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

DIEGO GONÇALVES RODRIGUES

Detecção e proteção de blocos básicos suscetíveis através da análise sistemática de single bit-flip

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

Orientador: Prof. Dr. Álvaro Freitas Moreira
Co-orientador: Prof. Dr. Luigi Carro

Porto Alegre
2015

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rodrigues, Diego Gonçalves

Detecção e proteção de blocos básicos suscetíveis através da análise sistemática de single bit-flip / Diego Gonçalves Rodrigues. – Porto Alegre: PPGC da UFRGS, 2015.

78 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Orientador: Álvaro Freitas Moreira; Coorientador: Luigi Carro.

1. Sistemas Embarcados. 2. Falhas Transientes. 3. Tolerância à Falhas. 4. Verificação de Assinaturas. I. Moreira, Álvaro Freitas. II. Carro, Luigi. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

RESUMO

Partículas radioativas, ao atingirem o hardware dos sistemas computacionais, podem resultar em comportamentos inesperados durante a execução de um software. Tais comportamentos inesperados podem persistir por toda a vida útil do sistema ou podem ter uma duração limitada. Nesse último caso, temos o que chamamos de falhas transientes.

Falhas transientes podem fazer com que as instruções do programa executem em uma sequência incorreta, o que chamamos de erros de fluxo de controle *(Control-flow errors - CFEs)*. Estudos mostram que entre 33% e 77% das falhas transientes que afetam o hardware se manifestam como erros de fluxo de controle, dependendo do tipo do processador. Se o sistema não realizar nenhuma verificação em tempo de execução, um erro de fluxo de controle pode não ser detectado, o que pode resultar em uma execução incorreta do programa.

Sistemas projetados para aplicações de baixo custo voltados para sistemas embarcados, onde os custos e desempenho são os fatores principais, utilizam técnicas baseadas em software para aumentar a confiabilidade do sistema. As técnicas baseadas em software para detecção de CFEs são conhecidas como *signature monitoring* ou *signature checking*. Essas técnicas introduzem código extra em todos os blocos básicos do programa com a finalidade de detectar os CFEs. Esse código extra implica em *overhead*, que pode ter uma grande variação dependendo da técnica utilizada.

Na tentativa de minimizar o *overhead* imposto pelas técnicas de detecção de CFEs, neste trabalho foi desenvolvida a técnica de detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*. O objetivo da técnica é detectar os blocos básicos suscetíveis do programa através da análise sistemática de *single bit-flip* e proteger apenas esses blocos básicos.

A técnica foi avaliada em termos de sua taxa de cobertura de falhas e desempenho. Para avaliar a taxa de cobertura falhas foram realizadas várias campanhas de injeção de falhas nos programas da suíte de *benchmarks Mibench*. A avaliação de desempenho foi feita com base na quantidade de instruções de máquina executadas pelos *benchmarks*, comparando quantidade de instruções antes e depois da utilização da técnica detecção e proteção de blocos básicos suscetíveis.

Os resultados dos experimentos mostram que é possível reduzir em até 27,93% a quantidade de blocos básicos protegidos e ao mesmo tempo manter uma alta taxa de cobertura de falhas. Porém, em termos de desempenho, o ganho não ficou na mesma proporção da quantidade de blocos básicos não protegidos, ficando abaixo do esperado.

Palavras-chave: Sistemas Embarcados, Falhas Transientes, Tolerância à Falhas, Verificação de Assinaturas.

Detection and protection of susceptible basic blocks through systematic bit-flip analysis

ABSTRACT

Radioactive particles hitting the hardware of computer systems may result in unexpected behavior during software execution. Such unexpected behavior may persist for the lifetime of the system or may have a limited duration. In the latter case, we have what is called a transient fault.

Transient faults may cause the program instructions to execute in an incorrect sequence. This incorrect sequence is called a control flow error (CFE). Research show that between 33% to 77% of transient faults manifest themselves as CFEs, depending on the type of the processor. If the system does not perform any verification at runtime, a control flow error may be not detected, which can result in incorrect program execution.

Systems projected to low-cost embedded applications, where cost and performance are the main factors, use software based techniques to improve system reliability. Software based techniques to detect CFEs are known as signature monitoring or signature checking. These techniques insert extra code in each basic block of the program in order to detect CFEs. This extra code add an undesirable overhead in the program, which can have large variation depending on the technique used.

In the attempt to minimize the overhead added by CFEs detection techniques, this work developed a technique of detection and protection of susceptible basic blocks through systematic bit-flip analysis. The purpose of this technique is to detect the susceptible basic blocks of the program through the systematic bit-flip analysis and to protect only these basic blocks.

The technique was evaluated based on its fault coverage rate and performance. To evaluate the fault coverage rate a fault injection campaing was performed in the programs of the Mibench benchmark suite. The performance evaluation was based in the number of instruction executed by each benchmark, comparing the number of instructions before and after the use of the proposed technique.

The experimental results show that is possible to reduce up to 27,93% the amount of protected basic blocks, while keeping a high faults coverage hate. However, in terms of performance, the gain was not in the same proportion, being lower than expected.

Keywords: Embedded Systems, Transient faults, Fault Tolerance, Signature Checking.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABFT	Algorithm Based Fault Tolerance
ACFC	Assertions for Control Flow Checking
ACS	Abstract Control Signatures
AST	Abstract Syntax Tree
CEDA	Control-Flow Error Detection Using Assertions
CCA	Control Flow Checking using Assertions
CFCSS	Control Flow Checking by Software Signature
CFE	Control-Flow Error
CFG	Control Flow Graph
CISC	Complex instruction set computing
COTS	Commercial-off-the-shelf
CRC	Cyclic Redundancy Check
ECCA	Enhanced Control-flow Checking Using Assertion
FCFC	Flexible Control Flow Check
FFT	Fast Fourier Transform
IDE	Integrated Development Environment
GDB	GNU Debugger
LLC	LLVM static compiler
LLVM	Low Level Virtual Machine
LLVM-IR	LLVM intermediate representation
NOP	No Operation
PC	Program Counter
RISC	Reduced instruction set computing
SBL	Susceptible Basic blocks List
SIHFT	Software Implemented Hardware Fault Tolerance
YACCA	Yet Another Control-flow Checking Using Assertions

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto e Motivação

Sistemas computacionais estão constantemente expostos a eventos externos que podem comprometer o seu funcionamento, tal como a radiação. Partículas radioativas, ao atingirem o hardware dos sistemas computacionais podem, por exemplo, levar a falhas permanentes, como a inutilização de componentes devido a um alto aquecimento, e podem, também, resultar em comportamentos inesperados durante a execução de um software, por exemplo. Tais comportamentos inesperados podem persistir por toda a vida útil do sistema ou podem ter uma duração limitada. Nesse último caso, temos o que chamamos de falhas transientes.

Falhas transientes podem causar uma sequência incorreta na execução das instruções do programa, o que chamamos de erros de fluxo de controle *(Control-flow errors - CFEs)*. Um erro de fluxo de controle ocorre quando a sequência de instruções executadas pelo processador sob o efeito de uma falha é diferente da sequência de instruções livre de falhas. Estudos mostram que entre 33% e 77% das falhas transientes que afetam o hardware se manifestam como erros de fluxo de controle, dependendo do tipo do processador (OHLSSON; RIMEN; GUNNEFLO, 1992) e (SCHUETTE; SHEN, 1987). Se o sistema não fizer verificações em tempo de execução, um erro de fluxo de controle pode não ser detectado, o que pode resultar em uma sequência incorreta na execução do programa.

Para aplicações críticas como em sistemas bancários, centrais nucleares ou sistemas de controle aéreo, o objetivo principal é obter a maior confiabilidade possível, sendo assim, os custos para o desenvolvimento de *chips* que atendam a essa necessidade não são considerados como fator de impedimento. Esses sistemas, portanto, geralmente utilizam técnicas de tolerância a falhas baseadas em hardware. Por outro lado, em processadores projetados para aplicações de baixo custo voltados para sistemas embarcados *(commercial-off-the-shelf (COTS))*, onde os custos e desempenho são os fatores principais, técnicas baseadas em software para aumentar a confiabilidade do sistema são as mais indicadas. Essas técnicas são conhecidas como *Software Implemented Hardware Fault Tolerance* - SIHFT, ou ainda, *Algorithm Based Fault Tolerance* - ABFT. (GOLOUBEVA; REBAUDENGO; REORDA; VIOLANTE, 2006)

Detecção de erros de fluxo de controle tem sido um tópico muito estudado ao longo dos últimos anos, e diversas técnicas baseadas em software foram criadas para detectar esse tipo de erro. Essas técnicas geralmente consistem na manutenção e atualização, em tempo de execução, de variáveis que refletem o fluxo de controle do programa. Através do monitoramento dessas variáveis é possível detectar violações no fluxo de controle do programa. Dentre essas técnicas, conhecidas como *signature monitoring* ou *signature checking*, podemos destacar: *Enhanced Control-flow Checking Using Assertions (ECCA)*

(ALKHALIFA; NAIR; KRISHNAMURTHY; ABRAHAM, 1999), *Control Flow Checking by Software Signatures (CFCSS)* (OH; SHIRVANI; MCCLUSKEY, 2002), *Assertions for Control Flow Checking (ACFC)* (VENKATASUBRAMANIAN; HAYES; MURRAY, 2003), *Yet Another Control-flow Checking Using Assertions (YACCA)* (GOLOUBEVA; REBAUDENGO; REORDA; VIOLANTE, 2003) e *Control-Flow Error Detection Using Assertions (CEDA)* (VEMU; ABRAHAM, 2011).

Porém a adoção de uma técnica baseada em *signature checking* implica na introdução de código adicional específico ao programa. Esse código consiste de instruções para o monitoramento do fluxo de controle e para detecção de qualquer violação que ocorra no fluxo de controle. O *overhead* introduzido pode ter uma grande variação dependendo da técnica utilizada. Em (VEMU; ABRAHAM, 2011), por exemplo, vemos que, em termos de memória, o *overhead* observado teve uma variação de 25% a 60% na técnica CFCSS, de 35% a 60% na técnica CEDA e de 60% a 140% na técnica YACCA. Em termos de desempenho, o *overhead* variou de 4,45% a 57,7% na técnica CFCSS, de 33,9% a 84,32% na técnica CEDA e de 3,15% a 57,8% na técnica YACCA.

