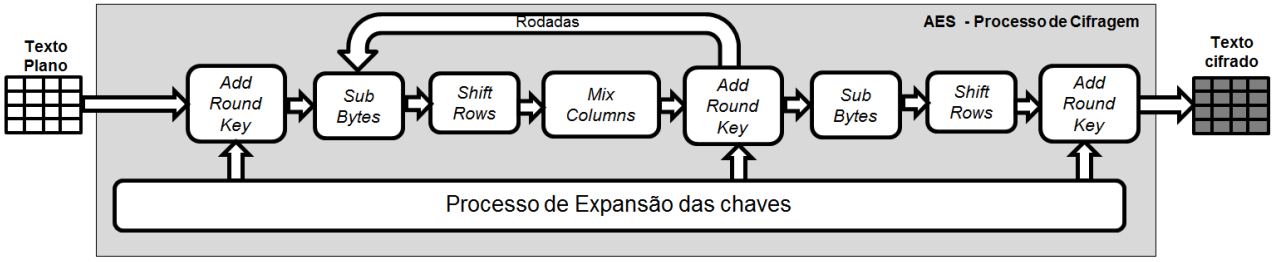


Figura 4.9: Funções executadas durante o processamento do AES

Cada uma das dez formas de onda [12] apresentadas na figura 4.10 representa a emissão eletromagnética realizada pela execução de todos os blocos da rodada (Figura 4.9).

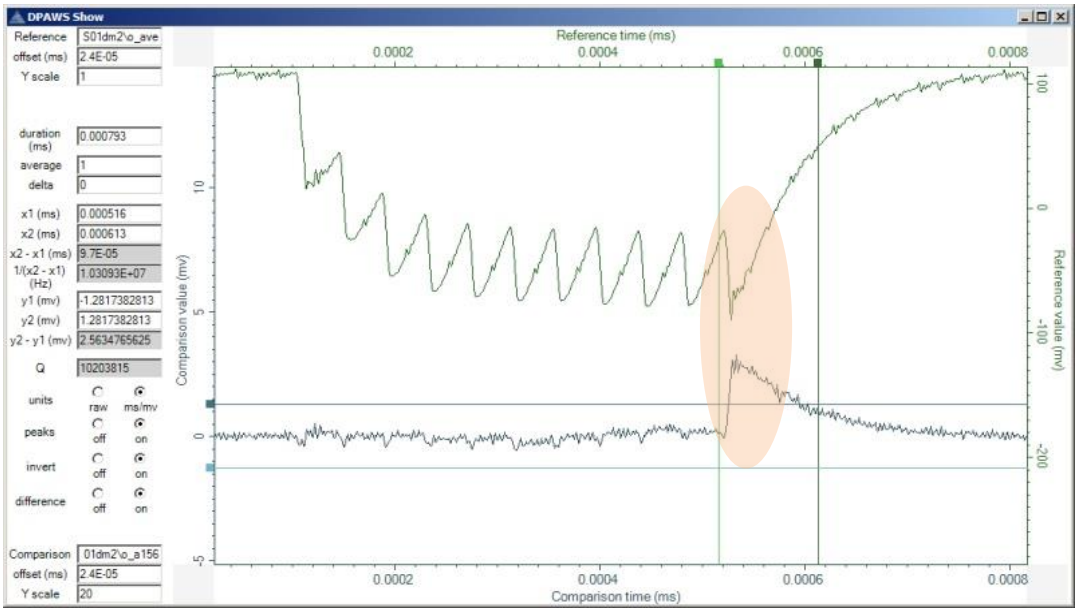


Figura 4.10: Análise side-channel do algoritmo AES [12]

A figura 4.11 apresenta os processos do algoritmo Twofish que são executados de maneira simultânea à primeira rodada do AES. Estes processos geram as chaves S e as primeiras chaves para as operações de *Whitening*, que serão utilizadas pelo Twofish durante o processo de cifragem.

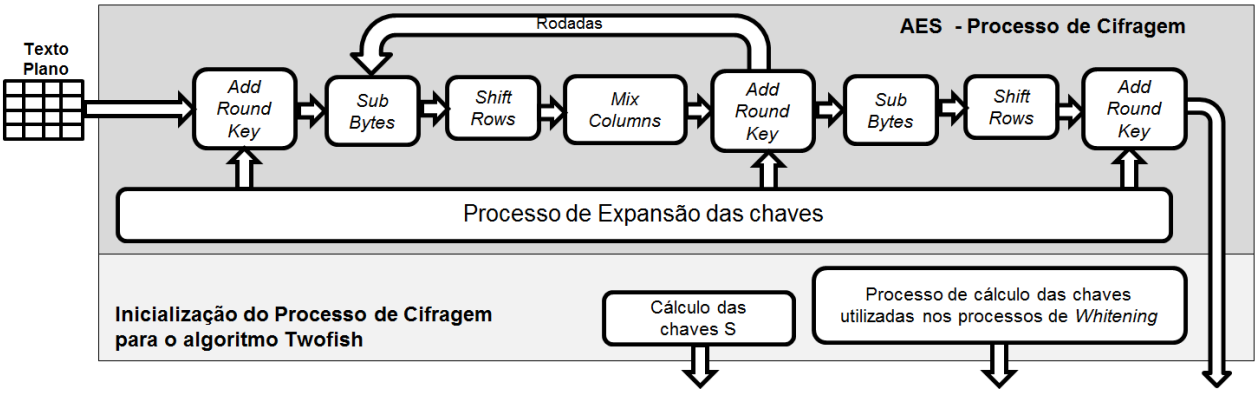


Figura 4.11: Esquemático do dispositivo AES-Twofish (transição)

Após o cálculo da primeira palavra os algoritmos serão executados simultaneamente, gerando sinais relacionados aos processamentos de palavras distintas, com chaves distintas. O algoritmo Twofish realiza o processo de expansão das chaves no modo *on-the-fly*, aumentando o número de sinais que são processados simultaneamente. A figura 4.12 apresenta os blocos que funcionam simultaneamente no algoritmo híbrido para o processamento de cifragem de cada rodada, com exceção da primeira. A combinação da execução simultânea destas funções irá gerar o sinal emitido pelo dispositivo.

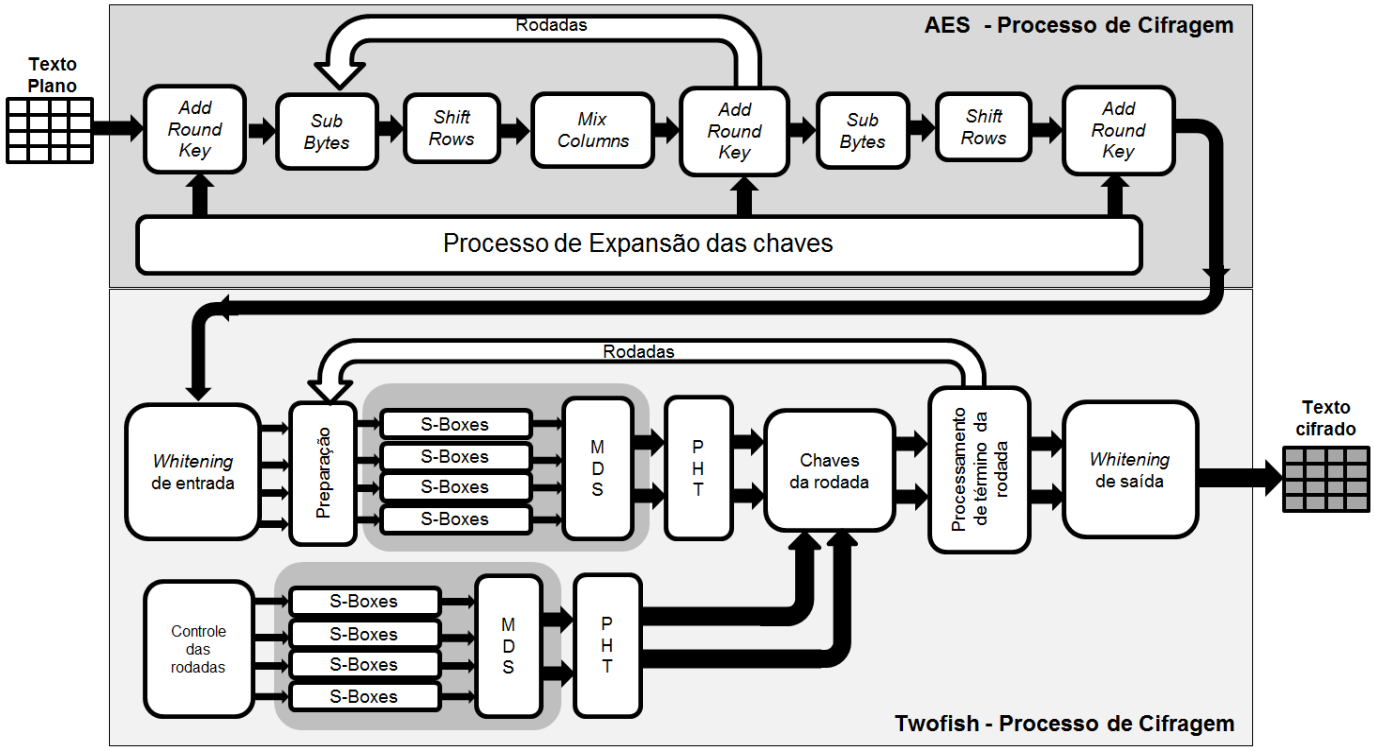


Figura 4.12: Esquemático do dispositivo AES-Twofish (cifragem)

O processo de decifragem utiliza a ordem inversa, onde o primeiro dispositivo a ser utilizado será o Twofish-128. Este processo inverso possui a vantagem que, durante a primeira rodada de decifragem do Twofish-128, o AES-128 realiza o cálculo de todas as sub-chaves que ele necessita para o processo de decifragem. Esta característica elimina a desvantagem de tempo de execução apresentada pelo AES na decifragem, quando em *hardware*. A figura 4.13 apresenta o esquemático do processo de decifragem.

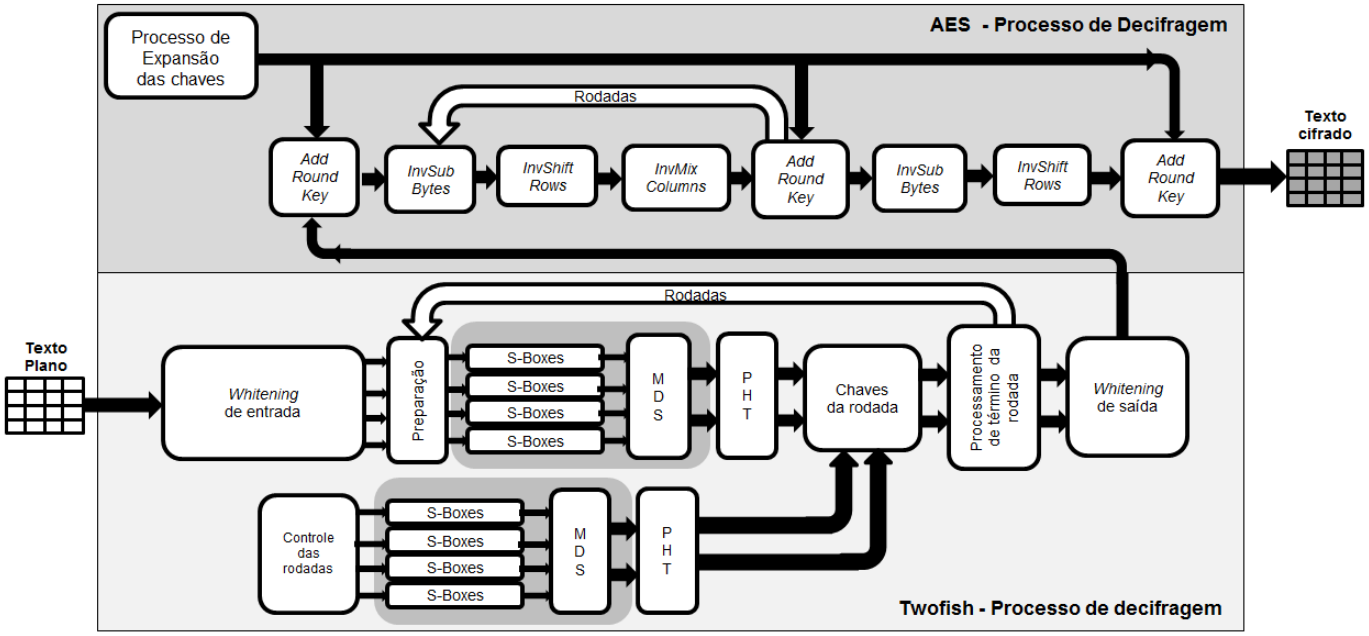


Figura 4.13: Esquemático do dispositivo AES-Twofish (decifragem)

A figura 4.14 apresenta a descrição de *hardware* da entidade da instância mais alta do dispositivo híbrido. Ela utiliza os mesmos sinais e barramentos definidos para o Twofish, apresentados na figura 4.6.

```
ENTITY hibrid256 IS
PORT(
        clk          : IN std_logic;
        rst          : IN std_logic;
        in_task      : IN std_logic;
        in_load      : IN std_logic;
        in_data_key : IN std_logic_vector(31 downto 0);
        in_sel_input: IN std_logic_vector(3 downto 0);
        out_data : OUT std_logic_vector(31 downto 0);
        out_sel_output: OUT std_logic_vector(3 downto 0);
        out_ready    : OUT std_logic;
        out_error : OUT std_logic
  );
END hibrid256;
```

Figura 4.14:Descrição de hardware da entidade do topo do dispositivo híbrido

A tabela 4.6 apresenta os sinais de entrada para o dispositivo. O sinal in_sel_input realiza o controle da ordem dos dados que são colocados no barramento in_data_key, responsável pela leitura da chave e do texto a ser processado. Para que os valores de "0000" a "1011" sejam lidos pelo barramento in_sel_input e processados é necessário que o sinal in_load esteja em nível alto. O sinal in_task define o processo a ser executado: nível baixo representa cifragem e nível alto representa decifragem.

Tabela 4.6: Sinais de controle do dispositivo

in_sel_input	**Ação**
0000	Leitura da palavra (bits 0 a 31)
0001	Leitura da palavra (bits 32 a 63)
0010	Leitura da palavra (bits 64 a 95)
0011	Leitura da palavra (bits 96 a 127)
0100	Leitura da chave para o Twofish (bits 0 a 31)
0101	Leitura da chave para o Twofish (bits 32 a 63)
0110	Leitura da chave para o Twofish (bits 64 a 95)
0111	Leitura da chave para o Twofish (bits 96 a 127)
1000	Leitura da chave para o AES (bits 128 a 159)
1001	Leitura da chave para o AES (bits 160 a 191)
1010	Leitura da chave para o AES (bits 192 a 223)
1011	Leitura da chave para o AES (bits 224 a 255)
1100	Realizar o processamento da informação
1101	Realizar a leitura da informação processada
1110	Preparação para leitura do próximo dado
1111	A ser definido

A tabela 4.7 apresenta os sinais de estado do dispositivo. Este barramento pode ser utilizado para a leitura das informações após o processamento e o acompanhamento das ações executadas.

Tabela 4.7: Sinais de status do dispositivo

out_sel_output	**Ação**
0000	Palavra processada disponível (bits 0 a 31)
0001	Palavra processada disponível (bits 32 a 63)
0010	Palavra processada disponível (bits 64 a 95)
0011	Palavra processada disponível (bits 96 a 127)
0100	Aguardando valores de entrada
0101	Valores de entrada inseridos com sucesso
0110	Processando a informação / Ocupado
0111 - 1101	A ser definido
1110	Preparação para leitura do próximo dado
1111	NOP

Em 2011, Paul Kocher publicou um artigo com as vulnerabilidades do AES, apresentando a fragilidade de alguns algoritmos que utilizam a rede de permutação e substituição quando submetidos à análise do tipo *side-channel*. A combinação dos algoritmos AES e Twofish foi desenvolvida de modo que a rodada mais vulnerável do AES seja executada junto a dois processos do Twofish: um de geração de chaves (*on-the-fly*) e outro de processamento de palavras (cifragem ou decifragem). Este desenvolvimento foi realizado com o objetivo de descaracterizar o sinal que está sendo processado e dificultar a realização de uma análise do tipo *side-channel*. O dispositivo utiliza os blocos funcionais já testados e validados nas seções anteriores.

Na construção deste dispositivo híbrido foi utilizada a placa Altera Cyclone IV EP4CE115F29C7 e foram ocupados 22 mil elementos lógicos. Utilizando clock interno da placa (50 MHz), têm-se o tempo aproximado de processamento de uma palavra igual a 30 µs.

Um artigo referente a este dispositivo de 256 bits foi submetido à avaliação e os resultados do mesmo serão publicados posteriormente.

Capítulo 5

Conclusões

Neste trabalho foram apresentadas algumas melhorias desenvolvidas para o cálculo das funções g e h através da compactação das funções S-Box, reduzindo a área ocupada em *hardware*. Foi realizado, também, o desenvolvimento de um dispositivo Twofish-256, com ocupação de área menor do que o AES-128, fornecendo um nível muito maior de segurança em uma área menor.

Além destes resultados, foi realizado o desenvolvimento de um dispositivo de criptografia híbrida, de chave de 256 bits, que tem como objetivo utilizar uma combinação das características dos algoritmos AES e Twofish apresentando, assim, uma maior segurança e escalabilidade, com objetivo de aumentar a complexidade em análises do tipo *side-channel*. A execução simultânea de processos referentes a cada um dos algoritmos permite que as emissões do dispositivo não sejam características somente de um deles, mas uma mistura de sinais independentes entre si.

É importante realizar algumas considerações com relação à competição AES, uma vez que os critérios e valores de pontuação não foram livremente divulgados pela organização. Pesquisas e análises independentes foram realizadas durante a competição entre os especialistas e empresas participantes fornecendo uma pontuação aos candidatos conforme os critérios de avaliação disponíveis. Uma das análises da pontuação final da competição é apresentada por [89,90] e consta na tabela 5.1. Nesta avaliação não foram utilizados pesos para os critérios avaliados.

Tabela 5.1: Critérios de avaliação dos competidores- AES (Pesos iguais) [89,90]

Critério	Rijndael	Serpent	Twofish	MARS	RC6	Pesos
Segurança Geral	2	3	3	3	2	1
Dificuldade de implementação	3	3	2	1	1	1
Desempenho em *software*	3	1	1	2	2	1
Desempenho em *smart cards*	3	3	2	1	1	1
Desempenho em *hardware*	3	3	2	1	2	1
Características do projeto	2	1	3	2	1	1
Total	**16**	**14**	**13**	**10**	**9**	

A partir desta análise é possível verificar que o algoritmo Rijndael venceu a competição por responder de maneira satisfatória aos critérios apresentados pelo NIST, como: desempenho,

custo de desenvolvimento em diversas plataformas, código pequeno e sua margem de segurança adequada aos padrões solicitados. É importante salientar que a mesma pontuação para cada um dos critérios não representa o mesmo nível de desempenho dos algoritmos. Por exemplo, o algoritmo Twofish (com grandes chaves) e o algoritmo Serpent tiveram índices de segurança muito maiores que os outros três finalistas. Rijndael e RC6 tiveram os melhores resultados levando em consideração somente a eficiência em *software*.

Mudando os pesos da tabela anterior de modo que a segurança dos algoritmos seja o critério com maior valor, temos como resultado a tabela 5.2, onde os algoritmos Rijndael e Serpent empatam na primeira posição e o algoritmo Twofish fica na segunda posição.

