UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CARLOS ROBERTO MORATELLI

TÉCNICAS PARA O PROJETO DE HARDWARE CRIPTOGRÁFICO TOLERANTE A FALHAS

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Marcelo Soares Lubaszeswki
Orientador

Porto Alegre, Abril de 2007

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Moratelli, Carlos Roberto

TÉCNICAS PARA O PROJETO DE HARDWARE CRIPTOGRÁFICO TOLERANTE A FALHAS / Carlos Roberto Moratelli. – Porto Alegre: PPGC da UFRGS, 2007.

84 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2007. Orientador: Marcelo Soares Lubaszeswki.

1. Smart Cards. 2. Hardware Criptográfico. 3. Fault Attacks. 4. Tolerância a Falhas. I. Lubaszeswki, Marcelo Soares. II. Título.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

CI	Circuito Integrado
GSM	Global System for Mobile Communications
PIN	Personal Identification Number
SoC	System on Chip
CPU	Central Processor Unit
ROM	Read Only Memory
EEPROM	Eletrically Erasable and Programable ROM
RAM	Random Access Memory
I/O	Input/Output
SO	Sistema Operacional
AMBA	Advanced Microcontroller Bus Architecture
ASB	Advanced System Bus
APB	Advanced Peripherical Bus
RNG	Random Number Generator
PC	Personal Computer
VPN	Virtual Private Network
MDC	Modification Detection Codes
MAC	Message Authentication Codes
DES	Data Encryption Standard
NSA	National Security Agency
AES	Advanced Encryption Standard
NIST	National Institute of Standards and Technology
SPA	Simple Power Analysis
DPA	Differential Power Analysis
SEMA	Single Eletromagnetic Attack
DEMA	Differential Eletromagnetic Attack

DFA	Differential Fault Attack
ASIC	Application-Specific Integrated Circuit
CMOS	Complementary Metal-Oxide-Semiconductor
VHDL	VHSIC Hardware Description Language
FPGA	Field Programmable Gate Array
PGSC	Processo de Geração de Sinais de Controle
PIF	Processo de Injeção de Falhas

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho tem como foco principal o estudo de um tipo específico de ataque a sistemas criptográficos. A implementação em *hardware*, de algoritmos criptográficos, apresenta uma série de vulnerabilidades, as quais, não foram previstas no projeto original de tais algoritmos. Os principais alvos destes tipos de ataque são dispositivos portáteis que implementam algoritmos criptográfico em *hardware* devido as limitações de seus processadores embarcados. Um exemplo deste tipo de dispositivo são os *Smart Cards*, os quais, são extensamente utilizados nos sistemas GSM de telefonia móvel e estão sendo adotados no ramo bancário. Tais dispositivos podem ser atacados de diferentes maneiras, por exemplo, analisando-se a energia consumida pelo dispositivo, o tempo gasto no processamento ou ainda explorando a suscetibilidade do *hardware* a ocorrência de falhas transientes. O objetivo de tais ataques é a extração de informações sigilosas armazenadas no cartão como, por exemplo, a chave criptográfica. Ataques por injeção maliciosa de falhas no *hardware* são comumente chamados de DFA (*Differencial Fault Attack*) ou simplesmente *fault attack*. O objetivo deste trabalho foi estudar como ataques por DFA ocorrem em diferentes algoritmos e propor soluções para impedir tais ataques. Os algoritmos criptográficos abordados foram o DES e o AES, por serem amplamente conhecidos e utilizados. São apresentadas diferentes soluções capazes de ajudar a impedir a execução de ataques por DFA. Tais soluções são baseadas em técnicas de tolerância a falhas, as quais, foram incorporadas à implementações em *hardware* dos algoritmos estudados. As soluções apresentadas são capazes de lidar com múltiplas falhas simultaneamente e, em muitos casos a ocorrência de falhas torna-se transparente ao usuário ou atacante. Isso confere um novo nível de segurança, na qual, o atacante é incapaz de ter certeza a respeito da eficácio de seu método de injeção de falhas. A validação foi realizada através de simulações de injeção de falhas simples e múltiplas. Os resultados mostram uma boa eficácia dos mecanismos propostos, desta forma, elevando o nível de segurança nos sistemas protegidos. Além disso, foram mantidos os compromissos com área e desempenho.

Palavras-chave: Smart Cards, Hardware Criptográfico, Fault Attacks, Tolerância a Falhas.

Técnicas para o projeto de *hardware* criptográfico tolerante a falhas

ABSTRACT

This work focuses on the study of a particular kind of attack against cryptographic systems. The hardware implementation of cryptographic algorithms present a number of vulnerabilities not taken into account in the original design of the algorithms. The main targets of such attacks are portable devices which include cryptographic hardware due to limitations in their embedded processors, like the Smart Cards, which are already largely used in GSM mobile phones and are beginning to spread in banking applications. These devices can be attacked in several ways, e.g., by analysing the power consummed by the device, the time it takes to perform an operation, or even by exploring the susceptibility of the hardware to the occurrence of transient faults. These attacks aim to extract sensitive information stored in the device, such as a cryptographic key. Attacks based on the malicious injection of hardware faults are commonly called Differential Fault Attacks (DFA), or simply fault attacks. The goal of the present work was to study how fault attacks are executed against different algorithms, and to propose solutions to avoid such attacks. The algorithms selected for this study were the DES and the AES, both well known and largely deployed. Different solutions to help avoid fault attacks are presented. The solutions are based on fault tolerance techniques, and were included in hardware implementations of the selected algorithms.The proposed solutions are capable to handle multiple simultaneous faults, and, in many cases, the faults are detected and corrected in a way that is transparent for the user and the attacker. This provides a new level of security, where the attacker is unable to verify the efficiency of the fault injection procedure. Validation was performed through single and multiple fault injection simulations. The results showed the efficiency of the proposed mechanisms, thus providing more security to the protected systems. A performance and area compromise was kept as well.

Keywords: Smart Cards, Hardware Criptográfico, Fault Attacks, Tolerância a Falhas.

1 INTRODUÇÃO

O mundo tem passado por mudanças sem precedentes em sua história. Dispositivos eletrônicos estão cada vez mais presentes em nosso dia-a-dia, tornando-se mais sofisticados e realizando tarefas mais complexas. Tais dispositivos estão presentes não apenas em nossas casas, escritórios ou automóveis, mas são também objetos pessoais levados em bolsos, carteiras ou mochilas. O avanço da microeletrônica permitiu a miniaturização, a invenção de novos dispositivos e a agregação de diferentes funcionalidades em um mesmo aparelho portátil. Por exemplo, atualmente um aparelho celular é capaz, além da transmissão e recebimento de voz, de capturar imagens (fotografia e vídeo), fazer transmissão de dados multimídia, permitir acesso a Internet e ainda fornecer jogos.

As mais diversas áreas da atuação humana estão intrinsecamente ligadas aos recentes avanços tecnológicos. Três importantes áreas incluem o ramo bancário, a Internet e a telefonia. Além de tais áreas, muitas outras apresentam algo em comum: a necessidade de mecanismos de segurança cada vez mais robustos. A tecnologia tem causado avanços substanciais no que diz respeito à segurança de informações. Isso instiga a adoção de sistemas digitais para usos que exigem alta segurança. Porém, o uso de tecnologia digital, por exemplo, no ramo bancário, implica que grandes somas financeiras dependem dos mecanismos de segurança do sistema. Outras áreas, como telefonia e Internet, fornecem serviços que não poderiam existir sem um nível apropriado de segurança. Portanto, os mecanismos que garantem segurança estão em constante desenvolvimento. Enquanto que novos métodos são descobertos, outros são aperfeiçoados e destinados a outros usos.

Neste cenário de constante mudanças, dispositivos digitais portáteis estão assumindo papéis em setores onde alta segurança é um fator fundamental. Tais dispositivos são responsáveis, por exemplo, por transações bancárias, identificação de usuários e controle de acesso. Fraudes em tais sistemas podem causar grandes perdas financeiras e outros problemas para as partes lesadas. Portanto, os dispositivos de segurança envolvidos nestes cenários estão em constante prova. Os principais dispositivos digitais usados em sistemas seguros são os *tokens* criptográficos. *Tokens* são dispositivos físicos portáteis capazes de realizar serviços como autenticação de usuários e transações eletrônicas entre outros. Para isso, eles apresentam capacidade de processamento e armazenamento de dados. Tais dispositivos são encontrados em diferentes formatos. Um tipo de *token* que vem ganhando espaço rapidamente são os *Smart Cards*. Tais *tokens* são semelhantes aos cartões de crédito convencionais e são usados em diferentes serviços. Exemplos destes serviços são telefonia móvel e mesmo como substitutos dos atuais cartões magnéticos.

A alta segurança associada aos *tokens* é principalmente devida ao uso de modernos algoritmos criptográficos. Exemplos de algoritmos criptográficos extensamente usados são o DES, AES e RSA. Processamento criptográfico demanda um razoável poder computacional. Muitas vezes os dispositivos criptográficos, como os *Smart Cards*, possuem

processadores embarcados com poder de processamento bastante restrito. Para executar o processamento em tempo aceitável, os algoritmos criptográficos são, normalmente, implementados em *hardware*. Porém, implementações de algoritmos criptográficos em *hardware* apresentam uma série de novas vulnerabilidades, que sua especificação original não previu. O projeto de um algoritmo criptográfico leva em conta a dificuldade de se obter o texto original ou a chave criptográfica a partir de blocos de textos já criptografados. A implementação destes algoritmos em *hardware* e a larga presença deles em dispositivos móveis, torna fácil a posse e exploração das vulnerabilidades por indivíduos não autorizados. As principais técnicas de ataque para *hardwares* criptográficos são: DFA, DPA e *timing analisys*. DFA é uma técnica que explora o comportamento dos algoritmos sob a indução de falhas transientes. DPA é a exploração do consumo energético visando a obtenção da chave criptográfica. Por fim, *timing analisys* permite a dedução de informações sigilosas, a partir da monitoração do tempo de processamento.

Devido às vulnerabilidades presentes em implementações de algoritmos criptográficos em *hardware* e à alta exigência em termos de segurança que envolve as aplicações, o estudo e a determinação dos riscos reais envolvidos são extremamente necessários. A exposição de possíveis falhas nos sistemas já existentes coloca a credibilidade dos mesmos em discussão. Portanto, um estudo apurado dos diferentes tipos de ataques a *hardwares* criptográficos é de suma importância, pois a expansão e a aplicação dos *tokens* em novos serviços depende da aceitação e da confiança da comunidade envolvida. Esta dissertação dedica-se, especificamente, ao estudo de ataques por DFA a algoritmos criptográficos implementados em *hardware*. Todas as áreas que envolvem ataques a *hardwares* criptográficos merecem cuidado. Porém, devido à complexidade de tais ataques, assim como, da elaboração de soluções efetivas, é necessário restringir a área de estudo. Restrito a ataques do tipo DFA, foi possível aprofundar os estudos sobre o ataque, abordar dois diferentes algoritmos e ainda refinar as soluções elaboradas.

Ataques do tipo DFA baseiam-se, principalmente, na injeção de falhas transientes no *hardware* criptográfico. Portanto, as soluções elaboradas são baseadas em esquemas de tolerância a falhas. Muitas soluções propostas anteriormente a este trabalho restringem-se à detecção de falhas, ou no máximo à detecção e correção de falhas simples. Os esquemas de tolerância a falhas para algoritmos criptográficos propostos neste trabalho são capazes de lidar com múltiplas falhas simultaneamente. Isso significa que o *hardware* protegido pode, não apenas detectar, mas também corrigir múltiplas falhas simultaneamente. Tolerância a falhas adiciona um novo nível de segurança se comparado a apenas detecção de falhas. Em um sistema tolerante a falhas, onde a ocorrência de falhas é transparente, ou seja, não causa efeitos visíveis ao atacante, gera dúvidas a ele quanto à efetividade do esquema de injeção de falhas utilizado. Portanto, impedindo ou pelo menos atrasando a efetivação do ataque.

Os algoritmos criptográficos adotados para estudo foram o DES e o AES. O DES foi um algoritmo amplamente utilizado e de fácil entendimento. Portanto, um ótimo algoritmo para realização dos estudos iniciais. O AES é o algoritmo criptográfico adotado como padrão para aplicações comerciais. Ele é amplamente aceito e um dos principais algoritmos da atualidade. Foi adotado em 2001, depois de longos estudos, em substituição ao DES. O trabalho aborda como algoritmos são atacados por DFA, e soluções com base em tais ataques são propostos. Por fim, são apresentados os resultados e é feita uma avaliação das soluções propostas para cada algoritmo.

Este trabalho está organizado conforme explicado a seguir. O Capítulo 2 apresenta os *tokens* criptográficos. São abordadas suas principais funcionalidades e aplicações. É

dado ênfase aos *Smart Cards* por serem amplamente utilizados. A arquitetura e os componentes de *hardware* de um *Smart Card* típico são apresentados. O Capítulo 3 começa com a apresentação dos requisitos para sistemas de segurança, o que inclui conceitos de integridade, sigilo entre outros. Após, são apresentados os mecanismos usados para implementar tais requisitos. Por fim, o capítulo faz uma revisão dos três principais algoritmos criptográficos: DES, AES e RSA. O Capítulo 4 faz uma rápida apresentação das diferentes técnicas de ataques a *hardware* criptográfico. É dado ênfase a ataques por DFA. É mostrado como os algoritmos DES, AES e RSA podem ser quebrados por injeção de falhas. Na sequência, são apresentadas possíveis contramedidas para DFA. O capítulo é encerrado com a exposição dos principais trabalhos relacionados. No Capítulo 5 é realizada uma simulação de ataque por DFA no algoritmo criptográfico DES. Esta simulação inclui a injeção controlada de falhas em uma descrição do DES em linguagem de descrição de *hardware* e o desenvolvimento de um programa em linguagem Java para extração da chave criptográfica a partir de textos cifrados com falhas. O mesmo capítulo apresenta uma contramedida para DFA baseada em duplicação parcial do *hardware*. Esta contramedida é implementada, validada e os resultados são apresentados. O Capítulo 6 apresenta as técnicas de tolerância a falhas formuladas para proteção do AES contra DFA. Foram estudas e implementadas duas técnicas para proteção de memória contra falhas transientes, e outras duas técnicas para proteção das partes combinacionais. Tais técnicas foram combinadas e implementadas resultando em quatro diferentes versões do AES. Cada versão apresenta diferentes características, como área, desempenho e robustez a falhas. Todas as implementações são validadas a partir de injeção exaustiva de falhas simples e múltiplas. Por fim, os resultados são apresentados e comparados. Finalmente, o Capítulo 7 traz a conclusão deste estudo e a apresentação dos trabalhos futuros.

2 DISPOSITIVOS CRIPTOGRÁFICOS PORTÁTEIS

A tecnologia de semicondutores tem permitido o surgimento de dispositivos portáteis com capacidade de processamento e armazenamento úteis para muitas necessidades. O ramo bancário, a Internet, a telefonia e outras aplicações de uso diário, têm estado a par desta evolução tecnológica. Em muitas destas áreas comerciais, principalmente aquelas onde segurança é um fator primordial, novas aplicações têm surgido tomando vantagem de tais dispositivos. Os atuais cartões magnéticos, extensamente usados em aplicações financeiras, estão cedendo lugar a novos dispositivos eletrônicos. Estes dispositivos têm como principal característica a segurança, garantida pelo uso de modernos algoritmos criptográficos. Sua capacidade de processamento associado à segurança tem permitido o surgimento de uma nova gama de aplicações, como dinheiro virtual e transações financeiras sem conectividade com a Internet. Os principais dispositivos que estão permitindo essa evolução são os *tokens* criptográficos. Um exemplo de *token* criptográfico são os *Smart Cards*. Este capítulo tem o objetivo de apresentar tais dispositivos. A sessão 2.1 apresenta um breve histórico do surgimento, benefícios e aplicações dos *Smart Cards*. Ao final desta sessão é apresentado o *hardware* típico de um *Smart Card* com seus principais componentes. Este capítulo 2.2 apresenta outros tipos de *tokens* criptográficos, onde são expostas suas principais características, aplicações e benefícios intrínsecos.

2.1 Smart Cards

O termo *Smart Card* surgiu em 1980, criado pelo publicitário francês Roy Bright. Porém, seu surgimento ocorreu anos antes, em torno de 1970. A atribuição desta invenção gera muitas controvérsias. Em fevereiro de 1969, dois engenheiros alemães chamados Jürgen Dethloff e Helmut Gröttrupp, requereram patente na Alemanha sobre o que seriam os primeiros *Smart Cards*. A patente DE 1945 777 C3 foi concedida em 1982, intitulada "Identificakanden/Identifikationsschalter". Independentemente, Kunitaka Arimura do Instituto de Tecnologia Arimura no Japão requereu patente para um *Smart Card* em março de 1970. Em março de 1971, Paul Castrucci da IBM requereu a patente intitulada *Information Card*, nos Estados Unidos. Entre 1974 e 1979, foram registradas 47 patentes relacionadas a *Smart Cards* e suas possíveis aplicações, em 11 países (JURGENSEN; GUTHERY, 2002).

Atualmente o mercado de *Smart Cards* tem passado por um rápido crescimento. Isso é atribuído a diversos fatores. As principais patentes já expiraram permitindo que novas companhias entrem no mercado, muitas delas trazendo importantes inovações. A preocupação com a segurança dos cartões magnéticos tem crescido. Isto, associado com a evolução da microeletrônica e o baixo custo de produção dos CI's (Circuitos Integrados), tem tornado viável a substituição dos cartões magnéticos por *Smart Cards*. Outro fator

determinante para a rápida expansão do mercado é o surgimento de diversas aplicações que exigem cartões com capacidade de processamento, por exemplo, *health cards*, telefonia móvel (GSM - *Global System for Mobile Communications*) e decodificação de sinais de satélite televisivos (HENDRY, 1998).

A Figura 2.1(a) exibe a aparência de um *Smart Card* típico. Na parte externa, o cartão pode apresentar logotipos da empresa ou sistema ao qual o cartão pertence, assim como foto, nome e dados pessoais de seu portador. A Figura 2.1(b) apresenta detalhes dos pinos de comunicação e do CI logo abaixo da interface de comunicação. Se no passado os custos e falta de aplicações convenientes para seu uso impediram a expansão dos *Smart Cards*, atualmente existem fortes razões para investimentos, pesquisas e desenvolvimento desta tecnologia.

(a) Aparência típica de um *Smart Card*.

(b) Detalhe dos pinos de comunicação e do CI do cartão.

Figura 2.1: Aparência e localização do CI nos *Smart Cards* atuais.

2.1.1 Benefícios

O crescente interesse em *Smart Cards* resulta dos benefícios que eles podem prover. Seu principal benefício é a capacidade de processamento. Outras características como portabilidade e facilidade de uso são fatores chave para sua ampla aceitação. Os usuários podem carregar os *Smart Cards* em uma carteira, assim como fazem com cartões magnéticos convencionais. Seu uso também é semelhante aos cartões magnéticos, pois basta inseri-lo em um leitor apropriado ao iniciar a transação e retirá-lo ao final (CHEN, 2004). Além disso, algumas aplicações exigem capacidade de armazenamento, o que um cartão magnético não pode disponibilizar. Um exemplo básico são os cartões com informações a respeito da saúde de pacientes, chamados de *health cards*. Outro benefício marcante é a segurança que está intrinsecamente associada à sua capacidade de processamento. *Smart Cards* possuem diversas características que permitem não apenas armazenar informações de forma segura, mas também se comunicar com outros sistemas computacionais com segurança (HENDRY, 1998).

Smart Cards são resistentes a ataques porque não dependem de recursos externos potencialmente vulneráveis (CHEN, 2004). Obter informações confidenciais do cartão exige acesso a ele, bons conhecimentos do seu *hardware* e *software* e equipamentos adicionais. A segurança do cartão é baseada principalmente no uso de algoritmos criptográficos. Os

dados armazenados no cartão são criptografados, assim como os dados trocados entre o cartão e mundo externo. Associado a isso, o usuário do cartão possui um número denominado PIN (*Personal Identification Number*) usado para acessar o cartão. O PIN evita que o cartão seja usado por pessoas não autorizadas.

2.1.2 Aplicações

A grande motivação para introdução dos *Smart Cards* no dia-a-dia das pessoas é a segurança, seguida pela sua capacidade de armazenamento de informações e facilidade de uso. Com isso, eles são freqüentemente usados para armazenamento e autenticação, garantindo segurança nas transações. As subseções a seguir ilustram algumas aplicações bastante difundidas atualmente e que exigem uma ou várias destas características.

2.1.2.1 Telefonia e Telecomunicações

Uma das primeiras aplicações em larga escala envolvendo *Smart Cards* foi na telefonia pública francesa e alemã, no inicio dos anos 80 (CHEN, 2004). Os cartões armazenavam créditos que eram debitados durante as chamadas. Quando os créditos se esgotavam, o usuário podia recarregar seu cartão em algum posto autorizado, evitando o desperdício de cartões. Este esquema oferecia um mecanismo com baixo custo de manutenção e seguro contra fraudes para acesso à telefonia pública.

Atualmente, a indústria de telecomunicações tem usado largamente, *Smart Cards* na telefonia móvel para garantir segurança. O melhor exemplo é o padrão GSM adotado em diversos países. O futuro da telefonia móvel é prover muito mais que apenas comunicação de voz. A intensa competitividade das empresas de telecomunicações, força com que estas agreguem novos serviços rapidamente, tais como, acesso à banco, comércio e acesso a Web entre outros, a partir de dispositivos móveis. No padrão GSM, *Smart Cards* são a base para verificar, identificar e garantir segurança à transmissão de dados (CHEN, 2004).

As companhias de TV a cabo e satélite oferecem planos diferenciados aos seus assinantes. Ao contrário das redes de computadores, onde os serviços disponibilizados para determinados usuários podem ser controlados pelos provedores de acesso, as operadoras de TV não podem controlar o que será transmitido para cada assinante. Desta forma, o controle dos serviços ou as restrições de uso impostas pelo plano do assinante, devem ser feitas na própria casa do assinante através do dispositivo receptor. Assim, o sinal transmitido é criptografado (HENDRY, 1998) e o dispositivo receptor de cada usuário deve ser responsável em descriptografar e disponibilizar os serviços de direito do assinante. O dispositivo receptor faz uso de um *Smart Card* fornecido pela operadora de TV. Este *Smart Card* contém as chaves criptográficas necessárias e as informações dos serviços disponíveis para o assinante. O *Smart Card* pode também prover outros serviços, como controle de programas aceitáveis para crianças. Através deste esquema também é possível o envio de mensagens diferenciadas para determinados grupos ou regiões (HENDRY, 1998).

2.1.2.2 Aplicações Financeiras

A principal aplicação para *Smart Cards* é no ramo de serviços bancários. Existem três principais usos: débito, crédito e armazenamento de valores monetários. Atualmente, os cartões magnéticos conseguem prover apenas dois destes serviços: débito e crédito. A introdução de *Smart Cards* no ramo bancário habilita o armazenamento de dinheiro no próprio cartão permitindo uma gama de novas aplicações. Este sistema favorece aplicações que não podem manter a conectividade com a Internet todo o tempo. Como exemplo

pode-se citar o pagamento de passagem de ônibus urbano sem a necessidade de um cobrador. Neste sistema, ao entrar no ônibus o usuário insere seu *Smart Card* em um leitor que debita o valor da passagem diretamente do cartão, destravando a roleta. Ao final do dia, os valores monetários armazenados no ônibus são transferidos para o sistema da empresa. A contribuição dos *Smart Cards* para os já existentes cartões de débito e crédito é a possibilidade de transações *off-line*, além de uma camada adicional de segurança, conforme será discutido a seguir.

Para suportar tais aplicações os *Smart Cards* devem possuir comandos especialmente projetados para prover transações seguras. Um requisito importante é a atomicidade das transações. Isso significa que se o objetivo é identificar o usuário e transferir um montante de dinheiro de seu cartão para o sistema em uso, todas as operações devem ser executadas como um único procedimento. Assim, uma operação é inteiramente realizada ou falha por completo. Este esquema evita que valores sejam criados ou destruídos (JURGENSEN; GUTHERY, 2002).

O meio mais comum de fraude com cartões de crédito magnéticos é o roubo. Quando um cartão é perdido ou roubado, quem possuir o cartão é capaz de realizar transações já que o cartão não exige uma senha e muitas vezes o lojista não confere a assinatura do cliente. Desta forma, enquanto o cartão não é bloqueado, o uso malicioso pode causar danos financeiros graves. O principal requisito para um cartão de débito ou crédito inteligente é reduzir as oportunidades de fraude sem aumentar a necessidade de autenticação *on-line* (HENDRY, 1998). Além disso, ele precisa se tornar financeiramente viável para pequenas transações, às vezes, de apenas alguns centavos. Isso aumenta a necessidade de transações *off-line* seguras.

Durante uma transação, ocorre a identificação do cartão, a fim de determinar a autenticidade do mesmo. Em seguida é necessário identificar o portador do cartão, para evitar o uso de cartões roubados ou perdidos. A identificação do cartão é feita por mecanismos criptográficos que serão melhor detalhadas no Capítulo 3. O PIN, conforme mencionado anteriormente (Subseção 2.1.1) é requerido para identificação do usuário. Ambas operações podem ocorrer *off-line*.

2.1.3 Organização e Arquitetura de *Smart Cards*

O principal componente de um *Smart Card* é o circuito integrado presente dentro do cartão. Este circuito embarcado tem se tornado bastante complexo, e atualmente possui diferentes componentes formando, assim, um SoC (*System on Chip*). O SoC deve suportar CPU (*Central Processor Unit*), todas as memórias necessárias (ROM - *Read Only Memory*, EEPROM - *Electrically Erasable and Programable ROM* e RAM - *Random Access Memory*), co-processadores e ainda as interfaces de I/O (*Input/Output*). A Figura 2.2 mostra a arquitetura de um SoC que pode ser utilizada para diferentes aplicações, inclusive Smart Cards.

Inicialmente, os processadores embarcados em *Smart Cards* possuíam capacidade de processamento de apenas 8-bit. Exemplos de processadores que foram muito usados para este tipo de aplicação são o Motorola 6805 e o Intel 8051, com freqüência de clock de até 5 *MHz*. Atualmente, com os grandes avanços na concepção de CI's e o surgimento de novas aplicações que exigem maior poder de processamento passou-se a utilizar processadores de 16 ou até 32-bit. Kim et. al. (KIM; AL., 2003) sugere uma arquitetura de 32-bit para *Smart Cards*. Tal arquitetura é apresentada na Figura 2.3.

Essa plataforma foi desenvolvida para atender a requisitos de comunicação com ou sem fio. Desta forma, possui duas interfaces de comunicação: *contact* (possui pinos

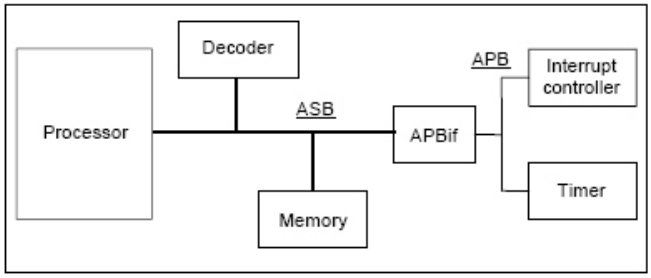


Figura 2.2: Componentes de um SoC básico, adaptado de (KIM; AL., 2003)

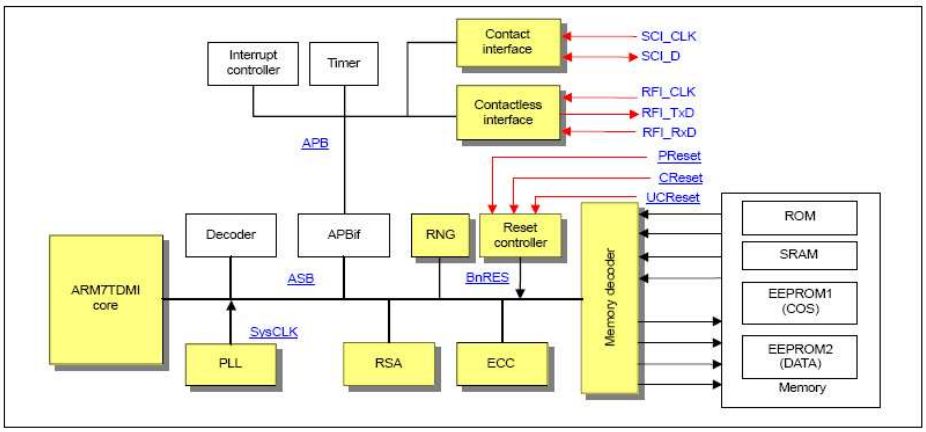


Figura 2.3: Arquitetura de 32-bit para *Smart Cards*

de contato) e *contactless* (funciona por radiofreqüência). Ambas interfaces seguem os padrões ISO/IEC 7816 (ISO/IEC 7816, 1998) e ISOC/IEC 14443 (ISO/IEC 14443, 2000).