1.2 Objetivos e resultados alcançados

Como veremos adiante, as técnicas para detecção de erros de fluxo de controle introduzem, em tempo de compilação, código extra em todos os blocos básicos do programa. Esse código extra em todos os blocos básicos é responsável pela detecção dos CFEs e também é responsável pelo *overhead* associados às técnicas de detecção de CFEs.

Neste trabalho desenvolvemos uma técnica que tem por objetivo reduzir o *overhead* associado às técnicas de detecção de CFEs e ao mesmo tempo garantir que a taxa de detecção permaneça inalterada.

A ideia da técnica consiste em detectar e proteger apenas os blocos básicos do programa que podem ser destino de um erro de fluxo de controle, os chamados blocos básicos suscetíveis.

Seguindo nosso modelo de falhas - que assume a ocorrência de apenas um bit-flip no operando das instruções de desvio do programa - a partir de um programa totalmente protegido pela técnica FCFC (*Flexible Control Flow Check*) é feita uma análise sistemática de *single bit-flip* nas operações de desvio deste programa, com o objetivo de identificar os blocos básicos suscetíveis a erros de fluxo de controles.

Após identificar os blocos básicos suscetíveis, dois refinamentos são feitos: (detalhes no Capítulo 5): a) No primeiro refinamento são removidos os blocos básicos suscetíveis afetados por CFEs inter-blocos; b) já no segundo refinamento são removidos os blocos básicos suscetíveis que são destino somente de desvios errados (mas presentes no grafo de fluxo de controle do programa).

De posse da lista dos blocos básicos suscetíveis, uma nova versão do programa é gerada, sendo que nessa versão apenas esses blocos básicos são protegidos, o que, por consequência, possibilitaria a diminuição do *overhead*.

A técnica para detecção de erros de fluxo de controle utilizada, FCFC, é uma técnica desenvolvida ao longo deste trabalho que foi adaptada a partir da técnica CFCSS. Ela foi implementada como um passo de transformação para o compilador LLVM (*Low Level Virtual Machine*) (LATTNER, 2002). O LLVM é uma infraestrutura de compilação *open source* que provê diversas transformações e análises de programas, além de ser composto por diversos módulos e funcionalidades que permitem a criação de novas transformações.

Os experimentos para avaliar a taxa de detecção de CFEs foram realizados através um

injetor de falhas também desenvolvido ao longo deste trabalho.

Com a técnica para detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*, no primeiro refinamento, foi possível reduzir entre 1,56% e 24,02% o número de blocos básicos protegidos pela técnica de detecção de CFEs. Já no segundo refinamento, foi possível reduzir entre 4,76% e 27,93% o número de blocos básicos protegidos.

Porém, a diminuição do *overhead* não aconteceu na mesma proporção da diminuição dos blocos básicos protegidos. Na realidade, não foi possível demonstra a diminuição do *overhead* quando comparamos uma versão do programa onde todos os blocos básicos foram protegidos com outra versão onde apenas os blocos básicos suscetíveis foram protegidos.

1.2.1 Contribuições

Abaixo segue um resumo com as principais contribuições e implementações resultantes desse trabalho:

- Implementação em LLVM das técnicas *Control Flow Checking by Software Signatures* (CFCSS) (OH; SHIRVANI; MCCLUSKEY, 2002), *Control-Flow Error Detection Using Assertions* (CEDA) (VEMU; ABRAHAM, 2011). Todos os fontes encontram-se disponíveis no seguinte endereço eletrônico: `https://github.com/diegogrodrigues/llvm`. Essas duas técnicas foram implementadas para fins de comparação com a técnica introduzida por este trabalho.

- Criação e implementação de uma nova técnica de para detecção de erros de fluxo de controle, FCFC (*Flexible Control Flow Check*), em conjunto com a estudante de intercâmbio Ghazaleh Nazarian. Os fontes também se encontram no endereço eletrônico: `https://github.com/diegogrodrigues/llvm`.

- Implementação de um injetor de falhas baseado em software, voltado para injeção de falhas que geram erros de fluxo de controle. O injetor de falhas encontra-se disponível no seguinte endereço eletrônico: `http://sourceforge.net/projects/faultsinjectionframework/`.

- Criação e implementação da técnica de detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*, que tem por objetivo reduzir o *overhed* imposto pelas técnicas de detecção de erros de fluxo de controle.

1.3 Organização do texto

Este trabalho está organizado da seguinte forma: o Capítulo 2 apresenta a definição de erros de fluxo de controle, o modelo de falhas utilizado neste trabalho e a metodologia dos experimentos para avaliação da taxa de cobertura dos *benchmarks* utilizados ao longo do trabalho.

No Capítulo 3 são apresentadas as principais características das técnicas para detecção de erros de fluxo de controle por software. O capítulo também apresenta os detalhes de duas técnicas de detecção de erros de fluxo de controle implementadas neste trabalho: *Control Flow Checking by Software Signatures* (CFCSS) (OH; SHIRVANI; MCCLUSKEY, 2002) e *Control-Flow Error Detection Using Assertions* (CEDA) (VEMU; ABRAHAM, 2011).

No Capítulo 4 é apresentada a técnica *Flexible Control Flow Check* (FCFC), uma técnica desenvolvida a partir da combinação da técnica CFCSS com o método de atualização local de assinaturas.

No Capítulo 5 é descrita a técnica de detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*. Nesse capítulo são apresentadas as etapas da análise e proteção dos blocos básicos suscetíveis. É dada a definição de blocos básicos suscetíveis. É apresentada a análise sistemática de *single bit-flip* e os refinamentos feitos em cima da lista de blocos básicos suscetíveis. É descrito como os programas são transformados de forma a proteger apenas os blocos básicos suscetíveis. Finalmente, é feita a avaliação da taxa de cobertura dos *benchmarks* utilizados neste trabalho.

No Capítulo 6 é apresentada a avaliação de desempenho dos *benchmarks* utilizados neste trabalho onde apenas os blocos básicos suscetíveis foram protegidos com a técnica FCFC, com base na técnica de detecção e e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*.

Por fim, no Capítulo 7 são apresentadas as conclusões desta pesquisa e trabalhos futuros.

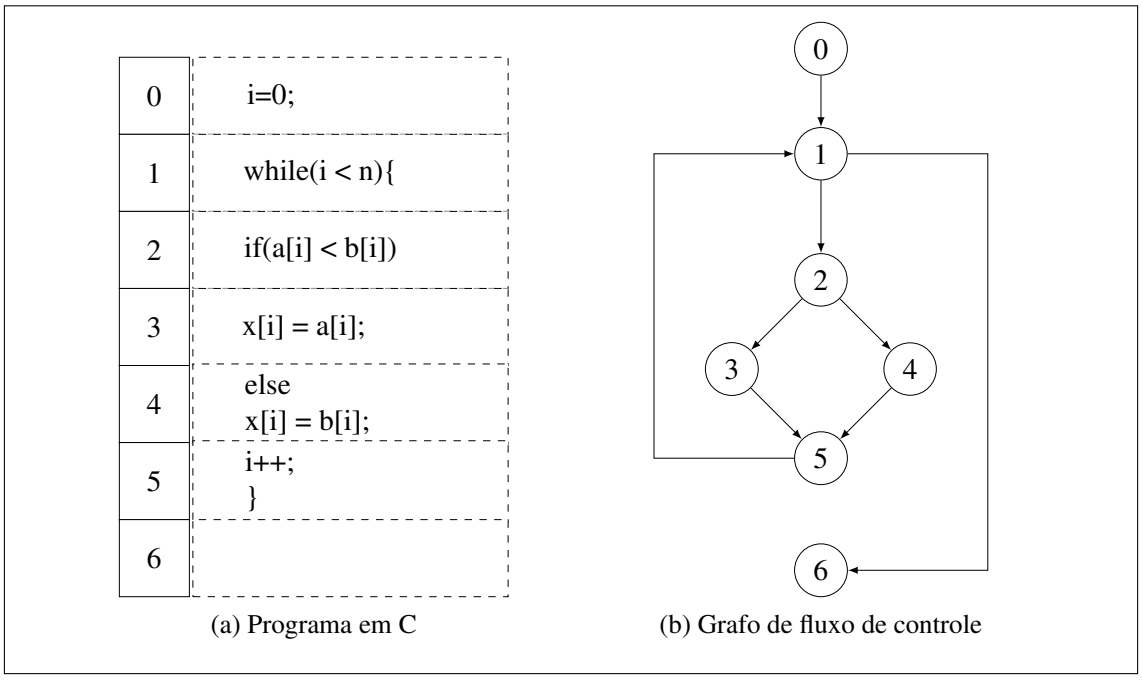
2 ERROS DE FLUXO DE CONTROLE E MODELO DE FALHAS

Este capítulo inicia apresentando a definição de erros de fluxo de controle. Posteriormente, na Seção 2.2, é apresentado o modelo de falhas utilizado ao longo do trabalho e a metodologia utilizada nos experimentos para simular erros de fluxo de controle.

2.1 Erros de fluxo de controle

O **fluxo de controle** de um programa pode ser representado por um grafo de fluxo de controle (*Control Flow Graph* - CFG). Em um CFG, os vértices representam os blocos básicos de um programa, e as arestas conectando blocos básicos representam a possibilidade de transferência de controle de um bloco básico para outro. Um **bloco básico** é uma sequência de instruções onde a execução inicia na primeira instrução e termina na última, sem a presença de desvios a não ser na última instrução. A Figura 2.1 ilustra o conceito de fluxo de controle e blocos básicos, onde podemos ver o trecho de um programa e seu CFG correspondente.

Figura 2.1: Um programa e seu grafo de fluxo de controle



(a) Programa em C

(b) Grafo de fluxo de controle

A obtenção do conjunto de blocos básicos predecessores e o conjunto de blocos básicos sucessores é importante, pois com base nessas informações as técnicas de tolerância a falhas são capazes de verificar se o fluxo de execução do sistema está percorrendo o caminho esperado no grafo.

Um bloco básico A pertence ao conjunto de predecessores de um bloco básico B se, e somente se, houver uma aresta partindo de A com destino B. De forma similar, um bloco básico B é considerado sucessor de A se for o destino de uma aresta com origem em A. No CFG da Figura 2.1, usando como exemplo o bloco básico 5, os seus predecessores são os blocos básicos 3 e 4, enquanto seu sucessor é o bloco básico 1.

