

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

EDUARDO CHIELLE

**CFT-tool: Ferramenta Configurável para
Aplicação de Técnicas de Detecção de Falhas
em Processadores por Software**

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

Prof. Dr. Fernanda Lima Kastensmidt
Orientadora

Porto Alegre, fevereiro de 2012.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Chielle, Eduardo

CFT-tool: Ferramenta Configurável para Aplicação de Técnicas de Detecção de Falhas em Processadores por *Software* / Eduardo Chielle. – Porto Alegre: Programa de Pós-Graduação em Computação, 2012.

104 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2012. Orientadora: Fernanda Lima Kastensmidt.

1.Técnicas de detecção em *software*. 2. Processadores. 3. Injeção de Falhas. I. Kastensmidt, Fernanda G. L.. II. CFT-tool: Ferramenta Configurável para Aplicação de Técnicas de Detecção de Falhas em Processadores por *Software*.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenador do PPGC: Prof. Álvaro Freitas Moreira

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Quero agradecer aos meus pais, Márcia e Roque, e irmãos, Daniel, Douglas e Gustavo, que apesar de estarem longe, sempre se fizeram presentes.

Agradeço também ao pessoal do laboratório e demais laboratórios da redondeza, Anelise, Ângelo, Carol, Duda, Felipe (Tonin), Jimmy, Jorge, José Eduardo, José Rodrigo, Lucas Rosa, Lucas Tambara, Maurício, Paulo, Raul Barth, Raul Chipana, Samuel e William por terem contribuído, ou não, para o bom andamento do mestrado.

Ao pessoal oriundo da EC11 André Zangado e Rodrigo Stilera e aos adjacentes Bruno Piteco e Tiago Pinta, pelo apoio tático e momentos de descontração. Aos irmãos ninjas alemães, Adriana e Angel Tateishi, pelo tererê nosso de cada dia. E, também, aos amigos Cleber, Cicero, Giane e Lê.

Não posso esquecer a professora Fernanda, que me deu a oportunidade e a orientação necessária para a concretização deste trabalho.

Por fim, agradeço aos demais amigos de Porto Alegre, Foz do Iguaçu e outras andanças por onde passei.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS.....	7
LISTA DE FIGURAS.....	8
LISTA DE TABELAS.....	10
RESUMO.....	11
ABSTRACT.....	12
1 INTRODUÇÃO.....	13
2 CONCEITOS DE DETECÇÃO DE ERROS EM SOFTWARE.....	17
2.1 Técnicas de Detecção de Erros em Software.....	17
2.1.1 Técnicas para proteção dos dados.....	17
2.1.2 Técnicas para proteção do controle.....	18
2.1.3 Técnicas híbridas.....	20
2.2 Programas que Implementam Técnicas de Detecção em Software.....	22
2.2.1 Proteção em linguagem de alto nível.....	22
2.2.2 Proteção em linguagem assembly.....	22
2.2.3 Proteção em código de máquina.....	22
3 CFT-TOOL: FERRAMENTA PARA APLICAÇÃO DE TÉCNICAS DE PROTEÇÃO EM SOFTWARE PARA DIFERENTES PROCESSADORES.....	23
3.1 Módulo de Configuração.....	25
3.1.1 Configurações gerais.....	26
3.1.2 Configurações das instruções.....	27
3.1.3 Configurações dos registradores.....	29
3.1.4 Configurações do memory dump.....	29
3.1.5 Configurações das técnicas.....	31
3.2 Módulo de Verificação.....	33
3.3 Módulo de Complementação.....	34
3.3.1 Identificação de sub-rotinas não presentes no código pré-ligação.....	36
3.3.2 Conversão de código do formato memory dump para assembly.....	37

3.4	Módulo de Proteção a Falhas Transientes.....	39
3.5	Técnicas de Detecção em Software Implementadas	40
3.5.1	Variáveis 1	40
3.5.2	Variáveis 2	41
3.5.3	Variáveis 3	42
3.5.4	Branches	42
3.5.5	Assinaturas.....	43
4	RESULTADOS E VALIDAÇÃO	45
4.1	Processadores Utilizados.....	45
4.1.1	miniMIPS.....	45
4.1.2	LEON3.....	47
4.2	Metodologia de Injeção de Falhas.....	51
4.3	Metodologia de Classificação de Erros.....	53
4.4	Validação.....	56
4.4.1	Programas selecionados para validação	56
4.4.2	Validação das aplicação das técnicas de detecção em software pela CFT-tool	58
4.4.3	Validação da configurabilidade da ferramenta	67
5	AVALIAÇÃO DO USO SELETIVO DOS REGISTRADORES.....	74
5.1	Métodos	74
5.1.1	Método Static First (SF)	75
5.1.2	Método Static Target (ST)	75
5.1.3	Método Static Source (SS).....	75
5.1.4	Método Static Source and Target (SST)	75
5.1.5	Método Dynamic Target (DT)	75
5.1.6	Método Dynamic Source (DS).....	75
5.1.7	Método Dynamic Source and Target (DST)	75
5.2	Uso dos Registradores	76
5.3	Tempo de Execução e Ocupação de Memória	78
5.4	Resultados de Injeção de Falhas	81
6	CONCLUSÃO E TRABALHOS FUTUROS	86
	REFERÊNCIAS.....	88
	ANEXO A - DETALHES TÉCNICOS DA IMPLEMENTAÇÃO	90
	Classe CFT.....	90
	Classe AssemblyGenerator	90
	Classe CodeGenerator.....	91
	Classe Architecture	91
	Classe ConfigParser.....	92
	Classe DB	92
	Classe Loader.....	92
	Classe Techniques.....	93
	Classe ErrorManager	93

Classe Manager.....	93
Classe ErrorTreatment	93
Classe BRA.....	94
Classe SIG.....	94
Classe VAR1.....	95
Classe VAR2.....	95
Classe VAR3.....	95
Classe Global	96
Classe Instruction.....	96
Classe Register.....	96
Classe Util.....	96
Classe Checker.....	97
Classe Format	97

ANEXO B - ARTIGOS 98

LISTA DE ABREVIATURAS E SIGLAS

BID	Basic Block Identifier
BRA	Branches (Desvios Condicionais)
CCA	Control Flow Checking using Assertions
CFCSS	Control Flow Checking by Software Signatures
CFID	Control Flow Identifier
CFT	Configurable Fault Tolerant
CISC	Complex Instruction Set Computer
DSP	Digital Signal Processor
ECCA	Enhanced Control Flow Checking using Assertions
EDAC	Error Detection And Correction
EDDI	Error Detection by Duplicated Instructions
HPCT	Hardening Post Compiling Translator
LATW	Latin American Test Workshop
PC	Program Counter
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SEE	Single Events Effects
SET	Single Event Transient
SEU	Single Event Upset
SIG	Assinaturas
SPARC	Scalable Processor Architecture
SWIFT	Software Implemented Fault Tolerance
TEA2	TETRA Encryption Algorithm 2
VAR1	Variáveis 1
VAR2	Variáveis 2
VAR3	Variáveis 3
VHDL	VHSIC Hardware Description Language

LISTA DE FIGURAS

<i>Figura 1.1: Níveis de abstração e processos de transformação do programa.</i>	15
<i>Figura 2.1: Aplicação da técnica EDDI (REIS, 2005).</i>	18
<i>Figura 2.2: Exemplo da técnica CCA (ALKHALIFA, 1999).</i>	19
<i>Figura 3.1: Visão geral do funcionamento da CFT-tool.</i>	23
<i>Figura 3.2: Processo de aplicação da proteção pela ferramenta CFT-tool.</i>	24
<i>Figura 3.3: Exemplo da configuração do formato do label.</i>	26
<i>Figura 3.4: Exemplo de instrução que deve ser declarada nas configurações gerais devido às suas funcionalidades.</i>	26
<i>Figura 3.5: Configuração do branch delay slot para quando as instruções são reordenadas no código assembly.</i>	27
<i>Figura 3.6: Configuração dos desvios condicionais logicamente inversos.</i>	27
<i>Figura 3.7: Exemplo de formato de instrução.</i>	28
<i>Figura 3.8: Exemplo de grupo de instruções.</i>	28
<i>Figura 3.9: Exemplo de grupo de instruções com instruções sobrecarregadas.</i>	28
<i>Figura 3.10: Exemplo de grupo de registradores.</i>	29
<i>Figura 3.11: Exemplo do formato de label no memory dump.</i>	29
<i>Figura 3.12: Exemplo do formato de instruções no memory dump.</i>	30
<i>Figura 3.13: Nome dos registradores no formato memory dump (à esquerda) e nome equivalente em assembly (à direita).</i>	30
<i>Figura 3.14: Nome das instruções no formato memory dump (à esquerda) e nome equivalente em assembly (à direita).</i>	31
<i>Figura 3.15: Configurações dos mnemônicos das instruções de salto incondicional para registrador, salto incondicional para endereço de memória e desvios condicionais.</i>	31
<i>Figura 3.16: Técnicas de detecção em software selecionadas.</i>	32
<i>Figura 3.17: Técnicas de detecção em software selecionadas.</i>	32
<i>Figura 3.18: Exemplo de configuração do modo de seleção de registradores.</i>	33
<i>Figura 3.19: Tag de comentário configurada (acima) e linha de código com comentário (abaixo).</i>	33
<i>Figura 3.20: Instrução analisada (à esquerda) e formato esperado para a instrução (à direita).</i>	34
<i>Figura 3.21: Trecho de código no formato memory dump.</i>	35
<i>Figura 3.22: Exemplo de chamada para sub-rotina.</i>	36
<i>Figura 3.23: Labels presentes no código assembly.</i>	36
<i>Figura 3.24: Exemplo de sub-rotina não descrita no código pré ligação.</i>	37
<i>Figura 3.25: Sub-rotina extraída da descrição no formato memory dump (à esquerda) e código assembly equivalente gerado pelo módulo de complementação (à direita).</i>	38
<i>Figura 3.26: Visão geral do módulo de proteção a SEEs.</i>	39
<i>Figura 3.27: Código original (à esquerda) e código com a técnica VAR1 (à direita).</i>	40
<i>Figura 3.28: Código original (à esquerda) e código com a técnica VAR2 (à direita).</i>	41
<i>Figura 3.29: À esquerda o código original e à direita o código com a técnica VAR3.</i>	42
<i>Figura 3.30: À esquerda o código original e à direita o código com a técnica BRA.</i>	43
<i>Figura 3.31: Código original (à esquerda) e código com a técnica BRA (à direita).</i>	44
<i>Figura 4.1: Comparação e desvio no processador miniMIPS.</i>	46
<i>Figura 4.2: Registradores do processador miniMIPS.</i>	46
<i>Figura 4.3: Transformações do programa para execução no processador miniMIPS.</i>	47
<i>Figura 4.4: Instrução de comparação para o processador LEON3.</i>	48
<i>Figura 4.5: Instrução de desvio condicional para o processador LEON3.</i>	48
<i>Figura 4.6: Relação dos registradores do processador LEON3 (SPARC, 1991).</i>	48

<i>Figura 4.7: Janela de registradores do processador LEON3 (SPARC, 1991).</i>	49
<i>Figura 4.8: Fluxo de compilação para execução no processador LEON3.</i>	50
<i>Figura 4.9: Formato SREC.</i>	51
<i>Figura 4.10: Inserção de falhas em nível lógico através do comando force.</i>	52
<i>Figura 4.11: Avaliação do valor do sinal pelo comando examine.</i>	52
<i>Figura 4.12: Atribuição do valor do sinal a uma variável.</i>	52
<i>Figura 4.13: Visualização de parte do conteúdo da memória.</i>	52
<i>Figura 4.14: Atribuição de parte do conteúdo da memória a uma variável.</i>	53
<i>Figura 4.15: PC da execução base, sem falhas (à esquerda) e PC da execução com erro, não detectado, nos dados (à direita).</i>	54
<i>Figura 4.16: PC da execução base, sem falhas (à esquerda) e PC da execução com erro, detectado, nos dados (à direita).</i>	55
<i>Figura 4.17: PC da execução base, sem falhas (à esquerda) e PC da execução com erro, não detectado, no controle (à direita).</i>	55
<i>Figura 4.18: PC da execução base, sem falhas (à esquerda) e PC da execução com erro, detectado, no controle (à direita).</i>	56
<i>Figura 4.19: Código, em C, do programa de multiplicação de matrizes.</i>	57
<i>Figura 4.20: Código, em C, do algoritmo de ordenação bubble sort.</i>	57
<i>Figura 4.21: Código, em C, do algoritmo encriptação.</i>	58
<i>Figura 4.22: Tempos de execução para a multiplicação de matrizes com diferentes técnicas aplicadas.</i>	59
<i>Figura 4.23: Ocupação em memória da multiplicação de matrizes com diferentes técnicas aplicadas.</i>	59
<i>Figura 4.24: Tempos de execução para o bubble sort com diferentes técnicas aplicadas.</i>	60
<i>Figura 4.25: Ocupação em memória do bubble sort com diferentes técnicas aplicadas.</i>	61
<i>Figura 4.26: Tempos de execução para o algoritmo de encriptação (TEA2) com diferentes técnicas aplicadas.</i>	61
<i>Figura 4.27: Ocupação em memória do algoritmo de encriptação com diferentes técnicas aplicadas.</i>	62
<i>Figura 4.28: Proteção em código de máquina, à esquerda, e proteção em código assembly, à direita.</i>	66
<i>Figura 4.29: Tempos de execução para a multiplicação de matrizes rodando sobre o processador LEON3.</i>	68
<i>Figura 4.30: Ocupação em memória da multiplicação de matrizes rodando sobre o processador LEON3.</i>	68
<i>Figura 4.31: Tempos de execução para o bubble sort (LEON3).</i>	69
<i>Figura 4.32: Ocupação em memória do bubble sort com diferentes técnicas aplicadas, rodando sobre o processador LEON3.</i>	69
<i>Figura 4.33: Tempos de execução para o algoritmo de encriptação (TEA2) com diferentes técnicas aplicadas, rodando sobre o processador LEON3.</i>	70
<i>Figura 4.34: Ocupação em memória do algoritmo de encriptação com diferentes técnicas aplicadas, rodando sobre o processador LEON3.</i>	71
<i>Figura 5.1: Uso dos registradores para o bubble sort: destino (representa o contador destino), fonte (representa o contador fonte) e combinado (representa o contador combinado).</i>	76
<i>Figura 5.2: Uso dos registradores para a multiplicação de matrizes.</i>	77
<i>Figura 5.3: Uso dos registradores para o algoritmo de encriptação TEA2.</i>	77
<i>Figura 5.4: Tempos de execução para o bubble sort.</i>	78
<i>Figura 5.5: Ocupações de memória para o bubble sort.</i>	79
<i>Figura 5.6: Tempos de execução para a multiplicação de matrizes.</i>	79
<i>Figura 5.7: Ocupações de memória para a multiplicação de matrizes.</i>	80
<i>Figura 5.8: Tempos de execução para o algoritmo de encriptação TEA2.</i>	80
<i>Figura 5.9: Ocupações de memória para o algoritmo de encriptação TEA2.</i>	81

LISTA DE TABELAS

<i>Tabela 4.1: Tipos de dados do formato SREC</i>	51
<i>Tabela 4.2: Multiplicação de Matrizes (MIPS) - CFT-tool</i>	63
<i>Tabela 4.3: Multiplicação de Matrizes (MIPS) – HPCT Suite</i>	63
<i>Tabela 4.4: Bubble Sort (MIPS) - CFT-tool</i>	64
<i>Tabela 4.5: Bubble Sort (MIPS) - HPCT Suite</i>	65
<i>Tabela 4.6: Algoritmo de Encriptação TEA2 (MIPS) - CFT-tool</i>	65
<i>Tabela 4.7: Algoritmo de Encriptação TEA2 (MIPS) – HPCT Suite</i>	66
<i>Tabela 4.8: Multiplicação de Matrizes (LEON3) – CFT-tool</i>	71
<i>Tabela 4.9: Bubble Sort (LEON3) – CFT-tool</i>	72
<i>Tabela 4.10: Algoritmo de Encriptação TEA2 (LEON3) – CFT-tool</i>	73
<i>Tabela 5.1: Bubble Sort</i>	82
<i>Tabela 5.2: Multiplicação de Matrizes</i>	83
<i>Tabela 5.3: Algoritmo de Encriptação (TEA2)</i>	84

RESUMO

Este trabalho apresenta uma ferramenta configurável, denominada de CFT-tool, capaz de aplicar automaticamente técnicas de detecção de erros em *software* com o objetivo de proteger processadores com diferentes arquiteturas e organizações contra falhas transientes no *hardware*. As técnicas baseadas em redundância e comparação são aplicadas pela CFT-tool no código *assembly* de um programa desprotegido, compilado para a arquitetura alvo.

A ferramenta desenvolvida foi validada utilizando dois processadores distintos: miniMIPS e LEON3. O processador miniMIPS foi utilizado para verificar a eficiência, em termos de taxa de detecção de erros, tempo de execução e ocupação de memória, das técnicas de detecção em *software* aplicadas pela CFT-tool, comparando os resultados obtidos com os presentes na literatura. O processador LEON3 foi selecionado por ser amplamente utilizado em aplicações espaciais e por ser baseado em uma arquitetura diferente da arquitetura do processador miniMIPS. Com o processador LEON3 é verificada a configurabilidade da CFT-tool, isto é, a capacidade dela de aplicar técnicas de detecção em *software* em um código compilado para um diferente processador, o mantendo funcional e sendo capaz de detectar erros. A CFT-tool pode ser utilizada para proteger programas para outras arquiteturas e organizações através da modificação dos arquivos de configuração da ferramenta. A configuração das técnicas é definida segundo as especificações da aplicação, recursos do processador e seleções do usuário.

Programas foram protegidos e falhas foram injetadas em nível lógico em ambos os processadores. Para o processador miniMIPS, as taxas de detecção de erros, os tempos de execução e as ocupações de memórias dos programas protegidos se mostraram compatíveis com os resultados presentes na literatura. Resultados semelhantes foram encontrados para o processador LEON3. Diferenças entre os resultados ocorrem devido às características da arquitetura.

A ferramenta CFT-tool por ser configurável pode proteger o código na integralidade ou selecionar partes do código e registradores que serão redundantes e protegidos. A vantagem de proteger parte do código é reduzir o custo final em termos de tempo de processamento e ocupação de memória. Uma análise do impacto da seleção seletiva de registradores na taxa de detecção de erros é apresentada. E diretivas de alcançar um comprometimento ótimo entre quantidade de registradores protegidos, taxa de detecção de erros e custo são discutidas.

Palavras-Chave: tolerância a falhas, falhas transientes, SEU, SET, técnicas de detecção em *software*.

CFT-tool: Configurable Tool to Application of Faults Detection Techniques in Processors by Software

ABSTRACT

This work presents a configurable tool, called CFT-tool, capable of automatically applying software-based error detection techniques aiming to protect processors with different architectures and organizations against transient faults in the hardware. The techniques are based on redundancy and comparison. They are applied by CFT-tool in the assembly code of an unprotected program, compiled to the target architecture.

The developed tool was validated using two distinct processors: miniMIPS and LEON3. The miniMIPS processor has been utilized to verify the efficiency of the software-based techniques applied by CFT-tool in the assembly code of unprotected programs in terms of error detection rate, runtime and memory occupation, comparing the obtained results with those presented in the literature. The LEON3 processor was selected because it is largely adopted in space applications and because it is based on a different architecture than miniMIPS processor. The configurability of the CFT-tool is verified with the LEON3 processor, that is, the capability of the tool at applying software-based detection techniques in a code compiled to a different processor, maintaining it functional and capable of detecting errors. The CFT-tool can be utilized to protect programs compiled to other architectures and organizations by modifying the configuration files of the tool. The configuration of the techniques is defined by the specifications of the application, processor resources and selections of the user.

Programs were protected and faults were injected in logical level in both processors. When using the miniMIPS processor, the error detection rates, runtimes and memory occupations of the protected programs are comparable to the results presented in the literature. Similar results are reached with the LEON3 processor. Differences between the results are due to architecture features.

The CFT-tool can be configurable to protect the entire code or to select portions of the code or registers that will be redundant and protected. The advantage of protecting portions of the code is to reduce the final cost in terms of runtime and memory occupation. An analysis of the impact of selective selection of registers in the error detection rate is also presented. And policies to reach an optimum commitment between amount of protected registers, error detection rate and cost are discussed.

Keywords: fault tolerance, transient faults, SEU, SET, software-based detection techniques.

1 INTRODUÇÃO

A evolução da indústria de semicondutores tem possibilitado a fabricação de circuitos integrados cada vez menores, fazendo com que o tamanho dos transistores se aproxime dos limites físicos (THOMPSON, 2005). Desse modo circuitos integrados compostos por transistores de dimensão reduzida, operando em baixa tensão de alimentação e alta frequência de operação, são mais sensíveis a defeitos de fabricação e falhas devido ao ruído do meio (DODD, 2010).

As falhas podem ser permanentes, transientes ou intermitentes. As falhas transientes podem vir do meio, como fonte de radiação ou de influencia eletromagnética. Single Event Effects (SEE) ou soft errors (NICOLESCU, 2003) são os efeitos transientes que ocorrem em circuitos integrados operando no espaço e mais recentemente em circuitos integrados fabricados em tecnologia nanométrica operando também na terra. São causados por efeitos da radiação devido à interação de nêutrons com o silício gerando partículas secundárias como alfa. Quando este efeito ocorre em um elemento de memória, é denominado Single Event Upset (SEU) e caracteriza-se pela inversão do valor armazenado no flip-flop (um bit-flip). Já quando esse efeito transiente afeta uma porta lógica de um bloco combinacional, é denominado Single Event Transient (SET) e é observado como um pulso de tensão transiente (*glitch*) de duração variável conforme carga coletada.

Um fator que tem aumentado a sensibilidade dos circuitos aos efeitos da radiação é o aumento do número de transistores por área, possibilitado pela redução do seu tamanho, o que faz com que as chances de uma partícula energizada afetem um ou mais nós sensíveis do circuito ao mesmo tempo sejam maiores, gerando múltiplas falhas.

Além disso, o Single Event Transient (SET) tem se mostrado um problema crescente com o avanço da tecnologia que tem permitido cada vez maiores frequências de operação. Enquanto a frequência está aumentando, o tempo de um pulso transiente tem se mantido constante (FERLET-CAVROIS, 2005), aumentando, com isso, a probabilidade de um pulso transiente causar um erro, pois, com um menor intervalo entre cada borda do relógio, a chance desse pulso ser capturado pela borda de relógio aumenta.

Para garantir o funcionamento correto de um circuito integrado na presença de falhas, podem ser usadas técnicas de tolerância a falhas que detectam ou detectam e corrigem falhas. A falha pode ocorrer e se manifestar como erro ou ser mascarada pela lógica do circuito e não ocasionando erro. Quando a falha se manifesta, normalmente se detecta o erro que ela gerou. No caso dos processadores, podemos ver o efeito das falhas SEU e SET como erros nos dados gerados pelo programa ou como erros no fluxo de execução do código original.

Para proteger os processadores, técnicas de tolerância a falhas podem ser aplicadas em *hardware* e/ou em *software*, fazendo uso de redundância de *hardware* ou de instruções no espaço ou no tempo para detectar falhas que se manifestem como erros nos dados ou no fluxo de execução do processador. O impacto do uso de técnicas de tolerância pode ser medido em área, desempenho, potência, tempo de execução e taxa de detecção de erros.

Nas técnicas em *hardware*, o sistema físico é alterado. O circuito normalmente é duplicado ou triplicado e verificadores ou votadores são inseridos (PRADHAN, 1995). Pode-se também usar códigos de correção de erro (EDAC) nos elementos de memória. Conforme as técnicas em *hardware* implementadas, o sistema apresenta diferentes custos como diminuição na frequência de operação, aumento em área e potência, além de possuir um alto custo de projeto e fabricação (ASENSI, 2011).

Já as técnicas em *software* mantém a parte física inalterada, permitindo o uso de processadores comerciais que são consideravelmente mais baratos que um processador específico desenvolvido para ser tolerante a falhas (processador rad-hard). O que deve ser modificado é somente o código do programa a ser protegido. O programa original é alterado onde técnicas de proteção em *software* são adicionadas, tais como técnicas de duplicação de variáveis, por exemplo, EDDI (OH, 2002a), ou técnicas de controle de fluxo, por exemplo, CFCSS (OH, 2002b), ou aplicando regras de transformação de código (CHEYNET, 2000). Contudo, essas aumentam consideravelmente o tempo de computação e a ocupação de memória da aplicação.

Para realizar a proteção de microprocessadores em *software* podem-se aplicar técnicas de proteção em três diferentes níveis de linguagem. Uma possibilidade é proteger o código em nível de linguagem de programação, por exemplo, na linguagem C. Contudo, o código protegido em alto nível está sujeito a otimizações pelo compilador o qual pode remover as redundâncias geradas pela técnica e tornar a proteção ineficaz. Outra possibilidade em alto nível é alterar o compilador para que esse não remova as redundâncias aplicadas pela técnica, tornando-a eficiente para a proteção da arquitetura alvo. A técnica também pode ser aplicada ao código *assembly* do programa, dessa forma, ficando livre das otimizações realizadas pelo compilador e não necessitando modificar o compilador ou o montador. Por fim, pode-se proteger diretamente o código de máquina do programa, o qual, como em *assembly* não necessita modificar o compilador ou o montador.

Aplicando as técnicas de proteção em *software* ao código de máquina tem-se a garantia que essas não sofrerão alterações, pois estão sendo aplicadas ao produto final do programa. Contudo, algumas dificuldades surgem na aplicação das técnicas de proteção nesse nível. Primeiramente, em alguns processadores, visando um aumento no desempenho, um determinado número de instruções após instruções de desvio é sempre executado. Para garantir a integridade do programa, o montador move essas instruções de desvio para cima. Dessa forma, essas instruções estão logicamente fora de ordem e isso deve ser levado em conta na hora da aplicação de técnicas de proteção para que a corretude do programa não seja perdida. O outro problema que afeta a aplicação dessas técnicas em código de máquina diz respeito aos endereços de destino dos desvios. Com a aplicação das técnicas de proteção em *software*, as instruções do programa têm seus endereços alterados, portanto, para que o programa funcione corretamente, é necessário calcular os novos endereços de destino das instruções de desvio.

Uma alternativa para a aplicação das técnicas de proteção sem a necessidade de se preocupar com essas questões relativas ao processador é aplicá-las em um nível de abstração mais elevado. Contudo, aplicar as técnicas em uma linguagem de alto nível não é uma boa alternativa, visto que nesse nível o programa está sujeito às otimizações realizadas pelo compilador que remove as redundâncias no processo de compilação e consequentemente as proteções aplicadas pelas técnicas, deixando o programa desprotegido. Portanto, a saída é aplicar as técnicas em um nível intermediário, modificando o código *assembly*. O código *assembly* não está sujeito às otimizações realizadas pelo compilador, visto que ele é gerado justamente pelo compilador, após o processo de compilação. Isso pode ser visto na figura 1.1, a qual apresenta os diferentes níveis de abstração do programa e as transformações pela qual o código em alto nível sofre até a geração do código de máquina.

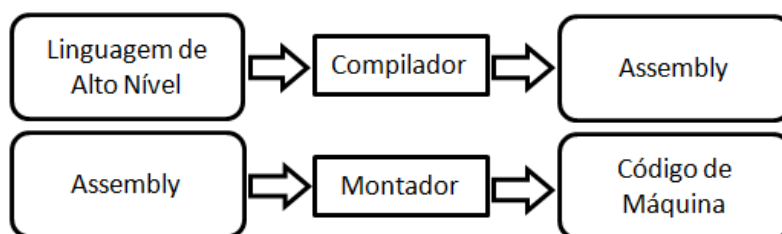


Figura 1.1: Níveis de abstração e processos de transformação do programa.

Depois de compilado, o programa encontra-se em sua versão em código *assembly*, um nível acima do código final que será executado pelo processador, o código de máquina. Entre o código *assembly* e o código de máquina existe o montador. O montador altera algumas instruções do código *assembly* no processo de montagem para o código de máquina, visando otimizá-lo. Um exemplo disso é a modificação das posições dos desvios, citado como um dos problemas de se aplicar as técnicas de proteção em código de máquina. Outro fator é que nem todas as instruções em código *assembly* são um-para-um em relação ao código de máquina, ou seja, para algumas instruções em *assembly*, o montador precisa gerar mais de uma instrução única instrução para realizar a função referente àquela instrução.

Este trabalho apresenta a CFT-tool, uma ferramenta capaz de proteger diversos microprocessadores com diferentes organizações e arquitetura contra falhas transientes através da detecção do efeito destas falhas. Para isso, é necessário que essa ferramenta seja configurável e que seja capaz de aplicar uma gama de técnicas de tolerância a falhas, capazes de detectar erros, ou seja, técnicas de detecção de erros em *software*, as quais possam ser configuradas pelo usuário, conforme custos como área, tempo de execução e taxa de detecção. Visto que a ferramenta deve ser independente de arquitetura, a melhor opção para se aplicar as técnicas seria na linguagem de alto nível, a qual é independente de arquitetura, porém, o compilador, no processo de otimização do código, acaba removendo as proteções aplicadas pela técnica. A outra possibilidade em alto nível é modificar o compilador para não retirar as redundâncias aplicadas pela técnica no processo de otimização. Essa abordagem torna a proteção eficiente, mas torna a ferramenta dependente da arquitetura para qual o compilador foi projetado. Como a ferramenta deve ser genérica, e modificar o compilador de cada arquitetura alvo é inviável, essa abordagem também foi rejeitada. As duas possibilidades restantes consistem em aplicar a proteção em *assembly* ou código de máquina. No entanto, ambas estão diretamente atreladas à arquitetura, o que também não torna essas abordagens genéricas. A solução encontrada é descrever a arquitetura alvo para que a ferramenta

seja capaz de interpretá-la e aplicar técnicas de tolerância a falhas à arquitetura descrita. Dessa forma, para alterar a arquitetura alvo, basta modificar a descrição da arquitetura.

A escolha em se aplicar as técnicas em *assembly* ou código de máquina ocorre pela facilidade de se aplicar as técnicas em *assembly* em relação ao código de máquina. Aplicando as técnicas em código de máquina, é necessário corrigir os endereços alvo instruções de desvio que tiveram seus alvos deslocados de posição no processo de aplicação das técnicas, além de levar em consideração se as instruções desvio foram deslocadas pelo montador, caso o processador alvo executa sempre um determinado número de instruções após os desvios. Por outro lado, aplicando-se as técnicas de proteção em *assembly*, abstraem-se esses problemas já que os destinos dos desvios são indicados por *labels* e as instruções estão na posição correta. Por esses motivos, foi escolhido aplicar as técnicas em *assembly*, onde pode ser utilizado um compilador e montador existente para a arquitetura alvo, sem necessitar modificá-los.

O trabalho foi dividido nas seguintes etapas. Primeiramente, algumas técnicas de proteção foram aplicadas em *assembly* e código de máquina a um caso de estudo para verificar a taxa de detecção da técnica sendo aplicada em *assembly* em relação ao código de máquina. Em seguida, a ferramenta foi desenvolvida implementando um conjunto de técnicas de proteção em *software* que se mostraram eficientes. Essas técnicas foram testadas e comparadas com resultados obtidos por Azambuja (2010), que adaptou um conjunto de técnicas para a aplicação em código de máquina e realizou os testes para o processador miniMIPS (HANGOUT, 2009), o qual também foi utilizado nesta dissertação. Depois de verificada a eficiência da ferramenta em nível *assembly*, a configurabilidade desta foi validada utilizando o processador LEON3 (GAISLER, 2004), o qual é bastante utilizado em aplicações espaciais. A validação final foi feita implementando um conjunto de técnicas de detecção a falhas em *software* nos processadores alvo e comparando as taxas de detecção de erros através de uma campanha de injeção de falhas, onde milhares de falhas transientes foram injetadas através de simulação lógica, sendo a eficiência da ferramenta analisada. Os processadores escolhidos são todos RISC, mas a maioria das técnicas aplica-se também em processadores CISC e superescalares (REBAUDENGO, 2000) (OH, 2002a). Mas caberá um estudo no futuro para otimizar as técnicas para essas arquiteturas assim como VLIW, por exemplo.

2 CONCEITOS DE DETECÇÃO DE ERROS EM SOFTWARE

Neste capítulo são apresentadas técnicas de detecção em *software* presentes na literatura. Além disso, também são abordadas, neste capítulo, ferramentas que implementam técnicas de detecção em *software* em diferentes níveis de linguagem são abordadas.

2.1 Técnicas de Detecção de Erros em Software

Em um sistema computacional, falhas transientes podem afetar o fluxo de execução de um programa (controle) ou modificar os dados armazenados em registradores ou na memória. Para proteção dos processadores, foram desenvolvidas diversas técnicas para proteção dos dados e do controle.

As técnicas de detecção em *software* podem ser classificadas conforme seu enfoque na proteção, sendo divididas em técnicas para proteção dos dados, técnicas para proteção do controle e técnicas de proteção híbridas. A seguir, são detalhadas essas três áreas.

2.1.1 Técnicas para proteção dos dados

As técnicas de proteção dos dados têm por objetivo proteger os dados armazenados na memória e nos registradores. Tais técnicas duplicam as variáveis, criando cópias dessas variáveis, onde toda operação realizada sobre a variável original é também realizada sobre sua cópia. Operações de comparação entre a variável original e a cópia são realizadas em diferentes pontos do programa para verificar a integridade dos dados. Os pontos onde essas instruções de verificação são inseridas dependem da técnica utilizada. Esse tipo de técnica não é projetado para a proteção do fluxo de controle do programa, mas mesmo assim é capaz de detectar alguns erros causados por desvios no fluxo de controle que acabam causando diferenciação entre as variáveis originais e suas cópias.

Um exemplo de técnica de proteção aos dados implementada somente em *software*, proposta por Oh (2002a) é a EDDI (Error Detection by Duplicated Instructions). A EDDI é uma técnica capaz de detectar as falhas que venham a afetar os dados. Ela duplica toda a informação, ou seja, toda variável é duplicada e toda a operação que é executada sobre a variável original também é executada sobre sua cópia. Para garantir que os dados estejam consistentes, é realizada uma comparação no valor das variáveis originais e suas cópias, verificando se ambas têm o mesmo valor. Essa verificação é realizada antes dos dados serem armazenados na memória, ou seja, antes das instruções

de *store*, pois o armazenamento correto dos dados na memória implica em uma execução correta do programa. Portanto, realizar a comparação entre os dados antes das instruções de *store* torna-se uma boa estratégia. Contudo, verificar os dados somente antes de gravar na memória é insuficiente, já que dados errados podem levar os desvios a tomar a direção incorreta no fluxo de execução, afetando a corretude do programa. Portanto, os dados devem ser verificados também antes das instruções de desvio.

Na figura 2.1 é apresentado um exemplo de aplicação da técnica EDDI. Podemos ver que a instrução *load*, a qual carrega um valor de uma posição de memória indicada pela variável "GLOBAL" no registrador 12 foi duplicada, sendo carregada uma cópia do valor no registrador 22. Em seguida, a instrução *add* foi também duplicada, onde as cópias dos registradores 11, 12 e 13 são, respectivamente, os registradores 21, 22 e 23. A instrução *store* também foi duplicada. Para garantir que o dado também esteja duplicado na memória, um deslocamento é somado ao registrador que indica o endereço de memória. Contudo, como se trata de uma instrução de armazenamento, deve-se realizar a verificação dos registradores originais com suas respectivas cópias para garantir a integridade dos dados. Isso pode ser visto nas comparações entre os registradores 11 e 21 e os registradores 12 e 22. Caso ambos possuam o mesmo valor, o programa é executado normalmente, no entanto, se ocorrer uma discrepância em uma das comparações, o erro é detectado e o programa passa para a rotina de tratamento de erro.

<pre>ld r12=[GLOBAL] add r11=r12,r13 st m[r11]=r12</pre>	<pre>ld r12=[GLOBAL] 1: ld r22=[GLOBAL+offset] add r11=r12,r13 2: add r21=r22,r23 3: cmp.neq.unc p1,p0=r11,r21 4: cmp.neq.or p1,p0=r12,r22 5: (p1) br faultDetected st m[r11]=r12 6: st m[r21+offset]=r22</pre>
(a) Original Code	(b) EDDI Code

Figura 2.1: Aplicação da técnica EDDI (REIS, 2005).

Desconsiderando as falhas detectadas pelo sistema operacional, a técnica EDDI consegue uma taxa de detecção de cerca de 93% para Fast Fourier Transform e multiplicação de matrizes (OH, 2002a) para as falhas que afetam os dados. Contudo, todos os registradores do programa precisam ser duplicados juntamente com todas as operações sobre tais registradores. Também é necessário incluir a verificação desses registradores com suas cópias antes de cada instrução de armazenamento ou desvio onde esses registradores estiverem sendo utilizados. Portanto, o aumento na ocupação de memória e o tempo de execução são significativos e deve ser levado em conta na utilização dessa técnica. Além disso, essa técnica não garante a detecção de erros que afetem o controle, o qual pode causar uma saída incorreta nos dados.