O processador embarcado é um ARM7TDMI de 32-bit que atende a requisitos de baixo consumo, fator importante para *Smart Cards* que possuem interface *contactless*. O SoC possui três tipos distintos de memória: ROM (16 Kb), SRAM (16 Kb) e EEPROM (192 Kb). A ROM armazena as rotinas de inicialização, assim como as rotinas para descarga do SO (Sistema Operacional) do cartão. A SRAM é a memória de programa. A EEPROM armazena o SO e as informações não voláteis do usuário. A conectividade entre os módulos é dada por barramentos que seguem o padrão AMBA (*Advanced Microcontroller Bus Architecture*) (LTD, 1997). O barramento ASB (*Advanced System Bus*) é usado em conexões entre módulos que exigem maior velocidade de comunicação como, por exemplo, processador, co-processadores e memória. O barramento APB (*Advanced Peripherical Bus*) é usado na interconexão de periféricos que exigem taxas de comunicação menores visando reduzir o consumo de potência. Exemplos destes periféricos são: controlador de interrupção, temporizadores e interfaces de comunicação.

É importante notar que a arquitetura descrita aqui adere a duas tendências: permitir que *Smart Cards* rodem múltiplas aplicações e a interoperabilidade. Rodar múltiplas aplicações permite que um único cartão seja utilizado para diferentes serviços. Para facilitar o desenvolvimento de multi-aplicações, assim como agregar segurança ao software embarcado, uma plataforma de desenvolvimento baseada em Java, denominada Java Card (CHEN, 2004), tem se tornado muito popular. A interoperabilidade permite manter a compatibilidade entre os cartões produzidos pelos diversos fabricantes e seus leitores de cartões. Por isso, existe a padronização das interfaces de comunicação assim como dos protocolos que trafegam por essas interfaces.

2.1.3.1 Co-processador Criptográfico

Uma importante característica do SoC proposto por Kim et. al. (KIM; AL., 2003), designado para *Smart Cards*, é a presença de dois *hardwares* distintos dedicados à criptografia e ainda um gerador de números aleatórios. O surgimento dos *Smart Cards* está ligado à necessidade de agregar mais segurança aos processos digitais. Por sua vez, a criptografia de dados é a forma usual de garantir segurança a tais processos. Conforme será abordado no Capitulo 3, os algoritmos criptográficos são dispendiosos demais para rodar nos processadores embarcados destes dispositivos. Tais processadores são projetados com metas rígidas de economia de energia e área em detrimento de sua capacidade de processamento. Assim, desde o surgimento dos *Smart Cards* processadores dedicados à criptografia fazem parte de seu *hardware*, com a função de auxiliar o processador principal em funções criptográficas complexas.

O SoC em questão, conforme a Figura 2.3, possui um co-processador que implementa uma versão do algoritmo RSA de 1024-bit. Tal co-processador pode executar multiplicações e exponenciações modulares para dados de 1024-bit, usando o algoritmo de Montgomery. Desta forma, é capaz de codificar ou decodificar um bloco de 1024-bit de dados em aproximadamente 180 ms, rodando a 10 *MHz*. O segundo co-processador criptográfico executa uma versão do algoritmo ECC de 163-bit. Este co-processador pode executar operações de multiplicação escalar e polinomial e multiplicação polinomial inversa em GF(2^{163}). Um *hardware* dedicado denominado RNG (*Random Number Generator*) é capaz de gerar números aleatórios de 32-bit usados nas operações criptográficas (KIM; AL., 2003).

Diferentes aplicações podem exigir algoritmos criptográficos diferenciados. Portanto, outras plataformas apresentam co-processadores que implementam algoritmos criptográficos distintos, como o DES (ver Subseção 3.3.1), Triplo-DES e AES (ver Subseção 3.3.2).

2.2 Outros tipos de *Tokens* Criptográficos

Tokens criptográficos (ou às vezes chamado de *hardwares tokens*, *security tokens* ou mesmo *authentication tokens*) podem ser usados para auxiliar usuários no processo de autenticação em sistemas computacionais. A maioria dos computadores e redes usa nomes de usuários e senhas simples para proteger-se de acessos ilegais. Porém, em certos casos o uso de senhas convencionais não é aceitável, pois elas podem ser facilmente descobertas ou compartilhadas. Alguns tokens possuem interfaces de comunicação facilmente encontradas nos PC's (*Personal Computer*) atuais, justamente por estarem associados a serviços ligados a eles.

A Figura 2.4 mostra dois modelos de *tokens* criptográfico. Os *tokens* são tipicamente pequenos o suficiente para serem levados em uma carteira. Freqüentemente são projetados para poderem ser levados em um chaveiro. Alguns destes dispositivos são bastante simples, enquanto outros oferecem diversos métodos de autenticação. Alguns *tokens* são projetados para operar com mecanismos de comunicação *bluetooth*. Normalmente, são combinados com um controlador USB (Figura 2.4(a)) provendo duas formas de comunicação com dispositivos externos. Porém, é comum encontrar *tokens* que operam independentemente de qualquer outro dispositivo. Estes apenas geram um número de autenticação que o usuário deve inserir manualmente no sistema. Um exemplo deste tipo de *token*, é mostrado na Figura 2.4(b). A maioria deles apresenta características para impedir tentativas de acesso indevido ao *hardware* (*tamper resistance*).

(a) USB Token desenvolvido pela Aladdin Knowledge Systems eToken (ETOKEN, 2006a).

(b) SecurID desenvolvido pela RSA Security (SECURITY, 2006).

Figura 2.4: Exemplos de tokens criptográficos.

2.2.1 Autenticação Forte

Um *token* pode oferecer maior segurança porque permite esquemas de autenticação mais fortes. Autenticação forte, ou autenticação de dois fatores, comumente envolve um dispositivo físico usado para provar a identidade do usuário. A autenticação tradicional exige apenas um fator, o conhecimento da senha, enquanto que a autenticação forte é baseada em dois ou mais dos seguintes fatores:

- Algo que o usuário sabe: Algo que usuário precise se lembrar, como uma senha, um PIN ou a resposta a uma questão pessoal.
- Algo que usuário tenha: Algo que o usuário precise carregar fisicamente, por exemplo, um *token* ou um cartão.
- Algo que o usuário é: Características biométricas do usuário, como impressões digitais ou características faciais.

O emprego da autenticação de dois fatores foi facilitada pela introdução dos *tokens* criptográficos, principalmente *Smart Cards*, como um segundo fator de autenticação. Neste caso, algo que o usuário tenha (*token* ou *Smart Card*) é associado a algo que usuário sabe (senha ou PIN).

2.2.2 Principais Aplicações

A Figura 2.5 cita apenas alguns dos diversos serviços oferecidos pelos *tokens*. Eles podem armazenar chaves criptográficas, certificados digitais ou informações biométricas, como impressão digital. Assim, é possível prover acesso seguro a VPN's (*Virtual Private Network*), assinatura digital, criptografia de dados em discos e autenticação de inicialização (ETOKEN, 2006b).

Tokens podem ser empregados para permitir acesso seguro à rede interna de uma organização. Garantindo desde acesso local seguro (identificação de usuários internos para permitir acesso aos servidores da companhia) até acesso remoto seguro (identificação de usuários externos para prover comunicação segura entre uma máquina remota e a VPN ou servidor Web). Permitir acesso remoto para empregados, clientes, fornecedores e sócios permite prover novas ferramentas e serviços (ETOKEN, 2006b).

Figura 2.5: Exemplos de serviços fornecidos por *tokens*.

Single Sign-On (SSO) é um esquema de identificação de usuários que permite que um usuário identifique-se uma vez e ganhe acesso a múltiplos recursos em um sistema. Alguns tipos de soluções SSO usam *tokens* para armazenar o *software* que permite a autenticação.

Autenticação de pré-inicialização é uma solução que visa proteger a inicialização para evitar que um usuário malicioso possa inicializar o sistema. Assim, rotinas de autenticação são carregadas antes da inicialização do sistema operacional e apenas após a devida identificação do usuário o sistema operacional pode ser iniciado. Outro esquema de proteção associado a PC's é a criptografia de disco. Os arquivos de usuário são armazenados de forma criptografada. Assim, mesmo com o roubo do equipamento, os dados permanecem invioláveis. Este tipo de solução é normalmente transparente ao usuário, sendo que este, apenas necessita conectar o *token* ao PC e entrar com seu PIN.

3 SEGURANÇA EM DISPOSITIVOS CRIPTOGRÁFICOS PORTÁTEIS

O Capítulo 2 apresentou os dispositivos criptográficos portáteis mais utilizados. Para prover segurança de dados, tais dispositivos implementam técnicas de segurança que incluem desde protocolos de comunicação até algoritmos criptográficos complexos. Este capítulo tem o objetivo de apresentar os principais mecanismos utilizados na segurança de dados. Na Seção 3.1 são discutidos os requisitos que um sistema seguro deve incorporar. Na Seção 3.2, é discutido como tais requisitos podem ser atendidos. Por fim, a Seção 3.3 apresenta os principais algoritmos criptográficos utilizados na atualidade.

3.1 Requisitos de Segurança

Existem diferentes formas de um sistema falhar colocando em risco sua segurança. O risco associado à falha de um sistema pode envolver perdas financeiras, exposição de informações sigilosas ou até mesmo perda de vidas humanas . *Smart Cards* estão associados principalmente a operações financeiras e sigilosas, porém, podem também estar envolvidos em sistemas de suporte à vida humana (HENDRY, 1998). A maioria dos sistemas computacionais, incluindo *Smart Cards*, deve reunir alguns dos requisitos abaixo:

- Segurança de vidas humanas: Aviões e naves espaciais exigem sistemas computacionais para navegação e suporte à vida. Porém, tais sistemas estão sujeitos a falhas e a construção destas aeronaves, sem o suporte computacional é impossível. Assim, esquemas de tolerância a falhas devem ser implementados visando diminuir drasticamente os riscos de falhas. Eventualmente, *Smart Cards* podem fazer parte de sistemas que envolvem risco de perda de vidas humanas à medida que são aplicados em certas instalações, como centrais nucleares. Os cartões podem ser usados para dar acesso a áreas de alto risco. Nestes casos é sempre importante avaliar quais são os riscos e a real necessidade de esquemas de tolerância a falhas, por causa do custo associado.

- Perda de mensagens: Computadores ligados a redes, como a Internet, lidam com perda de mensagens há muito tempo. O custo da perda de mensagens depende de como o protocolo controla estas perdas. Normalmente, mensagens enumeradas permitem o reenvio apenas das mensagens perdidas. *Smart Cards* continuamente precisam lidar com transações e eventos. É fácil perder eventos, a menos que estes estejam enumerados. Isso também evita mensagens duplicadas. É necessário recuperar mensagens perdidas para poder finalizar as transações corretamente.

- *Accuracy*: Este requisito aborda a possibilidade dos dados serem corrompidos durante a transmissão ou processamento. Erros normalmente ocorrem por causa de maus contatos, interferência elétrica ou diferentes formas de radiação. *Smart Cards* devem lidar com estas falhas. Conforme será abordado no Capítulo 4, falhas transientes, injetadas maliciosamente no sistema, podem levar à obtenção dos dados sigilosos do cartão. Portanto, esquemas de proteção são comumente adotados. Paridade, CRCs (cyclic redundancy checks), circuitos *self-checking* e MACs (*Message Authentication Checks*) são usados para proteger o processamento e transmissão de informações críticas do sistema.

- Integridade dos dados: Os dados armazenados no cartão devem ser protegidos contra alterações, sejam estas acidentais ou maliciosas. Ataques maliciosos sobre a integridade dos dados armazenados são extremamente graves quando eles envolvem chaves criptográficas ou códigos de autorização (HENDRY, 1998). Projetistas de *Smart Cards* devem assumir que tais ataques irão ocorrer e projetar sistemas capazes de lidar com a integridade em face destes ataques. Os Capitulos 5 e 6 abordam esquemas de tolerância a falhas para os algoritmos criptográficos DES e AES, respectivamente

- Sigilo: A grande maioria das aplicações para *Smart Cards* exige que, o sigilo ou privacidade das informações processadas ou armazenadas seja assegurado. Tais aplicações não apenas armazenam informações médicas ou limites de créditos, mas também dados de acesso a outros sistemas que mantêm informações importantes, cuja violação implica em sérios danos. É importante tomar conta dos riscos envolvidos na questão de sigilo e das possíveis medidas para evitar grandes problemas (HENDRY, 1998). Criptografia é a ferramenta mais freqüentemente usada para garantir sigilo de informações. A Seção 3.3 mostrará os principais algoritmos criptográficos utilizados em Smart Cards.

- Não repúdio: Não repúdio significa que, uma vez realizada uma transação, ambos, cliente e comerciante, não podem negar tal ocorrência. Assinatura digital, usando criptografia de chave pública, pode garantir este requisito. A assinatura digital é extensamente usada na Web em diversos tipos de transações eletrônicas e passaram a ser uma importante ferramenta em sistemas baseados em Smart Cards. A Subseção 3.2.1.2 explicará em maiores detalhes a assinatura digital.

Com base nestes requisitos percebe-se que existem dois importantes fatores para impedir que usuários maliciosos obtenham informações sensíveis. Um dos fatores é a robustez dos protocolos e do *hardware* em tolerar falhas. Os protocolos de comunicação devem ser capazes de detectar e corrigir falhas nas transmissões, enquanto que o *hardware* deve ser capaz de processar as informações detectando e, se possível, corrigindo eventuais falhas. O outro fator é o uso de criptografia para sigilo das informações. Na Seção 2.1.3 foi visto que *Smart Cards* utilizam algoritmos criptográficos implementados em *hardware* para melhorar o desempenho das aplicações. No Capítulo 4 será mostrado que a ocorrência de falhas em determinados pontos do *hardware* criptográfico leva á obtenção de informações sigilosas. Portanto, tolerância a falhas deve fazer parte dos co-processadores criptográficos presentes nos cartões.

3.2 Mecanismos de segurança

Esta sessão trata exclusivamente de mecanismos criptográficos de segurança. Criptografia é usada para autenticar as entidades do sistema, como usuários, cartões e terminais, e também para criptografar a comunicação do cartão com o mundo externo. Diferentes esquemas criptográficos permitem diferentes serviços. O objetivo desta sessão é explicar como os diferentes serviços (autenticação, integridade, não repúdio e privacidade) são obtidos com o uso de algoritmos criptográficos.

3.2.1 Autenticação e Integridade

Autenticação é um processo semelhante à apresentação de um documento físico, onde a assinatura é única e pode ser reconhecida por todas as partes envolvidas na transação. Estritamente associada à autenticação existe a integridade, que permite determinar se a informação foi de alguma forma alterada (DREIFUS, 1998). Eletronicamente, autenticação e integridade envolvem funções *hash* e assinatura digital. Um valor numérico é gerado a partir do documento, o qual deseja-se autenticar, este valor é assinado digitalmente e enviado ao destinatário.

3.2.1.1 Funções Hash

Funções *hash* tomam uma mensagem como entrada e produzem uma saída denominada código *hash*, valor *hash* ou simplesmente *hash*. Estas funções mapeiam extensas mensagens para pequenos valores de tamanho determinado. Este valor é como um resumo da mensagem servindo para identificar univocamente o documento original. O *hash* de uma mensagem é anexado a mensagem original e envido ao destinatário que, por sua vez, recalcula o *hash* e confere se este é igual ao enviado. Se eles forem diferentes, isso indica que a integridade do documento original foi violada. Existem duas classes de funções hash (Figura 3.1), segundo (MENEZES; OORCHOT; VANSTONE, 1996):

- *Modification detection codes* (MDCs): MDCs tomam como entrada apenas a mensagem original e fornecem como saída um código *hash*. Este tipo de função *hash* é combinado com outros mecanismos criptográficos para permitir autenticação.
- *Message Authentication Codes* (MACs): MACs tomam como entrada não apenas a mensagem original, mas também uma chave simétrica (ver seção 2.2.2) do usuário. Assim MAC's fornecem um mecanismo de autenticação e integridade sem uso de algoritmos criptográficos adicionais.

A partir de uma mensagem de entrada, qualquer pessoa pode calcular o *hash* usando MDCs, porém somente quem tiver a chave criptográfica pode gerar o *hash* correto usando MAC. Assim, MDCs, sem o auxílio de outros mecanismos, permitem apenas detecção de modificação, enquanto que MACs, por si só, permitem autenticação e integridade. É importante lembrar que uma função *hash* é uma função sem retorno, ou seja, a mensagem original não pode ser recuperada a partir de seu *hash*. Por isso, o código *hash* deve ser anexado a mensagem original e ambos enviados ao destinatário. *Smart Cards* usam autenticação para estabelecer um ambiente seguro onde os participantes, mesmo sem conhecer as outras partes envolvidas, possam confiar (JURGENSEN; GUTHERY, 2002).

3.2.1.2 Assinatura digital

Na maioria das vezes, funções *hash* são associadas a algoritmos criptográficos para fornecerem autenticação. Apesar de as que funções MACs permitem o uso de senhas si-

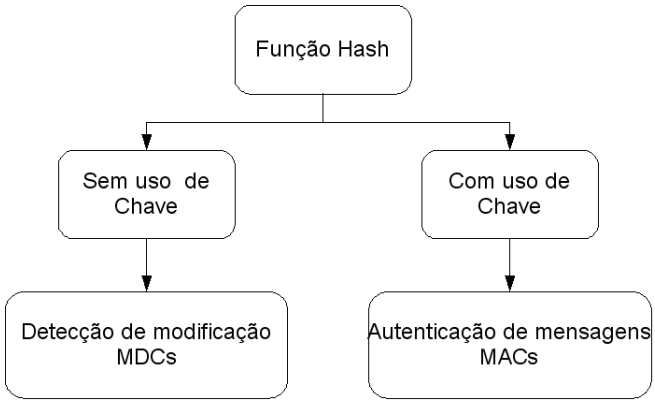


Figura 3.1: Classificação simplificada das funções *hash*.

métricas, esquemas mais robustos são necessários. Algoritmos criptográficos simétricos são aqueles que usam a mesma chave para criptografar e descriptografar os dados. As propriedades destes algoritmos serão mais bem detalhadas na Subseção 3.2.2. Por enquanto é importante saber que para troca de documentos entre duas partes interessadas (A e B) basta que ambas tenham conhecimento da chave secreta. Assim, ao receber um documento criptografado, o destinatário B sabe que a única parte capaz de gerar tal documento é o remetente A, pois além de B, A é o único possuidor da chave secreta. Porém, neste mesmo cenário, A pode negar a autoria do documento, já que B poderia ter gerado o documento com uso da chave secreta compartilhada entre eles. Neste caso, é impossível para uma terceira parte julgar quem está falando a verdade. Assim, algoritmos simétricos não fornecem todas as propriedades necessárias para resolver tal problema, e uma nova classe de algoritmos criptográficos precisou ser criada.

O conceito de assinatura digital surgiu em 1976 através do trabalho publicado por Diffie e Hellman (DIFFIE; HELLMAN, 1976). Tal conceito foi posto em prática com o surgimento dos algoritmos criptográficos de chave pública, ou também conhecidos como assimétricos. Em 1977, três pesquisadores (Rivest, Shamir e Adleman) desenvolveram o primeiro algoritmo desta nova classe, denominado RSA (RIVEST; SHAMIR; ADELMAN, 1977). A principal propriedade desta classe de algoritmos é o uso de chaves distintas para criptografar e descriptografar os dados. Os detalhes de geração e aplicação das chaves no processo criptográfico serão vistos na Subseção 3.3.3. Por enquanto, é importante saber que um usuário detém duas chaves, denominadas pública e privada. Como o próprio nome diz, uma chave é de domínio privado, ou seja, apenas o usuário conhece e a outra é de domínio público, ou seja, é de conhecimento de todos. Quando um texto é criptografado com uma chave privada, apenas a chave pública correspondente pode ser usada para descriptografar o texto.

A Figura 3.2 ilustra o esquema de geração e verificação de assinaturas digitais com o uso de algoritmos assimétricos e funções *hash*. A primeira etapa de geração de uma assinatura digital é a aplicação da função *hash* para gerar um pequeno resumo do documento. Em seguida o remetente ou emissor criptografa o código *hash* com o uso de um algoritmo de chave pública usando sua chave privada. O último passo do emissor é anexar o *hash* criptografado ao documento original e enviá-lo ao destinatário. Uma vez em posse do documento assinado digitalmente, o destinatário calcula novamente o *hash* a partir da mensagem original. Em seguida, decriptografa o código *hash* enviado pelo emissor. Neste ponto é necessário ter conhecimento da chave pública do emissor, pois apenas ela é capaz de descriptografar corretamente a mensagem. Em posse dos dois códigos *hash*

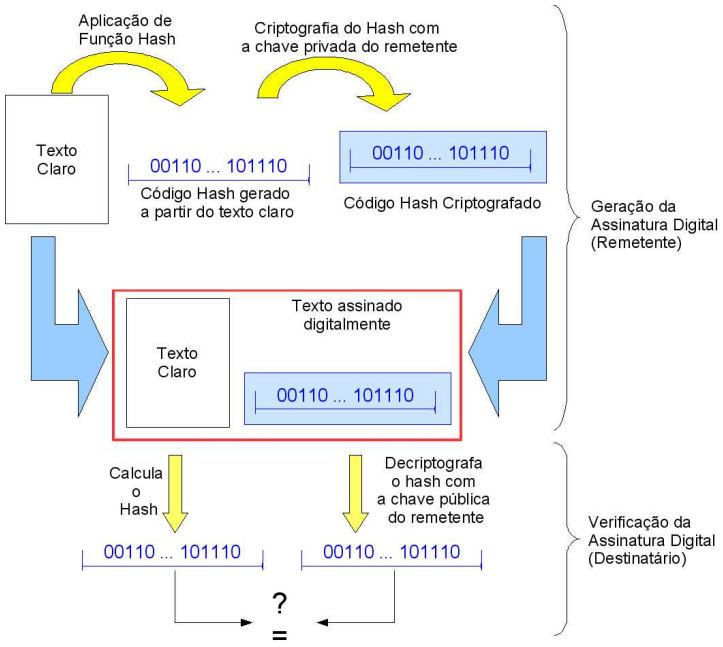


Figura 3.2: Geração e verificação de assinaturas digitais.

(o calculado novamente e o descriptografado) o destinatário os compara. Se forem iguais significa que a mensagem chegou corretamente e que o emissor é realmente o esperado. Caso sejam diferentes isso implica que o documento foi modificado ou a assinatura não foi gerada com a chave privada do emissor.

Uma questão importante é porque não criptografar diretamente o documento original com a chave privada do emissor, obtendo assim assinatura digital e privacidade? Um problema que envolve algoritmos assimétricos é que eles são computacionalmente dispendiosos se comparados com os algoritmos simétricos. Portanto, por questões de desempenho ao invés de criptografar o documento inteiro é preferível criptografar apenas seu código *hash*.

3.2.2 Privacidade

Privacidade é o uso da criptografia para impedir que pessoas não autorizadas tenham acesso às informações. Estas informações podem estar trafegando por uma rede de dados, armazenadas em discos permanentes ou mesmo em mensagens trocadas entre um *Smart Card* e seu leitor. Por exemplo, em uma transação financeira com o uso de cartões inteligentes, o número da conta e outras informações relevantes para a transação são trocadas entre o cartão e o leitor. A obtenção de tais informações por partes não autorizadas acarreta um grande risco ao usuário. A criptografia assimétrica também pode ser usada para obter privacidade, porém seu uso é restrito a casos onde pequenas quantidades de informação precisam ser criptografadas. Para processos criptográficos que exigem privacidade de uma grande quantidade de informação, ou mesmo em casos onde existe uma intensa troca de mensagem entre as partes envolvidas, a criptografia simétrica é aplicada. Este tipo de criptografia é menos intensiva computacionalmente. A criptografia simétrica também é conhecida como criptografia de chave única ou de chave secreta. Algoritmos criptográficos simétricos fazem uso de uma única chave para criptografar e descriptografar os dados (JURGENSEN; GUTHERY, 2002). Isso é ilustrado na figura 3.3. A chave criptográfica é de conhecimento de ambas as partes envolvidas e a revelação da chave por

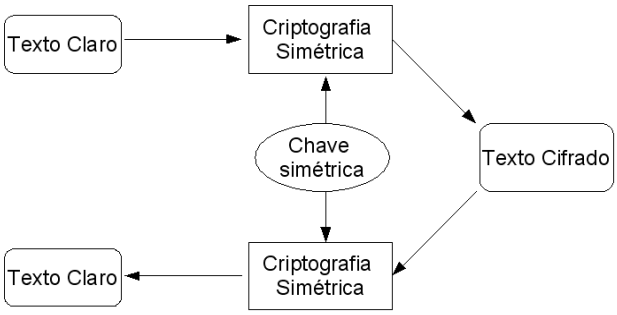


Figura 3.3: Criptografia com chave simétrica. Adaptado de (JURGENSEN; GUTHERY, 2002).

uma das partes compromete ambos.

Dois exemplos de algoritmos simétricos amplamente usados são o DES (MENEZES; OORCHOT; VANSTONE, 1996) e o AES (STINSON, 2002). Estes serão detalhados nas Subseções 3.3.1 e 3.3.2.

3.3 Algoritmos Criptográficos

Esta seção trás uma visão mais detalhada a respeito de três algoritmos criptográficos amplamente utilizados na Web. Tais algoritmos são: AES e DES (criptografia simétrica) e RSA (criptografia assimétrica). Sabe-se até aqui que *Smart Cards* fazem um extenso uso de criptografia para obter segurança. Conforme visto no Capitulo 2, por questões de desempenho tais algoritmos são implementados em *hardware* quando aplicados a *Smart Cards*. As subseções a seguir darão uma breve visão dos algoritmos DES, AES e RSA, respectivamente. Será dado maior ênfase nos algoritmos DES e AES, pois foram os alvos de estudo deste trabalho.

3.3.1 Data Encryption Standard (DES)

O DES (*Data Encryption Standart*) foi o algoritmo padrão usado na criptografia simétrica por cerca de 20 anos. Ele foi selecionado como algoritmo criptográfico padrão pela NSA (*National Security Agency*) nos Estados Unidos em 1976 e acabou sendo largamente difundido pelo mundo todo. Inicialmente, sua adoção gerou muitas controvérsias com relação à sua segurança. Imaginava-se que a NSA poderia conhecer alguma fraqueza do algoritmo. Além disso, apesar de sua chave ter tamanho de 64-bit, apenas 56-bit são efetivos. Os 8 bits restantes são usados como paridade. A adoção de uma chave de 56-bit ao invés de 64-bit gerou a desconfiança de que isto estava sendo feito para facilitar um possível ataque por força bruta (MENEZES; OORCHOT; VANSTONE, 1996). O DES passou por uma intensa análise acadêmica, mas até hoje os melhores ataques ainda são pouco efetivos.

O DES é uma combinação de uma estrutura de Feistel e um cifrador de produto que processa blocos de entrada com 64-bit e fornece blocos criptografados de mesmo tamanho, tomando como entrada uma chave de 56-bit. Um cifrador de Feistel (SIMMONS, 1991) é um algoritmo que tem a seguinte estrutura:

$$L_{i-1} \oplus f(R_{i-1}, K_i)$$

Onde L_i e R_i é o texto de entrada divido em dois blocos de mesmo tamanho. O bloco

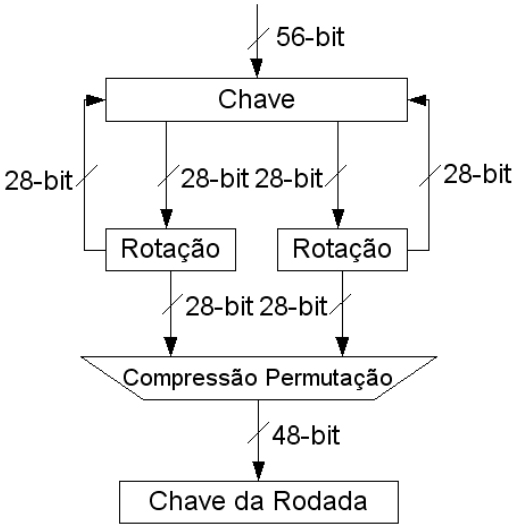


Figura 3.4: Diagrama de blocos do algoritmo de expansão de chaves.

R_i passa por uma função que envolve a chave K_i. Ao final desta função, o texto de saída passa por uma operação ou-exclusivo com L_i. O texto pode passar por diversas iterações nesta função. Esta estrutura apresenta uma interessante característica, onde a mesma função criptográfica f é capaz de criptografar e descriptografar os dados simplesmente invertendo a ordem das chaves (K). Isto torna desnecessária a implementação de uma segunda função para descriptografar os dados. A idéia básica de um cifrador de produto é construir funções criptográficas complexas a partir da composição de operações básicas, as quais, oferecem complementaridade. Transposição, transformações lineares, operadores aritméticos, multiplicação modular e substituições simples são exemplos de operações básicas (MENEZES; OORCHOT; VANSTONE, 1996). No caso do DES, a função f é composta por operações simples que agregadas formam uma função criptográfica complexa.

A criptografia no DES é realizada através de duas operações básicas de criptografia: confusão e difusão. A principal operação realizada no DES é uma combinação simples destas duas técnicas sobre o texto baseado na chave (SCHNEIER, 1996). A confusão consiste em substituir blocos do texto original por outros dados, enquanto que a difusão nada mais é que uma permutação dos bits do texto original.