Tabela 5.2: Critérios de avaliação dos competidores- AES (Segurança) [89,90]

Critério	Rijndael	Serpent	Twofish	MARS	RC6	Pesos
Segurança Geral	2	3	3	3	2	3
Dificuldade de implementação	3	3	2	1	1	1
Desempenho em *software*	3	1	1	2	2	1
Desempenho em *smart cards*	3	3	2	1	1	1
Desempenho em *hardware*	3	3	2	1	2	1
Características do projeto	2	1	3	2	1	1
Total	**20**	**20**	**19**	**16**	**13**	

Mantendo um alto peso para a caraterística de segurança e ignorando o desempenho em *hardware* e *smart cards* temos como resultado a tabela 5.3, onde o algoritmo Twofish aparece em primeiro lugar e os algoritmos Rijndael, Serpent e MARS empatam na segunda posição.

Tabela 5.3: Critérios de avaliação dos competidores- AES (Segurança e software) [89,90]

Critério	Rijndael	Serpent	Twofish	MARS	RC6	Pesos
Segurança Geral	2	3	3	3	2	3
Dificuldade de implementação	3	3	2	1	1	1
Desempenho em *software*	3	1	1	2	2	1
Desempenho em *smart cards*	3	3	2	1	1	0
Desempenho em *hardware*	3	3	2	1	2	0
Características do projeto	2	1	3	2	1	1
Total	14	14	15	14	10	

O algoritmo Rijndael foi escolhido vencedor por responder aos critérios apresentados pelo NIST, mas os outros cinco finalistas possuem eficiência computacional, flexibilidade de desenvolvimento e podem ser utilizados em outras aplicações de acordo com suas características. Os algoritmos MARS, RC6, Rijndael, Serpent e Twofish foram analisados e testados por matemáticos, criptólogos, criptoanalistas e especialistas do mundo todo. Por este motivo vários trabalhos acadêmicos têm realizado análises comparativas entre o AES e os outros quatro finalistas, fazendo a opção por um dos outros quatro algoritmos de acordo com o objetivo da aplicação [91-96].

Outro ponto importante com relação à utilização dos outros finalistas, em especial, à utilização do Twofish, é que grupos de pesquisa e equipes de desenvolvimento em segurança tem utilizado o Twofish como um dos algoritmos padronizados para a criptografia simétrica. O PGP (*Pretty Good Privacy*) é um *software* de encriptação de dados, desenvolvido por Davi Phil Zimmermann Arimateia e que, atualmente, padronizou como algoritmos de criptografia simétrica o AES e o Twofish. Nenhum dos outros finalistas do AES é utilizado pelo PGP [97-101].

As técnicas de extração de dados, em especial o ataque do tipo *side-channel*, devem ser levadas em consideração durante o desenvolvimento do algoritmo AES. Neste caso, se faz necessária a utilização de um dispositivo adicional para a geração de ruídos que elimine a relação entre a informação emitida e o dado secreto. Este dispositivo ocuparia mais espaço, não fornecendo nenhum nível adicional de segurança relacionado ao tamanho da chave. Como foi apresentado, geradores de sinais aleatórios possuem vulnerabilidades que também poderiam ser atacadas [10,67,73].

A utilização de um algoritmo de criptografia tão forte quanto o AES e que forneça um nível de sinal que descaracterize o funcionamento do mesmo é uma alternativa interessante. O Twofish-128 possibilita a execução de duas funções g, que utilizam as chaves L_0 e L_1, e duas funções h utilizadas na expansão das sub-chaves, que utilizam as chaves S_0, S_1, S_2 e S_3. Neste cenário existe a utilização de seis chaves independentes entre si, além do processamento da palavra da rodada, sendo executados simultaneamente ao AES. Com a utilização do algoritmo Twofish 256, as chaves das funções g passam a ser L_0, L_1, L_2 e L_3, conforme figura 4.1. As chaves das funções h passam a ser S_0, S_1, S_2, S_3, S_4, S_5, S_6 e S_7. Neste cenário, existe a utilização de doze chaves independentes entre si, além do processamento da palavra da rodada. Estas informações podem vir a apresentar um nível de sinal eficiente para a descaracterização do funcionamento do AES. A utilização de *pipelines* poderá auxiliar na quebra da relação existente entre o sinal emitido e o sinal processado [1,2,7,10].

No processamento do AES, todas as chaves devem ser calculadas antes da realização da decifragem, o que prejudica o tempo de execução deste processo em *hardware*, se comparado à cifragem. A combinação deste algoritmo com o Twofish permite que este tempo de processamento não seja perdido, contribuindo com o aumento da vazão do processo. A construção do dispositivo em módulos contribui com sua escalabilidade, fornecendo blocos eficientes e permitindo que o sistema possa crescer através da implementação de *pipelines* ou de outras combinações de blocos estruturais [1,2,5,6].

A comparação realizada entre as implementações do AES-128 e as versões do Twofish (128 e 128/192/256) confirmaram a flexibilidade do Twofish para a utilização em *hardware*

reconfigurável, apresentando um nível de segurança muito maior que o AES e com menor utilização de elementos lógicos. Sendo que ele possui a vantagem de apresentar o modo de expansão das chaves *on-the-fly*, o que pode funcionar como uma medida de geração de sinais simultâneos ao processamento, além de utilizar a mesma estrutura de *hardware* para a cifragem e para a decifragem. O projeto de um S-Box compacto reduziu a área ocupada pelo Twofish em *hardware*, proporcionando um ganho significativo que pode ser verificado no desenvolvimento do algoritmo utilizando chaves de tamanho de 256 bits [85,87,88]. Deste modo, as combinações do algoritmo AES-128 com os algoritmos Twofish-128 e Twofish-256 apresentam dispositivos que oferecem uma segurança maior do que o AES-256 em termos de ataque de força bruta e aumentam a complexidade de realização de uma análise do tipo *side-channel*, com menor ocupação de área e desempenho similar.

O 2º ECRYPT (*European Network of Excellence in Cryptology*), realizado em 2012, apresentou os tamanhos de chave necessários para cada tipo de criptografia, de modo que um nível mínimo de segurança pudesse ser alcançado. A tabela 5.4 apresenta a equivalência entre tamanhos de chave para diversos métodos de criptografia. A coluna denominada *Número de bits* representa a segurança relacionada a um algoritmo de criptografia simétrica. Os outros métodos de criptografia apresentados são o RSA (*Rivest-Shamir-Adleman*), logaritmos discretos (DLOG) e criptografia de curvas elípticas (ECC). A comparação entre os algoritmos é realizada tendo-se como base o número de bits necessários para fornecer o mesmo nível de segurança [102].

Tabela 5.4: Equivalência de tamanhos de chaves de criptografia [102]

Número de bits	**RSA**	**DLOG**		**ECC**
		Campo	***Sub-campo***	
48	480	480	96	96
56	640	640	112	112
64	816	816	128	128
80	1248	1248	160	160
112	2432	2432	224	224
128	3248	3248	256	256
160	5312	5312	320	320
192	7936	7936	384	384
256	15424	15424	512	512

O algoritmo RSA é um dos primeiros criptosistemas de chave pública e é amplamente utilizado para transmissão segura de dados. Foi desenvolvido por Ron Rivest, Adi Shamir e Leonard Adleman, em 1977. Neste tipo de criptosistema são utilizadas duas chaves: uma pública e outra privada. O RSA está baseado na dificuldade prática de se realizar a fatoração do produto de dois grandes números primos, o problema de *factoring*. A execução deste algoritmo é relativamente lenta. Ele é utilizado mais freqüentemente para a transferência e compartilhamento das chaves de

um algoritmo de criptografia simétrica. DLOG faz referência a logaritmos discretos, que representam um grupo para os quais o processamento computacional é aparentemente difícil. Em alguns casos não há algoritmos eficientes conhecidos para o pior caso. Existe um grande tráfego na internet que utiliza grupos que são da ordem de 1024-bits. A criptografia de curvas elípticas (ECC) é um tipo de criptografia de chave pública baseada na estrutura algébrica de curvas elípticas sobre campos finitos. Ela requer chaves menores comparadas à criptografia baseada em campos Galois para fornecer segurança equivalente. Curvas elípticas são aplicáveis para criptografia, assinaturas digitais e outras tarefas. Elas também são usadas em vários algoritmos de fatoração de inteiros, que têm aplicações em criptografia. Seu uso em criptografia foi sugerido por Neal Koblitz e Victor S. Miller, em 1985. Algoritmos de criptografia de curva elíptica entraram em uso amplo em 2004 [1,2,5-7,9].

O NIST informou, através de relatório técnico publicado em 2015, que a utilização de algoritmo de criptografia simétrica com chave de tamanho 256 bits é segura, garantindo a utilização destes dispositivos nos sistemas de segurança atuais [103].

A partir destas análises, o resultado final obtido neste trabalho representa a combinação de algumas funcionalidades complementares dos algoritmos Twofish e AES, com um maior nível de segurança e maior escalabilidade, se comparado ao AES-256. Foi apresentada, também, a melhoria de implementação em *hardware* com uma redução significativa de área do dispositivo a partir do bloco básico (S-Box). Os resultados obtidos permitirão que estes dispositivos sejam utilizados em projetos futuros e na definição de novas estruturas e combinações entre os algoritmos.

5.1 Trabalhos futuros

Tendo em vista a participação de diversas empresas e grupos de pesquisa no desenvolvimento de medidas de contenção e na verificação das vulnerabilidades do tipo *side-channel*, a principal atividade prevista como continuação para este projeto é a realização destas análises nos dispositivos desenvolvidos, através da utilização da ferramenta *ChipWhisperer* [10, 65-69].

Uma segunda atividade prevista é o aumento no tamanho das chaves dos algoritmos. A utilização de um AES-256 junto a um Twofish-256 fornecerá uma segurança maior ao dispositivo. Deste modo, será possível trabalhar com chaves de 512 ou 1024 bits, de acordo com o arranjo que for desenvolvido.

Há, ainda, a possibilidade de aplicação deste projeto e de suas melhorias na área automotiva, de equipamentos médicos, internet das coisas e em *Smart Grids*.

Com a aquisição da fabricante Altera pela Intel e a divulgação dos projetos que a mesma apresentou para a área de dispositivos reconfiguráveis, é possível que este projeto e suas melhorias possam ser utilizados em microprocessadores reconfiguráveis.

Referências Bibliográficas

[1] TERADA, Routo. Segurança de dados: criptografia em redes de computador . 2. ed. rev. ampl. São Paulo, SP: Blucher, 2008. 305 p.

[2] STALLINGS, Willian. Criptografia e segurança de redes: Princípios e práticas. 4. ed. São Paulo: Pearson Prentice Hall, 2008. 512 p.

[3] E.F.F. CRACKING DES - Secrets of How federal Encryption Research, agencies Wiretap Politics subvert & Chip Design. Editora O'Reilly Media. 1998.

[4] FIPS-197, Federal Information Processing Standards Publication FIPS-197, Advanced Encryption Standard (AES). Disponível em: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>. Acesso realizado em Outubro/2015.

[5] MORENO, E. D.; PEREIRA, F. D.; CHIARAMONTE, R. B. Criptografia em Software e hardware. Novatec. 2005

[6] MENEZES, A.; van OORSCHOT, P. and VANSTONE, S. Handbook of Applied Cryptography, CRC Press, 1996.

[7] PAAR, Christof; PELZL, Jan. Understanding Cryptography – A Textbook for Students and Practitioners. Springer-Verlag Berlin Heidelberg, 2010.

[8] SCHNEIER, B.; KELSEY, J.; WHITING, D.; WAGNER, D.; HALL, C.; FERGUSON, N. Twofish: A 128-Bit Block Cipher. 1998. Disponível em: <http://www.counterpane.com/twofish.html>. Acesso realizado em Outubro/2015.

[9] PESCATORE, J. Using Hardware-Enabled Trusted Crypto to Thwart Advanced Threats. A SANS Whitepaper Written, September/2015.

[10] LNCS 8622. “Constructive Side-Channel Analysis and Secure Design”. 5th International Workshop. COSADE, 2014. Paris, France. April, 2014.

[11] MANGARD, S.; OSWALD, E.; Popp, T. “Power Analysis Attacks – Revealing the secrets of Smart Cards”. Springer, 2010.

[12] KOCHER, P. C., JAFFE, J.; Jun, B. Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp.388-397. Springer, Heidelberg (1999).

[13] KOCHER, P. C.; JAFFE, J.; et al., “Introduction to differential power analysis”, Journal of Cryptographic Engineering 1[1], pp 5-27 (2011)

[14] IEEE CS. Enhancing Security and Privacy in Traffic-Monitoring Systems. Pervasive Computing. 2006.

[15] MICROSEMI. Introduction to ADAS and Secure Connected Car. White Paper . 2015.

[16] BRUTON, J. A. Securing CAN Bus Communication: An Analysis of Cryptographic Approaches. National University of Ireland, Galway. August, 2014.

[17] GOMATHISANKARAN, M.; NAMUDURI, K. R. Secure Embedded Platform for Networked Automotive Systems. University of North Texas, Digital Library. 2011.

[18] PIKE, L.; SHARP, J.; TULLSEN, M.; HICKEY, P. C.; BIELMAN, J. Securing the Automobile: a Comprehensive Approach. Embedded Security in Cars Conference, May 2015.

[19] EFTHYMIOU, C.; Kalogridis, G.; , Smart Grid Privacy via Anonymization of Smart Metering Data, Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on , vol., no., pp.238-243, 4-6 Oct. 2010.

[20] CRYPTO MUSEUM. Disponível em: < http://www.cryptomuseum.com/>. Acesso realizado em 12 de Outubro de 2016.

[21] ALMEIDA, H. "A longa (e interminável) construção da biografia do padre Landell". In: Klöckner, Luciano & Cachafeiro, Manolo Silveiro (orgs.). Por que o Pe. Roberto Landell de Moura foi inovador? Conhecimento, fé e ciência. EdiPUCRS, 2012, pp. 17-37.

[22] CASONATTO, O. D. O Padre Landell de Moura e a Ciência. Monografia. Pontifícia Universidade Católica do Rio Grande Sul, 2010.

[23] SANTOS, C. A. A. "Landell de Moura ou Marconi, quem é o Pioneiro?" In: XXVI Congresso Brasileiro de Ciências da Comunicação. Belo Horizonte, 2003.

[24] FORNARI, E. O "incrível" Padre Landell de Moura. col: Biblioteca do Exército; 537. Coleção General Benício; v. 224, 2 ed. Rio de Janeiro: Biblioteca do Exército, 1984.

[25] BLETCHLEY PARK. Second World War Disponível em: <https://www.bletchleypark.org.uk/content/hist/worldwartwo/enigma.rhtm>. Acesso realizado em 12 de Outubro de 2016.

[26] IWM. Alan Turing and the Enigma. Disponível em: <http://www.iwm.org.uk/history/how-alan-turing-cracked-the-enigma-code>. Acesso realizado em 12 de Outubro de 2016.

[27] MITSURI, Matsui. Linear Criptanalysis Method for DES Cipher. In advances in Criptology, Lectures Notes in Computer Science Vol. 765, Springer Verlag, pp. 386-397, 1993.

[28] DAEMEN, J.; RIJMEN, V. AES Proposal: Rijndael. 1999. Disponível em: <http://csrc.nist.gov/ >. Acesso realizado em Outubro/2015.

[29] DAEMEN, J. and RIJMEN, V. The design of Rijndael: AES - The Advanced Encryption Standard. Springer-Verlag. 2002.

[30] SCHNEIER, B.; KELSEY, J.; WHITING, D.; WAGNER, D.; HALL, C.; FERGUSON, N. On the Twofish Key Schedule. 1999. Disponível em: <http://csrc.nist.gov/ >. Acesso realizado em Outubro/2015.

[31] FERGUSON, N. Upper bounds on differential characteristics in Twofish. Counterpane Systems. 1998.

[32] AES REQUIREMENTS. Disponível em: <http://csrc.nist.gov/publications/nistbul/itl97-02.txt>. Acesso realizado em 12 de Outubro de 2016.

[33] DAEMEN, J. and RIJMEN, V. A Specification for The AES Algorithm. NIST (National Institute of Standards and Technology). Disponível em: <http://csrc.nist.gov/archive/aes/rijndael/wsdindex.html>. Acesso realizado em Outubro/2015.

[34] KOCHER, Paul (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. Advances in Cryptology—CRYPTO'96. Lecture Notes in Computer Science1109: 104–113.

[35] LERMAN, L.; BONTEMPI, G.; MARKOWITCH, O. "A machine learning approach against a masked AES". Journal of Cryptographic Engineering, vol. 5, number 2, pages 123-139, 2015.

[36] PROUFF, E. "DPA Attacks and S-Boxes". Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers. Springer Berlin Heidelberg, 2005.

[37] MORADI, A. "Side-Channel Leakage through Static Power". Cryptographic Hardware and Embedded Systems -- CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings. Springer Berlin Heidelberg, 2014.

[38] OSWALD, E.; Mangard, S. and Pramstaller, N.. Secure and Efficient Masking of AES - A Mission Impossible? Cryptology ePrint Archive (http://eprint.iacr.org/), Report 2004/134, 2004.

[39] DAYALAMURTH, D.Forensic Memory Dump Analysis And Recovery Of The Artefacts Of Using Tor Bundle Browser. Proceedings of the 11th Australian Digital Forensics Conference. University, Perth, Western Australia. December, 2013.

[40] AMARI, K. Techniques and Tools for Recovering and Analyzing Data from Volatile Memory. SANS Institute InfoSec. March, 2009.

[41] STIRPARO, P.; FOVINO, I. N.; KOUNELIS, I. Data-in-Use leakages from Android Memory - Test and Analysis. IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob). 2013

[42] GARTNER. Software Security Is Soft Security: Hardware Is Required. 2000. Disponível em: <www.gartner.com/document/359830>. Acesso realizado em Outubro/2015.

[43] SANS. Disponível em: <www.sans.org/reading-room/whitepapers/analyst/implementing-hardware-roots-trust-trusted-platform-module-age-35070>. Acesso realizado em Outubro/2015.

[44] SOJA, R. Automotive Security: From Standards to Implementation. Freescale. Disponível em: <http://cache.freescale.com/files/automotive/doc/white_paper/AUTOSECURITYWP.pdf>. Acesso realizado em Outubro/2015.

[45] HSM - Hardware Security Modules: Critical to Information Risk Management. Disponível em: <http://mpa.co.nz/media/34423/hsm-critical-to-information-risk-management.pdf>. Acesso realizado em Outubro/2015.

[46] BARCO SILEX FPGA. Design Speeds Transactions In Atos Worldline Hardware Security Module. Barco Silex. 2013. Disponível em: <http://www.electronicspecifier.com/design-automation/adyton-barco-silex-ip-atos-worldline-fpga-design-speeds-transactions-hardware-security-module>. Acesso realizado em Outubro/2015.

[47] ATTRIDGE, J. An Overview of Hardware Security Modules. 2002

[48] OPENDNSSEC. HSM Buyers' Guide. Disponível em: <https://wiki.opendnssec.org/display/DOCREF/HSM+Buyers'+Guide>. Acesso realizado em Outubro/2015.

[49] THE THREE TENENTS - U.S. Air Force Software Protection Initiative. Disponível em: <http://www.spi.dod.mil/tenets.htm>. Acesso realizado em Outubro/2015.

[50] INFOSEC INSTITUTE. The Top Five Cyber Security Vulnerabilities. Disponível em: <http://resources.infosecinstitute.com/the-top-five-cyber-security-vulnerabilities-in-terms-of-potential-for-catastrophic-damage/ >. Acesso realizado em Outubro/2015.

[51] CISECURITY - Center for Internet Security, Critical Security Controls. Disponível em: <www.cisecurity.org/critical-controls.cfm>. Acesso realizado em Outubro/2015.

