PUCRS

FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
MESTRADO EM ENGENHARIA ELÉTRICA

BRUNO CASAGRANDE PORCHER

DETECÇÃO DE ATAQUES POR CONTROLE DE FLUXO DE EXECUÇÃO EM SISTEMAS EMBARCADOS: UMA ABORDAGEM EM HARDWARE

Porto Alegre
2017

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

Bruno Casagrande Porcher

Detecção de Ataques por Controle de Fluxo de Execução em Sistemas Embarcados: uma Abordagem em Hardware

Porto Alegre - RS, Brasil

2017

Bruno Casagrande Porcher

Detecção de Ataques por Controle de Fluxo de Execução em Sistemas Embarcados: uma Abordagem em Hardware

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade do Rio Grande do Sul, como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Sinais, Sistemas e Tecnologia da Informação
Linha de Pesquisa: Sistemas de Computação.

Pontifícia Universidade do Rio Grande do Sul – PUCRS

Faculdade de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica

Orientador: Fabian Luis Vargas

Coorientador: Ariel Lutenberg

Porto Alegre - RS, Brasil

2017

Ficha Catalográfica

P833 Porcher, Bruno Casagrande

Detecção de ataques por controle de fluxo de execução em sistemas embarcados : uma abordagem em hardware / Bruno Casagrande Porcher . – 2017.
40 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Engenharia Elétrica, PUCRS.

Orientador: Prof. Dr. Fabian Luis Vargas.
Co-orientador: Prof. Dr. Ariel Lutenberg.

1. Controle de fluxo de execução. 2. Aplicação crítica. 3. Processadores single-core. 4. Checagem de integridade de código. I. Vargas, Fabian Luis. II. Lutenberg, Ariel. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA - PPGEE

DETECÇÃO DE ATAQUES POR CONTROLE DE FLUXO DE EXECUÇÃO EM SISTEMAS EMBARCADOS: UMA ABORDAGEM DE HARDWARE

CANDIDATO: BRUNO CASAGRANDE PORCHER

Esta Dissertação de Mestrado foi julgada para obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

DR. FABIAN LUIS VARGAS - ORIENTADOR

DR. ARIEL LUTENBERG - CO-ORIENTADOR

BANCA EXAMINADORA

DR. CÉSAR AUGUSTO MISSIO MARCON - FACIN - FACULDADE DE INFORMÁTICA - PUCRS

DRA. LETÍCIA MARIA BOLZANI POEHLS - DO PPGEE - FENG - PUCRS

Agradecimentos

Agradeço primeiramente ao meu orientador Dr. Fabian Luis Vargas, e meu coorientador Ariel Lutenberg pelos ensinamentos concedidos durante o desenvolvimento deste trabalho, sendo estes de vital importância para conclusão deste curso.

Aos colegas da Universidad de Buenos Aires que proporcionaram grande ajuda no desenvolvimento deste trabalho durante a minha estadia na Argentina.

Aos colegas do Laboratório EASE que me aconselharam e ajudaram no desenvolvimento deste trabalho, todas suas contribuições foram essenciais no desenvolvimento desta dissertação.

Agradeço também ao Programa de Pós Graduação em Engenharia Elétrica e a Pontifícia Universidade Católica pelas oportunidades ofertadas desde a minha graduação.

A CAPES pela bolsa de estudos proporcionada a mim para a realização deste curso de mestrado e para a execução de uma parte fora do país.

E finalizo agradecendo aos meus pais, a minha família e a minha noiva por estar sempre ao meu lado, dando apoio nas horas que mais precisei.

Obrigado a todos!

Resumo

O uso de sistemas computacionais está presente nos mais diversos meios em que vivemos e esta rápida expansão acaba por expor a população aos mais diversos tipos de vulnerabilidades. Um erro em um sistema crítico poderá ocasionar desde prejuízos financeiros, roubo de dados, danos ambientais ou até riscos à vida humana. Este trabalho foi desenvolvido visando dificultar que a tomada do controle de sistemas computacionais seja feita por um usuário mal intencionado. Este trabalho propõe uma abordagem em hardware para detecção de ataques que eventualmente causem qualquer tipo de alteração no fluxo de execução de um programa, com o diferencial de que não é necessário nenhuma alteração, nem mesmo o conhecimento prévio do código-fonte do programa da aplicação em questão. Assim, em mais detalhes, o objetivo deste trabalho é assegurar a confiabilidade de um sistema crítico, do ponto de vista em que o *software* que foi desenvolvido pelo programador seja idêntico ao *software* que está sendo executado no processador. Para isso serão utilizados pontos de checagem no programa capazes de verificarem a integridade do sistema durante a sua execução. A técnica proposta foi implementada através de um *software* que por sua vez, é responsável pela identificação prévia dos blocos básicos através do arquivo executável do sistema crítico, e um hardware dedicado, denominado de *Watchdog*, instanciado juntamente com o processador do sistema crítico. Para a validação da técnica proposta foram realizadas simulações funcionais e a avaliação foi realizada a partir de trechos de códigos capazes de exporem vulnerabilidades da base de dados, denominada *Common Vulnerabilities and Exposures* (CVE, 2017). A validação e a avaliação foram realizadas adotando uma versão soft-core do processador LEON3. Os resultados experimentais demonstraram a eficiência da técnica proposta em termos de detecção de corrupções em trechos de código e na execução de trechos de código não pertencentes ao programa original. Finalmente uma analise das principais penalidades agregadas pela técnica foram realizadas.

Palavras-chaves: Controle de Fluxo de Execução, Aplicação Crítica, Processadores Single-Core, Checagem de Integridade de Código.

Abstract

The use of computer systems is present in the most diverse environments in which we live and this rapid expansion exposes the population to the most diverse types of vulnerabilities. Errors in critical systems may result in financial loss, data theft, environmental damage or may even endanger human life. This work was developed to make it more difficult for malicious users to take control of computer systems. A hardware-based approach to detect attacks that cause changes to the program's execution flow, but with no necessity for change or even the previous knowledge of the source code, is proposed. Thus, the purpose of this work is to ensure reliability by guaranteeing that the software running on the processor is equal to the one developed by the programmer. To do so, checkpoints in the program verify the integrity of the system during its execution. The proposed technique is implemented by software, which is responsible for the prior identification of the basic blocks using the critical system's executable file. A dedicated hardware, denominated Watchdog is instantiated with the processor of the critical system and validated by functional simulations. The technique's evaluation was carried out by executing in the soft-core version of a LEON3 processor for code sections, which are capable of exposing the database's, denominated Common Vulnerabilities and Exposures (CVE, 2017). The experimental results demonstrate th proposed technique's efficiency in terms of corruption detection in code snippets and in the execution of snippets of code not belonging to the original program. Finally, an analysis of the main overheads is performed.

Key-words: Control Flow Execution, Critical Applications, Single-Core Processors, Code Integrity Check.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AHB	*Advanced High-performance Bus*
AMBA	*Advanced Microcontroller Bus Architecture*
CAD	*Computer Aided Design*
CERT	*Computer Emergency Readiness Team*
CFG	*Control Flow Graph*
CVE	*Common Vulnerabilities and Exposures*
DMA	*Direct Memory Access*
DRAM	*Dynamic Random-Access Memory*
FOSS	*Free and Open-Source Software*
FPGA	*Field Programmable Gate Array*
FPU	*Floating-Point Unit*
FTP	*File Transfer Protocol*
GPL	*General Public License*
IEEE	*Institute of Electrical and Electronics Engineers*
LGPL	*Library General Public License*
MMU	*Memory Management Unit*
PCI	*Peripheral Component Interconnect*
RAM	*Random-Access Memory*
RISC	*Reduced Instruction Set Computer*
ROM	*Read Only Memory*
SIS	*Signatured Instruction Streams*
SOC	*System-On-a-Chip*
SPARC-V8	*Scalable Processor Architecture Version 8*
VHDL	*VHSIC Hardware Description Language*

Sumário

1 Introdução

Sistemas computacionais críticos são alvos frequentes dos mais variados tipos de ataques que, por sua vez, apresentam umgrande potencial de afetarem dados processados e armazenados. Os eventuais danos ocasionados pela execução errônea de um sistema crítico podem ocasionar desde o não cumprimento de uma meta previamente estabelecida, como perdas econômicas, riscos a vida humana ou danos ambientais (SOMMERVILLE et al., 2003). A presença de sistemas computacionais nos mais variados dispositivos, como eletrodomésticos, automóveis, dispositivos portáteis e até reatores nucleares exige que a confiabilidade seja tratada como ponto chave no projeto dos mesmos, seguido por performance e consumo de energia (KANUPARTHI et al., 2012). Em outras palavras, um sistema crítico deve ser confiável e para tanto torna-se fundamental uma análise detalhada do nível de robustez do mesmo.

Neste contexto esta dissertação de mestrado propõe uma abordagem para o controle de fluxo de execução da aplicação que dispensa o conhecimento do código-fonte da aplicação que será executada. A técnica tem como objetivo detectar possíveis tentativas de intrusão que possam alterar o fluxo de execução do programa, bem como vulnerabilidades no desenvolvimento do *software* ou até mesmo tentativas de alteração na memória de instruções do processador. Em mais detalhes, a partir do arquivo executável será obtido o código de montagem (assembly) do programa e através do *software* proposto serão identificados os trechos de execução contíguos, chamados de blocos básicos, onde será gerado um *hash* juntamente com o número de instruções de cada um dos blocos básicos. Antes da execução do programa estas informações serão armazenadas e comparadas com os *hashes* gerados em tempo de execução. Caso haja qualquer divergência seja no número de instruções de um bloco básico ou no *hash* gerado em tempo de execução, um sinal de erro será indicado. Esta abordagem não permite a utilização grafo de controle de fluxo de execução (*Control Flow Graph*), pois o objetivo é também detectar os casos em que não seja possível determinar com precisão o fluxo do programa antes de sua execução. Isso acontece quando o destino de uma instrução de salto está condicionada a uma entrada externa ao sistema, por exemplo. Neste caso todos os possíveis destinos já estarão previamente mapeados pela técnica proposta e deverão ser devidamente identificados durante a execução do sistema crítico. Note que a proposta não realiza alterações no código do programa, logo, um mesmo *software* compilado para um determinado microprocessador que possui o hardware proposto, irá funcionar normalmente em um mesmo processador que não possua o hardware proposto neste trabalho, denominado *Watchdog*. Convém mencionar que a inserção do *Watchdog* junto ao núcleo do processador visa garantir a confiabilidade do sistema, aliada a um custo nulo de degradação de performance do sistema já que o controle

de fluxo é feito em paralelo à execução da aplicação. Além disso, a única situação em que poderá ocorrer algum tipo de interferência na execução do sistema é quando um desvio no fluxo do programa for detectado. Neste caso a interrupção da execução do programa poderá ser, em alguns casos a melhor solução para evitar maiores prejuízos até que medidas corretivas sejam adotadas. Para validar a capacidade de detecção de ataques da técnica proposta serão feitas simulações de tipos de ataques conhecidos e também serão executados trechos de código publicados na internet (CVE, 2017) explorando vulnerabilidades de *softwares* conhecidos. Na etapa de avaliação serão feitos levantamentos dos principais *overheads* gerados pela técnica proposta e a sua comparação com o estado-da-arte no controle de fluxo de execução de aplicações críticas.

1.1 Objetivos Geral e Específico

O objetivo da técnica proposta neste trabalho é garantir a integridade da aplicação que está sendo executada em sistemas críticos, através do controle do fluxo de execução das instruções da aplicação, sem gerar nenhum tipo de degradação de desempenho do sistema e com a menor latência de detecção possível. A técnica proposta baseia-se na inclusão de um hardware de monitoramento de instruções, denominado *Watchdog*, dentro do core do processador, com o mínimo de acréscimo de área possível. Neste contexto, identificam-se os seguintes objetivos específicos:

- Especificação da técnica proposta: de posse do arquivo executável da aplicação é possível "desmontá-lo"para a obtenção do código *assembly* da aplicação crítica que será executada. Este código é então dividido em blocos básicos onde são estabelecidos pontos de checagem através de um código de identificação único, denominado *hash*, entre um bloco básico e outro. Durante a execução do programa ao final de cada bloco básico haverá uma comparação do valor do *hash* gerado em tempo de execução com o gerado antes da execução, e caso um desses *hashes* não esteja idêntico ao valor calculado antes da execução, o hardware irá sinalizar um erro onde medidas de recuperação poderão ser tomadas;

- Implementação do hardware *Watchdog*: este componente é o responsável por monitorar as instruções da aplicação que estão sendo executadas no estágio de execução no pipeline do processador. De posse desses dados o *Watchdog* irá gerar os *hashes* e contar o número de instruções de cada bloco básico em tempo de execução. Ao final de cada bloco básico o *Watchdog* irá verificar se o *hash* e o número de instruções está correto. Caso haja alguma divergência um erro será sinalizado.

- Validação do *Watchdog*: serão simulados tipos de ataques conhecidos e também serão executados trechos de código explorando vulnerabilidades publicamente conhecidas para demonstrar a capacidade de detecção de ataques pelo *Watchdog* proposto;

- Avaliação da técnica: durante esta etapa será calculada a capacidade de detecção da técnica e seus principais *overheads*.

1.2 Motivação

A confiabilidade do sistema durante a execução de aplicações críticas é essencial, e para tal, faz-se necessária a utilização de técnicas de tolerância à falhas e/ou que evitem possíveis falhas ou eventuais ataques ao sistema. Com o aumento na complexidade dos sistemas de aplicações críticas, as falhas se tornam mais frequentes, assim reduzindo a confiabilidade dos sistemas (LEW; DILLON; FORWARD, 1988). Note que uma entrada mal intencionada pode resultar no comprometimento total do sistema. Um exemplo disso seria devido a uma escrita além da área delimitada de um vetor, o que por sua vez, poderia resultar em um *buffer overflow* onde o fluxo do programa eventualmente teria um destino inesperado, ocasionando desde erros ou até mesmo a obtenção de privilégios administrativos por um invasor (COWAN et al., 1998). Neste trabalho é proposta uma técnica baseada em hardware que visa realizar o monitoramento do fluxo de execução do programa da aplicação, tendo como referência apenas o arquivo executável da aplicação, ou seja, sendo desnecessária a posse do código fonte da mesma. Este hardware deve garantir que o *software* que está sendo executado mantenha-se idêntico ao que foi desenvolvido na fase de projeto. Esta garantia se dá através da divisão do código em blocos básicos e da geração de um número de checagem, denominado *hash*, para cada um destes blocos. Deste modo somente blocos básicos que foram identificados a partir de uma versão integra da aplicação poderão ser executados. Qualquer alteração, seja no *opcode* de uma instrução ou no valor de um de seus operandos, bem como saltos para trechos de código desconhecidos serão detectadas, e então a partir da sinalização em hardware deste problema, ações corretivas poderão ser tomadas. Para validar a técnica proposta, utilizou-se como estudo de caso um sistema baseado no processador LEON3 na versão de apenas um núcleo. O LEON3 é um modelo em VHDL sintetizável de um processador de 32 bits, compatível com a arquitetura SPARC V8 (COBHAM GAISLER, 2016). Este modelo é altamente configurável e bastante apropriado para o design de sistemas em um chip (SOCs). Já no que diz respeito a avaliação, a capacidade de detecção de falhas foi estimada através de uma série de experimentos de injeção de falhas e as principais penalidades associadas a técnica foram calculadas.

1.3 Estrutura do Manuscrito

O Capítulo 1 descreve o contexto e o principal desafio tratado neste trabalho. Além disso, o capítulo descreve sucintamente o funcionamento da técnica proposta, a motivação

que levou ao desenvolvimento deste trabalho e cita o objetivo geral e os específicos deste trabalho.

O Capítulo 2 expõe algumas das vulnerabilidades as quais os sistemas críticos estão expostos hoje em dia e também apresenta o estado-da-arte das técnicas de detecção de ataques existentes. Alem disso, também é apresentada uma descrição sobre a arquitetura do *hardware* que foi utilizado para a validação da técnica.

O Capítulo 3 apresenta uma descrição profunda da técnica proposta e descreve detalhadamente a sua especificação. O trabalho propõe a execução de um *software* que fará a identificação dos blocos básicos da aplicação crítica antes de sua execução, e a implementação do *Watchdog* junto ao núcleo do processador que irá monitorar a aplicação durante a sua execução.

O Capítulo 4 descreve as simulações realizadas para a validação funcional da técnica proposta e apresenta os resultados obtidos em cada uma das simulações.

O Capítulo 5 apresenta uma avaliação geral da técnica proposta em termos de capacidade de detecção e seus principais custos associados à sua implementação e execução. É realizada também uma avaliação da implementação desta técnica em comparação com o estado-da-arte.

O Capítulo 6 apresenta as considerações finais sobre a utilização da técnica e quais as melhorias que poderiam ser implementadas em trabalhos futuros.

2 Fundamentação Teórica

Este capítulo descreve os principais conceitos relevantes no contexto deste trabalho, bem como o estado-da-arte associado a técnica proposta.

2.1 Tipos de Ataques

A seguir serão definidos alguns dos tipos de ataques que podem ocorrer em um sistema e como eles podem ser evitados utilizando a técnica proposta:

- **Cold Boot Attack:** nesta técnica um usuário mal intencionado rouba os dados remanescentes na memória DRAM mesmo após o desligamento do sistema, poderá assim buscar por chaves criptográficas que podem comprometer seriamente a segurança estando nas mãos de um invasor com acesso físico ao sistema (HALDERMAN et al., 2009). De posse desses dados o invasor poderá obter o acesso à informações confidenciais e até tomar o controle total do sistema com privilégios administrativos;
- **Firewire DMA Attack:** nesta técnica um usuário mal intencionado utiliza a conexão firewire do barramento PCI para realizar requisições DMA maliciosas para ganhar controle da memória, poderia então alterar instruções do programa diretamente na mémoria para iniciar um ataque em tempo de execução. A criptografia de dados do disco rígido é alvo frequente de ataques deste tipo, através da obtenção da chave criptográfica que geralmente é armazenada na memória RAM (BLASS; ROBERTSON, 2012);
- **Buffer Overflow Attack:** nesta técnica um usuário mal intencionado explora falhas na verificação do tamanho de um array de entrada sendo armazenado em um buffer, a escrita fora da área delimitada de um buffer pode causar uma alteração no endereço de retorno de uma função, ocasionando no desvio não autorizado para um trecho de código malicioso (COWAN et al., 1998);
- **Code Injection Attack**, nesta técnica um usuário mal intencionado se aproveita de falhas de segurança no sistema crítico e coloca como entrada valores inesperados pelo sistema o que pode resultar num comportamento que pode alterar o curso de execução do programa, permitindo ao invasor ganhar acesso ao sistema e assim executar seu próprio código malicioso (RILEY; JIANG; XU, 2010).