O algoritmo DES possui 16 iterações ou *rounds*, ou seja, o bloco de entrada passa pela mesma função criptográfica 16 vezes. Onde, em cada *round*, a única diferença é a chave usada pela função criptográfica. Apesar de receber uma única chave de entrada, internamente, o DES gera 16 sub-chaves, uma para cada *round* da criptografia. A Figura 3.4 apresenta um diagrama de blocos do, assim chamado, algoritmo de expansão de chaves. A chave original possui 56-bit e é dividida em dois blocos de 28-bit. Cada bloco é rotacionado para a esquerda um ou dois bits, dependendo do *round*. Após a rotação é aplicada uma operação de Compressão/Permutação. Essa operação faz a permutação dos bits além de selecionar um subconjunto destes bits para ser usado como chave em um *round*. Assim, dos 56 bits apenas 48 são usados na sub-chave. Tal operação, como todas as outras permutações ou substituições do DES são baseadas em tabelas pré-definidas. Os valores de permutação que definem a operação de Compressão/Permutação são apresentados na Tabela 3.1, obtida em (SCHNEIER, 1996). A tabela deve ser lida da esquerda para a direita e de cima para baixo. Assim, a primeira posição da tabela possui o valor

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Tabela 3.1: Compressão/Permutação.

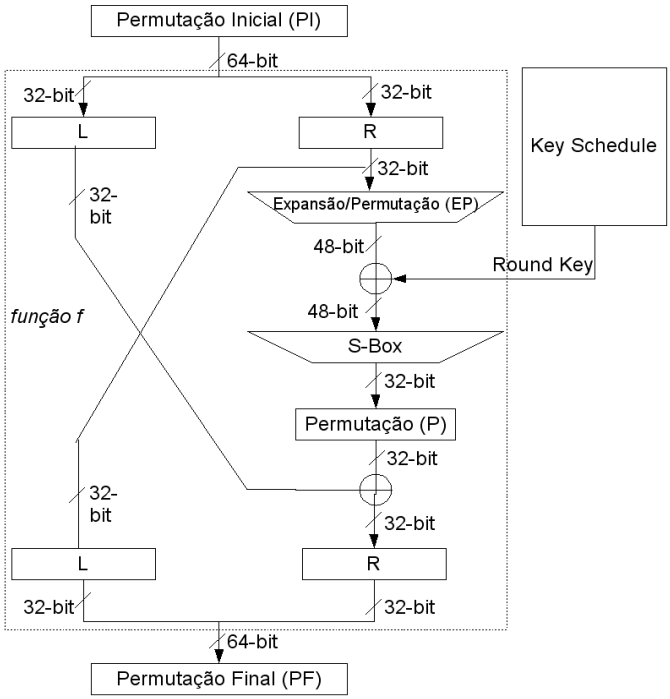


Figura 3.5: Diagrama de blocos do DES.

14. Isso significa que o 14° bit do vetor original assumirá a posição do primeiro bit no novo vetor. O segundo valor é 17, assim o 17° bit assumirá a segunda posição no novo vetor. A partir daqui, todas as tabelas de permutação apresentadas para o DES devem ser interpretadas desta mesma forma.

A Figura 3.5 é um diagrama de blocos do algoritmo DES. A primeira operação sobre o bloco de entrada é uma permutação, denominada Permutação Inicial (PI). A tabela de permutação para tal operação pode ser obtida em (SCHNEIER, 1996). Após a operação PI os dados passam para a próxima etapa de criptografia, a função f. Esta é a principal operação do DES e é subdividida em operações mais simples.

Na função f, o bloco de 64-bit é dividido em dois sub-blocos de 32-bit, identificados como R e L na Figura 3.5. Então, o bloco R sofre uma operação denominada Expansão/Permutação. Tal operação toma 32 bits de entrada e faz uma expansão para 48 bits, além da permutação. Desta forma, alguns dos bits de entrada são repetidos na saída. A operação de Expansão/Permutação é representada na Tabela 3.2.

O resultado da Expansão/Permutação passa por uma operação ou-exclusivo com a sub-chave escalonada pelo módulo *key schedule*. O resultado desta operação é movido para a próxima operação, denominada Substituição ou *SBox*. As *SBoxes* são formadas por oito diferentes tabelas de substituição. Cada *SBox* toma como entrada 6-bit e retorna como saída 4-bit. Assim, os 48-bit que chegam as *SBoxes* são divididos em 8 independentes grupos de 6-bit cada. Cada grupo é tratado individualmente por uma *SBox*. A título de

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

Tabela 3.2: Expansão/Permutação

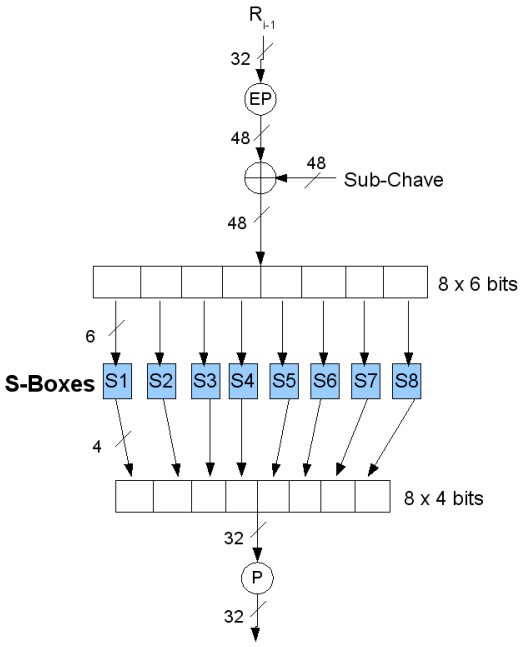


Figura 3.6: Detalhe das *SBoxes*.

exemplo, a *SBox* denominada *S1* na Figura 3.6 é mostrada na Tabela 3.3. As tabelas de substituição das *SBoxes* restantes podem ser encontradas em (SCHNEIER, 1996). Cada tabela de substituição tem 4 linhas e 16 colunas. O valor de entrada da *SBox* seleciona o valor de saída de uma maneira bastante específica. Considerando a entrada com 6-bit e nomeando-os como $b_1, b_2, b_3, b_4, b_5, b_6$, os bits b_1 e b_6 são combinados ($b_1 b_6$) e usados para indexar uma linha da tabela. Os 4-bit restantes são combinados ($b_2 b_3 b_4 b_5$) e usados para indexar uma das 16 colunas da tabela. As *SBoxes* são de suma importância para o DES. As outras operações do algoritmo são lineares e fáceis de analisar. As *SBoxes* são não lineares e dão segurança ao DES.

Os 32-bit resultantes da operação de substituição das *SBoxes* passam por uma nova permutação, denominada P. Novamente, a tabela da Permutação P pode ser vista em (SCHNEIER, 1996). Após tal permutação, o resultado passa por uma operação ou-exclusivo com a metade esquerda (L). Assim, chega ao fim um *round* da criptografia.

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Tabela 3.3: *SBox* 1.

As etapas da função f descritas aqui, são repetidas 16 vezes, onde para cada *round*, uma nova sub-chave é aplicada. Ao final dos 16 *rounds*, os dois blocos R e L são concatenados (RL) e passam por uma última permutação, denominada Permutação Final (PF). Esta operação é exatamente o inverso da Permutação Inicial (PI).

Como dito anteriormente, no caso do DES, o mesmo algoritmo usado para criptografar pode ser usado para descriptografar. A única diferença é que as chaves aplicadas para criptografar ($K_1, K_2, K_3, K_4, ..., K_{16}$) devem ser aplicadas na ordem inversa ($K_{16}, K_{15}, K_{14}, ...K_1$). Existe uma pequena diferença no algoritmo de geração das sub-chaves, para descriptografar os deslocamentos são para a direita e o número de deslocamentos para cada sub-chave pode ser visto em (SCHNEIER, 1996).

3.3.2 Advanced Encryption Standard (AES)

Com o desenvolvimento da computação, os computadores atuais tornaram-se milhares de vezes mais poderosos que seus predecessores. O algoritmo DES sobreviveu por cerca de 30 anos a essa constante evolução. Porém, no final dos anos 90, com *hardwares* dedicados contendo centenas de processadores, era possível quebrar por força bruta, um bloco cifrado pelo DES em questão de horas. Sistemas distribuídos também foram usados para demonstrar que o uso de chaves de até 64-bit estava obsoleto. Assim, a complexidade de 2^{56} oferecida pelo DES já não o tornava suficientemente seguro e um novo algoritmo precisava ser adotado.

Em 2 de Janeiro de 1997, o NIST (*National Institute of Standards and Technology*) dos Estados Unidos anunciou que escolheria um sucessor para o DES. Foi aberta uma chamada para submissão de algoritmos candidatos e o interesse da comunidade científica foi muito intenso. No total, houve 15 submissões. Os pesquisadores tiveram a chance de submeter seus algoritmos criptográficos assim como analisar as fraquezas de seus concorrentes. Os algoritmos foram analisados não apenas do ponto de vista de segurança, mas também pela sua simplicidade, facilidade de implementação tanto em *software* como *hardware* e eficiência. Em agosto de 1999, o NIST anunciou os 5 algoritmos finalistas: MARS, RC6, Rijndael, Serpent e Twofish. Todos os algoritmos finalistas continuaram sendo extensamente analisados pela comunidade científica. Até que em outubro de 2000, o NIST anunciou o algoritmo Rijndael como sendo o escolhido para substituir o DES. O anúncio oficial do novo algoritmo foi feito atráves do documento (FIPS, 2001). Desde então, o AES-Rinjdael tornou-se extensamente usado em todo mundo.

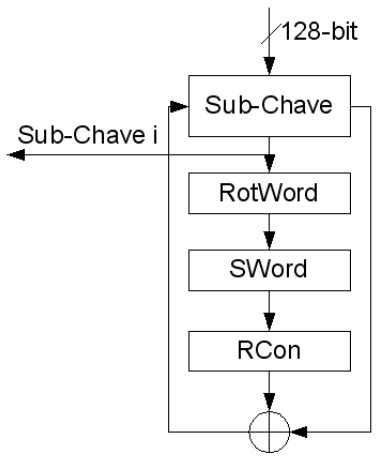


Figura 3.7: Digrama de Blocos simplificado do algoritmo de expansão de chaves do AES.

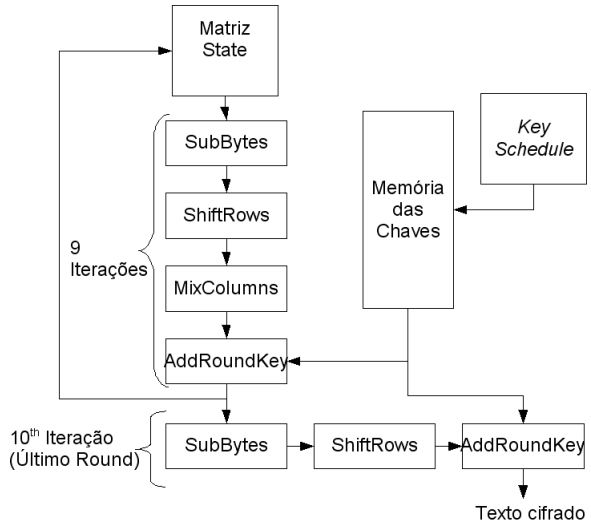


Figura 3.8: Diagrama de blocos simplificado do AES.

O AES aceita blocos de entrada de 128-bit. Sua chave tem tamanhos variáveis de 128, 192 e 256-bit. Diferentemente de seu antecessor que realizava criptografia manipulando bits, o AES opera em nível de bytes. Ou seja, a informação mínima operada pelo AES é um byte. Todos os bytes no algoritmo AES são interpretados como elementos de um campo finito. Estes elementos podem ser adicionados e multiplicados, mas estas operações são diferentes das comumente usadas. Maiores detalhes sobre as operações em campos finitos podem ser vistas em (FIPS, 2001). Por motivos de simplicidade será abordada apenas a versão de 128-bit do AES.

O primeiro passo do processo criptográfico é a expansão da chave ou *Key Schedule*. A chave original de 128, 192 ou 256-bit é expandida para 10 novas sub-chaves de tamanho correspondente. A Figura 3.7 apresenta um diagrama de blocos do algoritmo de expansão de chaves do AES. A primeira chave gerada é a própria chave original. Cada nova chave gerada, a partir da chave original, passa por quatro operações distintas. A primeira operação é denominada *RotWord* que toma como entrada quatro *bytes* ($b_1b_2b_3b_4$) e realiza uma rotação para a esquerda de uma posição ($b_2b_3b_4b_1$). A próxima operação, é uma operação de substituição usando a Tabela 3.4, isso será melhor explicado adiante. Em seguida, a operação *Rcon* é aplicada. Esta é uma operação de exponenciação em campo finito. Finalmente, é aplicada uma operação ou-exclusivo dos *bytes* gerados por estas três operações com os *bytes* da chave gerada anteriormente.

O bloco de entrada é armazenado em uma matriz 4x4, na versão de 128-bit, denominada *Matriz State*. Cada posição da matriz corresponde a um *byte*. O AES possui 4 diferentes operações criptográficas:

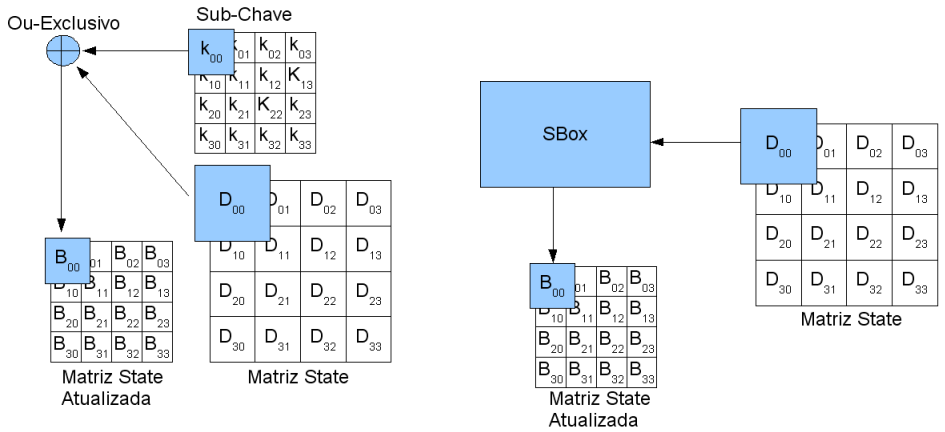
- *AddRoundKey*;
- *SubBytes*;
- *ShiftRows*;
- *MixColumns*.

A Figura 3.8 apresenta um diagrama de blocos do AES. Ele apresenta 10 iterações ou *rounds*. Cada iteração do AES corresponde à aplicação das quatro operações sobre

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

Tabela 3.4: *SBox* do algoritmo AES.

a *Matriz State*, exceto para a última iteração onde a operação *MixColunms* é omitida. A primeira operação, denominada *AddRoundKey*, é um ou-exclusivo com a sub-chave correspondente à iteração. Esta operação é executada *byte*-a-*byte* com as posições correspondentes, conforme a Figura 3.9(a).



(a) *AddRoundKey*. (b) *SubBytes*.

Figura 3.9: Detalhes das operações criptográficas *AddRoundkey* e *SubBytes*.

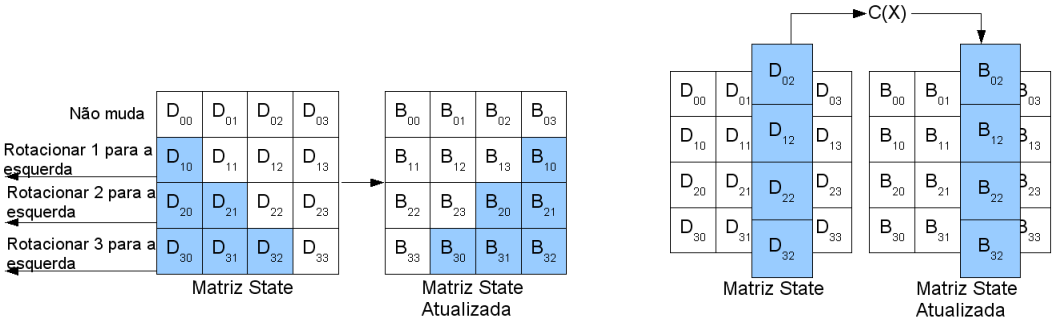
A segunda operação é chamada de *SubBytes*. De fato, esta operação faz a substituição dos *bytes* da *Matrix State* com base em uma tabela denominada *Sbox*. Diferentemente do algoritmo DES (ver Subseção 3.3.1), o AES possui apenas uma *Sbox* com 256 entradas. A Figura 3.9(b) ilustra tal operação e a Tabela 3.4 mostra os valores de substituição. A função da *SBox* no AES, assim como no DES, é prover uma operação não linear garantindo segurança ao algoritmo. A *Sbox* é consultada de forma que cada *byte* da *Matriz State* é tomado individualmente. Chamando os *bits* de um *byte* como $b_1b_2b_3b_4b_5b_6b_7b_8$, então, os primeiros 4 *bits* ($b_1b_2b_3b_4$) são usados para indexar uma linha da matriz, enquanto que os

outros 4 *bits* $b_5b_6b_7b_8$ são usados para indexar uma coluna.

A terceira operação aplicada sobre a *Matriz State* é denominada *ShiftRows*. Esta operação simplesmente faz rotação com os *bytes* da matriz de forma que a primeira linha permanece sem mudanças, a segunda linha é rotacionada em uma posição, a terceira linha é rotacionada em duas posições e, por último, a quarta linha é rotacionada em 3 posições. A Figura 3.10(a) ilustra esta operação.

Finalmente, a última operação aplicada sobre a *Matriz State* durante um *round* é denominada *MixColuns*. A Figura 3.10(b) ilustra esta operação. Tal operação é efetuada coluna-por-coluna da *Matriz State*, tratando cada coluna como um polinômio de 4 termos sobre um campo finito específico denominado *Galois Field* (2^8), multiplicado módulo $x^4 + 1$ por um polinômio fixo $a(x)$ (FIPS, 2001), dado por:

$$a(x) = 3x^3 + x^2 + x + 2.$$



(a) *ShiftRows*. (b) *MixColumns*.

Figura 3.10: Detalhes das operações criptográficas *ShiftRows* e *MixColumns*.

Ao contrário do DES, o AES não é um cifrador de Feistel (ver Subseção 3.3.1). Desta forma, o algoritmo usado para criptografar não consegue executar a operação inversa. Porém, cada uma das quatro operações criptográficas citadas são inversíveis. Assim, a descriptografia é realizada por um segundo algoritmo que aplica as operações inversas sobre os dados criptografados. Maiores detalhes sobre as operações inversas podem ser encontradas em (FIPS, 2001).

3.3.3 Rivest-Shamir-Adleman(RSA)

O RSA pertence a um grupo de algoritmos criptográficos denominado assimétricos ou de chave pública. O conceito que cerca tais algoritmos foi proposto por Whitfield Diffie e Martin Hellman, em 1976 (DIFFIE; HELLMAN, 1976). A contribuição para a criptografia foi a noção de que chaves criptográficas poderiam existir em pares (uma chave para criptografar e outra para descriptografar), e ainda, que seria impossível obter uma chave a partir da outra (SCHNEIER, 1996). Porém, até aquele momento não havia nenhum algoritmo capaz de atender a tais requesitos. Em 1977, em resposta ao desafio lançado por Diffie e Hellman, Ron Rivest, Adi Shamir e Leonard Adleman desenvolveram o algoritmo RSA (RIVEST; SHAMIR; ADELMAN, 1977). Até hoje, o RSA é considerado seguro para aplicações que envolvem sigilo e assinatura digital. Porém, é muitas vezes mais lento que algoritmos simétricos como o AES ou DES. Um dos fatores que ajudou o RSA a se tornar popular, além de ser considerado seguro, é a sua facilidade de entendimento e implementação.

A segurança que o RSA proporciona é baseada na dificuldade de se fatorar números primos muito grandes. Por isso a necessidade de se usar chaves de 1024-bit em contraste a algoritmos simétricos que são considerados seguros mesmo usando chaves de apenas 128-bit. O par de chaves gerado é, na verdade, um par de números primos muito grandes (100, 200 ou mais dígitos). O princípio de geração das chaves inicia pela escolha de dois números primos muito grandes (p e q). A criação da primeira chave, denominada chave pública, é feita escolhendo-se aleatoriamente um número e que seja relativamente primo à $(p-1)(q-1)$. A segunda chave, chamada de chave privada, é calculada fazendo-se:

$$d = e^{-1} mod((p-1)(q-1))$$

A chave pública é de livre distribuição. Quando alguém deseja enviar uma mensagem secreta, ele deve criptografar a mensagem com a chave pública do destinatário. A mensagem apenas poderá ser descriptografada com a chave privada que deve ser mantida em segredo. Para obter assinatura digital, o detentor de uma chave privada criptografa o documento desejado com sua chave privada. O documento poderá ser aberto apenas com a chave pública do mesmo. Assim, mais ninguém poderia ter gerado o documento assinado, garantindo não repúdio.

O processo de criptografia é feito, primeiramente, calculando-se: $n = pq$. Agora, supondo que a mensagem a ser criptograda seja m e que o texto criptografado seja c, então calcula-se:

$$c = m^e mod n$$

Para descriptografar o texto c e obter novamente m deve-se fazer:

$$m = c^d mod n$$

Os números p e q podem ser descartados após o cálculo de n, porém, nunca revelados.

Em *hardware* o RSA é cerca de 1000 vezes mais lento que o DES (SCHNEIER, 1996). Por esse motivo o RSA é usado apenas para assinatura digital ou pequenos blocos de dados, como trocas de chaves de sessão.

4 CRIPTOANÁLISE EM DISPOSITIVOS CRIPTOGRÁFICOS

Sistemas criptográficos são alvo de uma vasta variedade de ataques. Tais ataques visam extrair informações confidenciais ou alterar as informações contidas nestes sistemas. A ciência que explora as vulnerabilidades dos algoritmos criptográficos é chamada de criptoanálise. A criptoanálise tenta encontrar métodos para recuperar dados criptografados sem ter acesso a chave criptográfica. Ela também ajuda a determinar o quão resistente um algoritmo criptográfico pode ser. Em *software*, a criptoanálise fica restrita à exploração matemática dos algoritmos criptográficos ou à análise das fraquezas das arquiteturas de código usadas. Porém, nas implementações de algoritmos criptográficos em *hardware* uma série de outras vulnerabilidades são expostas e novas técnicas de criptoanálise são utilizadas. Ataques contra *hardware* criptográfico exploram características intrínsecas ao *hardware*, como análise da potência consumida, emissões eletromagnéticas, suscetibilidade à injeção de falhas transientes ou, até mesmo, engenharia reversa. Portanto, medidas contra exploração das fraquezas em *hardware* criptográfico precisam ser tomadas.

Este capítulo aborda inicialmente, na Seção 4.1, alguns tipos de ataque a *hardware* criptográfico. A Seção 4.2 é dedicada a ataques por injeção de falhas ou simplesmente *fault attacks*. Em seguida, na Seção 4.3, é mostrado como *fault attacks* podem ser usados para quebrar sistemas criptográficos baseados nos algoritmos DES, AES e RSA, respectivamente. Após, são abordadas possíveis contramedidas em nível de *hardware* para delinear tais ataques. Finalmente, a Seção 4.5 aborda os principais trabalhos já realizados para conter ataques contra os algoritmos citados.

4.1 Ameaças contra *hardwares* criptográficos

Criptoanálise em *hardware* pode ser uma técnica eficiente e é a ferramenta mais utilizada contra criptografia em *hardware*. Em (RENAUDIN; AL., 2004), os autores fazem uma classificação das diferentes áreas da criptoanálise. Tal classificação é apresentada na Figura 4.1. A criptoanálise divide-se em duas áreas: *software* e *hardware*. Em *software* os ataques são divididos em duas subcategorias: o primeiro grupo inclui técnicas matemáticas usadas para quebrar algoritmos criptográficos. Este tipo de ataque exige grande conhecimento matemático. O segundo grupo inclui os métodos que exploram as vulnerabilidades dos softwares embarcados nos dispositivos criptográficos. Tais vulnerabilidades podem ser exploradas, por exemplo, usando técnicas como *buffer overflow* ou *trojam horses* (RUSSELL; GANGEMI, 1991). Obter sucesso neste tipo de ataque exige bons conhecimentos em computação, além da exploração de possíveis falhas de implementação nos *softwares*.

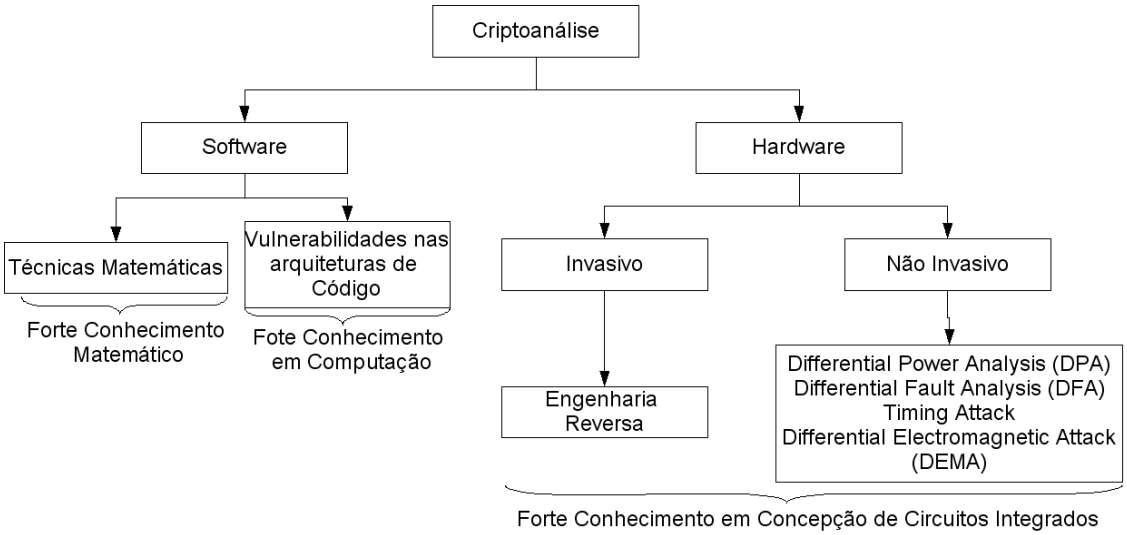


Figura 4.1: Diferentes formas de criptoanálise. Adaptado de (RENAUDIN; AL., 2004).

A criptoanálise em *hardware* também é dividida em dois grupos: ataques invasivos e ataques não invasivos. Ataques invasivos utilizam técnicas para acessar as conexões do CI, com a finalidade de observar e manipular os sinais. Estes normalmente, levam à destruição do dispositivo observado. Estes ataques exigem laboratórios e equipamentos especiais o que, muitas vezes, significa altos custos. Ataques não invasivos são também conhecidos como *Side-Channel Attacks* e exploram as vulnerabilidades dos algoritmos criptográficos implementados em *hardware*. Tais vulnerabilidades podem, por exemplo, abrir caminho para obtenção da chave criptográfica armazenada no dispositivo. Tanto ataques invasivos quanto não invasivos exigem forte conhecimento em concepção de circuitos integrados. Diversas técnicas foram desenvolvidas para a realização de ataques não invasivos, como por exemplo:

- *Power Analysis*: Baseia-se na análise do consumo de corrente de um dispositivo criptográfico durante o processo de criptografia. A contínua observação permite estabelecer uma correlação entre os dados processados e o consumo de corrente. *Simple Power Analysis* (SPA) é a forma mais simples de *Power Analysis* e é executada pela representação e observação visual do consumo de corrente. Por outro lado, existe a *Differential Power Analysis* (DPA) que baseia-se em estudos estatísticos. Ambas as técnicas são grandes ameaças à segurança para a criptografia em *hardware* (KOCHER; JAFFE; JUN, 1999).

- *Timing Attack*: Consiste em medir e analisar o tempo requerido para executar algoritmos criptográficos. O tempo de processamento em alguns algoritmos depende dos dados de entrada e isso fornece valiosas informações a respeito dos dados processados (KOCHER, 1996).

- *Eletromagnetic Attack*: Semelhante a *Power Analysis*, porém baseia-se nas emissões eletromagnéticas do CI sob observação. Este tipo de ataque, chamado *Single Eletromagnetic Attack* (SEMA) e *Differential Eletromagnetic Attack* (DEMA), tem demonstrado resultados efetivos (GANDOLFI; MOURTEL; OLIVIER, 2001), e isto induz ao estudo de contramedidas.

- *Fault Attack*: São ataques baseados na injeção de falhas no *hardware* criptográfico. Devido a sua importância para este trabalho, este tipo de ataque será melhor abordado na Seção 4.2.