Um **erro de fluxo de controle** - *Control Flow Error* (CFE) - ocorre quando observamos um desvio não esperado na execução do programa. CFEs podem ser classificados como **errados** ou **ilegais**:

- **CFEs errados**: também chamados de *desvios errados*, acontecem quando, devido a uma falha, ao invés de executar o desvio de um bloco básico A para um bloco básico B, sucessor de A, executa um desvio de A para o bloco básico C, também sucessor de A, desvio esse não esperado para esse momento da execução. Na Figura 2.2 a aresta **A** ilustra um exemplo deste tipo de erro. Um *desvio errado* acontece se o controle for desviado do bloco básico 1 para o bloco básico 6, quando o desvio esperado no momento da execução é do bloco básico 1 para o bloco básico 2.

- **CFEs ilegais**: ocorrem quando um desvio executado não existe no CFG. Podemos classificar os CFEs ilegais em três diferentes tipos:

    1. Quando ocorre um desvio inexistente no CFG entre diferentes blocos básicos do programa. A aresta **B** da Figura 2.2 ilustra este tipo de CFE, chamado de CFE inter-bloco.
    2. Quando ocorre um desvio para fora do CFG original do programa. A aresta **C** da Figura 2.2 ilustra este tipo de CFE.
    3. Quando ocorre um desvio intra-bloco, ou seja, quando os bloco básico de origem e destino do desvio ilegal é o mesmo. A aresta **D** da Figura 2.2 ilustra este tipo de CFE.

As técnicas para detecção de erros de fluxo de controle reportadas na literatura detectam apenas um subconjunto de CFEs ilegais inter-blocos.

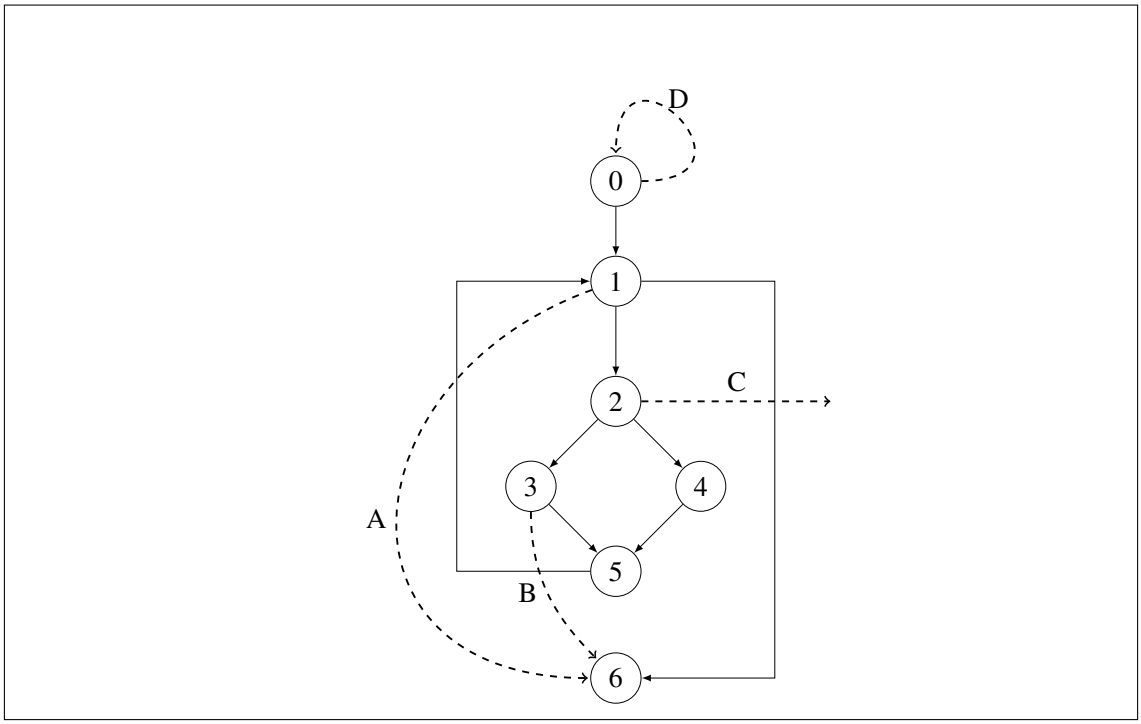
2.2 Modelo de falhas para simular CFEs

Ao implementar uma técnica que se propõe a detectar erros de fluxo de controle é necessário avaliar a eficácia desta técnica. Para tanto se faz necessário um injetor de falhas. O injetor de falhas nada mais é do que uma ferramenta que permite simular falhas no nosso software ou hardware.

Ao injetar uma falha, o injetor assume a existência de um modelo de falhas e a eficácia da técnica deve ser avaliada em relação a esse modelo. Em trabalhos passados como (VEMU; ABRAHAM, 2011) e (OH; SHIRVANI; MCCLUSKEY, 2002), é utilizado o seguinte modelo de falhas para injetar CFEs no programa:

1. *Criação de desvio*, onde o valor do contador do programa (*program counter* - PC) é corrompido,

Figura 2.2: Tipos de erros de fluxo de controle



2. *Remoção de desvio*, no qual uma instrução de desvio é corrompida e resulta em outra instrução que não é de desvio, e

3. *Alteração do operando de desvio*, onde o campo de destino de um desvio é corrompido.

Esses trabalhos assumem também a possibilidade de ocorrência de um único *bit-flip* na execução do programa (*single bit-flip*).

Criação de desvio pode ocorrer se uma instrução que não é de desvio se transforma em um desvio, e remoção de desvio pode ocorrer se um desvio se transforma em uma operação que não é de desvio. Porém, a probabilidade de se transformar o *opcode* de uma instrução que não é de desvio em um *opcode* de uma instrução de desvio e vice-versa, devido a um único *bit-flip* é extremamente baixa, além de depender da codificação dos *opcode* e do conjunto de instruções da arquitetura (No Apêndice C apresentamos um breve estudo sobre esse assunto). Todavia, é importante observar que uma criação de um desvio pode ocorrer devido a um *bit-flip* no *program counter*. Entretanto, a probabilidade de um erro de fluxo de controle dar-se em decorrência disto é relativamente baixa pois trata-se de um circuito muito pequeno se comparado com o restante do processador. Por estas razões – e também por recentes trabalhos que utilizam o mesmo modelo de falhas (KHUDIA; MAHLKE, 2013) - consideramos que os modelos de *criação de desvio* e *remoção de desvio* não são realísticos.

Por esse motivo, o modelo de falhas adotado nesse trabalho irá assumir *single bit-flip* e erros de fluxo de controle causados somente por *alteração do operando de instruções de desvio*. O universo de falhas injetadas nesse trabalho será constituído somente por CFEs causados por single bit-flip no operando de instruções de desvio. Fazem parte desse universo CFEs errados (mas legais, por estarem previsto no CFG do programa) e CFEs

ilegais (não previstos no CFG do programa). A técnica desenvolvida nesse trabalho se propões a detectar um subconjunto dos CFEs ilegais, os CFEs ilegais inter-blocos.

2.3 Metodologia dos experimentos para simular CFEs

Cada experimento realizado ao longo deste trabalho consistiu na realização de uma campanha de injeção de 1.000 falhas em diferentes versões dos programas da suíte de *benchmarks* Mibench (GUTHAUS; RINGENBERG; ERNST; AUSTIN; MUDGE ; BROWN, 2001). Todos os experimentos utilizaram o modelo de falhas *alteração do operando de desvio*.

Outro ponto importante diz respeito à quantidade de vezes que cada instrução é executada. A probabilidade de uma falha ocorrer no operando de um desvio que é executado mil vezes é bem maior do que uma falha ocorrer no operando de um desvio que é executado apenas uma vez. Em trabalho anteriores, por exemplo, (VEMU; ABRAHAM, 2011), esse fato não é levado em consideração. Portanto, o injetor de falhas deste trabalho leva em consideração o número de vezes que cada instrução foi executada para definir a probabilidade de injetar, ou não, uma falha em uma determinada instrução.

As falhas foram injetadas diretamente sobre o código binário de cada programa, com o auxílio do *framework* de depuração GDB *(Gnu Debugger)*. Devido a dificuldade de se encontrar uma ferramenta pronta para a injeção de falhas ao longo deste trabalho também foi desenvolvido um injetor de falhas cuja descrição encontra-se no *Apêndice B*.

Depois de injetadas as falhas em programas protegidos pela técnica os comportamentos observados são categorizados da seguinte forma:

- *Exited correct (EC)* - O programa terminou sua execução e o CFE gerado não foi detectado, mas, mesmo assim, gerou uma saída correta.
- *Exited wrong (EW)* - O programa terminou sua execução e o CFE gerado não foi detectado e gerou uma saída incorreta.
- *Detected (D)* - O CFE gerado foi detectado pela técnica de detecção de erros de fluxo de controle.
- *Segmentation fault (S)* - A execução do programa foi abortada, na tentativa de acessar em endereço inválido de memória. Sua execução causou um CFE detectado pelo sistema operacional.
- *Illegal instruction (I)* - CFE ilegal detectado pelo sistema operacional. O CFE levou o *program counter* a apontar para um endereço válido, mas que não representa uma instrução válida.
- *Outros erros detectados pelo sistema operacional (OS)* - Outros tipos de erros detectados pelo sistema operacional, tais como SIGKILL, SIGTRAP, SIGBUS, etc. Esses erros foram agrupados pois ocorreram com uma frequência muito menor.
- *Time out (T)* - O tempo de execução do programa ultrapassou o *timeout* estabelecido pelo injetor de falhas.

2.3.1 *Benchmarks* utilizados

Um conjunto de 8 programas da suíte de *benchmarks* Mibench foi utilizado nos experimentos. Cada programa foi compilado para a arquitetura AMD64 (x86-64).

Abaixo uma breve descrição de cada um dos *benchmarks* utilizados ao longo deste trabalho:

- Basicmath: Executa cálculos matemáticos simples que não necessitam de hardware específico, como solução de funções cúbicas, raízes quadradas, conversões de unidades, entre outras.
- 32-bit Cyclic Redundancy Check (CRC): Algoritmo de verificação utilizado para identificação de erros durante transmissões de dados.
- Dijkstra: Algoritmo que calcula caminhos possíveis entre pares de vértices em um grafo e seleciona o menor.
- Fast Fourier Transform (FFT): representa a transformada de Fourier, utilizada no processamento de sinais digitais.
- Patricia: Programa usado para representar tabelas de roteamentos utilizados para prestação de serviços nas áreas de redes de computadores e conexões.
- Qsort: Programa utilizado para a ordenação de dados.
- Rijndael: Executa operações de encriptação e decriptação de arquivos para garantir segurança.
- Stringsearch: realiza a busca de palavras em sentenças, operação bastante usada no processamento de textos.