Para que qualquer um dos ataques supracitados possam ser evitados o programa original já deverá ter sido previamente remontado e possuir suas específicas *hashes* (de-

finido na seção 3.2) e tamanhos armazenados em memória, portanto, quando o invasor tentar executar qualquer instrução que não esteja no programa original, ou até mesmo um bit de uma instrução que tenha sido invertido de forma acidental ou intencional, um erro será apontado pelo hardware de monitoramento de fluxo de execução. Nos casos em que o invasor tem a possibilidade de alterar diretamente o conteúdo de um registrador, como no Firewire DMA Attack, uma instrução de salto para o endereço contido naquele registrador pode fazer com que o programa tome um curso totalmente inesperado. Nestes casos as outras técnicas de controle de fluxo não detectarão a intrusão, pois dependem da instrução de checagem ao final de cada bloco básico, ou seja, se não houver a instrução de verificação de hash no código que está sendo executado, nada será percebido. Como a técnica descrita neste documento não realiza alterações no código que está sendo executado, qualquer código que não seja previamente conhecido não passará despercebido.

2.2 O Processador LEON3

O LEON3 é um modelo sintetizável em VHDL de 32-bits, baseado na arquitetura RISC SPARC-V8 e seu conjunto de instruções. Possui um pipeline de sete estágios, FPU em conformidade com a IEEE-754 e coprocessador opcional, arquitetura Harvard com memória de dados e instruções independentes e interface de barramento AMBA-2.0 AHB. Este modelo é altamente configurável e adaptável para designs SOC (WEAVER; GREMOND, 1992).

O projeto LEON foi originalmente projetado pela European Space Agency (ESA) em meados de 1997, com o intuito de estudar e desenvolver um processador de alta performance para ser utilizado em projetos aeroespaciais na Europa (ANDERSSON; GAISLER; WEIGAND, 2010). Foram então desenvolvidas as duas primeiras versões do LEON que tinham os objetivos de fornecer um design de processador aberto, portátil e não proprietário, sendo disponibilizado em dois modelos de licensa, LGPL/GPL e FLOSS que podem ser utilizadas para fins educacionais e de pesquisa, ou uma licença pode ser adquirida a baixos custos, representando uma fração do custo de núcleos IP para integração em hardware proprietário (EE TIMES, 2005).

A versão do LEON utilizada neste trabalho (LEON3) foi posteriormente disponibilizada pela Gaisler Research, que acabou sendo adquirida pela companhia americana Aeroflex Inc. em 2008, e em Maio de 2014 a Aeroflex foi adquirida pela companhia britanica Cobham, que mantém o projeto LEON atualmente (THE WALL STREET JOURNAL, 2014).

2.2.1 Integer Unit (IU)

A IU possui registradores de uso geral e controla toda operação do processador. Esta unidade executa intruções aritméticas com inteiros e calcula endereços de memória para carregamentos e armazenamentos de dados. É responsável também pelos contadores de programa e controla a execução de instruções para o FPU e o coprocessador. Uma implementação da IU pode conter de 40 até 120 registradores de 32-bits de uso geral. Isto corresponde em 8 registradores globais, mas uma pilha circular de 2 até 32 conjuntos de 16 registradores, chamadas janelas de registradores (WEAVER; GREMOND, 1992). A IU do LEON3 utiliza um pipeline de sete estágios, são eles (GAISLER et al., 2007):

- **1. Instruction Fetch**: caso o cache de instruções esteja habilitado, a instrução será buscada do cache de instruções. Caso contrário, a busca será direcionada para o barramento AHB.
- **2. Decode**: a instrução é decodificada e o endereço de destino de saltos e chamadas é gerado.
- **3. Register access**: operandos são lidos dos registradores ou são repassados de dados internos.
- **4. Execute**: ULA, lógica, e operações de deslocamento são executadas. Para operações de memória e para JMPL/RETT, o endereço será gerado. Neste estágio o Watchdog proposto por este documento está incluso, e portanto, calcula o *hash* com todas as instruções que passam por este estágio.
- **5. Memory**: o cache de dados é lido ou atualizado neste estágio.
- **6. Exceptions**: *traps* e interrupções são resolvidas. Para leituras da cache, o dado é alinhado apropriadamente.
- **7. Write**: o resultado de qualquer ULA, lógica, deslocamento, ou operação da cache é escrita de volta no registrador.

2.2.2 Sistema de Cache

O pipeline do processador LEON3 implementa uma arquitetura Harvard com barramentos de instruções e de dados independentes, conectados a dois controladores de cache também independentes. Enquanto a execução não ocasiona um *miss* na cache, os controladores da cache podem fornecer uma busca de instrução de 32-bits e uma carregamento/armazenamento de dados por ciclo de clock, mantendo a execução do pipeline em sua velocidade máxima. Cada controlador pode ser configurado com diferentes capacidades de armazenamento e políticas de substituição de dados, e também é possível anexar uma RAM local para cada controlador.

Em um *miss* na cache, o controlador irá indicar um sinal de parada para o pipeline que deverá aguardar a chegada do dado e então continuará sua execução. Um dos componentes inclusos na cache é o *buffer* de escrita, que permite que os armazenamentos ocorram em paralelo com a execução das instruções.

2.2.3 Conjunto de Instruções

O conjunto de instruções do LEON3 é dividido em seia categorias, são elas:

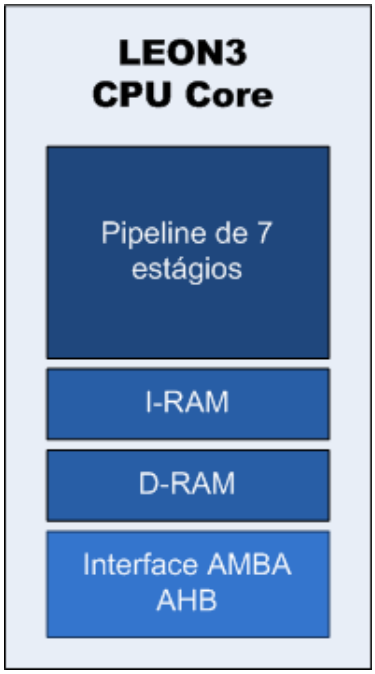
- **Load/store**: as instruções de carregamento e armazenamento são as únicas instruções que acessam a memória. Elas utilizam dois registradores ou um registrador ou um deslocamento de 13-bits sinalizado para calcular um endereço de memória de 32-bits.
- **Arithmetic/logical/shift**: as instruções de aritmética, lógica e deslocamento realizam este tipo de operação, com uma exceção, estas instruções calculam um resultado que é uma função de dois operandos de origem, o resultado é escrito num registrador de destino ou descartado.
- **Control transfer**: as instruções de transferência de controle incluem saltos por deslocamento e chamadas de função, saltos por endereço armazenado em registradores e *traps* condicionais.
- **Read/write control register**: as instruções de leitura e escrita de registradores de controle lêem e escrevem os conteúdos em registradores de estado e status visíveis por *software*.
- **Floating-point operate**: instruções de ponto flutuante são utilizadas para executar todos os cálculos de ponto flutuante. São instruções que operam de registrador para registrador em registradores de ponto flutuante.
- **Coprocessor operate**: são instruções de operação do coprocessador e definidas pelo coprocessador implementado, se houver.

2.2.4 Configuração Utilizada

Para que possa ser feito um controle sequencial das instruções que estão sendo executadas, algumas limitações foram impostas para que o funcionamento da técnica se tornasse eficaz na detecção de ataques. Para tanto foi utilizada a versão soft-core do LEON3, onde todas as instruções do programa sejam executadas em um mesmo core, simplificando assim o seu monitoramento. O cache, a MMU e a FPU foram desabilitados a fim de evitar que instruções fossem executadas fora da sequência do programa original, pois instruções que pudessem ser adiantadas ou atrasadas afetariam diretamente no

calculo dos *hashes*. A figura 1 apresenta o diagrama simplificado do modelo utilizado na demonstração da técnica.

Figura 1 – Diagrama simplificado do core do LEON3

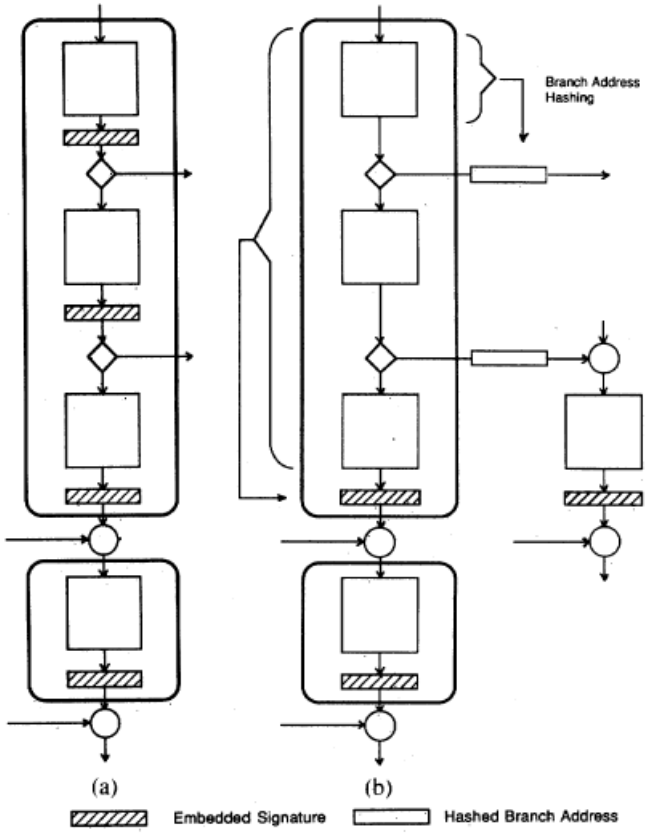


2.3 Estado-da-Arte: Técnicas para Detecção de Ataques

Algumas das técnicas atuais utilizadas para o monitoramento de controle de fluxo se baseiam na montagem de um Grafo de Fluxo de Controle (CFG) (LI; TAN; XU, 2010) para o programa. Neste grafo os vértices representam os blocos básicos e os arcos representam a origem e o destino de um trecho não contiguo, ou seja, um bloco básico é definido por um ponto de entrada onde há apenas uma sequência válida de instruções do ponto de entrada até o ponto de saída. A partir da montagem deste grafo, uma das abordagens utilizadas se chama *Signatured Instruction Streams* (SIS) (SCHUETTE; SHEN et al., 1987), onde as instruções são codificadas e uma assinatura é inserida no final de cada bloco durante a compilação do programa crítico, nesse caso o código de máquina obtido será diferente do código de máquina obtido pelo compilador original da arquitetura e deste modo não poderá ser executado em uma mesma arquitetura que não esteja com estas modificações inseridas pela técnica. A partir da comparação do *hash* inserido no final do bloco com o *hash* gerado em tempo de execução é possível verificar a integridade de cada bloco. A figura 2(a) representa o código gerado pela técnica com as assinaturas inseridas após o final de cada bloco básico, e a figura 2(b) representa o mesmo código gerado pela técnica proposta, mas agora utilizando um método chamado *Branch Address Hashing* (SCHUETTE; SHEN et al., 1987), que é utilizado para economizar espaço em memória mesclando a assinatura do bloco básico através da operação OU-Exclusivo (XOR) com

o endereço de destino do salto. Consequentemente, é desnecessária a utilização de um espaço em memória apenas para armazenar cada assinatura de cada um dos blocos básicos. Através da compilação, a técnica de *Branch Address Hashing* irá fazer com que o endereço de salto no código objeto seja inválido. Todavia, a assinatura gerada em tempo de execução será utilizada para obter o endereço correto que havia sido mesclado com a assinatura durante a compilação. Se a assinatura gerada em tempo de execução estiver correta, então o endereço de salto original também estará correto e será então um dos destinos previstos pelo CFG. Este endereço será então utilizado pelo processador.

Figura 2 – Diagrama proposto pela técnica chamada *Signatured Instruction Streams*



Fonte: SCHUETTE; SHEN et al. 1987

Uma outra técnica proposta para controle de integridade de fluxo de execução é chamada de *Dynamic Sequence Checker* (DSC) (KANUPARTHI; RAJENDRAN; KARRI, 2016), que gera assinaturas para cada um dos blocos básicos do programa crítico em tempo de compilação, de modo que a distância de Hamming entre dois blocos básicos devidamente conectados é uma constante conhecida. Em tempo de execução, a distância de Hamming entre as assinaturas dos blocos básicos de origem e destino é calculada e comparada com a constante conhecida, para assim estabelecer o controle de fluxo. A execução do programa crítico é interrompida se a distância de Hamming não estiver correta.

Ataques em tempo de execução é um dos principais tipos de ataques existentes, isso acontece devido à maioria dos programas serem escritos em linguagens inseguras (como C ou C++), que não verificam os limites de área reservada para o programa, para escrita através de uma entrada de dados. Neste caso uma entrada de dados ilegal poderia ir além da área de dados reservada para o programa crítico e então sobrescrever o endereço de retorno de uma função diretamente na pilha do programa, fazendo com que um invasor possa tomar o controle do sistema. Este tipo de ataque pode ser detectado por técnicas que verificam a integridade do endereço de retorno, como as técnicas citadas anteriormente. Porém, invasores podem burlar estes mecanismos usando a técnica chamada *Code Reuse Attacks* (CRA). CRA's podem ilegalmente transferir o controle para um código existente sem a necessidade de injetar algum código malicioso. Neste caso o invasor utiliza vulnerabilidades de trechos de código conhecidos do programa crítico, apenas inserindo entradas específicas que manipulam o fluxo do programa para obter um resultado malicioso. Algumas técnicas de proteção contra CRA's já foram propostas, algumas técnicas baseadas em integridade de controle de fluxo em software (ABADI et al., 2005) e outras propostas em hardware (KANUPARTHI; KARRI, 2015), porém, as técnicas que oferecem este tipo de proteção acabam sofrendo com a alta degradação de desempenho do sistema, uma vez que interrompem a execução do programa crítico durante a verificação de integridade.

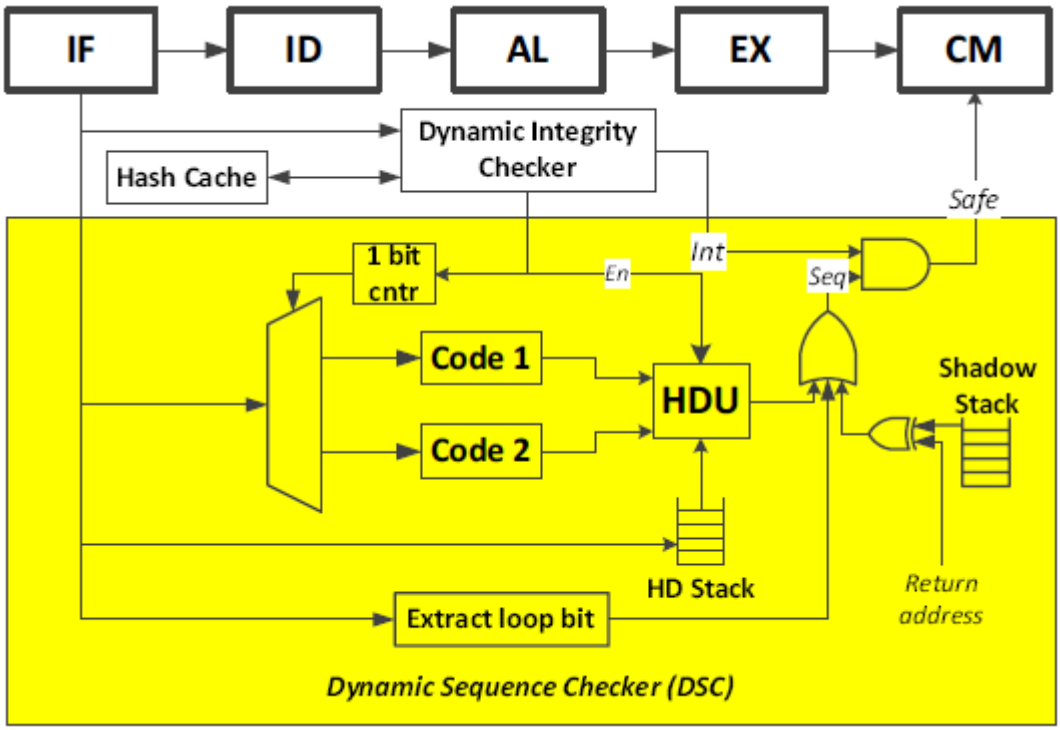
A técnica DSC assegura que o controle de fluxo seguirá de acordo com o CFG do programa crítico, evitando ataques que modificam instruções que estão na memória, transfiram o controle para um trecho de código que esteja fora da área de endereçamento reservada para o programa crítico, transfiram o controle para o meio de uma sequência de bytes de instrução, ou por último, transfiram ilegalmente o fluxo para um trecho de código autentico do programa crítico.

CRA's assim como Programação Orientada a Retorno (*Return Oriented Programming*, ROP), utilizam trechos de código do programa sob ataque para criar códigos de ataque arbitrários (SHACHAM, 2007). O invasor identifica pequenas sequências de código, chamadas *gadgets*, que terminam com uma instrução de retorno. Através da identificação de um número suficiente de *gadgets*, é possível gerar um código arbitrário de ataque e então explorar vulnerabilidades como o ataque de *buffer overflow*, para inserir uma sequência de endereços de retorno correspondente à sequência de *gadgets*. Após um *gadget* finalizar sua execução, o controle é transferido para o próximo, e assim por diante.