[52] PCWORLD. Beyond Google, rogue digital certificates also targeted Yahoo domains, possibly others. July 10, 2014.

[53] HEARTBLEED. Disponível em: <http://heartbleed.com>. Acesso realizado em Outubro/2015.

[54] POODLE. Disponível em: < www.openssl.org/~bodo/ssl-poodle.pdf>. Acesso realizado em Outubro/2015.

[55] SANS - Institute Reading Room. Implementing Hardware Roots of Trust: The Trusted Platform Module Comes of Age. June 2013.

[56] NIST - INITIATING REVIEW OF CRYPTOGRAPHIC STANDARDS DEVELOPMENT PROCESS. Disponível em: <http://csrc.nist.gov/groups/ST/crypto-review/index.html>. Acesso realizado em Outubro/2015.

[57] NISTIR 7977. Disponível em: <http://csrc.nist.gov/publications/drafts/nistir-7977/nistir_7977_draft.pdf>. Acesso realizado em Outubro/2015.

[58] VERIZON. Quantify the impact of a data breach with new data from the 2015. Disponível em: <http://www.verizonenterprise.com/DBIR/2015/?utm_source=pr&utm_medium=pr&utm_campaign=dbir2015>. Acesso realizado em Outubro/2015.

[59] KECCAK. Disponível em: <http://keccak.noekeon.org/>. Acesso realizado em Outubro/2015.

[60] FIPS 180-4. Disponível em: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>. Acesso realizado em Outubro/2015.

[61] TRUECRYPT. Disponível em: <https://truecrypt.ch/>. Acesso realizado em Outubro/2015.

[62] EGRESS - The 'Snowden effect' shakes industry confidence in Cloud-based communication solutions and will continue to influence future IT procurement, survey claims. Disponível em: <http://www.egress.com/2014-market-survey/>. Acesso realizado em Outubro/2015.

[63] TCNEXT. Disponível em: <http://truecrypt.sourceforge.net>. Acesso realizado em Outubro/2015.

[64] SULLIVAN, M. NSA director just admitted that government copies of encryption keys are a big security risk. 2015. Disponível em: <http://venturebeat.com/2015/09/24/nsa-director-just-admitted-that-government-copies-of-encryption-keys-are-a-security-risk/>. Acesso realizado em Outubro/2015.

[65] PROJECT VAULT. Side-Channel Power Analysis of AES Core. Disponível em: <http://colinoflynn.com/tag/fpga/ >. Acesso realizado em Outubro/2015.

[66] GREENBERG, A. Googlers' Epic Hack Exploits How Memory Leaks Electricity. WIRED, 2015. Disponível em: <http://www.wired.com/2015/03/google-hack-dram-memory-electric-leaks>. Acesso realizado em Outubro/2015.

[67] NewAE. Disponível em: <https://newae.com/>. Acesso realizado em 12 de Outubro de 2016.

[68] KICKSTARTER. ChipWhisperer. Disponível em: <https://www.kickstarter.com/projects/coflynn/chipwhisperer-lite-a-new-era-of-hardware-security>. Acesso realizado em 12 de Outubro de 2016.

[69] HACKADAY. ChipWhisperer. Disponível em: <https://hackaday.io/project/956-chipwhisperer-security-research>. Acesso realizado em 12 de Outubro de 2016.

[70] ROSA, Fernando Henrique Ferraz Pereira da; JUNIO, Vagner Aparecido Pedro. Gerando Números Aleatórios. 2002. Laboratório de Matemática Aplicada Prof. Dr. Eduardo Colli. Disponível em: <http://feferraz.net/files_/lista/random_numbers.pdf>. Acesso em: 15 nov. 2014.

[71] PEREIRA JÚNIOR, Álvaro Rodrigues; FREITAS, Maria Eugênia de Almeida; LACERDA, Wilian Soares. Geração de Números Aleatórios. Sinergia, São Paulo, v. 3, n. 2, p.154-161, jul./dez. 2002. Semestral. Disponível em: <http://www.cefetsp.br/edu/prp/sinergia/complemento/sinergia_2002_n2/pdf_s/segmentos/artigo_12_v3_n2.pdf>. Acesso em: 15 nov. 2014.

[72] NAKAMURA, Dionathan. Geração de números aleatórios. 2011. Instituto de Matemática e Estatística. Universidade de São Paulo. Disponível em: <http://www3.ime.usp.br/~isc2002/lsd/attachments/063_principal.pdf>. Acesso em: 15 nov. 2014.

[73] BERKENBROCK, Gian Ricardo. Geração de Números Aleatórios. [s.d.]. Instituto Tecnológico de Aeronáutica. Disponível em: <http://www.comp.ita.br/~gian/teep37/cap_2-geracao_de_numeros_aleatorios.signed.pdf>. Acesso em: 15 nov. 2014.

[74] CAIXINHA, João Miguel Faleiro; GODINHO, Pedro. Estudo do estado da arte sobre algoritmos para geração de números aleatórios. Instituto Politécnico de Beja. Escola Superior de Tecnologia e Gestão. Disponível em: <http://caixinha.alojamentogratuito.com/documentos/5946_6355_Estudo_da_Arte_PRNG.pdf>. Acesso em: 15 nov. 2014.

[75] VIEIRA, Carlos Eduardo Costa; E SOUZA, Reinaldo de Castro; RIBEIRO, Celso Carneiro. Um estudo comparativo entre três geradores de números aleatórios. 2004. PUC. Disponível em: <ftp://ftp.inf.puc-rio.br/pub/docs/techreports/04_16_vieira.pdf>. Acesso em: 14 nov. 2014.

[76] MAZZOTTI, Bruno Franciscon. Co-projeto de hardware/software do filtro de partículas para localização em tempo real de robôs móveis. 2010. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2010. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-14052010-164204/>. Acesso em: 15 nov. 2014.

[77] JAGANNATAM, Archana. Mersenne Twister – A Pseudo Random Number Generator and its variants. 2008. Disponível em: < http://cryptography.gmu.edu/~jkaps/download.php?docid=1083>. Acesso em: 15 nov. 2014.

[78] LACHTER, Carlos Alberto. Desenvolvimento de um analisador de erro para redes Ethernet Óptica. 2007. 90 f. Dissertação (Mestrado) - Curso de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007. Disponível em: <http://www.maxwell.vrac.puc-rio.br/Busca_etds.php?strSecao=resultado&nrSeq=11319@1>. Acesso em: 15 nov. 2014.

[79] CASILLO, L. Projeto e implementação em FPGA de um processador com conjunto de instrução reconfigurável utilizando VHDL. Universidade Federal do Rio Grande do Norte, Natal-RN. 2005

[80] BOTROS, Nazeih M. HDL Programming Fundamentals: VHDL and Verilog. Davinci Engineering, 2005.

[81] ARANTES, D. and CARDOSO, F. FPGA e Fluxo de Projeto. In: DECOM-FEECUNICAMP, Campinas-SP. 2008

[82] PEDRONI, Volnei A. Circuit Design with VHDL. Cambridge, MA: MIT Press, 2004

[83] SCHNEIER, B.; WHITING, D. A Performance Comparison of the Five AES Finalists. 2000.

[84] Carnegie Mellon University - School of Computer Science. Study of MDS matrix used in Twofish AES algorithm. Disponível em: <http://www.cs.cmu.edu/~aarti/pubs/MDS.pdf>. Acesso realizado em 12 de Outubro de 2016.

[85] GOMES, O. S. M. ; MORENO, R. L. A compact S-Box module for 128/192/256-bit symmetric cryptography hardware. In: 9th International Conference on Developments in eSystems Engineering 2016 – DeSE, 2016, Liverpool, UK.

[86] GOMES, O. S. M. ; PIMENTA, T. C. ; MORENO, R. L. . A Highly Efficient FPGA Implementation of AES Cryptography. In: 2nd IEEE Latin American Symposium on Circuits and Systems 2011 - LASCAS, 2011, Bogotá. p. 86-89.

[87] GOMES, O. S. M. ; MORENO, R. L. A compact 128-bits symmetric cryptography hardware module. In: 8th IEEE International Conference on Information Technology and Electrical Engineering 2016 - ICITEE, 2016, Yogyakarta, Indonesia.

[88] GOMES, O. S. M. ; MORENO, R. L. A compact 128/192/256-bits symmetric cryptography hardware module. In: Conference on Computational Interdisciplinary Sciences 2016 - CCiS, 2016, Instituto Nacional de Pesquisas Espaciais (INPE) - São José dos Campos, Brazil.

[89] AES Finalists Analysis. Disponível em: <http://crypto.stackexchange.com/questions/11104/how-exactly-was-the-finalist-chosen-in-the-nist-aes-competition>. Acesso realizado em 12 de Outubro de 2016.

[90] AES Cryptography. Disponível em: <http://www.javaworld.com/article/2076084/java-security/aes--cryptography-advances-into-the-future.html>. Acesso realizado em 12 de Outubro de 2016.

[91] RIZVI, S.A.M, HUSSAIN, S. Z., WADHWA, N. “Performance Analysis of AES and Twofish Encryption Schemes”. 2011 International Conference on Communication Systems and Network Technologies.

[92] GÖTZFRIED, J. “Advanced Vector Extensions to Accelerate Crypto Primitives”. Friedrich-Alexander-Universität, Erlangen-Nürnberg. Bachelor Thesis. July, 2012.

[93] VERMA, H. K., SINGH, R. K. “Performance Analysis of RC6, Twofish and Rijndael Block Cipher Algorithms”. International Journal of Computer Applications (0975 – 8887), Volume 42– No.16, March 2012.

[94] MUSHTAQUE, A.; DHIMAN, H.; HUSSAIN, S.; MAHESHWARI, S. “Evaluation of DES, TDES, AES, Blowfish and Two fish Encryption Algorithm: Based on Space Complexity”. International Journal of Engineering Research & Technology (IJERT), Vol. 3, Issue 4, April – 2014.

[95] HU, L; LI, Y.; LI, T.;, LI, H., CHU, J. “The efficiency improved scheme for secure access control of digital video distribution”. Multimedia Tools and Applications. pp 1-18. January 2015.

[96] KHANAPUR, N. .; PATRO, A. “Design and Implementation of Enhanced version of MRC6 algorithm for data security”. International Journal of Advanced Computer Research, Volume 5, Issue 19, June/2015.

[97] RFC 4880. Disponível em: <https://tools.ietf.org/html/rfc4880>. Acesso realizado em Outubro/2015.

[98] OPENPGP. Disponível em: <http://www.openpgp.org/>. Acesso realizado em Outubro/2015.

[99] FREECODE GNU PG. Disponivel em: <http://freecode.com/projects/gnupg/>. Acesso realizado em Outubro/2015.

[100] GNUPG. Disponível em: <https://www.gnupg.org/documentation/manuals/gcrypt/Available-ciphers.html>. Acesso realizado em Outubro/2015.

[101] LibGCrypt. Disponível em: <http://directory.fsf.org/wiki/Libgcrypt>. Acesso realizado em Outubro/2015.

[102] ICT-2007-216676 - ECRYPT II - European Network of Excellence in Cryptology II - ECRYPT II Yearly Report on Algorithms and Keysizes - (2011-2012) - Revision 1.0 - 30. Sept 2012.

[103] NIST, Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths, SP 800-131A. June, 2015.

Apêndice A

Funcionamento do algoritmo AES-Rijndael

O algoritmo de criptografia AES, trabalha sobre o Campo de Galois GF, ou seja, todas as operações matemáticas utilizadas são realizadas sobre este campo, e usa-se o polinômio irredutível $m(x) = x^8 + x^4 + x^3 + x + 1$. Outros polinômios poderiam ser utilizados no desenvolvimento deste algoritmo mas, segundo os criadores do AES, m(x) foi escolhido pois ele é o primeiro polinômio irredutível de grau 8, quando multiplicado em $GF(2^8)$, apresentado em uma listagem desenvolvida por R. Lidl e H. Niederreiter, no livro "*Introduction to finite fields and their applications*" (*Cambridge University Press* - 1986) [28].

Tabela A.1: Valores de todos os passos do AES

Fase de Cifragem	Rodada	Palavra de 16 bytes	Rodada	Palavra/chave de 16 bytes
Chave Inicial	00	3243F6A8885A308D313198A2E03707		
Palavra Inicial	00	2B7E151628AED2A6ABF7158809CF4		
Palavra da Rodada	01	193DE3BEA0F4E22B9AC68D2AE9F84	06	FL006F55C1924CEF7CC88B325DB
Após SubBytes	01	D42711AEE0BF98F1B8B45DE51E415	06	A163A8FC784F29DF10E83D234CD
Após ShiftRows	01	D4BF5D30E0B452AEB84111FL1E279	06	A14F3DFE78E803FC10D5A8DF4C6
Após MixColumns	01	046681E5E0CB199A48F8D37A280626	06	4B868D6D2C4A8980339DF4E837D2
Chave da rodada	01	A0FAFE1788542CB123A339392A6C76	06	6D88A37A110B3EFDDBF98641CA0
Palavra da Rodada	02	A49C7FF2689F352B6B5BEA43026A50	07	260E2E173D41B77DE86472A9FDD
Após SubBytes	02	49DED28945DB96F17F39871A770253	07	F7AB31F02783A9FF9B4340D354B5
Após ShiftRows	02	49DB873B453953897F02D2F177DE96	07	F783403F27433DF09BB531FF54AB
Após MixColumns	02	584DCAF11B4B5AACDBE7CAA81B6B	07	1415B5BF461615EC274656D7342A
Chave da rodada	02	F2C295F27A96B9435935807A7359F6	07	4E54F70E5F5FC9F384A64FB24EA6
Palavra da Rodada	03	AA8F5F0361DDE3EF82D24AD268324	08	5A4142B11949DCLFA3E019657A8C
Após SubBytes	03	AC73CF7BEFCLLLDF13B5D6B545235	08	BE832CC8D43B86C00AE1D44DDA
Após ShiftRows	03	ACC1D6B8EFB55A7B1323CFDF45731	08	BE3BD4FED4E1F2C80A642CC0DA
Após MixColumns	03	75EC0993200B633353C0CF7CBB25D	08	00512FDLB1C889FF54766DCDFA1
Chave da rodada	03	3D80477D4716FE3E1E237E446D7A88	08	EAD27321B58DBAD2312BF5607F8
Palavra da Rodada	04	486C4EEE671D9D0D4DE3B138D65F5	09	EA835CF00445332D655D98AD8596
Após SubBytes	04	52502F2885A45ED7E311C807F6CF6A	09	87EC4A8CF26EC3D84D4C4695979
Após ShiftRows	04	52A4C89485116A28E3CF2FD7F6505E	09	876E46A6F24CE78C4D904AD897E
Após MixColumns	04	0FD6DAA9603138BF6FC0106B5EB31	09	473794ED40D4E4A5A3703AA64C9F
Chave da rodada	04	EF44A541A8525B7FB671253BDB0BA	09	AC7766F319FADC2128D12941575C
Palavra da Rodada	05	E0927FE8C86363C0D9B1355085B8B	10	EB40F21E592E38848BA113E71BC3
Após SubBytes	05	E14FD29BE8FBFBBA35C89653976CA	10	E9098972CB31075F3D327D94AF2E
Após ShiftRows	05	E1FB967CE8C8AE9B356CD2BA974F	10	E9317DB5CB322C723D2E895FAF0
Após MixColumns	05	25D1A9ADBD11D168B63A338E4C4C	10	
Chave da rodada	05	D4D1C6F87C839D87CAF2B8BC11F91	10	D014F9A8CGEE258GE13F0CC8B66
			Saída	3925841D02DC09FBDC118597196A

Nesta seção será apresentado o funcionamento de uma rodada do algoritmo AES [4,28,29]. A tabela A.1 apresenta todos os valores intermediários para o cálculo da cifragem de 128 bits, da palavra *0x*`2b7e151628aed2a6abf7158809cf4f3c`, com a utilização da chave *0x*`3243f6a8885a308d313198a2e0370734`.

A figura A.1 apresenta os valores das sub-chaves para cada uma das 10 rodadas, resultantes do cálculo da expansão das chaves.

	COLUNA_1	COLUNA_2	COLUNA_3	COLUNA_4
Chave inicial	2b 7e 15 16	28 ae d2 a6	ab f7 15 88	09 cf 4f 3c
Chave Rodada 01:	a0 fa fe 17	88 54 2c b1	23 a3 39 39	2a 6c 76 05
Chave Rodada 02:	f2 c2 95 f2	7a 96 b9 43	59 35 80 7a	73 59 f6 7f
Chave Rodada 03:	3d 80 47 7d	47 16 fe 3e	1e 23 7e 44	6d 7a 88 3b
Chave Rodada 04:	ef 44 a5 41	a8 52 5b 7f	b6 71 25 3b	db 0b ad 00
Chave Rodada 05:	d4 d1 c6 f8	7c 83 9d 87	ca f2 b8 bc	11 f9 15 bc
Chave Rodada 06:	6d 88 a3 7a	11 0b 3e fd	db f9 86 41	ca 00 93 fd
Chave Rodada 07:	4e 54 f7 0e	5f 5f c9 f3	84 a6 4f b2	4e a6 dc 4f
Chave Rodada 08:	ea d2 73 21	b5 8d ba d2	31 2b f5 60	7f 8d 29 2f
Chave Rodada 09:	ac 77 66 f3	19 fa dc 21	28 d1 29 41	57 5c 00 6e
Chave Rodada 10:	d0 14 f9 a8	c9 ee 25 89	e1 3f 0c c8	b6 63 0c a6

Figura A.1: Sub-chaves provenientes do processo de Expansão da Chave

A figura A.2 apresenta, novamente, um esquemático do funcionamento do algoritmo. Nesta seção serão realizados os cálculos dos valores de entrada e da primeira rodada.

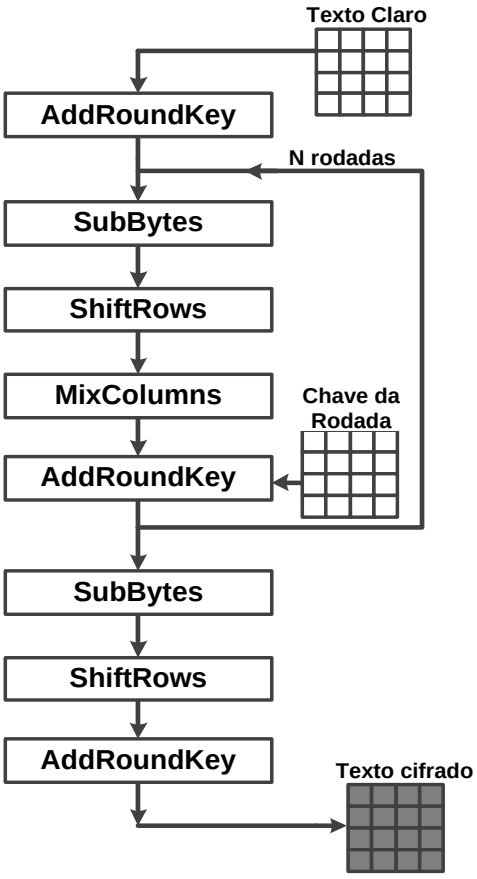


Figura A.2: Processo de Cifragem do algoritmo AES

A primeira operação a ser realizada é a função *AddRoundKey*. Essa função realiza uma operação Ou-Exclusivo entre a palavra de entrada e a chave inicial, que são:

Chave Inicial:	`3243F6A8885A308D313198A2E0370734`
Palavra Inicial:	`2B7E151628AED2A6ABF7158809CF4F3C`
Resultado:	`193DE3BEA0F4E22B9AC68D2AE9F84808`

O resultado da primeira operação será submetido à função *SubBytes*. A operação *SubBytes* realiza a substituição dos valores de entrada de acordo com a tabela A.2, que representa o S-Box do algoritmo AES. Deste modo, o valor `19` será substituído pelo valor presente na linha x = 1, coluna y = 9, que é `D4`.