Diferentes ataques em *hardware* exigem diferentes contramedidas. Raramente uma contramedida é capaz de ser eficaz contra dois ou mais métodos, pois a natureza dos ataques são bastante diferentes, apesar de todos se basearem nas vulnerabilidades das implementações físicas. Soluções para impedir ataques contra DPA podem agir, por exemplo, tornando o consumo de corrente constante ou fazendo com que não exista correlação entre o consumo e os dados processados. Porém, tais soluções não impedem *timing attacks* ou tornam o *hardware* tolerante a *fault attacks*. Assim, diferentes soluções devem coexistir no mesmo sistema afim de garantir segurança contra diferentes tipos de ataques.

4.2 *Differential Fault Analysis (DFA)*

Differential Fault Analysis (DFA) ou simplesmente *fault attack* explora vulnerabilidades nos algoritmos criptográficos em situações de mal funcionamento. Se erros computacionais afetarem certos sistemas criptográficos, estes permitem acesso a informações que podem levar à reconstrução da chave criptográfica armazenada no próprio dispositivo. O atacante pode, repetidamente, computar o resultado correto e com falhas, compará-los, e tentar deduzir a chave. *Smart Cards* e outros tipos de *tokens* criptográficos, conforme visto no capítulo 2, armazenam chaves criptográficas, o que os torna os principais alvos da execução de DFA.

Segundo (HESS et al., 2000), existem dois principais modelos de falhas para DFA. O primeiro é chamado de *modelo de falhas transiente*. Neste modelo é assumido que o atacante pode alterar o valor de um *bit* (ou um pequeno número deles) armazenado em um registrador. O momento no qual o *bit* é alterado e a posição da memória são aleatórios, porém, ocorrem durante a computação. Existem diversas formas para injeção de falhas transientes, conforme será visto a seguir. Falhas transientes incidem sobre memórias voláteis ou lógica combinacional, afetando apenas a execução. O *hardware* passa a comportar-se normalmente após sua reinicialização. O segundo modelo de falhas é chamado de *modelo de falhas persistente*. Neste modelo, o atacante causa falhas permanentes no dispositivo. Ele pode usar luz ultravioleta ou raio-X para alterar memórias não voláteis como EEPROM, ou ainda destruir células de memória ROM ou introduzir falhas do tipo *stuck-at*.

Em Setembro de 1996, Boneh, Demillo e Lipton de Bellcore (BONEH; DEMILLO; LIPTON, 1997), descreveram um ataque contra algoritmos criptográficos assimétricos no modelo de falhas transientes. Neste ataque, implementações do RSA que usam o *teorema chinês do resto* (CRT) para melhorar o desempenho, podem ser completamente quebradas se um erro ocorrer em uma das operações de exponenciação. Mesmo em implementações que não usam CRT a chave criptográfica pode ser deduzida se erros afetarem uma das últimas operações de exponenciação. No ano seguinte, Biham e Shamir publicaram um trabalho mostrando que um ataque baseado nos mesmos princípios poderia ser usado para quebrar quase todos os algoritmos simétricos (A. SHAMIR, 1997). Em adição, eles demonstraram um segundo método de ataque usando o modelo de falhas persistente, supondo que o atacante possui meios de cortar fios ou destruir portas.

Existem diversos caminhos para se efetuar injeção de falhas transientes, as principais técnicas segundo (HESS et al., 2000) são:

- *Glitch Attacks*: Historicamente, foi a primeira técnica usada para causar falhas em *Smart Cards*. Consiste em submeter os pinos de VCC (alimentação), GND (terra) ou *clock* a condições anormais. Isso significa sobrecarga de alimentação ou um sinal de *clock* irregular. Para obter sucesso, o atacante precisa acertar o momento correto, a intensidade e a duração do *glitch*. *Glitchs* muito irregulares podem causar danos permanentes no cartão e sua inutilização. Por outro lado, é uma forma de ataque bastante simples e que exige equipamentos baratos. Atualmente, os *Smart Cards* podem resistir a este tipo de ataque graças a adição de filtros, conversores DC ou sensores de alimentação. O cartão pode detectar o ataque reagindo, por exemplo, com uma reinicialização (HESS et al., 2000).

- *Light Attacks*: Surgiu em 2000, introduzido por (SKOROBOGATOV; ANDERSON, 2002). Segundo (HESS et al., 2000), é o método mais comum usado atualmente para *fault attacks*. O ataque baseia-se na emissão de luz para perturbar o silício do *chip*. É um ataque semi-invasivo, pois requer a retirada da camada de plástico que envolve o cartão, para que a luz seja aplicada diretamente sobre o silício. O atacante precisa lidar com alguns parâmetros para aumentar as chances de sucesso do ataque: a intensidade da luz, a localização e extensão da área atingida. Uma vantagem deste tipo de ataque sobre *glitch attacks* é a sua precisão, pois o atacante pode escolher a área do *chip* que será atacada. Segundo (HESS et al., 2000), algumas contramedidas podem evitar que o *hadware* seja sucetível a ataques menos elaborados, mas com um laboratório mais sofisticado, o atacante ainda pode obter sucesso com esta técnica.

- *Magnetic Attacks*: Outra forma de causar injeção de falhas transientes é a submissão do *hardware* alvo à emissão de pulsos magnéticos. O campo magnético cria correntes elétricas na superfície do componente, o qual pode gerar uma falha. Na prática, o ataque é realizado com materiais baratos, porém, para melhorar os resultados é interessante remover a camada de plástico que envolve o silício. Os parâmetros que o atacante deve avaliar são: a intensidade do campo magnético, sua duração e localização (HESS et al., 2000).

Inicialmente, DFA foi criticado por ser puramente teórico. Contudo, ataques bem sucedidos foram conduzidos contra sistemas de *pay-TV* (ANDERSON; KUHN, 1996), permitindo que os usuários assistissem a todos os canais sem pagar por eles. Em (ANDERSON; KUHN, 1997), os autores apresentam técnicas baratas, porém bastante efetivas, para execução de *fault attacks*. Em (SKOROBOGATOV; ANDERSON, 2002), é apresentado um ataque prático, novamente, usando equipamentos baratos e de fácil aquisição. Neste tipo de ataque, a injeção de falhas é feita com ajuda de um *laser*, caracterizando um *Light Attack*. Ainda, injeção de falhas utilizando *Glitchs* é um método interessante, porque pode ser facilmente implementando para atuar de forma que o atacante não precise dispor de total acesso ao dispositivo e sem que o usuário perceba. Imagine que um leitor de *Smart Cards* seja modificado para, primeiramente, efetuar a transação desejada e, em seguida, causar perturbações na alimentação do *Smart Card* afim de extrair blocos criptografados com falhas para análise posterior. Neste cenário, a chave criptográfica armazenada no cartão pode ser descoberta sem que usuário perceba a tempo de cancelar o cartão.

Devido às diversas vulnerabilidades que as implementações em *hardware* de algoritmos criptográficos apresentam, principalmente com relação à suscetibilidade à injeção de

falhas, contrametidas precisam ser tomadas. As tecnologias de semicondutores são aprimoradas, porém, tornando-se mais sensíveis à injeção de falhas transientes. Ao mesmo tempo, com o barateamento das tecnologias, os atacantes podem ter acesso a novos equipamentos e, conseqüentemente, executar ataques mais elaborados.

4.3 Quebrando Algoritmos Criptográficos por DFA

Esta seção é destinada a demonstrar formalmente como certos algoritmos criptográficos podem ser quebrados por injeção de falhas. Para compreensão das técnicas de criptoanálise aqui descritas, é necessário um bom entendimento dos algoritmos criptográficos sob análise, o que significa a leitura da Seção 3.3. Os ataques descritos objetivam a obtenção da chave criptográfica armazenada no dispositivo. Tais ataques são independentes do método de injeção de falhas utilizado, porém exigem uma precisão razoável na injeção de falhas.

4.3.1 Atacando o DES

O ataque demonstrado aqui visa obter a chave criptográfica, do algoritmo DES, a partir da análise de um texto cifrado corretamente e diversos textos cifrados com a injeção de falhas transientes em *hardware* durante o processo criptográfico. Para efetuar este ataque, o atacante não precisa ter nenhum conhecimento prévio do texto original ou da chave. A injeção de falhas tem como alvo a 15° iteração da *função f*, ou seja, as falhas transientes devem atingir algum *bit* na 15° iteração ou diretamente o registrador R_{15}. A Figura 4.2 ilustra tal situação. Em um ataque real, o atacante não tem tanta precisão para a injeção da falha e, muita vezes, a falha pode atingir um *bit* randomicamente. Porém, o atacante pode identificar as falhas que se manifestaram antes da 15° iteração. Devido às operações criptográficas, um único *bit* alterado afeta muitos outros *bits* a cada nova iteração. Assim, um bloco criptografado com um número de *bits* afetados muito maior que o esperado deve ser descartado. Por simplicidade, a explicação a seguir considera falhas apenas na 15° iteração.

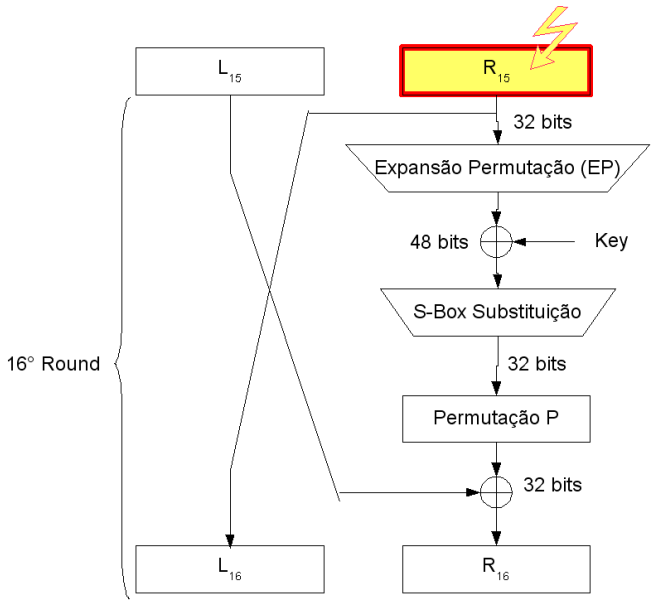


Figura 4.2: Última iteração da *função f*. Detalhe da injeção de falhas no registrador R_{15}.

Na Figura 4.2, L_{15} e R_{15} são provenientes da 15° iteração. Após a execução da última operação são obtidos L_{16} e R_{16}. R_{16} pode ser expresso como:

$$R_{16} = P(S(EP(R_{15}) \oplus K_{16})) \oplus L_{15}$$

$$R_{15} = L_{16}$$

$$R_{16} = P(S(EP(L_{16}) \oplus K_{16})) \oplus L_{15}. \quad (4.1)$$

P é a Permutação P, S são as $S-Boxes$, EP é a operação de Expansão/Permutação e $\oplus$ é a operação ou-exclusivo. Observe que nesta equação existem duas variáveis desconhecidas: K_{16} e L_{15}. Se uma falha ocorrer durante a 15° iteração no lado direito, ou seja, se substituirmos R_{15} por um $\hat{R}_{15}$ faltoso, então temos:

$$\hat{R}_{16} = P(S(EP(\hat{R}_{15}) \oplus K_{16})) \oplus L_{15}$$

$$\hat{R}_{15} = \hat{L}_{16}$$

$$\hat{R}_{16} = P(S(EP(\hat{L}_{16}) \oplus K_{16})) \oplus L_{15}. \quad (4.2)$$

Se realizarmos uma operação $\oplus$ entre as Equações 4.1 e 4.2, obtemos:

$$R_{16} \oplus \hat{R}_{16} = \{P(S(EP(L_{16}) \oplus K_{16})) \oplus L_{15}\} \oplus \{P(S(EP(\hat{L}_{16}) \oplus K_{16})) \oplus L_{15}\}$$

$$= \{P(S(EP(L_{16}) \oplus K_{16}))\} \oplus \{P(S(EP(\hat{L}_{16}) \oplus K_{16}))\} \quad (4.3)$$

Observe que na Equação 4.3 L_{15} pôde ser eliminado devido às propriedades da operação ou-exclusivo. Assim, a única variável desconhecida é K_{16} que é justamente o valor procurado, ou seja, a chave secreta armazenada no dispositivo. Os outros valores (L_{16}, R_{16}, $\hat{R}_{16}$ e $\hat{L}_{16}$) são saídas do DES. Obtendo-se K_{16} é possível reverter o processo de geração das sub-chaves e obter 48 *bits* da chave original. Os outros 8 *bits* podem ser encontrados por força bruta (256 tentativas). Um experimento foi realizado afim de simular a injeção de falhas e a obtenção da chave criptográfica a partir de textos cifrados com falhas. Tal experimento é detalhado na Seção 5.2.

4.3.2 Atacando o AES

Diversas técnicas de DFA foram apresentadas contra o AES, conforme visto em (PIRET; QUISQUATER, 2003), (DUSART; LETOURNEUX; VIVOLO, 2003) e (GIRAUD, 2003). Tais ataques assumem diferentes modelos de falhas. Alguns ataques são muito realistas e são uma real ameaça para dispositivos seguros. Por simplicidade, um ataque simples descrito em (GIRAUD, 2003) é mostrado aqui. Este ataque assume que somente uma falha ocorre em somente um *bit* da *Matriz State*. Esta falha deve ocorrer no início da última iteração. A Figura 4.3 ilustra tal situação. O objetivo deste ataque é a obtenção da chave K^{10}, a qual, é a chave da última iteração do AES.

Uma iteração do AES é descrita como uma equação que mostra a ordem das operações criptográficas. Assim, a última iteração do algoritmo AES pode ser escrita como:

$$C = ShiftRows(SubBytes(M^9)) \oplus K^{10} \quad (4.4)$$

Onde, C é o texto cifrado após a última iteração. M^9 é o resultado armazenado na *Matriz State* após a 9° iteração. K^{10} é a sub-chave criptográfica na última iteração. O resultado da tabela de substituição (*S-box*) sob o *byte* M^i_j foi denotado por *SubByte*(M^i_j).

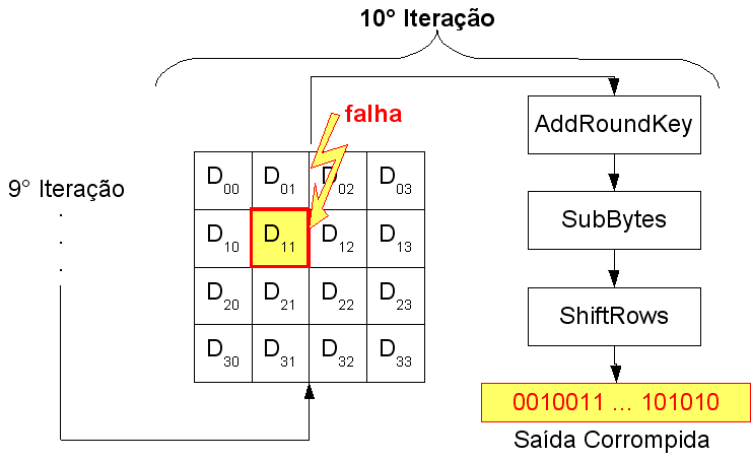


Figura 4.3: Última iteração (10°) do AES. Detalhe da injeção de falhas em um *byte* arbitrário na *Matriz State*.

ShiftRow(j) é a posição do j^{th} *byte* após a aplicação da operação *ShiftRow()*. Assim, a Equação 4.4 pode ser reescrita como:

$$ShiftRows(C) = SubBytes(M_j^9)) \oplus ShiftRows(K^{10}), \tag{4.5}$$

$$j = \{0, ..., 15\}$$

A operação $ShiftRows()$ foi aplicada sobre C e K^{10} para manter a equivalência. Agora, induzindo uma falha e_j sobre o j^{th} *byte* de M^9 antes da última iteração, temos:

$$ShiftRow(D) = SubByte(M_j^9 \oplus e_j)) \oplus ShiftRows(K^{10}), \tag{4.6}$$

onde, D é a saída com falhas. Realizando uma operação XOR entre as equações 4.5 e 4.6 obtêm-se:

$$ShiftRow(C) \oplus ShifRow(D) = SubByte(M_j^9 \oplus e_j)) \oplus SubByte(M_j^9) \tag{4.7}$$

A operação $ShifRows(K^{10})$, presente nas equações 4.5 e 4.6, pôde ser removida de 4.7 devido as propriedade da operação XOR. Para resolver a Equação 4.7 é necessário, primeiramente, determinar $ShiftRow(j)$ que é a posição onde $C \oplus D$ é diferente de zero. Assim, encontramos j. Depois é preciso determinar M_j^9, o que pode ser feito por um método incremental, onde um valor para e_j é suposto e é determinado o conjunto de possíveis valores para M_j^9, os quais verificam a Equação 4.7. Com outros textos cifrados com falhas o conjunto de valores possíveis para M_j^9 é decrementado até restar o valor procurado. Novas falhas devem ser injetadas nos outros *bytes* de M^9 afim de determinar todos os *bytes* da chave. Com posse do texto cifrado C e o valor de M^9, é fácil obter a sub-chave da última iteração (K^{10}) pela Equação 4.4. Finalmente, aplicando o processo inverso do *key schedule* é possível obter a chave original (K).

Segundo (GIRAUD, 2003), usando apenas 3 textos cifrados com falhas no mesmo *byte*, há 97% de chance de sucesso em obtê-lo. Então é possível obter os 128-bit da chave usando menos que 50 textos cifrados com falhas. Além disso, este ataque opera independentemente sob cada *byte*, assim, induzindo falhas simples sobre diferentes *bytes* simultaneamente, é possível obter a chave com um número reduzido de textos faltosos.

4.3.3 Atacando o RSA

Em (BONEH; DEMILLO; LIPTON, 1997) é apresentado um ataque baseado em propriedades algébricas contra o RSA, que foi conhecido posteriormente como Ataque de Bellcore. Conforme visto na Subseção 3.3.3, para criptografar uma mensagem (m) é feito: $m^e mod n$, onde n é o produto de dois números primos ($n = pq$). Porém, isso é computacionalmente custoso. Para reduzir o tempo de processamento, principalmente em sistemas com poucos recursos, como *Smart Cards*, é usada a seguinte técnica: computa-se $E_1 = m^e mod p$ e $E_2 = m^e mod q$. Então, usa-se o Teorema do Resto Chinês para computar $E = m^e mod n$. Para aplicar tal teorema precisa-se calcular dois inteiros a e b, tal que:

$$\begin{cases} a \equiv 1(mod p) \\ a \equiv 0(mod q) \end{cases} \text{ e } \begin{cases} b \equiv 0(mod p) \\ b \equiv 1(mod q) \end{cases}.$$

Tais inteiros sempre existem e podem ser facilmente encontrados, dado p e q. E é determinado por:

$$E = aE_1 + bE_2(mod n). \tag{4.8}$$

Assim, a criptografia de E é computada pela combinação linear de E_1 e E_2. Este esquema é muito mais eficiente (cerca de 4 vezes mais rápido) que calcular diretamente módulo n.

O problema desta implementação é que o uso de DFA permite fatorar n com pouco esforço computacional. Uma vez que os fatores de n (p e q) são descobertos, o algoritmo é quebrado. O ataque é baseado na obtenção de duas assinaturas de uma mesma mensagem: uma correta e outra faltosa.

Suponha que uma falha transiente ocorra durante a computação de E_1 ou E_2. Sem perda de generalidade, será considerado que uma falha atinja E_1. E_1 com falhas será denotado por $\hat{E}_1$. Então,

$$mdc(E - \hat{E}, N) = mdc(a(E_1 - \hat{E}_1), n) = q. \tag{4.9}$$

Assim, usando um bloco cifrado com falha e um bloco sem falhas, o módulo n pode ser facilmente fatorado. Isso mostra que implementações do RSA baseadas no Teorema do Resto Chinês são vulneráveis a DFA.

4.4 Contramedidas para DFA

Devido às vulnerabilidades apresentadas, contramedidas devem ser aplicadas para garantir a robustez dos dispositivos criptográficos contra *fault attacks*. Tornar o *hardware* tolerante a *fault attacks* significa que este deve possuir algum mecanismo mínimo de detecção que permita interromper o funcionamento do dispositivo sob a ocorrência de falhas transientes ou permanentes. Outra possibilidade é, sob a detecção de um possível ataque, emitir um resultado qualquer que não possa ser usado na descoberta da chave. Porém, em muitos casos, apenas detecção não é suficiente e mecanismos de detecção e correção devem ser empregados. Quando se deseja atingir um alto grau de disponibilidade e confiabilidade, além da segurança, técnicas amplamente usadas em sistemas tolerantes a falhas podem ser empregadas como mecanismos de segurança.

Adicionando técnicas de tolerância a falhas, o sucesso de um ataque por DFA não depende mais apenas da injeção de falhas e de técnicas de criptoanálise, mas sim, da injeção de falhas que não podem ser detectadas. Além disso, o ataque é bem sucedido apenas quando o atacante consegue atingir sua meta final, por exemplo, obter a chave criptográfica. Assim, é importante ressaltar que, em sistemas tolerantes a falhas, o sucesso da geração de falhas não necessariamente implica na obtenção de resultados faltosos úteis para a criptoanálise posterior. Isso leva o atacante à exploração das arquiteturas tolerantes a falhas utilizadas afim de determinar modelos de falhas que o sistema sob ataque não pode tolerar. Além disso, ofuscar as falhas leva o atacante a uma incerteza do sucesso na geração de falhas, assim como dos módulos atingidos. Desta forma, tolerância a falhas leva a um acréscimo significativo na segurança quando comparado a sistemas com apenas detecção de falhas.

Diferentes medidas podem ser aplicadas para DFA. Tais contramedidas variam muito e podem, por exemplo, realizar a mesma computação duas vezes, a fim de comparar as saídas (redundância temporal) e apenas emitir o resultado caso sejam iguais. Outras soluções podem também sugerir alterações no algoritmo criptográfico de forma que a injeção de falhas não permita a extração de informações relevantes, conforme proposto em (BLÖMER; OTTO; SEIFERT, 2003) para o algoritmo RSA. Porém, esta última técnica não garante o sucesso da computação para falhas transientes ocorridas naturalmente, e esta é uma preocupação que deve ser levada em conta. Uma solução interessante contra injeção de falhas é recalcular determinadas etapas do processamento que são vulneráveis a *fault attacks*, o que pode ser feito em paralelo, adicionando *hardware* redundante (redundância espacial). Porém, na prática, tal análise exige um estudo bastante apurado a respeito das conseqüências deste procedimento. Recalcular uma etapa de um algoritmo exige maior tempo de execução ou *hardware* adicional. Isso implica em maior consumo energético e custo de produção. Portanto, uma solução efetiva precisa, além de impedir o ataque, atender aos requisitos de consumo e custo. No capítulo 5 é abordada uma técnica de proteção para o DES baseada em duplicação parcial do *hardware* afim de recalcular etapas vulneráveis do algoritmo, em paralelo.

Uma técnica sugerida por (BONEH; DEMILLO; LIPTON, 1997) é o uso de circuitos *self-checking*. Em circuitos com propriedades que garantem *self-checking*, falhas transientes podem ser automaticamente detectadas por um sub-circuito chamado *checker* (ABRAMOVICI; BREUER; FRIEDMAN, 1994). Ao detectar uma falha o *checker* pode, por exemplo, forçar a reinicialização do sistema, impedindo assim a injeção de falhas no *hardware*. Um *checker* é geralmente definido como um circuito de uma única saída, o qual assume o valor 0 para qualquer entrada correta e 1 quando ocorre uma detecção de erro (ABRAMOVICI; BREUER; FRIEDMAN, 1994). Um circuito *self-checking* baseia-se na idéia, de que muitas vezes, é possível determinar se a saída de um circuito está correta sem conhecê-la previamente. Para tanto, é adicionado ao circuito original *bits* de redundância. Esses *bits* configuram códigos que são caracterizados pela sua habilidade de detectar/corrigir erros. Os *bits* adicionais são chamados de *check bits* (ABRAMOVICI; BREUER; FRIEDMAN, 1994) e constituem um código em específico, os bits restantes constituem dados. As características da codificação utilizada permitirão apenas detecção ou detecção seguida de correção. Além disso, um determinado código está restrito a um número máximo de erros que ele pode detectar/corrigir. Com estas características o *hardware* passa a ser projetado para suportar certas classes de erros, que passam a ser detectados e/ou corrigidos automaticamente. Além disso, o tipo de código utilizado pode variar dependendo da aplicação. Para transmissão de dados em barramentos, o uso de *bits*

	Injeção	**Erros**
Referência	5600	955(17%)
Protegido	5600	377(7%)

Tabela 4.1: Injeção de falhas para o DES.

de paridade (ABRAMOVICI; BREUER; FRIEDMAN, 1994) pode ser suficiente, porém, para um somador os *check bits* do resultado dependem dos *check bits* dos operandos, exigindo assim, codificações mais adequadas. No Capítulo 6, técnicas de *self-checking* foram utilizadas para dar proteção ao algoritmo AES, além de permitir recuperação em situação de falhas.

Os mecanismos de proteção elaborados neste trabalho, conforme será abordado nos capítulos 5 e 6, são capazes de prevenir ataques detectando e corrigindo falhas transientes ocorridas naturalmente ou por injeção maliciosa. Além disso, tais mecanismos são capazes de delinear com múltiplas falhas simultaneamente. Isso foi feito considerando-se a viabilidade técnica e comercial. Na próxima seção serão abordados os principais trabalhos elaborados para prevenir DFA.

4.5 Trabalhos Relacionados

Existem muitos trabalhos que prometem contramedidas para *fault attacks* para os principais algoritmos criptográficos, conforme será apresentado nesta seção. Devido à natureza dos CI's e das falhas, sejam aquelas causadas por motivos naturais ou aquelas geradas maliciosamente, é difícil encontrar soluções efetivas em nível de portas lógicas. Assim, a maior parte dos estudos propõe soluções em nível algorítmico.

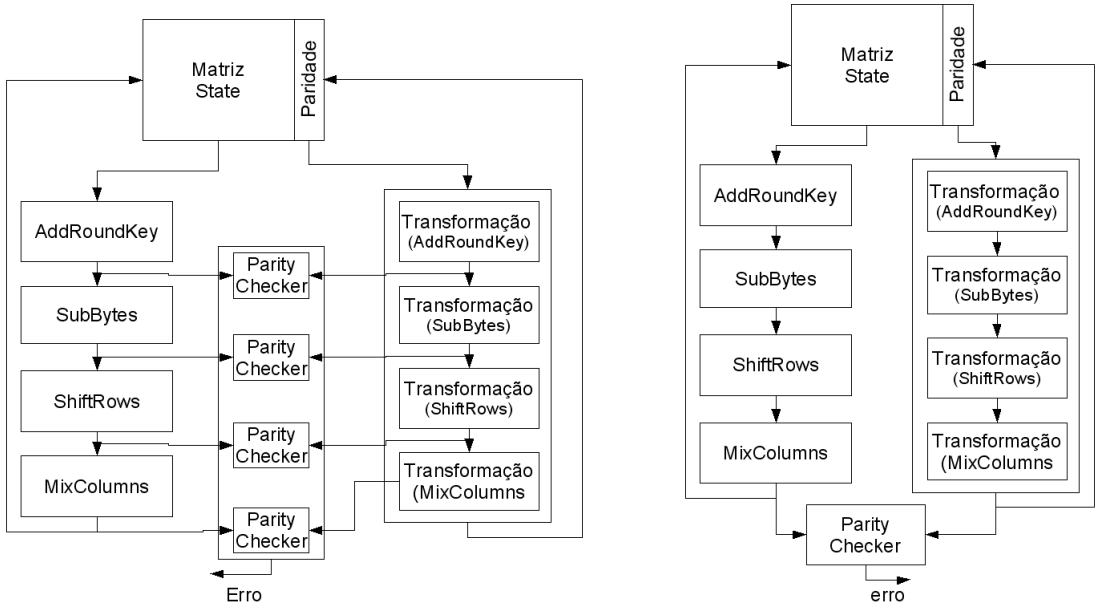
São raras as oportunidades de confrontar teoria e prática, especialmente em se tratando de injeção de falhas em *hardware*. O principal motivo é o alto custo da produção de ASICs para a realização de testes práticos. Porém, (MONNET et al., 2006) é um dos poucos trabalhos que apresentam injeção de falhas feita diretamente sobre o *hardware* criptográfico. Neste caso, o algoritmo criptográfico atacado foi o DES. Foram produzidas duas versões do *core*, uma protegida com os mecanismos no trabalho e a outra versão sem proteção usada como referência. A tecnologia de fabricação usada foi CMOS 130 nanômetros. Diferentemente da maioria das implementações, esta versão do DES é assíncrona e certas características dos circuitos assíncronos são aproveitadas como mecanismos de tolerância a falhas. A validação foi realizada com a injeção de falhas utilizando um *laser*. A Tabela 4.1 mostra os resultados para a injeção de falhas nas duas versões. A versão não protegida (referência) apresentou falhas em 17% das 5600 injeções, enquanto que a versão com mecânismos de tolerância a falhas apresentou problemas apenas em 7% da vezes. Foram injetadas falhas apenas nas regiões do *core* ocupadas pelas *Sboxes*, já que o mecanismo de proteção foi implementado apenas para elas. Para a versão protegida, as falhas que não puderam ser corrigidas causaram a parada do processador, impedindo que o atacante obtenha textos cifrados com falhas. O mecânismo de proteção implicou em um *overhead* de 8% para proteção apenas das *SBoxes*.