Os dois principais passos utilizados pela técnica de DSC são: busca dos códigos de Hamming (gerados durante a compilação) para o processador, e checar a integridade das transferências de controle do programa crítico. A figura 3 apresenta o diagrama do hardware proposto pela técnica de DSC, onde é possível observar que já existe no processador um sistema de Checagem de Integridade Dinâmica (*Dynamic Integrity Checker*, DIC), que possui sua própria memória de *hashes*. Sendo que a técnica de DSC utiliza

além da memória de distâncias de Hamming (*HD Stack*), uma outra memória chamada *Shadow Stack* que armazena os endereços de retorno das chamadas de função, esta para garantir que ao final da execução de uma função o controle do programa crítico retornará para o endereço de origem da chamada, sejam por chamadas de função diretas (quando o endereço de destino está junto da instrução de salto) ou indiretas (quando o endereço de destino está armazenado em um registrador ou memória).

Figura 3 – Diagrama do hardware proposto pela técnica de DSC



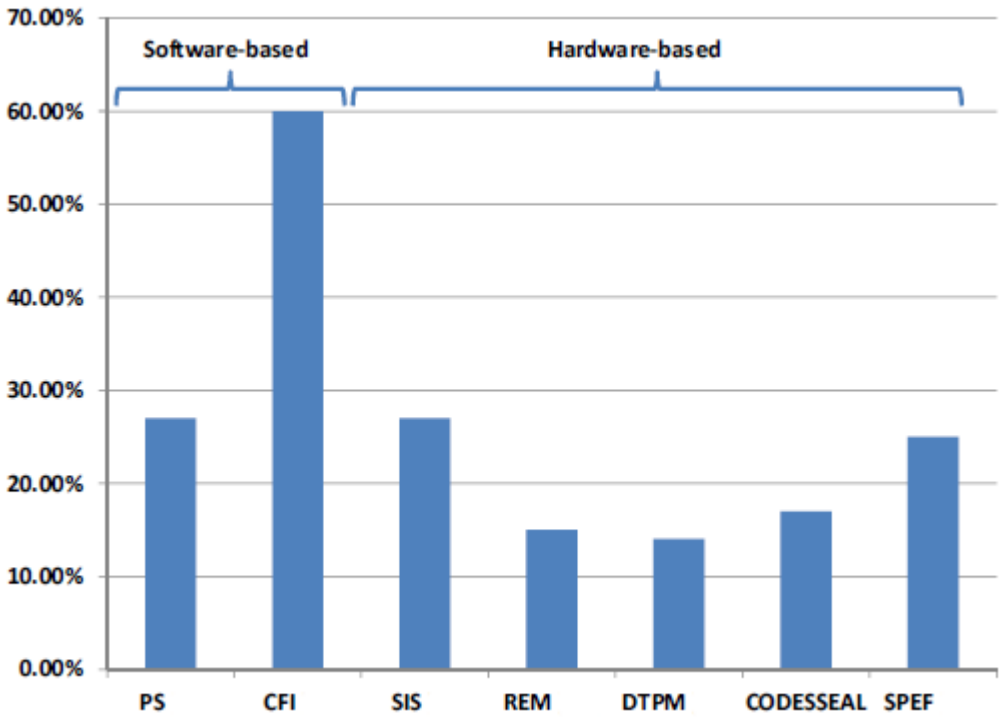
Fonte: KANUPARTHI; RAJENDRAN; KARRI 2016

Em geral a técnica de DSC cobre os mesmos cenários de ataque propostos neste documento, mesmo os casos de saltos indiretos que não são cobertos pela maioria das técnicas que utilizam CFG, onde o endereço de destino de um salto pode estar armazenado em um registrador ou em um endereço de memória. O que ambas as técnicas fazem é garantir que o controle do programa não seja alterado para o meio de uma sequência de bytes qualquer, ou para o meio de um bloco básico, meio de uma função, ou para um código de ataque malicioso. As principais desvantagens da utilização de DSC frente à técnica proposta, são de ter a necessidade da posse do código fonte para obtenção dos códigos de Hamming e apresenta uma pequena degradação de desempenho, estimada em 4,7%. Por outro lado, a técnica de DSC apresenta uma baixa utilização de memória comparada à técnica proposta e um mesmo hardware é capaz de executar diferentes aplicações, sem a necessidade de sofrer alterações.

As técnicas vistas até agora visando garantir a integridade de execução de um programa crítico, possuem suas vantagens e desvantagens, e dentre os principais problemas encontrados está a degradação de desempenho. Uma proposta que permite realizar as devidas checagens de integridade durante a execução do sistema crítico, com baixa degradação de desempenho e baixo consumo de energia, chama-se Checagem de Integridade Dinâmica baseada em Hardware (*Hardware-based Dynamic Integrity Checking*) (KANUPARTHI et al., 2012). Consiste no cálculo de assinaturas (*hashes*) das instruções que estão sendo executadas e a comparação com as assinaturas previamente calculadas, antes da execução do programa crítico. Esta abordagem não necessita que o pipeline pare para aguardar a checagem de integridade, pois permite que as instruções submetam os resultados das operações antes mesmo do final da checagem de integridade. Isto se dá devido ao fato de que o banco de registradores possui uma cópia de backup (chamado *Shadow Register File*), e tão somente a checagem de integridade tenha sido concluída, a memória cache e a memória principal poderão então ser atualizadas. Deste modo, caso a verificação de integridade tenha encontrado alguma alteração nas instruções, o sistema poderá ser restaurado para um estado conhecido. A figura 4 apresenta a média de degradação de performance de técnicas existentes de checagem de integridade dinâmicas, as baseadas em software CFI (ABADI et al., 2009), PS (KIRIANSKY et al., 2002) e as basseadas em hardware SIS (SCHUETTE; SHEN et al., 1987), REM (FISKIRAN; LEE, 2004), DTPM (KANUPARTHI; ZAHRAN; KARRI, 2010), CODESSEAL (GELBART et al., 2005), SPEF (KIROVSKI; DRINIĆ; POTKONJAK, 2002).

Nos casos em que existam dependências de dados entre as instruções que estão no pipeline, os dados serão redirecionados diretamente do *Shadow Register File* (para instruções que não atualizam a memória) e no *Store Buffer* (para as instruções que atualizam dados na memória). Uma vez que a integridade do programa crítico é verificada com sucesso, os dados são escritos na memória cache ou no banco de registradores original.

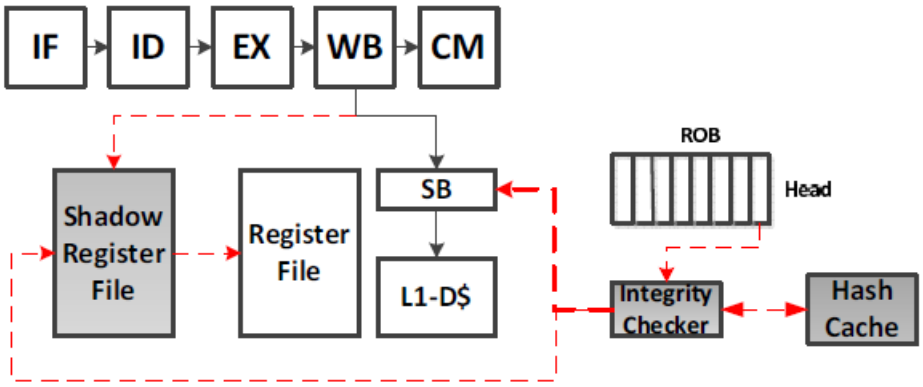
Figura 4 – Média de degradação de performance de técnicas existentes de checagem de integridade dinâmicas



Fonte: KANUPARTHI et al. 2012

A figura 5 apresenta do diagrama de arquitetura da técnica de checagem de integridade dinâmica baseada em hardware, onde os blocos em cinza representam os componentes adicionados pela técnica e as linhas tracejadas representam as novas interconexões. O estágio de *Write Back* do *pipeline* irá escrever os resultados das operações no *Shadow Register File* até que o módulo *Integrity Checker* envie o sinal de confirmação para que os registradores originais sejam atualizados. O módulo *Hash Cash* armazena os *hashes* dos blocos básicos mais recentemente executados, e que foram gerados antes da execução do sistema crítico. A técnica utiliza os sinais do estágio de *Write Back* devido à presença do *Reorder Buffer* (ROB) que apresenta as instruções do programa crítico em sua ordem original. Um dado que deve ser escrito na cache permanecerá armazenado no *Store Buffer* (SB) até que a checagem de integridade seja finalizada com sucesso. Nos casos de erro de integridade, os dados originais poderão ser restaurados.

Figura 5 – Diagrama de arquitetura da técnica de checagem de integridade dinâmica baseada em hardware



Fonte: KANUPARTHI et al. 2012

Esta técnica não necessita que o *pipeline* interrompa sua execução para aguardar a checagem de integridade, justamente por armazenar cópias temporárias dos dados que estão sendo processados. Dentre os números apresentados pela técnica estão: um acréscimo de área de 4,25%, um aumento de 2,45% de consumo de energia e uma degradação de desempenho de 1,66%. Possui ainda como vantagens sobre a técnica proposta neste documento, a sua capacidade de execução de diferentes aplicações críticas sem que sejam feitas adaptações no hardware. Como as outras técnicas abordadas no início desta seção, esta proposta não apresenta uma solução para verificação de integridade em casos de saltos dinâmicos, onde o destino de um salto está armazenado em memória ou registrador. E assim como todos os outros casos apresentados nesta seção, possui como desvantagem a necessidade de recompilação do código fonte do programa crítico.

3 A Técnica Proposta

Este capítulo apresenta a especificação da técnica proposta, incluindo requisitos funcionais e qualitativos da mesma. Além disso este capítulo descreve detalhes acerca da implementação da técnica. A técnica proposta baseia-se no desenvolvimento de um *software* de geração de *hashes* e na introdução de um *Watchdog* junto ao estágio de execução do *pipeline*, junto ao núcleo do processador para o monitoramento do fluxo de execução de instruções de uma aplicação crítica, tendo como referência apenas as instruções obtidas a partir do arquivo executável da aplicação, sendo assim, desnecessária assim a posse do código fonte da mesma. Note que o *Watchdog* deverá garantir que o programa que está sendo executado seja idêntico ao que foi desenvolvido. Para tanto deverá ser feita a divisão do código em blocos básicos e a geração de um número de checagem, *hash*, para cada um destes blocos. Deste modo somente blocos básicos que foram identificados a partir de uma versão integra da aplicação crítica poderão ser executados. Qualquer alteração, seja no *opcode* de uma instrução ou no valor de um de seus operandos, bem como saltos para trechos de código desconhecidos serão detectados, e então a partir da sinalização em hardware deste problema, ações corretivas poderão ser tomadas. Neste contexto, as próximas seções deste capítulo descrevem como são definidos os blocos básicos e como a integridade do programa será constatada durante a sua execução. A figura 6 apresenta o diagrama geral da técnica proposta, contendo os sinais obtidos pelo *Watchdog* durante o estágio de execução do pipeline do processador LEON3.

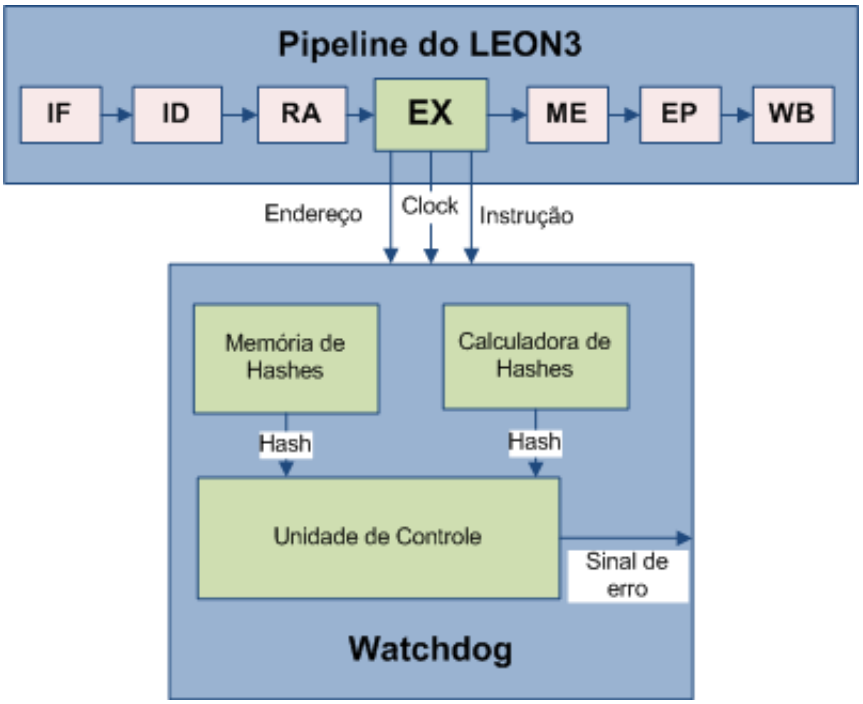
Neste sentido, identificam-se os seguintes requisitos funcionais do *Watchdog*:

- detectar qualquer tipo de alteração nas instruções do programa, até mesmo se apenas um bit da aplicação for modificado o *Watchdog* deverá ser capaz de detectar;
- detectar se o bloco básico que está sendo executado faz parte do programa original, pois se de alguma forma um usuário mal intencionado conseguir modificar o fluxo do programa, o *Watchdog* deverá ser capaz de verificar;
- calcular o *hash* do bloco básico durante a execução da aplicação crítica e verificar se o valor coincide com o *hash* gerado na etapa de pré-execução;
- verificar se o número de instruções de cada bloco básico está correto. Durante a etapa de pré-processamento os blocos básicos devem ser identificados e deverão ter um *hash* e o número de instruções correspondente armazenado;
- apontar um sinal de erro para os casos de erro descritos acima.

Já no que diz respeito à implementação, identificam-se os seguintes requisitos qualitativos do *Watchdog*:

- baixo *overhead* de desempenho para que não interfira na aplicação que está sendo executada;
- baixa latência de detecção, pois caso demore para constatar o ataque, a segurança do sistema pode já estar comprometida;
- baixo *overhead* de área para que seu custo de implementação seja viável no projeto do sistema.

Figura 6 – Diagrama da técnica proposta



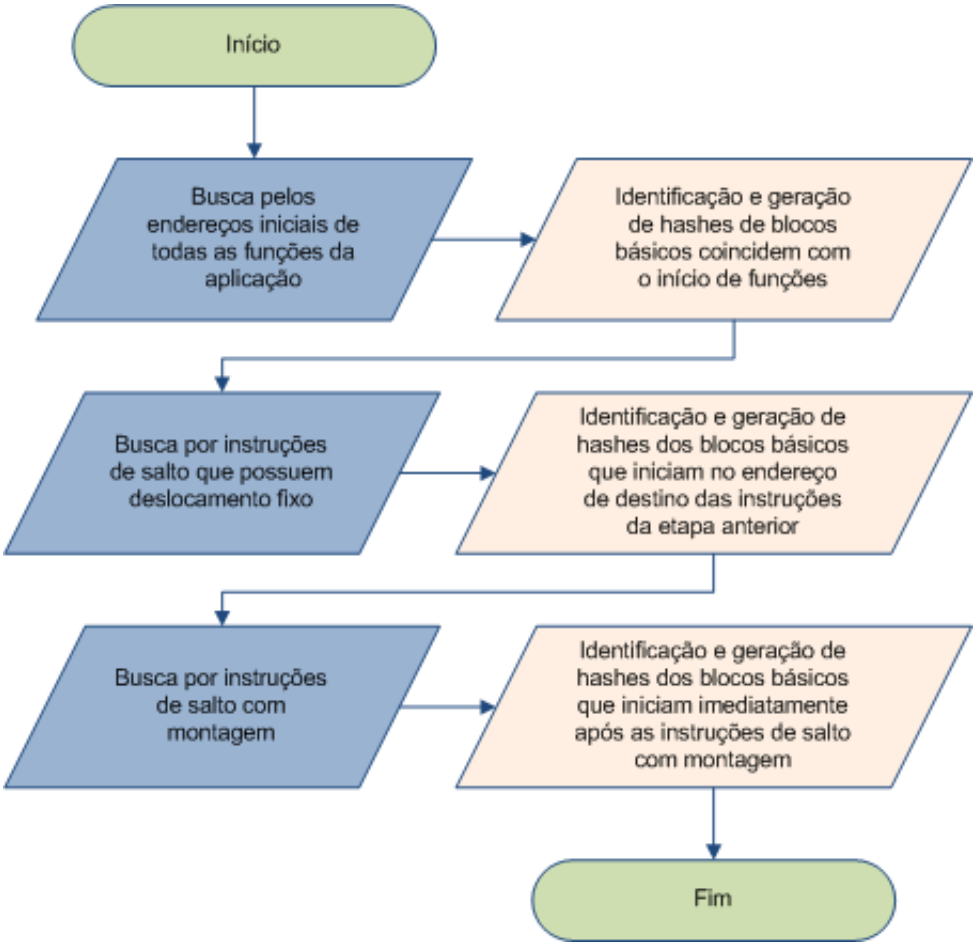
A seguir serão descritos em detalhes a forma como a técnica divide o programa da aplicação em blocos básicos, bem como a especificação do *Watchdog* proposto.

3.1 Identificação de Blocos Básicos da Aplicação

Um bloco básico consiste em um conjunto de instruções contíguas em que existe apenas uma entrada e uma saída. As entradas são o início de um bloco básico, ou seja, são todos os endereços possíveis para o qual o programa possa saltar e portanto, determinarão o número de blocos básicos existentes em um programa. As saídas serão as quebras de continuidade no endereçamento do programa e portanto, todas as instruções de salto.