Tabela A.2: S-Box [29]

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

O valor `19` será substituído pelo valor `D4`; o valor `3D` será substituído pelo valor `27`, e assim, sucessivamente, chegando ao seguinte resultado da função *SubBytes*:

Entrada:	`193DE3BEA0F4E22B9AC68D2AE9F84808`
Saída	`D42711AEE0BF98F1B8B45DE51E415230`

Após o cálculo da função *SubBytes*, seu resultado será utilizado como entrada da função *Shiftrows*. A figura A.3 apresenta as mudanças realizadas com o processamento da função *Shiftrows*. A figura A.3(a) representa o bloco de entrada e a figura (b) representa o bloco de saída.

D4	E0	B8	1E
27	BF	B4	41
11	98	5D	52
AE	F1	E5	30

(a)

D4	E0	B8	1E
BF	B4	41	27
5D	52	11	98
30	AE	F1	E5

(b)

Figura A.3: Execução da Função ShiftRows

Com relação às informações processadas nesta função, a primeira linha não sofre alteração; a segunda linha sofre apenas um deslocamento; a terceira linha sofre dois deslocamentos e a quarta linha sofre três deslocamentos. Temos os seguintes valores finais:

Entrada: `D42711AEE0BF98F1B8B45DE51E415230`
Saída `D4BF5D30E0B452AEB84111FL1E2798E5`

A função *MixColumns* realiza uma multiplicação de matrizes, onde os elementos do estado são considerados polinômios sobre GF(2^8). Para a realização da função *MixColumns*, utiliza-se outra matriz calculada a partir da matemática de Galois, que é formada por 4 linhas e 4 colunas. A figura A.4 será empregada para exemplificar a operação, onde o bloco (a) é a palavra de entrada e o bloco (b) é a matriz utilizada nesta função.

D4	E0	B8	1E
BF	B4	41	27
5D	52	11	98
30	AE	F1	E5

(a)

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

(b)

Figura A.4: Palavra e Constante utilizada para Função MixColumns

O conteúdo da primeira célula da primeira coluna do resultado é representado pelo cálculo:

$(D4 \cdot 02) \oplus (BF \cdot 03) \oplus (5D \cdot 01) \oplus (30 \cdot 01)$

1) Realizando o cálculo D4 · 02:

A representação binária do valor $D4_{16}$ é igual a 11010100_2.

É importante lembrar que a multiplicação de valores por x (02) pode ser desenvolvida como o deslocamento de 1 bit à esquerda, seguido da operação Ou-Exclusivo com 00011011_2, somente se o bit mais à esquerda antes do deslocamento seja igual a 1 [2]. Assim, temos:

$D4_{16} \cdot 02_{16} = 11010100_2 << 1$ (deslocamento à esquerda)

$= 10101000_2 \oplus 00011011_2$ (operação XOR, devido ao bit 1 mais à esquerda)

$= 10110011_2 = B3_{16}$

A partir daí, temos que o resultado da operação D4 · 02 é igual a B3.

2) Realizando o cálculo BF · 03:

A representação binária do valor BF é igual a 10111111_2.

De acordo com as informações apresentadas na operação anterior, existem alguns pontos que devem ser lembrados para a realização desta operação. O valor 03_{16} pode ser representado, em

sua forma binária, por 11_2, que é o resultado da operação (10 ⊕ 01) [2]. A partir destas informações, temos:

$BF_{16} \cdot 03_{16} = (10_2 \oplus 01_2) \cdot (10111111_2)$

$= (10111111_2 \cdot 10_2) \oplus (10111111_2 \cdot 01_2)$

$= (10111111_2 \cdot 10_2) \oplus 10111111_2$

$= (01111110_2 \oplus 00011011_2) \oplus 10111111_2$

$= 11011010_2 = DA_{16}$

A partir daí, temos que o resultado da operação BF · 03 é igual a DA_{16}.

3) Realizando os cálculos (5D · 01) e (30 · 01):

Estas operações realizam uma multiplicação por 1 e continuam da mesma maneira, como segue:

$5D = 0101\ 1101_2$ e $30 = 0011\ 0000_2$

Dando continuidade aos cálculos, temos:

$(D4 \cdot 02) \oplus (BF \cdot 03) \oplus (5D \cdot 01) \oplus (30 \cdot 01) =$

$= B3_{16} \oplus DA_{16} \oplus 5D_{16} \oplus 30_{16} =$

$= 10110011_2 \oplus 11011010_2 \oplus 01011101_2 \oplus 00110000_2 =$

$= 00000100_2 = 04_{16}$

Com isso, o valor da primeira célula da primeira coluna será:

$(D4 \cdot 02) \oplus (BF \cdot 03) \oplus (5D \cdot 01) \oplus (30 \cdot 01) = 04_{16}$

O conteúdo da segunda célula da primeira coluna do resultado é representado pelo cálculo:

(D4 · 01) ⊕ (BF · 02) ⊕ (5D · 03) ⊕ (30 · 01)

1) Realizando o cálculo BF · 02, temos:

$BF \cdot 02 = 10111111_2 << 1 =$

$= 01111110_2 \oplus 00011011_2 =$

$= 01100101_2 = 65_{16}$

2) Realizando o cálculo 5D · 03, temos:

$5D \cdot 03 = (01011101_2 \cdot 02) \oplus (01011101_2) =$

$= 10111010_2 \oplus 01011101_2 =$

$= 11100111_2 = E7_{16}$

Dando continuidade aos cálculos, temos:

$(D4 \cdot 01) \oplus (BF \cdot 02) \oplus (5D \cdot 03) \oplus (30 \cdot 01) =$

$= D4_{16} \oplus 65_{16} \oplus E7_{16} \oplus 30_{16} =$

$= 11010100_2 \oplus 01100101_2 \oplus 11100111_2 \oplus 00110000_2 =$

$= 01100110_2 = 66_{16}$

Realizando a mesma sequência de cálculos para as outras células e para as outras colunas, chegamos aos seguintes valores finais:

Entrada:	`D4BF5D30E0B452AEB84111FL1E2798E5`
Saída	`046681E5E0CB199A48F8D37A2806264C`

A última operação para encerrar esta rodada é a função *AddRoundKey*, que deverá realizar uma operação Ou-Exclusivo entre a saída da função *MixColumns* e a chave da rodada 1.

Chave da Rodada 1:	`A0FAFE1788542CB123A339392A6C7605`
Saída da função *MixColumns*:	`046681E5E0CB199A48F8D37A2806264C`
Resultado da 1ª Rodada:	`A49C7FF2689F352B6B5BEA43026A5049`

O resultado da primeira rodada será utilizado como entrada para a segunda rodada. Este valor será utilizado como entrada da função *SubBytes*, conforme apresentado na figura A.2.

Apêndice B

Funcionamento do algoritmo Twofish

Nesta seção será apresentado o funcionamento de uma rodada do algoritmo Twofish. Conforme apresentado anteriormente, o funcionamento do algoritmo é representado pelas figuras B.1 e B.2 [8, 30]. A figura B.1 apresenta o processo de cifragem de um bloco de 128 bits de texto plano. A figura B.2 apresenta o processo de expansão das chaves de tamanho 128 bits.

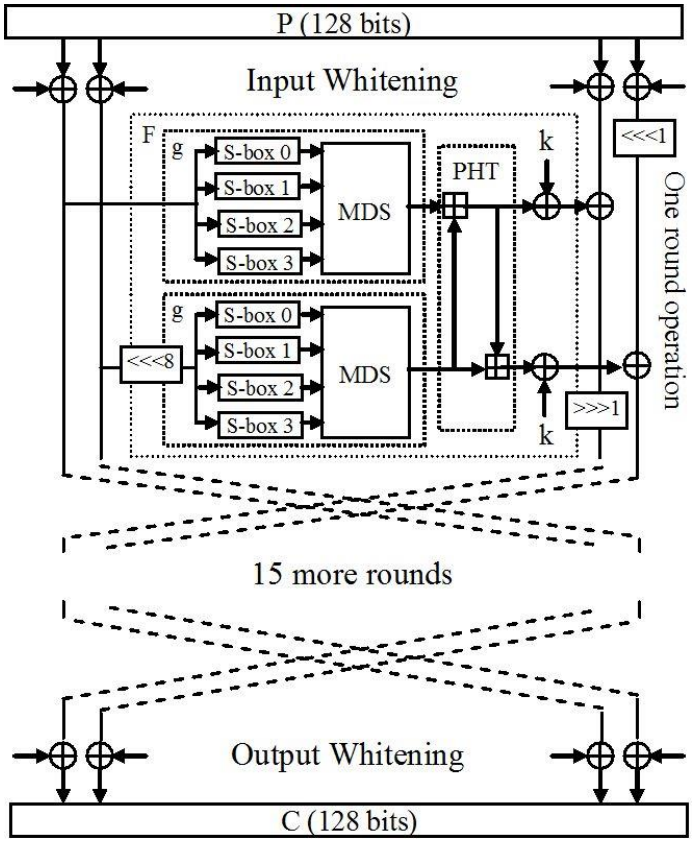


Figura B.1 - Processo de cifragem do algoritmo Twofish [5]

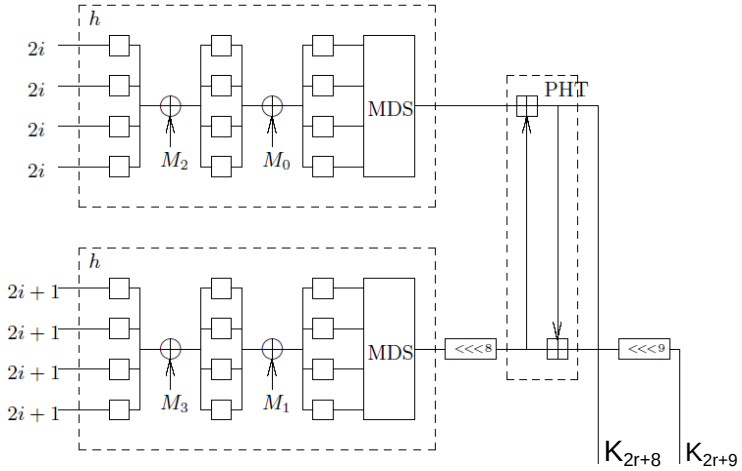


Figura B.2 - Expansão das chaves [5]

B.1 – Valores de entrada

O valor de entrada é lido de acordo com a convenção *little-endian.* Admita o seguinte valor de 128 bits, que representa o valor de entrada para o processo de encriptação, denominado "Texto Plano", já dividido em quatro partes de 32 bits: *0x*7654326BFEDCBA9889ABCDEF01234567.

Sejam as partes da informação de entrada:

P_0 = *0x*7654326B (parte menos significativa da informação);

P_1 = *0x*FEDCBA98;

P_2 = *0x*89ABCDEF;

P_3 = *0x*01234567 (parte mais significativa da informação).

A palavra P_0 é aquela que se encontra mais à esquerda na representação da figuraB.1. O processo de *input whitening* ocorre logo após a leitura e particionamento destes dados. As chaves K0 até K3 já foram previamente calculadas, são iguais e valem:

$Chave_0 = Chave_1 = Chave_2 = Chave_3$ = *0x*00000000

Os valores Q representam o resultado da operação após a execução do *input whitening*:

$Q_0 = Chave_0 \oplus P_0$ = *0x*7654326B

$Q_1 = Chave_1 \oplus P_1$ = *0x*FEDCBA98

$Q_2 = Chave_2 \oplus P_2$ = *0x*89ABCDEF

$Q_3 = Chave_3 \oplus P_3$ = *0x*01234567

Após a realização da operação de *input whitening*, os valores Q_0 e Q_1 serão utilizados como as entradas da rede de Feistel, nos blocos menos significativo e mais significativo, respectivamente. Os valores Q_0 e Q_1 serão utilizados sem modificações como entradas da próxima

rodada. Os valores Q_2 e Q_3 precisam aguardar o processamento da rede de Feistel para que possam prosseguir para a próxima rodada.

B.2 – S-Boxes

Após serem lidos pelos blocos da rede de Feistel, os valores Q_0 e Q_1 serão processados pelas funções g. O valor Q_0 é lido diretamente pela função g. O valor Q_1 sofre uma rotação à esquerda de 8 bits, transformando-se em R_1. Seja $R_1 = Q_1 << 8 =$ `0xDCBA98FE`.

B.3 – Função g

A função g para uma chave de 128 bits, tem sua configuração para os S-Boxes e funções q apresentada na figura B.3. O valor $Q_0 =$ `0x7654326B`. É dividido em 4 grupos de 8 bits, que são: `x = 0x6B`; `y = 0x32`; `z = 0x54`; e `w = 0x76`.

O valor $R_1 =$ `0xDCBA98FE`. É dividido em 4 grupos de 8 bits, que são: `a = 0xDC`; `b = 0xBA`; `c = 0x98`; e `d = 0xFE`. Admita que as chaves L_0 até L_3 já foram previamente calculadas e valem: $L_0 = L_1 = L_2 = L_3 =$ `0x00000000`.

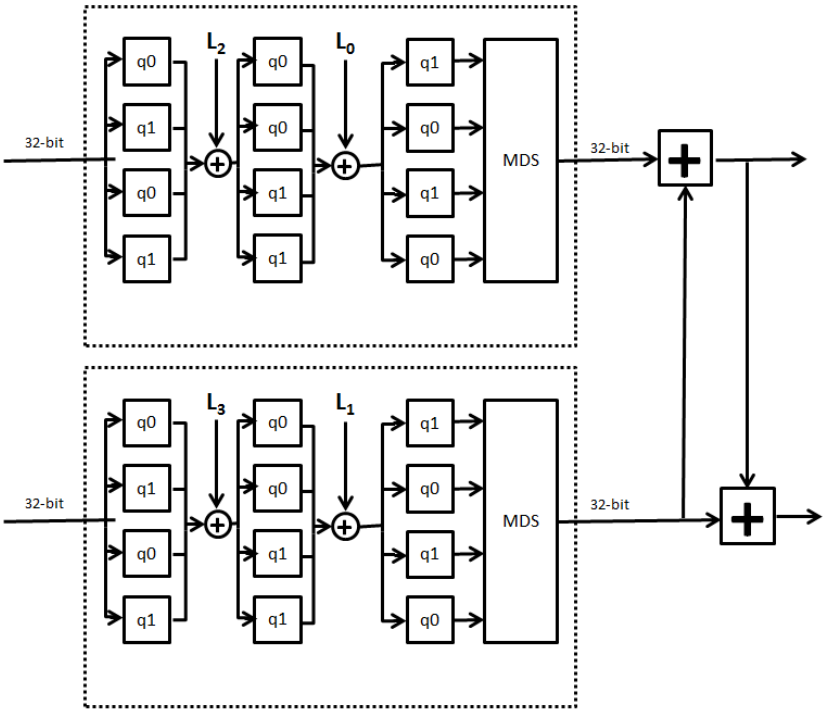


Figura B.3 – Função g para uma chave de 128 bits

Os valores **x**, **z**, **a** e **c** serão processados por funções q_0, enquanto os valores `y`, `w`, `c` e `d` serão processados por funções q_1, conforme apresentado na figura B.3. A função *q* funciona de acordo com a figura B.4 e a tabela B.1.

Tabela B.1: Parâmetros da função q

Entradas		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
q_0	t_0	8	1	7	D	6	F	3	2	0	B	5	9	E	C	A	4
	t_1	E	C	B	8	1	2	3	5	F	4	A	6	7	0	9	D
	t_2	B	A	5	E	6	D	9	0	C	8	F	3	2	4	7	1
	t_3	D	7	F	4	1	2	6	E	9	B	3	0	8	5	C	A
q_1	t_0	2	8	B	D	F	7	6	E	3	1	9	4	0	A	C	5
	t_1	1	E	2	B	4	C	3	7	6	D	A	5	F	9	0	8
	t_2	4	C	7	5	1	6	9	A	0	E	D	8	2	B	3	F
	t_3	B	9	5	1	C	3	D	E	6	4	7	F	2	0	8	A

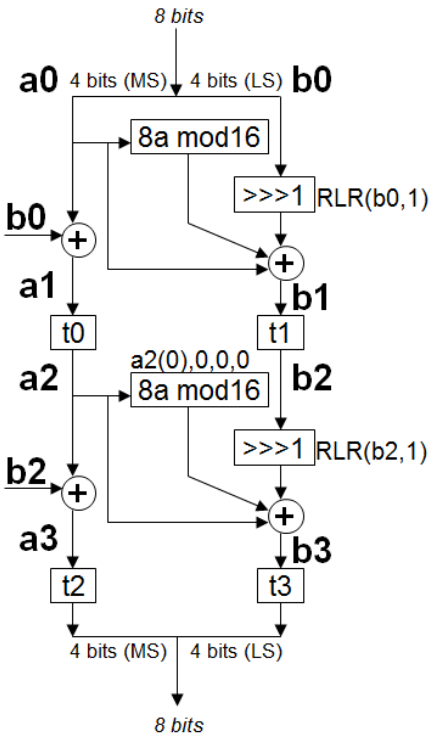


Figura B.4: Função q

O processamento do valor $x = 0x6B = 01101011_2$ ocorre da seguinte maneira:

1) O valor é dividido ao meio, sendo que a_0 recebe o *nibble* mais significativo e b_0 recebe o *nibble* menos significativo.

2) São realizados os cálculos para que se encontrem a_1 e b_1:

$$a_1 = a_0 \oplus b_0 = 0110_2 \oplus 1011_2 = 1101_2$$

$$b_1 = a_0 \oplus [(8 * a_0) \bmod 16] \oplus (b_0 >> 1)$$

$$= 0110_2 \oplus 0000_2 \oplus 1101_2 = 1011_2$$

3) Para que se encontrem os valores de a_2 e b_2 é necessário utilizar a tabela B.1, sabendo que o valor de q é igual a 0.

$$a_2 = t0(a_1) = t0(1101_2) = t0(0xD) = 0xC = 1100_2$$

$$b_2 = t1(b_1) = t1(1011_2) = t1(0xB) = 0x6 = 0110_2$$

4) São realizados os cálculos para que se encontrem a_3 e b_3:

$a_3 = a_2 \oplus b_2 = 1100_2 \oplus 0110_2 = 1010_2$

$b_3 = a_2 \oplus [(8 * a_2) \bmod 16] \oplus (b_2 >> 1)$

$= 1100_2 \oplus 0000_2 \oplus 0011_2 = 1111_2$

5) Para que se encontrem os valores de a_4 e b_4 é necessário utilizar a tabela B.1, sabendo que o valor de q é igual a 0.

$a_4 = t2(a_3) = t2(1010_2) = t2(0xA) = 0xD = 1101_2$

$b_4 = t3(b_3) = t3(1111_2) = t3(0xF) = 0xA = 1010_2$

O resultado do processamento do valor de x(*0x*6B) é igual a x_1 = 0xFA. No caso de x, ele foi processado por um bloco q_0. Caso fosse um bloco q_1, a única mudança seria a leitura da tabela nas linhas t_n de q_1.

Após o processamento dos primeiros blocos *q*, os valores são agrupados novamente para a operação XOR (Ou-Exclusivo) com a primeira chave S.

Os valores intermediários para o processamento S-Box de Q_0 são: 0x7654326B, 0xAE2E70FA, 0x9F2D1863 e 0xFED534FF. Para R_1 são: 0XDCBA98FE, 0x535D181C, 0x7C0D34AF e 0x44D79D84. Os valores de saída dos S-Boxes são utilizados para o cálculo da função MDS.