Para o AES, uma solução sugerida por (BERTONI et al., 2002) utiliza a idéia de circuitos *self-checking* discutida na Seção 4.4. Tal solução implementa um esquema de detecção de falhas usando *bits* de paridade. Quando uma falha é detectada a computação é interrompida sem exibir qualquer resultado. Para cada *byte* da *Matriz State* um *bit*

Módulo	***Overhead***
SubBytes	12,5%
ShiftRows	12,5%
MixColumns	10,2%
AddRoundKey	12,5%

Tabela 4.2: *Overhead* para o esquema de detecção de falhas para o AES proposto por (BERTONI et al., 2002).

de paridade foi associado. Para cada uma das quatro operações criptográficas do AES, uma operação equivalente é executada sobre o *bit* de paridade, afim de manter a validade do mesmo. A Figura 4.4 ilustra tal esquema. No Capítulo 6 a implementação destas operações de equivalência serão abordadas com mais detalhes. Este esquema pode ser adotado com diferentes latências na detecção de falhas. Por exemplo, a checagem da validade da paridade pode se feita no final de cada operação, o que implica em menor latência para a detecção de falhas. Porém, exige o uso de quatro checadores de paridade e causa um acréscimo no tempo de computação. A Figura 4.4(a) ilustra tal esquema. Outra forma é fazer a checagem da paridade apenas no final de cada iteração. Isso implica no uso de apenas um comparador, um acréscimo menor no tempo de processamento, porém, uma maior latência para a detecção de falhas. A Figura 4.4(b) ilustra este esquema.



(a) Detecção de falhas no final de cada operação.

(b) Detecção de falhas no final de cada iteração.

Figura 4.4: Esquema de detecção de falhas no AES proposto por (BERTONI et al., 2002).

A Tabela 4.2 mostra o *overhead* de área para a implementação do módulo preditor de paridade para cada uma das operações do AES. Esta solução é bastante atraente em termos de consumo de área. Porém, não é capaz de corrigir a falha e retomar a execução. No Capítulo 6 este mesmo esquema será associado com mecanismos de correção de erros em memória para permitir detecção e correção de falhas.

Uma solução que suporta recuperação de falhas é proposta por (BREVEGLIERI; KOREN; MAISTRI, 2005) para uma implementação específica do AES em *hardware* proposta por (MANGARD; AIGNER; DOMINIKUS, 2003). A Figura 4.5 é uma representação simplificada desta implementação. Os dados de entrada são armazenados nos chamados *data cells* (DC's). Cada DC armazena um *byte*. Conforme a figura, existem quatro *Sboxes* (SB's) responsáveis pela operação *SubBytes*. Isso significa que os dados são rotacionados verticalmente e cada SB é responsável por uma coluna. A operação *ShiftRows* é executada pelos multiplexadores postos logo após a primeira linha. As operações *MixColumns* e *AddRoundKey* são computadas em apenas um ciclo de *clock*. Para isto, cada DC comunica-se com os outros DC's na mesma coluna. O *Key Schedule* atualiza as mesmas SB's economizando área.

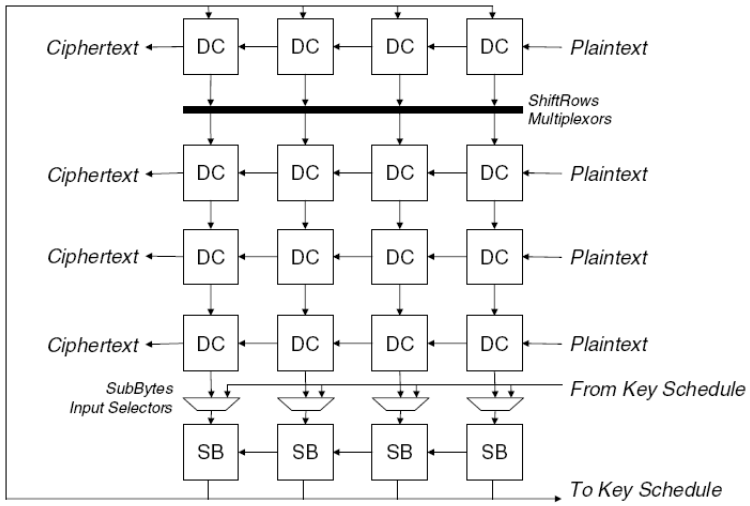


Figura 4.5: Representação da arquitetura para o AES proposta por (MANGARD; AIGNER; DOMINIKUS, 2003).

Para suportar recuperação de falhas, (BREVEGLIERI; KOREN; MAISTRI, 2005) propôs modificações na arquitetura apresentada. Tal proposta une o esquema de predição de paridade apresentado por (BERTONI et al., 2002) com a adição de DC's sobressalentes. Essas modificações são aplicadas sobre as linhas dos DC's. A Figura 4.6 apresenta esta idéia. Considerando que os quatro DC's são chamados de A, B, C e D e que o DC sobressalente é denominado E, se um erro ocorrer em B, então a entrada de B é direcionada para C, a entrada de C é direcionada para D e, finalmente, a entrada de D é direcionada para E.

Tal solução suporta uma falha por linha, então, quatro falhas simultâneas em linhas diferentes podem ser recuperadas. Duas falhas ocorridas na mesma linha causa a suspensão da execução. Porém, como essa implementação usa o esquema de predição de paridade de (BERTONI et al., 2002) ela também é vulnerável a múltiplas falhas dentro de um mesmo DC. Sendo que números pares de falhas não são detectados. Desta forma, os autores conseguiram tolerância a falhas simples, mas com um acréscimo de área que ficou em 40% para apenas detecção e 134% para detecção e correção.

Para o RSA, (BLöMER; OTTO; SEIFERT, 2003) propõe alterações no algoritmo RSA-CRT (RSA que usa o teorema do resto chinês) para torna-lo resistente a DFA. Tal solução reduz consideravelmente as chances de um atacante conseguir extrair textos úteis para criptoanálise. Esta solução apresenta algumas desvantagens, como o aumento do

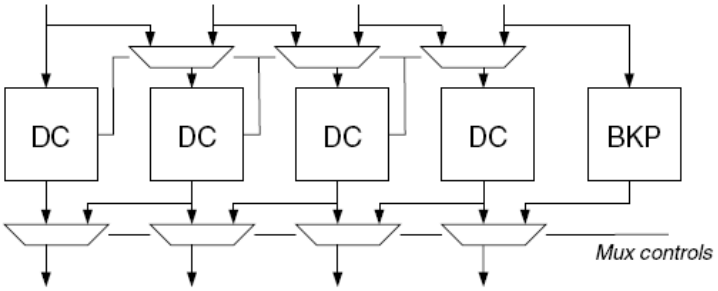


Figura 4.6: Linha reconfigurável com elemento sobressalente.

tempo de execução, pois o novo algoritmo proposto é mais oneroso. Além disso, este tipo de solução propõe apenas mudanças na implementação do algoritmo, deixando o *hardware* vulnerável a falhas ocorridas naturalmente. As falhas injetadas maliciosamente ainda afetam os textos cifrados.

Com visto nos trabalhos apresentados percebe-se a necessidade de esquemas mais eficientes contra DFA. Os capítulos a seguir apresentam técnicas para o projeto de hardwares criptográficos tolerantes a falhas. Os algoritmos abordados foram o DES e o AES, respectivamente.

5 CONTRAMEDIDA PARA DFA BASEADA EM DUPLICAÇÃO PARCIAL

Em vista das vulnerabilidades apresentadas no Capítulo 4, este capítulo propõe uma abordagem para detecção e correção de falhas concorrentes em *hardware* criptográfico. Tal abordagem é baseada na duplicação parcial do *hardware*. Este esquema pode lidar com falhas simples e múltiplas e mostrou ser uma interessante medida para proteção de sistemas criptográficos contra DFA. A principal característica da duplicação parcial é a capacidade de proteger as partes mais vulneráveis do *core*, ou seja, as partes onde a injeção de falhas pode levar à descoberta de informações sigilosas. Esta proteção extra é conseguida porque a duplicação parcial fornece tolerância a falhas, onde múltiplas falhas podem ser detectadas e corrigidas simultaneamente. Para a validação da técnica foi utilizado o algoritmo criptográfico DES, por ser um algoritmo amplamente utilizado e de fácil entendimento. Como parte do estudo das vulnerabilidades do DES, uma simulação de um ataque por injeção de falhas foi realizada em uma implementação em linguagem de descrição de *hardware* VHDL. Na seqüência, foi desenvolvido um programa, em linguagem Java, capaz de recuperar a chave criptográfica a partir de blocos cifrados com falhas. Assim, comprova-se empiricamente a validade das equações apresentadas na Subseção 4.3.1. Os resultados da simulação de ataque foram usados como base para a formulação da contramedida, que culminou no desenvolvimento de um co-processador criptográfico. Por fim, os resultados mostram que é possível atingir um nível considerável de proteção com custos aceitáveis.

Este capítulo é organizado da seguinte forma: A Seção 5.1 aborda a arquitetura da implementação do DES, que foi utilizada como plataforma dos experimentos realizados. A Seção 5.2 mostra como foi conduzido o experimento de DFA contra o DES. A Seção 5.3 apresenta o esquema de duplicação parcial do *hardware* aplicado e validado no DES. Por fim, a Seção 5.4 faz as considerações finais sobre a técnica de duplicação parcial e os resultados obtidos em sua aplicação ao DES.

5.1 Arquitetura da implementação do DES

Esta seção traz detalhes da versão do DES utilizado para validação do esquema de duplicação parcial. Esta implementação foi obtida em (MCQUEEN, 2005). Tal versão do DES implementa tanto o processo de criptografia quanto o de descriptografia. De fato, implementar os dois processos é simples, pois é necessário apenas inverter a aplicação das sub-chaves para descriptografar. Esta característica é interessante para a duplicação parcial, já que partes duplicadas podem proteger o *core* tanto na criptografia quanto na descriptografia, o que mais uma vez, torna o DES uma boa plataforma de testes para tal

técnica.

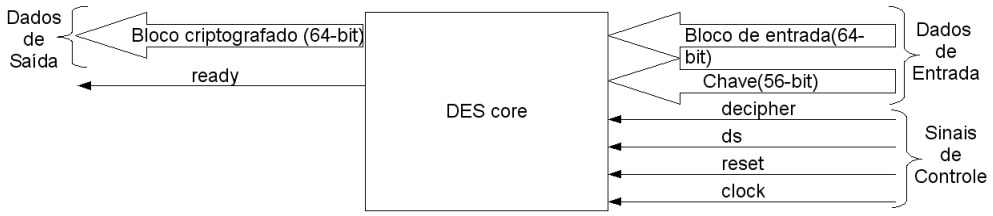


Figura 5.1: Diagrama simplificado das entradas e saídas do *core*.

A Figura 5.1 apresenta os sinais de entrada e saída do *core*. Como entrada, o *core* recebe 64-*bit* de dados e a chave criptográfica de 56-*bit*. Como saída, ele fornece 64-*bit* de dados processados (criptografados ou descriptografados). O sinal de controle *decipher* é ajustado para 1 para indicar uma operação de descriptografia, enquanto que 0 indica operação de criptografia. O sinal *ds* sinaliza o inicio das operações criptográficas do *core*. Quando as operações criptográficas são finalizadas o sinal *ready* é ajustado para 1 e os dados são apresentados na saída.

A arquitetura do *core* é semelhante à Figura 3.5. Sendo que a função f é implementada de forma combinacional. Na entrada da função f existem dois registradores denominados L e R. Tais registradores armazenam os dados parcialmente criptografados ao final de cada iteração, até a 16° iteração. O módulo gerador das sub-chaves (*key schedule*) também é combinacional. As sub-chaves são geradas na inicialização do *core* e armazenadas em uma memória.

5.2 Simulação de DFA no DES

Para ajudar a entender como um ataque por DFA acontece, foi realizada uma simulação de ataque. Esta simulação consistiu na injeção de falhas no *core* do DES descrito na seção anterior. Com posse dos blocos cifrados com falhas, foi desenvolvido um programa em linguagem Java capaz de determinar a sub-chave K_{16}, conforme o método descrito na Subseção 4.3.1. A Subseção 5.2.1 explica como o experimento de injeção de falhas foi conduzido. A Subseção 5.2.2 descreve em detalhes o programa desenvolvido. Por fim, a Subseção 5.2.3 apresenta os resultado obtidos.

5.2.1 Configuração do experimento de injeção de falhas

Para o experimento de injeção de falhas no *core* do DES é importante destacar que foi seguido o modelo de falhas apresentado na Subseção 4.3.1, porém, com uma pequena extensão. Tal modelo considera apenas falhas simples injetadas no registrador R na 15° iteração, daqui para frente chamado de R_{15}. Este experimento vai um pouco mais além, injetando múltiplas falhas no registrador R_{15}. O objetivo desta extensão do modelo de falhas original é mostrar que múltiplas falhas podem ser úteis ao atacante, diminuindo o montante de informação necessário para extração da chave.

Para permitir injeção de falhas no registrador R_{15}, o *core* original precisou ser modificado. Tal modificação consistiu na adição de um multiplexador que permite selecionar entre o valor correto e o valor com falhas. A geração do valor com falhas é feita através da aplicação de uma máscara no valor original usando-se uma operação ou-exclusivo. Tal técnica é ilustrada na Figura 5.2 e é baseada em (LIMA KASTENSMIDT, 2003). O valor com falha deve ser injetado no momento correto para atingir o registrador R na 15° itera-

ção da função f, conforme explicado na Subseção 4.3.1. Para isso, um processo VHDL foi adicionado para controlar o multiplexador e selecionar o valor com a máscara no tempo determinado. A máscara é determinada aleatoriamente, e consiste em uma palavra de 32-*bit* com 0's nas posições que não devem ser afetadas e 1's nas posições que devem apresentar falhas. A quantidade de falhas simultâneas é ajustada estaticamente antes da simulação, porém, as posições das falhas variam aleatoriamente durante a simulação.

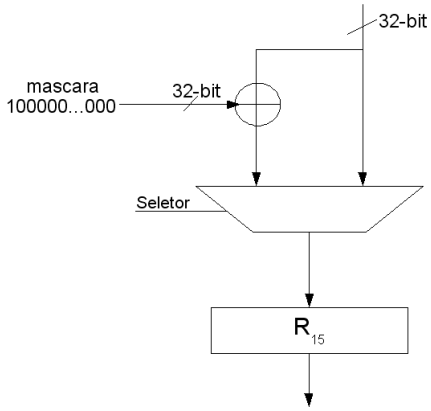


Figura 5.2: Esquema para injeção de falhas no registrador R_{15}.

O modelo de falhas adotado para este ataque é bastante restrito, já que supõe que o atacante possuirá técnicas, equipamentos e conhecimentos bastante específicos a respeito do *core*. Porém, tal modelo é didático e pôde ser simulado e demonstrado. Este experimento é importante para demostrar a validade das técnicas de DFA. Contudo, a técnica de proteção proposta na Seção 5.3 é capaz de oferecer um nível maior de proteção, mesmo para modelos de falhas mais elaborados. Isto é possível porque, tal técnica é baseada na detecção e correção de falhas, sem se prender a um modelo de falhas específico.

5.2.2 Programa para extração da sub-chave K_{16}

O programa toma como entrada um único bloco de dados criptografado sem injeção de falhas e diversos blocos criptografados com injeção de falhas. Todos os blocos devem ter sido criptografados a partir do mesmo bloco de dados e da mesma chave. Como saída, o programa retorna a chave criptográfica usada ou um subconjunto de possíveis chaves. Para determinar a chave correta, o programa precisa de uma certa quantidade de blocos cifrados com falhas. Caso a quantidade de blocos não seja suficiente, o programa retorna um subconjunto de chaves, onde a chave correta está inclusa.

Para inferir a chave K_{16}, a partir de blocos criptografados, o programa implementa um método de resolução para a Equação 4.3. É necessário determinar a chave K_{16} em um espaço de busca de 2^{48} possibilidades. Porém, devido às vulnerabilidades exploradas pela injeção de falhas, esse espaço de busca é reduzido a poucas tentativas. Isso é possível por causa da independência das *SBoxes*, conforme será explicado na seqüência. A Figura 3.6, mostra como ocorre a substituição de uma palavra de 48-*bit* por uma de 32-*bit* na função f. A palavra na entrada das *S-Boxes* é dividida em 8 blocos de 6-*bit* cada. Então, cada bloco de 6-*bit* é tratado independentemente dos outros blocos. Como saída, as *S-Boxes* fornecem 6 blocos de 4-*bits* cada. Esta independência das *S-Boxes* sobre os dados causam uma vulnerabilidade que é explorada pelo programa de busca da chave.

A Figura 5.3 é um diagrama de blocos do *datapath* do programa . O programa começa determinando L_{16} e R_{16}, que são oriundos do bloco criptografado sem falhas. Para isso,

o programa aplica a inversa da permutação final, denominada *Inv_PF* e separa o bloco resultante de 64-*bit* em dois sub-blocos de 32-*bit* cada (L_{16} e R_{16}). Estes valores serão usados ao longo de toda a execução. Deste ponto em diante, o programa executa um certo número de iterações que vai até a determinação da chave K_{16} ou o esgotamento dos blocos de dados criptografados com falhas. As iterações são representadas pelas linhas pontilhadas. A cada iteração, um novo bloco com falhas é tomado como entrada e o espaço de busca é reduzido pela eliminação de chaves que não satisfazem a Equação 4.3. Cada nova iteração começa com a determinação de $\hat{L}_{16}$ e $\hat{R}_{16}$ a partir do bloco cifrado com falhas pela aplicação da permutação final inversa (*Inv_PF*). O passo seguinte é a aplicação da operação permutação/expansão sobre L_{16} e $\hat{L}_{16}$. Neste ponto, entra a vulnerabilidade das *S-Boxes* citada anteriormente. É necessário, para cada bloco de 6-*bit* oriundo da operação expansão/permutação, determinar se:

$$R_{16} \oplus \hat{R}_{16} \neq 0.$$

Se a desigualdade for verdadeira, então uma falha atingiu o bloco de 6-*bit* em questão. Isso significa que este bloco poderá ser usado para inferir parte da chave criptográfica. Agora, é necessário testar todas as possíveis chaves para o bloco de 6-*bit* e determinar quais satisfazem a Equação 4.3. Porém, as possíveis combinações de *bits* para a formação da chave estão limitadas a um espaço de busca de 2^6, já que a palavra de 48-*bit* oriunda da operação permutação/expansão pode ser sub-dividida em 8 blocos de 6-*bit*. Então, uma operação ou-exclusivo é executada sobre L_{16} e $\hat{L}_{16}$ com uma chave candidata. O resultado passa pela sua respectiva *S-Box* e a operação de permutação P é executada, para ambos L_{16} e $\hat{L}_{16}$. Finalmente a igualdade entre os lados da Equação 4.3 pode ser verificada. Se for verdadeira, o sub-bloco da chave testado é mantido no espaço de busca. Caso contrário, ele é eliminado diminuindo o espaço de busca.

O espaço de busca tratado pelo programa é inicialmente 8×2^6, o que é muito inferior as 2^{48} tentativas necessárias para determinar K_{16} por força bruta. Ainda, a cada iteração o número de possibilidades é diminuído pela exclusão dos sub-blocos da chave que não pertencem à chave verdadeira. A chave verdadeira consegue satisfazer à Equação 4.3 para qualquer falha simples ou múltipla ocorrida em R_{15}. Obtendo-se a chave K_{16}, é possível realizar o processo inverso do algoritmo de expansão da chave e obter 48-*bit* da chave original. Os outros 8-*bit* podem ser determinados por força bruta, em no máximo 256 tentativas.

5.2.3 Resultados da simulação de DFA

Uma única falha no registrador R_{15} pode afetar até duas *S-Boxes* devido à operação de expansão/permutação, onde alguns *bits* são duplicados. Quando falhas atingem mais de uma *S-Box*, o programa descrito pode reduzir o número de blocos necessários para convergir para a chave. Ele verifica cada *S-Box* determinando se existe alguma falha incidente. Caso esta falha exista, ele realiza o procedimento para determinar os sub-blocos da chave que satisfazem a Equação 4.3, conforme descrito na seção anterior. Ainda, se múltiplas falhas atingirem o registrador R_{15}, as mesmas atingirão várias *S-Boxes* simultaneamente e o número de blocos necessários para encontrar a chave será diminuído. Então, um atacante pode atuar causando múltiplas falhas no registrador R_{15}, a fim de aumentar a eficiência do ataque.

O experimento de injeção de falhas consistiu na injeção de 1 a 8 falhas a cada conjunto de simulações. O propósito desta injeção controlada de falhas foi analisar a quantidade de blocos necessários para obter a chave criptográfica. Para cada número de falhas injetadas

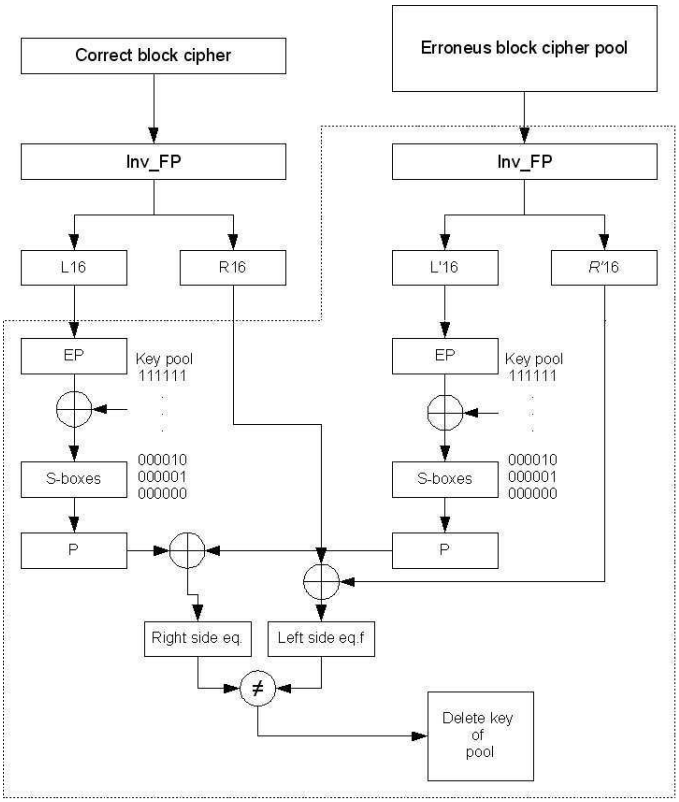


Figura 5.3: Diagrama de blocos do algoritmo de busca de chave.

foram realizadas uma bateria de 10 simulações, afim de determinar a média de blocos necessária para encontrar a chave K_{16}. Cada simulação consistiu de 200 execuções, onde uma execução significa realizar as 16 iterações criptográficas, processando um bloco de dados. Desta forma, houve 80 simulações de falhas, onde cada simulação gerou 200 blocos criptografados com falhas, totalizando 16000 blocos cifrados com falhas. Cada grupo de blocos cifrados com um determinado número de falhas foi submetido ao programa para obtenção da chave K_{16}. O número de blocos necessários para convergir para a chave em cada caso foi calculado. O gráfico apresentado na Figura 5.4 foi plotado com base na média de blocos necessários para obtenção da chave em relação ao número de falhas injetadas.

A Figura 5.4 mostra, basicamente, que quanto mais falhas forem injetadas menos blocos cifrados são necessários. Por exemplo, para uma única falha em R_{15}, são necessários mais de 35 blocos, enquanto que para 5 falhas simultâneas são necessários menos de 10 blocos. Isso acontece porque com um maior número de falhas mais *S-Boxes* são atingidas e mais *bits* da chave podem ser inferidos a partir do mesmo bloco. Contudo, a partir de 6 falhas simultâneas, a quantidade de blocos necessários decresce lentamente. Isto é devido ao fato de que mais de uma falha atinge a mesma *S-Box*, o que não ajuda na dedução da chave. Em um ataque de injeção de falhas ideal, cada falha deveria atingir um *bit* distinto de R_{15}. Todavia, em um ataque real, esta precisão na injeção de falhas é improvável. Além disso, falhas podem atingir o registrador R_{15} em outras iterações, ou ainda outras partes do *hardware*. Em um ataque mais elaborado, capaz de lidar com falhas em iterações anteriores a 15°, tais falhas ainda são úteis para obtenção da chave. Ou ainda, usando-se o mesmo modelo de ataque, pode-se identificar e excluir blocos, cujas falhas atingiram outras iterações, que não a 15°.

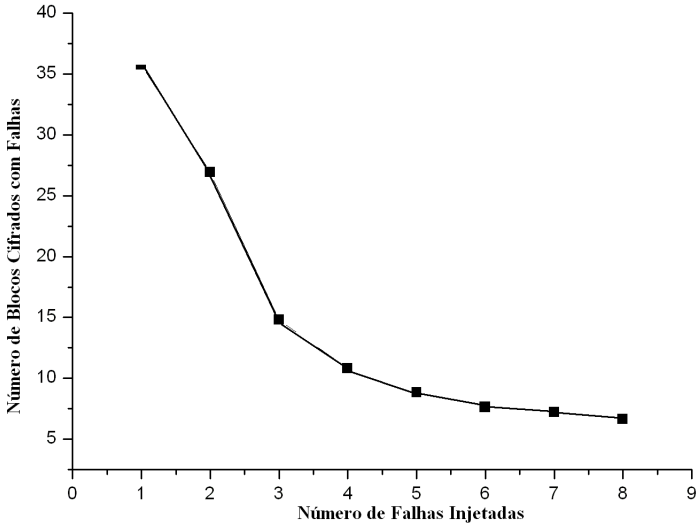


Figura 5.4: Relação entre o número de falhas injetadas e o número de blocos cifrados necessários para obter K_{16}.

5.3 Duplicação parcial aplicada ao DES

Os experimentos realizados na Seção 5.2 demonstram a vulnerabilidade dos sistemas criptográficos baseados no DES a ataques com DFA. Para lidar com esta vulnerabilidade, um esquema de proteção baseado na duplicação parcial do *hardware* foi idealizado. Este esquema mostrou-se efetivo para conter ataques por DFA, além de poder ser adotado em outros algoritmos criptográficos como o AES, conforme será abordado no Capítulo 6. O restante desta seção apresenta o esquema de proteção proposto, assim como os resultados atingidos.

5.3.1 Função f tolerante a DFA

Com base nas deduções da Subseção 4.3.1 e nos resultados apresentados na Seção 5.2, concluiu-se que a principal vulnerabilidade do DES esta na função f. Assim, são necessários mecanismos de detecção de falhas capazes de impedir a apresentação de resultados incorretos na ocorrência de falhas na função f. Contudo, um grau adicional de proteção pode ser alcançado se, além da detecção, ocorre a correção das falhas sem interrupção do processamento. Para um atacante, a transparência na ocorrência de falhas causa dúvidas na eficácia do método de injeção de falhas. Pois, ele é incapaz de distinguir se existe algum mecanismo de tolerância a falhas protegendo o *core*, ou se seu método de injeção de falhas é ineficiente. A duplicação parcial, usada para detecção de falhas, também pode fornecer um método de correção para falhas transientes, isso, com pequenas modificações na lógica de controle.

Alternativamente à duplicação parcial, outros métodos de proteção foram avaliados, conforme discutido a seguir:

- Duplicação total do *core*: Neste esquema, um segundo processador criptográfico, semelhante ao original, executa o processamento em paralelo. Ao final, os resultados são comparados e caso sejam diferentes, os dados precisam ser computados

novamente. A principal vantagem desta técnica é que todo o *core* é protegido. Porém, o *overhead* de área é mais que o dobro, devido ao esquema de comparação e lógica de controle adicional, o que também implica em maior consumo de energético. Além disso, a latência para detecção e correção de falhas é muito superior à duplicação parcial, já que os resultados são comparados apenas ao final;

- Redundância temporal: Na redundância temporal o mesmo processador computa os dados duas vezes. Ao final da primeira execução, o resultado é armazenado temporariamente para poder ser comparado com o resultado da segunda execução. Este esquema tem como vantagens pequeno *overhead* de área e boa cobertura de falhas. Porém, demanda ao menos o dobro de tempo de execução, mesmo para execuções sem falhas. Mais tempo de execução significa maior consumo energético, o que pode ser um problema para dispositivos móveis com severas restrições de consumo. Além disso, a latência para detecção e correção de falhas é maior que na duplicação total do *core*. Ainda, Biham e Shamir (A. SHAMIR, 1997) afirmam que computar os dados criptográficos duas vezes pode não ser suficientemente seguro, pois a probabilidade da mesma falha ocorrer durante ambas as execuções pode não ser baixa o suficiente. Isso ocorre porque o atacante pode estar focando a injeção de falhas em um determinado registrador.