Antes da execução do programa crítico, na etapa de mapeamento das entradas, podem haver casos em que não seja possível determinar o destino de um salto, como no caso de um tipo específico de salto onde o endereço de destino não é um parâmetro fixo da instrução, mas sim um valor armazenado em um registrador. É praticamente impossível antever com segurança o valor que estará armazenado em um registrador durante a execução do programa crítico, já que seu conteúdo pode estar vinculado a uma entrada externa, ou então ao resultado de alguma operação. Instruções deste tipo são tipicamente utilizadas quando uma função é chamada por ponteiro, nesses casos pressupomos que toda e qualquer chamada com destino previamente indeterminável deverá ser para o início de uma função qualquer. Como todo início de função é também o início de um bloco básico, estes casos estarão devidamente cobertos. Devido a não termos acesso ao código fonte e existir a possibilidade de ocorrerem esses tipos de salto, a técnica não permite a montagem de um Grafo de Fluxo de Controle (CFG). Resumidamente, serão identificados três tipos de blocos básicos, em primeiro lugar aqueles que coincidem com o início de uma função, em segundo lugar serão identificados os blocos básicos que coincidem com o endereço de destino de todas as instruções de salto da aplicação crítica e por último os blocos básicos que iniciam a partir da instrução seguinte a uma instrução de salto com "montagem", uma vez que após a execução do bloco básico de destino deste salto, a execução do programa crítico deverá retornar para a instrução seguinte a instrução de salto. A figura 7 apresenta o fluxograma do algoritmo de varredura do código da aplicação crítica na busca dos blocos básicos.

Figura 7 – Fluxograma do algoritmo de identificação dos blocos básicos



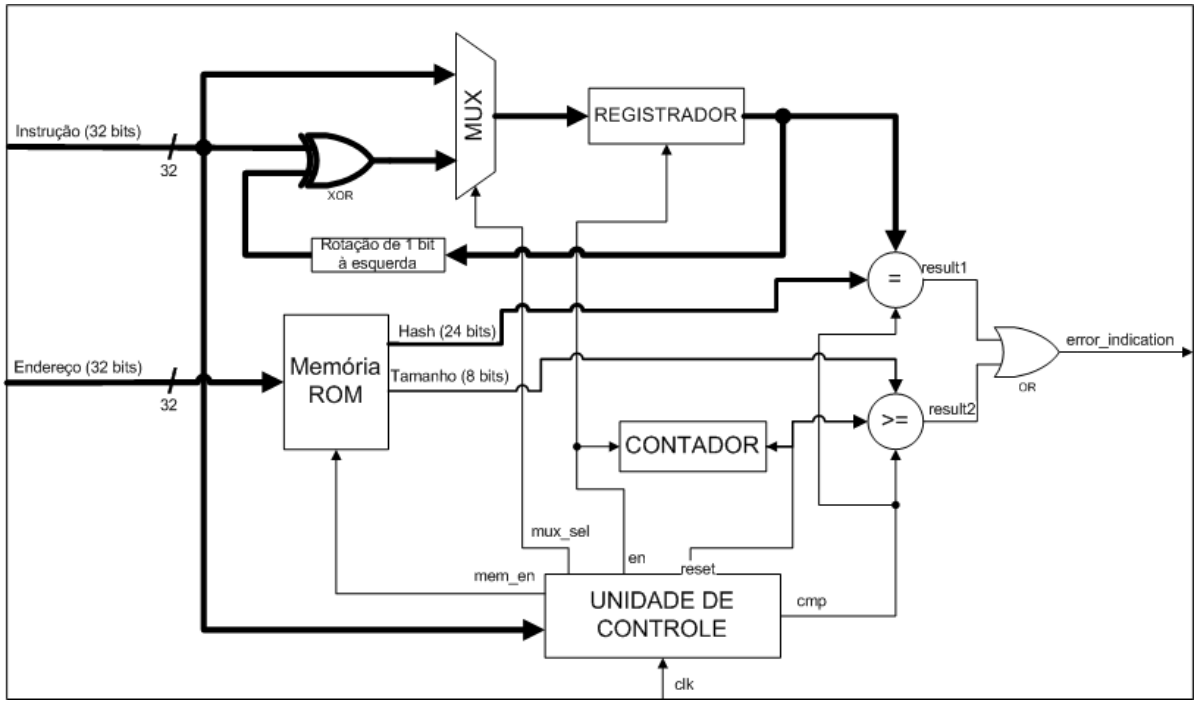
3.2 Especificação do Watchdog

O *Watchdog* proposto tem por objetivo realizar o monitoramento do fluxo de execução das instruções de uma determinada aplicação crítica. Para que isto seja feito a técnica proposta é dividida em duas etapas, sendo a primeira de pré-processamento que irá identificar a estrutura da aplicação e gerar os *hashes*. E a segunda etapa consiste no monitoramento de sinais do estágio de execução pipeline do processador. Basicamente são obtidos os sinais de clock, endereço e instrução, para que os *hashes* sejam gerados em tempo de execução e permitam assim gerar pontos de checagem para validação de integridade da aplicação ao final da execução de cada bloco básico. Através de um *software* que remonta o arquivo binário da aplicação crítica antes de sua execução, são definidos os blocos básicos com seus respectivos *hashes* e tamanhos, estes com 24 e 8 bits respectivamente serão armazenados em uma memória do tipo estática e somente para leitura durante a execução do programa crítico, já que esta não deverá ser alterada após a síntese do programa no FGPA. Devido a necessidade de detecção de erros com a menor latência possível, para que a tomada de decisão seja feita antes da execução de um trecho de código malicioso, a implementação da técnica é feita através de lógica combinacional e flip-flops,

sem a utilização de memória RAM. Isto faz com que o hardware seja altamente ligado à aplicação crítica, mas por outro lado, a busca por um *hash* e seu respectivo tamanho não levará mais do que um ciclo de clock.

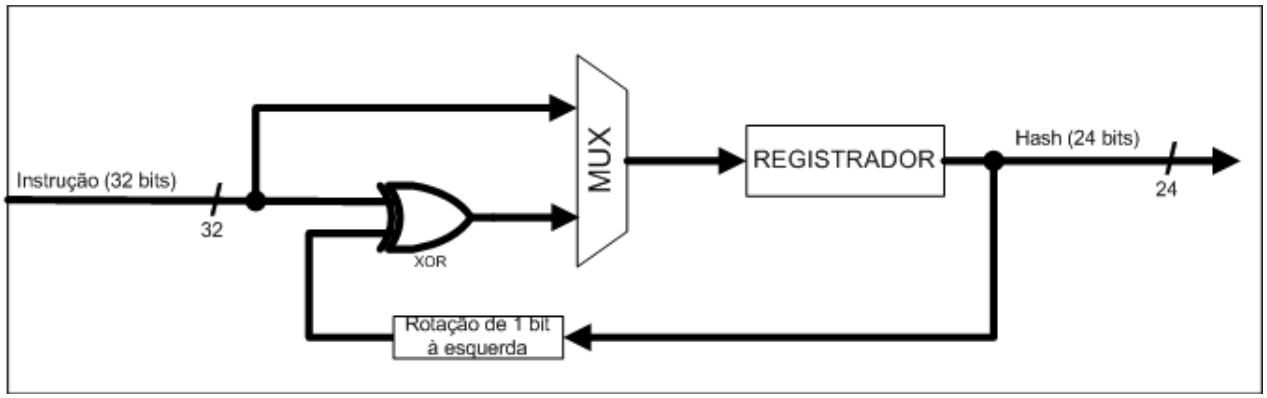
Durante a execução do programa crítico, o *Watchdog* (Figura 8) irá monitorar o endereço e a instrução que está sendo executada, através dos sinais oriundos diretamente do estágio de execução do pipeline do Leon3. O watchdog irá aguardar a execução da primeira instrução do programa na memória RAM, normalmente localizada no endereço "$0x40000000$", quando constatar que este endereço está sendo executado, imediatamente buscará na memória o *hash* para este bloco básico e o tamanho do mesmo. De posse do *hash* do bloco básico e de seu numero de instruções, o *Watchdog* passará a computar o valor do *hash* até que seja executada alguma instrução de salto (CALL, BRANCH ou JUMP), ou que o contador decrescente de instruções chegue a zero. No momento em que é executada a última instrução de um bloco básico, a Unidade de Controle do *Watchdog* irá habilitar o comparador para verificar se o *hash* gerado em tempo de execução está idêntico ao *hash* gerado previamente. Caso o *hash* previamente gerado e o *hash* gerado em tempo de execução sejam diferentes, o sinal de saída *error indication* passará para 1, onde procedimentos de recuperação poderão ser providenciados. Caso a comparação esteja correta, o endereço de destino do salto, que indica o endereço de início do bloco básico seguinte, será buscado na memória para obter o valor do *hash* e tamanho do próximo bloco básico, assim este processo se repetirá. Como cada bloco básico possui uma quantidade de instruções previamente conhecida, a cada nova instrução que está sendo executada um contador localizado junto à Unidade de Controle do *Watchdog* será decrementado, caso o contador atinja zero e não haja uma instrução de salto, o sinal de indicação de erro *error indication* passará para o valor 1, possibilitando a execução de possíveis procedimentos de recuperação. Este contador foi implementado com a presunção de que um invasor possa ter injetado instruções maliciosas e que não haja uma verificação de *hashes* até que um grande estrago já tenha sido feito. Existe um caso específico em que o contador se torna inoperante propositadamente, que é quando um bloco básico possui originalmente mais de 254 instruções, como o contador é de 8 bits não é possível armazenar números maiores que 255, logo, este valor representa a desativação do contador apenas para este bloco básico. Esta solução foi criada para que o *Watchdog* não crie uma restrição que limite o funcionamento da aplicação crítica devido ao tamanho de um bloco básico, nesses casos o contador é desabilitado até o fim do bloco básico atual.

Figura 8 – Diagrama geral da arquitetura do Watchdog



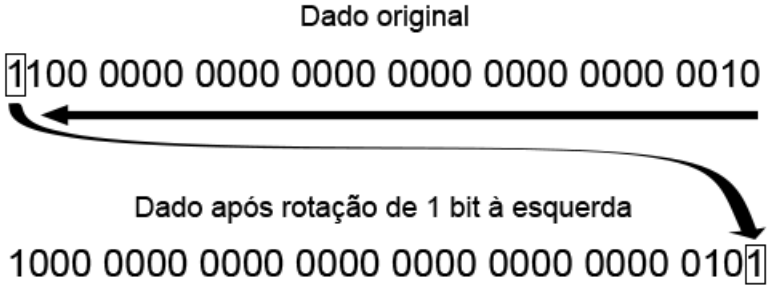
A partir da separação da área de instruções do arquivo executável de uma aplicação crítica é possível verificar quais as instruções que serão realmente executadas. Então, com base nos critérios estabelecidos durante a definição do bloco básico, serão identificados os endereços iniciais e finais de cada um dos blocos blocos básicos. Uma vez, com os blocos básicos devidamente identificados, gera-se um *hash* de 24 bits para cada um destes blocos. Os valor do *hash* é obtido através de operações numéricas e sequenciais envolvendo todas as instruções contidas no bloco, outros 8 bits serão utilizados para armazenar o número de instruções de cada um dos blocos básicos, totalizando então 32 bits para a validação da integridade de cada bloco básico. Essa operação de identificação de blocos básicos e o calculo de seus *hashes* será realizada através da execução do *software* proposto apenas uma vez antes da execução do programa e então os dados obtidos a partir do arquivo da aplicação serão armazenados em uma memória de 32 bits junto ao *Watchdog* dentro do processador. Durante a execução do programa os blocos básicos serão identificados pelo *Watchdog* e os *hashes* serão calculados, o *hash* de cada bloco deverá corresponder ao gerado antes da execução, assim como o número de instruções do bloco básico também será verificado.

Figura 9 – Parte do Watchdog responsável pelo cálculo dinâmico dos hashes em tempo de execução



A parte do hardware responsável pela geração dos *hashes* em tempo de execução está ilustrada na figura 9. Durante a execução da primeira instrução de um bloco básico, o multiplexador irá encaminhar diretamente a instrução que irá inicializar o valor do registrador de 32 bits. Todas as instruções seguintes até o final do bloco básico serão somadas logicamente através da operação "*Xor*" com o valor de saída do registrador, e então o resultado desta operação é novamente armazenado no registrador. Como característica da operação "*Xor*" a ordem dos fatores de entrada não altera o resultado de saída, e como o objetivo deste sistema é garantir a integridade de execução das instruções de um programa exatamente na mesma ordem em que foram implementadas, foi criada uma solução para este problema através da operação de rotação de um bit à esquerda, esta operação desloca todos os bits de saída do registrador um bit à esquerda, sendo que o bit mais à esquerda é reposicionado na primeira posição à direita, conforme a figura 10. Deste modo só existe uma sequência de execução das instruções de um bloco básico, e caso alguma das instruções seja executada fora da ordem do programa original, o *hash* gerado em tempo de execução para este bloco básico não corresponderá com valor o pré-calculado, ocasionando assim uma comparação de *hashes* diferentes e então será apontado o sinal de erro.

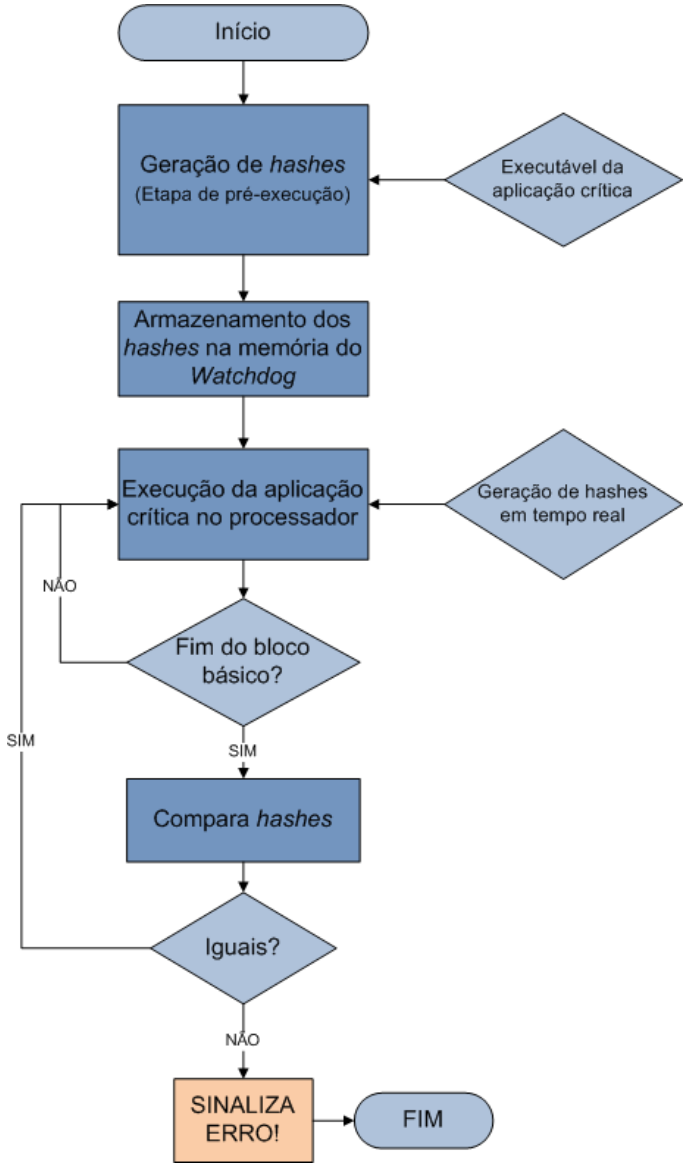
Figura 10 – Rotação de um bit à esquerda



A técnica proposta foi desenvolvida utilizando o conjunto de instruções da arquitetura SPARC-V8 que é utilizada pelo processador LEON3, esta escolha foi motivada por sua arquitetura aberta e a disponibilidade do código-fonte para fácil customização. Maiores informações sobre a arquitetura podem ser vistas na seção 2.2.

A figura 11 apresenta um fluxograma simplificado da técnica utilizada pelo *software* proposto, mas que evidencia o processo de geração de *hashes*. Em um primeiro momento são gerados hashes para todos os blocos básicos existentes em uma aplicação crítica a partir de seu arquivo executável, estes dados então são armazenados na memória do *Watchdog* e a aplicação crítica poderá ser executada diretamente na RAM do processador. Durante a execução da aplicação, o *Watchdog* irá monitorar as instruções que estão sendo executadas no núcleo do processador e irá calcular o *hash* para cada um dos blocos básicos que está sendo executado. Ao final de cada bloco básico há um momento de checagem, onde o *hash* gerado antes da execução é comparado ao *hash* gerado durante a execução da aplicação crítica, e em caso de divergência um sinal de erro deverá ser sinalizado pelo *Watchdog*. É importante ressaltar que a comparação dos *hashes* ocorre dentro do *Watchdog* em paralelo à execução da aplicação crítica, não interferindo na execução das instruções no pipeline do processador.

Figura 11 – Fluxograma do processo de geração de hashes

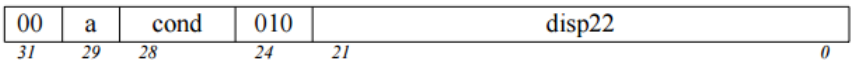


O *software* proposto é o responsável pela geração dos *hashes* e pela contagem do número de instruções de cada bloco básico na etapa de pré-execução do programa, e tem como único parâmetro de entrada o arquivo executável da aplicação crítica compilada para o LEON3. O *software* foi desenvolvido utilizando a linguagem C# e extrai as instruções do programa diretamente da seção ".text" do arquivo executável. Todos os blocos básicos desta seção serão devidamente identificados e serão computados seus respectivos *hashes* e tamanhos. Para que possam ser identificados todos os blocos básicos do programa crítico, o software proposto irá executar três tipos de "varreduras" de instruções. Na primeira varredura, as instruções são identificadas de forma sequencial do início ao fim da seção "*.text*", neste processo são identificados os blocos básicos que iniciam após as instruções de salto, e então são calculados os *hashes* e os tamanhos dos respectivos blocos básicos. Durante a segunda varredura todas as funções são mapeadas e é identificado somente o

primeiro bloco básico de cada função do programa crítico, ou seja, do endereço inicial da função até a primeira instrução de salto, deste modo é esperado que sempre que haja uma chamada de função por ponteiro o destino seja um destes blocos básicos. Na terceira varredura são identificados os saltos do tipo "*branch*" que possuem um deslocamento fixo, ou seja, o endereço de destino deste salto neste caso poderá ser em qualquer parte de uma função, até mesmo no meio de outro bloco básico, este calculo culminará na criação de outro bloco básico, agora possivelmente menor, mas em uma mesma região do programa crítico onde já há um outro bloco básico previamente mapeado. Podem haver casos em que o *software* calcule duas vezes o *hash* para um mesmo bloco básico, no caso de haver um "*branch*" com destino para o início da função, por exemplo. Neste caso o programa identifica que já foi calculado o *hash* para este bloco básico e o elimina para evitar a existência de blocos duplicados.