2.1.2 Técnicas para proteção do controle

As técnicas de proteção ao controle buscam garantir a integridade do fluxo de execução do programa. As técnicas presentes na literatura dividem o programa em blocos básicos, que são porções de código sem instruções de desvio. Para cada bloco básico é atribuída uma assinatura que o identifica. Existem três tipos de falhas que afetam o controle: falhas que desviam o fluxo de execução do programa do meio de um

bloco básico para o meio de outro bloco básico; falhas que desviam o fluxo de execução de um bloco básico para o início de outro bloco; e falhas que desviam o fluxo de execução do programa para outra parte do bloco básico em execução. Esse último tipo de falha não é detectado pelas técnicas presentes na literatura.

Dentre as principais técnicas de proteção ao fluxo de controle temos a CCA (Control Flow Checking using Assertions), proposta por McFearin (1995), a qual divide o programa em blocos básicos. A técnica CCA precisa de três registradores para sua implementação, um para identificação do bloco básico, Basic Block Identifier (BID), e dois para identificação do fluxo de controle, Control Flow Identifier (CFID). Cada bloco básico possui um BID único que é assinalado no início da execução do bloco e é checado ao final da execução do bloco. O CFID do bloco básico é atribuído no fim da fila dos dois registradores de identificação quando o bloco é iniciado. E, ao final da execução do bloco, o CFID do bloco executado anteriormente é retirado. Nesse ponto, como pode ser visto na figura 2.2, o método é capaz de identificar se após a execução de um bloco básico, o fluxo de execução foi mudado para algum bloco que não deveria ser executado após aquele.

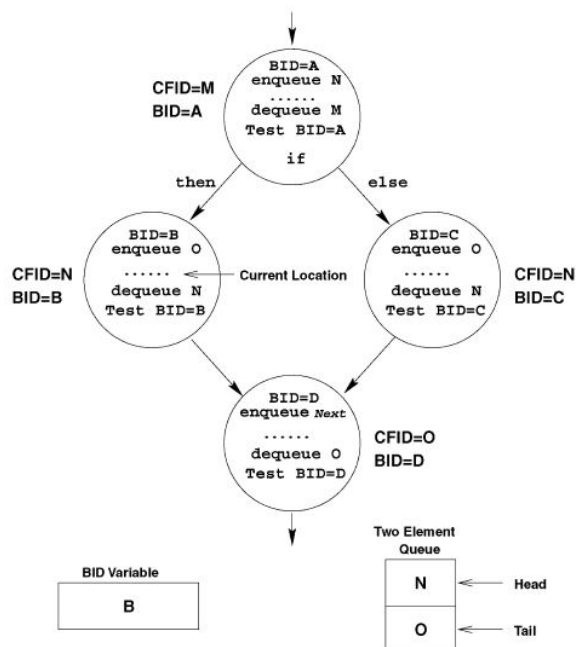


Figura 2.2: Exemplo da técnica CCA (ALKHALIFA, 1999).

No entanto, a técnica CCA não consegue detectar falhas quando ocorre uma mudança no fluxo para dentro de um mesmo bloco básico ou quando o fluxo de execução é mudado, após o teste do BID para o início de outro bloco básico com o mesmo CFID que o bloco que deveria ser executado.

Uma modificação na técnica CCA foi proposta por Alkhalifa (1999), trata-se da técnica ECCA (Enhanced Control Flow Checking using Assertions). A diferença da técnica CCA para a ECCA consiste em que essa última utiliza assinaturas de tempo real, ou seja, que as assinaturas CFID dos blocos se modificam em tempo de execução através de uma assinatura de ajuste que nada mais é que uma operação "ou exclusivo" entre a assinatura do bloco básico pai e do bloco básico filho. Isso garante que falhas que desviam o fluxo de controle de um bloco básico para outro que não deveria ser executado em seguida seja detectado. Contudo, apesar dessa técnica detectar até 85%

dos erros (ALKHALIFA, 1999), ela aumenta o tempo de execução em relação à técnica CCA por exigir o cálculo de assinaturas em tempo real. Também não consegue detectar falhas que desviem o fluxo de execução do programa para dentro do mesmo bloco básico que está sendo executado.

Outra técnica de proteção ao fluxo de controle é a CFCSS (Control Flow Checking by Software Signatures) (OH, 2002b) que, da mesma forma que as outras técnicas de proteção ao fluxo de controle apresentadas, divide o programa em blocos básicos e adiciona uma assinatura a cada um. Um registrador é necessário para a implementação desse método, ele armazena o valor da assinatura do bloco básico em execução e, ao iniciar a execução de um novo bloco básico, uma operação “ou exclusivo” é executada com o atual valor do registrador e uma constante, resultando na assinatura do bloco básico atual. Caso o resultado seja diferente da assinatura do bloco, uma falha no fluxo de execução do programa ocorreu.

Essa técnica tem um menor custo em relação às técnicas de proteção ao fluxo de controle citadas anteriormente, pois necessita apenas de um registrador para armazenar o valor da assinatura atual, da inserção de uma operação ou exclusivo e uma comparação com a assinatura por bloco básico. Contudo, sofre dos mesmos problemas da técnica CCA, ou seja, não é capaz de detectar falhas que causem um desvio no fluxo de execução do programa de um bloco básico pai para um bloco básico filho que não seja o bloco básico que deveria ser executado, além de não ser capaz de detectar falhas que mudem o fluxo de execução do programa para dentro de um mesmo bloco básico.

2.1.3 Técnicas híbridas

As técnicas de proteção híbridas buscam proteger tanto os dados quanto garantir que o fluxo de execução do programa seja seguido corretamente. O que esse tipo de técnica faz é aplicar regras de transformação sobre o código, somando características das técnicas de proteção aos dados com características das técnicas de proteção ao fluxo de controle.

Um conjunto de regras de transformação proposto por Rebaudengo (1999) estabelece regras para a proteção dos dados e regras para proteção do fluxo de controle. Para proteger os dados, toda variável deve ser duplicada, sendo que cada operação de escrita realizada sobre a variável original deve ser realizada sobre sua cópia. Em toda operação de leitura da variável, deve ser realizado, além de uma operação de leitura sobre sua cópia, uma verificação entre os valores armazenados na variável original e em sua cópia. Caso seus valores não sejam iguais, o erro é detectado.

Visando a detecção das falhas que afetam o fluxo de execução do programa, um valor inteiro está associado a cada bloco básico, sendo o valor atribuído no início da execução do bloco e testado no final. Técnica semelhante às assinaturas BID da técnica de proteção ao fluxo de controle CCA. Além disso, uma nova regra destinada à proteção do fluxo de controle foi incluída, na qual deve haver réplicas dos desvios condicionais, uma para caso o desvio seja tomado e outra para caso o desvio não seja tomado. No local para onde o fluxo de execução vai caso o desvio seja tomado é inserido um desvio condicional logicamente inverso ao original. E no local para onde o fluxo de execução vai caso o desvio não seja tomado é inserido um desvio igual ao original. Ambas as cópias desviam o fluxo de execução do programa para uma rotina de tratamento de erro caso sua condição seja verdadeira, caso contrário, o programa segue sua execução normalmente. Por fim, a cada sub-rotina do sistema é associado um valor inteiro. Esse

valor é atribuído a um registrador imediatamente antes do retorno da sub-rotina, sendo testado na instrução seguinte à chamada de sub-rotina, ou seja, logo após o retorno da sub-rotina, onde é verificado se o valor do registrador é condizente com o valor esperado para aquela sub-rotina. Segundo Rebaudengo (1999), essa técnica é capaz de detectar quase 100% dos erros que afetam os dados, mas não é capaz de fazer o mesmo com os erros que afetam o controle. Além disso, essa técnica aumenta em cerca de 444% o tempo de execução do programa, o que a torna ineficiente, visto que esse tempo é mais que suficiente para executar o programa três vezes, realizar uma votação entre os três resultados e definir o resultado correto.

Outra técnica que busca proteger tanto os dados quanto o fluxo de execução do programa foi proposta por Nicolescu (2003) e também sugere um conjunto de regras que devem ser aplicadas ao programa. As variáveis são divididas em dois tipos: variáveis temporais e variáveis finais. Ambas devem ser duplicadas e toda operação sobre a variável original deve também ser realizada sobre a cópia. Contudo, a verificação da consistência dos dados é aplicada somente às variáveis finais após cada operação de escrita. Variáveis finais são aquelas que serão armazenadas em memória. Essa separação entre variáveis temporais e finais é a principal diferença dessa técnica para a técnica de Rebaudengo (1999) e influencia significativamente na diminuição do tempo de execução do programa que tem um aumento de cerca de 144% em relação ao programa original frente aos 444% da técnica de Rebaudengo (1999).

Para proteger o fluxo de execução do programa é atribuído um valor inteiro e um valor lógico a cada bloco básico. O valor lógico é verdadeiro quando o bloco não está em execução e falso quando o bloco está sendo executado. Os valores de identificação do bloco básico são associados a um registrador no início da execução do bloco. Por fim, como em Rebaudengo (1999), as instruções de desvio são replicadas. Além disso, também são atribuídos valores a cada sub-rotina do programa, sendo que nessa técnica a verificação da sub-rotina é realizada em dois pontos, antes e após a execução da sub-rotina.

Como foi citado, o aumento no tempo de execução gerado por essa técnica é significativamente menor que o aumento da técnica de Rebaudengo (1999), entretanto, suas taxas de detecção são inferiores, sendo capaz de detectar cerca de 85% das falhas que afetam os dados e 56% das falhas que afetam o controle.

Podemos perceber que as técnicas buscam proteger os dados e o fluxo de controle do programa separadamente. Logo, uma abordagem lógica seria combinar técnicas de proteção aos dados com técnicas de proteção ao fluxo de controle. Foi isso que REIS (2005) fez, combinando a técnica de proteção aos dados EDDI com a técnica de proteção ao fluxo de controle CFCSS e realizando algumas alterações nessa nova técnica, denominada SWIFT (Software Implemented Fault Tolerance). Primeiramente, foi modificada a parte destinada a proteção do controle. Foram atribuídos assinaturas a todos os blocos básicos, como a técnica CFCSS original, contudo a verificação foi somente realizada sobre blocos que possuem instruções store, pois somente esses realizam escritas na memória. Uma segunda modificação consiste na retirada da verificação realizada pela EDDI nos desvios condicionais, pois os desvios já são protegidos pela técnica CFCSS. Como as demais técnicas apresentadas, SWIFT não é capaz de detectar os erros desviam o fluxo de execução do programa para dentro do mesmo bloco básico.

2.2 Programas que Implementam Técnicas de Detecção em Software

Um programa descrito em linguagem de alto nível passa por vários processos até chegar a seu código de máquina. Primeiramente, o código é compilado, sendo passado da linguagem de alto nível para *assembly* a qual está associada a uma determinada arquitetura. No processo de compilação, o compilador realiza otimizações no programa. Através do montador, o código de máquina é gerado a partir do código *assembly*. A figura 1.1 mostra o programa em seus diferentes níveis de abstração, indicando os processos pelos quais ele passa até que possa ser executado pelo processador.

2.2.1 Proteção em linguagem de alto nível

Como foi dito, as técnicas de proteção podem ser aplicadas em vários níveis de abstração. Uma ferramenta que altera o código em linguagem de alto nível, antes de passar por processo de compilação ou montagem chamada C2C Translator (NICOLESCU, 2003) aplica algumas técnicas de proteção em linguagem C. A aplicação de técnicas neste nível torna a ferramenta independente de arquitetura. Contudo, o compilador busca otimizar ao máximo o programa, retirando redundâncias, o que pode ocasionar no não funcionamento das técnicas aplicadas nesse nível. Além das complicações em verificar o *software* modificado onde redundâncias e testes foram inseridos a fim de garantir que erros não tenham sido inseridos no código.

Outra possibilidade é modificar o compilador, como fez Reis (2005), que modificou o compilador OpenIMPACT para aplicar a técnica SWIFT no processador Intel Itanium 2. As otimizações que removiam as proteções foram alteradas para respeitar a técnica aplicada. Contudo, esse processo tira a generalidade de uma ferramenta nesse nível, pois o que é ganho aplicando as técnicas em linguagem de alto nível é perdido pelo fato de para cada processador alvo, ter que ser realizadas modificações sobre o compilador para tal processador.

2.2.2 Proteção em linguagem assembly

Uma outra possibilidade consiste em modificar o código *assembly* do programa, evitando assim, que as otimizações do compilador removam a proteção. Esse nível é mais simples para aplicação das técnicas de proteção em relação ao código de máquina, pois, dependendo do processador, as instruções de desvio em código de máquina podem ser trocadas de posição após o processo de montagem, além de ser necessário recalcular o endereço de destino dos desvios, pois a aplicação da técnica modifica a posição das instruções na memória. Contudo, para se aplicar a técnica em *assembly*, algumas alterações que ocorrem no processo de montagem precisam ser verificadas para determinar a eficiência de se aplicar as técnicas de proteção nesse nível.

2.2.3 Proteção em código de máquina

Na outra ponta está a proteção pela modificação do código de máquina do programa. Uma ferramenta que trabalha nesse nível é o HPCT Suite (AZAMBUJA, 2010) a qual altera o código de máquina do programa para o processador miniMIPS. Nesse nível, tem-se a garantia que o código não sofrerá alterações, contudo, apesar da eficiência da ferramenta para tal processador, ela está atrelada a ele. Porém, como o objetivo é aplicar as técnicas de forma genérica e não intrusiva, a proteção em código de máquina se torna um empecilho para o desenvolvimento de uma ferramenta genérica nesse nível.

3 CFT-TOOL: FERRAMENTA PARA APLICAÇÃO DE TÉCNICAS DE PROTEÇÃO EM SOFTWARE PARA DIFERENTES PROCESSADORES

Neste trabalho, foi desenvolvida uma ferramenta, denominada CFT-tool, que realiza transformações sobre o código *assembly* de programas. Técnicas de detecção em *software* são aplicadas pela CFT-tool sobre o código *assembly* de programas. O código *assembly* é modificado, sendo inseridos instruções redundantes e verificadores. As técnicas de detecção em *software* que a ferramenta implementa foram validadas por Azambuja (2010). A CFT-tool é independente da arquitetura e organização do processador, podendo ser utilizada para proteger diferentes processadores, com diferentes organizações e arquiteturas.

A CFT-tool foi desenvolvida na linguagem Java, uma linguagem multiplataforma, a qual permite a execução da ferramenta em diferentes sistemas operacionais. A ferramenta é dividida em módulos, baseados em suas funcionalidades. A linguagem Java foi escolhida devido a sua flexibilidade, praticidade e documentação vasta e organizada. Além disso, a linguagem Java possui uma comunidade grande e ativa de usuários, com muitas funções que auxiliam no desenvolvimento de programas.

A figura 3.1 apresenta uma visão geral de como a CFT-tool trabalha. As configurações referentes à arquitetura e organização do processador alvo são carregadas. Da mesma forma, as configurações sobre as técnicas de detecção em *software* que serão aplicadas sobre o código *assembly* do programa a ser protegido também são carregadas. O código *assembly* do programa desprotegido é lido e processado pela CFT-tool. A CFT-tool realiza transformações sobre o código desprotegido, aplicando as técnicas de detecção em *software* selecionadas, gerando, ao final, um código *assembly* protegido.

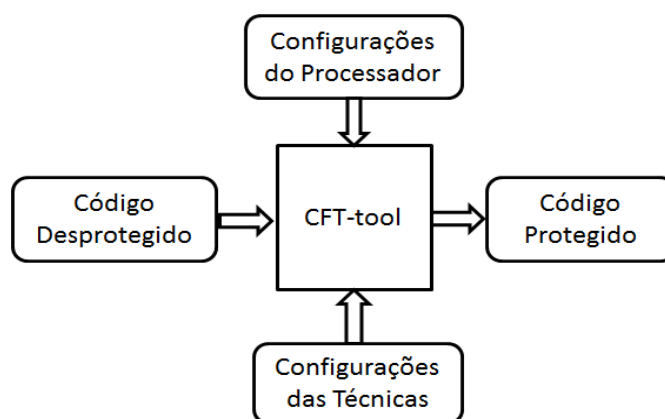


Figura 3.1: Visão geral do funcionamento da CFT-tool.

Há algumas vantagens no emprego de tolerância a falhas no código *assembly* em vez de no código de máquina. Por exemplo, no código de máquina, os endereços de destino dos desvios devem ser corrigidos, pois as instruções estarão em posições diferentes após a aplicação das técnicas de detecção em *software*. No código *assembly* isso não é necessário, pois o destino dos desvios são *labels*, sendo o endereço físico definido após o processo de montagem do código *assembly* para código de máquina. Além disso, o *branch delay slot*, que consiste em reordenar instruções de desvio para ganhar desempenho, aproveitando de características do *pipeline* do processador é outro fator. Somado a isso, aplicar as técnicas sobre o código *assembly* torna a CFT-tool mais portátil entre diferentes arquiteturas, pois o montador do processador alvo se encarrega de criar o arquivo executável. Por essas razões foi escolhido trabalhar com o código *assembly*, onde um compilador e um montador existentes podem ser utilizados sem que haja necessidade de qualquer modificação.

A figura 3.2 apresenta os estágios pelos quais o programa passa até que ser protegido pelas técnicas de detecção em *software*. O código do programa descrito em uma linguagem de alto nível é compilado, gerando o código *assembly* equivalente. Os números indicam a ordem de execução dos módulos da ferramenta.

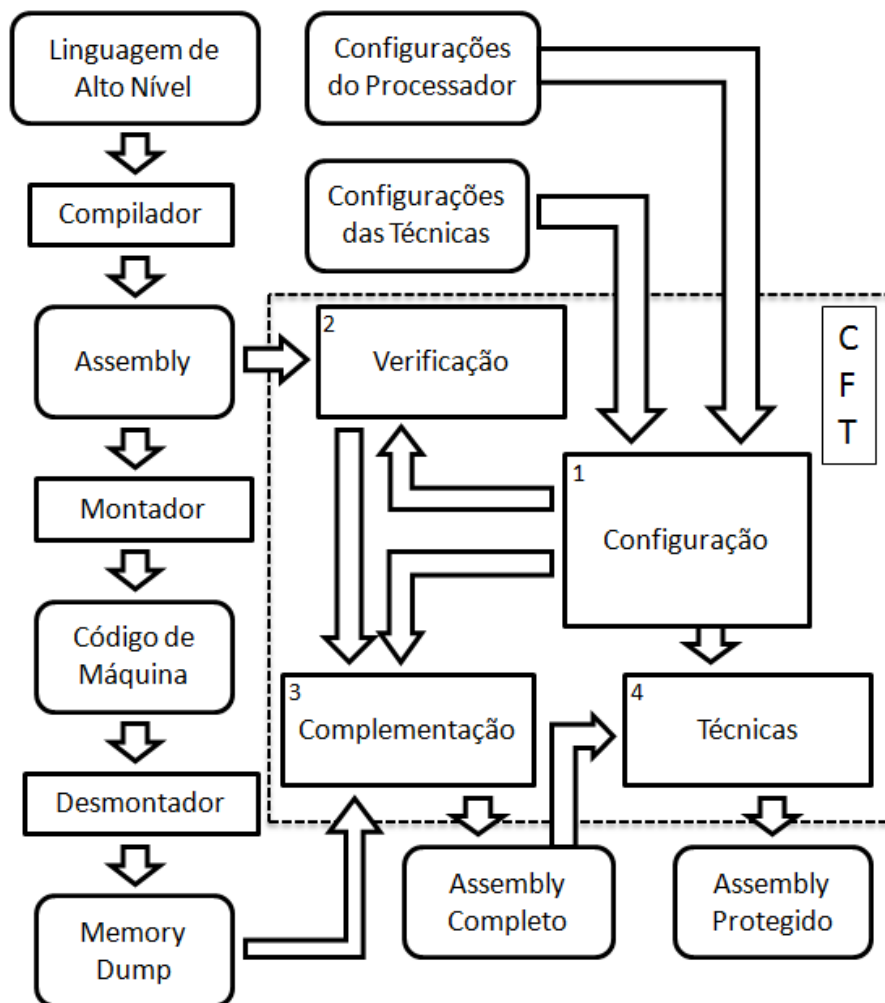


Figura 3.2: Processo de aplicação da proteção pela ferramenta CFT-tool.

A CFT-tool carrega as configurações referentes à arquitetura e organização do processador e as configurações sobre as técnicas de detecção em *software* a serem

aplicadas. Essas configurações são lidas dos arquivos de configurações, sendo elas informadas pelo usuário. Depois disso, o módulo de verificação lê o código *assembly* do programa a ser protegido e verifica se o código do programa é condizente com a arquitetura do processador informada. Se for condizente, o programa passa para o próximo estágio, caso contrário, um erro é reportado e a execução da CFT-tool é finalizada. No próximo estágio, o módulo de complementação é responsável com gerar um novo código *assembly*, contendo todas as sub-rotinas utilizadas pelo programa. Finalmente, o módulo de proteção a falhas transientes (SEE) aplica as técnicas de detecção em *software* selecionadas pelo usuário sobre o código *assembly* gerado pelo módulo de complementação, criando uma versão protegida, em *assembly*, do programa.

A ferramenta CFT-tool é dividida em quatro módulos. Esses módulos são apresentados em detalhes a seguir.

3.1 Módulo de Configuração

A CFT-tool é independente da arquitetura e organização do processador. Como as técnicas de detecção em *software* são aplicadas sobre o código *assembly* do programa e o *assembly* está atrelado a uma arquitetura, é necessário que as características da arquitetura e organização do processador alvo sejam passadas para a CFT-tool. As técnicas de detecção em *software* selecionadas para serem aplicadas sobre o código *assembly* do programa também precisam ser informadas à ferramenta. Para isso, um padrão foi criado, onde são utilizados arquivos contendo as configurações da arquitetura e organização do processador e das técnicas a serem aplicadas.

Todas as instruções da arquitetura devem ser descritas, informando seu mnemônico, o formato como ela é apresentada no código e seu tipo (aritmético, *load*, *store*, saltos incondicionais, desvios condicionais ou outros). Informações sobre os registradores também são necessárias. Devem ser informados seus nomes, se podem ser lidos e escritos explicitamente e seu tipo (global, local, de entrada ou de saída). Além disso, outras características sobre a arquitetura e organização do processador devem ser informadas. Por exemplo, o formato dos *labels*, das *tag* de comentário e informações sobre reordenamento de instruções. As técnicas de detecção em *software* que serão aplicadas e quais os registradores que serão protegidos também devem estar informadas nos arquivos de configurações.

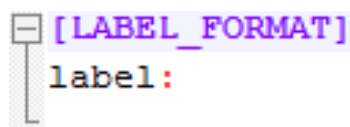
A função do módulo de configuração é carregar as configurações da arquitetura e organização do processador alvo e das técnicas de detecção em *software* a serem aplicadas e os padrões da arquitetura e as configurações aplicadas, informados nos arquivos de configuração, transformando em um formato que todos os outros módulos possam compreender. Ou seja, o módulo deve descrever toda a informação necessária sobre o processador e as técnicas para que os demais módulos possam executar suas funções de forma correta e independentemente da arquitetura.

O módulo de configuração é dividido em cinco partes. Configurações gerais apresentam características globais sobre a arquitetura e organização do processador. Configurações das instruções definem as instruções da arquitetura e configurações dos registradores contêm dados sobre os registradores existentes. As configurações do *memory dump* são necessárias para realizar a conversão de sub-rotinas do formato *memory dump* para *assembly*. E por fim, as configurações das técnicas apresentam as configurações para seleção e utilização das técnicas de detecção em *software*. Cada uma

dessas partes constitui um arquivo de configurações. As cinco partes do módulo de configuração são apresentadas em detalhes a seguir.

3.1.1 Configurações gerais

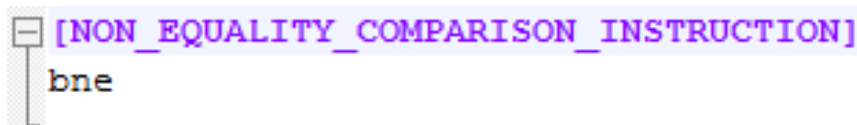
As configurações gerais contêm informações gerais sobre a arquitetura e organização do processador. O formato dos *labels*, a *tag* de comentário e o caractere de separação de operandos devem ser informados nesse arquivo. A figura 3.3 apresenta um exemplo de formato de *label* configurado. O nome *label* indica onde estará o nome do *label* na linha de código. Os demais caracteres servem para auxiliar na identificação do *label*.



```
[ LABEL_FORMAT ]
label:
```

Figura 3.3: Exemplo da configuração do formato do *label*.

Instruções que realizam alguma tarefa específica, necessária para que a CFT-tool consiga aplicar as técnicas de detecção em *software* sobre o código *assembly* do programa de forma correta, devem ter seu mnemônico informado nas configurações gerais. Por exemplo, a instrução que verifica se dois registradores têm valores diferentes é de extrema importância para a CFT-tool, pois ela faz a verificação entre os valores dos registradores originais e suas cópias. Por isso, ela deve ser declarada nas configurações gerais, como mostrado na figura 3.4. Outras instruções que se encaixam nessa classificação são as instruções de não operação, desvio condicional se diferente de zero, salto incondicional, associação de um valor constante a um registrador, cópia do valor de um registrador para outro, comparação entre registradores, comparação de registrador com constante e subtração de constante de um registrador.



```
[NON_EQUALITY_COMPARISON_INSTRUCTION]
bne
```

Figura 3.4: Exemplo de instrução que deve ser declarada nas configurações gerais devido às suas funcionalidades.

Branch delay slot é uma característica do *pipeline* de alguns processadores, onde instruções subsequentes à instrução de desvio em execução são sempre executadas, quer o desvio seja tomado ou não. Essa característica é mais comumente encontrada em processadores das arquiteturas DSP ou arquiteturas RISC mais antigas. No código, as instruções de desvio são deslocadas para cima, o número de vezes equivalente ao número de instruções que são sempre executadas após os desvios. Esse deslocamento pode acontecer no processo de compilação ou no processo de montagem. Se for realizado durante o processo de compilação, o código *assembly* terá instruções fora de ordem, e a CFT-tool precisa ter conhecimento disso para poder realizar as transformações no código *assembly* mantendo a corretude do programa. A informação sobre a existência do *branch delay slot* no código *assembly* deve ser informada no arquivo de configurações gerais, indicando o número de instruções que são sempre executadas após os desvios. A figura 3.5 mostra um exemplo dessa configuração para o processador LEON3. Por padrão, o valor dessa configuração é zero, portanto, não há necessidade de informá-lo caso essa característica não exista no processador alvo.

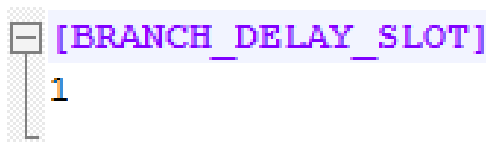


Figura 3.5: Configuração do *branch delay slot* para quando as instruções são reordenadas no código *assembly*.

Alguns registradores são utilizados por outras sub-rotinas antes do início da execução do programa. Esses registradores possuem algum valor atribuído e devem ser listados no arquivo de configurações gerais para que a CFT-tool não os utilize como registradores cópias.

Outra característica sobre registradores são as janelas de registradores, implementadas por algumas arquiteturas, como a SPARC. As janelas de registradores possuem registradores globais, acessáveis por todas as janelas, locais, exclusivos da janela em uso, e registradores de entrada e saída, para comunicação entre as janelas. Quando essa característica está presente no processador em uso, os mnemônicos das instruções que realizam a troca de uma janela de registradores para outra devem ser informados nos arquivos de configurações gerais.

As outras informações que devem estar contidas no arquivo de configurações gerais são o nome do arquivo que contém a representação do programa no formato *memory dump* e o mnemônico dos desvios condicionais, informando os desvios logicamente inversos na sequência. A figura 3.6 mostra como é feita essa associação entre os desvios logicamente inversos. Os mnemônicos de desvios condicionais logicamente inversos são separados por dois pontos e cada caso é separado por vírgula. As associações são realizadas nos dois sentidos, da primeira instrução para a segunda e da segunda para a primeira, ficando as duas ordens armazenadas na lista de desvios logicamente inversos do módulo de configuração.

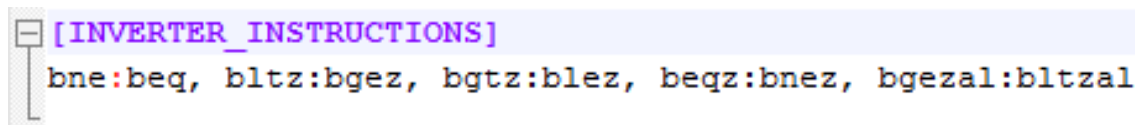


Figura 3.6: Configuração dos desvios condicionais logicamente inversos.

3.1.2 Configurações das instruções

O arquivo de configurações das instruções contém informações sobre o formato e tipo das instruções existentes na arquitetura do processador configurado. As instruções são configuradas em grupos, sendo que as instruções com as mesmas características são incluídas no mesmo grupo.

Os tipos aceitados são: *arithmetic*, *load*, *store*, *branch_to_target*, *jump_to_target*, *jump_to_register*, *no_operation* e *other*. *Arithmetic* é usado para definir instruções aritméticas. *Load* define instruções que carregam dados da memória para registradores e *store*, instruções que armazenam dados na memória. Os desvios são classificados com o tipo *branch_to_target*, que indica que o destino do desvio são *labels*, o que é adotado no código *assembly*. *Jump_to_target* serve para saltos incondicionais para *labels* e *jump_to_register*, para saltos para a posição indicada por um registrador. No código *assembly*, instruções do tipo *jump_to_register* são usadas somente no retorno de sub-rotinas. A instrução de não operação é classificada como *no_operation*. Instruções que não podem ser classificadas com nenhum dos tipos anteriores são do tipo *other*.

O formato da instrução serve para indicar a posição de cada elemento da instrução. A posição do mnemônico e dos operandos, sendo que esses ainda podem ser divididos em registradores fonte ou destino, valor imediato, valor de deslocamento e *label* de destino de desvios. A figura 3.7 apresenta um exemplo de formato de instrução. A posição do mnemônico é indicada pela identificação *ins*. O registrador destino é indicado por *rd* e registrador fonte por *rs*. O valor da constante de deslocamento, para operações de *load* e *store* é indicado por *offset*. Valores imediatos são referenciados por *imm*. Os destinos dos desvios, que são os nomes de *labels*, são identificados pela palavra *target*. Os outros caracteres presentes no formato são considerados constantes e servem para auxiliar na identificação do formato da instrução.

`ins rd,offset(rs)`

Figura 3.7: Exemplo de formato de instrução.

A figura 3.8 apresenta a configuração de um grupo de instruções. Essas instruções são do tipo aritmética. O formato das instruções desse grupo é formado pelo mnemônico da instrução, seguido por um espaço em branco e os operandos. Os operandos são o registrador destino e os registradores fontes, separados por vírgula.

```
[GROUP]
{INSTRUCTIONS}
add, addu, and, nor, or, sllv, slt, sltu, srav, srlv, sub, subu, xor
{FORMAT}
    ins rd,rs,rs
{TYPE}
arithmetic
```

Figura 3.8: Exemplo de grupo de instruções.

A CFT-tool aceita sobrecargas de instruções, isto é, diferentes instruções com o mesmo mnemônico. Por exemplo, uma instrução *add* que realiza a soma entre dois registradores e outra instrução, com o mesmo mnemônico *add*, que realiza a soma entre um registrador e um imediato. A figura 3.9 apresenta outro grupo de instruções. Comparando a figura 3.9 com a figura 3.8, se pode notar que os formatos diferem por um valor imediato na figura 3.9 em relação a um registrador fonte na figura 3.8. Nas duas figuras apresentadas, estão presentes em ambas as instruções *addu*, *slt*, *sltu* e *subu*. Essas instruções são sobrecarregadas, pois possuem o mesmo mnemônico, mas tem um formato diferente. A diferenciação entre elas é realizada pela CFT-tool levando em conta a quantidade e posição de cada identificador.

```
[GROUP]
{INSTRUCTIONS}
addi, addu, addiu, andi, ori, sll, slt, slti, sltiu, sltu, sra, srl, subu, xori
{FORMAT}
    ins rd,rs,imm
{TYPE}
arithmetic
```

Figura 3.9: Exemplo de grupo de instruções com instruções sobrecarregadas.

3.1.3 Configurações dos registradores

Assim como as instruções, os registradores são configurados em grupos. Em cada grupo deve ser informado o nome dos registradores, se eles podem ser lidos e/ou escritos explicitamente e o tipo a que pertencem. O tipo do registrador pode ser *global*, *local*, *input* ou *output*. Se o processador configurado não utiliza janela de registradores, todos os registradores são do tipo *global*.

Na figura 3.10 pode ser visto um exemplo de grupo de registradores. Nesse grupo estão presentes os registradores \$2, \$3, \$4, \$5, \$6, \$7, \$8, \$9, \$10, \$11, \$12, \$13, \$14 e \$15. Esses registradores podem ser lidos e escritos explicitamente, isto é, podem ser usados pelas instruções regulares configuradas sem restrições. Os registradores são do tipo *global*.

```
[GROUP]
{REGISTERS}
$2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14, $15
{READABLE}
YES
{WRITABLE}
YES
{TYPE}
global
```

Figura 3.10: Exemplo de grupo de registradores.

3.1.4 Configurações do memory dump

O arquivo de configurações do *memory dump* contém as informações referentes ao formato *memory dump* do processador configurado. Essas informações são necessárias para que o módulo de complementação possa converter o código de sub-rotinas do formato *memory dump* para código *assembly*.

Informações sobre os formatos dos dados são necessárias. O formato dos *labels*, que não fazem parte do código de máquina do programa, mas são utilizados no formato *memory dump* para definir o início de sub-rotinas. A figura 3.11 apresenta um exemplo de formato de *label* no formato *memory dump*. A letra *a* indica a posição na linha de código onde fica o endereço físico onde está situado o início da sub-rotina e a letra *t* indica o texto que contém o nome da sub-rotina, ou seja, o *label*. Os demais caracteres são constantes do formato configurado e servem para auxiliar na identificação dos elementos.

```
[LABEL_FORMAT]
a <t>:
```

Figura 3.11: Exemplo do formato de *label* no *memory dump*.

O formato das instruções também precisa ser informado, mas em vez de informar os formatos das instruções detalhadamente, como nas configurações de instruções, é necessário informar somente o formato de uma forma genérica. Na figura 3.12 é apresentado um exemplo do formato das instruções no *memory dump*. O caractere *a* indica a posição do endereço físico. Cada caractere *X* indica a posição de um dígito

hexadecimal que representam a palavra da instrução. A letra *i* indica a posição do mnemônico e a letra *o* indica a posição dos operandos, que podem, ou não, existir.

```
[INSTRUCTION_FORMAT]
a:  XX XX XX XX      i o
```

Figura 3.12: Exemplo do formato de instruções no *memory dump*.

A separação dos operandos é realizada por um caractere de separação de operandos que também deve ser informado. Além dos formatos mencionados, deve também ser informado o formato dos comentários. Há diferença no tratamento dos formatos do arquivo *memory dump* para os do código *assembly*. No código *assembly*, o tratamento dos formatos é mais flexível, devido à questão de espaçamento que pode variar e mesmo assim ser válida. No formato *memory dump* ela é rígida, e, portanto, o formato deve ser fielmente respeitado, inclusive nos espaçamentos.

Os nomes dos registradores podem variar do formato *memory dump* para o código *assembly* e, por esse motivo, o nome do registrador no formato *memory dump* deve ser informado e relacionado com seu equivalente em *assembly*. A figura 3.13 apresenta um exemplo parcial dessa configuração. À esquerda estão os nomes dos registradores no formato *memory dump* e a direita seu nome equivalente no código *assembly*.

```
[REGISTERS]
zero, $0
at, $1
v0, $2
v1, $3
a0, $4
a1, $5
a2, $6
a3, $7
t0, $8
t1, $9
t2, $10
```

Figura 3.13: Nome dos registradores no formato *memory dump* (à esquerda) e nome equivalente em *assembly* (à direita).

Da mesma forma que os registradores, as instruções também podem ter nomes diferentes no formato *memory dump* e em *assembly*. Contudo, em geral, as instruções possuem o mesmo nome, com algumas exceções. Por esse motivo, somente as instruções que tem nomes diferentes precisam ser informadas. O procedimento para a configuração dos nomes das instruções é semelhante ao dos registradores, mas a separação entre os dois nomes é realizada com um espaço, em vez de utilizar uma vírgula. Essa medida foi necessária, pois foi verificada a existência de mnemônicos de instruções contendo vírgulas no formato *memory dump*. A figura 3.14 apresenta um exemplo de instruções que precisaram de conversão de nomes.

[INSTRUCTIONS_TO_REPLACE]	
b, a b	
ba, a ba	
bne, a bne	
be, a be	
bg, a bg	
ble, a ble	
bge, a bge	
bl, a bl	
bgu, a bgu	
bleu, a bleu	

Figura 3.14: Nome das instruções no formato *memory dump* (à esquerda) e nome equivalente em *assembly* (à direita).