A contramedida proposta neste capítulo consiste em adicionar ao *core* um co-processador simplificado. Este co-processador irá assistir a execução do *core* principal detectando e corrigindo falhas transientes. Um esquema similar foi proposto por (WEAVER; GEBARA; BROWN, 2002) para permitir ajuste dinâmico da freqüência do processador. O intuito deste trabalho é permitir que o processador principal trabalhe a máxima freqüência sem erros de processamento. A freqüência é determinada através de verificação dinâmica, onde um co-processador simplificado re-executa as instruções detectando falhas transientes e reajustando a freqüência caso a incidência de falhas seja muito alta. Porém, o objetivo do co-processador proposto neste trabalho é apenas detectar e corrigir falhas transientes diminuindo as chances da ocorrência de êxito em um ataque por DFA.

Deste ponto em diante, o co-processador proposto será chamado de *hardware assistente*. O *hardware* assistente possui algumas vantagens em termos de segurança. Primeiramente, ele é uma versão simplificada do processador original, possuindo assim menos pontos de injeção de falhas. Em segundo, ele é um *hardware* combinacional. No Capítulo 6, será mostrado que lógica combinacional é menos vulnerável à ocorrência de falhas transientes que registradores. Conforme discutido nas seções anteriores, sabe-se que a principal vulnerabilidade do DES é a função f juntamente com o registrador R_{15}. Portanto, o *hardware* assistente consiste em uma implementação da função f, porém sem qualquer registrador.

Conforme mencionado anteriormente, a implementação do DES adotada possui a função f implementada de forma combinacional, o que facilitou a implementação do *hardware* assistente. O *hardware* assistente consiste basicamente em uma réplica da função f, um comparador e um verificador de paridade. Ele computa as mesmas entradas da função f do processador original para cada iteração, visando garantir a integridade dos dados. O registrador R_{15} presente no processador original foi protegido com paridade e o verificador de paridade do *hardware* assistente verifica sua validade. Caso seja detectado algum erro, a computação é interrompida. O comparador permite a comparação entre os resultados do *hardware* assistente e do processador principal para a função f. Se a comparação determinar que os resultados não são iguais, isso significa que houve falhas no

processamento e a última iteração precisa ser executada novamente. A comparação é efetuada ao final de cada iteração, o que permite uma menor latência na detecção e correção de erros.

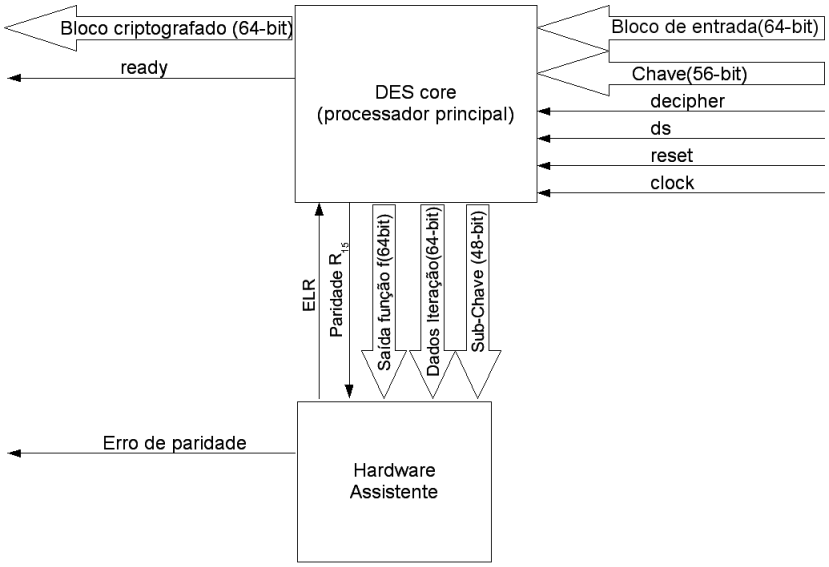


Figura 5.5: Diagrama de blocos do sistema resultante.

A Figura 5.5 mostra um diagrama de blocos do sistema resultante, ou seja, o processador principal, o *hardware* assistente, os sinais de entrada e saída e os sinais de comunicação entre os dois *cores*. O *hardware* assistente toma como entrada a sub-chave da iteração corrente (K_n), as entradas da função f que correspondem a L_n e R_n para cada iteração n e, finalmente, a paridade dos registradores L e R. O sinal *ELR* é usado pelo *hardware* assistente para sinalizar a detecção de uma falha na função f. Quando tal falha é sinalizada o processador principal executa novamente a última iteração a fim de eliminar as falhas no processamento. Antes da execução de cada iteração, o *hardware* assistente verifica a validade da paridade para as entradas da função f. Se houver falha na verificação de paridade o sistema não pode se recuperar, então a execução do *core* é interrompida e o sinal *erro de paridade* é usado para sinalizar ao sistema externo tal interrupção. Neste caso, nenhum resultado é exibido e o processador principal precisa ser reinicializado antes de iniciar um novo processamento.

Para suportar as modificações propostas, o processador principal precisou passar por algumas adaptações. A principal mudança foi a adição do sinal *ELR*. Conforme dito anteriormente, este sinal indica que a última iteração da função f precisa ser executada novamente. Para isto, o processador principal deve salvar as entradas da função f em um registrador auxiliar. Tal registrador é protegido por um *bit* de paridade, e este *bit* é usado pelo *hardware* assistente para verificação da consistência dos dados e interromper a execução caso seja necessário.

5.3.2 Resultados Experimentais

De acordo com a implementação do DES utilizada, o *hardware* assistente também foi implementado em linguagem de descrição de *hardware* VHDL. Os experimentos de injeção de falhas foram realizados com auxílio do ModelSim XE III 6.0a da Xilinx, e serviram para avaliar o nível de segurança obtido no sistema resultante. Tal sistema foi modificado para suportar injeção de falhas conforme o esquema descrito na Subseção 5.2.1. Foram

escolhidos pontos de injeção de falhas estratégicos a fim de permitir uma validação qualitativa do sistema. A injeção de falhas no sistema resultante ajudou a identificar as partes protegidas e desprotegidas e o nível de proteção alcançado.

Durante a simulação, foram injetadas falhas nas entradas da função f, o que significa que os registradores afetados foram os registradores R e L e suas réplicas, além de falhas dentro da função f e em partes do *hardware* antes e depois da função f. Estas falhas foram injetadas em momentos aleatórios da execução do *core*. A Tabela 5.1 resume os resultados experimentais. Tais resultados mostram que o sistema conseguiu detectar e corrigir todas as falhas injetadas dentro da função f. De fato, a duplicação da função f permite um alto grau de segurança devido à baixa probabilidade de ocorrência de falhas, tanto no *hardware* assistente quanto no processador original, que resultem em saídas iguais. Porém, a correção das falhas, neste ponto, exige um ciclo de *clock* a mais a cada iteração afetada. Um atacante seria capaz de perceber uma leve degradação no desempenho do *hardware* sob ataque. Falhas nos registradores das entradas da função f também puderam ser detectadas. Porém, neste ponto as falhas não podem ser corrigidas, pois o *bit* de paridade utilizado permite apenas detecção. Outro fator é que apenas falhas simples ou em número ímpar podem ser detectadas. Contudo, proteção adicional pode ser conseguida com o uso de códigos detectores/corretores de erros nos registradores, conforme será abordado no Capítulo 6. As falhas que ocorrem em alguma parte do *datapath* antes ou depois da função f não serão detectadas. Isso era esperado, pois tais partes do *hardware* não foram incluídas na duplicação parcial. Em termos de segurança, isso não é um problema já que falhas ocorridas em regiões fora da função f e suas entradas não permitem dedução da chave criptográfica. Além disso, cerca de 85% do tempo de processamento é gasto nas 16 iterações da função f, ou seja, o processamento concentra-se nas partes protegidas do *core*. O módulo gerador das sub-chaves continua sendo vulnerável à DFA, porém, ele também pode ser facilmente protegido por duplicação parcial.

Local da injeção da falha	**Efeito no sistema resultante**
Qualquer registrador do *datapath* antes da função f	Não detectado, o resultado é corrompido, porém a chave não pode ser revelada
Nos registradores das entradas da função f	Detectado, a execução é interrompida e o resultando não é apresentado
Nos sinais dentro da função f	Detectado e corrigido
Registradores ou sinais do *datapath* após a função f	Não detectado, o resultado é corrompido, porém a chave não pode ser revelada
Sinais do módulo gerador de sub-chaves	Corrompe os resultados e a chave pode ser revelada

Tabela 5.1: Comportamento do sistema protegido na ocorrência de falhas transientes.

A Tabela 5.2 apresenta os resultados de frequência de operação e área para o *core* original (DES *core*) e o sistema resultante. Tais informações foram obtidas através da ferramenta de síntese LeonardoSpectrum. Primeiramente, pode-se observar uma penalidade de performance de cerca de 10% para o sistema protegido. Esse descrécimo de performance é devido à adição do comparador ao final da função f, forçando a comparação no mesmo ciclo de *clock*. Visto que a função f já representava o caminho crítico do *core*. O *overhead* de área é de aproximadamente 38%, o que é muito inferior à duplicação total do *hardware*.

	Clock (*MHz*)	**Área (*gates*)**
DES *core*	165	3291
Sistema Resultante	150	4561

Tabela 5.2: Comparação entre o *core* original e o sistema resultante.

5.4 Considerações finais

A duplicação parcial mostrou ser uma eficiente técnica para proteção de lógica combinacional contra injeção maliciosa de falhas. A validação desta técnica com a aplicação no DES abriu portas para novos experimentos em outros algoritmos criptográficos. O Capítulo 6 apresenta técnicas para proteção do algoritmo criptográfico AES, onde a duplicação parcial foi utilizada como parte da solução. Assim, o uso desta técnica para proteção de algoritmos criptográficos começa na identificação dos principais componentes criptográficos vulneráveis a DFA e, posteriormente, na duplicação dos mesmos. Neste trabalho, a duplicação parcial foi utilizada exclusivamente para a proteção de lógica combinacional. A proteção dos registradores identificados como vulneráveis a DFA foi feita apenas com um *bit* de paridade. Este esquema, apesar de diminuir a taxa de sucesso na injeção de falhas, pode não ser suficientemente seguro. Pois, não apresenta boa cobertura de falhas e é incapaz de fornecer transparência, ou seja, correção das falhas.

A proteção de registradores e memórias por duplicação apresenta algumas restrições. Com duplicação consegue-se apenas detectar falhas, correção é conseguida apenas com triplicação do *hardware*. Isso implica em grande *overhead* de área. Portanto, outros esquemas de proteção são mais adequados para uso em memórias. No Capítulo 6 será abordado o uso de códigos mais robustos para detecção e correção de falhas em memórias que, por exemplo, podem ser aplicados à memória das chaves. O uso de códigos robustos para proteção de memórias implica também na eficiência da duplicação parcial. A correção das falhas no *hardware* duplicado consiste em uma nova leitura dos dados em uma memória protegida e a re-execução da operação.

6 TÉCNICAS DE TOLERÂNCIA A FALHAS PARA PROTEÇÃO DO AES CONTRA DFA

Neste capítulo são apresentadas medidas para a proteção do AES contra DFA. Quatro diferentes soluções são apresentadas. Primeiramente, dois diferentes códigos corretores de erros são considerados para proteção das memórias. Estes códigos tem diferentes requisitos de área e cobertura de falhas. Similarmente, duas independentes contramedidas para o *hardware* combinacional são propostas. As técnicas para proteção das memórias e das partes combinacionais podem ser combinadas formando quatro versões com diferenças em área e tolerância a falhas. Todas as versões podem detectar e corrigir múltiplas falhas no *datapath*. Para validação dos sistemas propostos uma injeção de falhas massiva foi realizada. Os resultados mostram que as diferentes versões podem tolerar todas as falhas simples e drasticamente diminuir a manifestação de falhas múltiplas. Além disso, a solução mostra-se viável com custos e performance do sistema.

O capítulo é organizado da seguinte forma: A Seção 6.1 trás uma breve explicação da arquitetura do AES implementada. Esta mesma arquitetura será usada para validação dos esquemas de proteção. A seção 6.2 aborda detalhadamente os mecanismos de tolerância a falhas implementados, e que visam dar proteção contra falhas transientes a implementação do AES. A Seção 6.3 mostra como os experimentos de injeção de falhas foram realizados e os resultados obtidos. Por fim, a Seção 6.4 faz a avaliação final do resultados, com base nos resultados de tolerância a falhas e resultados de síntese.

6.1 Arquitetura do AES proposta

Uma implementação do módulo de encriptação e expansão da chave foi realizada como meio de validação dos esquemas de proteção propostos neste capítulo. A implementação suporta chaves de 128-*bit* e foi realizada de forma que as operações criptográficas são aplicadas seqüencialmente sobre a *matriz state*. Três opções de arquitetura foram julgadas:

- Uma arquitetura de 8-*bit*, capaz de operar sobre um *byte* da *matriz state* a cada ciclo de *clock*, resultando em maior tempo de processamento e menor área;
- Uma arquitetura de 32-*bit*, capaz de operar em uma linha ou coluna da *matriz state* a cada ciclo de *clock*, resultando em área e tempo de processamento médio;
- Uma arquitetura de 128-*bit*, capaz de operar sobre toda *matriz state* em apenas um ciclo de *clock*, resultando na maior área entre as três opções, porém com o menor tempo de processamento.

Optou-se pela arquitetura de 32-*bit* por oferecer o melhor compromisso entre área e desempenho. Operando sobre 4 *bytes* a cada ciclo de *clock* são necessários 4 ciclos para completar cada operação. Assim, cada iteração composta por 4 operações leva 16 ciclos de *clock*, exceto a última iteração que possui apenas 3 operações. As 10 iterações levam um total de 144 ciclos de *clock*.

Cada operação é implementada como um componente combinacional e os elementos de armazenamento (*matriz state* e memória das chaves) são implementados em componentes independentes. Desta forma, as técnicas de proteção para as partes combinacionais podem ser empregadas e testadas independentemente das técnicas de proteção de memória.

Como qualquer dispositivo em *hardware* esta implementação está sujeita à ocorrência de falhas transientes que resultam em erros de computação. Um *hardware* criptográfico pode, ainda, conforme mencionado nos capítulos anteriores, ser alvo de ataques visando a extração de informações confidenciais, por injeção de falhas. Os resultados das simulação de injeção de falhas para a implementação sem proteção mostram que 48,75% das falhas simples que atingem a *matriz state*, 94,50% das falhas incidentes na mémoria das chaves e 0,9% das falhas na parte combinacional levam a erros de computação, comprovando a vulnerabilidade do *hardware* desprotegido à manifestação de erros nos resultados devido a falhas ocorridas durante o procesamento.

6.2 Mecanismos de tolerância a falhas

Para proteção do *datapath* da implementação do AES, dois diferentes esquemas para proteção de memória e dois diferentes esquemas para proteção da lógica combinacional foram implementados. Tais implementações tem diferentes requisitos de área e tolerância a falhas e podem ser combinadas resultando em quatro diferentes níveis de proteção. Para proteção das memórias, foram adotados os código de *Hamming* e *Reed-Solomon*, ambos os códigos são detectores/corretores de erros, conforme será melhor explicado na Subseção 6.2.1. Para proteção da lógica combinacional foi adotado duplicação parcial (semelhante ao que foi feito com o DES no Capítulo 5) e o esquema de predição de paridade descrito por (BERTONI et al., 2002), isso será detalhado na Subseção 6.2.2. A Figura 6.1 ilustra o esquema de proteção de forma genérica. As memórias foram ajustadas para armazenar *bits* de redundância. Estes *bits* formam palavras de códigos que permitem verificar a consistência dos dados armazenados. Para tanto, é necessário um módulo codificador/decodificador que codifique as palavras de dados para escrita na memória e as decodifique durante as leituras. Desta forma, os dados armazenados nas memórias ficam protegidos contra a ocorrência de falhas. Na parte combinacional, um *hardware* também combinacional foi adicionado para detectar possíveis ocorrências de falhas em paralelo às operações normais. Quando falhas são detectadas é necessário uma nova leitura na memória afim de recalcular a operação afetada.

A Subseção 6.2.1 apresenta os códigos detectores/corretores de erros utilizados. Serão discutidos os modelos de falhas suportados pelas implementações adotadas. Os detalhes de implementação são importantes para o entendimento dos resultados apresentados na Seção 6.3. Decisões de projeto são tomadas, entre outros fatores, com base em custos, tempo de implementação e requisitos da aplicação. Neste caso, tais decisões influenciaram no nível de tolerância a falhas alcançado. A Subseção 6.2.2 faz uma discussão detalhada a respeito dos esquemas de proteção adotados para a lógica combinacional e os respectivos modelos de falhas tolerados.

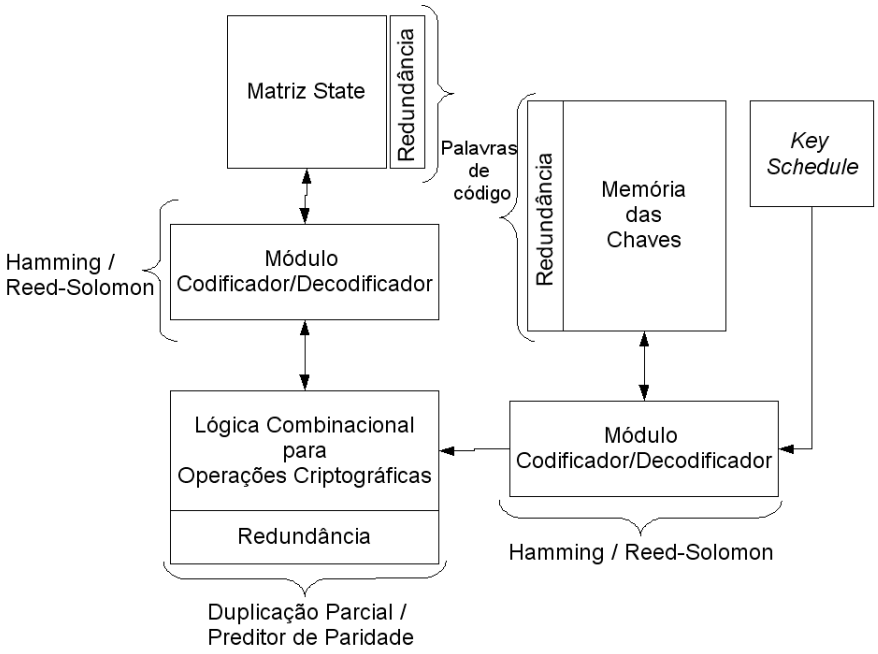


Figura 6.1: Diagrama de blocos do AES protegido.

6.2.1 Proteção das memórias

Os códigos corretores de erros *Hamming* e *Reed-Solomon* foram adotados por suportarem diferentes modelos de falhas, podendo assim, fornecer diferentes níveis de proteção e permitindo uma avaliação mais precisa da relação custo *versus* proteção. Os módulos codificadores/decodificadores foram implementados de forma combinacional para ambos os algoritmos. Tanto *Hamming* quanto *Reed-Solomon* adicionam *bits* de redundância às palavras de dados. Os *bits* de redundância são chamados de *palavras de código*. Assim, os dados codificados e armazenados na memória tem o formato apresentado na Figura 6.2. Os tópicos 6.2.1.1 e 6.2.1.2 apresentam os detalhes de cada código.

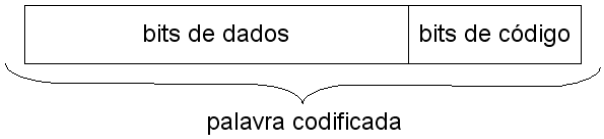


Figura 6.2: Formato dos dados codificados em memória.

6.2.1.1 Código de Hamming

O código de *Hamming* em sua forma normal é capaz de corrigir falhas simples. Sua codificação é baseada em paridade. Por exemplo, suponha que existam 8-bit de dados: $m_1m_2m_3m_4m_5m_6m_7m_8$. Para protegê-lo com *Hamming* adiciona-se 4-bit formando um código de 12-bit. Agora, numerando os *bits* de 1 a 12 como: $x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10}x_{11}x_{12}$, onde: $x_3x_5x_6x_7x_9x_{10}x_{11}x_{12}$ são $m_1m_2m_3m_4m_5m_6m_7m_8$, respectivamente. Os 4-bit adicionais são calculados da seguinte forma:

$$x_1 = x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

$$x_2 = x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11}$$

$$x_4 = x_5 \oplus x_6 \oplus x_7 \oplus x_{12}$$

$$x_8 = x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12}.$$

Se houver erro em um bit apenas, é possível detectar esse erro e corrigi-lo. Para isso fazemos o seguinte cálculo dos 4 bits, denominados k_1, k_2, k_3 e k_4, onde:

$$k_1 = x_1 \oplus x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11}$$

$$k_2 = x_2 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11}$$

$$k_3 = x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_{12}$$

$$k_4 = x_8 \oplus x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12}.$$

Se $k_1 = k_2 = k_3 = k_4 = 0$, então não há erro. Caso contrário, o número codificado pelos 4 bits $k_1k_2k_3k_4$ determina a posição do bit errado. Por exemplo, se $k_1k_2k_3k_4 = 0111$, então o bit errado é x_7.

Hamming em sua forma original possui a limitação de não poder distinguir se houve uma ou duas falhas. Isso é uma limitação grave, visto que, se houve duas ou mais falhas o processo de decodificação apresentará um resultado errado. Para minimizar tal problema, um *bit* de paridade extra é adicionado ao código original, este *bit* será denominado k_5 . K_5 representa a paridade de todos os *bits* de dados e dos *bits* de código juntos e servirá para indicar uma possível multiplicidade de falhas, da seguinte forma:

1. se não houve falhas o teste de paridade para k_5 não indicará erros e $k_1k_2k_3k_4$ será igual a zero, então nada precisa ser feito;

2. se uma falha ocorrer em qualquer ponto da palavra de código ou dados, incluindo k_5, então o teste de paridade para este *bit* indicará que houve um erro enquanto que $k_1k_2k_3k_4$ será diferente de zero, então o *bit* na posição $k_1k_2k_3k_4$ é corrigido;

3. se duas falhas ocorrerem $k_1k_2k_3k_4$ será diferente de zero, porém, a verificação de paridade para k_5 não indicará problemas, então, uma duplicidade de falhas foi detectada e não é possível corrigi-la;

4. se um número ímpar de falhas maior ou igual a três ocorrer, então, $k_1k_2k_3k_4$ será diferente de zero e k_5 indicará ocorrência de falhas, porém, não é possível saber se foi uma ou um número ímpar de falhas, assim, o valor será decodificado com falhas.

Com esta adição ao código de *Hamming* consegue-se um ganho importante na cobertura de falhas, sendo que apenas falhas de multiplicidade ímpar maior ou igual a 3 três são mascaradas, ao custo de apenas um *bit* a mais na palavra de código. Assim, uma palavra de dados de 8-bit, necessita de 4 *bits* adicionais para proteção com *Hamming* puro e 5 bits para a extensão apresentada. Uma palavra de 32-bit necessita no total 8 *bits* para o *Hamming* com extensão.

A descrição VHDL da biblioteca de *Hamming* foi gerada através de um programa gerador desenvolvido no Laboratório de Sistemas Embarcados (LSE) do Instituto de Informática da UFRGS e atualmente disponível em (OPENCORES.ORG, 2006). Tal programa gera a biblioteca para o tamanho de palavra desejado. No caso deste trabalho, os tamanhos de palavras foram 8-bit (*matriz state*) e 32-bit (memória das chaves). A biblioteca gerada dá suporte à extensão do código de *Hamming* descrita acima. Sendo que

quando uma multiplicidade de falhas é detectada um sinal de erro é gerado. A arquitetura desenvolvida neste trabalho usa tal sinal para interromper o processamento impedindo o término da computação com falhas na saída.

6.2.1.2 Código de Reed-Solomon

Reed-Solomon é um código corretor de erros projetado para ser capaz de corrigir múltiplos erros. Tal capacidade é devido à sua arquitetura baseada em blocos de *bits*, denominados *símbolos*. Isso significa que sua unidade básica não são *bits*, mas símbolos formados por conjuntos de *bits*. Uma implementação capaz de corrigir um símbolo por palavra codificada é capaz de corrigir qualquer número de *bits* errados em um único símbolo.

O código *Reed-Solomon* é especificado como $RS(n, k)$ com símbolos de s-bits, onde n é o número total de símbolos em uma palavra codificada e k é o número de símbolos de dados por palavra. O número de símbolos de proteção é igual a $n - k$. Um decodificador *Reed-Solomon* pode corrigir até t símbolos errados, onde $2t = n - k$. Porém, a implementação utilizada neste trabalho é capaz de corrigir apenas um símbolo por palavra codificada. Uma implementação em *hardware* capaz de corrigir mais de um símbolo é demasiadamente custosa (NEUBERGER, 2003).

A mensagem a ser codificada/decodificada é dividida entre diversos símbolos. Se um único erro ocorrer, temos duas incógnitas: a posição do símbolo errado dentro da mensagem, e o *bit* errado dentro do símbolo. Como qualquer problema, para se resolver duas incógnitas, são necessárias duas equações. A geração de tais equações é baseada na teoria dos campos finitos. A primeira regra básica da teoria de campos finitos é que todas as operações são executadas em módulo-2. Isto significa que a maioria das operações são simplesmente transformadas em portas ou-exclusivas (XOR). Além disso, não é necessário tratar números negativos, pois em operações ou-exclusivo $1 + 1 = 0$, então $1 = -1$. A segunda regra básica é que todas as operações são limitadas pelo campo finito em que estamos operando. Como resultado, qualquer operação que se deseja realizar nos dados produz um resultado que é contido em um conjunto finito de números. Tais características facilitam sua implementação em *hardware*.

Para ilustrar como ocorre a codificação de uma palavra de dados, em uma implementação que suporta a correção de apenas um símbolo , chamaremos de $A_0, A_1, A_2, ...A_n$ os n símbolos de dados. São adicionados dois símbolos de código, aqui denominados R e S.É necessário determinar R e S através do seguinte sistema de equações:

$$\begin{cases} A_0 \oplus A_1 \oplus ... \oplus A_n \oplus R \oplus S = 0 \\ \alpha^1 A_0 \oplus \alpha^2 A_1 \oplus ... \oplus \alpha^{n+1} A_n \oplus \alpha^{n+2} R \oplus \alpha^{n+3} S = 0. \end{cases}$$

Onde, α é um polinômio primitivo usado no cálculo. Isolando S e substituindo na segunda equação temos:

$$\begin{cases} S = A_0 \oplus A_1 \oplus ... \oplus A_n \oplus R \\ \alpha^1 A_0 \oplus \alpha^2 A_1 \oplus ... \oplus \alpha^{n+1} A_n \oplus \alpha^{n+2} R \oplus \alpha^{n+3}(A_0 \oplus A_1 \oplus ... \oplus A_n \oplus R). \end{cases}$$

Simplificando a segunda equação, obtêm-se:

$$R = \alpha^1 A_0 \oplus \alpha^{19}.A_1 ... \oplus \alpha^x A_n.$$

Finalmente, substituindo novamente na primeira equação:

$$S = \alpha^{18} A_0 \oplus \alpha^{11} A_1 \oplus ... \oplus \alpha^x A_n.$$

Assim, obtem-se R e S em termos de $A_0, A_1, A_2 ... A_n$. A palavra codificada pode então ser armazenada no formato $A_0, A_1, A_2 ... A_n R, S$. O processo de decodificação (correção) é realizado, primeiramente, calculando-se duas síndromes S_0 e S_1, da seguinte forma:

$$S_0 = A_0 \oplus A_1 \oplus ... \oplus A_n \oplus R \oplus S,$$
$$S_1 = \alpha^1 A_0 \oplus \alpha^2 A_1 \oplus ... \oplus \alpha^{n+1} A_n \oplus \alpha^{n+2} R \oplus \alpha^{n+3} S.$$

Se não existirem erros, S_0 e S_1 serão iguais a zero. Agora, supondo que um erro ϵ ocorreu em A_1. O valor de A_1 passa a ser $A_1 \oplus \epsilon$. Assim, as síndromes passam a ser:

$$S_0 = A_0 \oplus (A_1 \oplus \epsilon) \oplus ... \oplus A_n \oplus R \oplus S$$
$$= (A_0 \oplus A_1 \oplus ... \oplus A_n \oplus R \oplus S) \oplus \epsilon$$
$$S_1 = \alpha^1 A_0 \oplus (\alpha^2 A_1 \oplus \epsilon) \oplus ... \oplus \alpha^{n+1} A_n \oplus \alpha^{n+2} R \oplus \alpha^{n+3} S$$
$$= (\alpha^1 A_0 \oplus \alpha^2 A_1 \oplus ... \oplus \alpha^{n+1} A_n \oplus \alpha^{n+2} R \oplus \alpha^{n+3} S) \oplus \alpha^2 \epsilon.$$

As expressões entre parênteses são iguais a zero. Desta forma, S_0 fica com o valor ϵ e S_1 com $\alpha^2 \epsilon$. Ou seja, S_0 representa a falha, enquanto que S_1 é usado para determinar a localização do símbolo com falha, da seguinte forma:

$$k = S_1 / S_0 = \alpha^2 \epsilon / \epsilon = \alpha^2.$$

Onde k é a localização do símbolo. Finalmente, para corrigir o erro, basta executar um ou-exclusivo entre o símbolo indicado por k e S_0.