Instruções do tipo "*branch*" são comumente utilizados pelo compilador em laços do tipo "*for*" ou "*while*", e por sua característica percorrem por algumas vezes consecutivas um mesmo trecho de código. Deve-se notar, porém, que na arquitetura SPARC-V8 (WEAVER; GREMOND, 1992) utilizada pelo LEON3, existe um bit opcional nas instruções de "*branch*" que determina se a instrução imediatamente seguinte será ou não executada antes do salto, trata-se do bit "*a*", conforme a figura 12.

Figura 12 – Formato das instruções de branch

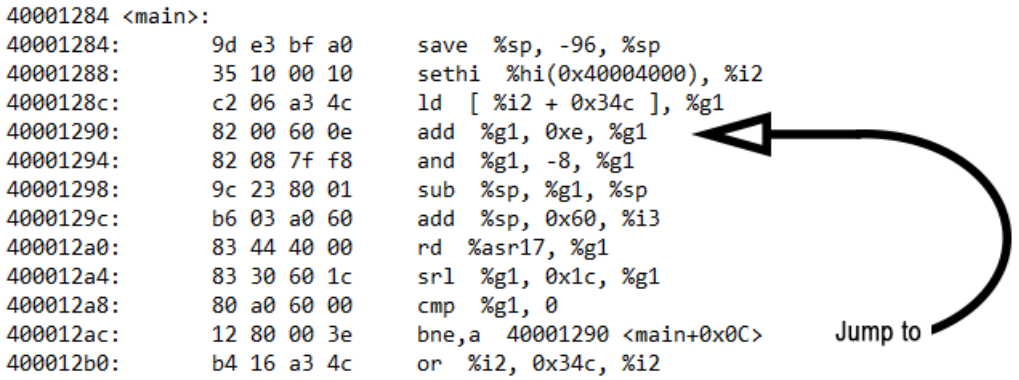


Fonte: WEAVER; GREMOND 1992

A utilização do bit "*a*" com valor um, em instruções do tipo "*branch*" não condicionais, faz com que a instrução imediatamente seguinte nunca seja executada. Já em instruções do tipo *branch* condicionais, a instrução seguinte é executada sempre que a condição de salto é verdadeira, já nos casos em que não ocorre o salto, a instrução seguinte não é executada. Para esses casos a determinação das instruções que compõem o bloco básico deverá ser cuidadosa, já que após a execução da instrução de salto que indica o fim do bloco anterior, a instrução imediatamente abaixo será executada e seguida pelas instruções no destino do salto, conforme a figura 13. Nesse caso a instrução "*Or*" que está no endereço "0x400012b0", será executada sempre que a condição de salto for verdadeira, no momento em que a instrução de salto for executada e a condição for falsa, será executada a instrução imediatamente após a instrução "*Or*".

Nos casos em que o valor do bit "*a*" é zero, ou nos outros tipos de saltos como "*Jump*" e "*Call*", a instrução seguinte sempre é executada.

Figura 13 – Exemplo de trecho de código com instrução de salto e bit "a"com valor 1



Para os casos em que o valor do bit "a" é igual a zero, o calculo do *hash* é mais simples, pois sempre leva em conta que a instrução seguinte a instrução de salto também faz parte do bloco básico, neste caso a instrução posterior à instrução de salto será a última do bloco básico. Já nos casos em que o salto é condicional e o valor do bit "a" é igual a um, a instrução de salto será a última do bloco básico e existirá um bloco básico com início na instrução imediatamente seguinte, seguido pelas instruções do endereço de destino do salto, ou seja, nesses casos haverá um bloco básico onde o endereço das instruções não é contíguo, mas isso não irá afetar o cálculo do *hash*, pois o somente o endereço da primeira instrução do bloco básico tem relevância neste processo, pois o processador executará as instruções na ordem correta.

Em um salto condicional, quando a condição de salto deixar de ser verdadeira, nos casos em que esteja sendo executado um laço, por exemplo, então o próximo bloco básico terá início na segunda instrução após a instrução de salto do bloco atual, da mesma forma de um salto não condicional em que o bit "a" é igual a um.

O monitoramento do fluxo de execução de um programa exige que exista uma etapa de coleta de dados antes da execução do programa crítico, a partir de um arquivo executável íntegro, e somente então ele poderá ser executado em um sistema que poderá ou não estar sujeito a vulnerabilidades. Nesta etapa o arquivo executável será remontado e então serão gerados os *hashes* para cada um dos possíveis blocos básicos, mesmo que nem todos sejam efetivamente executados durante a execução do programa crítico. Terminado este processo, os *hashes* e os tamanhos de cada bloco básico serão armazenados em uma memória estática combinacional, o que une fortemente o *hardware* à aplicação crítica, este modelo de memória é implementada durante a síntese da aplicação para o FPGA já que por questões de performance o mapeamento de memória é 1:1. Para obter um melhor aproveitamento do espaço em memória, foi definido que os *hashes* e os respectivos tamanhos de um bloco básico serão armazenados no mesmo endereço de memória. Portanto, apesar de os *hashes* serem computados com todos os 32 bits durante a execução do programa, apenas os 24 bits da parte menos significativa serão comparados

com o valor que está na memória estática. Os outros 8 bits armazenados junto ao *hash*, correspondem ao número de instruções do bloco básico, o que permite que hajam blocos básicos de até 254 instruções. Quando o número de instruções de um bloco básico é igual ou superior a 255 instruções, a checagem da integridade do programa crítico passa a ser apenas através do *hash* gerado. Isso pode vir a ser uma vulnerabilidade, no caso de ocorrer uma corrupção dentro de um desses blocos básicos onde não há a checagem de tamanho, sendo que somente quando ocorrer a próxima execução de um salto haverá uma checagem de integridade através da comparação de *hashes*. Isto foi necessário para que não hajam restrições na implementação de uma aplicação crítica. O *hash* e o tamanho de um bloco básico serão mapeados através do endereço inicial de cada bloco, conforme a tabela 1. A estrutura de memória do *Watchdog* será gerada apenas para os endereços de início de um bloco básico, que no exemplo da tabela 1, são os endereços 0x40000000, 0x40000010 e 0x4000001c.

Tabela 1 – Ilustração da estrutura dos blocos básicos

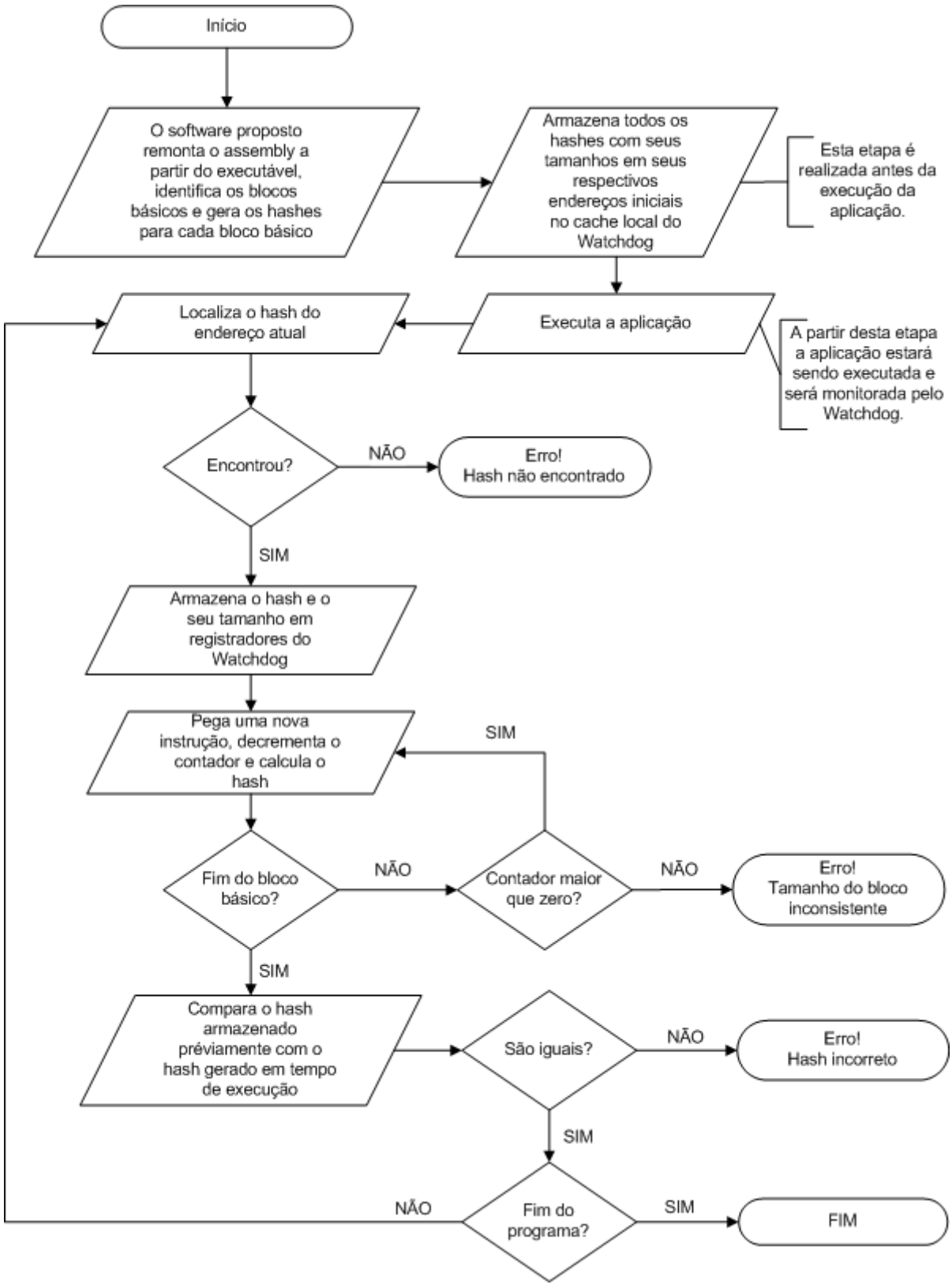
Memória de Instruções		
Índice	**Endereço**	**Instrução**
1	0x40000000	0x????????
2	0x40000004	0x????????
3	0x40000008	0x????????
4	**0x4000000c**	**JUMP**
1	0x40000010	0x????????
2	0x40000014	0x????????
3	**0x40000018**	**CALL**
1	0x4000001c	0x????????
2	**0x40000020**	**BRANCH**

Memória do Watchdog		
Endereço	**Hash**	**Tamanho**
0x40000000	**0x??????**	**4**
0x40000010	**0x??????**	**3**
0x4000001c	**0x??????**	**2**

Após o programa crítico e os *hashes* serem armazenados nas respectivas memórias do processador e do *watchdog*, a execução do sistema estará pronta para ser iniciada. Durante a execução da primeira instrução do programa crítico, geralmente localizada no endereço 0x40000000 (para o processador LEON3), será iniciado o primeiro bloco básico e o *Watchdog* se encarregará de localizar o *hash* e o tamanho do bloco neste mesmo endereço, mas agora na memória estática em que estes foram previamente armazenados junto ao *Watchdog*. Caso não seja encontrado o *hash* para este ou qualquer outro bloco básico, o *watchdog* sinalizará o erro e medidas corretivas deverão ser tomadas. Quando o *hash* e o tamanho do bloco básico são localizados corretamente, estes serão armazenados em registradores específicos do comparador de saída e ficarão disponíveis até o final do bloco básico atual. A cada nova instrução que é executada no pipeline do processador, o valor do tamanho do bloco básico é decrementado no contador do *watchdog* e o valor parcial *hash* também é calculado neste momento. Caso o valor do contador se iguale a zero antes da execução de uma instrução de salto, o sinal de erro do *Watchdog* será ativado e medidas

corretivas deverão ser tomadas. Da mesma forma, sempre que uma instrução de salto for executada o valor do contador deverá ser zero, já que uma instrução de salto poderia ser inserida maliciosamente no meio de um bloco básico, e então passaria despercebida. Assim que o bloco básico chega ao fim sem nenhum erro ser apontado, é feita a comparação do *hash* calculado em tempo de execução com o que está armazenado no comparador do *watchdog*, e caso os valores não coincidam o sinal de erro deverá ser apontado para que possam ser adotadas as medidas corretivas. Caso o programa não termine, a próxima instrução a ser executada será a instrução inicial do bloco básico seguinte e o processo descrito acima, conforme visto na figura 14, deverá recomeçar.

Figura 14 – Fluxograma de execução da técnica



4 Validação da Técnica Proposta

Este capítulo apresenta os casos de validação utilizados na verificação da capacidade de detecção de ataques da técnica proposta. Para isso foram utilizadas três abordagens, a primeira delas utilizando *buffer overflow* para corromper o endereço de retorno de uma função, a segunda abordagem é feita via alteração das instruções do código crítico diretamente na memória RAM do sistema. Na terceira abordagem, a manipulação do valor do endereço de retorno de uma função armazenada em um ponteiro.

4.1 Buffer Overflow

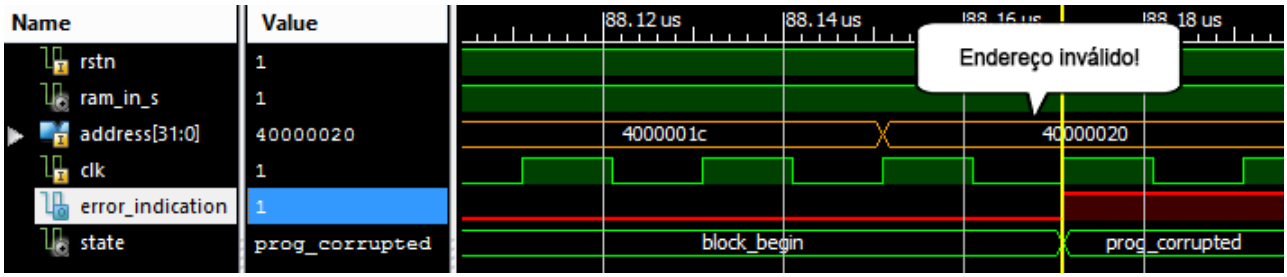
Estouro de buffer (*buffer overflow*) é uma das vulnerabilidades mais exploradas em *software* atualmente para a quebra de segurança de sistemas, segundo o grupo norte americano Computer Emergency Readiness Team (CERT). O trecho de código abaixo apresenta um caso de estouro de buffer, o programa inicia na função *main*, que imediatamente executa um salto para a função *writeBuffer*, onde o vetor *buff* é declarado com oito posições e através da função *memset* são escritos 112 bytes que excedem o tamanho do vetor, contendo o valor *0xff* e logo após são escritos mais quatro bytes que irão sobrescrever exatamente o endereço de retorno armazenado na pilha do programa, que anteriormente era de *0x40000040* e agora passou a ser *0x40000020*, fazendo com que o sistema continue sendo executado de um ponto inesperado do programa.

```
void writeBuffer(void)
{
        char buff[8];
        memset(buff, 0xff, 112);
        return;
}

void *memset(void *s, int c, size_t n)
{
    unsigned char* p=s;
    while(n--)
        *p++ = (unsigned char)c;
    *p++ = (unsigned char) 0x40;
    *p++ = (unsigned char) 0x00;
    *p++ = (unsigned char) 0x00;
```

```
    *p++ = (unsigned char) 0x20;
    return s;
}

int main(void){
        writeBuffer();
}
```

Neste caso como o endereço *0x40000020* não possui nenhum bloco básico associado, o sinal indicando o erro é apontado com o atraso de um ciclo de clock após a execução da instrução de salto. A figura 15 apresenta o trecho da simulação que demonstra a detecção do salto com destino inesperado.

Figura 15 – Detecção de estouro de buffer



4.2 Corrupção do Código Crítico Diretamente em RAM

Esta técnica utilizada na validação simula um ataque do tipo *DMA Attack*, descrito na seção 2.1. Neste caso a memória de instruções do programa é armazenada em um arquivo de texto que o simulador lê e executa, podendo assim ser facilmente manipulado para a validação da técnica.

O trecho abaixo contém as instruções de um bloco básico com 21 instruções que inicia no endereço *0x40000018* e vai até o endereço *0x40000068*. O tempo de execução deste bloco básico é de exatamente 364 ciclos de clock e o objetivo deste teste é saber qual o melhor caso e o pior caso de atraso na detecção do erro.

O pior caso ocorre quando há uma corrupção na primeira instrução do bloco básico, neste caso a instrução *save*, pois somente após a execução da última instrução do bloco básico será feita a comparação do *hash* para a constatação do problema. Assim, como a primeira instrução demora 10 ciclos de clock para ser executada e ainda restam 354 ciclos para terminar o bloco básico e ainda é necessário mais um ciclo para que seja feita a comparação dos *hashes*, neste caso o erro só será detectado 355 ciclos de clock após a instrução corrompida ter sido executada.