Os saltos incondicionais e os desvios precisam ser identificados. Um dos motivos se baseia na característica de algumas arquiteturas, o *branch delay slot*, onde as instruções de desvio são reordenadas e instruções subsequentes são sempre executadas. O reordenamento pode ocorrer durante a compilação ou durante a montagem. Se ocorrer durante a montagem, o reordenamento deve ser realizado na conversão das sub-rotinas, realizado pelo módulo de complementação. Portanto, é necessário informar, além das instruções de desvio, o número de instruções que são executadas após os desvios, da mesma forma que a apresentada no arquivo de configurações gerais.

A identificação dos saltos incondicionais e desvios condicionais deve ser realizada independentemente, pois haverá substituição do endereço físico de destino das instruções por um *label*. A identificação dos saltos incondicionais é ainda dividida em saltos para registrador e saltos para endereços de memória. Os saltos para registrador são retornos de sub-rotina e são mantidos da mesma forma. Os saltos para endereços de memória são convertidos em saltos para *labels*. A figura 3.15 apresenta essas três configurações.

[JUMPS_TO_REGISTER]	
jr, jalr	
[JUMPS_TO_MEMORY]	
j, jal	
[BRANCHES]	
beq, bneq, beqz, bnez, bgtz, bgez, bltz, blez	

Figura 3.15: Configurações dos mnemônicos das instruções de salto incondicional para registrador, salto incondicional para endereço de memória e desvios condicionais.

3.1.5 Configurações das técnicas

A seleção e configuração e seleção das técnicas de detecção em *software* a serem utilizadas é realizada através do arquivo de configurações das técnicas. As técnicas que serão aplicadas ao código *assembly* do programa devem ser configuradas conforme

apresentado na figura 3.16. O nome das técnicas deve ser informado na ordem em que serão aplicadas, separadas por vírgula.

```
[TECHNIQUES]
SIG, BRA, VAR3
```

Figura 3.16: Técnicas de detecção em *software* selecionadas.

Entre o valor original e as cópias dos dados armazenados em memória existe um deslocamento, esse deslocamento deve ser informado no arquivo de configurações das técnicas.

A sub-rotina de tratamento de erro utiliza um registrador para informar que um erro foi detectado. O nome do registrador e o valor que ele receberá caso um erro seja detectado deve estar informado no arquivo de configurações das técnicas. Um exemplo de configuração do registrador de erro é mostrado na figura 3.17. O registrador *\$11* será usado como registrador de erro pela sub-rotina de tratamento de erro. Caso um erro seja detectado, será atribuído o valor 1 a ele. Se o registrador *\$11* já estiver sendo usado pelo programa a ser protegido, todas as ocasiões dele serão substituídas no código *assembly* do programa por um registrador que não está sendo utilizado para que ele possa estar livre para a sub-rotina de tratamento de erro.

```
[ERROR_REGISTER]
$11

[ERROR_VALUE]
1
```

Figura 3.17: Técnicas de detecção em *software* selecionadas.

Nem sempre há registradores disponíveis suficientes para duplicar todos os registradores utilizados. Uma seleção dos registradores que serão duplicados pode aumentar consideravelmente a taxa de detecção de erros. Por esse motivo, um modo de prioridade de seleção de registradores deve ser escolhido. A CFT-tool implementa cinco modos de prioridade para seleção de registradores, são eles: *first*, *source*, *target*, *all* e *custom*. O modo *first* seleciona os registradores com base na ordem em que aparecem no código. O modo *source* realiza uma listagem de quantas vezes cada registrador foi utilizado como registrador fonte e aloca os registradores com base nessa lista, iniciando pelo mais utilizado. No modo *target*, os registradores são selecionados pelo número de vezes em que aparecem como registrador destino nas instruções, ou seja, o número de vezes que tem um novo valor atribuído no código *assembly*. O modo *all*, leva em conta o total de vezes que cada registrador aparece no código *assembly* do programa, quer seja como registrador fonte ou como registrador destino. Por fim, o modo *custom* permite ao usuário selecionar a ordem de prioridade desejada. Não há necessidade de listar todos os registradores utilizados. Os registradores não listados terão prioridade mais baixa que os listados e selecionados pela ordem que aparecerem no código. A figura 3.18 mostra um exemplo de modo de seleção de registradores. O modo selecionado foi *custom*. A ordem de prioridade dos registradores é *\$2*, *\$fp*, *\$3*, *\$4*, *\$sp*, *\$31* e *\$5*. Outros registradores que por ventura forem utilizados terão prioridade mais baixa que esses.


```

[ ] [PRIORITY_MODE]
    custom

[ ] [REGISTERS_BY_PRIORITY]
    $2, $fp, $3, $4, $sp, $31, $5

```

Figura 3.18: Exemplo de configuração do modo de seleção de registradores.

3.2 Módulo de Verificação

A função do Módulo de Verificação é ler o código *assembly* do programa gerado pelo compilador e verificar se o código confere com as especificações da arquitetura e organização do processador configurado. As informações referentes à arquitetura e organização do processador são informadas pelo Módulo de Configuração.

Primeiramente, o código *assembly* do programa é formatado para análise. O código original é lido e uma versão em arquivo em um arquivo temporário é criada, para preservar o código original do programa. Os comentários, os espaços em branco desnecessários e as linhas em branco são removidos. A figura 3.19 apresenta um exemplo de uma linha de código com comentário. O comentário deve ser configurado nos arquivos de configurações. A tag de comentário utilizada é passada pelo módulo de configuração. A tag e o texto da linha após a tag são removidos. Esse processo é realizado em todas as linhas do código. Em seguida, o código é novamente percorrido, sendo os espaços em branco desnecessário, após o último caractere de uma instrução ou qualquer espaço numa linha contendo um *label*, são removidos. As linhas em branco também são retiradas do código.

Configuração da tag de comentários:

```

[COMMENT_TAG]
#

```

Linha de código com comentário:

```

li $6,36          # 0x24

```

Figura 3.19: Tag de comentário configurada (acima) e linha de código com comentário (abaixo)

Depois da remoção dos comentários, espaços desnecessários e linhas em branco, o formato das instruções existentes no código é comparado com o formato informado nos arquivos de configuração. A figura 3.20 apresenta um exemplo de uma instrução retirada do código *assembly*. A instrução *addu \$6,\$2,80* possui o mnemônico da instrução, um espaço separando o mnemônico dos operandos, sendo eles separados por vírgula. O primeiro operando é um registrador destino, que vai receber o valor da execução da operação. O segundo é um registrador fonte, que terá seu valor usado na execução da instrução. E o terceiro é um imediato (constante), que também será usado como entrada na execução da instrução. Para que essa instrução seja válida, no arquivo de configuração referente às instruções deve estar informado a instrução com seu formato correto. O formato esperado para a instrução apresentada é *ins rd,rs,imm*. Se essa instrução estiver presente nas configurações e possuir esse formato, ela será

considerada válida. Caso contrário, a posição e o conteúdo da linha onde o problema foi detectado são adicionados a uma lista. Se, ao final do processo de verificação a lista não for vazia, indicando a ocorrência de divergência entre os formatos do código e da configuração, a lista é apresentada ao usuário, reportando os problemas, e a execução da CFT-tool é encerrada.

Instrução:	Configuração:
addu \$6,\$2,80	[GROUP] {INSTRUCTIONS} addu {FORMAT} ins rd,rs,imm {TYPE} arithmetic

Figura 3.20: Instrução analisada (à esquerda) e formato esperado para a instrução (à direita)

Quando um problema é detectado no processo de verificação do código *assembly*, normalmente o erro está nos arquivos de configurações, pois, é muito mais provável que o problema tenha ocorrido por erro do usuário ao configurar a CFT-tool do que o erro ser no código *assembly*. Se o código *assembly* foi gerado pelo compilador e um problema foi detectado pelo módulo de verificação, é muito provável que o erro tenha sido do usuário, pois é esperado que o compilador não esteja gerando um código *assembly* incorreto. O código pode estar errado se o código *assembly* passado para a ferramenta tenha sido compilado para uma arquitetura e outra tenha sido configurada, o que também pode ser caracterizado por um erro de configuração do usuário. Um problema no código *assembly* é possível caso ele tenha sido gerado manualmente. A ferramenta verifica e informa o erro, é função do usuário realizar a correção do problema.

Verificado que código *assembly* do programa a ser protegido confere com as especificações da arquitetura descrita, a CFT-tool inicia a execução do próximo módulo, o módulo de complementação, apresentado a seguir.

3.3 Módulo de Complementação

Durante o processo de compilação, onde um código *assembly* semanticamente igual ao código em linguagem de alto nível é gerado, o compilador faz uso de sub-rotinas pré-existentes e conhecidas pelo montador. Essas sub-rotinas, por serem conhecidas pelo montador, ou seja, estão presentes em bibliotecas, não tem seu código descrito em *assembly* antes dele passar pelo processo de ligação. As técnicas são aplicadas pela CFT-tool antes do processo de ligação.

Para que todo o programa possa ser protegido pelas técnicas de detecção em *software*, essas sub-rotinas não presentes no código pré-ligação devem ter seu código inserido na descrição em código *assembly* do programa.

Uma solução para esse problema é buscar o código equivalente da sub-rotina em um nível mais baixo de abstração, após o processo de ligação, e então convertê-la para

assembly. A opção, abaixo do código *assembly*, é o código de máquina. O código de máquina pode passar por um processo de desmontagem, realizado pelo desmontador (*disassembler*). O desmontador lê o código de máquina do programa e gera um código equivalente no formato *memory dump*. Esse código no formato *memory dump* é o código de máquina do programa descrito de uma forma mais legível. A figura 3.21 mostra um exemplo de trecho de código no formato *memory dump*. A primeira linha do exemplo tem função apenas informativa, indicando que uma chamada para sub-rotina foi realizada para esse endereço, o que indica que tal endereço marca o início de uma sub-rotina. As demais linhas são o código do programa propriamente dito. A primeira coluna indica o endereço físico onde a instrução se encontra. A segunda linha contém a palavra representada em hexadecimal. Para o processador do exemplo, se trata de uma palavra de 32 bits. Essa palavra de 32 bits é o código de máquina da instrução. A terceira coluna apresenta o código do programa de uma forma mais legível. Apesar de parecido com o código *assembly*, esse código na terceira coluna não é igual ao código *assembly*. As instruções podem estar fora de ordem, os registradores e instruções podem ter outros nomes e os saltos incondicionais e desvios condicionais são para endereços físicos, enquanto no código *assembly* é feito o uso de *labels*.

```

000002dc <main>:
 2dc: 27bdffd0   addiu  sp, sp, -48
 2e0: afbf002c   sw   ra, 44(sp)
 2e4: afbe0028   sw   s8, 40(sp)
 2e8: 03a0f021   move  s8, sp
 2ec: 3c020000   lui  v0, 0x0
 2f0: 8c420430   lw   v0, 1072(v0)
 2f4: 00000000   nop
 2f8: afc20010   sw   v0, 16(s8)
 2fc: 3c020000   lui  v0, 0x0
 300: 8c420434   lw   v0, 1076(v0)
 304: 00000000   nop
 308: 0c000040   jal  100 <encrypt>

```

Figura 3.21: Trecho de código no formato *memory dump*.

O código das sub-rotinas não presentes no código *assembly* pré-ligação não pode ser simplesmente retirado do arquivo *memory dump* e incluído no código *assembly* do programa, ele precisa ser tratado antes de ser incluído no código *assembly*.

A tarefa do módulo de complementação é gerar um código *assembly* completo, em que o código de tais sub-rotinas esteja incluído no código *assembly* do programa, para, com isso, a aplicação das técnicas de detecção em *software* poder ser aplicadas sobre o código completo do programa. O módulo de complementação percorre o código *assembly* do programa e verifica se há sub-rotinas não descritas no código *assembly* pré-ligação sendo utilizadas pelo programa. Caso haja, o módulo busca por essa mesma sub-rotina no código em formato *memory dump*, converte o código da sub-rotina para *assembly* e insere-a no código *assembly* do programa. Esse processo se repete até que o código de todas as sub-rotinas tenha sido incluído no código *assembly* do programa. A seguir, é apresentada a forma como são definidas as sub-rotinas a serem buscadas no arquivo *memory dump*.

3.3.1 Identificação de sub-rotinas não presentes no código pré-ligação

Para identificar as sub-rotinas não descritas no código pré-ligação, o módulo de complementação realiza uma varredura sobre o código *assembly* do programa buscando por instruções que realizam chamadas de sub-rotinas. O destino dessas instruções, que são nomes de sub-rotinas, é armazenado em uma lista. Na figura 3.22 podemos ver um exemplo de chamada para sub-rotina, realizada pela instrução *jal*, a qual chama a sub-rotina *encrypt*.

```

    addu    $2, $fp, 32
    move    $4, $2
    addu    $5, $fp, 16
    ➔ jal    encrypt
    move    $2, $0
    move    $sp, $fp
    lw     $31, 44($sp)

```

Figura 3.22: Exemplo de chamada para sub-rotina.

Em seguida, é realizada outra busca sobre o código *assembly* do programa, buscando, dessa vez, por *labels*, pois todo nome de sub-rotina é um *label*. Os *labels* são armazenados em outra lista. A figura 3.23 apresenta um exemplo de código *assembly*. As instruções foram suprimidas. No caso do programa do exemplo, a lista de *labels* teria os seguintes *labels*: *main*, *\$L2*, *\$L5*, *\$L4* e *\$L3*.

```

main:
    <instruções>
$L2:
    <instruções>
$L5:
    <instruções>
$L4:
    <instruções>
$L3:
    <instruções>

```

Figura 3.23: *Labels* presentes no código *assembly*.

A lista que contém as sub-rotinas chamadas é comparada com a lista de *labels*. Toda sub-rotina chamada que não estiver presente na lista de *labels* é identificada como uma sub-rotina não descrita no código pré-ligação. Um exemplo desse tipo de sub-rotina é apresentado na figura 3.24 (algumas instruções foram suprimidas). A sub-rotina *memcpy* foi chamada (instrução *jal*). No entanto, não há um *label* no código chamado *memcpy*, o que indica que se trata de uma sub-rotina não presente no código pré-ligação, a qual precisa ter seu código extraído do programa no formato *memory dump* e convertido para código *assembly*.

```

main:
    <instruções>
    move    $4,$2
    move    $5,$3
    li     $6,36
    ➔ jal memcpy
    sw     $0,96($fp)
    sw     $0,100($fp)
    <instruções>
$L2:
    <instruções>
$L3:
    <instruções>

```

Figura 3.24: Exemplo de sub-rotina não descrita no código pré-ligação.

Para cada uma dessas sub-rotinas, o módulo de complementação busca o código da sub-rotina na descrição do programa em formato *memory dump*, converte o código para *assembly* e o insere no código *assembly* do programa. A seguir é detalhado como essa conversão é realizada.

3.3.2 Conversão de código do formato *memory dump* para *assembly*

A conversão das sub-rotinas do formato *memory dump* para código *assembly* é realizada uma a uma, sendo que o código *assembly* gerado é inserido no código do código *assembly* do programa. Dessa forma, em vez do montador utilizar a sub-rotina pré-definida, ele utilizará a que está descrita no código *assembly*, durante o processo de montagem do código *assembly* para código de máquina.

A sub-rotina que deve ter o código convertido é localizada na descrição no formato *memory dump* e então convertida para *assembly*. Para realizar essa conversão, o módulo de complementação necessita saber informações sobre a arquitetura e organização do processador. Informações sobre as instruções e registradores, tanto para o código *assembly* quanto para o *memory dump*, devem estar configuradas nos arquivos de configuração. Para identificação dos *labels*, instruções, endereços de memória, registradores e comentários, dados sobre a formatação do arquivo também devem ser informados na configuração. Além disso, informações sobre o reordenamento de instruções devem ser informadas. O módulo de configuração se encarrega de interpretar os arquivos de configuração e servir como base de informações para o módulo de complementação.

Na figura 3.25 pode ser visto um exemplo de trecho de código de uma sub-rotina descrita no formato *memory dump* e seu código *assembly* equivalente gerado pelo módulo de complementação. A sub-rotina é localizada na descrição em *memory dump*. Seu código é extraído, sendo as colunas (endereço físico, palavra em hexadecimal e instrução) separadas em listas diferentes. Os endereços físicos de destino de saltos incondicionais e desvios condicionais são substituídos por *labels*. Tais *labels* são também adicionados nas posições de destino do desvio, ou seja, uma linha contendo o *label* criado é adicionada imediatamente antes da posição que contém o destino do

desvio. Os comentários são removidos. Em seguida, os registradores e as instruções tem seu nome convertido para o equivalente em código *assembly*. Devido ao *branch delay slot*, no processo de conversão do formato *memory dump* para *assembly*, as instruções devem ser postas em suas posições originais. No caso do exemplo da figura, sempre uma instrução depois dos saltos incondicionais e desvios condicionais é executada. Portanto, para manter a corretude do programa, as instruções de desvio devem ser movidas para baixo, sem, no entanto, mudar de um bloco de código para outro, isto é, sem cruzar por *labels*. Após isso, dependendo de características da arquitetura e do montador, as instruções de não operação podem ser removidas do código, visando otimizá-lo, pois o montador irá inseri-las quando necessário.

<pre> 00000300 <memcpy>: 300: 00a41025 or v0,a1,a0 304: 30420003 andi v0,v0,0x3 308: 14400026 bnez v0,3a4 <memcpy+0xa4> 30c: 00805021 move t2,a0 310: 00064102 srl t0,a2,0x4 314: 00a04821 move t1,a1 318: 30c6000f andi a2,a2,0xf 31c: 1100000d beqz t0,354 <memcpy+0x54> 320: 00803821 move a3,a0 324: 8d220000 lw v0,0(t1) 328: 8d230004 lw v1,4(t1) 32c: 8d240008 lw a0,8(t1) 330: 8d25000c lw a1,12(t1) 334: 2508ffff addiu t0,t0,-1 338: ace20000 sw v0,0(a3) 33c: ace30004 sw v1,4(a3) 340: ace40008 sw a0,8(a3) 344: ace5000c sw a1,12(a3) 348: 25290010 addiu t1,t1,16 34c: 1500ffff bnez t0,324 <memcpy+0x24> 350: 24e70010 addiu a3,a3,16 354: 00064082 srl t0,a2,0x2 358: 11000007 beqz t0,378 <memcpy+0x78> 35c: 30c60003 andi a2,a2,0x3 360: 8d220000 lw v0,0(t1) 364: 2508ffff addiu t0,t0,-1 </pre>	<pre> memcpy: or \$2,\$5,\$4 andi \$2,\$2,0x3 move \$10,\$4 bnez \$2,memcpy1 srl \$8,\$6,0x4 move \$9,\$5 andi \$6,\$6,0xf move \$7,\$4 beqz \$8,memcpy2 memcpy3: lw \$2,0(\$9) lw \$3,4(\$9) lw \$4,8(\$9) lw \$5,12(\$9) addiu \$8,\$8,-1 sw \$8,\$8(\$7) sw \$2,0(\$7) sw \$3,4(\$7) sw \$4,8(\$7) sw \$5,12(\$7) addiu \$9,\$9,16 addiu \$7,\$7,16 bnez \$8,memcpy3 memcpy2: srl \$8,\$6,0x2 andi \$6,\$6,0x3 beqz \$8,memcpy4 memcpy5: lw \$2,0(\$9) addiu \$8,\$8,-1 </pre>
--	---

Figura 3.25: Sub-rotina extraída da descrição no formato *memory dump* (à esquerda) e código *assembly* equivalente gerado pelo módulo de complementação (à direita).

Pode-se ver que o código *assembly* gerado pelo módulo de complementação e o código descrito no formato *memory dump*, o qual foi gerado por um programa desmontador, são bastante semelhantes. Contudo, vale salientar alguns pontos. Primeiramente, ambos os casos possuem o mesmo mnemônico para as instruções. Isso não necessariamente é válido, mas, em geral, o mnemônico das instruções é o mesmo nos dois casos. Contudo, os nomes dos registradores são diferentes. Além disso, um fator relevante está no endereço de destino dos saltos ou desvios condicionais. Para manter a corretude do código e, ao mesmo tempo, fazer proveito da flexibilidade do código *assembly* em relação ao código de máquina, são inseridos labels nos lugares dos endereços físicos, tanto nas instruções de salto incondicional ou desvio condicional, quanto no endereço de destino dessas instruções de desvio. Outro fator importante consiste que, para o processador do exemplo, as instruções subsequentes às instruções de desvio são sempre executadas, mesmo quando o desvio é tomado. Portanto, é necessário que tais instruções de desvio sejam trocadas de posição com a instrução seguinte no processo de conversão para *assembly*. Porém, vale ressaltar que as instruções devem ser mantidas em seu bloco de código, ou seja, nenhuma instrução que está após um *label* pode ir para antes desse ou o inverso, nenhuma instrução antes de um *label* pode ir para depois dele.

Após todas as sub-rotinas terem sido convertidas e inseridas no código *assembly* do programa, o processo é repetido, pois novas sub-rotinas não descritas no código pré-ligação podem estar sendo utilizadas pelas sub-rotinas recém inseridas no código *assembly*. Esse processo deve se repetir até que não sejam mais encontradas sub-rotinas não descritas no código pré-ligação.

3.4 Módulo de Proteção a Falhas Transientes

O módulo de proteção a falhas transientes, ou Single Event Effect (SEE), é responsável por gerenciar a aplicação das técnicas de detecção em *software* sobre o código *assembly* do programa a ser protegido. Ele aplica as técnicas conforme as configurações do usuário e insere a sub-rotina de tratamento de erro. As configurações sobre as técnicas e a aplicação delas são informadas pelo módulo de configuração. O módulo de proteção a SEEs também precisa de informações sobre a arquitetura e organização do processador, essas informações também são passadas pelo módulo de configuração.

A figura abaixo apresenta uma visão geral de como o módulo de proteção a SEEs funciona. Com base nas configurações informadas pelo usuário nos arquivos de configurações, as quais foram carregadas pelo módulo de configuração e transmitidas para o módulo de proteção a SEEs, o código *assembly* completo do programa, gerado pelo módulo de complementação é lido. Transformações são realizadas sobre ele, sendo as técnicas de detecção em *software*, que foram selecionadas, aplicadas sobre o código do programa. Como resultado desse processo, é gerado um código *assembly* protegido, com a mesma funcionalidade do programa original e a capacidade de detecção de erros das técnicas aplicadas, porém, com tempo de execução e área ocupada em memória diferentes.

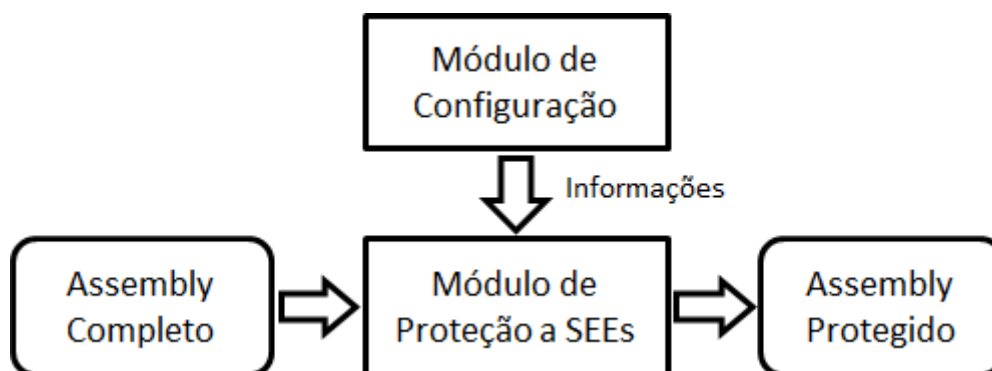


Figura 3.26: Visão geral do módulo de proteção a SEEs.

O código *assembly* original do programa a ser protegido poderia ser utilizado como entrada para o módulo de proteção a SEEs. Contudo, o programa não estaria completamente protegido, pois os códigos das sub-rotinas não presentes no código pré-ligação não seriam protegidas. O código *assembly* completo, gerado pelo módulo de complementação, contém essas sub-rotinas, e, por essa razão, é utilizado como entrada pelo módulo de proteção a SEEs. Transformações são realizadas sobre esse código *assembly* completo. As técnicas de detecção em *software* são aplicadas, resultando em um código *assembly* protegido.

As técnicas de detecção em *software* podem ser aplicadas em diferentes ordens. A ordem em que as técnicas selecionadas são aplicadas é informada através dos arquivos

de configurações e passada do módulo de configuração para o módulo de proteção a SEEs. O módulo de proteção a SEEs pega informações referentes às instruções, aos registradores, *labels*, e demais informações necessárias para o entendimento do código *assembly* do programa e, com isso, aplicar as técnicas de detecção em *software* sobre o código do programa. A seguir, são apresentadas as técnicas de detecção em *software* implementadas na CFT-tool.

3.5 Técnicas de Detecção em Software Implementadas

As técnicas de detecção em *software* implementadas na CFT-tool foram validadas por Azambuja (2010). Tais técnicas foram adaptadas para serem aplicadas sobre o código *assembly* do programa. Um exemplo dessas adaptações consiste na questão de saltos e desvios condicionais, os quais, em código de máquina precisam ter o valor do endereço de destino corrigido após a proteção, enquanto em *assembly* não necessitam de correção, pois os saltos e desvios são realizados para *labels*, os quais tem seu endereço definido após o processo de montagem do código *assembly* para o código de máquina.

Dentre as técnicas de detecção adotadas, as destinadas à proteção dos dados são as técnicas Variáveis 1, Variáveis 2 e Variáveis 3. Como técnicas para proteção do fluxo de controle do programa foram adotadas as técnicas Assinaturas e Branches.

3.5.1 Variáveis 1

A técnica Variáveis 1 (VAR1) (AZAMBUJA, 2010) é baseada nas regras de Rebaudengo (1999) destinadas a proteção dos dados, onde todas as variáveis são duplicadas, sendo realizadas sobre as cópias, as mesmas operações que foram realizadas sobre as variáveis originais. Além disso, são realizadas comparações entre as variáveis originais e suas respectivas cópias todas as vezes que essas forem lidas.

Um exemplo da aplicação da técnica VAR1 em código *assembly* é apresentado na figura 3.27. Para cada registrador, uma cópia é criada, sendo que as operações aplicadas sobre o registrador original também são aplicadas sobre a cópia. Um deslocamento é utilizado para que as operações nos dados da memória também sejam efetuadas sobre cópias. Instruções para comparar os registradores originais com as cópias são inseridas antes de todas as instruções onde o registrador é lido.

1:	bne \$5,\$15,error
2: lw \$2,0(\$5)	lw \$2,0(\$5)
3:	lw \$12,160(\$15)
4:	bne \$2,\$12,error
5:	bne \$3,\$13,error
6: addu \$2,\$3,\$2	addu \$2,\$3,\$2
7:	addu \$12,\$13,\$12
8:	bne \$2,\$12,error
9:	bne \$5,\$15,error
10: sw \$2,4(\$5)	sw \$2,4(\$5)
11:	sw \$12,164(\$15)

Figura 3.27: Código original (à esquerda) e código com a técnica VAR1 (à direita).

As linhas 2, 6 e 10 são instruções do código original. As instruções 3, 7 e 11 são cópias das instruções 2, 6 e 10, respectivamente. As demais instruções, das linhas 1, 4, 5, 8 e 9 são instruções que comparam os valores dos registradores lidos pelas instruções posteriores. O registrador \$2, utilizado na linha 2, não foi verificado, pois ele não foi lido pela instrução *lw*, a qual carrega um valor da memória em um registrador. Na verdade, um valor foi atribuído ao registrador \$2 pela execução dessa instrução. Essa mesma explicação serve para a não ocorrência de uma dupla checagem do registrador \$2 na verificação da instrução da linha 6. Caso um erro seja detectado por uma instrução de verificação, o fluxo do programa é desviado para uma sub-rotina de tratamento de erro.

3.5.2 Variáveis 2

Variáveis 2 (VAR2) (AZAMBUJA, 2010) implementa uma alternativa à técnica de proteção aos dados de Rebaudengo (1999), baseada na proposta de Velazco (2007). A diferença entre a técnica VAR2 e a técnica VAR1 consiste no local onde as instruções para verificação das variáveis originais e cópias são inseridas. Enquanto em VAR1 as variáveis são comparadas com suas cópias sempre antes de serem lidas, em VAR2 as variáveis são compradas após alguma operação de escrita sobre elas.

Na figura 3.28 pode ser visto um exemplo da aplicação da técnica VAR2 em código *assembly*. Para cada registrador, uma cópia é criada, sendo que as operações aplicadas sobre o registrador original também são aplicadas sobre a cópia. Um deslocamento é utilizado para que as operações nos dados da memória também sejam efetuadas sobre cópias. Instruções para comparar os registradores originais com as cópias são inseridas após todas as instruções onde um novo valor é atribuído ao registrador. Quando não há escrita sobre algum registrador na instrução, são inseridas instruções de comparação dos registradores lidos antes da instrução.

1: <i>lw</i> \$2, 0(\$5)	<i>lw</i> \$2, 0(\$5)
2:	<i>lw</i> \$12, 160(\$15)
3:	<i>bne</i> \$2, \$12, error
4: <i>addu</i> \$2, \$3, \$2	<i>addu</i> \$2, \$3, \$2
5:	<i>addu</i> \$12, \$13, \$12
6:	<i>bne</i> \$2, \$12, error
7:	<i>bne</i> \$2, \$12, error
8:	<i>bne</i> \$5, \$15, error
9: <i>sw</i> \$2, 4(\$5)	<i>sw</i> \$2, 4(\$5)
10:	<i>sw</i> \$12, 164(\$15)

Figura 3.28: Código original (à esquerda) e código com a técnica VAR2 (à direita).

As instruções do código do programa original são apresentadas nas linhas 1, 4 e 9. As instruções das linhas 2, 5 e 10 são suas respectivas cópias. As demais instruções, das linhas 3, 6, 7 e 8 são instruções que checam os valores dos registradores. As instruções das linhas 3 e 5 verificam os valores de registradores que sofreram uma operação de escrita e as instruções das linhas 7 e 8 checam os valores dos registradores lidos pela operação de escrita em memória, *sw*, devido ao fato dessa instrução não atribuir valor a algum registrador. Os verificadores das linhas 6 e 7 são iguais, permitindo, com a remoção de uma das linhas, a otimização do programa sem diminuição da cobertura de

erros. Caso um erro seja detectado, o fluxo do programa é desviado para uma sub-rotina de tratamento de erro.

3.5.3 Variáveis 3

A técnica Variáveis 3 (VAR3) (AZAMBUJA, 2010) é baseada na parte destinada a proteção dos dados da técnica SWIFT (REIS, 2005). Toda variável deve ser duplicada e toda operação aplicada sobre a variável original deve ser também aplicada sobre sua cópia. Uma verificação entre o valor da variável original e de sua cópia é realizado antes dessas serem utilizadas por instruções de acesso a memória ou por desvios.

A figura a seguir apresenta um exemplo da aplicação da técnica VAR3 em código *assembly*. Para cada registrador, uma cópia é criada, sendo que as operações aplicadas sobre o registrador original também são aplicadas sobre a cópia. Um deslocamento é utilizado para que as operações nos dados da memória também sejam efetuadas sobre cópias. Pode-se notar que antes da instrução *lw*, a qual carrega um valor da memória para o registrador, e antes da operação de escrita em memória, *sw*, são realizadas verificações sobre os registradores lidos por essas operações.

1:	<code>bne \$5,\$15,error</code>
2: <code>lw \$2,0(\$5)</code>	<code>lw \$2,0(\$5)</code>
3:	<code>lw \$12,160(\$15)</code>
4: <code>addu \$2,\$3,\$2</code>	<code>addu \$2,\$3,\$2</code>
5:	<code>addu \$12,\$13,\$12</code>
6:	<code>bne \$2,\$12,error</code>
7:	<code>bne \$5,\$15,error</code>
8: <code>sw \$2,4(\$5)</code>	<code>sw \$2,4(\$5)</code>
9:	<code>sw \$12,164(\$15)</code>

Figura 3.29: À esquerda o código original e à direita o código com a técnica VAR3.

As instruções do código do programa original são apresentadas nas linhas 2, 4 e 8. Suas respectivas cópias são apresentadas nas linhas 3, 5 e 9. As instruções de verificação são apresentadas nas linhas 1, 6 e 7. Caso uma discrepância entre os valores dos registradores comparados seja encontrada, é executada uma sub-rotina de tratamento de erro. Os registradores utilizados pela instrução aritmética da linha 4, *addu*, não foram verificados antes ou após a execução da instrução, pois essa técnica verifica somente acessos à memória e desvios.

3.5.4 Branches

A técnica BRA realiza a replicação das instruções de desvio, tanto para caso o desvio seja tomado quanto para caso o desvio não seja tomado. No endereço de destino para o caso do desvio ser tomado é inserida uma instrução logicamente inversa à instrução original. E no local para onde o fluxo de execução do programa vai caso o desvio não seja tomado é inserida uma instrução logicamente igual à original. Ambas as réplicas desviam o fluxo de execução do programa para uma rotina de tratamento de erro caso sua comparação seja verdadeira.

Na figura 3.30 pode-se ver a aplicação da técnica BRA em código *assembly*. É possível notar que a instrução de desvio foi replicada para seus dois casos, sendo que no

caso negativo a cópia é exatamente igual à instrução original e no caso de o desvio ser tomado, a instrução cópia inserida é logicamente inversa à instrução original.

1: bne \$2, \$0, \$L9	bne \$2, \$0, \$L9_1
2:	bne \$2, \$0, error
3: j \$L7	j \$L7
4:	j \$L9
5:	\$L9_1:
6:	beq \$2, \$0, error
7:	j \$L9
8: \$L9:	\$L9:
9: lw \$2, 60(\$fp)	lw \$2, 60(\$fp)

Figura 3.30: À esquerda o código original e à direita o código com a técnica BRA.

Aproveitando-se da flexibilidade do *assembly* em relação ao código de máquina, são inseridos blocos de tratamento para cada desvio que tenha como destino o mesmo bloco de código (*label*). Desse modo, é evitado que desvios diferentes que têm o mesmo destino acabem gerando um problema. Por exemplo, caso houvesse outra instrução de desvio indo para o *label* \$L9, seria criado um bloco com *label* \$L9_2 para seu tratamento.

No exemplo apresentado, as linhas 1, 3, 8 e 9 representam o código do programa original, onde a linha 8 contém um *label* e as outras 3 linhas, instruções. O destino do desvio da linha 1 é alterado para o bloco de tratamento \$L9_1, apresentado na linha 5. O desvio condicional logicamente igual ao desvio original (linha 1) é incluído logo abaixo do original (linha 2). O desvio logicamente inverso é a primeira instrução dentro do bloco de tratamento (linha 6). As instruções das linhas 4 e 7 são saltos (desvios incondicionais) e são usadas para manter a corretude do código. Após a aplicação da técnica em todo o código, o salto do último bloco de tratamento pode ser retirado, visto que logo em seguida vem o destino do salto. Além disso, é muito comum ocorrerem saltos ao final de cada bloco de código, ou seja, imediatamente antes do início de outro bloco, como pode ser visto no exemplo acima na linha 3 (instrução *j* \$L7). Nesses casos, o salto inserido no bloco de código original para manter a corretude do programa (linha 4) também pode ser removido.

3.5.5 Assinaturas

Como a maioria das técnicas destinadas à proteção do fluxo de controle do programa, a técnica Assinaturas (SIG) (AZAMBUJA, 2010) divide o programa em blocos básicos e associa assinaturas a cada bloco. A assinatura do bloco em execução é atribuída a um registrador no início da execução do bloco e testada ao final.

A figura 3.31 apresenta um exemplo da aplicação da técnica SIG em código *assembly*. Os desvios condicionais (instrução *bne*) e os saltos incondicionais (instrução *j*) ficam fora dos blocos básicos. Eles marcam, juntamente com o destino dos desvios (*labels*), o início e/ou o fim de um bloco básico.

1:	bne \$2,\$9,\$L9	bne \$2,\$9,\$L9
2:		mov \$11,1
3:	lw \$2,60(\$fp)	lw \$2,60(\$fp)
4:	add \$2,\$3,\$2	add \$2,\$3,\$2
5:		bne \$11,1,error
6:	\$L9:	\$L9:
7:		mov \$11,2
8:	sw \$2,0(\$5)	sw \$2,0(\$5)
9:		bne \$11,2,error
10:	j \$L7	j \$L7

Figura 3.31: Código original (à esquerda) e código com a técnica BRA (à direita).

O código do programa original é apresentado nas linhas 1, 3, 4, 6, 8 e 10. A linha 6 contém um *label* e as demais, instruções. As linhas 2 e 7 marcam o início de blocos básicos, nelas é atribuído o valor da assinatura de seu bloco básico a um registrador, no caso, o registrador \$11. As instruções da linha 5 e 9 verificam se o valor da assinatura contido no registrador é condizente com o valor esperado para aquele bloco básico, checando, dessa forma, se não houve um desvio no fluxo de execução do programa de um bloco básico para outro. Caso o valor do registrador seja diferente da assinatura do bloco básico em execução, uma sub-rotina de tratamento de erro é executada.

4 RESULTADOS E VALIDAÇÃO

Para validar a ferramenta CFT-tool, primeiramente é necessário validar o correto funcionamento e execução do código antes e após a alteração. Isso mostra que a inclusão das técnicas de detecção a falhas não alterou o funcionamento e os resultados esperados do *software*. Após essa validação, faz-se necessário analisar a eficiência das técnicas de detecção em *software* aplicadas pela ferramenta no código *assembly* do programa. O resultado do tempo de execução, ocupação de memória e taxa de detecção de erros deve ser compatível com resultados presentes na literatura.