3 IMPLEMENTAÇÃO E AVALIAÇÃO DAS TÉCNICAS CFCSS E CEDA

Esse capítulo tem como objetivo apresentar em detalhes duas técnicas já disponíveis na literatura da área: Control Flow Checking by Software Signatures (CFCSS) (OH; SHIRVANI; MCCLUSKEY, 2002) e Control-Flow Error Detection Using Assertions (CEDA) (VEMU; ABRAHAM, 2011). O objetivo dessas duas implementações foi comparar a técnica desenvolvida neste trabalho com outras técnicas da áreas.

O capítulo começa com uma breve introdução sobre técnicas para detecção de erros de fluxo de controle por software, na Seção 3.1. Posteriormente, na Seção 3.2, é feita uma avaliação da técnicas CFCSS e CEDA, e também são dados alguns detalhes sobre suas principais características.

3.1 Técnicas para detecção de erros de fluxo de controle por software

As técnicas para detecção de erros de fluxo de controle baseadas em software têm por objetivo contornar as limitações impostas pela técnicas baseadas em hardware. Aqui, código adicional é inserido no software para realizar a detecção dos erros de fluxo de controle. Essas técnicas são de baixo custo, uma vez que não envolvem a utilização de qualquer hardware adicional, tampouco requerem a modificação do hardware disponível. Entretanto, este tipo de técnica possui duas limitações:

(a) Elas são capazes de detectar apenas erros de fluxo de controle ilegais inter-bloco.

(b) É possível que uma falha afete as instruções de detecção, o que inviabiliza a detecção da falha.

Diversas técnicas deste tipo foram propostas ao longo dos últimos anos, dentre estas técnicas podemos destacar: *Control flow checking using assertions* (CCA) (KANAWATI; NAIR; KRISHNAMURTHY; ABRAHAM, 1996), *Enhanced Control-flow Checking Using Assertions* (ECCA) (ALKHALIFA; NAIR; KRISHNAMURTHY; ABRAHAM, 1999), *Control Flow Checking by Software Signatures* (CFCSS) (OH; SHIRVANI; MCCLUSKEY, 2002), *Assertions for Control Flow Checking* (ACFC) (VENKATASUBRAMANIAN; HAYES; MURRAY, 2003), *Yet Another Control-flow Checking using Assertions* (YACCA) (GOLOUBEVA; REBAUDENGO; REORDA; VIOLANTE, 2003), *Control-Flow Error Detection Using Assertions* (CEDA) (VEMU; ABRAHAM, 2011) e *Abstract Control Signatures* (ACS) (KHUDIA; MAHLKE, 2013).

Todas essas técnicas têm o mesmo princípio básico de funcionamento. Em tempo de execução, elas mantém uma variável global chamada assinatura (S), que é constantemente

atualizada e verificada. Um valor diferente do esperado no valor dessa variável ao longo da execução do programa evidencia um erro de fluxo de controle.

Em linhas gerais, os passos básicos dessas técnicas são:

- Em tempo de compilação:

    1. A técnica obtém, com a ajuda do compilador, o grafo de fluxo de controle do programa, como é mostrado na Figura 2.1.
    2. Depois, a técnica define para cada bloco básico do grafo uma valor, ao qual chamamos de assinatura local.
    3. Depois de definidas as assinaturas locais, a técnica adiciona ao código instruções extras que permitirão atualizar e verificar a assinatura global em tempo de execução.

- Em tempo de execução:

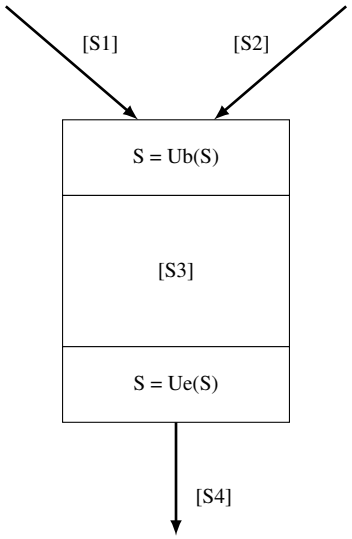
    4. Quando o fluxo de controle entra em um determinado bloco básico, o valor da assinatura global é atualizado através de uma função de atualização, que normalmente utiliza o valor da assinatura global e da assinatura local como argumentos.
    5. Depois de atualizada, a assinatura global é comparada com a assinatura local. Se não tiver ocorrido erro de fluxo de controle, os valores serão iguais. Então, o valor da assinatura é novamente atualizado na saída do bloco básico por outra função de atualização, e o fluxo segue para o bloco básico seguinte, fazendo o processo se repetir até cessar a execução do programa.
    6. Caso os valores sejam diferentes, evidencia-se um erro de fluxo de controle. Nesse caso o programa interrompe o fluxo original e direciona a execução para uma rotina que irá tratar o erro. Normalmente, a rotina gera uma mensagem informando o erro e interrompe a execução.

A Figura 3.1 ilustra um bloco básico com a estrutura básica das técnicas de detecção de erro de fluxo de controle por software. Podemos observar que o bloco básico possui dois vértices entrando e um vértice saindo. Na figura, S1, S2, S3 e S4 representam o valor da assinatura global em cada ponto de execução do programa, e *Ub* e *Ue* representam as funções de atualização, na entrada e saída do bloco básico, respectivamente. Instruções para verificar o valor da assinatura (não exibidas na figura) podem ser colocadas em diferentes pontos do bloco básico, na entrada ou na saída do bloco, ou em diferentes pontos do grafo de fluxo de controle. Isso irá variar de acordo com a técnica escolhida.

3.1.1 Classificação das técnicas de detecção de erros de fluxo de controle por software

As técnicas para detecção de erros de fluxo de controle por software podem ser divididas em duas categorias principais: *path-based assertions* e *predecessor/sucessor-asserting*. No método *path-based assertions*, apenas uma instrução de teste da assinatura global é adicionada ao final de cada grafo de fluxo de controle, com o intuito de verificar a execução correta desse caminho. No método *predecessor/sucessor-asserting*, todos os blocos básicos recebem um teste da assinatura global para verificar se o bloco básico anterior (posterior) no fluxo de controle é o predecessor (sucessor) correto.

Figura 3.1: Estrutura básica das técnicas de detecção de CFE por software.



Fonte: (VEMU; ABRAHAM, 2011), p. 1237

A diferença entre estas duas categorias é, portanto, o número de testes inseridos no programa. Como é possível perceber, o método *predecessor/sucessor-asserting* adiciona um maior número de verificações, por consequência ele gera um maior overhead e, potencialmente, uma maior cobertura de falhas. Por sua vez, o método *path-based assertions* insere menos verificações, tendo um menor overhead e menor taxa de cobertura.

ECCA, YACCA, CFCSS e CEDA são representantes do método predecessor/sucessor-asserting. ACFC e ACS são representantes do método *path-based assertions*.

Também podemos dividir o método *predecessor/sucessor-asserting* em duas subcategorias:

(a) Atualização incremental da assinatura global.

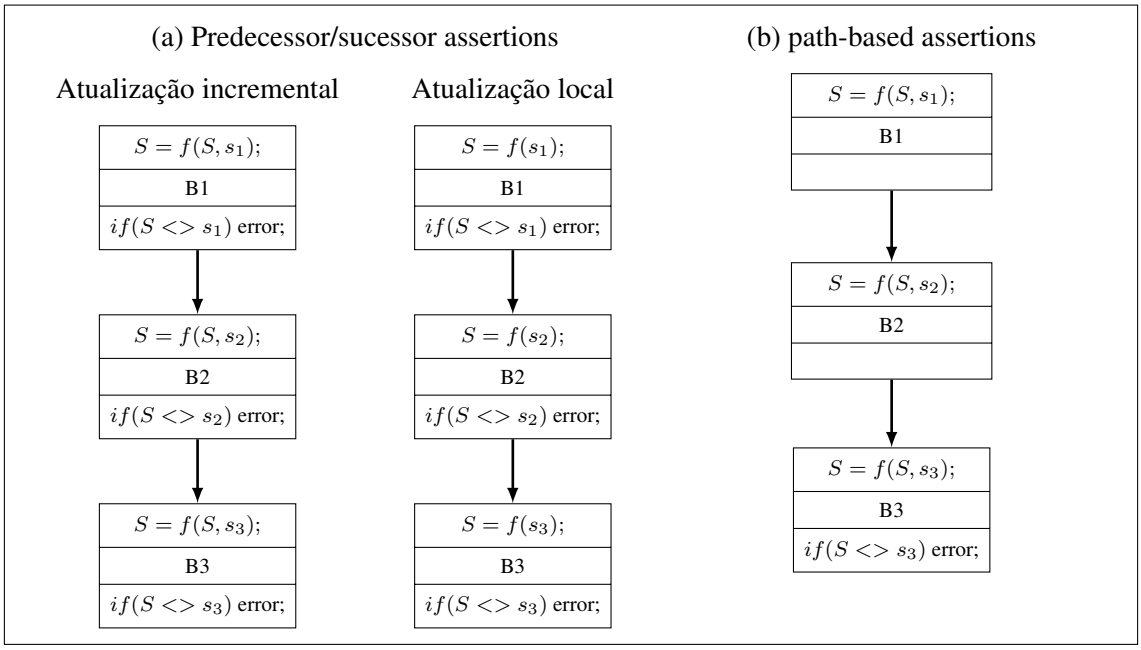
(b) Atualização local da assinatura global.

A Figura 3.2 ilustra a diferença entre estas duas subcategorias. No método de atualização incremental, a função de atualização da assinatura global utiliza o valor da própria assinatura global como entrada para o cálculo do novo valor. Isto significa que o valor da assinatura global de cada bloco básico depende de todas as funções de atualização ao longo do grafo de fluxo de controle. Nessa categoria podemos incluir as técnicas CEDA, CFCSS e YACCA.

Por outro lado, nos métodos do tipo atualização local, a função de atualização da assinatura global faz a atualização de forma independentemente dos demais blocos básicos. ECCA e CCA são exemplos dessa categoria.