O melhor caso irá ocorrer quando o erro estiver na última instrução do bloco básico, neste caso o erro será sinalizado apenas um ciclo de clock após a execução da instrução.

```
40000018 <main>:
40000018:       9d e3 bf 90     save  %sp, -112, %sp
4000001c:       82 10 24 00     mov  0x400, %g1
40000020:       c2 27 bf f4     st  %g1, [ %fp + -12 ]
40000024:       c2 07 bf f4     ld  [ %fp + -12 ], %g1
40000028:       82 00 60 01     inc  %g1
4000002c:       c2 27 bf f4     st  %g1, [ %fp + -12 ]
40000030:       c2 07 bf f4     ld  [ %fp + -12 ], %g1
40000034:       82 00 7f ff     add  %g1, -1, %g1
40000038:       c2 27 bf f4     st  %g1, [ %fp + -12 ]
4000003c:       c2 07 bf f4     ld  [ %fp + -12 ], %g1
40000040:       82 00 60 01     inc  %g1
40000044:       c2 27 bf f4     st  %g1, [ %fp + -12 ]
40000048:       c2 07 bf f4     ld  [ %fp + -12 ], %g1
4000004c:       82 00 7f ff     add  %g1, -1, %g1
40000050:       c2 27 bf f4     st  %g1, [ %fp + -12 ]
40000054:       c2 07 bf f4     ld  [ %fp + -12 ], %g1
40000058:       82 00 60 01     inc  %g1
4000005c:       c2 27 bf f4     st  %g1, [ %fp + -12 ]
40000060:       c2 07 bf f4     ld  [ %fp + -12 ], %g1
40000064:       82 00 7f ff     add  %g1, -1, %g1
40000068:       c2 27 bf f4     st  %g1, [ %fp + -12 ]
```

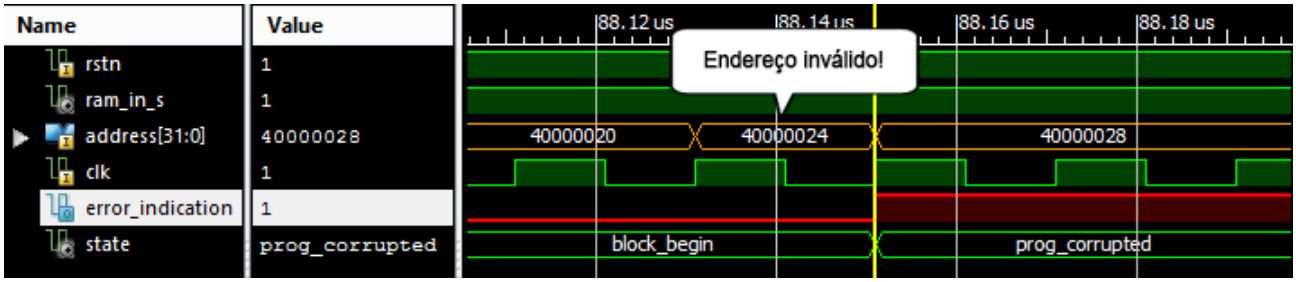
4.3 Manipulação do Endereço de Retorno de Função Armazenado em Ponteiro

O trecho de código abaixo utiliza um ponteiro para a chamada da função *soma*. O ponteiro neste caso chamado de *somaPtr* é atribuído dentro da função *main* com o valor do endereço da função *soma*, a instrução seguinte simula uma corrupção no valor deste endereço, fazendo com que o programa passe a ser executado no endereço *0x40000024* onde não se inicia um bloco básico, neste caso o erro é apontado após um ciclo de clock, conforme a figura 16.

```
int (*somaPtr)(int,int);

int soma(int x, int y) {
```

```
    return x+y;
}

int main(void){
        int sum;
        somaPtr = &soma;
        somaPtr = 0x40000024;    //ponteiro explicitamente corrompido.
        sum = (*somaPtr)(2, 3);
}
```

Figura 16 – Ponteiro de função corrompido



5 Avaliação da Técnica Proposta

A avaliação da técnica proposta foi realizada em duas etapas: primeiramente foram utilizados trechos de código de programas (*benchmarks*) *open source* para avaliar a capacidade de detecção de ataques em situações vulneráveis conhecidas, após foram calculadas os *overheads* de desempenho e de área agregados a partir da sua implementação.

5.1 Configuração Experimental

Para realizar as simulações, o hardware do *Watchdog* foi implementado em VHDL e seu modelo comportamental foi sintetizado através do pacote de aplicativos *ISE Design Suite* da *Xilinx*. Desde modo o modelo VHDL do processador LEON3 com o *Watchdog* pôde ser emulado, permitindo a visualização dos sinais que comprovaram o seu correto funcionamento, através do *software* ISim, que também faz parte do pacote de de aplicativos da Xilinx.

Para a avaliação dos *overheads* foram utilizados dois métodos: primeiramente para a verificação do *overhead* de desempenho foi utilizada a síntese comportamental que permitiu constatar visualmente o tempo de detecção em ciclos de *clock*. Para a verificação do *overhead* de área ocasionado pela inclusão do *Watchdog* junto ao processador LEON3, foi utilizada a ferramenta *PlanAhead* da *Xilinx* para a realização da síntese física junto ao FPGA Xilinx Spartan3E.

5.2 Avaliação dos Benchmarks

Como forma de tornar as análises mais realistas foram obtidos alguns trechos de código em C expondo vulnerabilidades reais que são publicadas pelo site *Common Vulnerabilities and Exposures* (CVE, 2017). O site funciona como um arquivo virtual de vulnerabilidades e facilita o compartilhamento de informações para o desenvolvimento de novas ferramentas de segurança virtual.

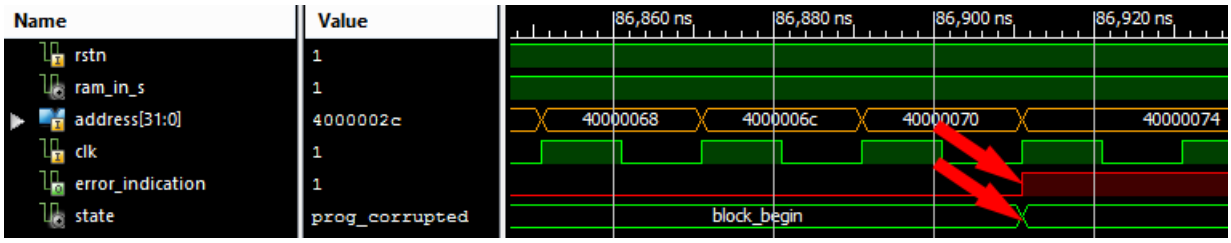
Para execução dos testes o processo é bastante simples, o trecho de código em C é incluído em um arquivo modelo e é compilado para o processador LEON3. Nos casos de sucesso um sinal de erro deverá ser apontado pelo *Watchdog*, para que possam ser adotadas as medidas corretivas. A figura 17 apresenta o momento da simulação em que o *Watchdog* detecta um *buffer overflow* que ocasionou a sobrescrita de um endereço de retorno de uma função causada por uma vulnerabilidade conhecida do programa Edbrowse. Neste momento o sinal *error indication* passa para 1 e o sinal interno do *Watchdog* chamado

Tabela 2 – Trechos de Código Vulneráveis Publicados pelo CVE

Programas Vulneráveis	**Número CVE**	**Graus de Severidade**	**Resultados**
Edbrowse	CVE-2006-6909	10,0 alta	Erro Sinalizado pelo Watchdog
MADWiFi	CVE-2006-6332	7,5 alta	Erro Sinalizado pelo Watchdog
OpenSER	CVE-2006-6749	9,3 alta	Erro Sinalizado pelo Watchdog
Samba	CVE-2007-0453	4,6 média	Erro Sinalizado pelo Watchdog
Sendmail	CVE-2003-0681	7,5 alta	Erro Sinalizado pelo Watchdog
Wu-ftpd	CVE-1999-0368	10,0 alta	Erro Sinalizado pelo Watchdog
Wu-ftpd	CVE-2003-0466	10,0 alta	Erro Sinalizado pelo Watchdog

state passa para o valor *prog corrupted*. Esta vulnerabilidade foi descoberta em 2006 e permitia que usuários mal intencionados executassem códigos arbitrários em um servidor FTP.

Figura 17 – Momento da simulação em que é gerado o sinal de exceção para o caso Edbrowse



Na tabela 2 é possível observar os exemplos utilizados na avaliação da técnica. Na primeira coluna estão os programas que em algum momento foram expostos à vulnerabilidade, seu número CVE que é um identificador único, utilizado para compartilhar as informações através de diferentes bases de dados, seu grau de severidade, conforme a *National Vulnerability Database* que classifica a relevância dos códigos vulneráveis e finalmente a última coluna indicando o sucesso da aplicação da técnica, apontando as corrupções corretamente em todas as vulnerabilidades testadas.

5.3 Limitações de Cobertura da Técnica Proposta

Esta seção descreve as condições para as quais a cobertura de detecção de tentativas de intrusão é limitada, seja por motivos de *software* nativo (*bootloader*) ou por perfil do código monitorado (que possui restrições em blocos básico de 255 instruções ou mais). Em mais detalhes:

- As partes do sistema que ficam armazenadas fora da memória RAM, que geralmente são trechos de código responsáveis por iniciar a execução do programa principal (*bootloader*), estão armazenados em uma memória estática e não estão sendo monitorados durante a sua execução. Neste caso existe a presunção de que o código é carregado no hardware do sistema crítico através de um sistema íntegro, já que este hardware estará fisicamente mais vulnerável quando realmente estiver sendo utilizado.

- O funcionamento do hardware proposto está condicionado a aplicações que são executadas em um único núcleo do processador, pois sistemas críticos geralmente utilizam este modelo de arquitetura. Para que fosse possível monitorar aplicações em sistemas multi-core deveriam ser feitas alterações no sistema, como por exemplo a inclusão de um hardware junto ao barramento de instruções para o monitoramento dos sinais de execução de cada núcleo individualmente.

- A técnica propõe que somente sejam executados blocos básicos íntegros, mas caso haja uma intrusão onde o objetivo do invasor seja executar repetidamente um dos blocos básicos sem modificá-lo, então a intrusão passará despercebida pelo *Watchdog* proposto.

- Casos em que os blocos básicos possuam muitas instruções poderão implicar em uma latência de detecção bastante longa, tendo em vista que a integridade só será verificada após a execução da instrução de salto ou quando o contador de instruções chegar a zero.

- O valor armazenado em memória que representa o tamanho de um bloco básico é de 8 bits, logo, caso um bloco básico possua mais de 255 instruções, este valor excederá a capacidade máxima representável por 8 bits, neste caso o *hash* será verificado normalmente, mas o contador do tamanho do bloco estará desativado.

- O *software* de geração de hashes durante a fase de pré-execução da aplicação crítica foi projetado para montar os blocos básicos de acordo com o conjunto de instruções da arquitetura SPARC-V8, a utilização do *Watchdog* em outro processador com um conjunto de instruções diferente exigirá uma readaptação do sistema para a desmontagem aplicação de outro tipo de arquitetura.

5.4 Overheads

O uso desta técnica não apresenta maiores custos de desempenho, tendo em vista que a execução do *Watchdog* ocorre em paralelo à execução do programa. A menos que seja levado em conta que para isso a cache, MMU e FPU foram desabilitados. Quando ao acréscimo de área do *Watchdog* em relação ao processador, foi utilizada a ferramenta de CAD PlanAhead para sintetizar o processador com o *Watchdog* na FPGA Xilinx Spartan3E, e a partir desta síntese foram obtidos os dados que podem ser observados na tabela 3.

Tabela 3 – Acréscimo de Área do Watchdog

Primitivas	**Leon + Watchdog**	**Watchdog**	**Acréscimo de área [%]**
FLOP_LATCH	2649	135	5,1%
LUT	8026	109	1,36%
MUXFX	663	4	0,6%
IO	117	0	0%
DMEM	13	0	0%
Total	**11468**	**248**	**2,16%**

Como é possível observar o acréscimo total devido a inclusão do *Watchdog* junto ao núcleo do processador é de apenas 2,16%, mas cabe ressaltar que a o acréscimo de área referente a memória combinacional que irá armazenar os *hashes* não está incluso, pois o seu tamanho está diretamente relacionado ao número de blocos básicos existentes em um determinado código crítico, ou seja, quanto mais *hashes* houverem para ser armazenados, maior será o espaço ocupado pelas mesmas. Para verificar o acréscimo de área ocasionado por uma aplicação, foi utilizado o exemplo de *buffer overflow*, descrito na seção 4.1. Neste caso esta pequena aplicação possui apenas cinco blocos básicos e conforme pode ser observado na tabela 4 possui um acréscimo de área total de 2,64%.

Tabela 4 – Acréscimo de Área do Watchdog Incluindo Memória de Hashes

Primitivas	**Leon + Watchdog + Memória**	**Watchdog + Memória**	**Acréscimo de área [%]**
FLOP_LATCH	2667	153	5,74%
LUT	8064	147	1,82%
MUXFX	663	4	0,6%
IO	117	0	0%
DMEM	13	0	0%
Total	**11524**	**304**	**2,64%**

5.5 Discussão Comparativa com o Estado-da-Arte

Conforme os casos vistos na seção 2.3, as principais técnicas existentes para o controle de fluxo de execução de uma aplicação crítica, utilizam como base o código fonte da mesma para a geração e armazenamento de *hashes*, deste modo ocasionando seus devidos *overheads* de desempenho e de área. O que este trabalho propõe é uma nova abordagem para o controle de fluxo de execução, com um tempo de detecção quase que instantâneo, levando apenas um ciclo de *clock* para o melhor caso. A limitação desta nova proposta está justamente no seu modo de implementação do *hardware* de memória, que por ser combinacional está fortemente ligada à aplicação crítica que está sendo executada, fazendo-se necessária uma nova síntese para que outra aplicação possa ser executada. Diferentemente das outras técnicas apresentadas que não demonstram vínculo ao *hardware* que está sendo utilizado. A principal vantagem de se utilizar a técnica proposta está na dispensa de necessidade de posse do código fonte, de recompilação de código ou de montagem do CFG. Este modelo apresenta uma capacidade de detecção de ataques igual ou superior às outras técnicas mencionadas e com um tempo de sinalização menor. Parte desta vantagem de desempenho ocorre pois não são realizadas duplicação de dados de registradores, por exemplo, para que sejam adotadas medidas de restauração em casos de detecção de ataques. Neste caso a adoção de técnicas que permitam a ação de medidas corretivas devem ser avaliadas e poderão em alguns casos ocasionar um acréscimo no *overhead* de desempenho da aplicação.

6 Considerações Finais

Este trabalho apresentou uma nova abordagem baseada em hardware para a proteção de sistemas. A técnica monitora o fluxo de execução do código da aplicação com o intuito de detectar eventuais ataques. Dentre os seus diferenciais está a possibilidade de detecção de ataques que alterem o destino de saltos que estejam armazenados em registradores, o que torna o seu destino imprevisível antes da execução do programa. Note que outras técnicas existentes na literatura utilizam grafos que indicam os possíveis caminhos que o programa poderia seguir, mas nestes casos a criação de grafos se torna impossível, uma vez que seu destino é imprevisível. A técnica proposta tem ainda como diferencial o fato de dispensar qualquer tipo de recompilação de código ou mesmo a posse do código-fonte da aplicação crítica.

O *Watchdog* implementado monitora as instruções que estão sendo executadas, gera *hashes* para os blocos básicos e compara com *hashes* gerados antes da execução do programa crítico. Resultados experimentais demonstraram que esta abordagem teve sucesso, uma vez que foi capaz de detectar corretamente todos os ataques considerados durante os experimentos. A abordagem apresentou baixo acréscimo de área (2,61% para o caso analisado; onde foi adotado o LEON3 soft-core mapeado em FPGA Xilinx Spartan3E, sem a inclusão da memória de *hashes*), e não apresentou degradação de performance, já que o *Watchdog* é operado em paralelo com a execução do processador. Por outro lado, a latência na detecção do ataque depende da complexidade do código do usuário, uma vez que a verificação da integridade do código ocorre apenas ao final de um bloco básico. Em outras palavras, caso o ponto afetado esteja no fim do bloco básico, poderá ocorrer então que o tempo de detecção do erro demore algumas dezenas de ciclos de clock.

6.1 Trabalhos Futuros

Para o futuro, prevê-se a inclusão de uma ação de recuperação em casos de erro, já que atualmente o erro é sinalizado mas o programa segue sendo executado. Para que a performance de execução da aplicação tenha seu desempenho melhorado deverá ser desenvolvida uma versão que funcione em múltiplos núcleos de modo que os *hashes* pré-computados possam ser lidos de qualquer um dos núcleos e assim possa ser feita a checagem de integridade. A ativação da memória cache faz com que algumas instruções sejam adiantadas durante a execução. Em relação à sequência de instruções da aplicação crítica, o que torna o cálculo dos *hashes* ainda mais complexo. Entretanto, convém mencionar que a sua implementação possibilita um aumento do desempenho da aplicação crítica, já que atualmente a busca dos dados e instruções é feita diretamente na memória RAM. Em

tempo, para ampliação da possibilidade de utilização da técnica em outras arquiteturas é necessária a tomada de algumas ações para que o *software* que gera os *hashes* antes da execução do programa possa reconhecer este novo conjunto de instruções, para assim poder identificar os blocos básicos.

Referências

ABADI, M. et al. Control-flow integrity. In: ACM. *Proceedings of the 12th ACM conference on Computer and communications security*. [S.l.], 2005. p. 340–353. Citado na página 22.

ABADI, M. et al. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, ACM, v. 13, n. 1, p. 4, 2009. Citado na página 24.

ANDERSSON, J.; GAISLER, J.; WEIGAND, R. Next generation multipurpose microprocessor. In: *Int. Conf. on Data Systems in Aerospace (DASIA), Hungary*. [S.l.: s.n.], 2010. Citado na página 17.

BLASS, E.-O.; ROBERTSON, W. Tresor-hunt: attacking cpu-bound encryption. In: ACM. *Proceedings of the 28th Annual Computer Security Applications Conference*. [S.l.], 2012. p. 71–78. Citado na página 16.

COBHAM GAISLER. *LEON3 Processor*. 2016. Acessado em: Março de 2016. Disponível em: <http://gaisler.com/index.php/products/processors/leon3>. Citado na página 14.

COWAN, C. et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: *Usenix Security*. [S.l.: s.n.], 1998. v. 98, p. 63–78. Citado 2 vezes nas páginas 14 e 16.

CVE. *Common Vulnerabilities and Exposures*. 2017. Acessado em: Fevereiro de 2017. Disponível em: <https://cve.mitre.org/>. Citado 3 vezes nas páginas 5, 13 e 45.

EE TIMES. *Free Sparc processor developer goes commercial*. 2005. Acessado em: Janeiro de 2017. Disponível em: <http://www.eetimes.com/document.asp?doc_id=1195625>. Citado na página 17.