A configurabilidade da CFT-tool, isto é, a capacidade dela de aplicar as técnicas de detecção em *software* em programas compilados para processadores de diferentes arquiteturas, também deve ser validada. Essa validação é realizada configurando a CFT-tool para um processador com arquitetura diferente da usada para verificar a eficácia das técnicas de detecção em *software*, aplicando elas sobre os programas compilados para o processador dessa outra arquitetura.

4.1 Processadores Utilizados

Dois processadores foram utilizados para validar a CFT-tool, miniMIPS e LEON3. O miniMIPS foi escolhido devido à existência, na literatura, de uma ferramenta que aplica as mesmas técnicas de detecção em *software* CFT-tool. Dessa forma, a correteza e eficiência da aplicação das técnicas pela CFT-tool podem ser verificadas. O processador LEON3 foi selecionado devido a sua ampla utilização em aplicações espaciais. Com ele será validada a configurabilidade da CFT-tool, isto é, confirmar que ela é independente de arquitetura, gerando um código funcional com técnicas de detecção em *software* aplicadas. A seguir são apresentadas as características relevantes, para a CFT-tool, dos processadores utilizados.

4.1.1 miniMIPS

O processador miniMIPS é um processador de 32 bits baseado na arquitetura MIPS. Ele implementa um conjunto reduzido de instruções dessa arquitetura e tem um *pipeline* de 5 estágios. Esse processador foi selecionado para a validação da CFT-tool, pois há na literatura uma ferramenta que implementa as técnicas de detecção em *software* que podem ser aplicadas pela CFT-tool. Dessa forma, é possível comparar a correteza da aplicação das técnicas pela CFT-tool e a eficiência das técnicas sendo aplicadas no código *assembly*.

As instruções possuem tamanho fixo de 32 bits. O processador miniMIPS é capaz de executar um total de 52 instruções da arquitetura MIPS. Por esse motivo, o

processador miniMIPS é baseado na arquitetura MIPS, implementando um conjunto reduzido de instruções dessa arquitetura.

As instruções de desvio possuem uma importante função na aplicação das técnicas de detecção em *software* pela CFT-tool. A forma como as comparações e desvios são realizados pelo processador são significativos. O processador miniMIPS realiza a comparação dos valores de registradores e o desvio na mesma instrução. Isto é, os valores dos registradores são comparados e, conforme o resultado dessa comparação, o desvio é tomado ou não. Um exemplo desse tipo de instrução é apresentado na figura 4.1. A instrução *bne* (*branch on not equal*) compara os registradores \$2 e \$0. O desvio para o *label* \$L5 será tomado se os dois registradores não forem iguais.

```
bne $2, $0, $L5
```

Figura 4.1: Comparação e desvio no processador miniMIPS.

O processador miniMIPS possui um conjunto de 32 registradores de 32 bits. Como não há janela de registradores, todos os registradores são configurados como globais, sendo que os registradores \$0 e \$1 não podem sofrer operações de escrita. O registrador \$0 por ter seu valor sempre constante em zero e o registrador \$1 por ter seu uso reservado para o montador. O conjunto de registradores do processador miniMIPS pode ser visto na figura 4.2.

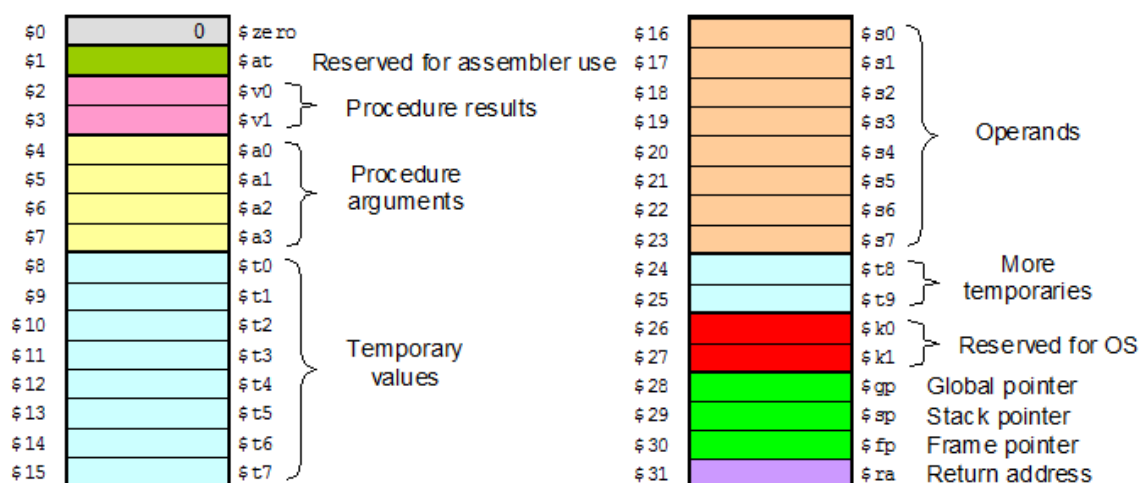


Figura 4.2: Registradores do processador miniMIPS.

Um programa descrito em uma linguagem de alto nível como C, por exemplo, passa por vários estágios até que possa ser executado pelo processador. O processador miniMIPS do estudo de caso está descrito em VHDL, uma linguagem de descrição de *hardware*, e executa os programas informados num arquivo no formato COE, por simulação, sendo utilizada, para isso, a ferramenta ModelSim. O formato COE consiste em uma cadeia de zeros e uns no formato texto, onde cada palavra de 32 bits é descrita por linha. Para o programa descrito em uma linguagem de alto nível chegar a esse formato, é necessário que ele passe por vários estágios. Primeiramente o programa é compilado, sendo gerado o código em *assembly* do programa. É sobre o código *assembly* que a CFT-tool trabalha, realizando modificações no código do programa e inserindo as técnicas de detecção em *software* escolhidas. O código *assembly* passa pelo montador, gerando o código de máquina do programa. Em seguida, o código de máquina é desmontado, processo realizado pelo desmontador. A saída do desmontador é

o código de máquina do programa no formato *memory dump*. Um programa auxiliar de conversão foi desenvolvido para retirar os valores hexadecimais da descrição do programa no formato *memory dump* e os converte para binário, escrevendo num arquivo em formato texto. Esse arquivo gerado é o programa no formato COE. A ferramenta HPCT Suite, usada para comparar e validar a CFT-tool trabalha sobre o formato COE. Na figura 4.3 são apresentadas as transformações por qual o programa passa até poder ser executado pelo processador miniMIPS.

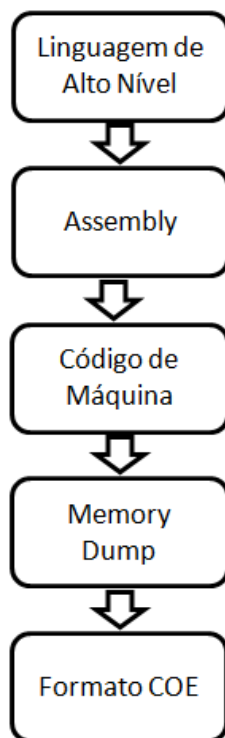


Figura 4.3: Transformações do programa para execução no processador miniMIPS.

4.1.2 LEON3

O processador LEON3 é um processador de 32 bits que implementa a arquitetura SPARC V8, amplamente utilizado em aplicações espaciais. O processador LEON3 possui um *pipeline* de 7 estágios. Esse processador foi selecionado para validar a configurabilidade da CFT-tool. Ele foi escolhido por dois motivos: é baseado em uma arquitetura diferente do processador miniMIPS e é amplamente utilizado em aplicações espaciais. Além disso, a disponibilidade do processador em VHDL permite a simulação lógica de injeção de falhas.

O processador LEON3 implementa as 72 instruções da arquitetura SPARC V8, O que torna esse processador compatível com a arquitetura SPARC V8. As instruções são de 32 bits.

A forma como comparações e desvios condicionais são realizados no LEON3 é diferente do processador miniMIPS. Essa informação é de vital importância para a aplicação das técnicas de detecção em *software* pela CFT-tool. Enquanto no miniMIPS, uma instrução é responsável por realizar a comparação e o desvio, no processador LEON3 essa tarefa é dividida em duas instruções. Uma instrução que compara dois registradores ou um registrador e um valor imediato e outra instrução de desvio condicional que se baseia no resultado da última comparação realizada, o qual fica

armazenado. Na figura 4.4 é apresentado um exemplo da instrução de comparação. A instrução *cmp* compara os registradores *%g2* e *%g1* e armazena internamente o resultado da comparação.

```
cmp %g2, %g1
```

Figura 4.4: Instrução de comparação para o processador LEON3.

Os desvios condicionais são realizados separadamente. A figura 4.5 apresenta um exemplo de instrução de desvio condicional. A instrução *ble* verifica se o valor armazenado da última comparação realizada indica se o registrador avaliado é menor, igual ou maior que o outro. Se for menor ou igual, o desvio é tomado. O destino do desvio do exemplo, caso ele seja tomado, é indicado pelo *label.LL4*.

```
ble .LL4
```

Figura 4.5: Instrução de desvio condicional para o processador LEON3.

O processador LEON3 implementa janela de registradores. Cada conjunto de registradores contém 8 registradores locais, 8 registradores de entrada e 8 registradores de um conjunto adjacente, os quais são chamados na janela atual como registradores de saída, conforme apresentado na figura 4.6. Em dado momento, uma instrução pode acessar os 8 registradores globais e outros 24 registradores, 8 locais, 8 de entrada e 8 de saída.

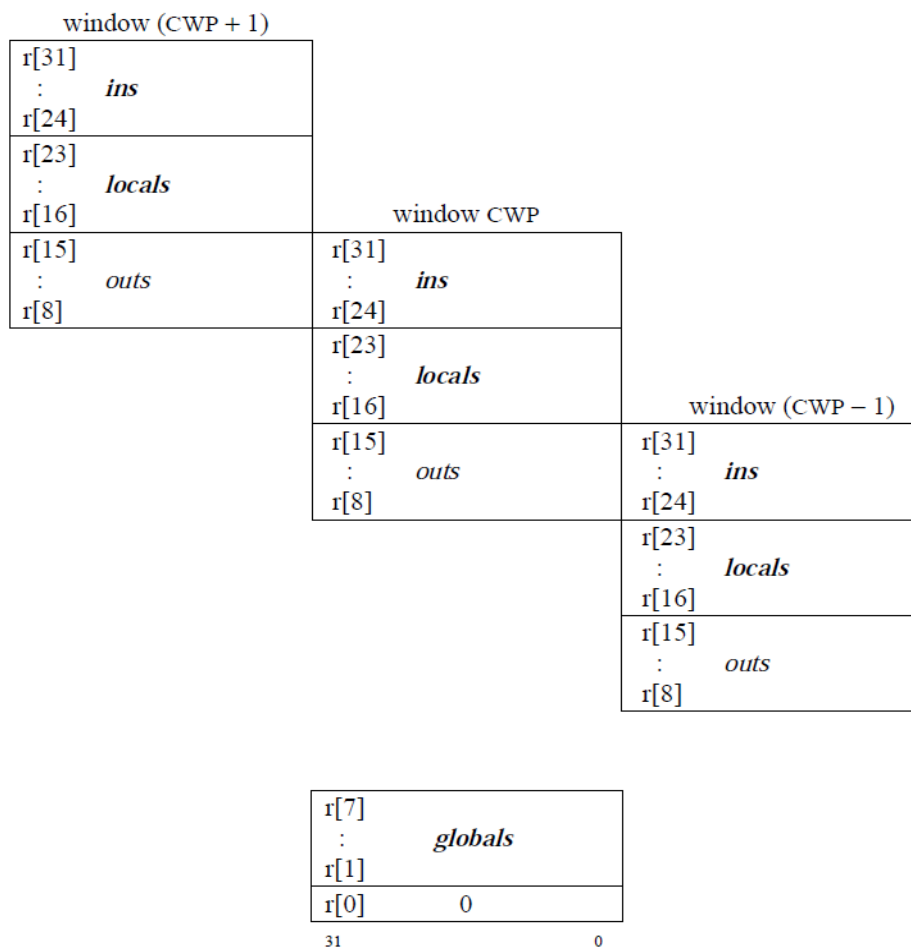


Figura 4.6: Relação dos registradores do processador LEON3 (SPARC, 1991).

O número de janelas ou conjuntos de registradores especificados na arquitetura SPARC V8 pode variar de 2 a 32 janelas, dependendo da implementação. Neste trabalho, foi utilizada a quantidade padrão de 8 janelas de registradores.

A comunicação entre as janelas de registradores é realizada utilizando os registradores de entrada e de saída. A figura 4.7 indica como esse processo é realizado. Os registradores são apresentados em conjuntos, sendo seu tipo indicado em negrito. Os registradores de entrada são representados pelo nome *ins*, os de saída pelo nome *outs* e os locais pelo nome *locals*. Os registradores globais não estão representados na figura. São implementadas 8 janelas de registradores. Se a janela corrente é a *w0* e é executada a instrução *restore*, a janela *w1* passa a ser a janela corrente. Agora, caso a janela corrente seja a janela *w0* e seja executada a instrução *save*, ocorrerá uma interrupção devido a *overflow* nas janelas de registradores. Os registradores de saída de uma janela são os registradores de entrada da janela adjacente.

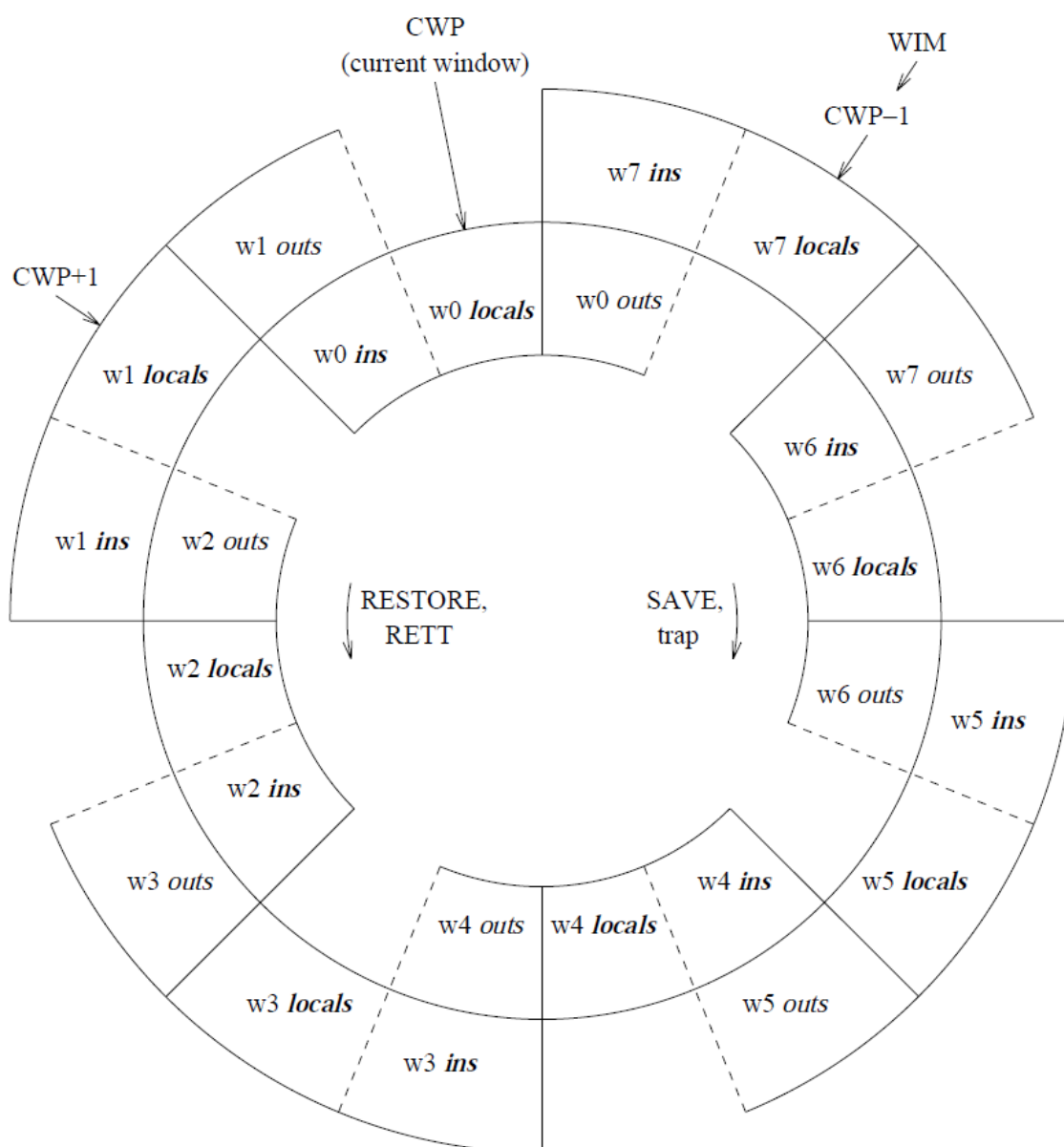


Figura 4.7: Janela de registradores do processador LEON3 (SPARC, 1991).

A versão VHDL do processador LEON3 utilizado nesse trabalho é disponibilizada pela Aeroflex Gaisler sob uma licença GNU GPL. Para que um programa descrito em uma linguagem de alto nível, como C, por exemplo, possa ser executado pelo processador LEON3 através de simulação lógica, rodando a descrição de *hardware* do processador LEON3 na ferramenta de simulação ModelSim, é necessário que o programa passe por vários estágios. O código em linguagem de alto nível é compilado, sendo gerado o código *assembly* equivalente. É sobre o código *assembly* que a CFT-tool realiza as transformações de código e aplica as técnicas de detecção em *software*. O código *assembly* é então compilado, sendo gerado o código de máquina. Por fim, é gerado um arquivo no formato SREC, o qual é executado pela versão em VHDL do processador LEON3. Essas transições podem ser vistas na figura 4.8.

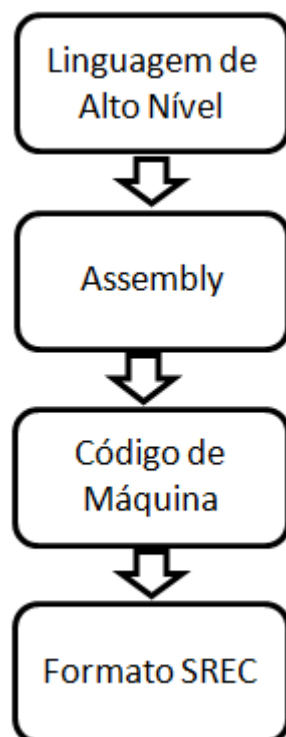


Figura 4.8: Fluxo de compilação para execução no processador LEON3.

O formato SREC consiste de uma série de caracteres ASCII, conforme apresentado na figura 4.9. Os registros estão em hexadecimal, são separados por linha e têm a seguinte estrutura:

1. O caractere S marca o início de código.
2. O segundo caractere indica o tipo do registro, conforme apresentado na tabela 4.1, podendo ser um dígito de 0 a 9.
3. Dois dígitos indicam, em hexadecimal, o número de *bytes* existentes após eles até o final do registro.
4. Quatro, seis ou oito dígitos hexadecimais indicam a posição na memória do primeiro byte do dado. A quantidade de dígitos para o endereço é definida conforme o tipo de dado selecionado.
5. Sequência de dígitos contendo os dados.
6. Dois dígitos para verificação da integridade do dado.

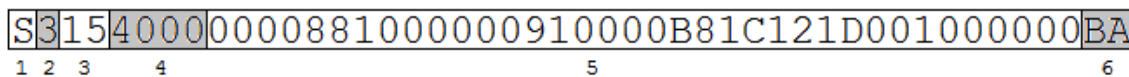


Figura 4.9: Formato SREC.

O registro do tipo S0 contém dados com especificações do fabricante. Os registros dos tipos S1, S2 e S3 contêm sequências de dados, variando somente o tamanho do endereço necessário. S1 usa um sistema de 16 bits, S2 usa 24 bits e S3 usa 32 bits. O registro do tipo S5 conta o número de registros anteriores a ele. Os demais registros, dos tipos S7, S8 e S9, contêm o endereço de início do programa.

Tabela 4.1: Tipos de dados do formato SREC

Tipo	Descrição	Endereço (bytes)	Sequência de Dados
S0	Início de bloco	2	Sim
S1	Sequência de dados	2	Sim
S2	Sequência de dados	3	Sim
S3	Sequência de dados	4	Sim
S5	Contador de registros	2	Não
S7	Fim de bloco	4	Não
S8	Fim de bloco	3	Não
S9	Fim de bloco	2	Não

4.2 Metodologia de Injeção de Falhas

Falhas transientes foram injetadas no nível lógico dos processadores, através da ferramenta Injetor de Falhas (AZAMBUJA, 2010) que injeta falhas automaticamente utilizando o simulador ModelSim (MENTOR GRAPHICS, 1999). O Injetor de Falhas modifica o valor lógico de algum sinal do processador durante um tempo determinado. Tanto o sinal afetado, quanto o momento em que a falha ocorre são aleatórios. Como se deseja verificar a eficiência da detecção das técnicas de detecção em *software* aplicadas pela CFT-tool, somente as falhas que causaram erro foram consideradas na análise. As falhas que não ocasionaram em erro, ou seja, aquelas mascaradas pela lógica do processador foram desconsideradas. O erro é determinado verificando se o resultado gerado pelo programa em execução é condizente, ou não, com o resultado esperado para o programa.

As falhas foram injetadas aleatoriamente no tempo e espaço na descrição RTL dos processadores miniMIPS e LEON3. O instante da ocorrência ou injeção da falha é definido entre o início e o fim da execução do programa e a duração da falha utilizada é de um ciclo de relógio. O local onde a falha é inserida, ou seja, o sinal de saída de uma porta lógica que pode ser ou não um flip-flop, também é selecionado aleatoriamente em uma lista contendo os sinais do processador. A falha é caracterizada pela inversão do valor lógico do sinal. Caso seu valor lógico seja um, passa a ser zero, e caso seu valor lógico seja zero, passa a ser um durante um ciclo de relógio. O programa é executado diversas vezes, sendo que a apenas uma falha é inserida por execução da aplicação. Não há acúmulo de falhas. A cada nova execução, todo o processador é reinicializado.

O programa é executado uma vez sem falhas durante um tempo de execução determinado. O valor da parte da memória que contém a saída do programa é armazenado. Esse valor serve de base de comparação para verificar se as falhas inseridas nas execuções com falhas resultaram em erro ou não.

As falhas são injetadas utilizando o comando *force* na ferramenta ModelSim. O comando *force*, quando seguido pelo argumento *freeze*, força o valor de um sinal informado com um valor estabelecido. O tempo da falha pode ser definido com o argumento *cancel*. Caso o argumento não seja usado, a falha será permanente. A figura 4.10 mostra um exemplo da utilização do comando *force*. O comando *force* é seguido do argumento *freeze* e do argumento *cancel*. O tempo de duração da falha no exemplo da figura 4.10 é de 42ns, informado logo após o argumento *cancel*. Na sequência, vem o sinal que terá o valor alterado e o valor que será atribuído ao sinal.

```
force -freeze -cancel 42ns /sim_minimips/u_minimips/u4_ex/u1_alu/res(5) 1
```

Figura 4.10: Inserção de falhas em nível lógico através do comando *force*.

Para determinar qual valor deverá ser atribuído à falha, o valor corrente do sinal deve ser avaliado. Essa avaliação é realizada com o comando *examine*. Na figura 4.11 é apresentado como o comando *examine* é realizado.

```
examine /sim_minimips/u_minimips/u4_ex/u1_alu/res(5)
```

Figura 4.11: Avaliação do valor do sinal pelo comando *examine*.

O retorno desse comando é o valor corrente do sinal, ele deve ser armazenado em uma variável para ser comparado e, com base nele, determinar qual valor deve ser atribuído ao sinal com o comando *force*. A atribuição do valor do sinal a uma variável é realizada com o comando *set*, apresentado na figura abaixo.

```
set fValue [examine /sim_minimips/u_minimips/u4_ex/u1_alu/res(5)]
```

Figura 4.12: Atribuição do valor do sinal a uma variável.

As falhas podem ou não ocasionar erros na saída da aplicação. Muitas são mascaradas no tempo, ficam latentes e não influenciam o resultado. Somente as falhas que causaram erro na execução e resultados gerados na aplicação são consideradas na análise. Para determinar se uma falha ocasionou um erro, o valor de saída do programa, presente na memória, é comparado com o valor esperado, determinado pela execução sem falhas. E o tempo de execução é observado para detectar erros no fluxo de execução. Para visualizar o conteúdo da memória em determinado tempo, é utilizado o comando *mem display*. A figura 4.13 apresenta um exemplo de uso do comando *mem display*. O retorno do comando são os dados contidos no intervalo da memória informado.

```
mem display -format hex -startaddress 1310440 -endaddress  
1310463 /sim_minimips/u_ram/ram_xilinx/u0/sp_primitive/mem
```

Figura 4.13: Visualização de parte do conteúdo da memória.

Para salvar o conteúdo retornado pelo comando *mem display* é utilizado o comando *set* em conjunto com o comando *mem display*. O valor é atribuído para a variável informada. Na figura 4.14 pode ser visto um exemplo da utilização desse comando.

```
set memResult mem display -format hex -startaddress 1310440 -endaddress
1310463 /sim_minimips/u_ram/ram_xilinx/u0/sp_primitive/mem
```

Figura 4.14: Atribuição de parte do conteúdo da memória a uma variável.

O conteúdo da saída do programa na memória da execução com falhas é comparado com o conteúdo da saída do programa na memória do programa base, sem falhas. Se os valores forem todos iguais, não ocorreu erro. Agora, caso algum valor difira do valor base, a execução do programa foi errônea.

4.3 Metodologia de Classificação de Erros

As falhas podem ser mascaradas ao longo da execução do programa ou gerar erros no fluxo de execução e/ou nos resultados gerados. Esses erros devem ser classificados conforme sua influência na execução do programa. Os erros no controle são causados por falhas que acabam alterando o fluxo de execução do programa, fazendo com que o registrador contador de programa, do inglês *program counter* (PC), mude para um valor não esperado. Os erros nos dados são erros que afetam os dados sendo manipulados. A execução do programa segue normalmente, mas o resultado final não condiz com o esperado.

A classificação dos erros em erro de dado ou erro de controle é realizada comparando os valores do PC no tempo e verificando se eles condizem com o da execução sem falhas. Essa classificação, em erros no controle e erros nos dados, é realizada de forma automática. Para isso, foi desenvolvido no Injetor de Falhas funcionalidades que realizam essa classificação.

O programa, que passará pela campanha de injeções de falhas, é executado normalmente uma vez, sem falhas. Nesse processo, a cada ciclo de relógio do processador, o valor do PC é salvo. Cada valor do PC salvo tem como identificador o tempo de execução corrente do programa. Após a execução do programa, parte da memória que contém os dados de saída do programa é armazenada.

A injeção de falhas é iniciada. O programa é executado, dessa vez, com falhas. Em cada execução uma falha é injetada. A partir do ponto de onde a falha foi injetada, o valor do PC passa a ser salvo a cada ciclo de relógio. Esse processo de salvar o PC é realizado durante determinado número de ciclos. Da mesma forma que a execução sem falhas, o identificador de cada PC salvo é o valor corrente do tempo de execução do programa. Foi verificado em simulações que o valor do PC salvo durante 40 ciclos ou mais de *clock* após a ocorrência da falha garante uma classificação fidedigna dos erros. Por garantia, nas simulações deste trabalho, o valor adotado foi de 100 ciclos de relógio.

Após isso, o programa é executado até seu término. O conteúdo da parte da memória que contém a saída dos dados do programa é verificado. Se o valor é condizente com o da execução sem falhas, não ocorreu erro. Agora, caso o valor da parte da memória que contém a saída do programa for diferente do valor correto, gerado pela execução sem falhar, um erro ocorreu. Quando acontece um erro, ele deve ser classificado em erro no controle ou erro nos dados.

Os valores salvos do PC da execução com falhas são comparados com os valores da execução base, sem falhas. Se todos os valores forem iguais, então não houve desvio no fluxo de execução do programa, o que caracteriza como um erro nos dados. Se algum valor do PC salvo for diferente, então houve desvio no fluxo de execução do programa. Como o programa pode possuir detecção de erro, esse desvio pode ter sido causado pela

técnica de detecção em *software* aplicada. Dessa forma, um erro nos dados que tenha sido detectado terá diferentes valores para o PC. Portanto, a sub-rotina de tratamento de erro deve ser desconsiderada da comparação. Para isso, as posições inicial e final da sub-rotina de tratamento de erro devem ser informadas. Assim, desvios para a sub-rotina de tratamento de erro não serão consideradas na comparação. Logo, um erro será classificado como erro de controle se houver um desvio não esperado no fluxo de execução do programa e o valor do PC não estiver dentro da área da sub-rotina de tratamento de erro. Caso contrário, o erro será classificado como erro nos dados.

Na figura 4.15 é apresentado um exemplo de erro nos dados. Como o PC da execução com falha é igual no tempo ao PC da execução normal, não houve desvio no fluxo de execução do programa, o que caracteriza um erro nos dados. O erro não foi detectado, pois o fluxo de execução do programa não foi desviado para a sub-rotina de tratamento de erro.

Base	Com erro
57c	57c
580	580
584	584
588	588
744	744
748	748
74c	74c
750	750
754	754
758	758

Figura 4.15: PC da execução base, sem falhas (à esquerda) e PC da execução com erro, não detectado, nos dados (à direita).

A figura 4.16 apresenta um exemplo de quando um erro afetando os dados é detectado. O retângulo maior da figura indica onde o PC da execução com falhas difere do PC da execução base. As duas últimas posições da figura possuem valores diferentes. Contudo, os valores diferentes da execução com falhas estão dentro da área da sub-rotina de tratamento de erro, indicado pelo retângulo menor de cantos arredondados. Isso significa que o erro foi detectado pela técnica de detecção em *software* aplicada e que se trata de um erro nos dados, pois não houve desvio no fluxo de execução do programa, somente para a sub-rotina de tratamento de erro.

Base	Com erro
57c	57c
580	580
584	584
588	588
744	744
748	748
74c	74c
750	750
754	858
758	85c

Figura 4.16: PC da execução base, sem falhas (à esquerda) e PC da execução com erro, detectado, nos dados (à direita).

Um erro no controle ocorre quando os valores do PC da execução com falhas diferem dos valores do PC da execução base e não estejam dentro da sub-rotina de tratamento de erro. A figura 4.17 apresenta um exemplo de erro afetando o controle. A partir da quinta linha da figura, os valores do PC diferem, indicando um desvio não esperado no fluxo de execução do programa com falhas, e os valores da execução com falhas não estão dentro da sub-rotina de tratamento de erro em nenhum caso. Portanto, um erro no controle ocorreu e esse erro não foi detectado.

Base	Com erro
57c	57c
580	580
584	584
588	588
744	58c
748	590
74c	594
750	598
754	524
758	528

Figura 4.17: PC da execução base, sem falhas (à esquerda) e PC da execução com erro, não detectado, no controle (à direita).

Na figura 4.18 pode ser visto um exemplo de um erro detectado que afeta o fluxo de execução do programa. A partir da quinta linha da figura os valores do PC da execução com erro diferem da execução base, indicado pelo retângulo maior com cantos retos. Esse erro foi detectado. Nas duas últimas linhas, o fluxo de execução do programa

com erro entra na sub-rotina de tratamento de erro, indicada pelo retângulo menor de cantos arredondados. O erro é classificado como erro no controle, pois há casos de valores do PC divergindo dos valores do PC base, sendo que os valores do PC da execução com erros não estão dentro da área da sub-rotina de tratamento de erro.

Base	Com erro
57c	57c
580	580
584	584
588	588
744	58c
748	590
74c	594
750	598
754	858
758	85c

Figura 4.18: PC da execução base, sem falhas (à esquerda) e PC da execução com erro, detectado, no controle (à direita).

4.4 Validação

A CFT-tool foi validada de duas formas. A primeira consiste em verificar se as técnicas de detecção em *software* estão corretamente implementadas. As taxas de detecções obtidas pelos programas protegidos com as técnicas aplicadas pela CFT-tool são comparadas com os mesmos programas protegidos por as mesmas técnicas de detecção em *software* aplicadas por outras ferramentas presentes na literatura. A segunda forma de validação consiste em verificar a configurabilidade da CFT-tool, ou seja, verificar se a CFT-tool é capaz de aplicar as técnicas de detecção em *software* em programas compilados para diferentes arquiteturas.

4.4.1 Programas selecionados para validação

Três programas foram selecionados para serem usados nos testes de validação: uma multiplicação de matrizes, um algoritmo de ordenação e um algoritmo de encriptação.

4.4.1.1 Multiplicação de matrizes

A multiplicação de matrizes possui uma grande quantidade de processamento de dados com poucos laços, sendo ideal para verificar a cobertura das técnicas de detecção em *software* na detecção de erros afetando os dados. Na figura 4.19 é apresentado o código, em linguagem C, do programa de multiplicação de matrizes. Esse programa multiplica duas matrizes de tamanho 3x3, resultando em outra matriz do mesmo tamanho.


```

1  int main()
2  {
3      int a[3][3]={{11,12,13},{21,22,23},{31,32,33}};
4      int b[3][3]={{28,27,26},{18,17,16},{8,7,6}};
5      int c[3][3]={{0,0,0},{0,0,0},{0,0,0}};
6      int i,j,v;
7      for(i=0; i<3; i++)
8          for(j=0; j<3; j++)
9              for (v=0; v<3; v++)
10                 c[i][j]=c[i][j]+a[i][v]*b[v][j];
11 }

```

Figura 4.19: Código, em C, do programa de multiplicação de matrizes.

4.4.1.2 Bubble sort

Os algoritmos de ordenações não possuem tanto processamento de dados quanto a multiplicação de matrizes. Eles possuem um grande número de laços, desvios e registradores utilizados no controle do fluxo de execução do programa.

Como algoritmo de ordenação, foi escolhido o *bubble sort*, que é o extremo dos algoritmos de ordenação na questão de comparação entre os valores a serem ordenados. A figura 4.20 apresenta o código do algoritmo de ordenação *bubble sort* utilizado nas fases de validação da CFT-tool.

```

1  int main()
2  {
3      int v[10]={5,10,7,5,9,4,2,3,1,6};
4      int i,j,aux;
5      int k=9;
6
7      for(i = 0; i < 10; i++)
8      {
9          for(j = 0; j < k; j++)
10         {
11             if(v[j] > v[j+1])
12             {
13                 aux = v[j];
14                 v[j] = v[j+1];
15                 v[j+1]=aux;
16             }
17         }
18         k--;
19     }
20 }

```

Figura 4.20: Código, em C, do algoritmo de ordenação *bubble sort*.

4.4.1.3 Algoritmo de encriptação (TEA2)

O algoritmo de encriptação selecionado foi o *TETRA Encryption Algorithm* (TEA2). Assim como a multiplicação de matrizes, possui uma grande quantidade de processamento de dados com poucos laços. Contudo, os dados do algoritmo de encriptação são mais concentrados, ou seja, menos registradores são utilizados pelo

programa durante sua execução. Na figura 4.21 é apresentado o código do algoritmo de encriptação TEA2.

```

1  #include "lib/stdint.h"
2  #include <stdlib.h>
3
4  void encrypt (uint32_t* v, uint32_t* k) {
5      uint32_t v0=v[0], v1=v[1], sum=0, i;
6      uint32_t delta=0x9e3779b9;
7      uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];
8      for (i=0; i < 32; i++) {
9          sum += delta;
10         v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
11         v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
12     }
13     v[0]=v0; v[1]=v1;
14 }
15
16 int main() {
17     unsigned int k[4] = {789012345,1234567890,321098765,876543210};
18     unsigned int v[2] = {418895811,1085367610};
19     encrypt(v,k);
20     return 0;
21 }

```

Figura 4.21: Código, em C, do algoritmo encriptação.

4.4.2 Validação das aplicações das técnicas de detecção em software pela CFT-tool

A corretude da aplicação das técnicas de detecção em *software* pela CFT-tool e a eficiência da aplicação das técnicas em código *assembly* devem ser validadas. Para isso, é necessário verificar a eficácia da CFT-tool na aplicação das técnicas de detecção em *software* implementadas em relação às mesmas técnicas implementadas por outra ferramenta presente na literatura.

A ferramenta HPCT Suite (AZAMBUJA, 2010) foi utilizada como base de comparação para validação da aplicação das técnicas de detecção em *software* pela CFT-tool e também para verificar a eficiência da aplicação das técnicas em código *assembly*. A HPCT Suite protege programas compilados para o processador miniMIPS modificando o código de máquina do programa.