Após o processamento da função MDS os valores relativos a Q_0 e R_1 são, respectivamente, R_0 = 0xC0039713 e S_1 = 0xFCA728C7.

B.4 – PHT

Os valores de entrada para a função PHT R_0 = 0xC0039713 e S_1 = 0xFCA728C7. Os valores de saída são, respectivamente, S_0 = 0xBCAABFDA e T_1 = 0XB951E8A1.

O próximo passo é o processamento com as chaves da rodada, que será calculado do mesmo modo que a função PHT, soma em módulo 2^{32}. As chaves das rodadas são representadas pelos índices 2r+8 e 2r+9 para a informação menos significativa e mais significativa, respectivamente. Na primeira rodada (r = 0) serão utilizadas as chaves de índice 8 e 9, definidas aleatória e arbitrariamente, neste caso:

$Chave_8$ = *0x12345678;* $Chave_9$ = *0xABCDEF01;*

Os valores após o cálculo com as chaves relativos a S_0 e T_1 são, respectivamente, T_0 = 0xCEDF1652 e U_1 = 0x651FD7A2.

Com isso, os valores de Q_0 = *0x*7654326B e R_1 = *0x*DCBA98FE, após serem processados pela rede de Feistel, são iguais a T_0 = 0xCEDF1652 e U_1 = 0x651FD7A2, respectivamente.

B.5 – Final da rodada

Voltando à análise da figura A.1, os valores encontrados na rede de Feistel serão processados junto aos valores iniciais Q_2 e Q_3.

O valor Q_3 (*0x*01234567) sofre uma alteração antes do processamento: uma rotação à esquerda, de um bit, transformando em R_3 = *0x*02468ACE.

$R_2 = Q_2 \oplus T_0 =$ *0x*89ABCDEF $\oplus$ *0x*CEDF1652 = *0x*4774DBBD

$S_3 = R_3 \oplus U_1 =$ *0x*02468ACE $\oplus$ *0x*651FD7A2= *0x*67595D6C

O valor R_2 (*0x*4774DBBD) sofre uma alteração antes do processamento: uma rotação à direita, de um bit, transformando em S_2 = *0x*A3BA6DDE.

A partir deste ponto, temos os valores resultantes da primeira rodada (r = 0), que são:

T_0 = 0xCEDF1652 (valor relativo a P_0)

U_1 = 0x651FD7A2 (valor relativo a P_1)

S_2 = 0xA3BA6DDE (valor relativo a P_2)

S_3 = 0x67595D6C (valor relativo a P_3)

Para a próxima rodada, os dois valores mais significativos são trocados com os dois valores menos significativos conforme a figura B.1, da seguinte maneira:

Q_0 recebe S_2;

Q_1 recebe S_3;

Q_2 recebe Q_0; e

Q_3 recebe Q_1.

Com os devidos processamentos realizados para esta rodada (r=0), é necessário voltar ao item B.2 e continuar a executar todos os passos para completar a segunda rodada (r=1). Este processo se repetirá até que se alcance a 16ª rodada (r=15). Após a 16ª rodada, deverá ser realizada um última troca das palavras (*switch*) e o processo de *whitening* de saída.

Após a realização destas operações o texto de saída estará pronto, criptografado.

Apêndice C

Descrição de hardware do algoritmo AES

Este anexo apresenta uma parte significativa dos códigos desenvolvidos e utilizados neste trabalho.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY sbox IS
PORT(
   in_sbox    :    IN  std_logic_vector(7 downto 0);
   out_sbox   :    OUT std_logic_vector(7 downto 0)
   );
END sbox;

-- Uma parte do código foi retirada, não prejudicando o entendimento do mesmo.

ARCHITECTURE sbox_arch OF sbox IS

BEGIN

        WITH in_sbox(7 downto 0) SELECT
                out_sbox(7 downto 0) <=

                   "01100011" WHEN "00000000", --(X"63")
                   "01111100" WHEN "00000001", --(X"7C")
                   "01110111" WHEN "00000010", --(X"77")
                   "01111011" WHEN "00000011", --(X"7B")
                   "11110010" WHEN "00000100", --(X"F2")
                   "01101011" WHEN "00000101", --(X"6B")
                   "01101111" WHEN "00000110", --(X"6F")
                   "11000101" WHEN "00000111", --(X"C5")
                   "00110000" WHEN "00001000", --(X"30")
                   "00000001" WHEN "00001001", --(X"01")
                   "01100111" WHEN "00001010", --(X"67")
                   "00101011" WHEN "00001011", --(X"2B")
                   "11111110" WHEN "00001100", --(X"FE")
                   "11010111" WHEN "00001101", --(X"D7")
                   "10101011" WHEN "00001110", --(X"AB")
                   "01110110" WHEN "00001111", --(X"76")

                   "11001010" WHEN "00010000", --(X"CA")
                   "10000010" WHEN "00010001", --(X"82")
                   "11001001" WHEN "00010010", --(X"C9")
                   "01111101" WHEN "00010011", --(X"7D")
                   "11111010" WHEN "00010100", --(X"FA")
                   "01011001" WHEN "00010101", --(X"59")
                   "01000111" WHEN "00010110", --(X"47")
                   "11110000" WHEN "00010111", --(X"F0")
                   "10101101" WHEN "00011000", --(X"AD")
                   "11010100" WHEN "00011001", --(X"D4")
                   "10100010" WHEN "00011010", --(X"A2")
                   "10101111" WHEN "00011011", --(X"AF")
                   "10011100" WHEN "00011100", --(X"9C")
                   "10100100" WHEN "00011101", --(X"A4")
```

```
                    "01110010" WHEN "00011110", --(X"72")
                    "11000000" WHEN "00011111", --(X"C0")

                    "10111010" WHEN "11000000", --(X"BA")
                    "01111000" WHEN "11000001", --(X"78")
                    "00100101" WHEN "11000010", --(X"25")
                    "00101110" WHEN "11000011", --(X"2E")
                    "00011100" WHEN "11000100", --(X"1C")
                    "10100110" WHEN "11000101", --(X"A6")
                    "10110100" WHEN "11000110", --(X"B4")
                    "11000110" WHEN "11000111", --(X"C6")
                    "11101000" WHEN "11001000", --(X"E8")
                    "11011101" WHEN "11001001", --(X"DD")
                    "01110100" WHEN "11001010", --(X"74")
                    "00011111" WHEN "11001011", --(X"1F")
                    "01001011" WHEN "11001100", --(X"4B")
                    "10111101" WHEN "11001101", --(X"BD")
                    "10001011" WHEN "11001110", --(X"8B")
                    "10001010" WHEN "11001111", --(X"8A")

                    "XXXXXXXX" WHEN OTHERS;

END sbox_arch;

--=============================================================
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY inv_sbox IS
PORT(
   in_sbox    :    IN  std_logic_vector(7 downto 0);
   out_sbox   :    OUT std_logic_vector(7 downto 0)
   );
END inv_sbox;

-- Uma parte do código foi retirada, não prejudicando o entendimento do mesmo.

ARCHITECTURE inv_sbox_arch OF inv_sbox IS

BEGIN

        WITH in_sbox(7 downto 0) SELECT

                out_sbox(7 downto 0) <=

                "01010010" WHEN "00000000", --(X"52")
                "00001001" WHEN "00000001", --(X"09")
                "01101010" WHEN "00000010", --(X"6a")
                "11010101" WHEN "00000011", --(X"D5")
                "00110000" WHEN "00000100", --(X"30")
                "00110110" WHEN "00000101", --(X"36")
                "10100101" WHEN "00000110", --(X"A5")
                "00111000" WHEN "00000111", --(X"38")
                "10111111" WHEN "00001000", --(X"BF")
                "01000000" WHEN "00001001", --(X"40")
                "10100011" WHEN "00001010", --(X"A3")
                "10011110" WHEN "00001011", --(X"9E")
                "10000001" WHEN "00001100", --(X"81")
                "11110011" WHEN "00001101", --(X"F3")
                "11010111" WHEN "00001110", --(X"D7")
                "11111011" WHEN "00001111", --(X"FB")

                "01111100" WHEN "00010000", --(X"7C")
                "11100011" WHEN "00010001", --(X"E3")
                "00111001" WHEN "00010010", --(X"39")
                "10000010" WHEN "00010011", --(X"82")
                "10011011" WHEN "00010100", --(X"9B")
                "00101111" WHEN "00010101", --(X"2F")
                "11111111" WHEN "00010110", --(X"FF")
                "10000111" WHEN "00010111", --(X"87")
                "00110100" WHEN "00011000", --(X"34")
```

```
                "10001110" WHEN "00011001", --(X"8E")
                "01000011" WHEN "00011010", --(X"43")
                "01000100" WHEN "00011011", --(X"44")
                "11000100" WHEN "00011100", --(X"C4")
                "11011110" WHEN "00011101", --(X"DE")
                "11101001" WHEN "00011110", --(X"E9")
                "11001011" WHEN "00011111", --(X"CB")

                "00010111" WHEN "11110000", --(X"17")
                "00101011" WHEN "11110001", --(X"2B")
                "00000100" WHEN "11110010", --(X"04")
                "01111110" WHEN "11110011", --(X"7E")
                "10111010" WHEN "11110100", --(X"BA")
                "01110111" WHEN "11110101", --(X"77")
                "11010110" WHEN "11110110", --(X"D6")
                "00100110" WHEN "11110111", --(X"26")
                "11100001" WHEN "11111000", --(X"E1")
                "01101001" WHEN "11111001", --(X"69")
                "00010100" WHEN "11111010", --(X"14")
                "01100011" WHEN "11111011", --(X"63")
                "01010101" WHEN "11111100", --(X"55")
                "00100001" WHEN "11111101", --(X"21")
                "00001100" WHEN "11111110", --(X"0C")
                "01111101" WHEN "11111111", --(X"7D")

                "XXXXXXXX" WHEN OTHERS;
END inv_sbox_arch;

--============================================================

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY shift_rows IS
PORT(
   in_shiftrows     :   IN  std_logic_vector(127 downto 0);
   out_shiftrows    :   OUT std_logic_vector(127 downto 0)
   );
END shift_rows;

-- Uma parte do código foi retirada, não prejudicando o entendimento do mesmo.

ARCHITECTURE shift_rows_arch OF shift_rows IS

                TYPE matrix_index IS array (15 downto 0) OF std_logic_vector(7 downto 0);
                SIGNAL matrix1, matrix2 : matrix_index;

BEGIN
                matrix2(0)  <=  matrix1(0);
                matrix2(1)  <=  matrix1(5);
                matrix2(2)  <=  matrix1(10);
                matrix2(3)  <=  matrix1(15);

                matrix2(4)  <=  matrix1(4);
                matrix2(5)  <=  matrix1(9);
                matrix2(6)  <=  matrix1(14);
                matrix2(7)  <=  matrix1(3);

                matrix2(8)  <=  matrix1(8);
                matrix2(9)  <=  matrix1(13);
                matrix2(10) <=  matrix1(2);
                matrix2(11) <=  matrix1(7);

                matrix2(12) <=  matrix1(12);
                matrix2(13) <=  matrix1(1);
                matrix2(14) <=  matrix1(6);
                matrix2(15) <=  matrix1(11);

END shift_rows_arch;
ENTITY inv_shift_rows IS
        PORT(
```

```
                in_shiftrows     :   IN  std_logic_vector(127 downto 0);
                out_shiftrows    :   OUT std_logic_vector(127 downto 0)
        );
END inv_shift_rows;

-- Uma parte do código foi retirada, não prejudicando o entendimento do mesmo.

ARCHITECTURE inv_shift_rows_arch OF inv_shift_rows IS

                TYPE matrix_index IS array (15 downto 0) OF std_logic_vector(7 downto 0);
                SIGNAL matrix1, matrix2 : matrix_index;

BEGIN
                matrix1(0)  <=  matrix2(0);
                matrix1(5)  <=  matrix2(1) ;
                matrix1(10) <=  matrix2(2) ;
                matrix1(15) <=  matrix2(3) ;

                matrix1(4)  <=  matrix2(4);
                matrix1(9)  <=  matrix2(5);
                matrix1(14) <=  matrix2(6);
                matrix1(3)  <=  matrix2(7);

                matrix1(8)  <=  matrix2(8);
                matrix1(13) <=  matrix2(9);
                matrix1(2)  <=  matrix2(10);
                matrix1(7)  <=  matrix2(11);

                matrix1(12) <=  matrix2(12);
                matrix1(1)  <=  matrix2(13);
                matrix1(6)  <=  matrix2(14);
                matrix1(11) <=  matrix2(15);

END inv_shift_rows_arch;


--============================================================


ENTITY mix_column IS
PORT(
   in_mixcolumns     :   IN  std_logic_vector(127 downto 0);
   out_mixcolumns    :   OUT std_logic_vector(127 downto 0)
   );
END mix_column;

-- Uma parte do código foi retirada, não prejudicando o entendimento do mesmo.

ARCHITECTURE mix_column_arch OF mix_column IS

                TYPE matrix_index is array (15 downto 0) of std_logic_vector(7 downto 0);
                TYPE shift_index is array (15 downto 0) of std_logic_vector(8 downto 0);
                SIGNAL shiftby_2, shiftby_3, xored : shift_index;
                SIGNAL matrix, matrix_out, multby_2, multby_3 : matrix_index;
BEGIN
                matrix_out(0)  <= multby_2(0)  XOR multby_3(1)  XOR matrix(2)  XOR matrix(3);
                matrix_out(4)  <= multby_2(4)  XOR multby_3(5)  XOR matrix(6)  XOR matrix(7);
                matrix_out(8)  <= multby_2(8)  XOR multby_3(9)  XOR matrix(10) XOR matrix(11);
                matrix_out(12) <= multby_2(12) XOR multby_3(13) XOR matrix(14) XOR matrix(15);


                matrix_out(1)  <= matrix(0)  XOR multby_2(1)  XOR multby_3(2)  XOR matrix(3);
                matrix_out(5)  <= matrix(4)  XOR multby_2(5)  XOR multby_3(6)  XOR matrix(7);
                matrix_out(9)  <= matrix(8)  XOR multby_2(9)  XOR multby_3(10) XOR matrix(11);
                matrix_out(13) <= matrix(12) XOR multby_2(13) XOR multby_3(14) XOR matrix(15);

                matrix_out(2)  <= matrix(0)  XOR matrix(1)  XOR multby_2(2)  XOR multby_3(3);
                matrix_out(6)  <= matrix(4)  XOR matrix(5)  XOR multby_2(6)  XOR multby_3(7);
                matrix_out(10) <= matrix(8)  XOR matrix(9)  XOR multby_2(10) XOR multby_3(11);
                matrix_out(14) <= matrix(12) XOR matrix(13) XOR multby_2(14) XOR multby_3(15);

                matrix_out(3)  <= multby_3(0)  XOR matrix(1)  XOR matrix(2)  XOR multby_2(3);
                matrix_out(7)  <= multby_3(4)  XOR matrix(5)  XOR matrix(6)  XOR multby_2(7);
```

```
                matrix_out(11) <= multby_3(8)  XOR matrix(9)  XOR matrix(10) XOR multby_2(11);
                matrix_out(15) <= multby_3(12) XOR matrix(13) XOR matrix(14) XOR multby_2(15);

END mix_column_arch;

--============================================================

ENTITY inv_mix_column IS
PORT(
   in_mixcolumns     :   IN  std_logic_vector(127 downto 0);
   out_mixcolumns    :   OUT std_logic_vector(127 downto 0)
   );
END inv_mix_column;

-- Uma parte do código foi retirada, não prejudicando o entendimento do mesmo.

ARCHITECTURE inv_mix_column_arch OF inv_mix_column IS

                TYPE matrix_index is array (15 downto 0) of std_logic_vector(7 downto 0);
                TYPE shift_index is array (15 downto 0) of std_logic_vector(8 downto 0);
                SIGNAL matrix, matrix_out, multby_0e, multby_0b,multby_0d,multby_09 : matrix_index;
BEGIN
                matrix_out(0)  <= multby_0e(0)  XOR multby_0b(1)  XOR multby_0d(2)  XOR multby_09(3);
                matrix_out(4)  <= multby_0e(4)  XOR multby_0b(5)  XOR multby_0d(6)  XOR multby_09(7);
                matrix_out(8)  <= multby_0e(8)  XOR multby_0b(9)  XOR multby_0d(10) XOR multby_09(11);
                matrix_out(12) <= multby_0e(12) XOR multby_0b(13) XOR multby_0d(14) XOR multby_09(15);

                matrix_out(1)  <= multby_09(0)  XOR multby_0e(1)  XOR multby_0b(2)  XOR multby_0d(3);
                matrix_out(5)  <= multby_09(4)  XOR multby_0e(5)  XOR multby_0b(6)  XOR multby_0d(7);
                matrix_out(9)  <= multby_09(8)  XOR multby_0e(9)  XOR multby_0b(10) XOR multby_0d(11);
                matrix_out(13) <= multby_09(12) XOR multby_0e(13) XOR multby_0b(14) XOR multby_0d(15);

                matrix_out(2)  <= multby_0d(0)  XOR multby_09(1)  XOR multby_0e(2)  XOR multby_0b(3);
                matrix_out(6)  <= multby_0d(4)  XOR multby_09(5)  XOR multby_0e(6)  XOR multby_0b(7);
                matrix_out(10) <= multby_0d(8)  XOR multby_09(9)  XOR multby_0e(10) XOR multby_0b(11);
                matrix_out(14) <= multby_0d(12) XOR multby_09(13) XOR multby_0e(14) XOR multby_0b(15);

                matrix_out(3)  <= multby_0b(0)  XOR multby_0d(1)  XOR multby_09(2)  XOR multby_0e(3);
                matrix_out(7)  <= multby_0b(4)  XOR multby_0d(5)  XOR multby_09(6)  XOR multby_0e(7);
                matrix_out(11) <= multby_0b(8)  XOR multby_0d(9)  XOR multby_09(10) XOR multby_0e(11);
                matrix_out(15) <= multby_0b(12) XOR multby_0d(13) XOR multby_09(14) XOR multby_0e(15);