A técnica para detecção de CFE proposta neste trabalho, descrita no Capítulo 4, pode ser classificada como sendo do tipo predecessor/sucessor com atualização local.

Figura 3.2: Categorias para atualização de assinaturas



3.2 Implementação de técnicas CFCSS e CEDA

O primeiro passo da pesquisa foi avaliar algumas técnicas conhecidas para detecção de erro de fluxo de controle por software. Para tal, foram implementadas duas técnicas: *Control Flow Checking by Software Signatures* (CFCSS) (OH; SHIRVANI; MCCLUSKEY, 2002) e *Control-Flow Error Detection Using Assertions* (CEDA) (VEMU; ABRAHAM, 2011). Ambas as técnicas foram escolhidas devido suas altas taxas de detecção de erros de fluxo de controle e por serem técnicas bem conhecidas na área.

A ferramenta utilizada para implementar as técnicas foi o *framework* LLVM (*Low Level Virtual Machine*) (LATTNER, 2002). O LLVM foi escolhido por ser uma ferramenta de código aberto, amplamente difundida e que permite facilmente a implementação de novas análises e transformações no código do programa, pois oferece uma plataforma com diversas funcionalidades para tal. Cada técnica foi implementada como uma transformação LLVM. Maiores detalhes sobre o LLVM são fornecidos no *Apêndice A*.

3.2.1 Comparação da cobertura de falhas entre CFCSS e CEDA

Para avaliar a implementação do CFCSS e CEDA foi realizado um conjunto de experimentos conforme descrito na Seção 2.3, onde foi utilizado o modelo de falhas *alteração do operando de desvio*, conforme descrido na Seção 2.2,

As Tabelas 3.1, 3.2 e 3.3 mostram os resultados desta etapa. A Tabela 3.1 exibe os resultados após a injeção de falhas nos *benchmarks* sem qualquer técnica de proteção. As Tabelas 3.2 e 3.3 mostram os resultados dos experimentos nos *benchmarks* protegidos pelas técnicas CFCSS e CEDA, respectivamente.

Os valores obtidos nesses experimentos foram coerentes com aqueles obtidos pelas técnica CFCSS e CEDA em seus artigos, (OH; SHIRVANI; MCCLUSKEY, 2002) e (VEMU; ABRAHAM, 2011), respectivamente.

A partir dos experimentos realizado com CFCSS e CEDA, percebeu-se que a maioria

Tabela 3.1: Benchmarks sem técnica de proteção

Benchmark	S	I	OS	T	EC	EW
basicmath	77,1%	15,8%	0,0%	0,0%	1,4%	5,7%
CRC	77,4%	1,7%	0,0%	0,0%	7,0%	13,9%
dijkstra	74,8%	9,2%	2,9%	0,0%	11,1%	2,0%
FFT	80,0%	8,1%	0,2%	0,7%	2,6%	8,4%
patricia	80,2%	5,0%	1,1%	0,7%	9,0%	4,0%
pbmsrch	80,2%	12,9%	0,0%	0,0%	4,6%	2,3%
qsort	92,4%	3,3%	0,0%	0,0%	2,5%	1,8%
rijndael	72,4%	6,1%	0,1%	0,0%	8,5%	12,9%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults.
T - Timeout. EC - Exited correct. EW - Exited wrong.

Tabela 3.2: Benchmarks protegidos com CFCSS

Benchmark	S	I	OS	T	EC	EW	D
basicmath	75,7%	10,8%	1,6%	0,0%	2,4%	0,6%	8,9%
CRC	72,6%	13,9%	1,0%	0,0%	3,2%	0,5%	8,8%
dijkstra	76,0%	10,3%	0,1%	0,4%	1,4%	0,0%	11,8%
FFT	72,7%	16,0%	0,2%	0,0%	1,5%	0,5%	9,1%
patricia	62,8%	21,6%	0,0%	0,0%	1,1%	1,2%	13,3%
pbmsrch	69,8%	16,2%	0,0%	0,0%	1,7%	0,0%	12,3%
qsort	81,6%	7,9%	0,3%	0,0%	3,0%	0,7%	6,5%
rijndael	68,2%	16,3%	1,0%	0,0%	0,6%	2,0%	11,9%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

das falhas inseridas não poderiam, em hipótese alguma, ser detectada por qualquer tipo de técnica do tipo *Signature Checking*, pois elas acarretavam em erros que geravam como destino endereços de memória que estavam fora do espaço de endereçamento do programa (*Segmenation fault*), coluna (S), em destaque, logo esses erros só poderiam ser detectados pelo sistema operacional. A partir desta percepção levantou-se a seguinte hipótese: *Talvez as técnicas estejam fazendo trabalho desnecessário pois, se grande parte das falhas são detectadas pelo sistema operacional, talvez não seja necessário proteger todos os blocos básicos.*

Com base nesta hipótese iniciou-se a segunda fase da pesquisa: Criar uma nova técnica, ou modificar uma técnica existente, de tal forma que apenas os blocos básicos suscetíveis a CFEs fossem protegidos, o que, consequentemente, poderia diminuir o *overhead*.

As seções a seguir dão mais alguns detalhes sobre as técnicas CFCSS e CEDA, procurando descrever as características que as diferenciam. A leitura destas seções é opcional.

3.2.2 Control Flow Checking by Software Signatures (CFCSS)

No CFCSS cada bloco básico recebe uma assinatura única de forma arbitrária em tempo de compilação, s_k. As instruções de atualização e verificação da assinatura são colocadas no início dos blocos básicos, antes das execução das instruções originais. Essas instruções são as seguintes:

Tabela 3.3: Benchmarks protegidos com CEDA

Benchmark	S	I	OS	T	EC	EW	D
basicmath	73,3%	15,1%	1,2%	0,0%	0,6%	0,1%	9,7%
CRC	75,2%	14,2%	2,0%	0,0%	1,1%	0,0%	7,5%
dijkstra	78,5%	14,3%	0,9%	0,0%	0,1%	0,0%	6,2%
FFT	68,8%	17,5%	0,9%	0,0%	0,6%	0,5%	11,7%
patricia	73,2%	17,1%	1,1%	0,0%	0,5%	0,2%	7,9%
pbmsrch	76,7%	14,0%	1,4%	0,0%	0,2%	0,0%	7,7%
qsort	83,1%	6,5%	0,2%	0,0%	1,4%	0,3%	8,5%
rijndael	67,1%	20,8%	1,6%	0,0%	0,5%	0,4%	9,6%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

(a) *Função de atualização da assinatura*: Operação *XOR* que gera um novo valor para a assinatura global em tempo de execução.

$$G = G \oplus d_k \tag{3.1}$$

Onde *G* é a assinatura global e d_k representa a diferença de assinatura entre dois blocos básicos (o bloco atual e seu predecessor). Este valor é calculado em tempo de compilação e depois utilizado na função de atualização da assinatura.

$$d_k = s_{k-1} \oplus s_k \tag{3.2}$$

(b) *Verificação do valor da assinatura*: Operação de desvio que verifica o valor da assinatura global em relação a assinatura local e realiza uma desvio para um bloco básico (*error_handler*), que irá tratar a exceção, caso os valores sejam diferentes.

$$br(G \neq s_k) \tag{3.3}$$

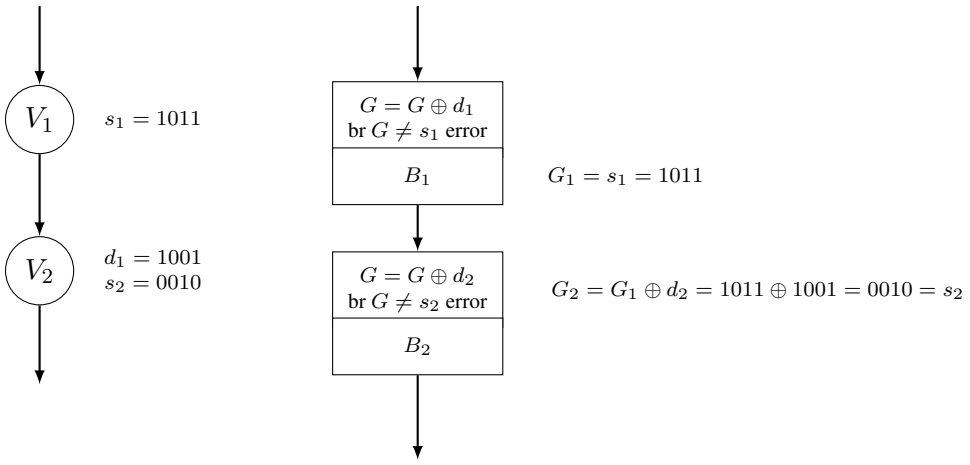
A Figura 3.3 exemplifica as instruções de atualização e verificação inseridas nos blocos básicos pelo CFCSS. À esquerda podemos observar os valores atribuídos aos blocos básicos em tempo de compilação. À direita observamos a aplicação das função de atualização e a verificação da assinatura.

Quando um bloco básico possui mais de um predecessor e um dos predecessores possui múltiplos sucessores, a função de atualização da assinatura global sofre alteração, passando a depender além da variável s_k, também de uma variável de ajuste, *D*. Cada bloco básico predecessor deve ter sua variável de ajuste. O cálculo da variável *D* é feito através da diferença de assinatura entre todos os blocos básicos predecessores do bloco básico que está sendo avaliando, com exceção do primeiro (escolhe-se arbitrariamente um dos blocos básicos como sendo o primeiro), que deve receber o valor zero.

$$D = s_{k-1} \oplus s_{k-2} \oplus s_{k-n} \tag{3.4}$$

A Figura 3.4 ilustra como é feita a atualização da assinatura global quando um bloco básico possui múltiplos predecessores e um dos predecessores possui múltiplos sucessores.

Figura 3.3: CFCSS



Fonte: (OH; SHIRVANI; MCCLUSKEY, 2002), pg 114

3.2.3 Control-Flow Error Detection Using Assertions (CEDA)

Na técnica CEDA, a atribuição de assinaturas aos blocos básicos possui um mecanismo bem mais complexo do que o CFCSS, que atribui um valor arbitrário para a assinatura de cada bloco básico. O objetivo deste mecanismo é reduzir o *aliasing*. Maiores detalhes sobre ele podem ser obtidos em (VEMU; ABRAHAM, 2011).