Os três programas selecionados foram protegidos com seis diferentes combinações das técnicas de detecção em *software*. Três delas com técnicas destinadas a proteção dos dados e as outras três com combinações de técnicas destinadas à proteção dos dados e à proteção do controle. Essas combinações foram escolhidas devido à baixa detecção de erros das técnicas de proteção do controle e do fato das injeções de falhas requererem bastante tempo de simulação. As versões protegidas com as técnicas de detecção em *software* destinada à proteção dos dados estão protegidas pelas técnicas Variáveis, uma com a técnica VAR1, outra com a técnica VAR2 e a terceira com a técnica VAR3. As três versões com proteção dos dados e do controle utilizam as técnicas BRA e SIG juntamente com uma técnica de proteção dos dados (VAR1, VAR2 e VAR3).

4.4.2.1 Tempo de execução e ocupação de Memória

Os tempos de execução e a ocupação de memória dos programas selecionados variam conforme a técnica aplicada. Os tempos de execução para a multiplicação de matrizes com as diferentes técnicas aplicadas por ambas as ferramentas, rodando sobre o processador miniMIPS, podem ser visto na figura 4.22. Os tempos para a multiplicação de matrizes protegido pela técnica VAR3 e pela combinação das técnicas SIG, BRA e VAR3 aplicados pela CFT-tool foi levemente menor do que as mesmas técnicas de detecção em *software* aplicadas pela HPCT Suite. Essas duas versões do programa apresentaram menor aumento no tempo de execução em relação ao programa desprotegido do que as outras técnicas. Para as outras técnicas, os tempos de execução dos programas protegidos pela CFT-tool foram um pouco maior que os tempos dos programas protegidos pela HPCT Suite. Isso se deve a características de como são inseridos os verificadores pela CFT-tool, em *assembly*, e pela HPCT Suite, no código de máquina.

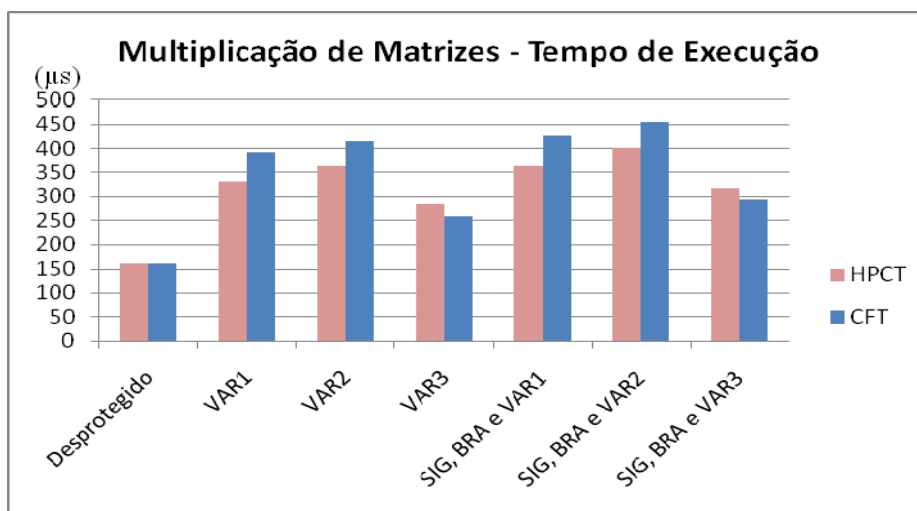


Figura 4.22: Tempos de execução para a multiplicação de matrizes com diferentes técnicas aplicadas.

A figura 4.23 apresenta a ocupação de memória (em bytes) da multiplicação de matrizes com as diferentes técnicas de detecção em *software* aplicadas pela CFT-tool e pela HPCT Suite.

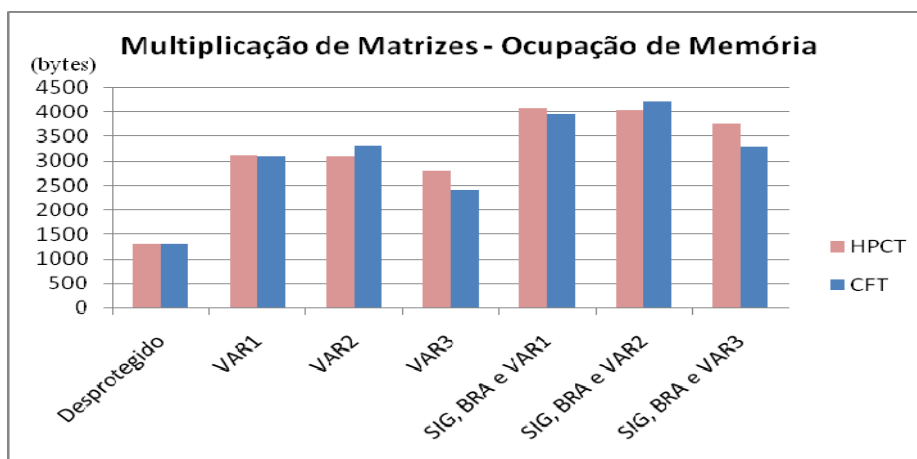


Figura 4.23: Ocupação em memória da multiplicação de matrizes com diferentes técnicas aplicadas.

A multiplicação de matrizes protegida pela CFT-tool com a técnica VAR3 e pela combinação de técnicas SIG, BRA e VAR3, ocupou menos memória que as mesmas versões protegidas pela HPCT Suite. Para as demais técnicas e combinação de técnicas, a ocupação de memória dos programas foi bastante semelhante, com variação máxima observada em torno de 18%.

Para o algoritmo de ordenação *bubble sort*, os tempos de execução das versões do programa com as diferentes combinações de técnicas de detecção em *software* selecionadas podem ser visto na figura 4.24. Os tempos de execução do *bubble sort* protegido com a técnica VAR3 foram bastante próximos para CFT-tool e HPCT-Suite, diferença de 4%. O mesmo pode ser dito do *bubble sort* protegido com as combinações de técnicas de detecção em *software* SIG, BRA e VAR1 e SIG, BRA e VAR3. Nos demais casos, devido a como os desvios são organizados pelo montador, os tempos de execução das versões geradas pela CFT-tool são um pouco maior que os tempos de execução das versões geradas pela HPCT Suite, onde variações de até 18% são observadas.

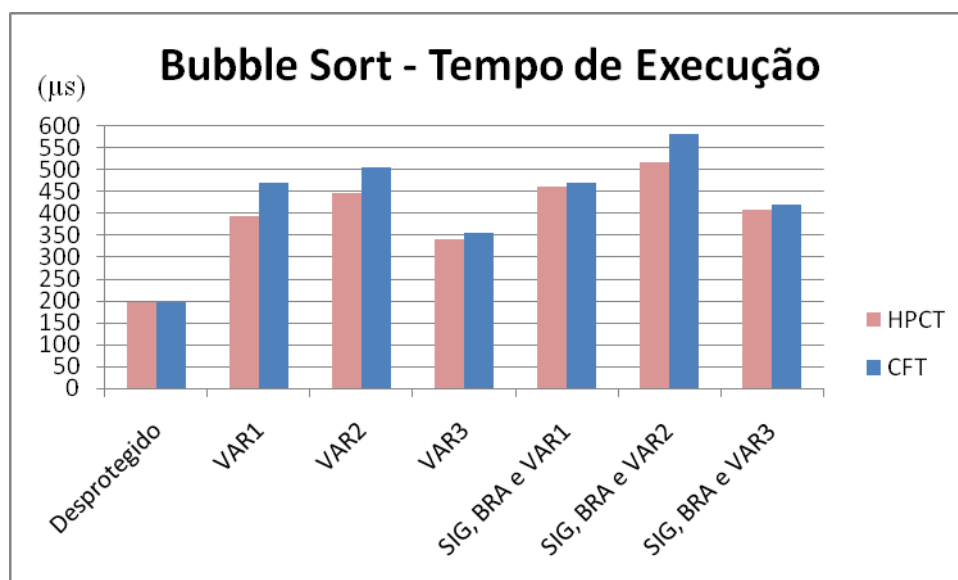


Figura 4.24: Tempos de execução para o *bubble sort* com diferentes técnicas aplicadas.

A ocupação em memória para as diferentes versões do *bubble sort* foi bastante semelhante para ambas as ferramentas na maioria dos casos, variações de até 14%. Para a versão do *bubble sort* com a técnica de detecção em *software* VAR2, a ocupação de memória do programa protegido pela CFT-tool foi um pouco maior que o programa protegido pela HPCT Suite, variações de até 8% foram observadas. E para as versões com a técnica VAR3 e com a combinação de técnicas SIG, BRA e VAR3, a ocupação de memória das versões protegidas pela CFT-tool foi menor que a ocupação de memória das versões protegidas pela HPCT Suite. Esses dados são apresentados na figura 4.25.

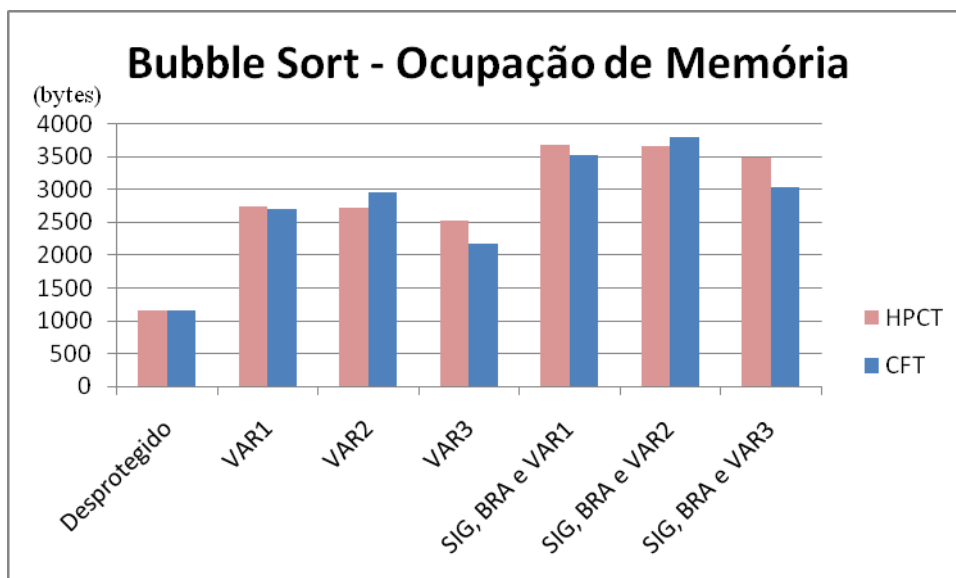


Figura 4.25: Ocupação em memória do *bubble sort* com diferentes técnicas aplicadas.

Os tempos de execução do algoritmo de encriptação (TEA2) são apresentados na figura 4.26. Em todos os casos, os tempos de execução para as versões protegidas pela CFT-tool foram maiores que a da HPCT Suite. As menores diferenças ocorreram para as versões com a técnica VAR3 e para a combinação de técnicas de detecção em *software* SIG, BRA e VAR3, em torno de 8%.

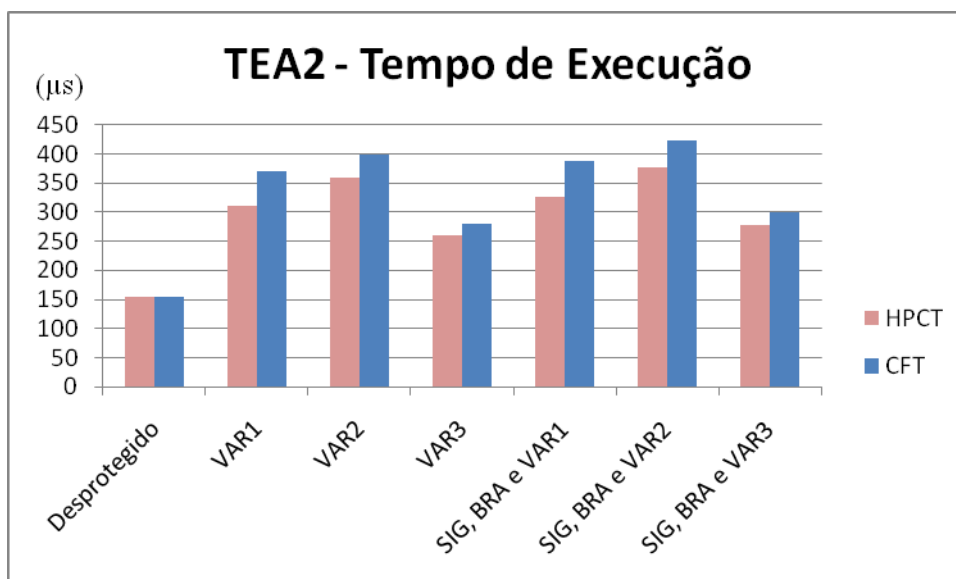


Figura 4.26: Tempos de execução para o algoritmo de encriptação (TEA2) com diferentes técnicas aplicadas.

A figura 4.27 apresenta a ocupação de memória do algoritmo de encriptação TEA2 com as diferentes versões do programa. Para a versão do algoritmo de encriptação com a técnica de detecção em *software* VAR2, a ocupação de memória do programa protegido pela CFT-tool foi um pouco maior que do programa protegido pela HPCT Suite. Para a versão com a técnica VAR3 e para a versão com a combinação de técnicas SIG, BRA e VAR3, a ocupação de memória dos programas gerados pela CFT-tool foi menor que dos programas gerados pela HPCT Suite. Nos demais casos, com outras

técnicas ou combinações de técnicas de detecção em *software*, a ocupação da memória foi semelhante, tendo diferenças máximas de 10%.

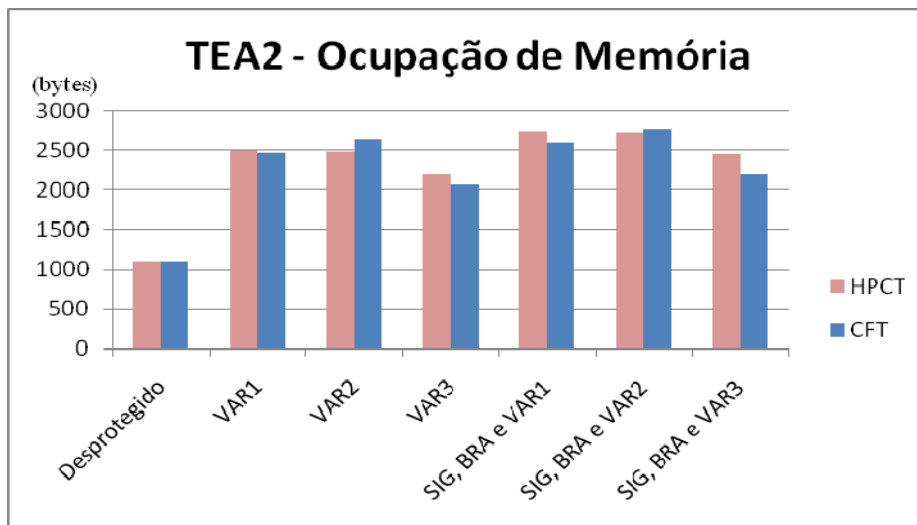


Figura 4.27: Ocupação em memória do algoritmo de encriptação com diferentes técnicas aplicadas.

Os tempos de execução para a maioria dos programas protegidos tanto pela CFT-tool quanto pela HPCT Suite foram semelhantes, como foi observado. Em alguns casos, devido a características do montador e a como as instruções de verificação são aplicadas, ou seja, como os desvios condicionais são inseridos no código, os tempos de execução podem ser maiores para as versões protegidas pela CFT-tool em relação à HPCT Suite. Contudo, nas versões com técnicas que contêm menos instruções de verificação, no caso, a versão com a técnica VAR3 e a com a combinação de técnicas SIG, BRA e VAR3, os tempos de execução dos programas protegidos pela CFT-tool tendem a serem menores que os mesmos programas protegidos pela HPCT Suite. Isso se deve à opção selecionada na configuração da CFT-tool de não duplicar o registrador \$0 que contém o valor da constante zero.

A ocupação de memória foi bastante semelhante para a maioria das técnicas e combinações de detecção em *software* para ambas as ferramentas. Uma menor ocupação de memória foi obtida pela CFT-tool para as versões dos programas protegidos com a técnica VAR3 e com a combinação de técnicas SIG, BRA e VAR3. Nos demais casos, a diferença foi pequena, menor que 15%.

4.4.2.2 Simulação de falhas

Uma campanha de injeção de falhas foi realizada para validar a CFT-tool. Falhas foram injetadas em nível lógico no processador miniMIPS, utilizando o Modelsim (MENTOR GRAPHICS). O local e o tempo onde cada falha é injetada são selecionados aleatoriamente. Somente uma falha foi injetada por cada execução do programa. Três programas foram protegidos pela CFT-tool: uma multiplicação de matrizes, um algoritmo de ordenação (*bubble sort*) e um algoritmo de encriptação (TEA2). Os programas foram protegidos com seis diferentes combinações de técnicas de detecção em *software*. Os três primeiros foram protegidos somente com técnicas de proteção dos dados, são elas: VAR1, VAR2 e VAR3. As outras três versões combina cada uma dessas técnicas de proteção dos dados com duas técnicas de proteção do controle, BRA e SIG.

O tempo necessário pela CFT-tool para proteger as aplicações é desprezível. O tempo depende do tamanho do código do programa e das técnicas selecionadas. CFT-tool leva menos de 5s para proteger os códigos utilizados na validação.

Com o objetivo de verificar a eficiência das técnicas de detecção em *software*, somente as falhas que causaram erros foram levadas em consideração. As falhas mascaradas pela lógica do processador foram ignoradas.

Depois de aplicar as técnicas de detecção em *software* sobre os programas, uma campanha de injeção de falhas foi realizada, onde 10 mil falhas foram injetadas em cada programa protegido. A tabela 4.2 mostra as taxas de detecção para a multiplicação de matrizes protegida com diferentes combinações das técnicas de detecção em *software* para o processador miniMIPS. Os programas foram protegidos pela CFT-tool.

Tabela 4.2: Multiplicação de Matrizes (MIPS) - CFT-tool

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	727	7.3	1961	97.9	2688	73.4
VAR2	757	7.9	1797	98.5	2554	71.7
VAR3	754	5.4	1749	96.4	2503	69.0
SIG, BRA e VAR1	775	7.5	1879	98.6	2654	72.0
SIG, BRA e VAR2	765	11.0	1652	97.8	2417	70.3
SIG, BRA e VAR3	756	8.9	1826	97.8	2582	71.8

Como base de comparação para validar a correta aplicação das técnicas de detecção em *software* pela CFT-tool, os mesmos programas foram protegidos pela HPCT Suite com as mesmas técnicas. As taxas de detecção obtidas pelas técnicas de detecção em *software* aplicadas pela HPCT Suite sobre a multiplicação de matrizes são apresentadas na tabela 4.3.

Tabela 4.3: Multiplicação de Matrizes (MIPS) – HPCT Suite

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	723	7.9	1930	99.3	2653	74.4
VAR2	768	11.6	1760	99.3	2528	72.6
VAR3	773	10.0	1596	95.6	2369	67.6
SIG, BRA e VAR1	888	5.0	1882	96.2	2770	66.9
SIG, BRA e VAR2	882	8.6	1672	96.0	2554	65.8
SIG, BRA e VAR3	748	11.9	1745	96.8	2493	71.3

As técnicas de variáveis aumentam consideravelmente o tempo de execução dos programas, mas conseguem uma boa cobertura de erros. A técnica VAR3 apresenta um melhor custo-benefício em termos de detecção de erros e aumento no tempo de execução. As técnicas combinadas apresentam um pequeno aumento no tempo de execução quando comparadas com os programas protegidos somente por técnicas de variáveis, obtendo um ganho máximo em torno de 2% em relação a somente as técnicas de variáveis. Contudo, o aumento na taxa de detecção de erros é insignificante. Somente para a combinação das técnicas SIG e BRA com a técnica VAR3 que houve uma melhora, mesmo assim pouco significativa. As taxas de detecção das técnicas aplicadas sobre o código *assembly*, pela CFT-tool, e sobre o código de máquina, pela HPCT Suite, são semelhantes.

A detecção de erros que afetam os dados foi consideravelmente alta, sendo que em todos os casos a taxa de detecção de erros nos dados ultrapassou 95%. Contudo, o mesmo não aconteceu com erros afetando o controle, onde a maioria das técnicas de detecção em *software* não ultrapassou os 10% de detecção. Na média, a taxa de detecção da multiplicação de matrizes protegida com diferentes técnicas de detecção em *software* ficou em 70%.

A tabela 4.4 apresenta o resultado da injeção de falhas para o *bubble sort* rodando no processador miniMIPS. Os programas foram protegidos com as diferentes combinações de técnicas de detecção em *software* pela CFT-tool. A taxa de detecção para as várias versões do *bubble sort* protegido pela CFT-tool ficou na faixa de 70%, sendo as falhas que afetaram os dados foram detectadas em quase sua totalidade, por volta de 98%, e as falhas que afetaram o controle foram pouco detectadas, em geral, menos de 10%.

Tabela 4.4: Bubble Sort (MIPS) - CFT-tool

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	694	7.9	1589	98.2	2283	70.8
VAR2	725	8.3	1388	98.1	2113	67.3
VAR3	703	7.7	1383	98.0	2086	67.5
SIG, BRA e VAR1	762	8.8	1614	97.7	2376	69.2
SIG, BRA e VAR2	720	11.1	1423	98.6	2143	69.2
SIG, BRA e VAR3	674	9.2	1328	97.6	2034	68.3

Os resultados para o mesmo programa com as técnicas de detecção em *software* aplicadas sobre o código de máquina pela HPCT Suite são mostrados na tabela 4.5. O *bubble sort* protegido por diferentes técnicas de detecção em *software* aplicadas pela HPCT Suite apresenta taxas de detecção de erros semelhantes às taxas de detecção obtidas pelo *bubble sort* protegido pelas mesmas técnicas aplicadas pela CFT-tool.

Tabela 4.5: Bubble Sort (MIPS) - HPCT Suite

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	719	6.5	1604	99.6	2323	70.8
VAR2	746	10.2	1378	99.1	2124	67.9
VAR3	742	7.0	1374	98.7	2116	66.5
SIG, BRA e VAR1	980	4.4	1458	99.7	2438	61.4
SIG, BRA e VAR2	906	6.6	1326	98.7	2232	61.3
SIG, BRA e VAR3	709	11.1	1404	98.9	2113	69.5

As taxas de detecção para o *bubble sort* protegido pela CFT-tool e pela HPCT Suite são semelhantes. Em comparação com a multiplicação de matrizes, as taxas de detecção de erros para os programas com as mesmas técnicas aplicadas é um pouco menor porque há, relativamente, mais falhas afetando o fluxo de execução do programa, ou seja, falhas que afetam o controle, as quais têm baixas taxas de detecção de erros. As taxas de detecção de erros afetando os dados são bastante eficientes, detectando mais de 98% dos erros desse tipo.

A tabela 4.6 mostra as taxas de detecção de erros para o algoritmo de encriptação (TEA2), protegido com as diferentes combinações de técnicas de detecção em *software*, rodando sobre o processador LEON3. Os programas foram protegidos por técnicas de detecção em *software* aplicadas sobre o código *assembly* do programa pela CFT-tool.

Tabela 4.6: Algoritmo de Encriptação TEA2 (MIPS) - CFT-tool

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	782	8.2	1749	96.3	2531	69.1
VAR2	772	8.7	1573	96.4	2345	67.5
VAR3	772	10.8	1657	94.3	2429	67.7
SIG, BRA e VAR1	817	9.2	1768	97.5	2585	69.6
SIG, BRA e VAR2	765	12.0	1709	96.0	2474	70.0
SIG, BRA e VAR3	717	10.0	1770	96.0	2487	71.2

Na tabela 4.7 são apresentadas as taxas de detecção de erros para o algoritmo de encriptação (TEA2), protegido com as diferentes combinações de técnicas de detecção em *software*, rodando sobre o processador LEON3. Os programas foram protegidos por técnicas de detecção em *software* aplicadas sobre o código de máquina do programa pela ferramenta HPCT Suite.

Tabela 4.7: Algoritmo de Encriptação TEA2 (MIPS) – HPCT Suite

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	748	8.3	1746	98.7	2494	71.6
VAR2	776	11.7	1487	98.9	2263	69.0
VAR3	802	7.0	1484	98.1	2286	66.1
SIG, BRA e VAR1	811	6.0	1745	95.9	2556	67.4
SIG, BRA e VAR2	891	10.7	1570	95.2	2461	64.6
SIG, BRA e VAR3	775	8.9	1607	94.8	2382	66.8

Semelhantemente aos programas anteriores, o algoritmo de encriptação TEA2 apresentou taxas de detecção, para ambas as ferramentas, na faixa de 70%. As taxas de detecção dos erros afetando o controle ficaram, em geral, abaixo dos 10%. Enquanto as taxas de detecção dos erros afetando os dados ficaram acima de 94%. Todos os programas protegidos apresentaram taxas de detecção semelhantes para as mesmas técnicas aplicadas sobre o código *assembly*, pela CFT-tool, e sobre o código de máquina, pela HPCT Suite.

A diferença dos tempos de execução entre as versões protegidas em nível *assembly* e código de máquina ocorre devido a como os desvios são tratados. Para o processador miniMIPS, do código *assembly* para o código de máquina, o montador troca as posições dos desvios condicionais e saltos incondicionais, movendo uma posição para cima, porque o processador sempre executa a instrução seguinte aos desvios, característica conhecida como *branch delay slot*. A HPCT Suite não respeita completamente essa condição na aplicação das técnicas de detecção em *software*. A figura 4.28 mostra um exemplo que explica essa diferença. A corretude para o código de máquina é mantida quando o programa executa sem erros, mas quando um erro é detectado, outro erro pode ser criado por uma instrução que não deveria ser executada. Isso pode interferir no tratamento do erro. Por outro lado, a CFT-tool permite configurar quais registradores devem ser duplicados. Como o registrador \$0 possui valor fixo em zero, foi escolhido, por opção, não duplicar esse registrador. Isso diminui o aumento no tempo de execução e na taxa de ocupação da memória dos programas, pois não há necessidade de realizar comparação do registrador \$0, pois ele não possui cópia.

01b8:	17ce0222 bne \$30,\$14,0xa44	01c4:	17cd0235 bne \$30,\$13,0xa9c
01bc:	14010221 bne \$0,\$1,0xa44	01c8:	00000000 nop
01c0:	afc00060 sw \$0,96(\$30)	01cc:	afc00060 sw \$0,96(\$30)
01c4:	adc10260 sw \$1,608(\$14)	01d0:	17cd0230 bne \$30,\$13,0xa9c
01c8:	17ce021e bne \$30,\$14,0xa44	01d4:	adba0100 sw \$0,256(\$13)
01cc:	1401021d bne \$0,\$1,0xa44	01d8:	afc00064 sw \$0,100(\$30)
01d0:	afc00064 sw \$0,100(\$30)	01dc:	17cd022b bne \$30,\$13,0xa9c
01d4:	adc10264 sw \$1,612(\$14)	01e0:	adba0104 sw \$0,260(\$13)
	HPCT Suite		CFT-tool

Figura 4.28: Proteção em código de máquina, à esquerda, e proteção em código *assembly*, à direita.

4.4.3 Validação da configurabilidade da ferramenta

A CFT-tool também precisa ser validada na questão da configurabilidade. Isso é feito verificando se as técnicas de detecção em *software* podem ser aplicadas sobre o código *assembly* de programas compilados para processadores com arquiteturas diversas, mantendo o programa funcional. Na validação da corretude da aplicação das técnicas pela CFT-tool e da eficiência das técnicas de detecção em *software* foi utilizado o processador miniMIPS, o qual possui uma ferramenta na literatura que pode ser utilizada como base de comparação. Nessa fase da validação, deve ser utilizado um processador de uma arquitetura diferente. O processador LEON3 foi selecionado. LEON3 é um processador com uma arquitetura SPARC V8, a qual implementa registradores organizados em janelas (janela de registradores), sendo os registradores divididos em local, global, de entrada e de saída.

As técnicas de detecção em *software* foram aplicadas para os três programas selecionados, multiplicação de matrizes, *bubble sort*, algoritmo de encriptação, compilados para a arquitetura SPARC V8. Da mesma forma que para o processador miniMIPS, para o processador LEON3, os três programas selecionados foram protegidos com seis diferentes combinações das técnicas de detecção em *software*. Três delas com técnicas destinadas a proteção dos dados e as outras três com combinações de técnicas destinadas à proteção dos dados e à proteção do controle. As versões protegidas com as técnicas de detecção em *software* destinada à proteção dos dados estão protegidas pelas técnicas Variáveis, uma com a técnica VAR1, outra com a técnica VAR2 e a terceira com a técnica VAR3. As três versões com proteção dos dados e do controle utilizam as técnicas BRA e SIG juntamente com uma técnica de proteção dos dados (VAR1, VAR2 e VAR3).

4.4.3.1 Tempo de execução e ocupação de Memória

Os tempos de execução e a ocupação de memória dos programas selecionados variam conforme a técnica aplicada. Os tempos de execução para a multiplicação de matrizes com as diferentes técnicas aplicadas rodando no processador LEON3 podem ser vistos na figura 4.29.

O aumento no tempo de execução da multiplicação de matrizes das versões protegidas com as técnicas VAR1 ou VAR2 é bastante significativo, sendo o tempo de execução dessas versões 3.30 e 2.98 vezes o tempo de execução original. Um melhor desempenho foi alcançado com o uso da técnica VAR3, onde o tempo de execução da multiplicação de matrizes é 2.18 vezes o tempo de execução da versão desprotegida.

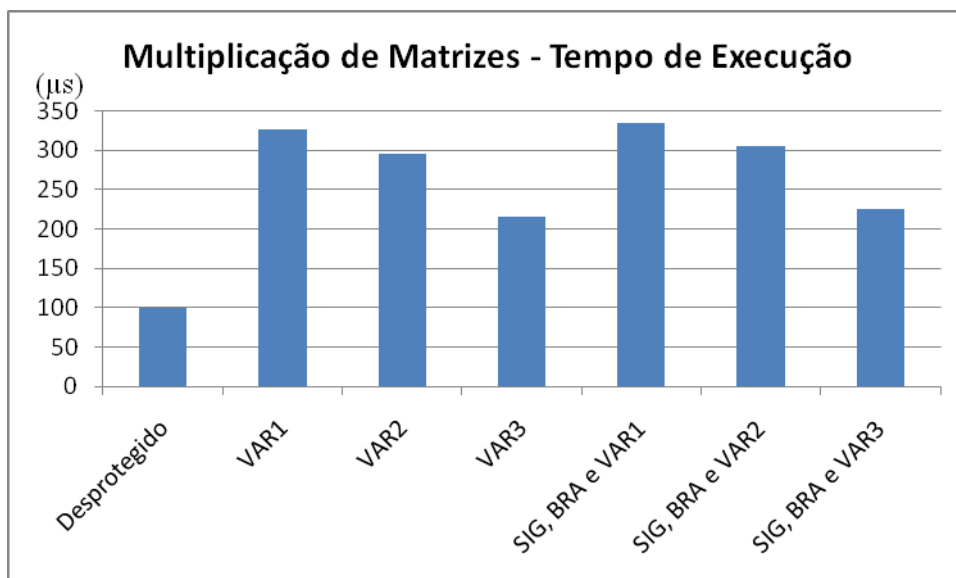


Figura 4.29: Tempos de execução para a multiplicação de matrizes rodando sobre o processador LEON3.

A figura 4.30 apresenta a ocupação de memória (em *bytes*) da multiplicação de matrizes com as diferentes técnicas de detecção em *software* aplicadas rodando sobre o processador LEON3. O aumento na ocupação da memória é bastante sensível nos programas protegidos com técnicas que possuem muitas instruções de verificação, ou seja, os programas que foram protegidos pela técnica VAR1 ou VAR2. Isso ocorre porque para realizar uma verificação é necessária a inserção de duas instruções, uma para comparar os registradores e outra para realizar o desvio. A versão da multiplicação de matrizes protegida com a técnica VAR3 tem um aumento da ocupação de memória bem menor que as outras versões, cerca de 1.95x o tamanho do código original, pois a técnica VAR3 utiliza bem menos instruções de verificação.

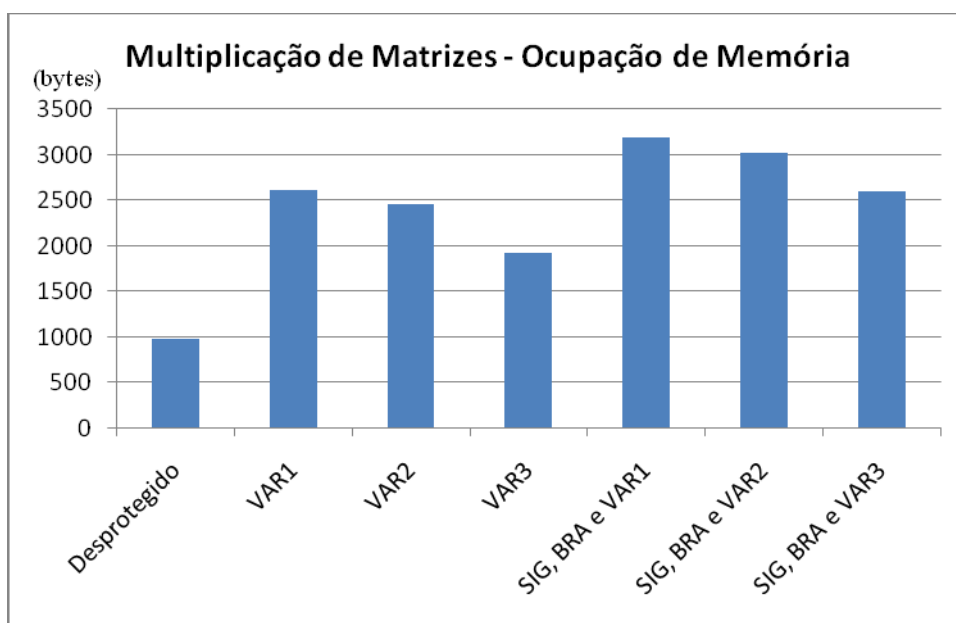


Figura 4.30: Ocupação em memória da multiplicação de matrizes rodando sobre o processador LEON3.

Para o algoritmo de ordenação *bubble sort*, os tempos de execução das versões do programa com as diferentes combinações de técnicas de detecção em *software* selecionadas rodando sobre o processador LEON3 podem ser visto na figura 4.31. O aumento no tempo de execução para as versões protegidas do *bubble sort* é bastante significativo. O aumento relativo é maior para o *bubble sort* protegido rodando no processador LEON3 em relação ao *bubble sort* protegido rodando no processador miniMIPS devido a como são inseridos os verificadores. Enquanto no miniMIPS é necessária a inserção de somente uma instrução para realizar a verificação, no LEON3 são necessárias duas instruções.

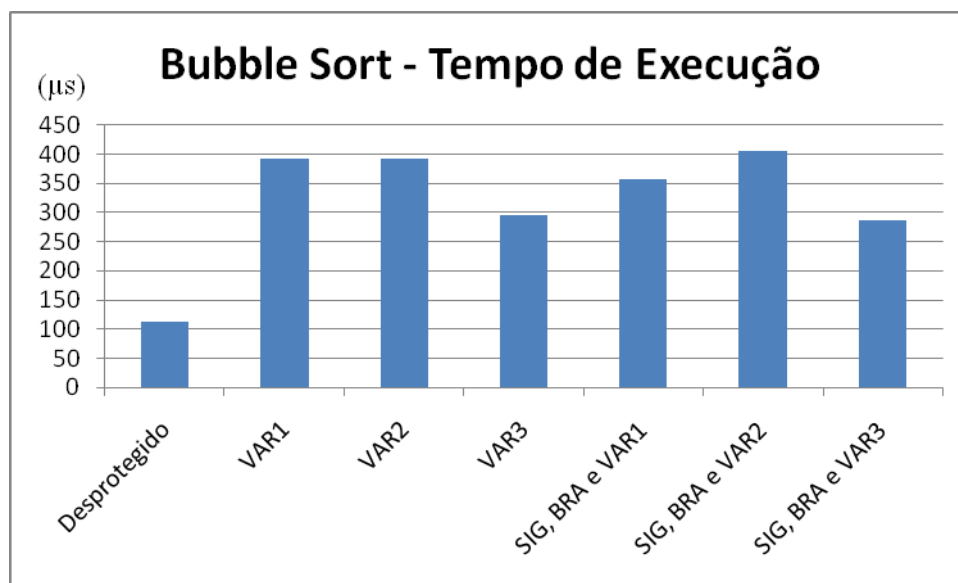


Figura 4.31: Tempos de execução para o *bubble sort* (LEON3).

A ocupação em memória para as diferentes versões do *bubble sort* pode ser visto na figura 4.32.