FISKIRAN, A. M.; LEE, R. B. Runtime execution monitoring (rem) to detect and prevent malicious code execution. In: IEEE. *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*. [S.l.], 2004. p. 452–457. Citado na página 24.

GAISLER, J. et al. Grlib ip core user's manual. *Gaisler research*, 2007. Citado na página 18.

GELBART, O. et al. Codesseal: Compiler/fpga approach to secure applications. *Intelligence and Security Informatics*, Springer, p. 1345–1354, 2005. Citado na página 24.

HALDERMAN, J. A. et al. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, ACM, v. 52, n. 5, p. 91–98, 2009. Citado na página 16.

KANUPARTHI, A.; KARRI, R. Reliable integrity checking in multicore processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, v. 12, n. 2, p. 10, 2015. Citado na página 22.

KANUPARTHI, A.; RAJENDRAN, J.; KARRI, R. Controlling your control flow graph. In: IEEE. *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. [S.l.], 2016. p. 43–48. Citado 2 vezes nas páginas 21 e 23.

KANUPARTHI, A. K. et al. A high-performance, low-overhead microarchitecture for secure program execution. In: IEEE. *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. [S.l.], 2012. p. 102–107. Citado 4 vezes nas páginas 12, 24, 25 e 26.

KANUPARTHI, A. K.; ZAHRAN, M.; KARRI, R. Feasibility study of dynamic trusted platform module. In: IEEE. *Computer Design (ICCD), 2010 IEEE International Conference on*. [S.l.], 2010. p. 350–355. Citado na página 24.

KIRIANSKY, V. et al. Secure execution via program shepherding. In: *USENIX Security Symposium*. [S.l.: s.n.], 2002. v. 92, p. 84. Citado na página 24.

KIROVSKI, D.; DRINIĆ, M.; POTKONJAK, M. Enabling trusted software integrity. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2002. v. 37, n. 10, p. 108–120. Citado na página 24.

LEW, K. S.; DILLON, T. S.; FORWARD, K. E. Software complexity and its impact on software reliability. *Software Engineering, IEEE Transactions on*, IEEE, v. 14, n. 11, p. 1645–1655, 1988. Citado na página 14.

LI, J.; TAN, Q.; XU, J. Reconstructing control flow graph for control flow checking. In: IEEE. *Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on*. [S.l.], 2010. v. 1, p. 527–531. Citado na página 20.

RILEY, R.; JIANG, X.; XU, D. An architectural approach to preventing code injection attacks. *Dependable and Secure Computing, IEEE Transactions on*, IEEE, v. 7, n. 4, p. 351–365, 2010. Citado na página 16.

SCHUETTE, M.; SHEN, J. P. et al. Processor control flow monitoring using signatured instruction streams. *Computers, IEEE Transactions on*, IEEE, v. 100, n. 3, p. 264–276, 1987. Citado 3 vezes nas páginas 20, 21 e 24.

SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: ACM. *Proceedings of the 14th ACM conference on Computer and communications security*. [S.l.], 2007. p. 552–561. Citado na página 22.

SOMMERVILLE, I. et al. *Engenharia de software*. [S.l.]: Addison Wesley São Paulo, 2003. v. 6. Citado na página 12.

THE WALL STREET JOURNAL. *Cobham Buys Aeroflex in 920 Million Dollars Deal*. 2014. Acessado em: Março de 2016. Disponível em: <https://www.wsj.com/articles/SB10001424052702304422704579573194156815228>. Citado na página 17.

WEAVER, D. L.; GREMOND, T. *The SPARC architecture manual*. [S.l.]: PTR Prentice Hall Englewood Cliffs, NJ 07632, 1992. Citado 3 vezes nas páginas 17, 18 e 36.

APÊNDICE A – Código-Fonte do Software de Geração de Hashes

```
using ELFSharp.ELF;
using ELFSharp.ELF.Sections;
using ELFSharp.ELF.Segments;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace sectionExtractor
{
  class Program
  {
    private static SortedDictionary<UInt32, String> basicBlocksTable = new
        SortedDictionary<UInt32, String>();
    private static System.IO.StreamWriter file;
    static void Main(string[] args)
    {
      var elf = ELFReader.Load<uint>("inputProgram.exe");
      string appPath = new Uri(System.IO.Path.GetDirectoryName(System.
          Reflection.Assembly.GetExecutingAssembly().GetName().CodeBase)).
          LocalPath;
      using (file =
          new System.IO.StreamWriter(appPath + @"\hashGenerator.txt"))
      {
        var functions = elf.GetSection(".symtab") as SymbolTable<uint>;
        SortedDictionary<UInt32, String> sectionTable = new
            SortedDictionary<UInt32, String>();
        uint initialAddress = elf.GetSection(".text").LoadAddress;
        uint finalAddress = elf.GetSection(".text").Size + initialAddress;
        foreach (var f in functions.Entries)
        {
          if (f.Name != "" && f.Value > 0 && f.PointedSectionIndex == 1)
```

```
  {
    if (!sectionTable.ContainsKey((UInt32)f.Value))
      sectionTable.Add((UInt32)f.Value, f.Name);
  }
}
var segments = elf.Segments.First().GetContents();
UInt32 instruction;
UInt32 hash=0;
uint offset, endset, endblock=0;
uint anterior = 0;
uint count = 0;
sectionTable.Remove(sectionTable.Keys.Last());
for (uint id = 0; id < (finalAddress-initialAddress); id++)
{
  if (id == 0 || ((id + 1) % 4 == 0))
  {
    if (id == 0)
    {
      instruction = ReverseBytes(BitConverter.ToUInt32(segments, (
          int)id));
      hash = instruction;
      count = 1;
    }
    else
    {
      instruction = ReverseBytes(BitConverter.ToUInt32(segments, (
          int)id + 1));
      hash = instruction ^ ((hash << 1) | (hash >> 31));
      count++;
    }
    if ((instruction >> 31 & 1) == 1 &&
      (instruction >> 30 & 1) == 0 &&
      (instruction >> 24 & 1) == 1 &&
      (instruction >> 23 & 1) == 1 &&
      (instruction >> 22 & 1) == 1 &&
      (instruction >> 21 & 1) == 0 &&
      (instruction >> 20 & 1) == 0)
      {
        if ((instruction >> 19 & 1) == 0) //JMP
```

```
    {
      instruction = ReverseBytes(BitConverter.ToUInt32(segments
          , (int)id + 5));
      hash = instruction ^ ((hash << 1) | (hash >> 31));
      count++;
      if (count > 254) count = 0xff;
      basicBlocksTable.Add((UInt32)(initialAddress + endblock),
          String.Format("{0} {1}", (hash & 0xffffff).ToString("
          X6"), count.ToString("X2")));
      endblock = id + 9;
      hash = 0;
      count = 0;
      id += 4; // jump 1 instruction
    }
    else
    {
      instruction = ReverseBytes(BitConverter.ToUInt32(segments
          , (int)id + 5));
      hash = instruction ^ ((hash << 1) | (hash >> 31));
      count++;
      if (count > 254) count = 0xff;
      basicBlocksTable.Add((UInt32)(initialAddress + endblock),
          String.Format("{0} {1}", (hash & 0xffffff).ToString("
          X6"), count.ToString("X2")));
      file.WriteLine("{0} {1} {2}", (endblock + initialAddress)
          .ToString("X8"), (hash & 0xffffff).ToString("X6"),
          count.ToString("X2"));
      endblock = id + 9;
      hash = 0;
      count = 0;
      id += 4; // jump 1 instruction
    }
  }
  else if ((instruction >> 31 & 1) == 0 &&
      (instruction >> 30 & 1) == 1) //CALL
  {
    file.WriteLine("{0} {1} - CALL", (offset + id + 1 +
        initialAddress).ToString("X8"), instruction_reverse.
        ToString("X8"));
```

```
  instruction = ReverseBytes(BitConverter.ToUInt32(segments,
     (int)id + 5));
  hash = instruction ^ ((hash << 1) | (hash >> 31));
  count++;
  if (count > 254) count = 0xff;
  basicBlocksTable.Add((UInt32)(initialAddress + endblock),
     String.Format("{0} {1}", (hash & 0xffffff).ToString("X6
     "), count.ToString("X2")));
  file.WriteLine("{0} {1} {2}", (endblock + initialAddress).
     ToString("X8"), (hash & 0xffffff).ToString("X6"), count.
     ToString("X2"));
  endblock = id + 9;
  hash = 0;
  count = 0;
  id += 4; // jump 1 instruction
}
else if ((instruction >> 31 & 1) == 0 &&
     (instruction >> 30 & 1) == 0 &&
    (instruction >> 24 & 1) == 0 &&
    (instruction >> 23 & 1) == 1 &&
    (instruction >> 22 & 1) == 0)//BRANCH
{
  instruction = ReverseBytes(BitConverter.ToUInt32(segments,
     (int)id + 5));
  hash = instruction ^ ((hash << 1) | (hash >> 31));
  count++;
  if (count > 254) count = 0xff;
  basicBlocksTable.Add((UInt32)(initialAddress + endblock),
     String.Format("{0} {1}", (hash & 0xffffff).ToString("X6
     "), count.ToString("X2")));
  file.WriteLine("{0} {1} {2}", (endblock + initialAddress).
     ToString("X8"), (hash & 0xffffff).ToString("X6"), count.
     ToString("X2"));
  endblock = id + 9;
  hash = 0;
  count = 0;
  id += 4; // jump 1 instruction
  file.WriteLine("{0} {1} - BRANCH to {2}", (offset + id + 1
     + initialAddress).ToString("X8"), instruction_reverse.
```

```
            ToString("X8"), shift_address.ToString("X8"));
        }
        else
        {
          file.WriteLine("{0} {1} NO JUMP", (offset + id + 1 +
              initialAddress).ToString("X8"), instruction.ToString("X8
              "));
        }

    }
}

anterior = 0;
LOOP 2 -> este for pega o primeiro bloco basico de cada section
for (int sectionIterator = 0; sectionIterator < sectionTable.Count
    ; sectionIterator++)
{
    offset = sectionTable.ElementAt(sectionIterator).Key -
        initialAddress;
    if (sectionIterator < sectionTable.Count - 1)//if not the last
        section
      endset = sectionTable.ElementAt(sectionIterator + 1).Key -
          initialAddress;
    else//if the last section
      endset = finalAddress - initialAddress;
    if (offset != anterior)
    {
      Console.WriteLine("Falha de continuidade!");
      Console.ReadKey();
    }
    anterior = endset;
    instruction = ReverseBytes(BitConverter.ToUInt32(segments, (int)
        offset));
    Console.WriteLine(ReverseBytes(instruction).ToString("X8"));
    file.WriteLine("{0} {1}", (offset + initialAddress).ToString("X8
        "), ReverseBytes(instruction).ToString("X8"));
    for (uint id = 0; id < finalAddress; id++) //id < finaladress ->
         dont stop when section end
    {
```

```
Console.Write(segments[id + offset].ToString("X2") + " ");
if (id == 0 || ((id + 1) % 4 == 0))
{
  if (id == 0)
  {
    hash = instruction;
    count = 1;
  }
  Console.WriteLine("");
  else
  {
    instruction = ReverseBytes(BitConverter.ToUInt32(segments,
       (int)offset + (int)id + 1));
    hash = instruction ^ ((hash << 1) | (hash >> 31));
    count++;
  }
  verifify the type of the instruction
  if ((instruction >> 31 & 1) == 1 &&
    (instruction >> 30 & 1) == 0 &&
    (instruction >> 24 & 1) == 1 &&
    (instruction >> 23 & 1) == 1 &&
    (instruction >> 22 & 1) == 1 &&
    (instruction >> 21 & 1) == 0 &&
    (instruction >> 20 & 1) == 0)
  {
    if ((instruction >> 19 & 1) == 0) //JMP
    {
      instruction = ReverseBytes(BitConverter.ToUInt32(segments
         , (int)offset + (int)id + 5));
      hash = instruction ^ ((hash << 1) | (hash >> 31));
      count++;
      if (count > 254) count = 0xff;
      if (!basicBlocksTable.ContainsKey((UInt32)(offset +
         initialAddress)))
      {
        file.WriteLine("{0} {1} {2}", (offset + initialAddress)
           .ToString("X8"), (hash & 0xffffff).ToString("X6"),
           count.ToString("X2"));
```

```
        basicBlocksTable.Add((UInt32)(offset + initialAddress),
            String.Format("{0} {1}", (hash & 0xffffff).ToString
           ("X6"), count.ToString("X2")));
      }
      id = finalAddress; //trick to get out loop
    }
    else //RET
    {
      instruction = ReverseBytes(BitConverter.ToUInt32(segments
         , (int)offset + (int)id + 5));
      hash = instruction ^ ((hash << 1) | (hash >> 31));
      count++;
      if (count > 254) count = 0xff;
      if (!basicBlocksTable.ContainsKey((UInt32)(offset +
         initialAddress)))
      {
        file.WriteLine("{0} {1} {2}", (offset + initialAddress)
           .ToString("X8"), (hash & 0xffffff).ToString("X6"),
           count.ToString("X2"));
        basicBlocksTable.Add((UInt32)(offset + initialAddress),
            String.Format("{0} {1}", (hash & 0xffffff).ToString
           ("X6"), count.ToString("X2")));
      }
      id = finalAddress; //trick to get out loop
    }
  }
  else if ((instruction >> 31 & 1) == 0 &&
       (instruction >> 30 & 1) == 1) //CALL
  {
    instruction = ReverseBytes(BitConverter.ToUInt32(segments,
       (int)offset + (int)id + 5));
    hash = instruction ^ ((hash << 1) | (hash >> 31));
    count++;
    if (count > 254) count = 0xff;
    if (!basicBlocksTable.ContainsKey((UInt32)(offset +
       initialAddress)))
    {
      file.WriteLine("{0} {1} {2}", (offset + initialAddress).
         ToString("X8"), (hash & 0xffffff).ToString("X6"),
```

```
                count.ToString("X2"));
            basicBlocksTable.Add((UInt32)(offset + initialAddress),
                String.Format("{0} {1}", (hash & 0xffffff).ToString("
                X6"), count.ToString("X2")));
          }
          id = finalAddress; //trick to get out loop
        }
        else if ((instruction >> 31 & 1) == 0 &&
            (instruction >> 30 & 1) == 0 &&
            (instruction >> 24 & 1) == 0 &&
            (instruction >> 23 & 1) == 1 &&
            (instruction >> 22 & 1) == 0)//BRANCH
        {
          instruction = ReverseBytes(BitConverter.ToUInt32(segments,
              (int)offset + (int)id + 5));
          hash = instruction ^ ((hash << 1) | (hash >> 31));
          count++;
          if (count > 254) count = 0xff;
          if (!basicBlocksTable.ContainsKey((UInt32)(offset +
              initialAddress)))
          {
            file.WriteLine("{0} {1} {2}", (offset + initialAddress).
                ToString("X8"), (hash & 0xffffff).ToString("X6"),
                count.ToString("X2"));
            basicBlocksTable.Add((UInt32)(offset + initialAddress),
                String.Format("{0} {1}", (hash & 0xffffff).ToString("
                X6"), count.ToString("X2")));
          }
          id = finalAddress; //trick to get out loop
        }
        else
        {
          file.WriteLine("{0} {1} NO JUMP", (offset + id + 1 +
              initialAddress).ToString("X8"), instruction.ToString("X8
              "));
        }
      }
    }
  }
```

```
for (uint id = 0; id < (finalAddress - initialAddress); id++)
{
  if (id == 0 || ((id + 1) % 4 == 0))
  {
    if (id == 0)
    {
      instruction = ReverseBytes(BitConverter.ToUInt32(segments, (
          int)id));
    }
    else
    {
      instruction = ReverseBytes(BitConverter.ToUInt32(segments, (
          int)id + 1));
    }
    if ((instruction >> 31 & 1) == 0 &&
      (instruction >> 30 & 1) == 0 &&
      (instruction >> 24 & 1) == 0 &&
      (instruction >> 23 & 1) == 1 &&
      (instruction >> 22 & 1) == 0) //BRANCH
    {
      int branchAddress = signExtension((int)instruction)+(int)id
          +1;
      for (uint id2 = (uint)branchAddress; id2 < (finalAddress -
          initialAddress); id2++)
      {
        if (id2 == branchAddress)//if first inst of block
        {
          instruction = ReverseBytes(BitConverter.ToUInt32(segments
              , (int)id2));
          hash = instruction;
          count = 1;
        }
        else
        {
          instruction = ReverseBytes(BitConverter.ToUInt32(segments
              , (int)id2 + 1));
          hash = instruction ^ ((hash << 1) | (hash >> 31));
          count++;
        }
```

```
if ((instruction >> 31 & 1) == 1 &&
  (instruction >> 30 & 1) == 0 &&
  (instruction >> 24 & 1) == 1 &&
  (instruction >> 23 & 1) == 1 &&
  (instruction >> 22 & 1) == 1 &&
  (instruction >> 21 & 1) == 0 &&
  (instruction >> 20 & 1) == 0)
{
  if ((instruction >> 19 & 1) == 0) //JMP
  {
    instruction = ReverseBytes(BitConverter.ToUInt32(
       segments, (int)id2 + 5));
    hash = instruction ^ ((hash << 1) | (hash >> 31));
    count++;
    if (count > 254) count = 0xff;
    if (!basicBlocksTable.ContainsKey((UInt32)(
       branchAddress + initialAddress)))
    {
      file.WriteLine("{0} {1} {2}", (offset +
         initialAddress).ToString("X8"), (hash & 0xffffff).
         ToString("X6"), count.ToString("X2"));
      basicBlocksTable.Add((UInt32)(branchAddress +
         initialAddress), String.Format("{0} {1}", (hash &
         0xffffff).ToString("X6"), count.ToString("X2")));
    }
    id2 = finalAddress; //trick to get out loop
  }
  else //RET
  {
    instruction = ReverseBytes(BitConverter.ToUInt32(
       segments, (int)id2 + 5));
    hash = instruction ^ ((hash << 1) | (hash >> 31));
    count++;
    if (count > 254) count = 0xff;
    if (!basicBlocksTable.ContainsKey((UInt32)(
       branchAddress + initialAddress)))
    {
      file.WriteLine("{0} {1} {2}", (offset +
         initialAddress).ToString("X8"), (hash & 0xffffff).
```

```
            ToString("X6"), count.ToString("X2"));
        basicBlocksTable.Add((UInt32)(branchAddress +
            initialAddress), String.Format("{0} {1}", (hash &
            0xffffff).ToString("X6"), count.ToString("X2")));
      }
      id2 = finalAddress; //trick to get out loop
    }
  }
  else if ((instruction >> 31 & 1) == 0 &&
       (instruction >> 30 & 1) == 1) //CALL
  {
    instruction = ReverseBytes(BitConverter.ToUInt32(segments
        , (int)id2 + 5));
    hash = instruction ^ ((hash << 1) | (hash >> 31));
    count++;
    if (count > 254) count = 0xff;
    if (!basicBlocksTable.ContainsKey((UInt32)(branchAddress
        + initialAddress)))
    {
      file.WriteLine("{0} {1} {2}", (offset + initialAddress)
          .ToString("X8"), (hash & 0xffffff).ToString("X6"),
          count.ToString("X2"));
      basicBlocksTable.Add((UInt32)(branchAddress +
          initialAddress), String.Format("{0} {1}", (hash & 0
          xffffff).ToString("X6"), count.ToString("X2")));
    }
    id2 = finalAddress; //trick to get out loop
  }
  else if ((instruction >> 31 & 1) == 0 &&
       (instruction >> 30 & 1) == 0 &&
       (instruction >> 24 & 1) == 0 &&
       (instruction >> 23 & 1) == 1 &&
       (instruction >> 22 & 1) == 0)//BRANCH
  {
    instruction = ReverseBytes(BitConverter.ToUInt32(segments
        , (int)id2 + 5));
    hash = instruction ^ ((hash << 1) | (hash >> 31));
    count++;
    if (count > 254) count = 0xff;
```

```
            if (!basicBlocksTable.ContainsKey((UInt32)(branchAddress
                + initialAddress)))
            {
              file.WriteLine("{0} {1} {2}", (offset + initialAddress)
                  .ToString("X8"), (hash & 0xffffff).ToString("X6"),
                  count.ToString("X2"));
              basicBlocksTable.Add((UInt32)(branchAddress +
                  initialAddress), String.Format("{0} {1}", (hash & 0
                  xffffff).ToString("X6"), count.ToString("X2")));
            }
            id2 = finalAddress; //trick to get out loop
          }
          else
          {
            file.WriteLine("{0} {1} NO JUMP", (offset + id + 1 +
                initialAddress).ToString("X8"), instruction.ToString("
                X8"));
          }
        }
      }
      else
      {
        file.WriteLine("{0} {1} NO JUMP", (offset + id + 1 +
            initialAddress).ToString("X8"), instruction.ToString("X8
            "));
      }
    }
  }
  writeHashes();
 }
}
public static UInt32 ReverseBytes(UInt32 value)
{
 return (value & 0x000000FFU) << 24 | (value & 0x0000FF00U) << 8 |
   (value & 0x00FF0000U) >> 8 | (value & 0xFF000000U) >> 24;
}
public static int signExtension(int instr)
{
 int value = (0x003FFFFF & instr);
```