Em relação a atualização, a assinatura global é atualizada duas vezes em cada bloco básico. Uma no início da execução do bloco e outra no final da execução dele. No início do bloco a atualização varia de acordo com o tipo do bloco básico. CEDA classifica os blocos básicos da seguinte forma:

- **Tipo A**: Um bloco básico é do tipo A se ele tem múltiplos predecessores e ao menos um dos seus predecessores tem múltiplos sucessores.
- **Tipo X**: Um bloco básico é do tipo X se não for do tipo A.

Então, se o bloco básico for do tipo A, a função de atualização será a seguinte, onde S representa a assinatura global, d_1 e d_2 representam as assinaturas associadas ao bloco básico (assinaturas locais), a primeira representa a assinatura de entrada do bloco e a segunda a assinatura de saída, (N_i) representa o bloco básico (ou nodo) que estamos avaliando:

$$S = S \wedge d_1(N_i) \tag{3.5}$$

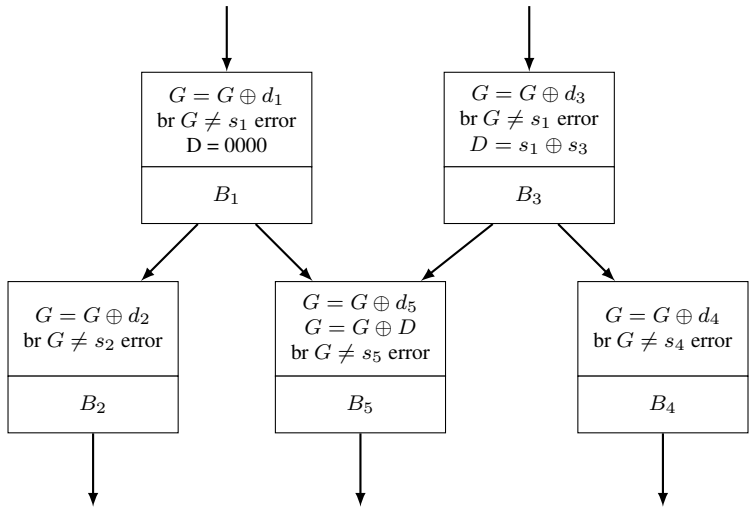
Se o bloco básico for do tipo X, a função de atualização será:

$$S = S \oplus d_1(N_i) \tag{3.6}$$

No final do bloco básico a função de atualização é sempre a mesma:

$$S = S \oplus d_2(N_i) \tag{3.7}$$

Figura 3.4: CFCSS: Múltiplos blocos básicos predecessores

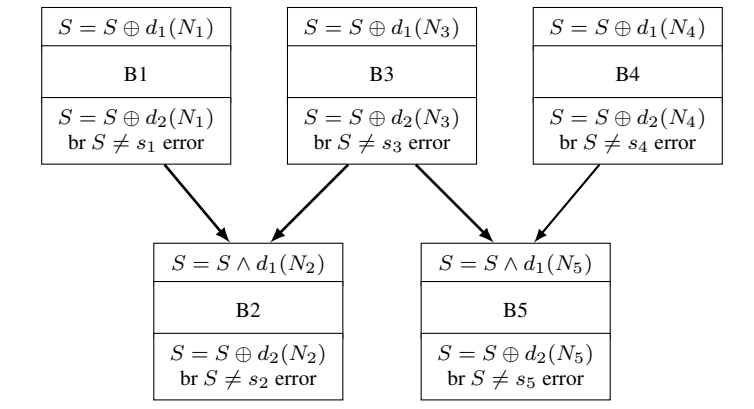


Fonte: (OH; SHIRVANI; MCCLUSKEY, 2002), p. 116

As instruções de verificação do valor da assinatura podem ser colocadas, segundo o autor da técnica CEDA, no inícios do bloco básico ou no final dele, dependendo da latência de detecção de erro desejada. Na nossa implementação, foi colocada apenas uma instrução de verificação no final de cada bloco básico.

A Figura 3.5 ilustra as instruções inseridas nos blocos básicos de um programa protegido pela técnica CEDA.

Figura 3.5: Instruções inseridas nos blocos básicos protegidos pela técnica CEDA



4 FLEXIBLE CONTROL FLOW CHECK (FCFC)

O objetivo da técnica desenvolvida neste trabalho é limitar as instruções de verificação e atualização de assinaturas (código extra inserido no programa) aos blocos básicos suscetíveis. Para que isso ocorra é necessário que a técnica de detecção de erros de fluxo de controle permita a remoção destas instruções, de forma independente dos demais blocos básicos do grafo de fluxo de controle do programa. Infelizmente, nos métodos do tipo *path-based assertions* não é possível remover instruções de verificação e atualização de assinaturas dos blocos básicos não suscetíveis, pois esse método insere instruções de atualização em todos os blocos básicos e instruções de teste apenas no último bloco básico do fluxo de controle, ou seja, existe uma dependência do valor da assinatura ao longo de todo o fluxo de controle. No método *predecessor/sucessor-asserting* ocorre algo semelhante, pois o valor da assinatura global depende dos valor de todas as assinaturas locais ao longo do fluxo de controle.

Além disso, as técnicas existentes costumam alterar o grafo de fluxo de controle do programa, pois para cada instrução de verificação da assinatura é necessário inserir um novo bloco básico no programa. As Figuras 4.1 e 4.2 mostram bem este problema. Na Figura 4.1 temos o grafo de fluxo de controle de uma função do *benchmark* FFT, sem qualquer técnica de proteção. Já na Figura 4.2 temos a mesma função protegida pela técnica CEDA. Como podemos perceber, CEDA introduziu sete blocos básicos ao grafo de fluxo de controle do programa.

Figura 4.1: CFG da função *IsPowerOfTwo* do benchmark FFT sem técnica de detecção

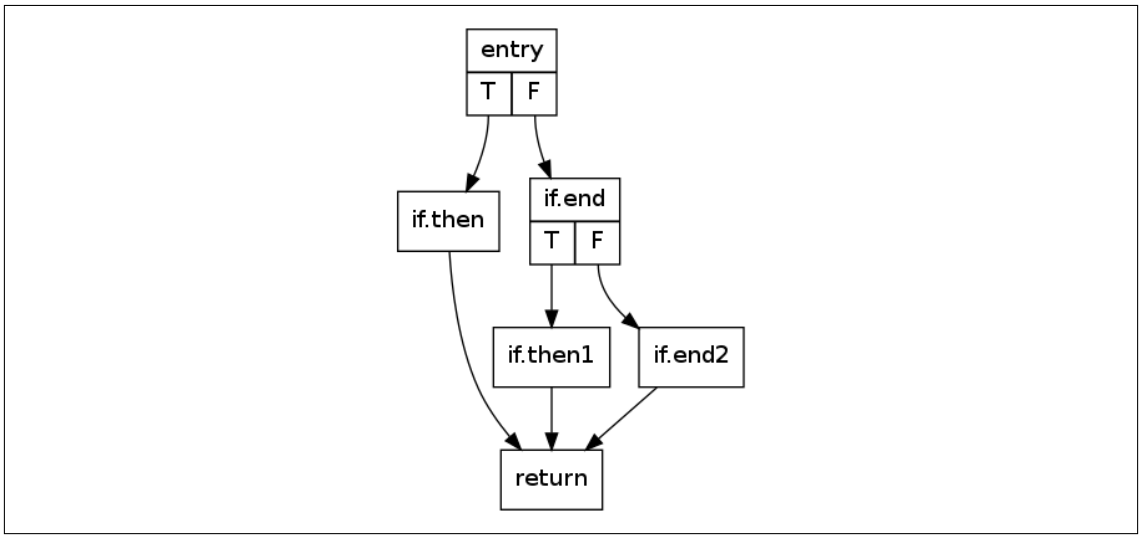
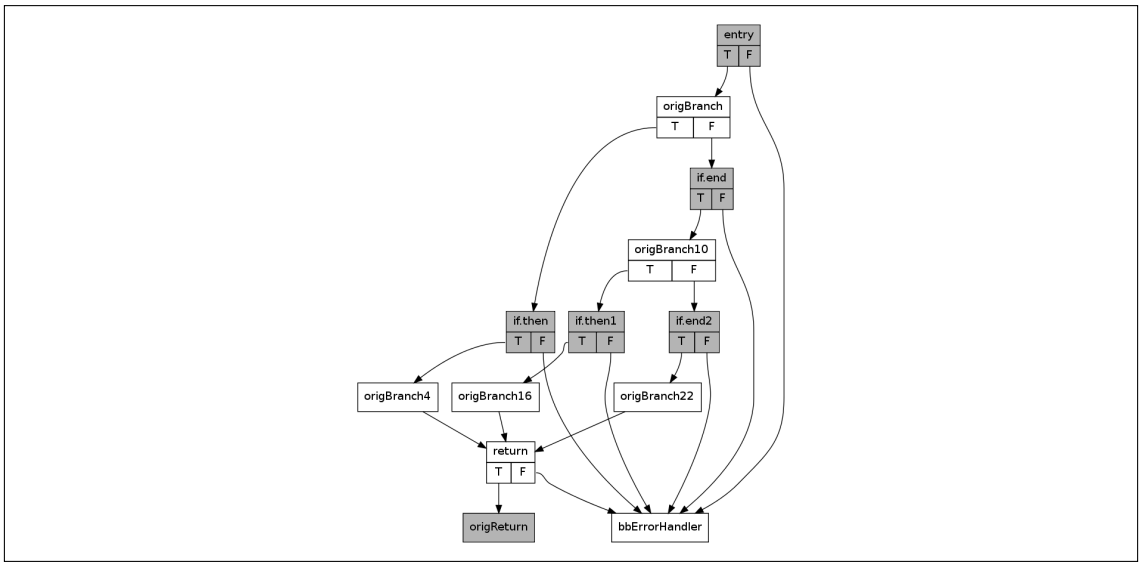


Figura 4.2: CFG da função *IsPowerOfTwo* do benchmark FFT protegido pela técnica CEDA



Como veremos a seguir a técnica de detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip* necessita que o grafo de fluxo de controle do programa permaneça inalterado, pois qualquer alteração no grafo de fluxo de controle implica em alterações nos endereços de memória das instruções do programa e, para que a análise sistemática de *single bit-flip* permaneça válida, os endereços das instruções do programa devem permanecer os mesmos ao longo de todas as etapas.