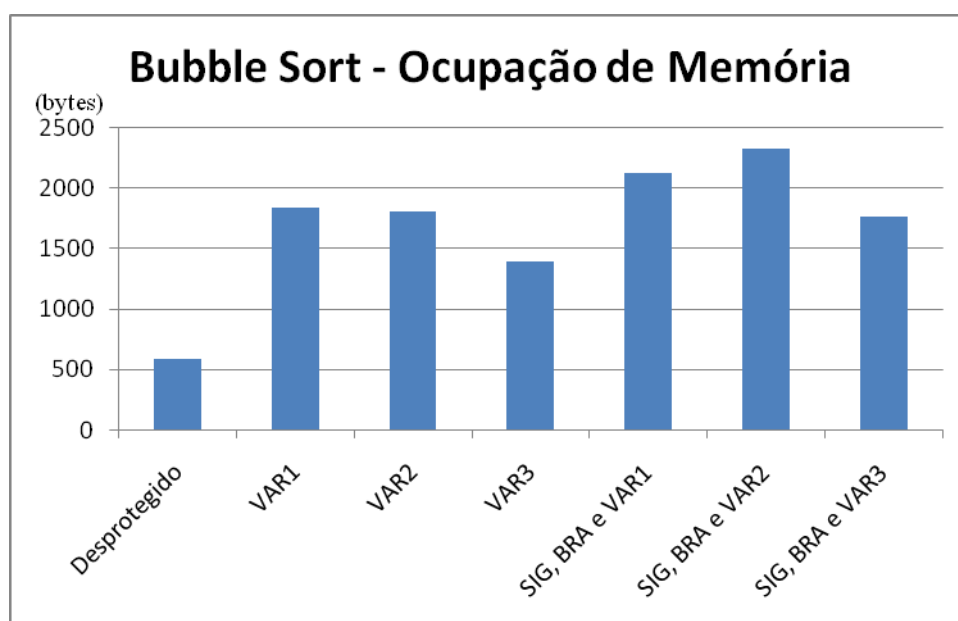


Figura 4.32: Ocupação em memória do *bubble sort* com diferentes técnicas aplicadas, rodando sobre o processador LEON3.

Da mesma forma que a multiplicação de matrizes, para as versões com as técnicas VAR1 ou VAR2, a ocupação de memória sobre um considerável aumento. A versão do *bubble sort* protegida com a técnica VAR3 não sofre um aumento tão grande quanto as demais técnicas, tendo o código 2.37 vezes o tamanho do código original. Contudo, o aumento relativo à versão desprotegida do *bubble sort* para o *bubble sort* protegido com a técnica VAR3 é maior do que o aumento relativo da multiplicação de matrizes protegida com a técnica VAR3 e comparada com sua versão desprotegida.

Os tempos de execução do algoritmo de encriptação (TEA2), rodando sobre o processador LEON3, são apresentados na figura 4.33. Assim como o *bubble sort*, o aumento relativo dos tempos de execução do algoritmo de encriptação TEA2 é maior para o LEON3 em relação ao miniMIPS, devido ao número de instruções necessárias para realizar as verificações. Esse aumento é mais sensível para as versões com as técnicas VAR1 e VAR2.

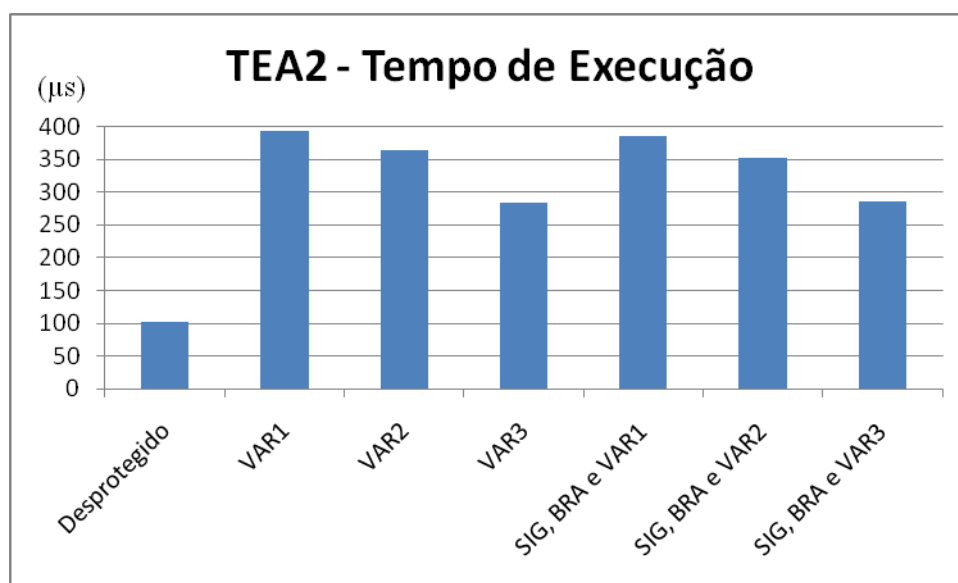


Figura 4.33: Tempos de execução para o algoritmo de encriptação (TEA2) com diferentes técnicas aplicadas, rodando sobre o processador LEON3.

A figura 4.34 apresenta a ocupação de memória do algoritmo de encriptação TEA2 com as diferentes versões do programa. O aumento da ocupação de memória para o algoritmo de encriptação TEA2 protegido com as diferentes técnicas ficou na faixa de 4 vezes o tamanho do programa original.

O aumento no tempo de execução foi relativamente maior para os programas protegidos pelas técnicas rodando no processador LEON3 que os mesmos programas com as mesmas técnicas rodando no processador miniMIPS. Esse aumento é mais significativo para os programas protegidos com as técnicas VAR1 e VAR2, as quais inserem mais verificações no código do programa. Da mesma forma, o aumento na ocupação de memória foi mais sensível para os programas compilados para o processador LEON3 justamente devido à forma como as verificações são realizadas. Primeiramente, os valores dos registradores são comparados e na sequência o desvio é, ou não, tomado, dependendo do resultado da comparação. Além disso, uma instrução de não operação (*nop*) deve ser também inserida. No miniMIPS, esse procedimento de verificação é realizado por uma instrução.

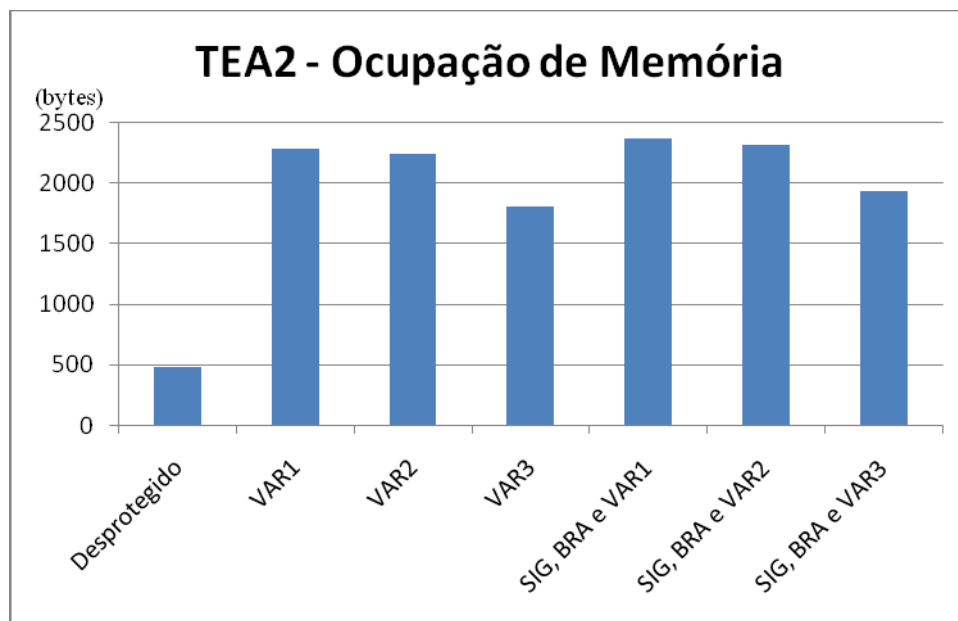


Figura 4.34: Ocupação em memória do algoritmo de encriptação com diferentes técnicas aplicadas, rodando sobre o processador LEON3.

4.4.3.2 Simulação de falhas

Depois de aplicar as técnicas de detecção em *software* sobre os programas, uma campanha de injeção de falhas foi realizada, onde 10 mil falhas foram injetadas em cada programa protegido. A tabela 4.8 mostra as taxas de detecção para a multiplicação de matrizes protegida com diferentes combinações das técnicas de detecção em *software* para o processador LEON3. Os programas foram protegidos pela CFT-tool. Não há registradores globais disponíveis para duplicar todos os registradores globais utilizados. Por isso, um método para selecionar os registradores que serão duplicados deve ser utilizado. A seleção de quais registradores serão duplicados é realizada levando em conta o número de vezes que cada registrador aparece no código do programa.

Tabela 4.8: Multiplicação de Matrizes (LEON3) – CFT-tool

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	336	7.1	648	82.1	984	56.5
VAR2	324	11.7	684	83.6	1008	60.5
VAR3	390	5.6	572	78.3	962	48.9
SIG, BRA e VAR1	294	6.8	550	80.7	844	55.0
SIG, BRA e VAR2	366	10.4	586	81.6	952	54.2
SIG, BRA e VAR3	472	4.7	474	61.2	946	33.0

A multiplicação protegida com a técnica VAR2 atingiu uma taxa de detecção de erros de 60.5%, um pouco abaixo dos resultados obtidos para o processador miniMIPS.

Isso ocorre pois não há registradores disponíveis suficientes para duplicar todos os registradores utilizados. Como a técnica SIG utiliza um registrador, há menos registradores para ser utilizado pela técnica de Variáveis quando combinadas, fazendo com que a taxa de detecção de erros diminua nesses casos. As taxas de detecção de erros com efeito nos dados ficou na faixa de 80%.

A tabela 4.9 apresenta o resultado da injeção de falhas para o *bubble sort* rodando no processador LEON3. Os programas foram protegidos com as diferentes combinações de técnicas de detecção em *software* pela CFT-tool. Não há registradores globais disponíveis para duplicar os sete registradores globais utilizados. Há três registradores globais disponíveis para as técnicas Variáveis sendo aplicadas isoladamente e somente dois registradores disponíveis para as técnicas de Variáveis quando estas estão combinadas com as técnicas SIG e BRA, pois a técnica de detecção em *software* SIG utiliza um registrador global.

Tabela 4.9: Bubble Sort (LEON3) – CFT-tool

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	276	9.1	552	88.2	828	61.8
VAR2	281	10.0	669	90.6	950	66.7
VAR3	328	4.3	570	88.8	898	57.9
SIG, BRA e VAR1	304	7.2	524	85.5	828	56.8
SIG, BRA e VAR2	276	15.2	698	94.0	974	71.7
SIG, BRA e VAR3	362	6.6	598	90.3	960	58.8

A técnica de detecção em *software* VAR2, sendo utilizada isoladamente ou em combinação com as técnicas SIG e BRA atingiu uma taxa de detecção equivalente ao obtido para o processador miniMIPS, mesmo não tendo registradores disponíveis suficientes para duplicar todos os registradores utilizados. A detecção de erros afetando os dados foi superior a 85% em todos os casos, chegando a 94% para a combinação das técnicas SIG, BRA e VAR2.

A tabela 4.10 mostra a taxa de detecção de erros para o algoritmo de encriptação (TEA2), protegido com as diferentes combinações de técnicas de detecção em *software*, rodando sobre o processador LEON3. Também não há registradores suficientes para duplicar todos os 6 registradores utilizados. Os registradores foram selecionados pelo número de vezes em que aparecem no código.

As taxas de detecção de erros para o algoritmo de encriptação TEA2 ficaram na faixa de 60%, com destaque para a versão protegida com a técnica VAR2, a qual atingiu uma taxa de detecção de erros de 67.2%, próxima à taxa de detecção obtida o processador miniMIPS. A detecção de erro com efeito nos dados ficou acima dos 80%, chegando a 88.9% para a versão protegida com a técnica VAR2.

Tabela 4.10: Algoritmo de Encriptação TEA2 (LEON3) – CFT-tool

Técnica	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção Total (%)
VAR1	296	11.5	736	86.4	1032	64.9
VAR2	296	8.8	794	88.9	1090	67.2
VAR3	342	7.0	970	83.7	1312	63.7
SIG, BRA e VAR1	312	9.6	798	83.5	1110	62.7
SIG, BRA e VAR2	336	7.1	840	80.7	1176	59.7
SIG, BRA e VAR3	302	8.6	914	80.1	1216	62.3

Não é possível atingir 100% de detecção de erros somente com as técnicas de detecção em *software*. Falhas que afetem o controle, desviando o fluxo de execução do programa para dentro do mesmo bloco básico em execução não são detectadas. Contudo, com a utilização de um *watchdog* que complemente as técnicas em *software*, é possível atingir taxas de detecção de erros muito próximas a 100% (AZAMBUJA, 2011).

5 AVALIAÇÃO DO USO SELETIVO DOS REGISTRADORES

As técnicas de detecção em *software* transformam o código do programa a ser protegido, fazendo ele tolerante a falhas. Contudo, essas técnicas aumentam o tempo de execução e a taxa de ocupação da memória.

As técnicas de detecção em *software*, que visam proteger os dados, duplicam os registradores, criando cópias deles. No entanto, em muitos casos, não há registradores disponíveis suficientes para duplicar todos os registradores usados pelo programa. Nesses casos, alguns registradores devem ser selecionados. A seleção dos registradores que serão duplicados pode mudar significativamente a taxa de detecção de erros do programa protegido. O tempo de execução e a ocupação em memória do programa também são diretamente afetados por essa decisão. Portanto, uma sábia seleção dos registradores deve ser realizada para aumentar a taxa de detecção de erros e/ou diminuir o aumento no tempo de execução e na ocupação da memória.

Neste capítulo é realizada uma análise do impacto da seleção seletiva dos registradores na taxa de detecção de erros. Isto é, como o método para selecionar os registradores pode influenciar a taxa de detecção de erros. Além disso, o método selecionado também influencia no tempo de execução e na ocupação de memória do programa protegido. O processador utilizado para a realização do estudo foi o miniMIPS. Diversos métodos para seleção dos registradores a serem duplicados, com diferentes quantidades de registradores disponíveis, são apresentados.

5.1 Métodos

Diversos métodos propostos para seleção de registradores são avaliados de diferentes maneiras: taxa de detecção de erros, tempo de execução e ocupação de memória. VAR3 foi a técnica de detecção em *software* selecionada para proteção dos programas. O objetivo da técnica VAR3 é proteger os dados, através da duplicação dos registradores, atribuindo cópias a cada um e inserindo instruções para verificar o valor do registrador duplicado com o valor contido no registrador cópia. Há uma quantidade de registradores disponíveis que podem ser utilizados para duplicar os registradores em uso. Essa quantidade pode ser menor que o número de registradores em uso. Os registradores a serem duplicados pela técnica VAR3 são informados pelo método selecionado. Cada método tem uma diferente maneira para selecionar os registradores. Sete métodos foram desenvolvidos, sendo que quatro deles analisam o código *assembly* do programa e três são baseados em dados da execução do programa. A seguir, os métodos propostos são apresentados.

5.1.1 Método Static First (SF)

O método *Static First* consiste em um método que lê o código fonte do programa e seleciona os registradores na ordem em que eles aparecem no código. Esse método é utilizado como uma base para comparação com outros métodos, porque ele não tem critério para selecionar os registradores.

5.1.2 Método Static Target (ST)

A seleção dos registradores pelo método *Static Target* é baseada no número de instruções em que um valor é atribuído ao registrador, isto é, os registradores são selecionados pelo número de vezes em que eles são usados como registradores destino no código.

5.1.3 Método Static Source (SS)

O método *Static Source* seleciona os registradores levando em conta o número de vezes em que eles são lidos no código. Isto é, o registrador mais lido tem a maior prioridade, o segundo registrador mais lido tem a segunda maior prioridade e assim por diante.

5.1.4 Método Static Source and Target (SST)

O método *Static Source and Target* consiste de uma junção dos métodos *Static Source* e *Static Target*. Ele considera quantas vezes o registrador aparece no código, quer seja como registrador fonte ou destino.

5.1.5 Método Dynamic Target (DT)

Assim como o método *Static Target*, este método seleciona os registradores baseado na quantidade de vezes que eles são usados como registradores destino. Contudo, neste método, a contagem do número de vezes em que cada registrador é utilizado como registrador destino é realizada dinamicamente. O programa é executado e o número de vezes em que o registrador for utilizado na execução do programa é considerado.

5.1.6 Método Dynamic Source (DS)

O método *Dynamic Source* seleciona os registradores baseado no número de vezes em que cada registrador é lido durante a execução do programa. O programa é executado e o uso de cada registrador é contabilizado. O registrador mais usado como registrador fonte tem a maior prioridade para ser duplicado.

5.1.7 Método Dynamic Source and Target (DST)

O método *Dynamic Source and Target* também seleciona os registradores dinamicamente. Ele leva em conta o número de vezes em que cada registrador é utilizado durante a execução do programa, quer seja como registrador fonte ou registrador destino. O registrador mais utilizado tem a maior prioridade para ser duplicado.

5.2 Uso dos Registradores

Para usar os métodos dinâmicos, é necessário determinar o número de vezes que cada registrador é usado pelo programa. Desde modo, a prioridade dos registradores pode ser informada. A versão desprotegida dos programas foi utilizada. Um contador foi inserido para cada registrador usado, de duas maneiras diferentes. Na primeira, o contador incrementa quando o valor é atribuído ao registrador. Na segunda, o contador incrementa quando o registrador é usado como registrador fonte, isto é, quando ele é lido por alguma instrução. Três programas foram utilizados, o algoritmo de ordenação *bubble sort*, uma multiplicação de matrizes e um algoritmo de encriptação, o TETRA Encryption Algorithm (TEA2). Com relação aos métodos estáticos, a contagem é realizada em cima do código *assembly* do programa de forma automática pela CFT-tool.

O gráfico da figura 5.1 apresenta os usos dos registradores para o *bubble sort*. Esses valores serão utilizados pelos métodos dinâmicos para configurar a CFT-tool. Os contadores dos registradores usados como registradores destino apresentam o número de vezes em que cada registrador teve um valor atribuído a ele. Os contadores fonte contabilizam o número de vezes em que um registrador foi lido por uma instrução. Os contadores combinado representam o total de vezes em que o registrador foi utilizado durante a execução do programa, quer sendo lido ou tendo um valor atribuído. O gráfico está na escala logarítmica.

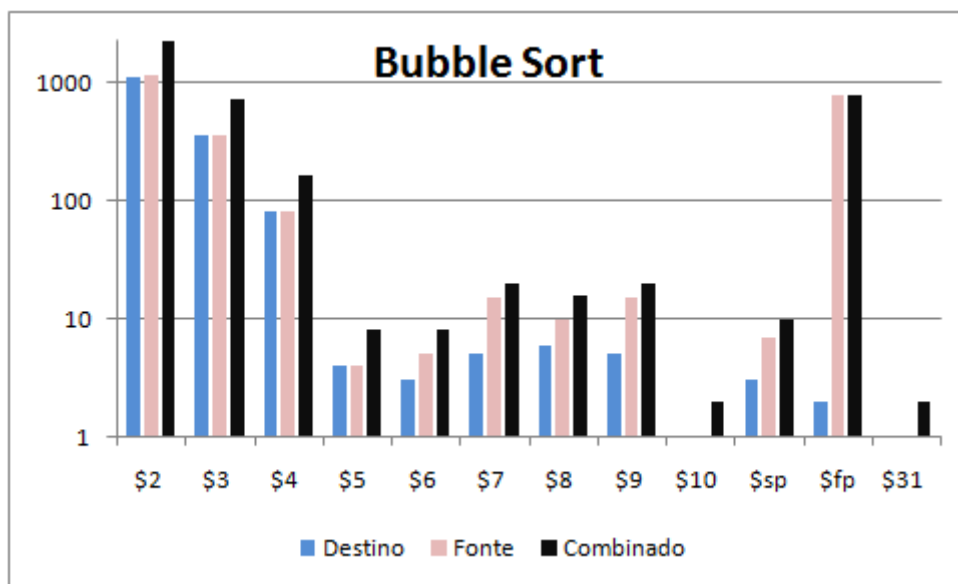


Figura 5.1: Uso dos registradores para o *bubble sort*: destino (representa o contador destino), fonte (representa o contador fonte) e combinado (representa o contador combinado).

O registrador \$2 é o mais usado como registrador destino e registrador fonte. O registrador \$fp é pouco usado como registrador destino, mas é bastante utilizado como registrador fonte. As utilizações dos registradores \$3 e \$4 são medianas tanto para uso como registrador fonte como destino. A utilização dos demais registradores é desprezível.

A figura 5.2 apresenta o gráfico da utilização dos registradores para a multiplicação de matrizes. As mesmas características, para os registradores \$2 e \$fp, apresentadas no *bubble sort* podem ser vistas na multiplicação de matrizes. Contudo, o registrador \$3 tem uma maior importância. Ele é o segundo mais importante para os métodos *Dynamic*

Target e *Dynamic Source and Target* e é o terceiro mais importante para o método *Dynamic Source*. Os registradores \$4, \$5 e \$6 têm uma baixa utilização. A utilização dos registradores \$10, \$sp e \$31 pode ser negligenciada quando comparada a dos demais registradores.

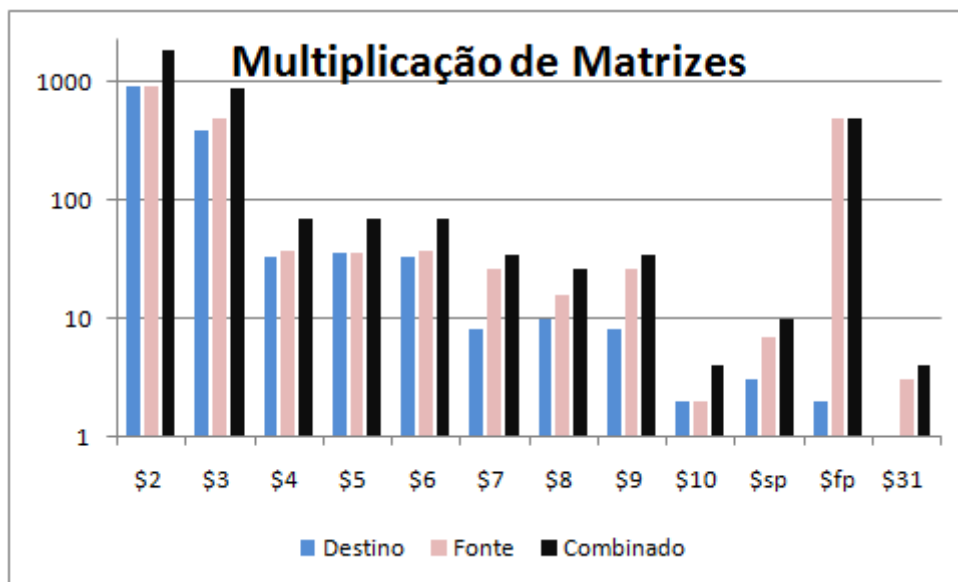


Figura 5.2: Uso dos registradores para a multiplicação de matrizes.

A figura 5.3 mostra a utilização dos registradores para o algoritmo de encriptação TEA2. Esse programa usa menos registradores do que o *bubble sort* e a multiplicação de matrizes. Desconsiderando o número de registradores usados, o algoritmo de encriptação TEA2 apresenta as mesmas características do *bubble sort* para a utilização dos registradores. O registrador \$2 é o mais usado, tanto como registrador fonte como registrador destino. O registrador \$fp é um pouco utilizado como registrador destino, mas há muitas operações de leitura sobre ele, fazendo com que ele seja bastante utilizado como registrador fonte. As utilizações dos registradores \$3 e \$4 são medianas tanto como registrador fonte como registrador destino. A utilização dos demais registradores é desprezível.

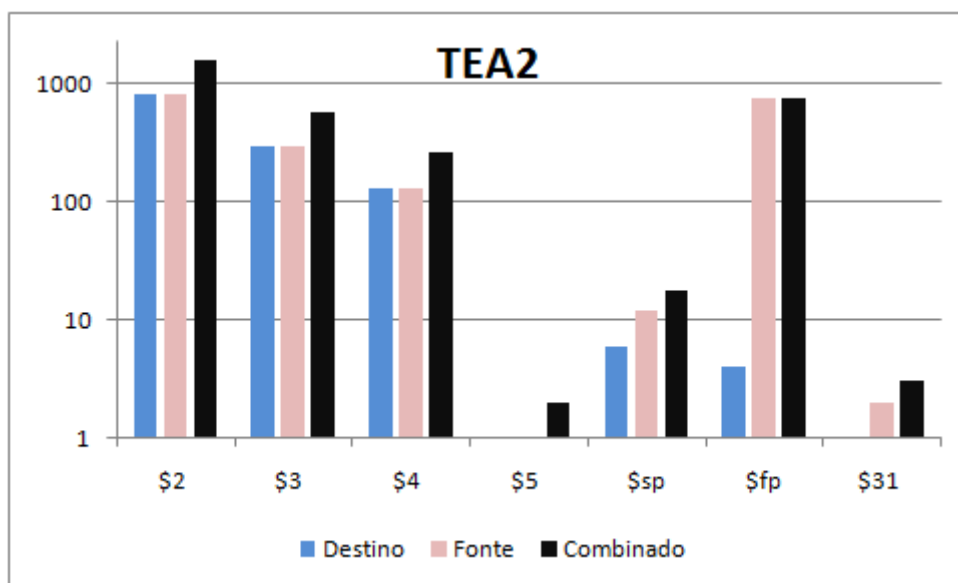


Figura 5.3: Uso dos registradores para o algoritmo de encriptação TEA2.

Poucos registradores são utilizados muitas vezes e a maioria dos registradores é utilizada poucas vezes. Esses poucos registradores representam quase a totalidade da utilização dos registradores pelos programas. Portanto, com uma pequena quantidade de registradores disponíveis, aproximadamente 6 registradores na multiplicação de matrizes e no *bubble sort* e 4 registradores no algoritmo de encriptação, é possível proteger a maioria das operações. Porém, isso também implica em um significativo aumento no tempo de execução e na ocupação de memória.

5.3 Tempo de Execução e Ocupação de Memória

Há sete métodos propostos. Para cada um, três diferentes quantidades de registradores disponíveis foram utilizadas. A quantidade de registradores disponíveis utilizada é maior que zero e menor que o número de registradores utilizados pelo programa. Uma versão com proteção total, onde todos os registradores foram duplicados também foi criada para comparação.

O *bubble sort* utiliza 12 registradores. As quantidades de registradores disponíveis para os métodos foram 3, 6 e 9, o que protege 25%, 50% e 75% dos registradores utilizados, respectivamente. A figura 5.4 mostra o tempo de execução para todas as versões do *bubble sort* protegido pelas técnicas selecionadas com diversos métodos para seleção de registradores e diferentes quantidades de registradores disponíveis. O tempo de execução da versão com proteção total também é apresentado.

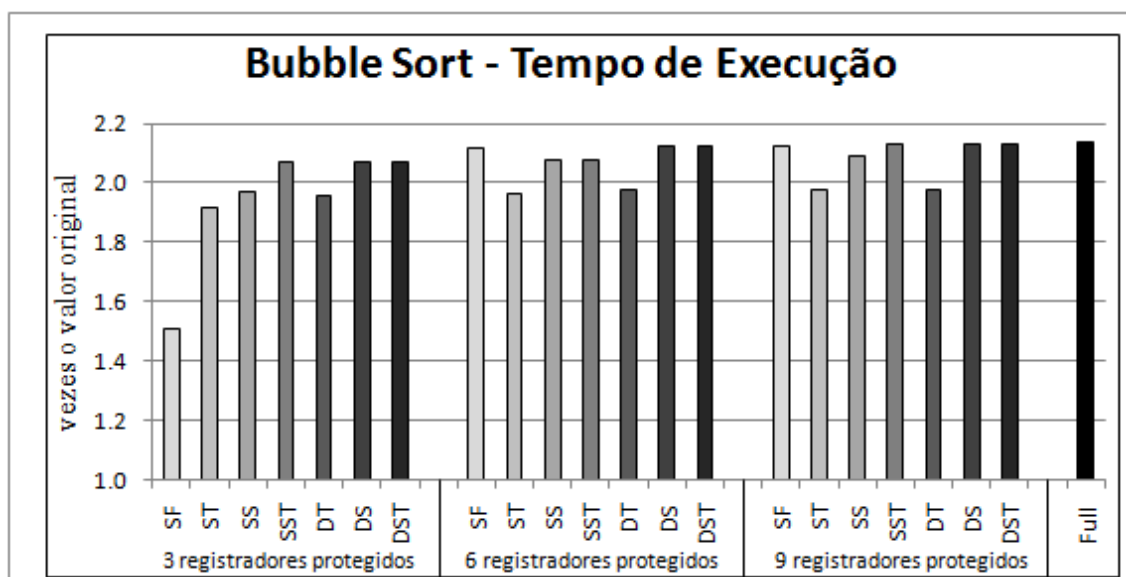


Figura 5.4: Tempos de execução para o *bubble sort*.

O tempo de execução da maioria dos programas protegidos é semelhante. Isso ocorre porque poucos registradores têm muito uso, fazendo a proteção deles significativamente mais custosa, em termos de tempo de execução, que os registradores menos utilizados. O *bubble sort* com seleção dos registradores pelo método *Static First* utilizando 3 registradores disponíveis tem um menor tempo de execução que as outras versões protegidas porque ele seleciona os registradores na ordem em que eles aparecem no código. Deste modo, ele duplica registradores com pouca utilização, fazendo com que a cobertura de erros seja menor.

A figura 5.5 apresenta a ocupação da memória para todas as versões do *bubble sort* protegidas pelas técnicas de detecção em *software* selecionadas com os diversos

métodos para diferentes quantidades de registradores disponíveis. Os registradores mais utilizados pelo programa foram selecionados para serem duplicados na maioria das versões. A diferença relativa entre o tempo de execução e a ocupação de memória se deve aos laços do *bubble sort*, que fazem um registrador contido em um desses laços ter um peso maior.

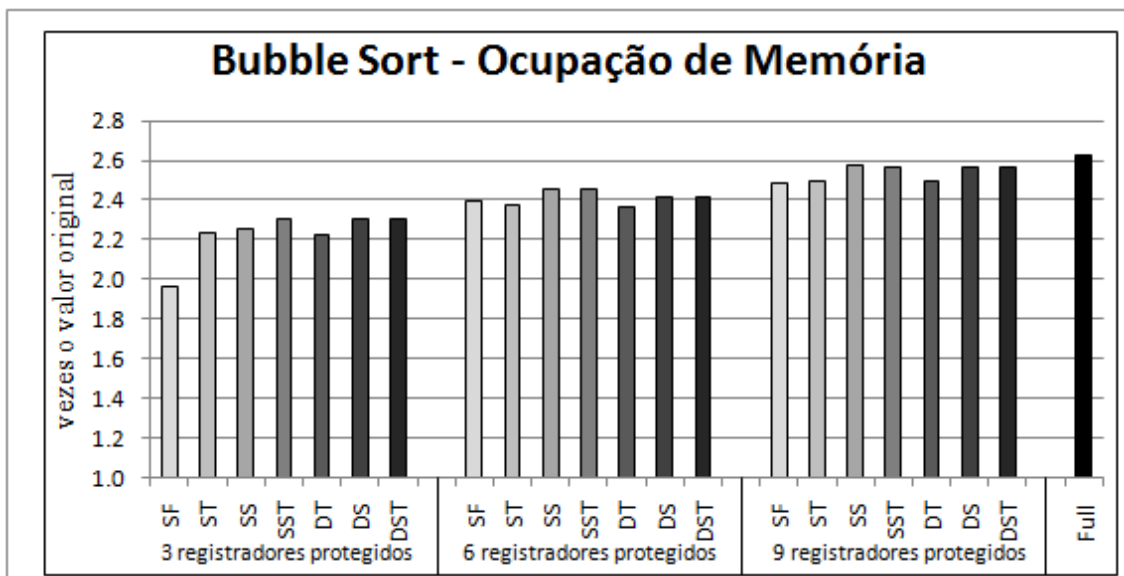


Figura 5.5: Ocupações de memória para o *bubble sort*.

A multiplicação de matriz utiliza 12 registradores. Por esse motivo, as quantidades de registradores disponíveis para os métodos são 3, 6 e 9, protegendo, respectivamente, 25%, 50% e 75% dos registradores utilizados. A figura 5.6 mostra os tempos de execução para todas as versões da multiplicação de matrizes protegidas pelas técnicas selecionadas utilizando diversos métodos para seleção dos registradores com diferentes quantidades de registradores disponíveis. É também apresentado o tempo de execução da versão com proteção total.

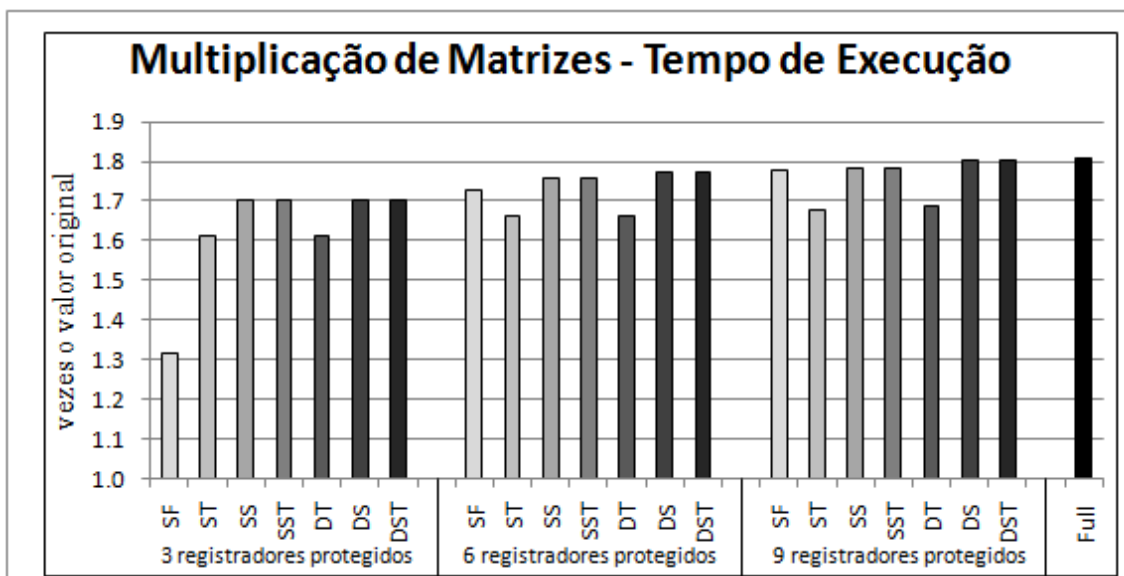


Figura 5.6: Tempos de execução para a multiplicação de matrizes.

A figura 5.7 apresenta as ocupações de memória de todas as versões da multiplicação de matrizes protegida pelas técnicas utilizando diversos métodos com diferentes quantidades de registradores disponíveis. A diferença proporcional do tempo de execução de um método para o outro é bastante semelhante à diferença proporcional da ocupação de memória.

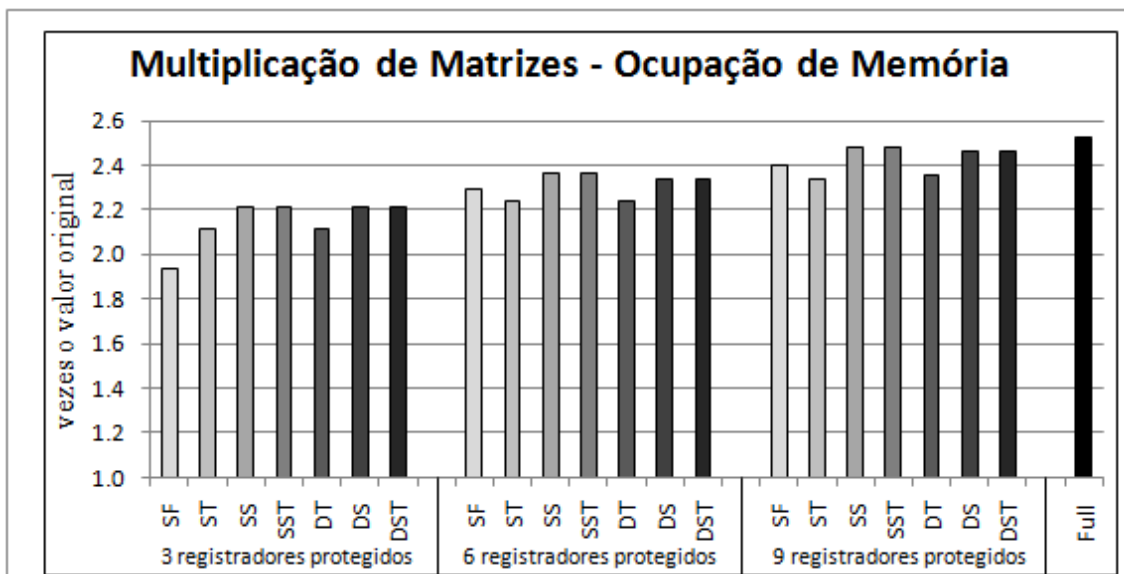


Figura 5.7: Ocupações de memória para a multiplicação de matrizes.

O algoritmo de encriptação TEA2 usa menos registradores que o *bubble sort* e a multiplicação de matrizes. Ele usa somente 7 registradores. Por esse motivo, para este programa, a quantidade de registradores disponíveis utilizada foi 2, 3 e 5, protegendo 28.6%, 42.9% e 71.4% dos registradores utilizados, respectivamente. A figura 5.8 apresenta os tempos de execução de todas as versões do algoritmo de encriptação TEA2 protegidos pelas técnicas selecionadas, utilizando diversos métodos para seleção dos registradores a serem duplicados com diferentes quantidades de registradores disponíveis. Os tempo de execução da versão com proteção total também é apresentado.