END inv_mix_column_arch;
```

Apêndice D

Descrição de hardware do algoritmo Twofish

Este anexo apresenta uma parte significativa dos códigos desenvolvidos e utilizados neste trabalho.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity sBox is – Traditional S-Box, g/h function without MDS
	port(
		clk		: IN std_logic;
		rst		: IN std_logic;
		in_load	: IN std_logic;
		in_data	: IN std_logic_vector (31 DOWNTO 0);
		in_L0	: IN std_logic_vector(31 DOWNTO 0); -- keys S0 / M0
		in_L1	: IN std_logic_vector(31 DOWNTO 0); -- keys S1 / M1
		in_L2	: IN std_logic_vector(31 DOWNTO 0); -- keys S2 / M2
		in_L3	: IN std_logic_vector(31 DOWNTO 0); -- keys S3 / M3
		out_cont : OUT std_logic_vector(3 DOWNTO 0);
		out_data	: OUT std_logic_vector(31 DOWNTO 0);
		out_ready : OUT std_logic
	);
end sBox;

architecture sBox_behav of sBox is
	component q_unit port(
		clk		: IN std_logic;
		rst		: IN std_logic;
		load		: IN std_logic;
		in_q		: IN std_logic;
		in_value : IN std_logic_vector(7 DOWNTO 0);
		out_ready: OUT std_logic;
		 out_q	: OUT std_logic_vector(7 DOWNTO 0)
	);
	end component;

	SIGNAL sig_L0 : std_logic_vector(31 DOWNTO 0); -- keySchedule S0
	SIGNAL sig_L1 : std_logic_vector(31 DOWNTO 0); -- keySchedule S1
```

```
	SIGNAL sig_in_load : std_logic;
	SIGNAL sig_out_ready : std_logic;
	SIGNAL sig_out_data : std_logic_vector(31 DOWNTO 0);
	SIGNAL sig_out_cont : std_logic_vector(3 DOWNTO 0);
	SIGNAL sig_in_qA0, sig_in_qB0, sig_in_qC0, sig_in_qD0	: std_logic;
	SIGNAL sig_in_valueA0, sig_in_valueB0, sig_in_valueC0, sig_in_valueD0: std_logic_vector(7 DOWNTO 0);
	SIGNAL sig_out_readyA0, sig_out_readyB0, sig_out_readyC0, sig_out_readyD0	: std_logic;
	SIGNAL sig_out_qA0, sig_out_qB0, sig_out_qC0, sig_out_qD0 : std_logic_vector(7 DOWNTO 0);
	SIGNAL sig_in_qA1, sig_in_qB1, sig_in_qC1, sig_in_qD1	: std_logic;
	SIGNAL sig_in_valueA1, sig_in_valueB1, sig_in_valueC1, sig_in_valueD1: std_logic_vector(7 DOWNTO 0);
	SIGNAL sig_out_readyA1, sig_out_readyB1, sig_out_readyC1, sig_out_readyD1	: std_logic;
	SIGNAL sig_out_qA1, sig_out_qB1, sig_out_qC1, sig_out_qD1 : std_logic_vector(7 DOWNTO 0);
	SIGNAL sig_in_qA2, sig_in_qB2, sig_in_qC2, sig_in_qD2	: std_logic;
	SIGNAL sig_in_valueA2, sig_in_valueB2, sig_in_valueC2, sig_in_valueD2: std_logic_vector(7 DOWNTO 0);
	SIGNAL sig_out_readyA2, sig_out_readyB2, sig_out_readyC2, sig_out_readyD2	: std_logic;
	SIGNAL sig_out_qA2, sig_out_qB2, sig_out_qC2, sig_out_qD2 : std_logic_vector(7 DOWNTO 0);
	SIGNAL sig_in_rstQ, sig_inLoadQ0, sig_inLoadQ1, sig_inLoadQ2	: std_logic;

	type state_sBox is (load, st1_0, st1_1, st1_2, st2_0, st2_1, st2_2, st3_0, st3_1, st3_2, stResult);
	signal presentState, nextState : state_sBox;

begin
	-- Last state of S-Box q functions
	QA0: q_unit port map( --LSByte
		clk		=> clk,
		rst		=> sig_in_rstQ,
		load		=> sig_inLoadQ0,
		in_q		=> sig_in_qA0,
		in_value => sig_in_valueA0,
		out_ready=> sig_out_readyA0,
		 out_q	=> sig_out_qA0
	);

	QB0: q_unit port map(
		clk		=> clk,
		rst		=> sig_in_rstQ,
		load		=> sig_inLoadQ0,
		in_q		=> sig_in_qB0,
		in_value => sig_in_valueB0,
		out_ready=> sig_out_readyB0,
		 out_q	=> sig_out_qB0
	);

	QC0: q_unit port map(
		clk		=> clk,
		rst		=> sig_in_rstQ,
		load		=> sig_inLoadQ0,
		in_q		=> sig_in_qC0,
		in_value => sig_in_valueC0,
```

```
                out_ready=> sig_out_readyC0,
                out_q    => sig_out_qC0
        );
        QD0: q_unit port map( --MSByte
                clk              => clk,
                rst              => sig_in_rstQ,
                load             => sig_inLoadQ0,
                in_q             => sig_in_qD0,
                in_value => sig_in_valueD0,
                out_ready=> sig_out_readyD0,
                out_q    => sig_out_qD0
        );
-----------------------------------------
        QA1: q_unit port map( --LSByte
                clk              => clk,
                rst              => sig_in_rstQ,
                load             => sig_inLoadQ1,
                in_q             => sig_in_qA1,
                in_value => sig_in_valueA1,
                out_ready=> sig_out_readyA1,
                 out_q   => sig_out_qA1
        );

        QB1: q_unit port map(
                clk              => clk,
                rst              => sig_in_rstQ,
                load             => sig_inLoadQ1,
                in_q             => sig_in_qB1,
                in_value => sig_in_valueB1,
                out_ready=> sig_out_readyB1,
                out_q    => sig_out_qB1
        );

        QC1: q_unit port map(
                clk              => clk,
                rst              => sig_in_rstQ,
                load             => sig_inLoadQ1,
                in_q             => sig_in_qC1,
                in_value => sig_in_valueC1,
                out_ready=> sig_out_readyC1,
                out_q    => sig_out_qC1
        );

        QD1: q_unit port map( --MSByte
                clk              => clk,
                rst              => sig_in_rstQ,
                load             => sig_inLoadQ1,
                in_q             => sig_in_qD1,
                in_value => sig_in_valueD1,
```

```
        out_ready=> sig_out_readyD1,
        out_q    => sig_out_qD1
);
QA2: q_unit port map( --LSByte
        clk              => clk,
        rst              => sig_in_rstQ,
        load             => sig_inLoadQ2,
        in_q             => sig_in_qA2,
        in_value => sig_in_valueA2,
        out_ready=> sig_out_readyA2,
        out_q    => sig_out_qA2
);

QB2: q_unit port map(
        clk              => clk,
        rst              => sig_in_rstQ,
        load             => sig_inLoadQ2,
        in_q             => sig_in_qB2,
        in_value => sig_in_valueB2,
        out_ready=> sig_out_readyB2,
        out_q    => sig_out_qB2
);

QC2: q_unit port map(
        clk              => clk,
        rst              => sig_in_rstQ,
        load             => sig_inLoadQ2,
        in_q             => sig_in_qC2,
        in_value => sig_in_valueC2,
        out_ready=> sig_out_readyC2,
        out_q    => sig_out_qC2
);

QD2: q_unit port map( --MSByte
        clk              => clk,
        rst              => sig_in_rstQ,
        load             => sig_inLoadQ2,
        in_q             => sig_in_qD2,
        in_value => sig_in_valueD2,
        out_ready=> sig_out_readyD2,
        out_q    => sig_out_qD2
);
sig_in_load <= in_load;

procMain: process(clk, rst)
begin
        if (rst = '1') then
                presentState <= load;
        elsif (clk'event and clk = '1') then
```

```
                presentState <= nextState;
        end if;
end process;
procStates: process(presentState, clk)
begin
        if (sig_in_load = '1') then
                sig_out_cont <= "0000";
                sig_in_rstQ <= '1';
                sig_in_valueA2 <= in_data(7 downto 0);
                sig_in_valueB2 <= in_data(15 downto 8);
                sig_in_valueC2 <= in_data(23 downto 16);
                sig_in_valueD2 <= in_data(31 downto 24);
                sig_L0 <= in_L0;
                sig_L1 <= in_L1;
                sig_out_ready <= '0';
                sig_out_data <= (others=>'0');
                nextState <= load;
        else
                case presentState is
                        when load =>
                                sig_in_qA2 <= '0';
                                sig_in_qB2 <= '1';
                                sig_in_qC2 <= '0';
                                sig_in_qD2 <= '1';
                                sig_inLoadQ2 <= '1';
                                sig_in_rstQ <= '0';
                                sig_out_cont <= "0001";
                                nextState <= st1_0;
                        ---------------------------------------
                        when st1_0 =>
                                nextState <= st1_1;
                                sig_inLoadQ2 <= '0';
                        ---------------------------------------
                        when st1_1 =>
                                if (sig_out_readyA2 = '1' and sig_out_readyB2 = '1' and
                                        sig_out_readyC2 = '1' and sig_out_readyD2 = '1') then
                                        nextState <= st1_2;
                                else
                                        nextState <= st1_1;
                                end if;
                                sig_out_cont <= "0010";
                sig_out_data <= sig_in_valueD2 & sig_in_valueC2 & sig_in_valueB2 & sig_in_valueA2;
                        ---------------------------------------
                        when st1_2 =>
                                sig_out_cont <= "0011";
                                sig_inLoadQ1 <= '1';
                                nextState <= st2_0;
                                sig_in_valueA1 <= sig_out_qA2 xor sig_L1(7 downto 0);
                                sig_in_valueB1 <= sig_out_qB2 xor sig_L1(15 downto 8);
```

```
                sig_in_valueC1 <= sig_out_qC2 xor sig_L1(23 downto 16);
                sig_in_valueD1 <= sig_out_qD2 xor sig_L1(31 downto 24);
                sig_in_qA1 <= '0';
                sig_in_qB1 <= '0';
                sig_in_qC1 <= '1';
                sig_in_qD1 <= '1';
        sig_out_data <= sig_out_qD1 & sig_out_qC1 & sig_out_qB1 & sig_out_qA1;
        ---------------------------------------
        when st2_0 =>
                nextState <= st2_1;
                sig_inLoadQ1 <= '0';
sig_out_data <= sig_in_valueD1 & sig_in_valueC1 & sig_in_valueB1 & sig_in_valueA1;
        ---------------------------------------
        when st2_1 =>
                if (sig_out_readyA1 = '1' and sig_out_readyB1 = '1' and
                        sig_out_readyC1 = '1' and sig_out_readyD1 = '1') then
                        nextState <= st2_2;
                else
                        nextState <= st2_1;
                end if;
                sig_out_cont <= "0100";
        ---------------------------------------
        when st2_2 =>
                sig_out_cont <= "0101";
                sig_inLoadQ0 <= '1';
                nextState <= st3_0;
                sig_in_valueA0 <= sig_out_qA1 xor sig_L0(7 downto 0);
                sig_in_valueB0 <= sig_out_qB1 xor sig_L0(15 downto 8);
                sig_in_valueC0 <= sig_out_qC1 xor sig_L0(23 downto 16);
                sig_in_valueD0 <= sig_out_qD1 xor sig_L0(31 downto 24);
                sig_in_qA0 <= '1';
                sig_in_qB0 <= '0';
                sig_in_qC0 <= '1';
                sig_in_qD0 <= '0';
        sig_out_data <= sig_out_qD0 & sig_out_qC0 & sig_out_qB0 & sig_out_qA0;
        ---------------------------------------
        when st3_0 =>
                nextState <= st3_1;
                sig_inLoadQ0 <= '0';
        sig_out_data <= sig_out_qD0 & sig_out_qC0 & sig_out_qB0 & sig_out_qA0;
        ---------------------------------------
        when st3_1 =>
                if (sig_out_readyA0 = '1' and sig_out_readyB0 = '1' and
                        sig_out_readyC0 = '1' and sig_out_readyD0 = '1') then
                        nextState <= st3_2;
                else
                        nextState <= st3_1;
                end if;
                sig_out_cont <= "0110";
```

```
                                    ---------------------------------------
                                    when st3_2 =>
                                            sig_out_cont <= "0111";
                                            nextState <= stResult;
                                    sig_out_data <= sig_out_qD0 & sig_out_qC0 & sig_out_qB0 & sig_out_qA0;
                                    ---------------------------------------
                                    when stResult =>
                                            sig_out_cont <= "1100";
                                            sig_out_ready <= '1';
                                    sig_out_data <= sig_out_qD0 & sig_out_qC0 & sig_out_qB0 & sig_out_qA0;
                                            nextState <= stResult;
                            end case;
                    end if;
  end process;
        out_ready <= sig_out_ready;
        out_data <= sig_out_data;
        out_cont <= sig_out_cont;
end sBox_behav;
--============================================================
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity sBox_tb is
end sBox_tb;

architecture testbench of sBox_tb is
        COMPONENT sBox
        port(
                clk             : IN std_logic;
                rst             : IN std_logic;
                in_load  : IN std_logic;
                in_data  : IN std_logic_vector (31 DOWNTO 0);
                in_L0    : IN std_logic_vector(31 DOWNTO 0); -- keySchedule S0
                in_L1    : IN std_logic_vector(31 DOWNTO 0); -- keySchedule S1
                in_L2    : IN std_logic_vector(31 DOWNTO 0); -- keySchedule S2
                in_L3    : IN std_logic_vector(31 DOWNTO 0); -- keySchedule S3
                out_cont  : OUT std_logic_vector(3 DOWNTO 0);
                out_data : OUT std_logic_vector(31 DOWNTO 0);
                out_ready : OUT std_logic
        );
        END COMPONENT;

        SIGNAL sig_clk          : std_logic;
        SIGNAL sig_rst          : std_logic;
        SIGNAL sig_in_load      : std_logic;
        SIGNAL sig_in_data      : std_logic_vector (31 DOWNTO 0);
        SIGNAL sig_in_L0        : std_logic_vector(31 DOWNTO 0); -- keySchedule S0
        SIGNAL sig_in_L1        : std_logic_vector(31 DOWNTO 0); -- keySchedule S1
```

```
        SIGNAL sig_in_L2          : std_logic_vector(31 DOWNTO 0); -- keySchedule S2
        SIGNAL sig_in_L3          : std_logic_vector(31 DOWNTO 0); -- keySchedule S3
        SIGNAL sig_out_cont  : std_logic_vector(3 DOWNTO 0);
        SIGNAL sig_out_data : std_logic_vector(31 DOWNTO 0);
        SIGNAL sig_out_ready : std_logic;

begin
        UUT: sBox PORT MAP(
                clk             => sig_clk,
                rst             => sig_rst,
                in_load  => sig_in_load,
                in_data  => sig_in_data,
                in_L0    => sig_in_L0,
                in_L1    => sig_in_L1,
                in_L2    => sig_in_L2,
                in_L3    => sig_in_L3,
                out_cont => sig_out_cont,
                out_data => sig_out_data,
                out_ready => sig_out_ready
        );

        clk_Process: PROCESS
        BEGIN
                sig_clk <= '0';
                WAIT FOR 5ns;
                sig_clk <= '1';
                WAIT FOR 5ns;
        END PROCESS;

        rst_Process: PROCESS
        BEGIN
                sig_rst <= '1';
                WAIT FOR 10ns;
                sig_rst <= '0';
                WAIT;
        END PROCESS;

        load_Process: PROCESS
        BEGIN
                sig_in_load <= '0';
                WAIT FOR 5ns;
                sig_in_L0 <= x"00000000";
                sig_in_L1 <= x"00000000";
                sig_in_L2 <= x"00000000";
                sig_in_L3 <= x"00000000";
                sig_in_data <= x"00000000";
                sig_in_load <= '1';
                WAIT FOR 20ns;
                sig_in_load <= '0';
```

```
                WAIT;
        END PROCESS;
end testbench;
library ieee;
use ieee.std_logic_1164.all;

entity bit_adder is
port    (
                inA, inB, in_carry : in std_logic;
                out_Sum, out_carry              : out std_logic
                );
end bit_adder;

architecture bit_adder_arch of bit_adder is
begin
        out_Sum <= in_carry XOR inA XOR inB;
        out_carry <= (inA AND inB) OR(in_carry AND (inA XOR inB));
end bit_adder_arch;

--============================================================
library ieee;
use ieee.std_logic_1164.all;

entity bit_adder32 is
        port    (
                in_dataA, in_dataB      : IN std_logic_vector(31 downto 0);
                out_data : OUT std_logic_vector(31 downto 0)
        );
end bit_adder32;

architecture bit_adder32_arch of bit_adder32  is
        signal  sig_intermediate_carry, sig_adder_out           : std_logic_vector(31 downto 0);
        signal  sig_zero                                        : std_logic;
        component bit_adder
        port    (
                inA, inB, in_carry : in std_logic;
                out_Sum, out_carry              : out std_logic
                );
        end component;

begin
        sig_zero <= '0';
        adder32bits: for i in 0 to 31 generate
                adder_LSb: if (i=0) generate
                        adder0: bit_adder port map      (
                                                inA => in_dataA(0),
                                                inB => in_dataB(0),
                                                in_carry => sig_zero,
                                                out_Sum => sig_adder_out(0),
```

```
                                        out_carry => sig_intermediate_carry(0));
                end generate adder_LSb;

                others_add: if (i>0) generate
                        next_adder: bit_adder  port map   (
                                        inA => in_dataA(i),
                                        inB => in_dataB(i),
                                        in_carry => sig_intermediate_carry(i-1),
                                        out_Sum => sig_adder_out(i),
                                        out_carry => sig_intermediate_carry(i));
                end generate others_add;
        end generate adder32bits;
        sig_intermediate_carry(31) <= '0';
        out_data <= sig_adder_out;
end bit_adder32_arch;

--============================================================

library ieee;
use ieee.std_logic_1164.all;

entity mult_5B is
port    (
                in_5b    : in std_logic_vector(7 downto 0);
                out_5b   : out std_logic_vector(7 downto 0)
                );
end mult_5B;

architecture mult_5B_arch of mult_5B is
begin
        out_5b(0) <= in_5b(2) XOR in_5b(0);
        out_5b(1) <= in_5b(3) XOR in_5b(1) XOR in_5b(0);
        out_5b(2) <= in_5b(4) XOR in_5b(2) XOR in_5b(1);
        out_5b(3) <= in_5b(5) XOR in_5b(3) XOR in_5b(0);
        out_5b(4) <= in_5b(6) XOR in_5b(4) XOR in_5b(1) XOR in_5b(0);
        out_5b(5) <= in_5b(7) XOR in_5b(5) XOR in_5b(1);
        out_5b(6) <= in_5b(6) XOR in_5b(0);
        out_5b(7) <= in_5b(7) XOR in_5b(1);
end mult_5B_arch;

--============================================================

library ieee;
use ieee.std_logic_1164.all;

entity mult_EF is
port    (
                in_ef    : in std_logic_vector(7 downto 0);
                out_ef   : out std_logic_vector(7 downto 0)
```

```
		);
end mult_EF;

architecture mult_EF_arch of mult_EF is

begin
	out_ef(0) <= in_ef(2) XOR in_ef(1) XOR in_ef(0);
	out_ef(1) <= in_ef(3) XOR in_ef(2) XOR in_ef(1) XOR in_ef(0);
	out_ef(2) <= in_ef(4) XOR in_ef(3) XOR in_ef(2) XOR in_ef(1) XOR in_ef(0);
	out_ef(3) <= in_ef(5) XOR in_ef(4) XOR in_ef(3) XOR in_ef(0);
	out_ef(4) <= in_ef(6) XOR in_ef(5) XOR in_ef(4) XOR in_ef(1);
	out_ef(5) <= in_ef(7) XOR in_ef(6) XOR in_ef(5) XOR in_ef(1) XOR in_ef(0);
	out_ef(6) <= in_ef(7) XOR in_ef(6) XOR in_ef(0);
	out_ef(7) <= in_ef(7) XOR in_ef(1) XOR in_ef(0);
end mult_EF_arch;

--============================================================

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.ALL;

entity mds is
	port(
		clk	: IN std_logic;
		in_load	: IN std_logic;
		in_Y0 : IN std_logic_vector(7 DOWNTO 0); --LSB
		in_Y1 : IN std_logic_vector(7 DOWNTO 0);
		in_Y2 : IN std_logic_vector(7 DOWNTO 0);
		in_Y3 : IN std_logic_vector(7 DOWNTO 0); --MSB
		out_Z : OUT std_logic_vector(31 DOWNTO 0);
		out_ready	: OUT std_logic
	);
end mds;

architecture mds_behav of mds is
	component mult_EF port	(
			in_ef : in std_logic_vector(7 downto 0);
			out_ef : out std_logic_vector(7 downto 0)	);
	end component;

	component mult_5B port	(
			in_5b : in std_logic_vector(7 downto 0);
			out_5b : out std_logic_vector(7 downto 0)	);
	end component;

	SIGNAL Y3, Y3_5B, Y3_EF: std_logic_vector(7 downto 0);
```

```
	SIGNAL Y2, Y2_5B, Y2_EF: std_logic_vector(7 downto 0);
	SIGNAL Y1, Y1_5B, Y1_EF: std_logic_vector(7 downto 0);
	SIGNAL Y0, Y0_5B, Y0_EF: std_logic_vector(7 downto 0);
	SIGNAL Z0, Z1, Z2, Z3: std_logic_vector(7 downto 0);
	SIGNAL sig_in_load: std_logic;
	SIGNAL sig_out_ready: std_logic;
	SIGNAL sig_out_Z : std_logic_vector(31 DOWNTO 0);

	type state_MDS is (stLoad, stCalc, stResult);
	signal presentState, nextState : state_MDS;
begin
	Y0 <= in_Y0;
	Y1 <= in_Y1;
	Y2 <= in_Y2;
	Y3 <= in_Y3;
	sig_in_load <= in_load;
	Y0xEF: mult_EF  port map( in_ef => Y0, out_ef => Y0_EF );
	Y0x5B: mult_5B  port map ( in_5b => Y0,  out_5b => Y0_5B );
	Y1xEF: mult_EF  port map ( in_ef => Y1,   out_ef => Y1_EF  );
	Y1x5B: mult_5B  port map ( in_5b => Y1,  out_5b => Y1_5B );
	Y2xEF: mult_EF  port map ( in_ef => Y2,   out_ef => Y2_EF );
	Y2x5B: mult_5B  port map ( in_5b => Y2,  out_5b => Y2_5B );
	Y3xEF: mult_EF  port map ( in_ef => Y3,  out_ef => Y3_EF  );
	Y3x5B: mult_5B  port map ( in_5b => Y3,  out_5b => Y3_5B );

	procMain: process(clk)
	begin
		if (sig_in_load = '1') then
			presentState <= stLoad;
		elsif (clk'event and clk = '1') then
			presentState <= nextState;
		end if;
	end process;

	procStates: process(presentState)
	begin
		case presentState is
			when stLoad =>
				sig_out_Z(31 downto 0) <= (others =>'0');
				sig_out_ready <= '0';
				nextState <= stCalc;
			---------------------------------------
			when stCalc =>
				Z0 <= Y0 XOR Y1_EF XOR Y2_5B XOR Y3_5B;
				Z1 <= Y0_5B XOR Y1_EF XOR Y2_EF XOR Y3;
				Z2 <= Y0_EF XOR Y1_5B XOR Y2 XOR Y3_EF;
				Z3 <= Y0_EF XOR Y1 XOR Y2_EF XOR Y3_5B;
				nextState <= stResult;
			---------------------------------------
```

```
                        when stResult =>
                                sig_out_ready <= '1';
                                sig_out_Z <= Z3 & Z2 & Z1 & Z0;
                                nextState <= stResult;
                        ---------------------------------------
                end case;
         end process;
        out_Z(31 downto 0) <= sig_out_Z(31 downto 0);
        out_ready <= sig_out_ready;
end mds_behav;

--============================================================
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity sKeysGen is
        port(
                clk       : IN std_logic;
                rst       : IN std_logic;
                in_load   : IN std_logic;
                in_MKey : IN std_logic_vector(63 DOWNTO 0);
                out_SKey : OUT std_logic_vector(31 DOWNTO 0);
                out_ready         : OUT std_logic
        );
end sKeysGen;

architecture sKeysGen_behav of sKeysGen is
        component mult_01  port (
                        in_data : in std_logic_vector(7 downto 0);
                        out_data : out std_logic_vector(7 downto 0) );
        end component;
        component mult_02  port (
                        in_data : in std_logic_vector(7 downto 0);
                        out_data : out std_logic_vector(7 downto 0) );
        end component;
        component mult_03 port  (
                        in_data : in std_logic_vector(7 downto 0);
                        out_data : out std_logic_vector(7 downto 0) );
        end component;
                component mult_04  port (
                        in_data : in std_logic_vector(7 downto 0);
                        out_data : out std_logic_vector(7 downto 0) );
        end component;
        component mult_19 port  (
                        in_data : in std_logic_vector(7 downto 0);
                        out_data : out std_logic_vector(7 downto 0) );
        end component;
```

```
component mult_1E  port (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_3D port  (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_47 port  (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_55 port  (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_56 port  (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_58 port  (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_5A port  (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_68  port (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_82  port (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_87  port (
                    in_data : in std_logic_vector(7 downto 0);
                    out_data : out std_logic_vector(7 downto 0) );
end component;
```

```
component mult_9E  port (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_A1 port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_A4 port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_AE port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_C1 port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_C6 port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_DB port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_E5 port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;

component mult_F3 port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;
component mult_FC port  (
                in_data : in std_logic_vector(7 downto 0);
                out_data : out std_logic_vector(7 downto 0) );
end component;
```

```
        SIGNAL sig_in_load        : std_logic;
        SIGNAL sig_out_SKey       : std_logic_vector(31 DOWNTO 0);
        SIGNAL sig_out_ready      : std_logic;
        SIGNAL m0, m1, m2, m3, m4, m5, m6, m7: std_logic_vector (7 downto 0);
        SIGNAL s0, s1, s2, s3: std_logic_vector (7 downto 0);
        SIGNAL m0_01, m1_A4, m2_55, m3_87, m4_5A, m5_58, m6_DB, m7_9E: std_logic_vector (7 downto 0);
        SIGNAL m0_A4, m1_56, m2_82, m3_F3, m4_1E, m5_C6, m6_68, m7_E5: std_logic_vector (7 downto 0);
        SIGNAL m0_02, m1_A1, m2_FC, m3_C1, m4_47, m5_AE, m6_3D, m7_19: std_logic_vector (7 downto 0);
        SIGNAL m1_55, m2_87, m3_5A, m4_58, m5_DB, m6_9E, m7_03      : std_logic_vector (7 downto 0);

        type state_SKeys is (reset, stLoad, stCalc, stResult);
        SIGNAL presentState, nextState : state_SKeys;
begin
        m0x01: mult_01  port map ( in_data => m0, out_data => m0_01 );
        m0x02: mult_02 port map( in_data => m0, out_data => m0_02);
        m0xA4: mult_A4 port map (in_data => m0, out_data => m0_A4);
        m1x55: mult_55 port map( in_data => m1, out_data => m1_55);
        m1x56: mult_56 port map( in_data => m1, out_data => m1_56 );
        m1xA1: mult_A1 port map(in_data => m1, out_data => m1_A1);
        m1xA4: mult_A4 port map( in_data => m1, out_data => m1_A4);
        m2x55: mult_55 port map (in_data => m2, out_data => m2_55);
        m2x82: mult_82 port map ( in_data => m2, out_data => m2_82 );
        m2x87: mult_87 port map ( in_data => m2, out_data => m2_87);
        m2xFC: mult_FC  port map(in_data => m2, out_data => m2_FC);
        m3x87: mult_87 port map (in_data => m3, out_data => m3_87);
        m3x5A: mult_5A port map (in_data => m3, out_data => m3_5A);
        m3xC1: mult_C1 port map ( in_data => m3, out_data => m3_C1 );
        m3xF3: mult_F3 port map ( in_data => m3, out_data => m3_F3 );
        m4x47: mult_47 port map ( in_data => m4, out_data => m4_47);
        m4x58: mult_58 port map ( in_data => m4, out_data => m4_58);
        m4x5A: mult_5A port map( in_data => m4, out_data => m4_5A);
        m4x1E: mult_1E port map( in_data => m4, out_data => m4_1E);
        m5x58: mult_58 port map( in_data => m5, out_data => m5_58);
        m5xAE: mult_AE port map( in_data => m5, out_data => m5_AE);
        m5xC6: mult_C6 port map (in_data => m5, out_data => m5_C6);
        m5xDB: mult_DB port map (in_data => m5, out_data => m5_DB);
        m6x3D: mult_3D port map (in_data => m6, out_data => m6_3D);
        m6x68: mult_68 port map (in_data => m6, out_data => m6_68);
        m6x9E: mult_9E port map (in_data => m6, out_data => m6_9E);
        m6xDB: mult_DB port map (in_data => m6, out_data => m6_DB);
        m7x03: mult_03 port map ( in_data => m7, out_data => m7_03);
        m7x19: mult_19 port map ( in_data => m7, out_data => m7_19);
        m7x9E: mult_9E port map (in_data => m7, out_data => m7_9E);
        m7xE5: mult_E5 port map ( in_data => m7, out_data => m7_E5);

        m0 <= in_MKey(7 downto 0);
        m1 <= in_MKey(15 downto 8);
        m2 <= in_MKey(23 downto 16);
        m3 <= in_MKey(31 downto 24);
```

```
	m4 <= in_MKey(39 downto 32);
	m5 <= in_MKey(47 downto 40);
	m6 <= in_MKey(55 downto 48);
	m7 <= in_MKey(63 downto 56);
	sig_in_load <= in_load;

	procMain: process(clk, rst)
	begin
		if (rst = '1') then
			presentState <= reset;
		elsif (sig_in_load = '1') then
			presentState <= stLoad;
		elsif (clk'event and clk = '1') then
			presentState <= nextState;
		end if;
	end process;

	procStates: process(presentState)
	begin
		case presentState is
			when reset =>
				sig_out_SKey <= (others =>'0');
				sig_out_ready <= '0';
				nextState <= reset;
			---------------------------------------
			when stLoad =>
				sig_out_ready <= '0';
				nextState <= stCalc;
			---------------------------------------
			when stCalc =>
		s0 <= m0_01 xor m1_A4 xor m2_55 xor m3_87 xor m4_5A xor m5_58 xor m6_DB xor m7_9E;
		s1 <= m0_A4 xor m1_56 xor m2_82 xor m3_F3 xor m4_1E xor m5_C6 xor m6_68 xor m7_E5;
		s2 <= m0_02 xor m1_A1 xor m2_FC xor m3_C1 xor m4_47 xor m5_AE xor m6_3D xor m7_19;
		s3 <= m0_A4 xor m1_55 xor m2_87 xor m3_5A xor m4_58 xor m5_DB xor m6_9E xor m7_03;
				nextState <= stResult;
			---------------------------------------
			when stResult =>
				sig_out_ready <= '1';
				sig_out_SKey <= s3 & s2 & s1 & s0;
				nextState <= stResult;
			---------------------------------------
		end case;
  end process;
	out_SKey <= sig_out_SKey;
	out_ready <= sig_out_ready;
end sKeysGen_behav;
```