Por essas razões, foi necessário desenvolver uma técnica eficiente que utilizasse *atualização local de assinaturas e que não causasse alterações no grafo de fluxo de controle do programa*. Para obter o resultado desejado combinamos a técnica CFCSS com o método de atualização local de assinaturas e chegamos a uma nova técnica que chamamos de *Flexible Control Flow Check* (FCFC).

A técnica FCFC foi implementada como uma transformação do compilador LLVM. Ela utiliza uma variável global, *G*, chamada de *Global Checking*, inicializada com valor zero (0), além de uma variável global, *S*, para verificar o valor da assinatura. Em tempo de compilação, cada bloco básico recebe uma assinatura local, que nada mais é do que um número inteiro único, ao qual chamaremos de *id* do bloco.

As funções de atualização das variáveis *G* e *S* operam da seguinte forma:

(a) Se o bloco básico possui apenas um predecessor

  (i) A assinatura global, *S*, é atualizada no final do bloco básico predecessor com o valor do *id* presente nesse bloco.

  (ii) No final do bloco básico sucessor, a assinatura global, *S*, é atualizada através da operação de ou exclusivo entre o valor atual de *S* e o valor do *id* do bloco básico predecessor. O valor da variável G é atualizada através da operação ou exclusivo entre o valor atual de *G* e o valor da assinatura global, *S*.

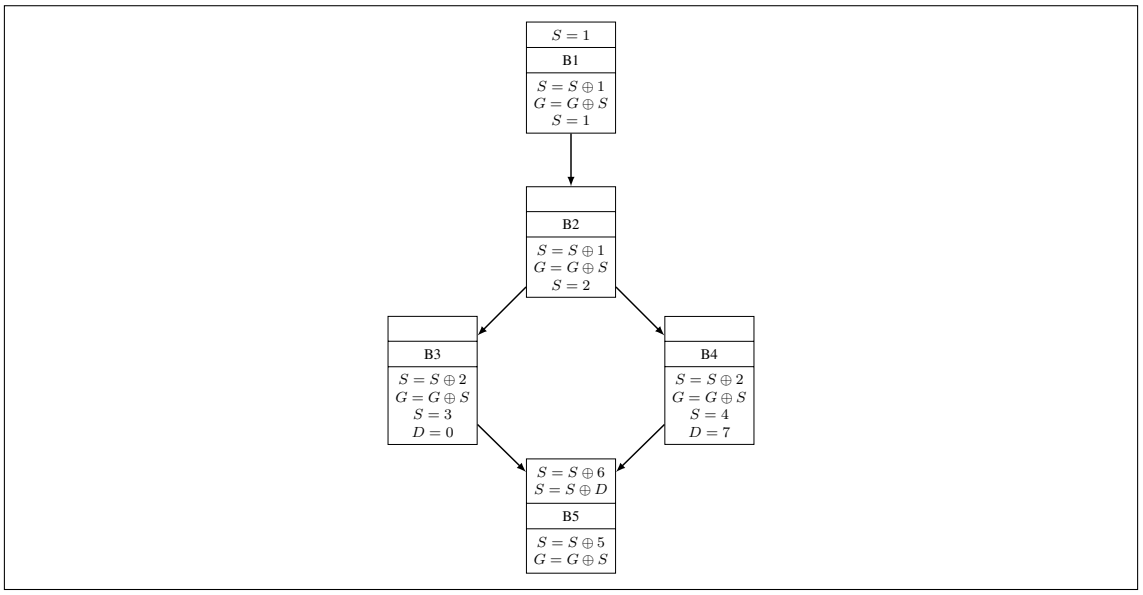
(b) Se o bloco básico possui mais de um predecessor

(i) A assinatura global é atualizada no predecessor com o *id* do predecessor, e uma variável auxiliar, *D*, é atualizada da seguinte forma: i) Se for o primeiro bloco predecessor, recebe valor zero; ii) Para cada bloco básico predecessores seguinte, o valor de *D* será igual a diferença de assinatura entre o primeiro bloco básico predecessor e o bloco básico que estamos avaliando. Por exemplo, no bloco básico B3 da Figura 4.3, o valor de *D* é zero, pois ele é considerado o primeiro bloco básico predecessor de B5. No bloco básico *B4*, o valor de *D* é igual a diferença de assinatura entre B4 e B3, ou seja, $D = 4 \oplus 3$.

(ii) No início do bloco básico sucessor a assinatura global é atualizada através da operação de ou exclusivo entre a assinatura global e a diferença de assinatura entre o bloco básico sucessor e seu primeiro predecessor. No exemplo da Figura 4.3, no bloco básico B5 ocorre o ou exclusivo entre a assinatura global e o valor 6, que representa a diferença de assinatura entre o bloco básico B5 e o bloco básico B3 (primeiro predecessor). Depois, a assinatura global é atualizada através da operação de ou exclusivo entre a assinatura global e a variável *D*.

(iii) No final do bloco básico sucessor, a assinatura global, *S*, é atualizada através da operação de ou exclusivo entre o valor atual de *S* e o valor do *id* do bloco básico predecessor. O valor da variável G é atualizada através da operação ou exclusivo entre o valor atual de *G* e o valor da assinatura global, *S*.

Figura 4.3: Blocos básicos protegidos por FCFC



Ao final da execução do programa, o valor da variável G é verificado. Se não tiver ocorrido nenhum erro de fluxo de controle, o valor da variável será igual a zero. Se tiver ocorrido um erro de fluxo de controle, o valor será diferente de zero e o programa irá gerar uma mensagem alertando sobre o problema.

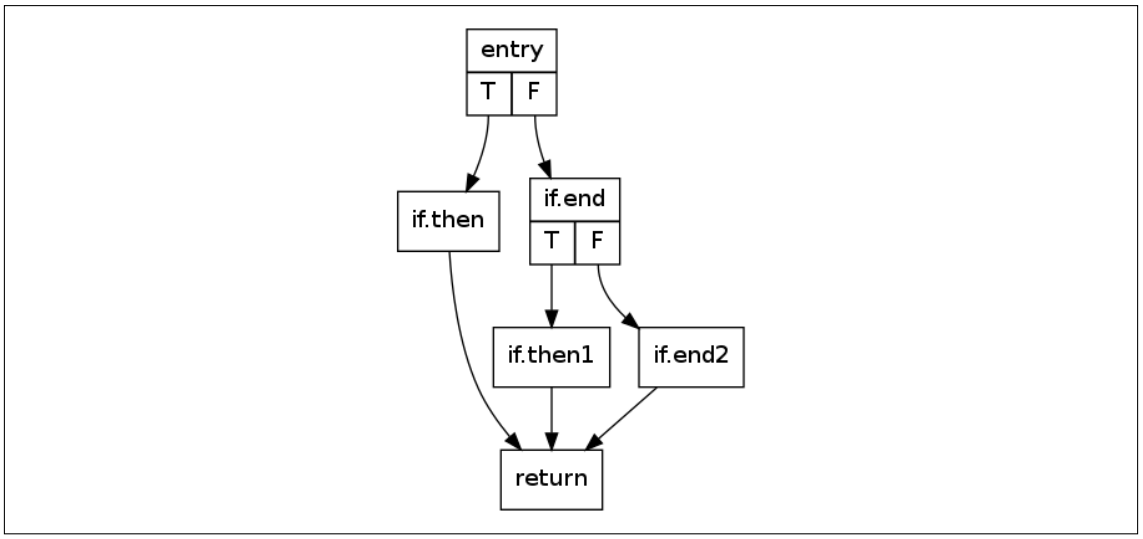
Tanto na técnica CEDA como na CFCSS, a assinatura é verificada em cada bloco básico, o que implica na inserção de instruções de comparação do valor da assinatura e de instruções de desvio, por consequência, também implica na geração de novos blocos

básicos, alterando demasiadamente o fluxo de controle dos programas. Como podemos observar, a técnica FCFC evita a alteração do grafo de fluxo de controle do programa, pois evita a inserção de novos blocos básicos. É necessário apenas um novo bloco básico ao final do programa, fora do grafo de fluxo de controle original, para verificar o valor da variável *G*. A Figura 4.3 ilustra as instruções inseridas nos blocos básicos do programa pela técnica FCFC.

4.1 Avaliação da técnica FCFC

No que diz respeito a não alterar o grafo de fluxo de controle do programa, FCFC mostrou-se muito eficaz. A Figura 4.4 mostra o fluxo de controle da mesma função do *benchmark* FFT exibida na Figura 4.1, agora protegida pela técnica FCFC. Como podemos perceber, o grafo de fluxo de controle das Figuras 4.1 e 4.4 são idênticos, pois FCFC não inseriu novos blocos básicos no grafo de fluxo de controle da função.

Figura 4.4: CFG da função *IsPowerOfTwo* do benchmark FFT protegida pela técnica FCFC



Em relação a capacidade de detecção de erros de fluxo de controle, FCFC mostrou-se similar às técnicas existentes. A Tabela 4.1 mostra o resultado do experimentos realizados com os *benchmarks* protegidos pela técnica FCFC tendo como modelo de falhas *single bit-flip* nos operandos das instruções de desvio.