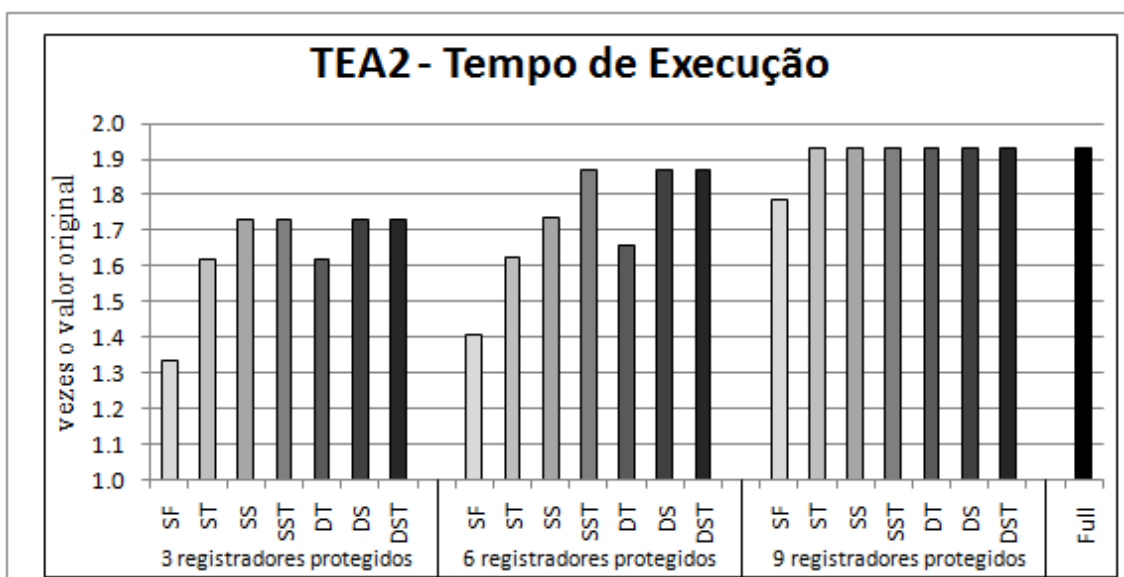


Figura 5.8: Tempos de execução para o algoritmo de encriptação TEA2.

A figura 5.9 apresenta a ocupação de memória para todas as versões do algoritmo de encriptação TEA2 protegido pela técnica de detecção em *software* selecionada, VAR3, utilizando diversos métodos para seleção dos registradores a serem duplicados com diferentes quantidades de registradores disponíveis. Assim como a multiplicação de matrizes, a diferença relativa entre o tempo de execução de um método para outro é bastante semelhante à diferença relativa da ocupação de memória entre esses métodos.

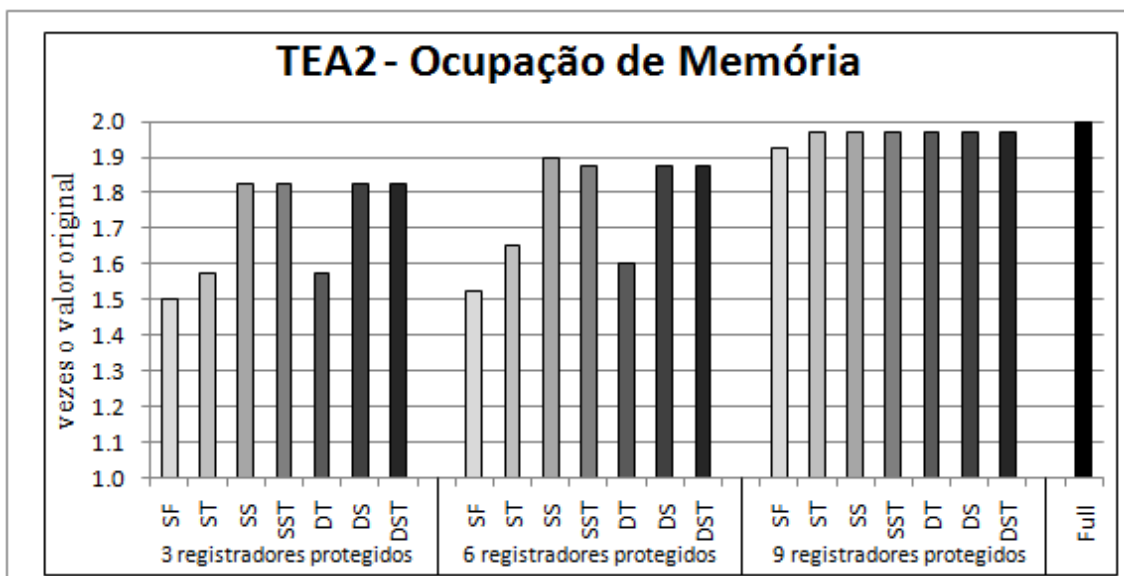


Figura 5.9: Ocupações de memória para o algoritmo de encriptação TEA2.

O tempo de execução e a ocupação de memória aumentam significativamente, mesmo quando há poucos registradores disponíveis para duplicação. Isso ocorre porque os registradores selecionados são muito mais utilizados que os registradores não selecionados para serem duplicados. Deste modo, quando a quantidade de registradores disponíveis cresce, o tempo de execução e a ocupação de memória não aumentam linearmente, devido à baixa utilização dos registradores selecionados com menor prioridade.

5.4 Resultados de Injeção de Falhas

Uma campanha de injeção de falhas foi realizada para avaliar a taxa de detecção de erros para os programas protegidos com técnicas de detecção em *software* utilizando diversos métodos para seleção dos registradores com diferentes quantidades de registradores disponíveis. Falhas foram injetadas no nível lógico do processador miniMIPS a ferramenta de simulação ModelSim. O local e o tempo em que cada falha foi inserida são selecionados aleatoriamente. Apenas uma falha foi injetada por execução. Os programas protegidos foram: um *bubble sort*, uma multiplicação de matrizes e um algoritmo de encriptação (TEA2). Os programas foram protegidos com a técnica VAR3 em todos os casos. A variação consiste no número de registradores disponíveis e do método de seleção de registradores utilizado. Esses dois fatores influenciam em como a técnica VAR3 é aplicada, visto que é ela que utiliza redundância nos dados.

Com o objetivo de verificar a eficiência das técnicas de detecção em *software* com os métodos de seleção de registradores propostos, somente as falhas que causaram erros foram consideradas. As falhas mascaradas pela lógica do processador foram ignoradas.

Um total de 10 mil falhas foram injetadas em cada programa protegido. Os erros foram classificados conforme seu efeito: se causaram efeito no controle ou se causaram efeito nos dados. Em alguns casos, o código do programa gerado com dois diferentes métodos foi o mesmo. Nesses casos, somente um deles foi submetido a falhas, sendo o resultado replicado para o outro. A tabela 5.1 mostra a taxa de detecção de erros para o *bubble sort* protegido pelas técnicas com diversos métodos de seleção de registradores e diferentes quantidades de registradores disponíveis rodando no processador miniMIPS. Os programas foram protegidos pela CFT-tool.

Tabela 5.1: Bubble Sort

Regs	Método	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção (%)
3	SF	770	5.3	852	75.5	1622	42.2
3	ST	781	6.8	911	89.4	1692	51.2
3	SS	711	7.6	1022	88.0	1733	55.0
3	SST	772	9.1	1158	93.9	1930	59.9
3	DT	777	6.8	1102	92.7	1879	57.2
3	DS	712	7.0	1169	92.2	1881	60.0
3	DST	712	7.0	1169	92.2	1881	60.0
6	SF	763	9.4	1274	96.8	2037	64.1
6	ST	759	10.0	1045	92.1	1804	57.5
6	SS	790	8.2	1155	93.2	1945	58.7
6	SST	763	7.5	1141	93.9	1904	59.2
6	DT	771	6.7	1105	92.7	1876	57.4
6	DS	759	8.0	1295	97.3	2054	64.3
6	DST	759	8.0	1295	97.3	2054	64.3
9	SF	715	8.0	1316	97.3	2031	65.9
9	ST	733	9.1	1070	93.2	1803	59.0
9	SS	758	6.6	1272	94.7	2030	61.8
9	SST	733	8.2	1317	97.2	2050	65.4
9	DT	709	9.7	1096	92.7	1805	60.1
9	DS	727	8.9	1311	97.6	2038	65.9
9	DST	727	8.9	1311	97.6	2038	65.9
12	Full	739	8.9	1300	96.9	2039	65.0

O *bubble sort* protegido pelas técnicas usando os métodos *Dynamic Source* e *Dynamic Source and Target* conseguiram taxas de detecção próximas a da versão com proteção total, mesmo quando havia somente 6 registradores disponíveis. Portanto, uma taxa de detecção de erros semelhante à versão com proteção total pode ser alcançada sem haver registradores disponíveis suficientes para duplicar todos os registradores em uso.

A tabela 5.2 apresenta o resultado da campanha de injeção de falhas para a multiplicação de matrizes.

Tabela 5.2: Multiplicação de Matrizes

Regs	Método	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção (%)
3	SF	864	3.6	1022	57.6	1886	32.9
3	ST	773	7.8	1157	76.7	1930	49.1
3	SS	720	6.4	1194	72.4	1914	47.5
3	SST	734	4.0	1230	74.6	1964	48.2
3	DT	734	4.0	1230	74.6	1964	48.2
3	DS	734	4.0	1230	74.6	1964	48.2
3	DST	734	4.0	1230	74.6	1964	48.2
6	SF	784	6.3	1315	82.4	2099	54.0
6	ST	726	7.9	1460	90.6	2186	63.1
6	SS	737	8.3	1633	91.1	2370	65.4
6	SST	788	7.2	1661	90.8	2449	63.9
6	DT	792	8.7	1439	88.7	2231	60.3
6	DS	746	8.6	1677	95.3	2423	68.6
6	DST	746	8.6	1677	95.3	2423	68.6
9	SF	759	7.5	1558	95.2	2317	66.5
9	ST	786	8.7	1523	90.7	2309	62.8
9	SS	697	8.9	1666	92.7	2363	68.0
9	SST	742	6.9	1602	91.9	2344	65.0
9	DT	766	7.0	1557	90.8	2323	63.2
9	DS	779	8.0	1811	97.1	2590	70.3
9	DST	779	8.0	1811	97.1	2590	70.3
12	Full	767	9.4	1773	97.7	2540	71.0

A taxa de detecção de erros utilizando os métodos propostos para seleção de registradores é semelhante à versão com proteção total somente com 9 registradores disponíveis. As versões que utilizam os métodos *Dynamic Source and Target* e *Dynamic Source*, com 6 registradores disponíveis, conseguem uma considerável taxa de detecção de erros, apenas um pouco menor que a versão completa.

As técnicas de detecção em *software*, utilizando os diversos métodos propostos com diferentes quantidades de registradores disponíveis também foram aplicadas ao algoritmo de encriptação TEA2, rodando no processador miniMIPS. A tabela 5.3 mostra o resultado da campanha de injeção de falhas para esse programa.

Tabela 5.3: Algoritmo de Encriptação (TEA2)

Regs	Método	# Erros no Controle	Taxa de Detecção (%)	# Erros nos Dados	Taxa de Detecção (%)	# Total de Erros	Taxa de Detecção (%)
2	SF	811	2.2	1317	56.4	2128	35.8
2	ST	750	9.1	1156	66.4	1906	43.9
2	SS	764	6.5	1296	68.4	2060	45.5
2	SST	720	7.2	1390	67.1	2110	46.6
2	DT	750	9.1	1156	66.4	1906	43.9
2	DS	720	7.2	1390	67.1	2110	46.6
2	DST	720	7.2	1390	67.1	2110	46.6
3	SF	763	2.2	1241	56.8	2004	36.0
3	ST	719	7.1	1148	65.3	1867	42.9
3	SS	762	5.8	1329	69.0	2091	46.0
3	SST	752	8.8	1419	80.0	2171	55.3
3	DT	715	7.0	1300	88.8	2015	59.8
3	DS	752	8.8	1419	80.0	2171	55.3
3	DST	752	8.8	1419	80.0	2171	55.3
5	SF	726	6.3	1309	72.3	2035	48.7
5	ST	770	10.1	1589	96.2	2359	68.1
5	SS	734	10.1	1581	95.8	2315	68.6
5	SST	698	9.7	1638	94.3	2336	69.0
5	DT	756	10.7	1541	94.7	2297	67.1
5	DS	758	9.1	1564	96.1	2322	67.1
5	DST	758	9.1	1564	96.1	2322	67.7
7	Full	776	11.6	1559	95.3	2335	67.5

O algoritmo de encriptação TEA2 utiliza menos registradores que o *bubble sort* e a multiplicação de matrizes. Por essa razão, a quantidade de registradores disponíveis utilizada nos testes foi menor. Com exceção do método *Static First*, todos os métodos com 5 registradores disponíveis conseguiram uma taxa de detecção de erros semelhante à versão com proteção total. Para 3 registradores disponíveis, o método *Dynamic Target* conseguiu uma considerável taxa de detecção de erros.

Os métodos *Dynamic Source and Target* e *Dynamic Source* mostraram taxa de detecção de erros semelhante à versão com proteção em todos os registradores, mesmo quando a quantidade de registradores disponíveis é metade da quantidade de registradores utilizados. Além disso, os programas protegidos utilizando esses métodos apresentam redução na taxa ocupação da memória em até 13%, pois menos registradores foram duplicados. O tempo de execução é um pouco menor, pois os registradores duplicados são aqueles mais utilizados dinamicamente. A redução no tempo de execução para os métodos *Dynamic Source and Target* e *Dynamic Source* foi de até 10%. Quando não há suficientes registradores disponíveis, esses métodos podem ser usados para obter uma taxa de detecção de erros próxima a que seria alcançada com a duplicação de todos os registradores.

6 CONCLUSÃO E TRABALHOS FUTUROS

A CFT-tool foi apresentada neste trabalho. Trata-se de uma ferramenta configurável capaz de aplicar automaticamente técnicas de detecção em *software* no código *assembly* de programas a serem protegidos.

O objetivo da CFT-tool é proteger processadores com diferentes arquiteturas e organizações contra falhas transientes. Um conjunto de técnicas de detecção de erros em *software* foi implementado. Essas técnicas podem ser aplicadas pela CFT-tool, de forma automática, no código *assembly* de um programa compilado para a arquitetura alvo.

A ferramenta pode ser utilizada para proteger diversos processadores com diferentes arquiteturas e organizações. Para isso, ela utiliza arquivos de configurações, onde as informações necessárias sobre a arquitetura e organização do processador alvo são informadas. Além disso, a CFT-tool permite também configurar quais técnicas serão aplicadas no código *assembly* do programa e quais registradores devem ser protegidos, caso alguma técnica de proteção dos dados seja selecionada. Diversos métodos para determinar a prioridade que cada registrador tem para ser duplicado estão também implementados e podem ser selecionados pelo usuário.

Para verificar a eficiência da ferramenta, em termos de taxa de detecção de erros, tempo de execução e ocupação de memória, dos programas protegidos pelas técnicas aplicadas pela CFT-tool foi utilizado o processador miniMIPS, pois esse possui, na literatura, resultados para as técnicas implementadas. Deste modo, foi possível concluir que a CFT-tool apresenta resultados semelhantes em termos de taxa de detecção de erros, tempo de execução e ocupação de memória que os presentes na literatura. A variação na taxa de detecção de erros para as diversas técnicas aplicadas foi, em geral, inferior a 1%. O tempo de execução e a ocupação de memória apresentaram variação máxima de 15%, em alguns casos em favor da CFT-tool e outros em favor dos resultados presentes na literatura.

A CFT-tool é proposta como uma ferramenta configurável, independente da arquitetura e organização do processador, podendo ser utilizada por diversos processadores. Por esse motivo, foi necessário validar a configurabilidade da CFT-tool. Essa validação foi realizada utilizando o processador LEON3, pois esse é amplamente utilizado em aplicações espaciais e é baseado em uma arquitetura diferente da arquitetura do processador miniMIPS, utilizado na validação da correta aplicação e eficiência das técnicas implementadas na CFT-tool. A configurabilidade da CFT-tool, isto é, a capacidade dela de aplicar técnicas de detecção em *software* em um código compilado para um diferente processador, o mantendo funcional e sendo capaz de detectar erros, foi verificada com o processador LEON3. O processador LEON3 não

possui registradores disponíveis suficientes para duplicar todos os registradores utilizados pelo programa, deixando partes do código desprotegidas. Mesmo assim, as taxas de detecção de erros obtidas não foram muito inferiores às obtidas para o processador miniMIPS, ficando cerca de 10% abaixo dos resultados do miniMIPS. Além disso, o processador LEON3 possui instruções para comparação separada dos desvios condicionais, isso acarreta uma carga maior em termos de tempo de execução e ocupação de memória. Apesar disso, a CFT-tool mostrou-se capaz de aplicar as técnicas com eficácia em diferentes processadores, mantendo os programas funcionais e capazes de detectar erros. Para o processador miniMIPS, as taxas de detecção de erros ficaram por volta de 70% e para o LEON3 na faixa de 60%. A detecção de erros afetando os dados foi alta, sendo que para o processador LEON3 ela ficou na faixa de 85% de detecção de erros e para o processador miniMIPS, acima dos 95%.

Como há casos de falta de registradores disponíveis suficientes para duplicar todos os registradores utilizados, uma análise do impacto da seleção seletiva de registradores na taxa de detecção de erros foi realizada. Diversos métodos para seleção de registradores com diferentes quantidades de registradores disponíveis foram avaliados para um *bubble sort*, uma multiplicação de matrizes e um algoritmo de encriptação (TEA2). Os resultados mostraram que poucos registradores são responsáveis pela maioria do uso dos registradores pelo programa. Uma campanha de injeção de falhas mostrou que os métodos *Dynamic Source and Target* e *Dynamic Source* alcançam a mesma taxa de detecção de erros que o programa com proteção em todos os registradores, mesmo quando a quantidade de registradores disponíveis para esses métodos é metade do número de registradores utilizados.

A CFT-tool é capaz de aplicar diferentes técnicas de detecção em software independentemente da arquitetura, mostrando ser eficaz para diferentes processadores. Novas técnicas de detecção em *software* podem ser adicionadas à CFT-tool, o que possibilita obter taxas de detecção de erros maiores. A CFT-tool pode ser utilizada para proteger programas para outras arquiteturas e organizações modificando os arquivos de configuração. A configuração das técnicas é segundo as especificações da aplicação, recursos do processador e seleções do usuário.

Como trabalho futuro, novas técnicas de detecção em *software*, protegendo o código completo do programa e/ou registradores ou partes deles serão estudadas, principalmente as destinadas ao aumento na detecção de erros afetando o controle do programa, já que a detecção desses pelas técnicas implementadas foi inferior. As técnicas já implementadas e as novas técnicas a serem desenvolvidas serão testadas em outros processadores além dos já utilizados, com diferentes arquiteturas e organizações, abordando processadores RISC, CISC, DSP e VLIW. Modificações nas técnicas visando um melhor custo-benefício em termos de desempenho, ocupação de memória e taxa de detecção de erros serão propostas.

REFERÊNCIAS

AEROFLEX GAISLER, LEON3, 2004. Disponível em: <<http://www.gaisler.com>>, Acesso em: 2010.

ALKHALIFA, Z. et al. Design and evaluation of system-level checks for on-line control flow error detection. **IEEE Transactions on Parallel and Distributed Systems**, New York, v. 10, n. 6, p. 627-641, 1999.

ASENSI, S. C.; ALVAREZ, A. M.; CALLE, F. R.; PALOMO, F. R.; MIRANDA, H. G.; AGUIRRE, M. A. A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems. **IEEE Transactions on Nuclear Science**, vol. 58, n. 3, p. 1059-1065, jun. 2011.

AZAMBUJA, J. R. F. **Análise de Técnicas de Tolerância a Falhas Baseadas em Software para a Proteção de Microprocessadores**. 2010. 106f. Dissertação (Mestrado em Computação) - Instituto de Informática, UFRGS, Porto Alegre.

AZAMBUJA, J. R.; LAPOLLI, A.; ROSA, L.; KASTENSMIDT, F. L. Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique. **IEEE Transactions on Nuclear Science**, [S.l.], v. 58, n. 3, p. 993-1000, jun. 2011.

AZAMBUJA, J. R.; LAPOLLI, A.; ROSA, L.; KASTENSMIDT, F. L. Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors, **IEEE Latin American Symposium on Circuits and Systems**, Foz do Iguaçu, v. 1. p. 300-303., 2010.

AZAMBUJA, J. R.; SOUSA, F.; ROSA, L.; KASTENSMIDT, F. L.; Non-Intrusive Hybrid Signature-Based Technique to Detect SEU and SET Faults in Microprocessors. **European Conference on Radiation and Its Effects on Components and Systems**, v. 11, 2010.

CHEYNET, P; NICOLESCU, B.; VELAZCO, R.; REBAUDENGO, M.; REORDA, M. S.; VIOLANTE, M. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. **IEEE Transactions On Nuclear Science**, [S.l.], v. 47, n. 6, p. 2231-2236, dez. 2000.

DODD, P. E.; SHANEYFELT, M. R.; SCHWANK, J.R.; FELIX; J.A. Current and Future Challenges in Radiation Effects on CMOS Electronics. **IEEE Transactions on Nuclear Science**, v. 57, n. 4, p. 1747-1763, ago. 2010.

FERLET-CAVROIS. V. et al. Direct measurement of transient pulses induced by laser irradiation in deca-nanometer SOI devices. **IEEE Transactions On Nuclear Science**, Los Alamitos, v. 52, 2005.

HANGOUT, L. M. O. S. S.; JAN, S. The minimips project, 2009. Disponível em: <<http://www.opencores.org/projects.cgi/web/minimips/overview>>. Acesso em: out. 2010.

MCFEARIN, L; NAIR, V. Control-flow checking using assertions. **Conference on Dependable Computing for Critical Applications, Urbana-Champaign**, p. 103-112, 1995.

MENTOR GRAPHICS. ModelSim, 2010. Disponível em: <<http://www.model.com/content/modelsim-support>>. Acesso em: 2010.

NICOLESCU, B.; VELAZCO, R. Detection soft errors by a purely software approach: method, tools and experimental results. **Design, Automation and Test in Europe Conference and Exhibition**, p57-62, 2003.

OH, N.; MITRA, S.; McCLUSKEY, E. ED4I: error detection by diverse data and duplicated instructions. **IEEE Transactions on Computers**, v. 51, n. 2, p. 180- 199, 2002a.

OH, N.; SHIRVANI, E.; McCLUSKEY, E. Control-flow checking by software signatures. **IEEE Transactions on Reliability**, [S.l.], v. 51, n. 2, p. 111-122, mar. 2002b.

PRADHAN, D. Fault-tolerant computer system design. Upper Saddle River, USA : Prentice-Hall, 1995.

REBAUDENGO, M; REORDA, M. S.; TORCHIANO, M.; VIOLANTE, M. Soft-error detection through software fault-tolerance techniques. **IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems**, Albuquerque, p. 210-218, 1999.

REIS, G. A.; CHANG, J.; VACHHARAJANI, N.; RANGAN, R.; AUGUST, D. I. SWIFT: *software* implemented fault tolerance. **Symposium on Code Generation and Optimization**, San Francisco, p. 243-254, 2005.

SPARC INTERNATIONAL INC. The SPARC Architecture Manual Version 8, 1991.

THOMPSON, S. et al. In search of forever: continued transistor scaling one new material at a time. **IEEE Transactions on Semiconductor Manufacturing**, New York, v. 18, n.1, p. 26-36, 2005.

ANEXO A - DETALHES TÉCNICOS DA IMPLEMENTAÇÃO

As 23 classes em Java implementadas na ferramenta CFT-Tool, totalizando quase 6 mil linhas de código, estão organizadas em pacotes, sendo cada pacote destinado a uma funcionalidade da ferramenta. Os pacotes são: *cft*, *asmgen*, *config*, *error*, *techniques*, *util* e *verif*. O pacote *cft* é a raiz da ferramenta e contém todas as classes e pacotes. No pacote *asmgen* estão classes destinadas à realização do processo de complementação do código, onde as sub-rotinas não descritas no código pré ligação são localizadas e convertidas do formato *memory dump* para código *assembly*. O pacote *config* contém classes destinadas ao carregamento das informações referentes ao processador e as técnicas. Os dados são armazenados e informados a outras classes quando requisitados. O pacote *error* contém classes responsáveis pelo tratamento dos erros detectados pelas técnicas. As técnicas estão contidas no pacote *techniques*. As classes desse pacote são responsáveis por aplicar as técnicas de detecção em *software* sobre o código do programa a ser protegido. As classes do pacote *util* contém funções auxiliares que facilitam a execução de tarefas pelas demais classes da CFT-tool. No pacote *verif* estão classes responsáveis por verificar se o código do programa a ser protegido é compatível com as configurações da arquitetura e organização do processador apresentadas. Ela também prepara o código para proteção, removendo comentários, por exemplo.

A seguir, são detalhadas as classes implementadas na ferramenta CFT-tool.

Classe CFT

A classe CFT é a classe principal da CFT-tool. Ela é responsável pelo fluxo geral de execução da ferramenta, invocando a execução dos módulos na ordem correta para a aplicação das técnicas de detecção em *software* selecionadas, sobre o programa. Primeiro é invocada a classe que realiza o carregamento das configurações. Depois, a classe que remove os comentários e verifica se o código é condizente com as configurações informadas é chamada. A classe responsável pelo gerenciamento do módulo de complementação do código *assembly* é então chamada. Depois disso, a classe que controla o módulo de proteção é invocada. Durante esse processo são gerados vários arquivos temporários. O último deles é renomeado, pois se trata do código do programa protegido com as técnicas selecionadas. Os demais arquivos temporários são removidos e o programa é finalizado. Essa classe está na raiz da CFT-tool, pertencendo ao pacote *cft*.

Classe AssemblyGenerator

A classe *AssemblyGenerator* faz parte do pacote *asmgen*. Ela é responsável por localizar as sub-rotinas não descritas no código pré ligação presentes no código

assembly do programa a ser protegido. Ela gera uma lista com as sub-rotinas não descritas no código pré ligação, vai informando uma a uma a classe CodeGenerator e adiciona o código gerado ao final do código *assembly* do programa. Após todas as sub-rotinas da lista terem sido convertidas, é realizada uma nova busca por sub-rotinas não descritas no código pré ligação, pois as sub-rotinas recém inseridas no código *assembly* podem chamar novas sub-rotinas não descritas no código pré ligação que antes não estavam presentes. Esse processo se repete até que não haja mais sub-rotinas não descritas no código pré ligação. Para saber que se trata de uma sub-rotina não descrita no código pré ligação, todo o código *assembly* do programa é percorrido, sendo geradas duas listas, uma contendo todos os *labels*, que são os possíveis nomes das sub-rotinas presentes no código, e outra contendo todas as chamadas para sub-rotinas. A lista de sub-rotinas chamadas é comparada com a lista de sub-rotinas presentes no código. As sub-rotinas que foram chamadas, mas não estão presentes são identificadas como sendo sub-rotinas não descritas no código pré ligação e essas são passadas para a classe CodeGenerator.

Classe CodeGenerator

A classe CodeGenerator pertence ao pacote asmgen. Ela é responsável por converter o código do formato *memory dump* para código *assembly*. O nome de uma sub-rotina é informado no momento em que a classe é invocada pela classe AssemblyGenerator. Primeiramente, a sub-rotina é localizada no arquivo em formato *memory dump*. Depois disso, os endereços dos desvios, que no formato *memory dump* são endereços físicos, são convertidos para *labels*. Então nomes dos registradores e instruções são traduzidos para os equivalentes em *assembly*. Dependendo do processador, uma ou mais instruções sequenciais aos saltos e desvios condicionais são sempre executadas com o objetivo de se aproveitar de características do *pipeline* e com isso melhorar o desempenho. Nesses casos, as instruções que desviam o fluxo de execução do programa devem ser movidas para baixo para manter a corretude do programa. O montador irá movê-las para cima no momento da montagem. Contudo, as instruções devem permanecer no mesmo bloco de código, isto é, não podem cruzar um *label*. Por fim, dependendo das características do montador e da arquitetura do processador, as instruções de não operação podem ser removidas, pois o montador irá inseri-las se necessário. A sub-rotina terá seu código *assembly* gerado e escrito em um arquivo temporário. As informações referentes ao formato do *memory dump* são passadas pela classe ConfigParser.

Classe Architecture

As informações sobre a arquitetura e organização do processador alvo são armazenadas na classe Architecture após o carregamento dos arquivos de configurações. Todas as instruções e registradores são armazenados nessa classe. O formato dos *labels* e das instruções contendo seus padrões para identificação dos mnemônicos, registradores, imediatos, *labels*, também estão contidos na classe, que também contém dados sobre *branch delay slot*, que se trata de quantas instruções são executadas após os desvios, além de uma lista contendo os desvios condicionais e outra contendo os desvios logicamente inversos aos da primeira lista. As principais funções dessa classe são passar informações sobre as instruções, as quais, na CFT-tool são instâncias da classe Instruction. Os registradores, por sua vez, são instâncias da classe Register. A classe Architecture pertence ao pacote config.

Classe ConfigParser

A classe ConfigParser contém informações necessárias para que a classe CodeGenerator seja capaz de entender o programa no formato *memory dump*. Definições dos registradores, seus nomes no formato *memory dump* e o equivalente em *assembly*, bem como das instruções são informados por essa classe. As instruções ainda são classificadas conforme seu tipo, dando enfoque para desvios condicionais e saltos incondicionais que merecem atenção especial na conversão do formato *memory dump* para *assembly*. Dados sobre o *branch delay slot*, que se trata do número de instruções executadas após saltos e desvios, e sobre o padrão como os dados estão dispostos também são informados pela classe ConfigParser. Essa classe faz parte do pacote config.

Classe DB

A classe DB é a principal responsável pela comunicação entre as classes que realizam operações do programa a ser protegido e as classes que contêm as configurações da arquitetura e organização do processador e das técnicas a serem aplicadas. Ela serve de interface, disponibilizando funções que realizam a identificação de *labels*, instruções, registradores e valores imediatos. Dependendo da arquitetura, pode haver mais de uma instrução com o mesmo nome. A classe DB requisita à classe Architecture por instruções informando seu nome. A classe Architecture irá devolver uma lista contendo todas as instruções que têm aquele nome. Para realizar a identificação de qual instrução que se trata, a classe DB avalia os formatos das instruções, considerando a quantidade de registradores fonte e se há registrador destino. Quando alguma classe necessita saber se a instrução ou o registrador é de um tipo ou de outro, ela requisita a informação da classe DB, que por sua vez invoca a classe Architecture. As classes que trabalham com a manipulação do código *assembly* do programa precisam, muitas vezes, algum item isolado da instrução. Para isso elas invocam a função da classe DB que realiza tal função. A classe DB também serve de interface entre as configurações das técnicas e as classes que realizam operações sobre o código. A classe DB pertence ao pacote config.

Classe Loader

A classe Loader pertence ao pacote config e é responsável pela realização do carregamento das configurações referentes à arquitetura e organização do processador alvo e das técnicas de detecção em *software* a serem aplicadas sobre o código do programa a ser protegido. Primeiramente são carregadas as configurações gerais, tais como formato dos *labels*, tag de comentário, caractere de separação de operandos, dados sobre ocorrência do *branch delay slot* no código *assembly*, registradores pré-inicializados, que já continham valores antes do início da execução do programa. Também nas configurações gerais estão contidos alguns dados sobre as instruções, como os desvios condicionais e os desvios logicamente inversos a esses. Depois do carregamento das configurações gerais, é realizado o carregamento das informações referentes aos registradores. O nome dos registradores, bem como seu tipo e se ele pode ser lido ou escrito são carregados. Depois disso, são carregadas as informações referentes às instruções, tais como, seu mnemônico, formato, tipo (aritmético, *load*, *store*, saltos incondicionais, desvios condicionais, não operação e outro) e se utiliza algum registrador implicitamente, seja como registrador fonte ou registrador destino. Todas essas configurações são passadas para a classe DB que toma as devidas

providências, configurando as classes que serão posteriormente usadas como base para a realização das operações sobre o programa a ser protegido.

Classe Techniques

A classe Techniques armazena as configurações referentes à aplicação das técnicas de detecção em *software*. Os nomes das técnicas que serão aplicadas sobre o programa são armazenados nessa classe. Quando utilizado, o registrador de erro e seu valor em caso de detecção de erro são informados às demais classes da CFT-tool pela classe Techniques. Também é responsabilidade dessa classe informar o deslocamento entre os dados originais e suas cópias em memória. O modo de prioridade para seleção dos registradores, bem como uma lista contendo os registradores em ordem, quando utilizada a prioridade personalizada são armazenados nessa classe. A classe Techniques faz parte do pacote config.

Classe ErrorManager

A classe ErrorManager faz parte do pacote error e é responsável por informar que erros de configuração foram detectados, finalizando a execução da CFT-tool.

Classe Manager

A classe Manager faz parte do pacote techniques. Ela é responsável por gerenciar a aplicação das técnicas de detecção em *software*. Primeiramente, a classe Manager chama uma função da classe ErrorTreatment, a qual substitui o registrador que foi selecionado, caso algum tenha sido, para ser destinado a sub-rotina de tratamento de erro por um registrador livre, do mesmo tipo. As classes que aplicam as técnicas de detecção em *software* vão sendo invocadas na ordem em que foram informadas nos arquivos de configurações. Cada técnica aplicada vai gerando um novo arquivo temporário. O último arquivo temporário gerado, o qual possui todas as técnicas selecionadas aplicadas é passado para uma função da classe ErrorTreatment que adiciona a sub-rotina de tratamento de erro e retorna o código protegido. Após isso, a execução da CFT-tool volta para a classe principal CFT, onde os procedimentos finais são realizados.

Classe ErrorTreatment

A classe ErrorTreatment é responsável pela liberação do registrador de erro, que será destinado a sub-rotina de tratamento de erro e pela inclusão da sub-rotina de tratamento de erro ao código *assembly* do programa. Antes de aplicar as técnicas de detecção em *software*, a classe Manager irá chamar a função da classe ErrorTreatment responsável pela liberação do registrador destinado à sub-rotina de tratamento de erro. Essa função primeiramente identifica um registrador disponível que seja do mesmo tipo do registrador que será destinado à sub-rotina de tratamento de erro. O código do programa a ser protegido é percorrido e toda ocorrência do registrador de erro é substituída pelo registrador disponível selecionado. Desse modo, o registrador de erro fica disponível para ser usado pela sub-rotina de tratamento de erro. Após a aplicação das técnicas de detecção em *software*, a classe Manager irá invocar novamente a classe ErrorTreatment para que essa inclua a sub-rotina de tratamento de erro no código *assembly* do programa. Depois disso, a execução da CFT-tool é devolvida à classe Manager que, por sua vez, devolve a classe principal, CFT. A classe ErrorTreatment pertence ao pacote techniques.

Classe BRA

Esta classe aplica a técnica BRA sobre o código *assembly* do programa a ser protegido. A classe percorre o código *assembly* do programa a ser protegido, buscando por desvios condicionais. Caso o destino do desvio condicional não seja a sub-rotina de tratamento de erro, ele aplica a técnica BRA. Primeiramente, o destino do desvio é substituído pelo *label* referente ao bloco de tratamento desse desvio. Em seguida, o desvio é replicado, ou seja, é inserido um desvio igual abaixo dele, com uma única diferença, o destino do desvio é a sub-rotina de tratamento de erro. Feito isso, a classe busca pelo destino original do desvio, para lá inserir o bloco de tratamento de erro. Localizado o *label* do destino original do desvio, é inserido antes desse *label* um salto incondicional para esse *label*, para manter a corretude do código. Depois do salto, são inseridos o *label* do bloco de tratamento do desvio, um desvio logicamente inverso ao original com destino para a sub-rotina de tratamento de erro e um salto incondicional para o *label* do destino original do desvio que se está protegendo. Esse salto é necessário, pois mais de um desvio condicional pode ter o mesmo destino, fazendo com que mais de um bloco de tratamento seja criado sobre o mesmo *label*. Com o salto, a corretude é mantida. Após a aplicação da técnica, os saltos com destino para os *labels* imediatamente abaixo de si podem ser removidos. Instruções de não operação podem ser incluídas após os desvios condicionais e saltos incondicionais, se necessário, quando a característica do *branch delay slot* estiver presente no processador. A classe BRA faz parte do pacote *techniques*.