Anexo I

Vetores de teste do algoritmo AES

Este anexo apresenta uma parte significativa dos vetores de teste utilizados neste trabalho para a validação do AES. Estes vetores de teste podem ser encontrados na página do NIST: http://csrc.nist.gov/groups/STM/cavp/ .

```
KEYSIZE=128
KEY=000102030405060708090A0B0C0D0E0F
```

Intermediate Ciphertext Values (Encryption)

```
PT=000102030405060708090A0B0C0D0E0F
CT1=B5C9179EB1CC1199B9C51B92B5C8159D
CT2=2B65F6374C427C5B2FE3A9256896755B
CT3=D1015FCBB4EF65679688462076B9D6AD
CT4=8E17064A2A35A183729FE59FF3A591F1
CT5=D7557DD55999DB3259E2183D558DCDD2
CT6=73A96A5D7799A5F3111D2B63684B1F7F
CT7=1B6B853069EEFC749AFEFD7B57A04CD1
CT8=107EEADFB6F77933B5457A6F08F046B2
CT9=8EC166481A677AA96A14FF6ECE88C010
CT=0A940BB5416EF045F1C39458C653EA5A
```

Intermediate Ciphertext Values (Decryption)

```
CT=0A940BB5416EF045F1C39458C653EA5A
PT1=8EC166481A677AA96A14FF6ECE88C010
PT2=107EEADFB6F77933B5457A6F08F046B2
PT3=1B6B853069EEFC749AFEFD7B57A04CD1
PT4=73A96A5D7799A5F3111D2B63684B1F7F
PT5=D7557DD55999DB3259E2183D558DCDD2
PT6=8E17064A2A35A183729FE59FF3A591F1
PT7=D1015FCBB4EF65679688462076B9D6AD
PT8=2B65F6374C427C5B2FE3A9256896755B
PT9=B5C9179EB1CC1199B9C51B92B5C8159D
PT=000102030405060708090A0B0C0D0E0F
```

KEYSIZE=192
KEY=000102030405060708090A0B0C0D0E0F1011121314151617

Intermediate Ciphertext Values (Encryption)

PT=000102030405060708090A0B0C0D0E0F
CT1=73727170777675743B25919A3F20979D
CT2=C673B27A311EC2EB64AD47FF53B233D7
CT3=0B5CC6BA34C807E6496D79B46826A1E8
CT4=005B53A5B660E280307883487E4D1A4D
CT5=88A105F0DDD45F3674DBC3DE1A211B03
CT6=EB5CD8B5FD8A3F33F03A70FB5C620C06
CT7=909913B09BD2CC5A70B6C647931F0A1F
CT8=6EB6CA10E395AFD646B02C5E9E745A9F
CT9=2CFD2FC41AF82B8DFB80E9BD1C989ECE
CT10=31C5D5E27EAF073E5C21ADAAEAA969D4
CT11=1DB94956A7268B0DE963D27E55868580
CT=0060BFFE46834BB8DA5CF9A61FF220AE

Intermediate Ciphertext Values (Decryption)

CT=0060BFFE46834BB8DA5CF9A61FF220AE
PT1=1DB94956A7268B0DE963D27E55868580
PT2=31C5D5E27EAF073E5C21ADAAEAA969D4
PT3=2CFD2FC41AF82B8DFB80E9BD1C989ECE
PT4=6EB6CA10E395AFD646B02C5E9E745A9F
PT5=909913B09BD2CC5A70B6C647931F0A1F
PT6=EB5CD8B5FD8A3F33F03A70FB5C620C06
PT7=88A105F0DDD45F3674DBC3DE1A211B03
PT8=005B53A5B660E280307883487E4D1A4D
PT9=0B5CC6BA34C807E6496D79B46826A1E8
PT10=C673B27A311EC2EB64AD47FF53B233D7
PT11=73727170777675743B25919A3F20979D
PT=000102030405060708090A0B0C0D0E0F

==========

KEYSIZE=256
KEY=000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F

Intermediate Ciphertext Values (Encryption)

PT=000102030405060708090A0B0C0D0E0F
CT1=73727170777675747B7A79787F7E7D7C
CT2=4E5D32BB8B67FD1BD4CFEC9FFB20AC4F
CT3=96A212E486341549C4AAF7C843F0277A
CT4=0F45F284CDD0CB16E3EA81ECC891A4E1
CT5=E59BFC458A89063E0137BBE6DB63A058
CT6=1D958D960EA3143383C17D5CD87BA327
CT7=43843EF40D9219481935B77A586DB5DE
CT8=5AA5ABADBC40230CBA6124E9FAEEEFB5
CT9=DAD61937BDFD582927F14C990C5FC761
CT10=E8A48C5DEE5C0792AB6DFFF5B038529D
CT11=4B71E5A8BFB4E9A5312A18119E68E829
CT12=DCBA75CEE6589DDC0D289A172E8415B5
CT13=8A0E856B2074C1093104131D0628BFE8
CT=5A6E045708FB7196F02E553D02C3A692

Intermediate Ciphertext Values (Decryption)

CT=5A6E045708FB7196F02E553D02C3A692
PT1=8A0E856B2074C1093104131D0628BFE8
PT2=DCBA75CEE6589DDC0D289A172E8415B5
PT3=4B71E5A8BFB4E9A5312A18119E68E829
PT4=E8A48C5DEE5C0792AB6DFFF5B038529D
PT5=DAD61937BDFD582927F14C990C5FC761
PT6=5AA5ABADBC40230CBA6124E9FAEEEFB5
PT7=43843EF40D9219481935B77A586DB5DE
PT8=1D958D960EA3143383C17D5CD87BA327
PT9=E59BFC458A89063E0137BBE6DB63A058
PT10=0F45F284CDD0CB16E3EA81ECC891A4E1
PT11=96A212E486341549C4AAF7C843F0277A
PT12=4E5D32BB8B67FD1BD4CFEC9FFB20AC4F
PT13=73727170777675747B7A79787F7E7D7C
PT=000102030405060708090A0B0C0D0E0F