Semelhante ao código de *Hamming*, *Reed-Solomon* em sua versão pura realiza apenas correção. Neste caso, múltiplas falhas em diferentes símbolos causam erros na decodificação que não são sinalizados. Para aumentar a cobertura de falhas e permitir a sinalização de falhas não corrigíveis foi adicionado uma extensão à implementação original. Tal extensão consiste na adição de uma nova verificação no final da etapa de decodificação descrita acima. Foi visto que, se não existem falhas na palavra codificada:

$$A_0 \oplus A_1 \oplus ... \oplus A_n \oplus R \oplus S = 0.$$

Porém, tal equivalência não é válida caso ocorram falhas em algum dos elementos da equação (A_i, ReS). A verificação extra atua após a correção das possíveis falhas pela aplicação de S_0 na posição k. Então é feito um ou-exclusivo entre os símbolos de dados decodificados, ou seja, após a aplicação de S_0 e k e os símbolos R e S da palavra codificada. Se a equivalência acima não se confirmar, então, houve a ocorrência de múltiplas falhas em diferentes símbolos. Com isso, um sinal de erro é gerado e usado para interromper a execução do *core* sem exibir nenhum resultado, pois uma situação de falhas não recuperável foi detectada. Ainda assim, se duas ou mais falhas ocorrem em posições específicas de diferentes símbolos, mesmo após a decodificação, pode ocorrer sobreposição das falhas e conseqüentemente mascaramento das mesmas.

Assim como a biblioteca usada para codificação/decodificação de *hamming*, a descrição VHDL da biblioteca de *Reed-Solomon* foi gerada através de um programa gerador. Tal programa foi desenvolvido como trabalho de conclusão de curso em (NEUBERGER,

2003). O programa toma como entrada o tamanho da palavra de dados a ser protegida e o número de símbolos por palavra de código desejado. Com estes dados o programa sugere possíveis polinômios primitivos para serem usados na implementação da biblioteca. Após a seleção do polinômio desejado o programa fornece como saída a biblioteca. Porém, o programa gerador não implementa a extensão para detecção de falhas não corrigíveis descrita acima. Assim, as bibliotecas geradas para proteger palavras de 8 e 32-*bit* foram modificadas para incluir tal extensão.

Uma última característica da biblioteca de *Reed-Solomon* vale ser mencionada. Tal característica influenciará decisivamente nos resultados, conforme será apresentado na Subseção 6.3.2. A aplicação dos elementos S_0 e k corrige falhas apenas no elemento A_i, deixando os símbolos R e S desprotegidos. Assim, mesmo que uma falha simples atinja um dos símbolos R ou S, a extensão implementada indicará uma falha não recuperável causando o interrupção do processamento. De fato, a falha é não recuperável já que a implementação não permite a correção dos símbolos R e S, porém, a palavra é decodificada sem erros. Para a proteção da palavra de 8-*bit*, a mesma é estendida para 9-*bit* e dividida em 3 símbolos de 3-*bit*. Outros dois símbolos (R e S) de 3-*bit* cada são adicionados. Para a proteção da palavra de 32-*bit*, a mesma é dividida em 8 símbolos de 4-*bit* cada, sendo adicionados os símbolos R e S também com 4-*bit*. Desta forma, o número de *bits* de código é proporcionalmente menor para a proteção da palavra de 32-*bit*. Assim, para uma distribuição de falhas com probabilidade uniforme, o número de interrupções será menor para a palavra de 32-*bit*, havendo um ganho de eficiência na proteção.

6.2.2 Proteção da lógica combinacional

Conforme dito anteriormente, formam utilizados, independentemente, dois esquemas de proteção para a lógica combinacional. O primeiro, denominado duplicação parcial, é semelhante ao que foi aplicado ao DES no Capítulo 5. A segunda forma de proteção foi primeiramente descrita por (BERTONI et al., 2002) e implica em um *overhead* bastante inferior ao da duplicação parcial. As duas opções possuem níveis de detecção e correção bastante diferentes, enquanto o método de (BERTONI et al., 2002) implica na detecção de falhas simples, a duplicação parcial pode detectar múltiplas falhas desde que elas não ocorram de forma exatamente igual na parte duplicada. O tópicos 6.2.2.1 e 6.2.2.2 detalham os dois métodos usados.

6.2.2.1 Duplicação Parcial

A Figura 6.3 apresenta um diagrama de blocos da arquitetura proposta com proteção da lógica combinacional por duplicação parcial. A duplicação parcial do AES consistiu em duplicar as quatro operações criptográficas que, conforme explicado na Seção 6.1, são implementadas como módulos combinacionais. A arquitetura proposta permite não apenas duplicar, mas adicionar quantos módulos redundantes forem desejados. Para as finalidades deste trabalho, foi utilizada apenas duplicação. Assim, as operações são recomputadas em paralelo, e ao final de cada operação, os resultados de ambos os módulos são comparados. Resultados diferentes implicam que alguma falha ocorreu durante a operação. Assim, os dados são buscados novamente na memória protegida e a operação é refeita.

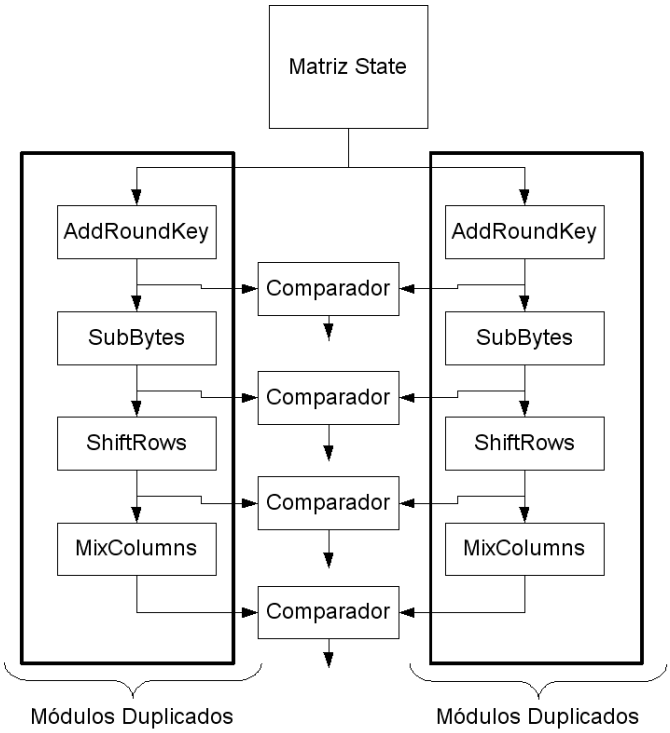


Figura 6.3: Diagrama de blocos para o esquema de proteção da lógica combinacional por duplicação parcial.

6.2.2.2 Predição de Paridade

O esquema de paridade de (BERTONI et al., 2002) propõe associar cada *byte* da *matriz state* a um *bit* de paridade, em um total de 16 *bits* de paridade. No caso desta implementação, cada elemento da *matriz state* foi expandido para permitir o armazenamento do *bit* de paridade. Este esquema permite detectar todas as falhas simples em diferentes *bytes* simultaneamente e múltiplas falhas ocorridas em números ímpares sobre o mesmo *byte*. Porém, devido às operações criptográficas do AES:

- Para cada operação criptográfica é necessário um método para predição de paridade;
- É necessário determinar pontos para verificação da validade da paridade.

Desta forma, para cada operação foi desenvolvido um método de predição, conforme descrito abaixo:

- *SubBytes*: A Sbox foi implementada como uma memória de 256x9 *bits*, onde o nono *bit* é o *bit* de paridade previamente calculado. Assim, o *bit* de paridade é substituído juntamente com o *byte*.
- *ShiftRows*: A predição para esta operação é bastante simples já que os *bits* de paridade estão armazenados juntamente com os seus respectivos *bytes*. Assim, são rotacionados automaticamente, sem a necessidade de lógica adicional.
- *MixColumns*: A predição para a operação *MixColumns* é a mais complexa matematicamente. Porém o conjunto final de equações é bastante simples e podem ser resolvidas facilmente com o uso de operações ou-exclusivo. A predição para os *bits*

de paridade de cada coluna é dada por:

$$p_{0,c} = p_{0,c} \oplus p_{2,c} \oplus p_{3,c} \oplus d^7_{0,c} \oplus d^7_{1,c}$$

$$p_{1,c} = p_{0,c} \oplus p_{1,c} \oplus p_{3,c} \oplus d^7_{1,c} \oplus d^7_{2,c}$$

$$p_{2,c} = p_{0,c} \oplus p_{1,c} \oplus p_{2,c} \oplus d^7_{2,c} \oplus d^7_{3,c}$$

$$p_{3,c} = p_{1,c} \oplus p_{2,c} \oplus p_{3,c} \oplus d^7_{3,c} \oplus d^7_{0,c}.$$

Onde, $p_{x,y}$ denota o *bit* de paridade na posição xy da *matrix state* e $d^7_{x,y}$é o *bit* mais significativo do *byte* na posição xy.

- *AddRoundKey*: A predição para esta operação consiste em realizar uma operação ou-exclusivo entre os *bit* de paridade da *matriz state* e os da chave. Para isso, os *bits* de paridade para cada *byte* da chave devem ser previamente calculados.

A Figura 6.4 apresenta um diagrama de blocos para o esquema de paridade implementado. A checagem da validade da paridade para cada operação é realizada ao final da mesma. Assim, possíveis falhas são detectadas o mais cedo possível e a correção é realizada antes que o processamento prossiga. A correção consiste em buscar a linha ou coluna afetada na memória e realizar novamente a operação. Desta forma, a operação será repetida tantas vezes quanto forem necessárias. Se falhas atingirem a lógica combinacional seguidamente, haverá uma redução no desempenho.

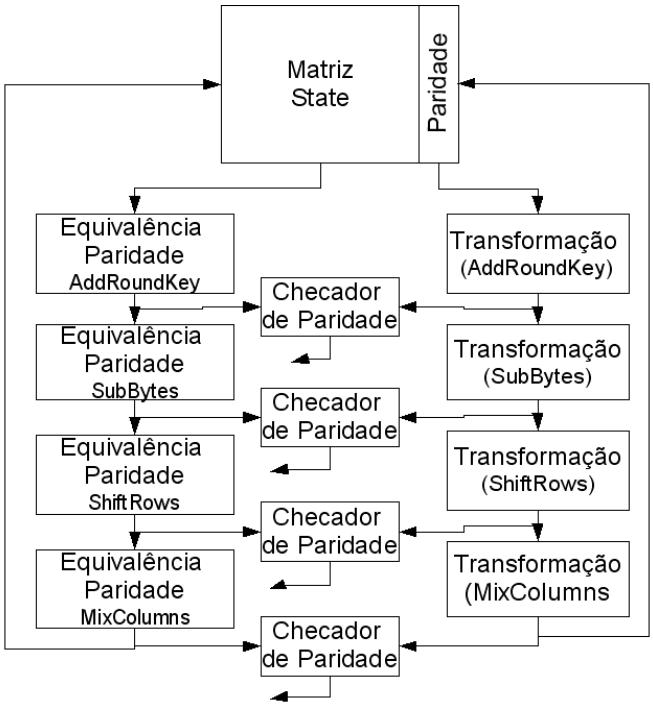


Figura 6.4: Diagrama de blocos para o esquema de proteção da lógica combinacional por paridade.

6.3 Validação do Sistema

Para validação do sistema de tolerância a falhas do AES foram realizados experimentos de injeção de falhas simples e múltiplas. Como plataforma de simulação utilizou-se o ModelSim XE 6.1e da Xilinxs. Como dito anteriormente, a linguagem de descrição de *hardware* utilizada foi VHDL. VHDL não permite observar ou manipular sinais em diferentes níveis da hierarquia sem uma rota até o sinal desejado. Assim, da forma convencional é necessário modificar a descrição original para acomodar mecanismos de observação e manipulação de sinais. Isso adiciona complexidade extra à arquitetura que se deseja simular, além de influenciar na área e nos atrasos. O ModelSim implementa uma biblioteca de funções que permitem simulações de falhas sem a necessidade de alterar a descrição original. Tal biblioteca foi usada para facilitar a configuração do ambiente de injeção de falhas. Este ambiente inclui o próprio ModelSim, como *software* de simulação, o *testebench* que é o arquivo que descreve os estímulos e define os parâmetros de injeção de falhas e, por fim, a descrição da arquitetura proposta. A Subseção 6.3.1 descreve o funcionamento do *testbench*, enquanto que a Subseção 6.3.2 exibe os resultados das simulações.

6.3.1 Configuração do experimento

O ModelSim implementa mecanismos para auxiliar na injeção de falhas em simulações de arquiteturas descritas em VHDL ou Verilog. O mecanismo de injeção de falhas implementado pelo ModelSim foi criado para facilitar injeção de falhas, tornando desnecessária a modificação da arquitetura. Tal mecanismo de injeção de falhas é denominado *Signal Spy*. O *Signal Spy* possui um conjunto de funções que permitem observar e manipular sinais em diferentes níveis da hierarquia sem a necessidade de uma rota até o sinal visado. Isso simplifica as simulações de falhas, pois é necessário apenas escrever um *testbench* que suporte tal recurso. O *Signal Spy* apresenta duas desvantagens. A primeira desvantagem é a não portabilidade, pois este não é suportado por outros simuladores. A segunda, é que os mecanismos de injeção de falhas não podem ser prototipados por fazerem parte apenas do *testbench* e por não serem sintetizáveis. Para este trabalho, tais desvantagens não são significativas, pois, neste caso, o *Signal Spy* foi aplicado apenas ao *testbench*, onde portabilidade não é um fator determinante. Além disso, uma quantidade significativa de simulações puderam ser realizadas pelo ModelSim em tempo hábil, sem a necessidade de prototipação em FPGA's. Outro fator determinante para a escolha da simulação em vez da prototipação é que um experimento real de injeção de falhas mostrou-se impraticável para esta situação. Tal experimento envolveria a prototipação da arquitetura proposta em tecnologia ASIC, o que envolveria custos significativos e tempo de projeto não suportado pelo cronograma de trabalho. Ainda, com os ASICs em mãos, seriam necessários equipamentos bastante específicos para a injeção de falhas, os quais não são disponibilizados pela universidade. Porém, simulação é uma parte do ciclo de projeto, utilizada na validação, e que culmina na prototipação. Desta forma, o primeiro passo para a implementação física da arquitetura proposta foi dado com a simulação, e esta garantiu sua validação funcional.

O *Signal Spy* oferece um conjunto de funções para injeção de falhas. Para este trabalho, foi utilizado um subconjunto destas funções. Tais funções podem ser vistas na Tabela 6.1. O restante das funções podem ser encontradas no manual do usuário do ModelSim (MODELSIM XILINX - USER'S MANUAL, VERSION 6.1E, 2006). Conforme a Tabela 6.1, a função *init_signal_spy()* associa um sinal de qualquer nível em uma arquitetura

Função do *Signal Spy*	**Utilidade**
init_signal_spy()	Associa um sinal da hierarquia a outro existente no *testbench*
signal_force()	Altera o valor do sinal especificado
signal_release()	Desfaz a associação feita pela função *ini_signal_spy()*

Tabela 6.1: Subconjunto de funções fornecidas pelo *Signal Spy*.

VHDL a um sinal existente no nível hierárquico desejado, neste caso, no *testbench*. Todas as modificações ocorridas no sinal original são refletidas no sinal associado. Isso é útil para monitoração do comportamento do sinal sob observação. A função *signal_force()* força um determinado valor ao sinal especificado. É esta função que faz a injeção de falha propriamente dita. Tal função tem uma série de parâmetros que permitem especificar o seu comportamento. Abaixo a função com seus parametros:

$$signal\_force(dest\_object, value, rel\_time, force\_type, cancel\_period, verbose)$$

Onde, *dest_object* é o sinal alvo. *Value* é o valor que será atribuído ao sinal alvo. *Real_time* é o momento em que a função atuará e é relativo ao início da simulação. *Force_type* permite mudar o tipo de persistência do valor atribuído. Por exemplo, um valor aceitável para *force_type* é *freeze*. *Freeze* determina que o valor atribuído permanecerá inalterado, pelo tempo determinado no parâmetro *cancel_period*. Isso é útil para gerar pulsos transientes em lógica combinacional, onde a largura do pulso é determinado por *cancel_period*. Outro valor possível é *deposit*. *Deposit* determina que o valor atribuído permanecerá apenas até a próxima escrita, até uma ocorrência de um novo comando *force_signal* ou até o tempo determinado em *cancel_period*. Isso é muito útil para simulação de injeção de falhas transientes em memórias, onde um *bit-flip* persiste até que uma nova escrita altere o seu valor. O parâmetro *verbose* permite exibir o resultado da chamada da função *singal_force* na tela. Por fim, a função *signal_release()* desfaz a associação entre sinais criada pela função *init_signal_spy()*.

Usando tais funcionalidades foi possível descrever um *testbench* capaz de realizar simulações de falhas para as diferentes partes do *datapath* da arquitetura proposta para o AES. Tais partes incluem: lógica combinacional, memória das chaves e a *matriz state*. O *testbench* foi descrito para automatizar o processo de geração de falhas, sendo que este permite a simulação de um número pré-determinado de execuções do *core*. Neste ponto, subentende-se por execução o momento, a partir do qual, o *core* lê os dados de entrada (bloco de dados e chave criptográfica) até a geração da saída (bloco criptografado). Uma execução pode ser abortada antes do término da criptografia de um bloco. Isto ocorre com a detecção de falhas que não podem ser recuperadas (somente para as simulações com esquema de proteção). Uma simulação é composta por diversas execuções e, durante cada execução, uma ou mais falhas podem ser injetadas. Ao final da simulação um arquivo de resultados é gerado. Além disso, o *testbench* possui uma série de parâmetros para adequação da simulação de injeção de falhas. A Tabela 6.2 resume tais parâmetros. Para validação do sistema foram realizadas 2000 execuções para injeção de falhas simples e 10000 execuções para múltiplas falhas, para cada parte do *core*. O número de falhas foi de apenas uma para a simulação de injeção de falhas simples e de 2 a 10 falhas para a simulação de falhas múltiplas. Neste segundo caso, um número entre 2 e 10 foi escolhido aleatoriamente para determinar o número de falhas a serem inseridas, conforme será visto

Parâmetros	Valores
Número de Simulações	10.000
Número de falhas por simulação	1 a 10
Intervalo para injeção da falha	0 a 1890087.0 ns
Duração da falha	50 a 600 ps

Tabela 6.2: Parâmetros para adequação das simulações.

a seguir. O momento da injeção das falhas fica no tempo entre 0 ns e 1890087.0 ns, que é o tempo que o *core* do AES demora para efetuar as 10 iterações criptográficas. Assim a falhas ocorrem em algum ponto do processo criptográfico entre a iteração 1 e 10, sendo que o momento exato da injeção é escolhido aleatoriamente. A duração da falha é válida apenas para injeção de falhas em lógica combinacional, sendo que este parâmetro determina a duração do pulso transitório, o qual foi determinado com base em (GADLAGE; AL., 2004).

Para simulação de falhas em memória, este parâmetro é desconsiderado.

Conforme explicado na Seção 6.2, os mecanismos de proteção para as diferentes partes (*matriz state*, memórias das chaves e partes combinacionais) operam independentemente uns dos outros. Assim, as simulações de injeção de falhas puderam ser realizadas separadamente. Apesar das funcionalidades do *Signal Spy* ajudarem, em muito, torna-se complexa a descrição de um único *testbench* que englobe todas as funcionalidades desejadas. Existem diferenças significativas entre as injeções de falhas nas memórias e na parte combinacional. A principal diferença está no modelo de falha gerado para lógica combinacional e memórias. Em lógica combinacional, o *testbench* precisa gerar um pulso transitório com uma determinada duração (observe a Tabela 6.2), enquanto que, para as memórias, a falha gerada é uma inversão de *bit* (*bit-flip*) que dura até a próxima escrita naquela posição. Além disso, para cada uma das partes foram realizadas três diferentes simulações. Uma simulação para o circuito sem proteção e outras duas com os esquemas de proteção propostos. Assim, no final houve um total de 9 simulações (três simulações para cada uma das três partes) para injeção de falhas simples e outras 9 simulações para injeção de múltiplas falhas.

A Figura 6.5 mostra a estrutura genérica do *testbench* desenvolvido, a qual é válida para as diferentes simulações. O *testbench* é composto por três processos. O primeiro processo é responsável pela reinicialização do *core*. A reinicialização ocorre no início da primeira execução, e depois, ao final de cada execução. O segundo processo, daqui para frente chamado de PGSC (Processo de Geração de Sinais de Controle), gera os *sinais de controle* do *core*. A Tabela 6.3 resume tais sinais e adiciona os *sinais de dados*, juntamente com suas funcionalidades. Os sinais de controle são gerados conforme o *core* avança nas etapas criptográficas, enquanto que os sinais de dados são mantidos constantes. Manter as mesmas entradas (bloco de entrada e chave criptográfica) para todas as execuções é importante para fins de comparação dos resultados. O terceiro processo, conhecido como PIF (Processo para injeção de falhas) realiza a injeção de falhas, propriamente dita. É este processo que manipula as funções do *Signal Spy* mostradas na Tabela 6.1 e os parâmetros vistos na Tabela 6.2. Outra função deste processo é gerar o um arquivo com o histórico da simulação. Este arquivo contém, para cada execução, as seguintes informações:

- Número da execução;

	Sinal	Funcionalidade
Sinais de Controle	rst	Sinal de reinicialização.
	clk	*Clock* do sistema.
	key_start	Inicia a geração das subchaves.
	key_ready	Indica o término da geração das subchaves.
	data_start	Inicia o processamento do bloco de entrada.
	data_ready	Indica o término da criptografia do bloco de entrada. A saída é exibida em data_out.
	fail	Indica que a execução foi abortada. Nenhum resultado é exibido.
Sinais de Dados	key_in	Chave criptográfica.
	data_in	Bloco de entrada.
	data_out	Bloco criptografado.

Tabela 6.3: Sinais de controle e dados gerados pelo processo PGSC.

- Momento da injeção da falha, seguido do sinal e do *bit* do sinal afetado. Em caso de múltiplas falhas, os dados das diversas falhas são distribuídos nas linhas subseqüentes.
- Resultado da execução representado por uma das 3 letras: S (resultado com falhas), F (execução suspensa), N (execução finalizada com êxito).
- Saída do *core* (sinal *data_out*). Em caso de suspensão da operação o valor em *data_out* é 0.

Ainda, ao final de todas as execuções uma contagem com o número de execuções que apresentaram falhas e o número de execuções interrompidas é salva no final do arquivo.

Os três processos são sincronizados, isto é, quando o processo PIF termina ele aguarda a finalização do processo PGSC. Caso o processo PGSC atinja o seu término primeiro, a geração de falhas deve terminar, já que a execução também terminou, então o processo PIF finaliza suas operações enquanto o processo PGSC aguarda. A sinalização entre os processos é feita através de uma variável de controle. Esta variável possui dois *bits* de sinalização. O *bit* na posição 0 pertence ao PIF, enquanto que o *bit* na posição 1 pertence ao processo PGSC. Assim, quando um processo terminar ele deve escrever o valor 1 em seu respectivo *bit*. Portanto, ambos os processo aguardam pelo valor '11' na variável de controle. Quando o processo responsável pela reinicialização do *core* detecta o valor '11' na variável de controle, ele determina a reinicialização do *core*. Terminada a reinicialização, uma nova execução é iniciada. Esse processo é executado repetidamente até o número de execuções determinado ser atingido.

Os parâmetros determinados aleatoriamente são gerados com ajuda de uma biblioteca de geração de números pseudo-aleatórios disponibilizada em (OPENCORES.ORG, 2006). Esta biblioteca é exclusiva para uso em *testbenches* por não ser sintetizável. Ela pode gerar números sob três diferentes distribuições númericas: uniforme, gaussiana e exponential. Para este experimento a distribuição escolhida foi uniforme. Assim, foi considerado que qualquer um dos sinais e seus respectivos *bits* tem a mesma probabilidade de serem sorteados. O uso desta biblioteca é bastante simples e, entre as funções disponíveis, utilizou-se duas delas: *init_uniform()* e *rand()*. A função *init_uniform()* é responsável por inicializar a geração de números com distribuição uniforme e usa três parâmetros:

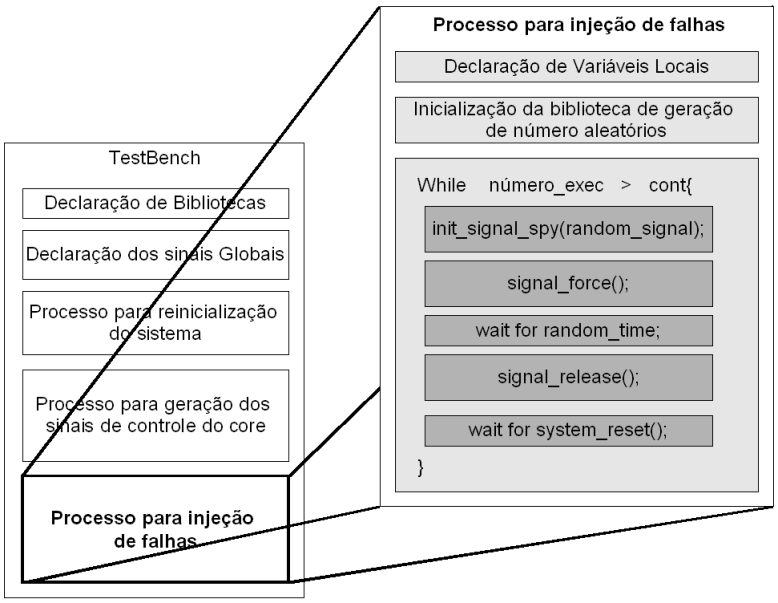


Figura 6.5: Detalhes da estrutura do *testbench* com ênfase no processo de injeção de falhas.

semente, *valor inicial*, *valor final*. Por se tratar de seqüências pseudo-aleatórias, sempre que uma mesma semente (conjunto de três números inteiros) for utilizada, a mesma seqüência pseudo-aleatória é gerada. Diferentes seqüências pseudo-aleatórias podem ser geradas com diferentes sementes. A geração de seqüências totalmente aleatórias é um processo demasiadamente complexo. Seqüências pseudo-aleatórias são usadas por serem facilmente geradas, e neste caso, a facilidade de se repetir uma seqüência é uma vantagem. Todas as simulações utilizaram as mesmas sementes para inicializar a seqüência pseudo-aleatória. Assim, parâmetros como número de falhas em cada execução, momento da injeção de falha e duração da falha são repetidos para as simulações com proteção e sem proteção. Isso ajuda a tornar a comparação justa. Outra vantagem é que os experimentos podem ser facilmente reproduzidos para fins de confirmação. A função *rand()* é responsável por obter o próximo número da seqüência pseudo-aleatória.

6.3.2 Resultados Experimentais

Utilizando o *testbench* descrito na subseção anterior foram realizadas um total de 18 simulações para injeção de falhas simples e múltiplas. Conforme mencionado nas seções anteriores, a injeção de falhas foi divida nas diferentes partes do *datapath* do AES, sendo importante relembrar que as partes citadas são: *matriz state*, memórias das chaves e lógica combinacional. Essa divisão para injeção de falhas não trás perda na qualidade dos resultados gerados, já que os mecanismos de injeção de falhas operam independentemente uns dos outros. Outro fator importante que permite a injeção de falhas separadamente é a execução seqüencial do *datapath*, isto é, não existem execuções paralelas no *core* sem proteção. Por outro lado, o *core* com proteção exibe áreas com processamento paralelo, que é o caso nos dois esquemas de proteção da lógica combinacional. Porém, nestes casos é feita injeção de falhas tanto para o *datapath* original como para o novo *hardware* operando em paralelo.

Os resultados para a injeção de falhas simples são apresentados na Tabela 6.4. A

primeira coluna da tabela indica a *localização* das falhas injetadas, ou seja, *matriz state*, memória das chaves ou lógica combinacional. A segunda coluna (*Esq. Prot.*) é o esquema de proteção utilizado. Para as memórias (*matriz state e memórias das chaves*) os esquemas podem ser *Hamming* ou *Reed-Solomon*, para a lógica combinacional os esquemas podem ser predição de paridade ou duplicação. Para ambos, são apresentados os resultados para injeção de falhas sem proteção. A terceira coluna é o número de execuções, que neste caso foram 2.000 para cada uma das simulações. A quarta coluna (Falhas), é o número de vezes que a falha injetada afetou a saída. A quinta coluna, denominada *Int*, é o número de vezes que a execução foi interrompida. Isso é válido somente para os esquemas de proteção de memória. Além disso, a versão sem proteção não possui mecanismos para interromper a execução na ocorrência de falhas. A sexta coluna, chamada de *Corr* é a porcentagem de vezes que a saída foi corrompida com base em 2.000 execuções. Por fim, a coluna *Susp* é a porcentagem de vezes que a execução foi interrompida sob a ocorrência de falhas, com base em 2.000 execuções.