Classe SIG

Essa classe está contida no pacote *techniques* e é responsável pela aplicação da técnica SIG. O código *assembly* do programa é percorrido. Os *labels*, desvios condicionais, saltos incondicionais marcam o início e fim dos blocos básicos. Algumas arquiteturas possuem instruções para comparação separadas dos desvios, essas instruções, quando existentes, marcam o fim de um bloco básico. A área após a instrução de comparação e o desvio é considerada área livre e não é protegida. Contudo, na grande maioria dos casos onde as instruções de comparação são separadas dos desvios condicionais, elas estão imediatamente antes deles, fazendo a aplicação dessa técnica efetiva nesse tipo de arquitetura também. As demais instruções além das citadas são consideradas instruções regulares e servem para auxiliar na definição de se o local avaliado se trata de um início, meio ou fim de bloco básico. A linha atual do código é lida e comparada com a anterior e, conforme a distribuição das instruções lidas e possivelmente do *label* lido, é definido a parte do bloco básico. Se for um início de bloco básico, é inserida uma instrução que atribui o valor de uma constante contendo o valor da assinatura do bloco básico em questão a um registrador selecionado para a aplicação dessa técnica. Caso seja um fim de bloco básico, é inserida uma instrução para comparar o valor do registrador destinado à aplicação da técnica com o valor esperado para a assinatura desse bloco básico, verificando, dessa forma, se houve, ou não, um desvio para outro bloco básico. Caso não exista instrução na arquitetura alvo para comparação de registrador com valor imediato, o valor esperado para a assinatura é subtraído do registrador utilizado pela técnica e o registrador é comparado com zero, utilizando uma instrução de comparação com zero ou comparando o registrador com o registrador zero, normalmente presente na maioria das arquiteturas e, que possui seu valor constante em zero. Se o processador utiliza *branch delay slot*, é inserido o número necessário de instruções de não operação após o desvio para manter a corretude do código. Caso a comparação seja realizada em uma instrução separada do desvio, a

instrução de comparação é inserida e, imediatamente depois é inserido o desvio com destino para a sub-rotina de tratamento de erro em caso de discrepância no valor do registrador e da assinatura do bloco básico em questão.

Classe VAR1

Essa classe pertence ao pacote *techniques* e realiza a aplicação da técnica VAR1 sobre o código *assembly* do programa a ser protegido. Primeiramente, uma lista contendo os registradores utilizados pelo programa e outra com os registradores livres são geradas. Os registradores livres são associados aos registradores utilizados, replicando as operações dos registradores originalmente utilizados. Caso não haja registradores livres suficientes para replicar todos os registradores utilizados, os registradores livres serão associados aos registradores utilizados seguindo o critério de prioridade de registradores informado. Após isso, a classe percorre o código do programa, replicando, sobre os registradores cópias, as mesmas operações que foram realizadas sobre os registradores utilizados no código original. Algumas instruções utilizam registradores que não estão explicitamente informados. Esses registradores também são considerados na aplicação da técnica. Além disso, são inseridos instruções para comparar o valor dos registradores originais com suas cópias, conforme as regras da técnica VAR1.

Classe VAR2

A aplicação da técnica de detecção em *software* VAR2 é realizada por essa classe. Da mesma forma que a classe VAR1, são criadas duas listas, uma lista contendo os registradores utilizados pelo programa e outra contendo os registradores livres. Os registradores livres são associados aos registradores utilizados, replicando as operações dos registradores originalmente utilizados. Essa associação dos registradores obedece uma regra de prioridade informada. Dessa forma, caso não haja registradores livres suficientes para replicar todos os registradores utilizados, os registradores livres serão associados aos registradores utilizados seguindo essa prioridade. Depois de selecionado os registradores cópias, o código do programa é percorrido, replicando, sobre os registradores cópias, as mesmas operações que foram realizadas sobre os registradores utilizados no código original. Algumas instruções utilizam registradores que não estão explicitamente informados. Esses registradores também são considerados na aplicação da técnica. Também são inseridos instruções para comparar o valor dos registradores originais com suas cópias, conforme as regras da técnica VAR2. Essa classe pertence ao pacote *techniques*.

Classe VAR3

A classe VAR3 pertence ao pacote *techniques*. Ela é responsável pela aplicação da técnica VAR3. Duas listas são criadas duas listas, uma lista contendo os registradores utilizados pelo programa e outra contendo os registradores livres. Da mesma forma que as classes VAR1 e VAR2, os registradores livres são associados aos registradores utilizados, conforme a regra de prioridade de registradores, replicando as operações dos registradores originalmente utilizados. Dessa forma, caso não haja registradores livres suficientes para replicar todos os registradores utilizados, os registradores livres serão associados aos registradores utilizados seguindo a ordem informada, conforme o critério de prioridades. Após a seleção dos registradores, o código do programa é percorrido, replicando as mesmas operações que foram realizadas sobre os registradores utilizados

no código original sobre os registradores cópias. Algumas instruções utilizam registradores que não estão explicitamente informados. Esses registradores também são considerados na aplicação da técnica. Além disso, são inseridos instruções para comparar o valor dos registradores originais com suas cópias, conforme as regras da técnica VAR3.

Classe Global

A classe Global faz parte do pacote util. Sua única função é a definição e controle dos nomes dos arquivos temporários, os quais podem ser solicitados por qualquer classe do programa. A centralização da geração dos nomes dos arquivos temporários torna mais fácil seu manuseio.

Classe Instruction

A classe Instruction serve para armazenar informações sobre as instruções. Uma instância dessa classe é criada para cada instrução informada no arquivo de configurações referentes às instruções. A classe Loader carrega as configurações, informa à classe DB, que, por sua vez chama a classe Architecture, a qual cria as instâncias da classe Instruction. Além disso, a classe Instruction é muito utilizada pelas demais classes da CFT-tool. Todo local que trabalha com tratamento de instruções cria instâncias dessa classe, como é o caso das classes Architecture, DB, SIG, BRA, VAR1, VAR2, VAR3 e ErrorTreatment. As instruções possuem cinco atributos que as definem, são eles: o nome da instrução, o formato dela (posição do mnemônico, dos registradores fonte e destino, dos valores imediatos), o tipo da instrução (aritmética, *load*, *store*, desvio condicional, salto incondicional, não operação e outro), além de dois vetores contendo possíveis registradores que são implicitamente utilizados pela instrução, um para registradores implicitamente utilizados como registrador fonte e outro para registradores utilizados implicitamente como registrador destino. É possível que duas instruções tenham o mesmo nome, por esse motivo, a identificação correta da instrução é realizada considerando, além do nome, características do formato. Esse processo é realizado pela classe DB. A classe Instruction pertence ao pacote util.

Classe Register

A classe Register é utilizada para representar os registradores do processador alvo. Cada registrador é uma instância da classe Register. Os registradores possuem quatro atributos: nome, que serve para identificar o registrador, tipo, que identifica se o registrador é global, local, de entrada ou de saída, além de dois valores lógicos que identificam se o registrador pode ser lido e/ou escrito explicitamente. Os diferentes tipos de registrador são úteis para quando o processador implementa registradores em janelas, onde as chamadas de sistema mudam os registradores locais, de entrada e de saída. A classe Register é utilizada pelas mesmas classes que utilizam a classe Instruction, são elas: Architecture, DB, SIG, BRA, VAR1, VAR2, VAR3 e ErrorTreatment. A classe Register faz parte do pacote util.

Classe Util

A classe Util implementa muitas funções que auxiliam as demais classes na execução de suas tarefas. Funções para conversão de bases, manipulação de listas, formatação e tratamento de texto, manipulação de arquivos e utilização de expressões regulares para identificação de elementos e padrões. A classe Util é muito utilizada e

tem fundamental importância para praticamente todas as classes da CFT-tool. Ela pertence ao pacote util.

Classe Checker

A classe Checker pertence ao pacote `verif`. Ela é responsável por verificar se o código *assembly* do programa a ser protegido é condizente com as configurações informadas nos arquivos de configurações. Caso seja, a CFT-tool segue para a próxima etapa do processo de proteção. Caso contrário, um erro é reportado e a execução é encerrada.

Classe Format

A classe Format realiza a formatação do código *assembly* do programa a ser protegido, removendo comentários e espaços desnecessários, preparando o código para ser protegido. Essa classe faz parte do pacote `verif`.

ANEXO B - ARTIGOS

Um artigo, relacionado ao tema da dissertação, é apresentado neste anexo. O artigo, denominado *Configurable Tool to Protect Processors against SEE by Software-based Detection Techniques*, foi aceito para publicação no Latin American Test Workshop (LATW), no ano de 2012.

Configurable Tool to Protect Processors against SEE by Software-based Detection Techniques

Eduardo Chielle, Raul Sérgio Barth, Ângelo Cardoso Lapolli, Fernanda Lima Kastensmidt
Instituto de Informática – PPGC - UFRGS
Porto Alegre, Brazil
{echielle, rsbarth, aclapolli, fglima}@inf.ufrgs.br

Abstract—This paper presents a tool capable of automatically adding fault detection capabilities in software to protect the processors against transient faults. The tool implements a set of configurable software-based detection techniques over the assembly code of an unprotected program. The developed tool has been validated for two distinct processors: MIPS and LEON3. But it can be extended to other architectures and organizations by changing the configuration files. A fault injection campaign was performed and simulation results show high detection rates to both processors and a small increase in area and runtime.

Keywords- *fault tolerance, soft errors, SEU, SET, software techniques*

I. INTRODUCTION

Integrated circuits fabricated with nanometric dimension transistors, operating in high frequency and low voltage supply, are more sensitive to transient faults, known as Single Event Effect (SEE) [1]. SEE is produced by highly energetic particle hits on sensitive circuit regions. When SEE affects a memory element, it is called Single Event Upset (SEU) and it is characterized as a bit-flip in the flip-flop. When this transient effect affects a logic gate of a combinational block, it is called Single Event Transient (SET) and can be perceived as a glitch with variable duration corresponding to the collected charge. SEE is common in circuits operating in radiation environment.

In processors, the effect of SEU and SET faults can be noticed by errors in the program data or errors that change the program flow. In order to protect a processor against SEE faults, the use of fault tolerance techniques capable of detecting and/or correcting these effects is needed. Among the existing fault tolerance techniques, there are hardware-based techniques and software-based techniques. In the hardware-based techniques, the physical system is modified. The circuit is usually duplicate or triplicate and checkers or voters are inserted [2]. Besides, it is possible to use error-correcting code (EDAC) in the memory elements too. According to implemented hardware-based techniques, the system can present variant overheads like reduction in the operating frequency, increase in area and power consumption, besides having a high design and manufacture costs [3].

The software-based techniques do not modify the hardware, allowing to use COTS (Commercial off-the-shelf) processors, which are substantially cheaper than radiation hardened (rad-hard) processors. Only the source code of the program is modified. Techniques which duplicate the variables, like EDDI – Error Detection by Duplicated Instructions [4], or techniques

to protect the control flow, like CFCSS – Control-Flow Checking by Software Signatures [5] or techniques which apply transformation rules over the code [6] are used by modifying the program's source code. These modifications make the software fault tolerant. However, these software-based techniques increase the runtime and the memory occupation of the program. When an error is detected, the processor and the program are restarted or a certain part of the code is recomputed.

The modification of the software-based techniques can be usually done in three different language levels: at high-level, assembly level and machine level languages. One can say that the best choice is applying the techniques at high-level language because it is independent to target processor architecture and organization. However, the compiler optimizes the code, removing the redundancies. It is possible to modify the compiler to do not remove these redundancies, but it makes the tool dependent of the target architecture. The remaining possibilities are the application of the techniques over the assembly or machine code. However, both are linked with the architecture. Consequently, the target architecture must be described to the tool, making the tool, in this way, capable of recognizing the program instructions and applying the selected techniques.

This paper presents a tool capable of protecting processors with diversified architectures against SEE by software-based techniques. The developed tool is configurable and capable of applying a set of software-based fault tolerance techniques that can be specified by the user according with the detection rate and costs in area and runtime. The tool is architecture independent because the information of the target processor architecture and organization is informed in a configuration file. The user can select as well the set of fault tolerant techniques to be implemented.

The paper is organized as follows. Section II presents an overview about the software-based detection techniques and the tools that apply automatically the techniques present in the literature. Section III describes the proposed tool and how it works. A fault injection campaign is presented in section IV. The achieved detection rate of the techniques implemented by the proposed tool to a MIPS processor [7] are compared to the results by using the same techniques applied over the machine code. Results in a LEON3 processor [8] are also presented. In section V, main conclusions and future works are presented.

II. RELATED WORKS

A. Software-based Techniques

There are four main areas in the literature about software-based techniques to protect processors. They are data protection techniques, control-flow protection techniques, hybrids techniques, which aim to protect both data and control-flow, and techniques to protect the memory. The first three are part of this work because they are non intrusive and they can be applied in many abstraction levels.

The data protection techniques aim at protecting against faults that affect the data stored in the memory or in the registers. These techniques duplicate the variables, creating copies and making, over the copy variable, the same operations that are performed over the original variable. Furthermore, instructions to check the program integrity are also inserted in some points of the code. These points depend on the selected technique.

This kind of technique is not designed to detect faults that affect the control-flow, but it is able to detect some errors caused by these faults, because they can make the original variable and the copy have different values, which can be detected by a checker. An example of these techniques is EDDI [4], VARI, VAR2 and VAR3 [9].

The control-flow protection techniques aim at protecting program flow. These techniques divide the code in basic blocks, which are portions of code without branch or jump instructions, being attributed a signature to each basic block. There are three kinds of faults which affect the control-flow: faults changing the program flow from a basic block to another one; faults changing the flow to the beginning of another basic block; and faults changing the program flow to another position of the same basic block. This last type of error is not detected by the current techniques existing in the literature. Examples of these techniques are CCA [10], ECCA [11], CFCSS [5], SIG and BRA [12].

The hybrid techniques are, in general, transformation rules that are applied to the program code and aim at protecting both data and control-flow. An example of this technique is the SWIFT technique [13], which is a join of the data protection technique EDDI and control-flow protection technique CFCSS with some optimizations.

B. Existing Tools

A program described in a high level language goes through several compilation stages until it is transformed into a machine code level. Fig. 1 shows these stages. First, the code is compiled, being optimized in this process, going from a high-level language description to an assembly description. So, the assembler reads the assembly code and generates the machine code.

The application of the software-based techniques can be usually done in three different language levels. The code can be modified in a high level language, like C. In this case, the process is independent of the processor organization and architecture. However, the source code in high level language is subject to optimizations by the compiler, which can remove the redundancies generated by the techniques, making the

protection ineffective. Another possibility is modifying the compiler so as not to remove the redundancies created by the technique. In this case, each compiler targeting each processor in particular must be modified, which is time consuming and it can generate errors that must be verified. The techniques also can be applied over the assembly code, thus, not being affected by the optimizations of the compiler. In this case, it is not necessary to modify the compiler or assembler. Furthermore, the program can be protected in the machine code. The disadvantage in this case is that it is highly connected with the architecture.

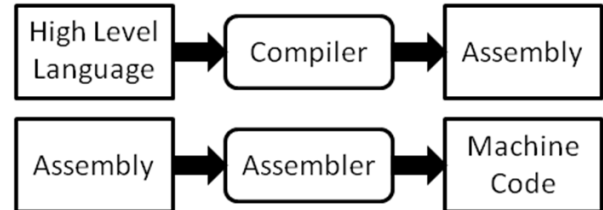


Figure 1. Program's abstraction levels

C2C Translator [14] is a tool that modifies the program's source code in a high level language, C language, before the stages of compiling and assembling. Applying the technique in C makes the tool independent of architecture, however, the compiler optimizes the code, removing the redundancies, which can make the technique do not work.

In reference [15], the compiler OpenIMPACT is modified to apply the SWIFT technique to the Intel Itanium 2 processor. The compiler was modified to not remove the redundancies created by the technique, making it efficient. However, although it gets the benefit of the generality of a high level language, this method is not generic, because for each target processor it is necessary to modify a compiler to that architecture.

The HPCT Suite [9] is a tool that modifies the program's machine code for a MIPS processor. In this level the tool is linked to the processor architecture, and it shows a high efficiency because it modifies the final program, making the technique executed by the processor exactly as it was applied. However, in this level, many calculations are required to correct the addresses of the branches and jump instructions. Furthermore, the tool is not independent of the processor architecture.

III. THE PROPOSED TOOL: CFT

In order to reduce the complexity of modifying the code at the machine level, a solution based on assembly code modifications was developed. The CFT-tool applies over the assembly code, in an automated way, the software-based detection techniques validated by reference [9]. There are some advantages on employing the fault tolerance at assembly level instead of at machine level. For example, in the machine code, the target addresses of the branches and jumps must be corrected. In assembly level, this does not need to be done because the target of branches and jumps are labels and its physical address is defined after the assembling stage. Besides, the branch delay slot consists in reordering the branches and

jumps instructions to gain performance and it is another concern to apply the techniques over the machine code. Furthermore, applying the techniques over the assembly code makes the CFT-tool more portable among different architectures, because the assembler of the target processor takes charge of generating the executable file. For these reasons, the choice was to work over the assembly code, where an existing compiler and assembler can be used without needing any modification.

Fig. 2 presents the stages in which the program goes through until it is protected by the techniques. The code in a high level language is compiled, generating its equivalent assembly code. The CFT-tool reads the configuration about the processor architecture and information about the techniques to be applied over the program code by the SEE Protection Module. The users inform the configuration files containing the processor organization and architecture and the selected software-based techniques. After that, the Verification Module reads the program's assembly code and verifies if the source code matches with the processor architecture informed. Then, the Filler Module generates a new assembly code. Finally, the SEE Protection Module applies the software-based detection techniques selected by the user over the assembly code, generated by the Filler Module, creating a protected assembly code.

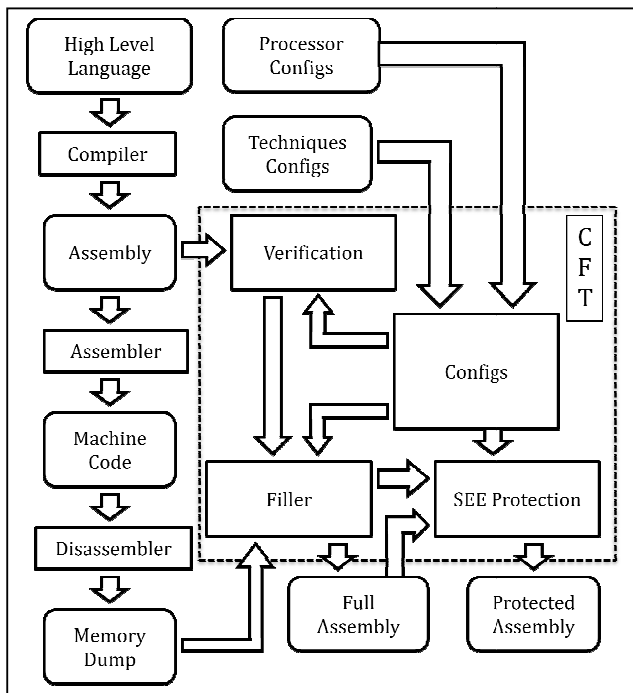


Figure 2. CFT-tool schematics and program stages

A. Configuration Module

As the tool must be independent of the processor architecture, it needs to understand the target processor architecture in some way. For this reason, the processor architecture must be described to the tool by informing features of the processor. This description is made in configuration files, where all required information about the processor is

informed. For example, data about the instructions; which type they are: arithmetic, branches, jumps, loads, stores. Also the format of the instruction must be informed; their mnemonics; which are the source registers and the destiny register; identify the immediate values; and the position of each one of these in the instruction. Data about the registers must be informed, such as name, if it is a special register, if the register is writable, as well others relevant characteristics about the target processor organization.

The tool also needs to know the techniques that must be applied over the code of the program to be protected and which registers must be protected, choosing their priority to get a copy register.

Therefore, the Configuration Module's function is to identify the architecture patterns, the selected techniques and provide the required information to other modules, independently of the target architecture.

B. Verification Module

The function of the Verification Module is to read the program's assembly code generated by the compiler and verifies if the code matches with the information of the specified architecture. The information about the target processor architecture is informed by the Configuration Module. If the program's assembly code matches with the architecture specification, the tool goes to the next stage, the Filler Module, otherwise, an error is reported.

C. Filler Module

In the compiling stage, there are subroutines generated by the compiler. Some of these subroutines are known by the assembler and, for this reason, they are not described in the program's assembly code, but only calls them. So, for the entire program be protected, these subroutines must be in the assembly code. Thereunto, a solution is converting these subroutines from the machine code to assembly code, using memory dump files.

This module aims at verifying if there is any of this kind of subroutine and converts its code to assembly. Firstly, the module search in the assembly code for calls to subroutines and verifies if they are present in the code. If not, the code must be converted. Fig. 3 shows an example of a subroutine called in someplace of the program, but which does not have its code implemented in assembly. The *jal* instruction calls for the *memcpy* subroutine that is not present in the source code.

```

...
main:
...
la $3,$LC1
move $4,$2
move $5,$3
li $6,36
jal memcpy
sw $0,96($fp)
sw $0,100($fp)
...
$L2:
...
$L3:
...
j $31
.end main

```

Figure 3. Call to a subroutine not implemented in the assembly code

```

00000300 <memcpy>:
300: 00a41025      or      v0,a1,a0
304: 30420003      andi   v0,v0,0x3
308: 14400026      bnez   v0,3a4 <memcpy+0xa4>
30c: 00805021      move   t2,a0
310: 00064102      srl    t0,a2,0x4
314: 00a04821      move   t1,a1
318: 30c6000f      andi   a2,a2,0xf
31c: 1100000d      beqz   t0,354 <memcpy+0x54>
320: 00803821      move   a3,a0
324: 8d220000      lw     v0,0(t1)
328: 8d230004      lw     v1,4(t1)
32c: 8d240008      lw     a0,8(t1)
330: 8d25000c      lw     a1,12(t1)
334: 2508ffff      addiu  t0,t0,-1
338: ace20000      sw     v0,0(a3)
33c: ace30004      sw     v1,4(a3)
340: ace40008      sw     a0,8(a3)
344: ace5000c      sw     a1,12(a3)
348: 25290010      addiu  t1,t1,16
34c: 1500fff5      bnez   t0,324 <memcpy+0x24>
350: 24e70010      addiu  a3,a3,16
354: 00064082      srl    t0,a2,0x2
358: 11000007      beqz   t0,378 <memcpy+0x78>
35c: 30c60003      andi   a2,a2,0x3
360: 8d220000      lw     v0,0(t1)
364: 2508ffff      addiu  t0,t0,-1

memcpy:
or $2,$5,$4
andi $2,$2,0x3
move $10,$4
bnez $2,memcpy1
srl $8,$6,0x4
move $9,$5
andi $6,$6,0xf
move $7,$4
beqz $8,memcpy2
memcpy3:
lw $2,0($9)
lw $3,4($9)
lw $4,8($9)
lw $5,12($9)
addiu $8,$8,-1
sw $2,0($7)
sw $3,4($7)
sw $4,8($7)
sw $5,12($7)
addiu $9,$9,16
addiu $7,$7,16
bnez $8,memcpy3
memcpy2:
srl $8,$6,0x2
andi $6,$6,0x3
beqz $8,memcpy4
memcpy5:
lw $2,0($9)
addiu $8,$8,-1

```

Figure 4. Example of a subroutine extracted from a memory dump file (left) and equivalent assembly code generated (right)

After finding the subroutines that are not implemented in the assembly code, the Filler Module searches the subroutines in the memory dump file and creates their equivalent code in assembly and inserts subroutines codes generated in the end of the program's assembly code. To do that, the Filler Module must have information about the architecture, like the specification of the instructions both in assembly and the memory dump format, existence of branch delay slot and the way how the data are organized.

Fig. 4 shows an example of a code in a memory dump file, on the left, and the equivalent assembly code generated by this module, on the right. They are similar but with some important differences. The registers names are not the same. The destination address of the jumps and branches in the memory dump file are physical addresses and in assembly are labels. In some architectures, the branches and jumps are shifted with next instruction to gain performance. For the architecture used as example, these shifts must be undone to keep the program working correctly, but making sure the instructions respect their code block, defined by the labels. This means that no instruction located after a label can be moved to before it and neither an instruction located before a label can be moved to after it. After that, if the architecture admits optimizations, the instructions that are not going to be executed are removed from the code, because the assembler will insert those again if required.

D. SEE Protection Module

The SEE Protection Module is responsible for applying the software-based detection techniques in the program that will be protected. When a detection technique is selected to be applied in the program, the Protection Module performs the transformation in the assembly code, creating a new protected assembly code. The techniques are in the order that the user has informed in the configuration files. All required information is informed in the Configuration Module.

The available techniques are: the ones protecting the data that are called Variables 1 (VAR1), Variables 2 (VAR2) and Variables 3 (VAR3) and the ones to protect branches and control flow effects that are called Branches (BRA) and

Signatures (SIG). The VAR1 technique is based on the transformation rules [15] to protect the data, which say every variable must be duplicated, where every operation over the original variable must be performed over the copy. Instructions to compare the value of the original variables with their copies must be inserted before every instruction that reads the variable. VAR2 is an alternative to the reference [15] data protection technique, based on reference [14] proposal. The only difference between the techniques is the location where the checkers are inserted. In the VAR2 technique, instead of inserting the instructions to check the original and copy variables before it is read, they are inserted after a writing over the variable. The other implemented technique to protect the data is the VAR3 technique. VAR3 is based on the SWIFT technique [13], which also duplicates the variables. In this technique, the checkers are inserted before the variable is read by load, store and branch instructions.

To protect the program flow, the technique SIG was implemented. This technique, like the most of techniques to detect errors affecting the control-flow, divides the source code in basic blocks and attributes signatures to each one. The basic block's signature is attributed to a variable in the beginning of the execution of the basic block. The value is verified at the end of the execution of the basic block to verify if a jump to another basic block did not occur. Besides this technique, there is another called BRA, based on a rule proposed by reference [15] to protect against errors affecting the branch instructions. In this technique, the branches are replicated. Right after where the branch instruction, another branch logically equal to the original, but with the destination addressed to the error treatment subroutine, is inserted. Furthermore, in the branch destination is inserted another branch, but it is logically inverse to the original and the target address is addressed to the error treatment subroutine. Labels and jumps must be inserted to keep the correctness of the program.

IV. VALIDATION RESULTS

A fault injection campaign was performed to validate the CFT-tool. Faults were injected at logical level in MIPS and LEON3 processors, using Modelsim [16]. The location and

time for each injected faults are randomly selected. Only one fault was injected at each execution of the program. Two programs were protected by CFT-tool: a matrix multiplication and a bubble sort. The matrix multiplication has a lot of data processing and a few loops, being ideal to verify the coverage of the techniques in detecting errors affecting data. On the other hand, the bubble sort does not have so much data processing and has a large number of loops, branches and control registers, which brings the faults to affect more the control flow than the matrix multiplication. The programs were protected with four different combinations of the techniques. The first three versions were protected only with data protection technique, VAR1, VAR2 and VAR3, and the last one was a combination of the data protection technique VAR3 with two control-flow protection techniques, BRA and SIG.

The time consumed by the CFT-tool to protect the applications is negligible. The time depends on the code size and the selected techniques. CFT takes less than 5s to protect the case study codes for all techniques and both processors.

Aiming at verifying the efficiency of the software-based detection techniques, only the faults that caused errors were taken into account. The faults masked by the logic of the processor were ignored.

After applying the techniques over the programs, a fault injection campaign was performed where 10,000 faults were injected in each protected program. Table 1 shows the detection rate to the matrix multiplication protected with the different combinations of the techniques to MIPS processor. The programs were protected by CFT-tool and HPCT Suite [9], which was adopted as a basis to comparison as a way to validate the correct application of the techniques by the CFT-tool.

Variables techniques increase the runtime, but also have good error coverage. VAR3 shows the best cost-benefit. The combined techniques presents a small increase in the detection rate with a small increase in the runtime compared to the ones only with variables techniques. The detection rates of the techniques applied over the assembly code by the CFT-tool are quite similar with the results obtained to machine code. They are within the margin of error.

TABLE I. MATRIX MULTIPLICATION (MIPS)

Technique	Exec. Time (μ s)	Memory Occupation (bytes)	Number of Errors	Detection Rate (%)
Unprotected	162	1308	-	-
VAR1 ¹	398 (2.46x)	3220 (2.46x)	2619	73.3
VAR1 ²	332 (2.05x)	3296 (2.52x)	2733	74.7
VAR2 ¹	421 (2.60x)	3452 (2.64x)	2543	71.6
VAR2 ²	365 (2.25x)	3260 (2.49x)	2548	72.5
VAR3 ¹	265 (1.64x)	2524 (1.93x)	2502	69.9
VAR3 ²	290 (1.79x)	2808 (2.15x)	2291	68.0
SIG+BRA+VAR3 ¹	300 (1.85x)	3432 (2.62x)	2708	70.8
SIG+BRA+VAR3 ²	319 (1.97x)	3776 (2.89x)	2462	69.2

¹ Codes protected by CFT-tool
² Codes protected by HPCT Suite

Table 2 presents the results of a bubble sort running in a MIPS processor. The results of the same program with the techniques applied over the machine code are presented below the results obtained by the program protected with the CFT-tool.

TABLE II. BUBBLE SORT (MIPS)

Technique	Exec. Time (μ s)	Memory Occupation (bytes)	Number of Errors	Detection Rate (%)
Unprotected	196	1156	-	-
VAR1 ¹	481 (2.45x)	2744 (2.37x)	2206	69.9
VAR1 ²	396 (2.02x)	2932 (2.54x)	2294	71.2
VAR2 ¹	515 (2.63x)	2996 (2.59x)	2090	66.9
VAR2 ²	448 (2.29x)	2912 (2.52x)	2116	67.6
VAR3 ¹	367 (1.87x)	2220 (1.92x)	2048	67.4
VAR3 ²	343 (1.75x)	2520 (2.18x)	1993	67.3
SIG+BRA+VAR3 ¹	430 (2.19x)	3092 (2.67x)	2133	70.0
SIG+BRA+VAR3 ²	408 (2.08x)	3492 (3.02x)	2152	68.2

¹ Codes protected by CFT-tool
² Codes protected by HPCT Suite

The runtime overhead increases in the bubble sort compared to the matrix multiplication. This happens because code blocks inside the loops have a relative bigger increase compared to a code block of the matrix multiplication. The detection rate is a little smaller because of more faults affecting the control-flow. Furthermore, the variables techniques also protect against faults affecting the program flow by duplicating the control registers.

The difference in the runtime overheads to assembly and machine is because of the branch and jumps treatments. From assembly to machine code, the assembler shifts all branches and jumps one position up, because this processor always executes an instruction after the branches and jumps. Fig. 5 shows an example that explains the difference. The correctness of the machine code is maintained when the program runs without errors, but when an error is detected another one can be created by an instruction that should be not executed. This feature can interfere in the error treatment.

```

01b8: 17ce0222 bne $30,$14,0xa44 | 01c4: 17cd0235 bne $30,$13,0xa9c
01bc: 14010221 bne $0,$1,0xa44 | 01c8: 00000000 nop
01c0: afc00060 sw $0,96($30) | 01cc: 141a0233 bne $0,$26,0xa9c
01c4: adc10260 sw $1,608($14) | 01d0: 00000000 nop
01c8: 17ce021e bne $30,$14,0xa44 | 01d4: afc00060 sw $0,96($30)
01cc: 1401021d bne $0,$1,0xa44 | 01d8: 17cd0230 bne $30,$13,0xa9c
01d0: afc00064 sw $0,100($30) | 01dc: adba0100 sw $26,256($13)
01d4: adc10264 sw $1,612($14) | 01e0: 141a022e bne $0,$26,0xa9c
| 01e4: 00000000 nop
| 01e8: afc00064 sw $0,100($30)
| 01ec: 17cd022b bne $30,$13,0xa9c
| 01f0: adba0104 sw $26,260($13)

```

Figure 5. Machine code (left) and assembly code (right)

The techniques were also applied to a matrix multiplication and a bubble sort running in a LEON3 processor to validate that the CFT-tool is configurable and architecture independent. Table 3 shows the same matrix multiplication with the same software-based techniques applied to a LEON3 processor. LEON3 is a processor with a SPARC V8 architecture that implements the registers in windows, where the registers are divided into local, global, input and output registers.

TABLE III. MATRIX MULTIPLICATION (LEON3)

Technique	Exec. Time (μ s)	Memory Occupation (bytes)	Number of Errors	Detection Rate (%)
Unprotected	99	984	-	-
VAR1	295 (2.98x)	2416 (2.46x)	587	55.4
VAR2	269 (2.72x)	2248 (2.28x)	600	54.8
VAR3	200 (2.02x)	1828 (1.86x)	586	36.7
SIG+BRA+VAR3	199 (2.01x)	2168 (2.20x)	569	31.3

The lower detection rate to the LEON3 processor can be explained because the LEON3 has no sufficient available registers to be used when variables needed to be duplicated. There are only five available registers to the variables techniques and four to the variable technique combined with the control-flow techniques, because the SIG technique requires one register, which can explain the lower detection rate to the combined techniques. Therefore, a lower detection rate is expected. But, despite the lower detection rate due to the lack of register, the same characteristics about the detection happen here, showing that the techniques are also efficient to this processor.

Table 4 shows the detection rate of a bubble sort program, protected with the different combination of the techniques running over a LEON3 processor.

TABLE IV. BUBBLE SORT (LEON3)

Technique	Exec. Time (μ s)	Memory Occupation (bytes)	Number of Errors	Detection Rate (%)
Unprotected	113	588	-	-
VAR1	343 (3.04x)	1660 (2.82x)	514	55.8
VAR2	315 (2.79x)	1540 (2.62x)	563	60.7
VAR3	260 (2.30x)	1264 (2.15x)	524	45.6
SIG+BRA+VAR3	248 (2.19x)	1636 (2.78x)	566	49.6

Also in these simulations we can see the same characteristics to MIPS processor about the detection rate of the techniques and a lower detection rate as in the matrix multiplication running in a LEON3 processor because there are no sufficient free registers to duplicate all used registers. In the bubble sort, there are only two available registers. However, the protected registers have a significant impact in the detection rate. The most read registers were the ones selected to be protected.

V. CONCLUSION AND FUTURE WORKS

In this paper, we presented the CFT-tool, a tool capable of protecting processors against SEU and SET faults. The CFT-tool implements automatically a set of configurable software-based detection techniques over the assembly code of an unprotected program. The tool works independently of the processor architecture. The tool is configured to the target processor by describing its architecture. A set of programs was protected by the CFT-tool to different processors: MIPS and

LEON3. A fault injection campaign was performed and simulation results show high detection rates to MIPS processor, around 70%; and a regular detection rate to LEON3 processor, around 50%. As future work, we intend to identify the best way to select the registers, when more registers than the available in the architecture are needed. Moreover, we will verify the CFT-tool in other architectures, such as ARM and PowerPC, to test the efficiency of the tool and of the techniques.

REFERENCES

- [1] P.E. Dodd, M.R. Shaneyfelt, J.R. Schwank and J.A. Felix, "Current and Future Challenges in Radiation Effects on CMOS Electronics", IEEE Transactions on Nuclear Science, vol. 57, n. 4, august 2010, pp. 1747-1763.
- [2] D. Pradhan, "Fault-tolerant computer system design", Upper Saddle River, USA : Prentice-Hall, 1995.
- [3] S.C. Asensi, A.M. Alvarez, F.R. Calle, F.R. Palomo, H.G. Miranda and M.A. Aguirre, "A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems", IEEE Transactions on Nuclear Science, vol. 58, n. 3, june 2011, pp. 1059-1065.
- [4] N. Oh, S. Mitra and E. McCluskey, "ED4I: error detection by diverse data and duplicated instructions", IEEE Transactions on Computers, vol. 51, n. 2, 2002, p. 180-199.
- [5] N. Oh, P.P. Shirvani, and McCluskey, "Control-flow checking by software signatures", IEEE Trans. on Reliability, vol. 51, Issue 1, March 2002, pp. 111-122
- [6] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M.S. Reorda and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors", IEEE Trans. on Nuclear Science, vol. 47, n. 6 (part 3), Dec 2000, p. 2231-2236.
- [7] L.M.O.S.S. Hangout and S. Jan. The minimips project, available online at <http://www.opencores.org/projects.cgi/web/minimips/overview>, 2010.
- [8] Aeroflex Gaisler, LEON3, available online at <http://www.gaisler.com>, 2010.
- [9] J.R. Azambuja, A. Lapolli, L. Rosa, F.L. Kastensmidt, "Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors", IEEE Latin American Symposium on Circuits and Systems, 2010.
- [10] L.D. Mcfearin and V.S.S. Nair, "Control-flow checking using assertions", Proceedings of the IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05), Urbana-Champaign, IL, USA, September 1995.
- [11] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy and J.A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection", IEEE Trans. on Parallel and Distributed Systems, vol. 10, Issue 6, June 1999, pp. 627-641.
- [12] J.R. Azambuja, A. Lapolli, L. Rosa, F.L. Kastensmidt, "Non-Intrusive Hybrid Signature-Based Technique to Detect SEU and SET Faults in Microprocessors". Proceedings of the European Conference on Radiation and Its Effects on Components and Systems, 2010.
- [13] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan and D.I. August, "SWIFT: software implemented fault tolerance", Proceedings of the Symposium on Code Generation and Optimization, 2005, p. 243-254.
- [14] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results", Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2003.
- [15] M. Rebaudengo, M.S. Reorda, M. Torchiano and M. Violante, "Soft-error detection through software fault-tolerance techniques". Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pages 210, 218, 1999.
- [16] Mentor Graphics, <http://www.model.com/content/modelsim-support>, 2010.