==========

KEYSIZE=128

PT=00000000000000000000000000000000

I=1
KEY=80000000000000000000000000000000
CT=0EDD33D3C621E546455BD8BA1418BEC8

I=2
KEY=40000000000000000000000000000000
CT=C0CC0C5DA5BD63ACD44A80774FAD5222

I=3
KEY=20000000000000000000000000000000
CT=2F0B4B71BC77851B9CA56D42EB8FF080

I=4
KEY=10000000000000000000000000000000
CT=6B1E2FFFE8A114009D8FE22F6DB5F876

I=5
KEY=08000000000000000000000000000000
CT=9AA042C315F94CBB97B62202F83358F5

I=6
KEY=04000000000000000000000000000000
CT=DBE01DE67E346A800C4C4B4880311DE4

I=7
KEY=02000000000000000000000000000000
CT=C117D2238D53836ACD92DDCDB85D6A21

I=8
KEY=01000000000000000000000000000000
CT=DC0ED85DF9611ABB7249CDD168C5467E

I=9
KEY=00800000000000000000000000000000
CT=807D678FFF1F56FA92DE3381904842F2

I=10
KEY=00400000000000000000000000000000
CT=0E53B3FCAD8E4B130EF73AEB957FB402

==========

KEYSIZE=192

PT=00000000000000000000000000000000

I=1
KEY=800000000000000000000000000000000000000000000000
CT=DE885DC87F5A92594082D02CC1E1B42C

I=2
KEY=400000000000000000000000000000000000000000000000
CT=C749194F94673F9DD2AA1932849630C1

I=3
KEY=200000000000000000000000000000000000000000000000
CT=0CEF643313912934D310297B90F56ECC

I=4
KEY=100000000000000000000000000000000000000000000000
CT=C4495D39D4A553B225FBA02A7B1B87E1

I=5
KEY=080000000000000000000000000000000000000000000000
CT=636D10B1A0BCAB541D680A7970ADC830

I=6
KEY=040000000000000000000000000000000000000000000000
CT=07CF045786BD6AFCC147D99E45A901A7

I=7
KEY=020000000000000000000000000000000000000000000000
CT=6A8E3F425A7599348F95398448827976

I=8
KEY=010000000000000000000000000000000000000000000000
CT=5518276836148A00D91089A20D8BFF57

I=9
KEY=008000000000000000000000000000000000000000000000
CT=F267E07B5E87E3BC20B969C61D4FCB06

I=10
KEY=004000000000000000000000000000000000000000000000
CT=5A1CDE69571D401BFCD20DEBADA2212C

==========

KEYSIZE=256

PT=00000000000000000000000000000000

I=1
KEY=8000000000000000000000000000000000000000000000000000000000000000
CT=E35A6DCB19B201A01EBCFA8AA22B5759

I=2
KEY=4000000000000000000000000000000000000000000000000000000000000000
CT=5075C2405B76F22F553488CAE47CE90B

I=3
KEY=2000000000000000000000000000000000000000000000000000000000000000
CT=49DF95D844A0145A7DE01C91793302D3

I=4
KEY=1000000000000000000000000000000000000000000000000000000000000000
CT=E7396D778E940B8418A86120E5F421FE

I=5
KEY=0800000000000000000000000000000000000000000000000000000000000000
CT=05F535C36FCEDE4657BE37F4087DB1EF

I=6
KEY=0400000000000000000000000000000000000000000000000000000000000000
CT=D0C1DDDD10DA777C68AB36AF51F2C204

I=7
KEY=0200000000000000000000000000000000000000000000000000000000000000
CT=1C55FB811B5C6464C4E5DE1535A75514

I=8
KEY=0100000000000000000000000000000000000000000000000000000000000000
CT=52917F3AE957D5230D3A2AF57C7B5A71

I=9
KEY=0080000000000000000000000000000000000000000000000000000000000000
CT=C6E3D5501752DD5E9AEF086D6B45D705

I=10
KEY=0040000000000000000000000000000000000000000000000000000000000000
CT=A24A9C7AF1D9B1E17E1C9A3E711B3FA7

Anexo II

Vetores de teste do algoritmo Twofish

Este anexo apresenta alguns dos vetores de teste utilizados neste trabalho para a validação do Twofish. Os vetores de teste podem ser encontrados na página dos criadores do algoritmo: https://www.schneier.com/code/ecb_ival.txt .

```
KEYSIZE=128

KEY=00000000000000000000000000000000

;
;makeKey:   Input key            -->  S-box key     [Encrypt]
;             00000000 00000000  -->  00000000
;             00000000 00000000  -->  00000000
;           Subkeys
;             52C54DDE 11F0626D   Input whiten
;             7CAC9D4A 4D1B4AAA
;             B7B83A10 1E7D0BEB  Output whiten
;             EE9C341F CFE14BE4
;             F98FFEF9 9C5B3C17  Round subkeys
;             15A48310 342A4D81
;             424D89FE C14724A7
;             311B834C FDE87320
;             3302778F 26CD67B4
;             7A6C6362 C2BAF60E
;             3411B994 D972C87F
;             84ADB1EA A7DEE434
;             54D2960F A2F7CAA8
;             A6B8FF8C 8014C425
;             6A748D1C EDBAF720
;             928EF78C 0338EE13
;             9949D6BE C8314176
;             07C07D68 ECAE7EA7
;             1FE71844 85C05C89
;             F298311E 696EA672
;

PT=00000000000000000000000000000000

Encrypt()

R[-1]: x= 00000000  00000000  00000000  00000000.
R[ 0]: x= 52C54DDE  11F0626D  7CAC9D4A  4D1B4AAA.
R[ 1]: x= 52C54DDE  11F0626D  C38DCAA4  7A0A91B6.    t0=C06D4949. t1=41B9BFC1.
R[ 2]: x= 55A538DE  5C5A4DB6  C38DCAA4  7A0A91B6.    t0=7C4536B9. t1=67A58299.
R[ 3]: x= 55A538DE  5C5A4DB6  899063BD  893E49A9.    t0=60DAC1A4. t1=2D84C23D.
```

```
R[ 4]: x= 2AE61A96  84BC42D3  899063BD  893E49A9.    t0=607AAEAD. t1=6ED2DBF9.
R[ 5]: x= 2AE61A96  84BC42D3  F14F2618  821B5F36.    t0=067D0B49. t1=318EACB4.
R[ 6]: x= 0FFE0AD1  D6B87B70  F14F2618  821B5F36.    t0=58554EDB. t1=62585CF7.
R[ 7]: x= 0FFE0AD1  D6B87B70  CD0D38A1  C069BD9B.    t0=839B0017. t1=B3A89DB0.
R[ 8]: x= A85CE579  DE2661CE  CD0D38A1  C069BD9B.    t0=E9BC6975. t1=F0DDA4C3.
R[ 9]: x= A85CE579  DE2661CE  7A39754C  973ABD2A.    t0=54687CDF. t1=9044BF4B.
R[10]: x= 013077D7  B3528BA1  7A39754C  973ABD2A.    t0=77FC927F. t1=8B8678CC.
R[11]: x= 013077D7  B3528BA1  D57933FD  F8EA8B1B.    t0=E3C81108. t1=828E7493.
R[12]: x= 64F0EAA1  DA27090C  D57933FD  F8EA8B1B.    t0=B33C25D6. t1=83068533.
R[13]: x= 64F0EAA1  DA27090C  F64F1005  99149A52.    t0=A0AA2F81. t1=FFF30DB7.
R[14]: x= B0681C46  606D0273  F64F1005  99149A52.    t0=114C17C5. t1=EB143CFF.
R[15]: x= B0681C46  606D0273  EB27628F  2C51191D.    t0=677DA87D. t1=989D1459.
R[16]: x= C1708BA9  9522A3CE  EB27628F  2C51191D.    t0=9357B338. t1=AC9926BF.
R[17]: x= 5C9F589F  322C12F6  2FECBFB6  5AC3E82A.
```

CT=9F589F5CF6122C32B6BFEC2F2AE8C35A

Decrypt()

CT=9F589F5CF6122C32B6BFEC2F2AE8C35A

```
R[17]: x= 5C9F589F  322C12F6  2FECBFB6  5AC3E82A.
R[16]: x= C1708BA9  9522A3CE  EB27628F  2C51191D.    t0=9357B338. t1=AC9926BF.
R[15]: x= B0681C46  606D0273  EB27628F  2C51191D.    t0=677DA87D. t1=989D1459.
R[14]: x= B0681C46  606D0273  F64F1005  99149A52.    t0=114C17C5. t1=EB143CFF.
R[13]: x= 64F0EAA1  DA27090C  F64F1005  99149A52.    t0=A0AA2F81. t1=FFF30DB7.
R[12]: x= 64F0EAA1  DA27090C  D57933FD  F8EA8B1B.    t0=B33C25D6. t1=83068533.
R[11]: x= 013077D7  B3528BA1  D57933FD  F8EA8B1B.    t0=E3C81108. t1=828E7493.
R[10]: x= 013077D7  B3528BA1  7A39754C  973ABD2A.    t0=77FC927F. t1=8B8678CC.
R[ 9]: x= A85CE579  DE2661CE  7A39754C  973ABD2A.    t0=54687CDF. t1=9044BF4B.
R[ 8]: x= A85CE579  DE2661CE  CD0D38A1  C069BD9B.    t0=E9BC6975. t1=F0DDA4C3.
R[ 7]: x= 0FFE0AD1  D6B87B70  CD0D38A1  C069BD9B.    t0=839B0017. t1=B3A89DB0.
R[ 6]: x= 0FFE0AD1  D6B87B70  F14F2618  821B5F36.    t0=58554EDB. t1=62585CF7.
R[ 5]: x= 2AE61A96  84BC42D3  F14F2618  821B5F36.    t0=067D0B49. t1=318EACB4.
R[ 4]: x= 2AE61A96  84BC42D3  899063BD  893E49A9.    t0=607AAEAD. t1=6ED2DBF9.
R[ 3]: x= 55A538DE  5C5A4DB6  899063BD  893E49A9.    t0=60DAC1A4. t1=2D84C23D.
R[ 2]: x= 55A538DE  5C5A4DB6  C38DCAA4  7A0A91B6.    t0=7C4536B9. t1=67A58299.
R[ 1]: x= 52C54DDE  11F0626D  C38DCAA4  7A0A91B6.    t0=C06D4949. t1=41B9BFC1.
R[ 0]: x= 52C54DDE  11F0626D  7CAC9D4A  4D1B4AAA.
R[-1]: x= 00000000  00000000  00000000  00000000.
```

PT=00000000000000000000000000000000

==========

KEYSIZE=192

KEY=0123456789ABCDEFFEDCBA98765432100011223344556677

```
;
;makeKey:   Input key           -->  S-box key     [Encrypt]
;             EFCDAB89 67452301  -->  B89FF6F2
;             10325476 98BADCFE  -->  B255BC4B
;             77665544 33221100  -->  45661061
;           Subkeys
;             38394A24 C36D1175   Input whiten
;             E802528F 219BFEB4
;             B9141AB4 BD3E70CD  Output whiten
;             AF609383 FD36908A
;             03EFB931 1D2EE7EC  Round subkeys
```

```
;             A7489D55 6E44B6E8
;             714AD667 653AD51F
;             B6315B66 B27C05AF
;             A06C8140 9853D419
;             4016E346 8D1C0DD4
;             F05480BE B6AF816F
;             2D7DC789 45B7BD3A
;             57F8A163 2BEFDA69
;             26AE7271 C2900D79
;             ED323794 3D3FFD80
;             5DE68E49 9C3D2478
;             DF326FE3 5911F70D
;             C229F13B B1364772
;             4235364D 0CEC363A
;             57C8DD1F 6A1AD61E
;
```

PT=00000000000000000000000000000000

Encrypt()

```
R[-1]: x= 00000000  00000000  00000000  00000000.
R[ 0]: x= 38394A24  C36D1175  E802528F  219BFEB4.
R[ 1]: x= 38394A24  C36D1175  9C263D67  5E68BE8F.    t0=988C8223. t1=33D1ECEC.
R[ 2]: x= C8F5099F  0C4B8F53  9C263D67  5E68BE8F.    t0=E8C880BC. t1=19C23B0A.
R[ 3]: x= C8F5099F  0C4B8F53  69948F5E  E67C030F.    t0=C615F1F6. t1=17AE5B7E.
R[ 4]: x= 07633866  59421079  69948F5E  E67C030F.    t0=90AB32AA. t1=7F56EB43.
R[ 5]: x= 07633866  59421079  C015BE79  149B9CEC.    t0=52971E00. t1=F6BC546D.
R[ 6]: x= A042B99D  709EF54B  C015BE79  149B9CEC.    t0=DAA00849. t1=2D2F5FCE.
R[ 7]: x= A042B99D  709EF54B  0CD39FA6  B250BEDA.    t0=EE03FB5B. t1=FB5A051C.
R[ 8]: x= F7B097FA  9E5C4FF7  0CD39FA6  B250BEDA.    t0=09A1B597. t1=18041948.
R[ 9]: x= F7B097FA  9E5C4FF7  77FC8B29  CC2B3F88.    t0=99C9694E. t1=F1687F43.
R[10]: x= A279C718  421A8D38  77FC8B29  CC2B3F88.    t0=5D174956. t1=2F7D5E04.
R[11]: x= A279C718  421A8D38  5B1A0904  12FEBF99.    t0=5BC40012. t1=78D2617B.
R[12]: x= E4409C22  702548A2  5B1A0904  12FEBF99.    t0=C251B3CE. t1=4AC0BD46.
R[13]: x= E4409C22  702548A2  5DDAA2A1  EFB2F051.    t0=91BC2070. t1=6FC0BBF3.
R[14]: x= 8561A604  825D2480  5DDAA2A1  EFB2F051.    t0=A7D24F8E. t1=84878F62.
R[15]: x= 8561A604  825D2480  5CC6CB7B  62A2CE64.    t0=93690387. t1=0EB8FA83.
R[16]: x= 17738CD3  B5142D18  5CC6CB7B  62A2CE64.    t0=5FE8370B. t1=F3D5AB78.
R[17]: x= E5D2D1CF  DF9CBEA9  B8131F50  4822BD92.
```

CT=CFD1D2E5A9BE9CDF501F13B892BD2248

Decrypt()

CT=CFD1D2E5A9BE9CDF501F13B892BD2248

```
R[17]: x= E5D2D1CF  DF9CBEA9  B8131F50  4822BD92.
R[16]: x= 17738CD3  B5142D18  5CC6CB7B  62A2CE64.    t0=5FE8370B. t1=F3D5AB78.
R[15]: x= 8561A604  825D2480  5CC6CB7B  62A2CE64.    t0=93690387. t1=0EB8FA83.
R[14]: x= 8561A604  825D2480  5DDAA2A1  EFB2F051.    t0=A7D24F8E. t1=84878F62.
R[13]: x= E4409C22  702548A2  5DDAA2A1  EFB2F051.    t0=91BC2070. t1=6FC0BBF3.
R[12]: x= E4409C22  702548A2  5B1A0904  12FEBF99.    t0=C251B3CE. t1=4AC0BD46.
R[11]: x= A279C718  421A8D38  5B1A0904  12FEBF99.    t0=5BC40012. t1=78D2617B.
R[10]: x= A279C718  421A8D38  77FC8B29  CC2B3F88.    t0=5D174956. t1=2F7D5E04.
R[ 9]: x= F7B097FA  9E5C4FF7  77FC8B29  CC2B3F88.    t0=99C9694E. t1=F1687F43.
R[ 8]: x= F7B097FA  9E5C4FF7  0CD39FA6  B250BEDA.    t0=09A1B597. t1=18041948.
R[ 7]: x= A042B99D  709EF54B  0CD39FA6  B250BEDA.    t0=EE03FB5B. t1=FB5A051C.
R[ 6]: x= A042B99D  709EF54B  C015BE79  149B9CEC.    t0=DAA00849. t1=2D2F5FCE.
R[ 5]: x= 07633866  59421079  C015BE79  149B9CEC.    t0=52971E00. t1=F6BC546D.
```

```
R[ 4]: x= 07633866  59421079  69948F5E  E67C030F.    t0=90AB32AA. t1=7F56EB43.
R[ 3]: x= C8F5099F  0C4B8F53  69948F5E  E67C030F.    t0=C615F1F6. t1=17AE5B7E.
R[ 2]: x= C8F5099F  0C4B8F53  9C263D67  5E68BE8F.    t0=E8C880BC. t1=19C23B0A.
R[ 1]: x= 38394A24  C36D1175  9C263D67  5E68BE8F.    t0=988C8223. t1=33D1ECEC.
R[ 0]: x= 38394A24  C36D1175  E802528F  219BFEB4.
R[-1]: x= 00000000  00000000  00000000  00000000.
```

PT=00000000000000000000000000000000

==========

KEYSIZE=256

KEY=0123456789ABCDEFFEDCBA98765432100011223344556677889 9AABBCCDDEEFF

```
;
;makeKey:   Input key           -->  S-box key     [Encrypt]
;             EFCDAB89 67452301  -->  B89FF6F2
;             10325476 98BADCFE  -->  B255BC4B
;             77665544 33221100  -->  45661061
;             FFEEDDCC BBAA9988  -->  8E4447F7
;           Subkeys
;             5EC769BF 44D13C60   Input whiten
;             76CD39B1 16750474
;             349C294B EC21F6D6  Output whiten
;             4FBD10B4 578DA0ED
;             C3479695 9B6958FB  Round subkeys
;             6A7FBC4E 0BF1830B
;             61B5E0FB D78D9730
;             7C6CF0C4 2F9109C8
;             E69EA8D1 ED99BDFF
;             35DC0BBD A03E5018
;             FB18EA0B 38BD43D3
;             76191781 37A9A0D3
;             72427BEA 911CC0B8
;             F1689449 71009CA9
;             B6363E89 494D9855
;             590BBC63 F95A28B5
;             FB72B4E1 2A43505C
;             BFD34176 5C133D12
;             3A9247F7 9A3331DD
;             EE7515E6 F0D54DCD
;
```

PT=00000000000000000000000000000000

Encrypt()

```
R[-1]: x= 00000000  00000000  00000000  00000000.
R[ 0]: x= 5EC769BF  44D13C60  76CD39B1  16750474.
R[ 1]: x= 5EC769BF  44D13C60  D38B6C9F  A23B7169.    t0=29C0736C. t1=E4D3D68D.
R[ 2]: x= 99424DFF  FBC14BFC  D38B6C9F  A23B7169.    t0=9D16BBB3. t1=64AD7A3F.
R[ 3]: x= 99424DFF  FBC14BFC  698BE047  6A997290.    t0=E66B9D19. t1=B87B2DFD.
R[ 4]: x= 2C125DD7  5A526278  698BE047  6A997290.    t0=0BB41F61. t1=3945E62C.
R[ 5]: x= 2C125DD7  5A526278  E35CD910  7CB57D06.    t0=D5397903. t1=F35A3092.
R[ 6]: x= D5178F25  00D35CC5  E35CD910  7CB57D06.    t0=8C8927A1. t1=C3D8103E.
R[ 7]: x= D5178F25  00D35CC5  D8447F91  65C2BD96.    t0=4D8B7489. t1=0B2FC79F.
R[ 8]: x= FF92E109  DF621C97  D8447F91  65C2BD96.    t0=C1176720. t1=F301CE95.
R[ 9]: x= FF92E109  DF621C97  28BFEFF5  D45666FB.    t0=9F3BEC03. t1=77BD388E.
R[10]: x= BB79AD2E  AA410F41  28BFEFF5  D45666FB.    t0=8C6DB451. t1=0B8B72BA.
R[11]: x= BB79AD2E  AA410F41  6576A3ED  BFF8215E.    t0=8A317EF8. t1=A1EAEAAE.
```

```
R[12]: x= 4A6BBAFF  439F4766  6576A3ED  BFF8215E.    t0=8F8307AA. t1=472014C3.
R[13]: x= 4A6BBAFF  439F4766  F7186836  04CA5304.    t0=CEB0BBE1. t1=C12302BE.
R[14]: x= CBD3C29D  BC31FEBE  F7186836  04CA5304.    t0=5CF5C93C. t1=C1033512.
R[15]: x= CBD3C29D  BC31FEBE  D4E77B7C  5415D5D3.    t0=853A6BB2. t1=9F09EB26.
R[16]: x= 85411C2B  7777DC05  D4E77B7C  5415D5D3.    t0=877AF61D. t1=4B61EEC7.
R[17]: x= E07B5237  B8342305  CAFC0C9F  20FA7CE8.
```

CT=37527BE0052334B89F0CFCCAE87CFA20

Decrypt()

CT=37527BE0052334B89F0CFCCAE87CFA20

```
R[17]: x= E07B5237  B8342305  CAFC0C9F  20FA7CE8.
R[16]: x= 85411C2B  7777DC05  D4E77B7C  5415D5D3.    t0=877AF61D. t1=4B61EEC7.
R[15]: x= CBD3C29D  BC31FEBE  D4E77B7C  5415D5D3.    t0=853A6BB2. t1=9F09EB26.
R[14]: x= CBD3C29D  BC31FEBE  F7186836  04CA5304.    t0=5CF5C93C. t1=C1033512.
R[13]: x= 4A6BBAFF  439F4766  F7186836  04CA5304.    t0=CEB0BBE1. t1=C12302BE.
R[12]: x= 4A6BBAFF  439F4766  6576A3ED  BFF8215E.    t0=8F8307AA. t1=472014C3.
R[11]: x= BB79AD2E  AA410F41  6576A3ED  BFF8215E.    t0=8A317EF8. t1=A1EAEAAE.
R[10]: x= BB79AD2E  AA410F41  28BFEFF5  D45666FB.    t0=8C6DB451. t1=0B8B72BA.
R[ 9]: x= FF92E109  DF621C97  28BFEFF5  D45666FB.    t0=9F3BEC03. t1=77BD388E.
R[ 8]: x= FF92E109  DF621C97  D8447F91  65C2BD96.    t0=C1176720. t1=F301CE95.
R[ 7]: x= D5178F25  00D35CC5  D8447F91  65C2BD96.    t0=4D8B7489. t1=0B2FC79F.
R[ 6]: x= D5178F25  00D35CC5  E35CD910  7CB57D06.    t0=8C8927A1. t1=C3D8103E.
R[ 5]: x= 2C125DD7  5A526278  E35CD910  7CB57D06.    t0=D5397903. t1=F35A3092.
R[ 4]: x= 2C125DD7  5A526278  698BE047  6A997290.    t0=0BB41F61. t1=3945E62C.
R[ 3]: x= 99424DFF  FBC14BFC  698BE047  6A997290.    t0=E66B9D19. t1=B87B2DFD.
R[ 2]: x= 99424DFF  FBC14BFC  D38B6C9F  A23B7169.    t0=9D16BBB3. t1=64AD7A3F.
R[ 1]: x= 5EC769BF  44D13C60  D38B6C9F  A23B7169.    t0=29C0736C. t1=E4D3D68D.
R[ 0]: x= 5EC769BF  44D13C60  76CD39B1  16750474.
R[-1]: x= 00000000  00000000  00000000  00000000.
```

PT=00000000000000000000000000000000