A vulnerabilidade do *core* sem proteção é bastante perceptível. O resultado da simulação mostra que 48,75% das execuções resultaram em respostas erradas no caso da *matriz state*. De fato, as memórias são a parte mais vulnerável do *core*. Um *bit-flip* persiste em uma memória até que uma nova escrita altere novamente seu valor. Durante a permanência da falha, se alguma leitura ocorrer, todo o processamento estará comprometido devido à leitura de um valor incorreto. O *bit* comprometido pela falha influenciará na computação e irá gerar novos *bits* errados, que serão novamente gravados na memória. Depois de algumas iterações criptográficas torna-se impossível recuperar o montante de falhas acumuladas e, ao final, todo o resultado estará inutilizado. Desta forma, é imprescindível a detecção e correção da falha o mais cedo possível. Continuando a análise da Tabela 6.4, código de Hamming conseguiu corrigir 100% das falhas simples. Com isso, a ocorrência de falhas para o usuário foi transparente, ou seja, ele não percebeu que falhas incidiram sobre o dispositivo criptográfico. Do ponto de vista de um atacante, ele não saberia distinguir se o seu método de injeção de falhas está sendo efetivo e as falhas foram mascaradas pelo esquema de tolerância a falhas ou, se simplesmente o seu método de injeção não conseguiu gerar falhas no *hardware*. Para o código corretor de erros Reed-Solomon, o cenário mostrou-se diferente. Apesar de não permitir que falhas simples afetassem a saída, ele não conseguiu recuperar todas as falhas e, em alguns casos (19,7%) a execução foi suspença. A suspensão da operação aconteceu devido a detalhes específicos da implementação da biblioteca do Reed-Solomon utilizada, conforme foi melhor detalhado na Subseção 6.2.1. Para o usuário, tal suspensão na operação implica em refazer a transação não finalizada ou simplismente um atraso no término da transação, dependendo da forma que o sistema externo ao *core* criptográfico trata tais sinalizações de erros. Do ponto de vista do atacante, isso representa uma vulnerabilidade importante, já que isto dá certeza que falhas estão atingindo o dispositivo. O próximo passo do atacante seria explorar um modelo de falhas que o esquema de tolerância a falhas não suporte e que permita a manifestação de falhas não detectáveis.

Para a injeção de falhas simples na memória das chaves o cenário é bastante semelhante. Porém, a principal diferença está na vulnerabilidade desta memória. A memória das chaves mostrou-se muito mais vulnerável a falhas transientes. Conforme visto na Tabela 6.4, 94,5% das falhas incidentes sobre a memória das chaves sem proteção resultaram em falhas no final da execução. Tal vulnerabilidade é devida ao fato de que a escrita na memória das chaves ocorre apenas durante a inicialização do *core*, quando as chaves são geradas e armazendas na memória. Assim, as falhas persistem até o final da compu-

tação, já que não existirão novas escritas nesta memória até o término da execução. Esta vulnerabilidade às falhas transientes, torna a memória das chaves um alvo interessante para um atacante. O código de Hamming teve o comportamente esperado, conseguindo corrigir 100% das falhas simples. Novamente, o código Reed-Solomon não conseguiu corrigir todas as falhas simples e causou a interrupção do processamento em 17,6% das vezes. Neste ponto, é importante deixar claro uma segunda diferença entre os resultados da simulação para a *matriz state* e a memória das chaves. Esta diferença está no percentual de interrupções do processamento para o código Reed-Solomon nas duas memórias. Observe que, para 975 falhas na *matriz state* que corromperam a saída, a computação foi interrompida 394 vezes. Por outro lado, das 1890 vezes que as falhas na memória das chaves se manisfestaram na saída, o *core* foi interrompido apenas 352 vezes. Isso indica uma maior eficiência do Reed-Solomon para proteção de palavras de dados maiores, que é o caso da memória das chaves (ver Subseção 6.2.1). Apesar da maior eficiência na proteção da memória das chaves, Reed-Solomon ainda perdeu para Hamming.

A parte combinacional do *core* mostrou ser a menos vulnerável a falhas. Sendo que, nas 2000 execuções para o *core* sem proteção apenas 0,9% delas manisfestaram falhas na saída. Isso ocorre porque nem todas as falhas chegam a ter duração suficiente para serem registradas, ou seja, as falhas podem ocorrer em um curto intervalo de tempo entre a borda de subida e a borda de descida do *clock*. Se os registradores atuarem apenas na borda de subida, então todos os pulsos transientes ocorridos antes disso serão mascarados. Além disso, mesmo que um pulso transitório esteja presente na entrada do registrador na borda de subida do *clock*, tal pulso não será registrado a menos que o registrador esteja habilitado para registrar suas entradas naquele memomento. Ambos os esquemas de proteção (predição de paridade e duplicação) conseguiram detectar 100% das falhas.

Localização	**Esq. Prot.**	**Exec.**	**Falhas**	**Int.**	**Corr. (%)**	**Susp. (%)**
Matriz State	nenhum	2000	975	-	48.75	-
	Hamming		0	0	0	0
	Reed-Solomon		0	394	0	19.7
Memória das chaves	nenhum	2000	1890	-	94.5	-
	Hamming		0	0	0	0
	Reed-Solomon		0	352	0	17.6
Lógica Comb.	nenhum	2000	18	-	0.9	-
	Duplicação		0	0	0	0
	Predição de Paridade		0	0	0	0

Tabela 6.4: Resultados para injeção de falhas simples.

As simulações para injeção de múltiplas falhas foram realizadas com o intuito de determinar a robustez dos esquemas de proteção em um cenário que ilustra um possível ataque. A injeção de falhas simples determinou que em muitas situações o *core* irá mascarar as falhas sem apresentar efeitos colaterais. Assim, um atacante precisará injetar múltiplas falhas simultaneamente afim de determinar uma vulnerabilidade nas arquiteturas de códigos usadas. Para as memórias foram realizadas 10000 execuções, dando assim, uma maior precisão estatística.

A Tabela 6.5 mostra que, para o *core* sem proteção, a *matriz state* é quase tão vulnerável quanto a memória das chaves. Em 94,98% das execuções, para a *matriz state*, as falhas resultaram em erros de computação, enquanto que, 99,88% das execuções resultaram em erros de computação para a memória das chaves. Isso com base em 10000 execuções. A

maior vulnerabilidade da *matriz state* a múltiplas falhas era esperada, já que arquitetura usada (32-bit), lê e escreve palavras de 32-bit na memória. Assim, mesmo que uma falha seja mascarada por uma escrita, outras falhas incidentes em outras regiões da memória irão permancer e possivelmente serão lidas nas próximas iterações.

O código de *Hamming* apresentou um ótimo desempenho para a proteção da *matriz state*. Das 10000 execuções, apenas uma vez as falhas injetadas levaram a erros de computação, e 0,026% das vezes houve interrupção no processamento. Para o *Reed-Solomon* houve 6 erros de computação, porém, 64,23% das vezes a computação foi interrompida. Novamente, *Hamming* apresentou uma maior eficiência. Para a proteção da memória das chaves, o cenário é semelhante. *Hamming* apresentou falhas de computação 0,09% e interrupção no processamento 7,55% das vezes. *Reed-Solomon* apresentou falhas em 2,67% e interrupção em 43,44% das vezes. A memória das chaves é composta por 10 linhas de 32-*bit*, enquanto que a *matriz state* possui 16 palavras de 8-*bit*(versão sem proteção). Para uma distribuição de falhas uniforme, as chances de ocorrência de uma falha em uma determinada célula de memória da *matriz state* é de $\frac{1}{16}$, pois existem 16 células. A probabilidade de ocorrência de duas falhas na mesma célula é de $\frac{1}{256}$. Enquanto que, a probabilidade de ocorrência de uma falha em uma determinada célula da memória das chaves é de $\frac{1}{10}$, e de duas falhas $\frac{1}{100}$. Desta forma, existem mais chances de múltiplas falhas atingirem uma mesma célula de memória das chaves do que da *matriz state*. Conforme os resultados das simulações de falhas simples, o *core* protegido consegue lidar com 100% das falhas, seja por mascaramento ou interrupção do processamento. Então, é justamente a multiplicidade de falhas em uma mesma célula de memória que leva a erros na computação dos dados, no caso das memórias. Assim, para as duas memórias protegidas, a memória das chaves ainda é mais vulnerável.

A parte combinacional, mesmo com a injeção de múltiplas falhas, ainda mostrou-se ser a mais resistente. Para 2000 execuções simuladas, apenas 90 vezes manifestou-se erros de computação na saída. O esquema de duplicação conseguiu lidar com todos estes erros, enquanto o esquema de predição de paridade não conseguiu lidar com 5 destes erros. A detecção de falhas na parte combinacional implica em um novo ciclo de *clock* para a correção das falhas ocorridas na operação atingida. Isso causa um pequeno atraso no término da operação, o que na maioria das vezes não será perceptível para o usuário. Porém, o atraso pode ser usado por um atacante como uma pista valiosa do sucesso da tentativa de injeção de falhas.

Localização	**Esq. Prot.**	**Exec.**	**Falhas**	**Int.**	**Corr. (%)**	**Susp. (%)**
Matriz State	nenhum	10000	9498	-	94,98	-
	Hamming		1	26	0,01	0,026
	Reed-Solomon		6	6423	0,06	64,23
Memória das Chaves	nenhum	10000	9988	-	99,88	-
	Hamming		9	755	0,09	7,55
	Reed-Solomon		267	4344	2,67	43,44
Lógica Comb.	nenhum	2000	90	-	4,5	-
	Duplicação		0	0	0	0
	Parity Prediction		5	0	0,25	0

Tabela 6.5: Resultados para injeção de falhas múltiplas.

6.4 Avaliação dos resultados

Os resultados experimentais de injeção de falhas simples são bastante positivos. O esquema de proteção do *core* com *Hamming* para as memórias e duplicação parcial ou predição de paridade para a parte combinacional consegue detectar e corrigir 100% das falhas. Isto significa transparência para o usuário e incerteza do sucesso da injeção de falha para o atacante. O código corretor *Reed-Solomon*, apesar de detectar 100% das falhas, não conseguiu corrigir todas. Interrupções no processamente podem ser perceptíveis ao usuário e dão pistas importantes ao atacante. Na proteção da parte combinacional, tanto predição de paridade quanto duplicação, conseguiram detectar e corrigir 100% das falhas. Assim, para a ocorrência de falhas simples, o código de *Hamming* pode ser associado ao esquema de predição de paridade ou duplicação parcial, atingindo 100% de detecção e correção das falhas no *datapath* do *core*. Mesmo com o uso do código *Reed-Solomon*, o *core* deixa de ser vulnerável à ação de falhas simples. Um atacante já não pode contar com falhas simples para o sucesso de seu ataque.

Para a injeção de falhas múltiplas, apesar de não detectar 100% das falhas em alguns casos, a cobertura atinjida é bastante positiva, portanto, dando um passo imporante na proteção de processadores criptográficos baseados no AES. Para a proteção das memórias, conseguiu-se reduzir a ocorrência de erros de processamento para menos de 1%. Isso é um grande ganho na proteção do *core*, visto que, na memória das chaves, por exemplo, quase 100% das falhas múltiplas causaram erros de processamento. Novamente, *Hamming* teve o melhor desempenho tanto na proteção da *matriz state* quanto da memória das chaves. O principal problema do *Reed-Solomon* é não suportar correção de falhas nos seus *bits* de código. Este fator foi determinante para que *Hamming* mostrasse superioridade. Por fim, para a lógica combinacional, a duplicação mostrou maior robustez à injeção de múltiplas falhas. A predição de paridade deixou que erros de computação se manifestassem em 0,25% das vezes. De fato, predição de paridade não suporta múltiplas falhas dentro de um mesmo *byte*, enquanto que, duplicação pode detectar qualquer número de falhas, desde que estas não ocorram nos mesmos sinais dos módulos duplicados. E mesmo assim, tais falhas teriam que acontecer no mesmo momento e, ambas durarem tempo suficiente para se manifestarem. Tal cenário é bastante improvável devido à complexidade da lógica combinacional.

As Tabelas 6.6 e 6.7 apresentam os resultados para síntese de área e *speed*, respectivamente. A síntese foi realizada com o Leonardo Spectrum da Mentor Graphics. A primeira coluna de ambas as tabelas indica o esquema de proteção usado. Estão presentes as quatro versões protegidas do *core* do AES e a versão sem proteção (na primeira linha de cada tabela). A segunda coluna, apresenta a área ocupada. O Leonardo Spectrum fornece esta informação através de um circuito equivalente mapeado para portas *nand* de duas entradas. Então a informação de área é dada através do número de portas *nands* necessárias para construir um circuito equivalente. A terceira coluna fornece a freqüência de *clock* em *MHz* atingida. A quarta coluna dá o percentual do *overhead* de área com base no *core* sem proteção. Por último, a quinta coluna indica o percentual de diminuição do *clock* devido ao *overhead* do esquema de proteção.

Os resultados de síntese mostram que o *overhead* de área é pequeno em vista do ganho de proteção alcançado. A freqüência de *clock* foi reduzida a aproximadamente *50%* do seu valor original independente do esquema de proteção adotado ou do tipo de síntese. Assim, a freqüência de operação deixa de ser um fator determinante para a adoção de uma ou outra técnica de proteção. Porém, as diferenças no *overhead* de área são bastante perceptíveis. A menor área foi alcançada utilizando-se os esquemas *Hamming* e predição

Proteção	**Área**	***Freqüência MHz***	***Overhead* de Área**(%)	***Overhead* de Freqüência (%)**
Sem Proteção	33040	143,1	-	-
Hamming/Paridade	33940	64,0	2,72	-55,28
Reed-Solomon/Paridade	34678	64,5	4,96	-55,07
Hamming/Duplicação	38322	64,5	15,99	-54,93
Reed-Solomon/Duplicação	39061	63,3	18,22	-55,77

Tabela 6.6: Resultados de síntese para área.

Proteção	**Área**	***Freqüência MHz***	***Overhead* de Área**(%)	***Overhead* de Freqüência (%)**
Sem Proteção	36466	155,2	-	-
Hamming/Paridade	39582	71,6	8,54	-53,87
Reed-Solomon/Paridade	40067	72,7	9,87	-53,16
Hamming/Duplicação	44743	73,1	22,70	-52,90
Reed-Solomon/Duplicação	45228	72,7	24,03	-53,16

Tabela 6.7: Resultados de síntese para *speed*.

de paridade. O *overhead* foi de apenas 2,72% para a síntese de área e de 8,54% para a síntese de *speed*. A proteção que envolve *Reed-Solomon* e predição de paridade deteve a segunda menor área. O *overhead* foi de 4,96% e 9,87% para síntese de área e *speed*, respectivamente. Aqui percebe-se um maior *overhead* de área do código de proteção de memórias *Reed-Solomon* em comparação com *Hamming*. A solução que utiliza *Hamming* e duplicação teve o terceiro maior *overhead*, sendo que este ficou em 15,99% para área e 22,70% para *speed*. Isso deixa claro que duplicação apresenta um *overhead* muito superior em comparação com predição de paridade. Por fim, o *core* protegido com *Reed-Solomon* e duplicação teve o maior *overhead* que ficou em 18,22% para síntese de área e 24,03% para síntese de *speed*.

Com base nas informações de tolerância a falhas (conseguidas através de simulação) e dos resultados de síntese, pode-se tomar uma decisão a respeito da adoção de uma das implementações avaliadas. Por ter uma eficiência menor na correção de falhas e um *overhead* ligeiramente maior, *Reed-Solomon* não mostrou ser um bom candidato. Assim restou a opção do uso de *Hamming* para a proteção das memórias. Para a parte combinacional, a decisão fica entre uma maior área com um nível de segurança maior (duplicação), ou menor área e um nível de segurança menor. Assim, a decisão fica por conta do projetista e este deverá levar em conta o nível de segurança desejado.

7 CONCLUSÃO

Este trabalho apresentou diferentes contramedidas para ataques baseados em injeção de falhas, ou seja, DFA. Foram apresentados os principais dispositivos criptográficos portáteis usados atualmente. Tais dispositivos implementam algoritmos criptográficos em *hardware* visando maior desempenho devido às características de baixo custo de seus processadores embarcados. As principais aplicações onde os dispositivos criptográficos são usados, assim como seus mecanismos de segurança, foram apresentados. Isso levou à exposição dos algoritmos criptográficos mais importantes: DES, AES e RSA. Na seqüência, foi demonstrando matematicamente como tais algoritmos podem ser quebrados através da injeção controlada de falhas. Isso demonstrou que a suscetibilidade de algoritmos criptográficos à DFA é uma vulnerabilidade grave, portanto, análises apuradas a respeito disto são necessárias. Em sistemas de alta segurança, a credibilidade dos mecanismos de segurança são de suma importância, mesmo para a contínua expansão deste mercado. As contramedidas propostas neste trabalho foram construídas com base no reuso de técnicas já existentes. Assim, métodos de detecção e correção de falhas já conhecidos e amplamente usados foram adaptados e utilizados para proteção de algoritmos criptográficos implementados em *hardware*.

O trabalho iniciou com o estudo a respeito de ataques por DFA contra o DES. Um ataque foi simulado revelando alguns pontos suscetíveis do algoritmo que podem levar à descoberta da chave secreta. Foi desenvolvido um programa para determinação da chave a partir de blocos criptografados com falhas, assim, mostrando de forma empírica a real ameaça do ataque. Então, uma contramedida foi proposta e consistiu na duplicação das partes críticas, ou seja, partes do *hardware* onde a injeção de falhas pode levar à obtenção de informações sigilosas. Este esquema foi implementado e validado, resultando em um *overhead* de 38% e perda de performance de cerca de 10%. Tal resultado é positivo, já que apresenta um *overhead* aceitável em comparação com outras técnicas, como a duplicação total. Além disso, a perda de performance é pequena. O *hardware* protegido reduz drasticamente a possibilidade de sucesso na ocorrência de um ataque, pois, impede a injeção de falhas nos componentes mais vulneráveis do DES. Outra vantagem da técnica é ser independente de implementação e, ainda, poder ser usada em outros algoritmos criptográficos, conforme foi demonstrado com sua aplicação na proteção do AES.

Devido ao sucesso da duplicação parcial no DES, tal técnica também foi aplicada ao AES. Porém, desta vez, com a adição de outros mecanismos de proteção para maior robustez. A duplicação parcial foi comparada com outra técnica de proteção para a parte combinacional. Tal técnica resultou em menor *overhead*, porém, menor cobertura de falhas, assim resultando em duas opções para proteção da lógica combinacional com diferentes características de área e proteção. Ainda foram usadas duas técnicas diferentes para proteção de dados em memórias: código de *Hamming* e *Reed-Solomon*, que apresentam

diferentes coberturas de falhas. Ambas as técnicas de proteção (memória e lógica combinacional) foram combinadas formando quatro diferentes versões do AES com proteção. As quatro versões foram validadas através de uma injeção maciça de falhas, onde diferentes simulações foram realizadas com diferentes números de falhas simultâneas sendo injetadas. Os resultados mostraram diferentes níveis de proteção associados a diferentes custos de área. No geral, as técnicas desenvolvidas apresentaram bons compromissos entre área, desempenho e cobertura de falhas se comparadas às soluções já existentes. Um passo importante para o ganho de proteção foi conseguido com a introdução de mecanismos de tolerância a múltiplas falhas. Isso permite tornar a ocorrência de falhas, em muitos casos, imperceptível não apenas para o usuário, mas principalmente para o atacante. O atacante não possui métodos exatos para injeção de falhas, portanto, o mascaramento da ocorrência de falhas causa incerteza ao atacante a respeito da eficiência do seu método de injeção de falhas.

A área de estudos que engloba ataques a dispositivos criptográficos em *hardware* é bastante ampla e ainda existem muitos caminhos pouco explorados. Com relação a DFA, outros algoritmos precisam ser explorados e novas contramedidas podem ser apresentadas. Técnicas de proteção contra falhas para máquinas de estados podem ser associadas às técnicas apresentadas para adicionar ainda mais segurança aos dispositivos. Além disso, existem algumas questões em aberto e que são importantes para a continuidade deste trabalho. É importante determinar como as soluções propostas afetam outras vulnerabilidades do *hardware* como DPA ou *timming analysis*. Isso responderia à seguinte questão: as contramedidas propostas neste trabalho tornam o dispositivo mais ou menos vulnerável com relação às outras formas de criptoanálise? Além disso, existem áreas paralelas que englobam estudos de outros tipos de ataques como DPA, *timming analysis* e a análise de emissões electromagnéticas e que podem ser abordados futuramente. O conhecimento das ameaças que cada técnica de criptoanálise impõe é importante para determinar o nível de segurança atingido com a implantação de novos sistemas baseados em dispositivos móveis. Outra área ainda pouco explorada é o estudo da correlação entre diferentes técnicas de proteção para diferentes tipos de ataques. Isto é, dadas diferentes soluções para diferentes formas de ataque, por exemplo DFA e DPA, é necessário determinar como tais soluções se comportam quando usadas juntas para a proteção de um mesmo sistema. Ou seja, qual o custo e o impacto no projeto para o uso de diferentes contramedidas em um mesmo dispositivo? Ou ainda, é possível formular contramedidas capazes de lidar com mais de um tipo de ataque?

REFERÊNCIAS

A. SHAMIR, E. B. e. Differential Fault Analysis of Secret Key Cryptosystems. **Proc. Advances in Cryptology - Crypto97**, [S.l.], 1997.

ABRAMOVICI, M.; BREUER, M. A.; FRIEDMAN, A. D. **Digital Systems Testing and Testable Design**. [S.l.]: IEEE Press, 1994.

ANDERSON, R.; KUHN, M. Tamper Resistance - a Cautionary Note. In: SECOND USENIX WORKSHOP ON ELECTRONIC COMMERCE, 1996. **Proceedings...** [S.l.: s.n.], 1996. p.1–11.

ANDERSON, R.; KUHN, M. Low Cost Attacks on Tamper Resistant Devices. In: IWSP: INTERNATIONAL WORKSHOP ON SECURITY PROTOCOLS, LNCS, 1997. **Anais...** [S.l.: s.n.], 1997.

BERTONI, G.; BREVEGLIERI, L.; KOREN, I.; MAISTRI, P.; PIURI, V. A Parity Code Based Fault Detection for an Implementation of the Advanced Encryption Standard. In: DFT '02: PROCEEDINGS OF THE 17TH IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT-TOLERANCE IN VLSI SYSTEMS, 2002, Washington, DC, USA. **Anais...** IEEE Computer Society, 2002. p.51–59.

BLöMER, J.; OTTO, M.; SEIFERT, J.-P. A new CRT-RSA algorithm secure against bellcore attacks. In: CCS '03: PROCEEDINGS OF THE 10TH ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 2003, New York, NY, USA. **Anais...** ACM Press, 2003. p.311–320.

BONEH, D.; DEMILLO, R. A.; LIPTON, R. J. On the Importance of Checking Cryptographic Protocols for Faults. **Lecture Notes in Computer Science, Advances in Cryptology, proceedings of EUROCRYPT'97**, [S.l.], p.37–51, 1997.

BREVEGLIERI, L.; KOREN, I.; MAISTRI, P. Incorporating Error Detection and Online Reconfiguration into a Regular Architecture for the Advanced Encryption Standard. **IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)**, [S.l.], 2005.

CHEN, Z. **Java Card Technology for Smart Cards - Architecture and Programmer's Guide.** 2nd.ed. [S.l.]: Addison Wesley, 2004.

DIFFIE, W.; HELLMAN, M. New Directions in Cryptography. **IEEE Trans. Inform. Theory**, [S.l.], 1976.

DREIFUS, H. **Smart cards** : a guide to building and managing smart card applications. [S.l.]: John Wiley, 1998.

DUSART, P.; LETOURNEUX, G.; VIVOLO, O. Differential Fault Analysis on A.E.S. In: CRYPTOLOGY EPRINT ARCHIVE: REPORT 2003/010. HTTP://WWW.IACR.ORG., 2003. **Anais...** [S.l.: s.n.], 2003.

ETOKEN, A. K. S. **eToken Strong Authentication and Password Management**. Disponível em: http://www.aladdin.com/etoken/ Acessado em Setembro de 2006.

ETOKEN, A. K. S. **Authentication Tokens**: the key to secure pcs and data. Disponível em: http://www.aladdin.com/etoken/brochures-whitepapers.asp. Acessado em Setembro 2006.

FIPS. **Announcing the ADVANCED ENCRYPTION STANDARD (AES)**. Disponível em: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Acessado em Novembro de 2006.

GADLAGE, M. J.; AL. et. Single Event Transient Pulsewidths in Digital Microcircuits. **IEEE Transactions on Nuclear Science**, [S.l.], v.51, n.6, p.3285–3290, 2004.

GANDOLFI, K.; MOURTEL, C.; OLIVIER, F. Electromagnetic Analysis: concrete results. **Lecture Notes in Computer Science**, [S.l.], v.2162, 2001.

GIRAUD, C. **DFA on AES**. Cryptology ePrint Archive, Report 2003/008.

HENDRY, M. **Smart Cards - Security and Applications**. 1st.ed. [S.l.]: Artech House Publishers, 1998.

HESS, E.; JANSSEN, N.; MEYER, B.; SCHüTZE, T. Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures - A Survey. **EUROSMART Security Conference**, [S.l.], 2000.

ISO/IEC 14443. Identification Cards-Contactless Integrated Circuit(s) Cards - proximity Cards.

ISO/IEC 7816. Identification Cards-Integrated Circuit(s) Cards with Contacts.

JURGENSEN, T. M.; GUTHERY, S. B. **Smart Cards - The Developer's Toolkit**. 1st.ed. [S.l.]: Prentice Hall, 2002.

KIM, W.; AL. et. A Platform-Based SoC Design of a 32-bit Smart Card. **ETRI Journal**, [S.l.], Dec. 2003.

KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. **Lecture Notes in Computer Science**, [S.l.], v.1109, p.104–113, 1996.

KOCHER, P.; JAFFE, J.; JUN, B. Differential Power Analysis. **Lecture Notes in Computer Science**, [S.l.], v.1666, p.388–397, 1999.

LIMA KASTENSMIDT, F. G. de. **Designing Single Event Upset Mitigation Techniques for Large SRAM-Based FPGA Components**. 2003. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.

LTD, A. **AMBA**. Advanced Microcontroller Bus Architecture Specification.

MANGARD, S.; AIGNER, M.; DOMINIKUS, S. A Highly Regular and Scalable AES Hardware Architecture. **IEEE Transactions on Computers**, Los Alamitos, CA, USA, v.52, n.4, p.483–491, 2003.

MCQUEEN, S. R. **OPENCORES.ORG**. http://www.opencores.org/projects.cgi/web/basicdes/overview.

MENEZES, A. J.; OORCHOT, P. C. V.; VANSTONE, S. A. **Handbook of Applied Cryptography**. [S.l.]: CRC, 1996.

MODELSIM Xilinx - User's Manual, Version 6.1e. [S.l.]: Xilinx, 2006.

MONNET, Y.; RENAUDIN, M.; LEVEUGLE, R.; FEYT, N.; MOITREL, P.; NZENGUET, F. M. Practical Evaluation of Fault Countermeasures on an Asynchronous DES Crypto Processor. In: IOLTS '06: PROCEEDINGS OF THE 12TH IEEE INTERNATIONAL SYMPOSIUM ON ON-LINE TESTING, 2006, Washington, DC, USA. **Anais...** IEEE Computer Society, 2006. p.125–130.

NEUBERGER, G. **Desenvolvimento de um Gerador e Otimizador Automático de Núcleos do Código de Correção de Erros Reed-Solomon**. Trabalho de Conclusão de Curso, UFRGS. 2003.

OPENCORES.ORG. Disponível em: www.opencores.org. Acessado em Setembro de 2006.

PIRET, G.; QUISQUATER, J.-J. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In: CRYPTOGRAPHIC HARDWARE AND EMBEDDED SYSTEMS - CHES 2003, 2003. **Anais...** [S.l.: s.n.], 2003.

RENAUDIN, M.; AL. et. High Security Smartcards. **Proceedings of the conference on Design, automation and test in Europe**, [S.l.], p.16–20, Feb. 2004.

RIVEST, R. L.; SHAMIR, A.; ADELMAN, L. M. **A METHOD FOR OBTAINING DIGITAL SIGNATURES AND PUBLIC-KEY CRYPTOSYSTEMS**. [S.l.: s.n.], 1977. (MIT/LCS/TM-82).

RUSSELL, D.; GANGEMI, G. T. **Computer Security Basics**. [S.l.]: O'Reilly and Associates, Inc., 1991.

SCHNEIER, B. **Applied Cryptography**. [S.l.]: John Wiley and Sons, Inc., 1996.

SECURITY, R. **RSA, The Security Division of EMC.** Disponível em: http://www.rsasecurity.com/.

SIMMONS, G. J. **Contemporary Cryptology**: the science of information integrity. [S.l.]: IEEE Press, 1991.

SKOROBOGATOV, S.; ANDERSON, R. Optical fault induction attacks. **Cryptographic Hardware and Embedded Systmes (CHES)**, [S.l.], 2002.

STINSON, D. **Cryptography**: theory and practice. 2nd.ed. [S.l.]: Chapman and Hall/CRC, 2002.

WEAVER, T. A. C.; GEBARA, F.; BROWN, R. Remora: a dynamic self-tuning processor. **University of Michigan CSE Technical Report CSE-TR-460-02**, [S.l.], 2002.