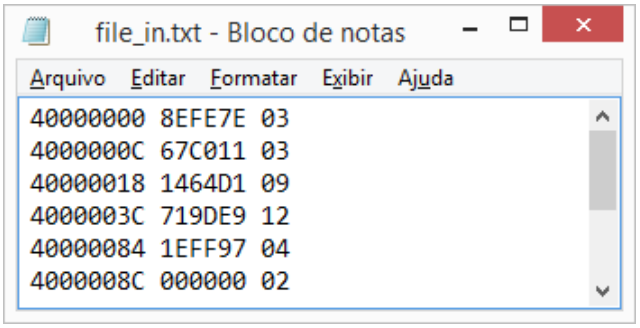
```
      int mask = 0x00200000;
      if ((mask & instr) == mask)
      {
        value += unchecked((int)0xFFC00000);
      }
      return value*4;
    }
    public static void writeHashes()
    {
      foreach (var entry in basicBlocksTable) //imprime a lista de sessoes
      {
        Console.WriteLine("{0} {1}", entry.Key.ToString("X8"), entry.Value
            );
        file.WriteLine("{0} {1}", entry.Key.ToString("X8"), entry.Value);
      }
    }
  }
}
```

APÊNDICE B – Manual de Uso da Ferramenta

Para iniciar o processo de execução da técnica o usuário estar utilizando o Sistema Operacional Windows 7 ou superior e então deverá colocar o arquivo executável da aplicação crítica na mesma pasta do programa de extração de *hashes*, chamado *sectionExtractor.exe*.

O *sectionExtractor.exe* ao ser executado irá procurar pelo arquivo da aplicação crítica que deverá se chamar *systest.exe*, caso não esteja com este nome o arquivo deverá ser renomeado. Então o *sectionExtractor.exe* identificará a seção da aplicação chamada *.text*, que contém as instruções da aplicação que serão armazenadas na memória RAM do processador. Após, iniciará a identificação dos blocos básicos da aplicação crítica e guardará os três valores que serão armazenados na memória do *Watchdog*, são eles: endereço inicial do bloco básico, *hash* de 24 bits e o número de instruções do bloco básico com 8 bits. Estes valores serão armazenados em um arquivo de texto contendo três colunas, estando separados por um espaço, que irá chamar *file_in.txt* e será salvo na pasta atual. A figura 18 demonstra um exemplo do arquivo de saída do programa *sectionExtractor.exe*.

Figura 18 – Exemplo do arquivo de saída do programa *sectionExtractor.exe*.



De posse do arquivo texto contendo estas informações, deverá ser executado o aplicativo chamado *vhdlGenerator.exe* que irá gerar um arquivo em linguagem VHDL contendo a memória estática de hashes e tamanhos em seus respectivos endereços iniciais. Este arquivo irá se chamar *hash_lenght_rom.vhd*, e terá sua estrutura conforme apresentado na figura 19. Com o arquivo VHDL gerado corretamente contendo as informações que serão armazenadas junto ao Watchdog, a síntese estará pronta para ser iniciada.

Figura 19 – Exemplo do arquivo VHDL de saída do programa *vhdlGenerator.exe*.

hash_length_rom.vhd - Bloco de notas

Arquivo Editar Formatar Exibir Ajuda

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

entity hash_length_rom is
port (
  clk         : in   std_logic;
  en          : in   std_logic;
  address     : in   std_logic_vector(31 downto 0);
  hash        : out  std_logic_vector(23 downto 0);
  length      : out  std_logic_vector(31 downto 24)
);
end;

architecture rtl of hash_length_rom is

begin

  -- hash rom
  process (clk) begin
    if (rising_edge(clk)) then
      if (en = '1') then
        case (address) is
          when X"40000000" => hash   <= X"8efe7e";
                              length <= X"03";
          when X"4000000c" => hash   <= X"67c011";
                              length <= X"03";
          when X"40000018" => hash   <= X"1464d1";
                              length <= X"09";
          when X"4000003c" => hash   <= X"719de9";
                              length <= X"12";
          when X"40000084" => hash   <= X"1eff97";
                              length <= X"04";
          when X"4000008c" => hash   <= X"000000";
                              length <= X"02";
          when others =>      hash   <= X"000000";
                              length <= X"00";
        end case;
      end if;
    end if;
  end process;

end;
```

APÊNDICE C – Modificação no LEON3 para a Inclusão do Watchdog

```
\\iu3.vhd
       use gaisler.libwatchdog.all;
       port ( ...
       watchdog_error : out std_logic
   );
       signal watchdog_address : std_logic_vector(31 downto 0);

begin

watchdog_address <= r.e.ctrl.pc & "00";
-- watchdog
  watch : watchdog
       port map
       (clk,
       rstn,
       not r.e.ctrl.annul,
       r.e.ctrl.inst, watchdog_address,
       watchdog_error);

\\libwatchdog.vhd

library ieee;
use ieee.std_logic_1164.all;

package libwatchdog is

  component watchdog is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;
    en : in std_ulogic;
    instruction : in std_logic_vector(31 downto 0);
    address : in std_logic_vector(31 downto 0);
```

```
    error_indication : out std_ulogic
  );
  end component;

  component controlunit is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;
    en : in std_ulogic;
    instruction : in std_logic_vector(31 downto 0);
    address : in std_logic_vector(31 downto 0);
    rom_length : in std_logic_vector(31 downto 24);
    rom_hash : in std_logic_vector(23 downto 0);
    runtime_hash : in std_logic_vector(23 downto 0);
    mux_sel : out std_logic;
    ff_en : out std_logic;
    rom_en : out std_logic;
    error_indication : out std_ulogic
  );
  end component;

  component hash_length_rom is
  port (
    clk : in std_logic;
    en : in std_logic;
    address : in std_logic_vector(31 downto 0);
    hash : out std_logic_vector(23 downto 0);
    length : out std_logic_vector(31 downto 24)
  );
  end component;

end package;

\\watchdog.vhd

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

```
library gaisler;
use gaisler.libwatchdog.all;

entity watchdog is

port (
  clk : in std_ulogic;
  rstn : in std_ulogic;
  en : in std_ulogic;
  instruction : in std_logic_vector(31 downto 0);
  address : in std_logic_vector(31 downto 0);
  error_indication : out std_ulogic
  );

end;

architecture rtl of watchdog is

  signal mux_sel : std_ulogic;
  signal ff_en : std_ulogic;
  signal rom_en : std_ulogic;

  signal mux_out : std_logic_vector(31 downto 0) := X"00000000";
  signal xor_out : std_logic_vector(31 downto 0) := X"00000000";
  signal ff_out : std_logic_vector(31 downto 0) := X"00000000";

  signal last_intruction_s : std_logic_vector(31 downto 0);
  signal last_address_s : std_logic_vector(31 downto 0);

  signal rom_hash_s : std_logic_vector(23 downto 0);
  signal rom_length_s : std_logic_vector(31 downto 24);

begin

  -- control unit
  cu : controlunit
    port map
    (clk,
    rstn,
```

```
  en,
  instruction,
  address,
  rom_length_s,
  rom_hash_s,
  xor_out(23 downto 0),
  mux_sel,
  ff_en,
  rom_en,
  error_indication);

-- last intruction
last_intruction_u : process(clk)
begin
  if (rising_edge(clk)) then
    last_intruction_s <= instruction;
  end if;
end process;

-- last address
last_address_u : process(clk)
begin
  if (rising_edge(clk)) then
    last_address_s <= address;
  end if;
end process;

--MUX
mux_out <= last_intruction_s when (mux_sel = '0') else xor_out;

--XOR
xor_out <= last_intruction_s xor std_logic_vector(unsigned(ff_out) rol
   1);

--FF
process (clk) begin
  if (rstn = '0') then
    ff_out <= X"00000000";
  elsif (rising_edge(clk)) then
```

```
      if (ff_en = '1') then
        ff_out <= mux_out;
      end if;
    end if;
  end process;

  hash_length_rom_u : hash_length_rom
    port map (
      clk => clk,
      en => rom_en,
      address => address,
      hash => rom_hash_s,
      length => rom_length_s
    );

end;

\\controlunit.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity controlunit is
port (
    clk : in std_ulogic;
    rstn : in std_ulogic;
    en : in std_ulogic;
    instruction : in std_logic_vector(31 downto 0);
    address : in std_logic_vector(31 downto 0);
    rom_length : in std_logic_vector(31 downto 24);
    rom_hash : in std_logic_vector(23 downto 0);
    runtime_hash : in std_logic_vector(23 downto 0);
    mux_sel : out std_logic;
    ff_en : out std_logic;
    rom_en : out std_logic;
    error_indication : out std_ulogic
  );
end;
```

```
architecture rtl of controlunit is

  signal mux_out : std_logic_vector(31 downto 0) := X"00000000";
  signal xor_out : std_logic_vector(31 downto 0) := X"00000000";
  signal ff_out : std_logic_vector(31 downto 0) := X"00000000";

  type state_type is (
    NOT_RAM,
    BLOCK_BEGIN,
    HASH_ENABLE,
    DELAY_SLOT,
    PROG_CORRUPTED
  );
  signal state, state_next: state_type;

  type instruction_type is (
    BRANCH,
    CALL,
    JUMP,
        RET,
        OTHER
  );
  signal instruction_current: instruction_type;

  signal last_address_s : std_logic_vector(31 downto 0);
  signal last_instruction_s : std_logic_vector(31 downto 0);
  signal rom_hash_s : std_logic_vector(23 downto 0);
  signal rom_length_s : std_logic_vector(07 downto 0);
  signal counter_s : std_logic_vector(07 downto 0);
  signal new_address_s : std_logic;
  signal delay_slot_instruction_s : std_logic;
  signal hash_cmp_s : std_logic;
  signal en_s : std_logic;
  signal ram_s : std_logic;
  signal ram_in_s : std_logic;

--instruction identification signals
  signal a : std_logic;
```

```
  signal cond : std_logic_vector(3 downto 0);
  signal op2 : std_logic_vector(2 downto 0);
  signal op3 : std_logic_vector(5 downto 0);
  signal disp22 : std_logic_vector(21 downto 0);
  signal disp30 : std_logic_vector(29 downto 0);

begin

  counter_u : process(clk)
  begin
    if (rising_edge(clk)) then
      if (rstn = '0') then
        counter_s <= (others => '0');
      else
        if new_address_s = '1' and state_next /= HASH_ENABLE then
          counter_s <= X"00";
        elsif en_s = '1' and new_address_s = '1' and state /= NOT_RAM then
          counter_s <= std_logic_vector(unsigned(counter_s) + 1);
        end if;
      end if;
    end if;
  end process;

  last_addrress_u : process(clk)
  begin
    if (rising_edge(clk)) then
      last_address_s <= address;
    end if;
  end process;

  last_instruction_u : process(clk)
  begin
    if (rising_edge(clk)) then
      last_instruction_s <= instruction;
    end if;
  end process;

  en_register_u : process(clk)
  begin
```

```
  if (rising_edge(clk)) then
    en_s <= en;
  end if;
end process;

ram_in_s <= '0' when
(ram_s = '0' and ram_in_s = '0')
or rstn = '0' else '1'; -- '1' before the program started

rom_hash_s <= rom_hash when state = BLOCK_BEGIN;
rom_length_s <= rom_length when state = BLOCK_BEGIN;

new_address_s <= '1' when last_address_s /= address else '0';

hash_cmp_s <= '1' when rom_hash_s = runtime_hash else '0';

delay_slot_instruction_s <= '1'; --TODO

ram_s <= '1' when address(31 downto 30) = "01" else '0';

state_logic_u : process (clk)
begin
  if (rising_edge(clk)) then
    if (rstn = '0') then
      state <= BLOCK_BEGIN;
    else
            if ram_s = '1' and en_s = '1' and new_address_s = '1' then
       state <= state_next;
            elsif ram_s = '0'
            and en_s = '1'
            and new_address_s = '1'
            and ram_in_s = '1' then
       state <= PROG_CORRUPTED;
      end if;
    end if;
  end if;
end process;

state_next_logic_u : process
```

```
(state,
instruction,
address,
new_address_s,
instruction_current,
rom_length,
rom_length_s)
begin
  case (state) is

        when BLOCK_BEGIN =>
              if rom_length_s = X"00" then
        state_next <= PROG_CORRUPTED;
      elsif rom_length_s = X"01" then
        state_next <= BLOCK_BEGIN;
      else
        state_next <= HASH_ENABLE;
      end if;

    when HASH_ENABLE =>
              if counter_s = std_logic_vector(unsigned(rom_length_s) - 1)
              and hash_cmp_s = '1' then
        state_next <= BLOCK_BEGIN;
      elsif counter_s = std_logic_vector(unsigned(rom_length_s) - 1)
      and hash_cmp_s = '0' then
        state_next <= PROG_CORRUPTED;
      else
        state_next <= HASH_ENABLE;
      end if;

    when DELAY_SLOT =>
      state_next <= BLOCK_BEGIN;

    when PROG_CORRUPTED =>
      state_next <= PROG_CORRUPTED;

    when others =>
      state_next <= PROG_CORRUPTED;
```

```
  end case;
end process;


-- rom_en
rom_en <= '1' when en = '1' else '0';

outputs_logic_u : process (state, new_address_s)
begin
  case (state) is
    when NOT_RAM =>
      mux_sel <= '1';
      ff_en <= '0';
      error_indication <= '0';

    when BLOCK_BEGIN =>
      mux_sel <= '0';
      ff_en <= new_address_s and en_s;
      error_indication <= '0';

    when HASH_ENABLE =>
      mux_sel <= '1';
      ff_en <= new_address_s and en_s;
      error_indication <= '0';

    when DELAY_SLOT =>
      mux_sel <= '1';
      ff_en <= new_address_s and en_s;
      error_indication <= '0';

    when PROG_CORRUPTED =>
      mux_sel <= '0';
      ff_en <= '0';
      error_indication <= '1';

    when others =>
      mux_sel <= '0';
      ff_en <= '0';
      error_indication <= '0';
```

```
    end case;
  end process;

  set_op : process (instruction)
  begin
    case (instruction(31 downto 30)) is
      when "00" =>
    a <= instruction(29);
    cond <= instruction(28 downto 25);
    op2 <= instruction(24 downto 22);
    disp22 <= instruction(21 downto 0);
    if op2 = "010" then
      instruction_current <= BRANCH;
    else
      instruction_current <= OTHER;
    end if;

      when "01" =>
        instruction_current <= CALL;
    disp30 <= instruction(29 downto 0);

    when "10" =>
    op3 <= instruction(24 downto 19);
    case (op3) is
      when "111000" =>
        instruction_current <= JUMP;
      when "111001" =>
        instruction_current <= RET;
      when others =>
        instruction_current <= OTHER;
    end case;

      when others =>
        instruction_current <= OTHER;
    end case;
  end process;

end;
```

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria Acadêmica
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: proacad@pucrs.br
Site: www.pucrs.br/proacad