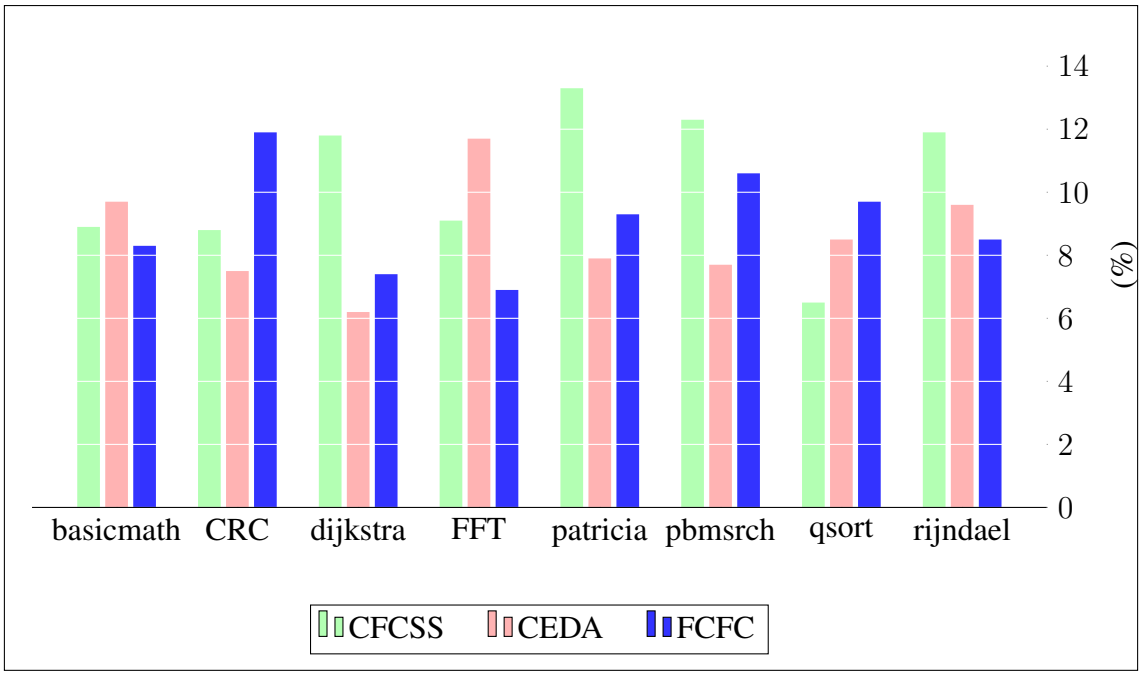
Já a Figura 4.5 mostra a comparação da técnica FCFC em relação às técnicas CFCSS e CEDA. No FCFC a média da taxa de detecção de erros foi de 9,06%. O que é muito próximo à média obtida pelas técnicas CFCSS e CEDA, 10,32% e 8,60%, respectivamente. Sendo assim, podemos concluir que FCFC possui uma eficiência, no que diz respeito a detecção de erros, equiparável às técnicas CFCSS e CEDA.

Tabela 4.1: FCFC: Taxa de cobertura

Benchmark	S	I	OS	T	EC	EW	D
basicmath	70,5%	14,6%	1,6%	0,0%	2,0%	3,0%	8,3%
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	2,2%	11,8%
dijkstra	79,2%	7,2%	2,2%	0,0%	4,0%	0,0%	7,4%
FFT	78,8%	7,2%	0,1%	0,0%	1,3%	5,7%	6,9%
patricia	73,8%	9,3%	0,1%	0,1%	6,4%	1,0%	9,3%
pbmsrch	76,3%	8,6%	0,1%	0,0%	4,2%	0,2%	10,6%
qsort	74,5%	9,4%	0,0%	0,0%	3,7%	2,7%	9,7%
rijndael	74,0%	9,1%	1,4%	0,0%	4,2%	2,8%	8,5%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

Figura 4.5: Comparação da taxa de cobertura de falhas



5 DETECÇÃO E PROTEÇÃO DE BLOCOS BÁSICOS SUSCETÍVEIS ATRAVÉS DA ANÁLISE SISTEMÁTICA DE *SINGLE BIT-FLIP*

Um *bit-flip* no operando de uma instrução de desvio pode gerar um erro de fluxo de controle. Para detectar esse tipo de erro, existem técnicas que adicionam código extra em todos os blocos básicos do programas. Apesar do código extra inserido, muitos dos erros de fluxo de controle não são detectados por esse código, eles são detectados pelo sistema operacional. Como demonstrado em trabalhos anteriores, como, por exemplo, em (ALKHALIFA; NAIR; KRISHNAMURTHY; ABRAHAM, 1999; OH; SHIRVANI; MCCLUSKEY, 2002; VEMU; ABRAHAM, 2011), aproximadamente entre 50% a 60% dos erros são detectados pelo sistema operacional. Do total de erros detectados pelos sistema operacional, grande parte representa erros do tipo *Segmentation fault*.

Segmentation fault indica que o programa tentou acessar uma posição inválida de memória, geralmente um posição fora do espaço de endereçamento do programa, portanto é impossível para o código extra inserido nos blocos básicos detectar esse tipo de erro.

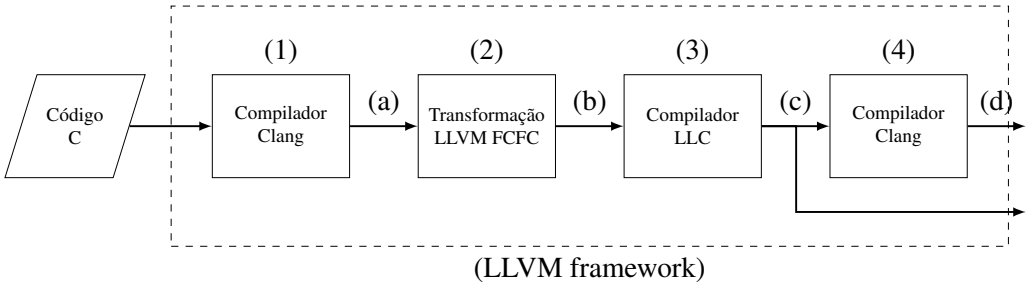
Através da análise sistemática de *single bit-flip* no operando das instruções de desvio é possível detectar quais erros de fluxo de controle atingirão um endereço de memória fora do espaço de endereçamento do programa e quais terão como destino o espaço de endereçamento do programa. A análise sistemática de *single bit-flip* também permite detectar quais blocos básicos do programa serão atingidos por erros de fluxo de controle causados por *sigle bit-flip* no operando de endereço de instruções de desvio - assim detectando os chamados blocos básicos suscetíveis - e quais não serão. A partir da detecção dos blocos básicos suscetíveis é possível proteger apenas esses blocos, evitando proteger os blocos básicos não suscetíveis, o que possibilita a redução do *overhead* imposto pela técnica de detecção de CFEs.

Este capítulo apresenta a técnica de detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*. Na Seção 5.1 são apresentadas as principais etapas da técnica. Na Seção 5.2 é apresentada a definição de bloco básico suscetível. Na Seção 5.3 é apresentado o processo de análise sistemática de *single bit-flip*. Não Seção 5.4 é descrita como a transformação LLVM é realizada nos programas. Nas seções 5.5 e 5.6 são avaliadas a taxa de cobertura dos *benchmarks* com o primeiro e segundo refinamento, respectivamente. Não Seção 5.7 é descrito como as instruções de desvio foram reposicionadas na tentativa de evitar a execução das instruções *NOP* e é feita uma nova avaliação das taxa de cobertura dos *benchmarks* com o primeiro e segundo refinamento, respectivamente. Por fim, na Seção 5.8 são descritos alguns problemas encontrados durante o processo de desenvolvimento da técnica.

5.1 Etapas da técnica de detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*

A Figura 5.1 mostra a visão esquemática da primeira etapa do processo de detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*. No passo 1, o código fonte é compilado, pelo compilador Clang, para a linguagem de representação intermediária do LLVM (LLVM-IR). No passo 2, a transformação FCFC é aplicada ao código intermediário. Essa etapa produz um código intermediário onde todos os blocos básicos estão protegidos pela técnica FCFC. No passo 3, o código na linguagem intermediária é compilado para o código *assembly* pelo compilador LLC do LLVM. Por fim, no passo 4, o código *assembly* é compilado, pelo compilador Clang, para um executável na arquitetura x86-64. Como podemos observar, essa etapa produz dois artefatos diferentes, o código *assembly* do programa e seu executável.

Figura 5.1: Detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*: Etapa 1



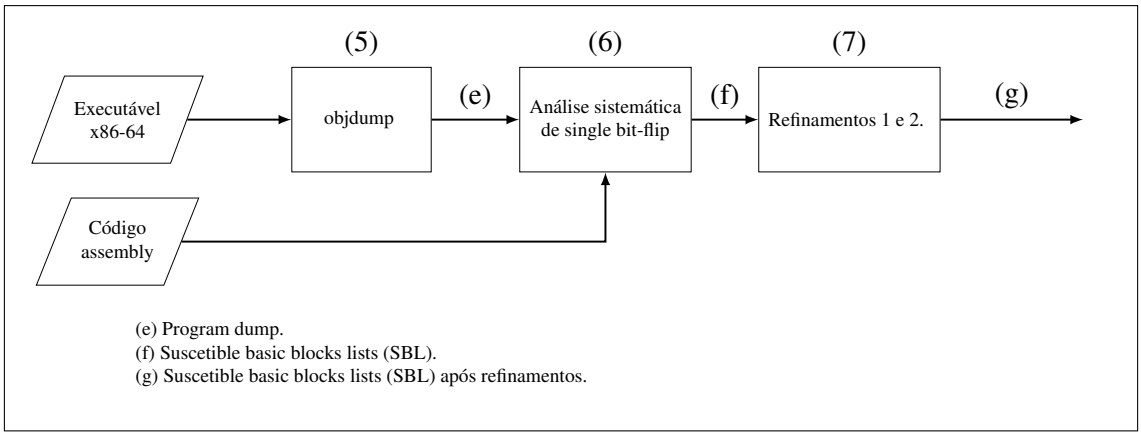
(a) LLVM-IR
(b) LLVM-IR + FCFC. Todos os blocos básicos protegidos.
(c) Código assembly (mnemônicos e endereços simbólicos). Todos os blocos básicos protegidos.
(d) Executável. Todos os blocos básicos protegidos.

A Figura 5.2 mostra a segunda etapa do processo. As entradas desta etapa são os código *assembly* e o executável x86-64 gerados na etapa anterior. No passo 5, o *dump* do programa x86-64 é gerado. O *dump* é um arquivo no formato texto que contém todas as instruções do programa e o endereço de memória de cada uma dessas instruções. Após, no passo 6, a análise sistemática de *single bit-flip* é realizada tendo como entrada o código *assemly* e o *dump* do programa. O resultado da análise sistemática de *single bit-flip* é a lista com todos os blocos básicos suscetíveis (*susceptible basic blocks list* - SBL). No passo 7 a lista de blocos básicos suscetíveis passa por dois refinamentos. No primeiro refinamento são removidos os blocos básicos afetados apenas por desvios ilegais *intra-bloco*. No segundo refinamento são removidos os blocos básicos afetados apenas por desvios errados, ou seja, desvios previstos no grafo de fluxo de controle do programa. Cada um dos refinamentos gera uma lista de blocos básicos suscetíveis diferente.

Cada um dos passos da etapa 2 é realizado fora do ambiente do compilador LLVM. A extração do *dump* do programa x86-64 é feita através comando Linux *objdump*. Já a análise sistemática de *single bit-flip* e os posteriores refinamentos, são feitos por *scripts* escritos na linguagem *bash*.

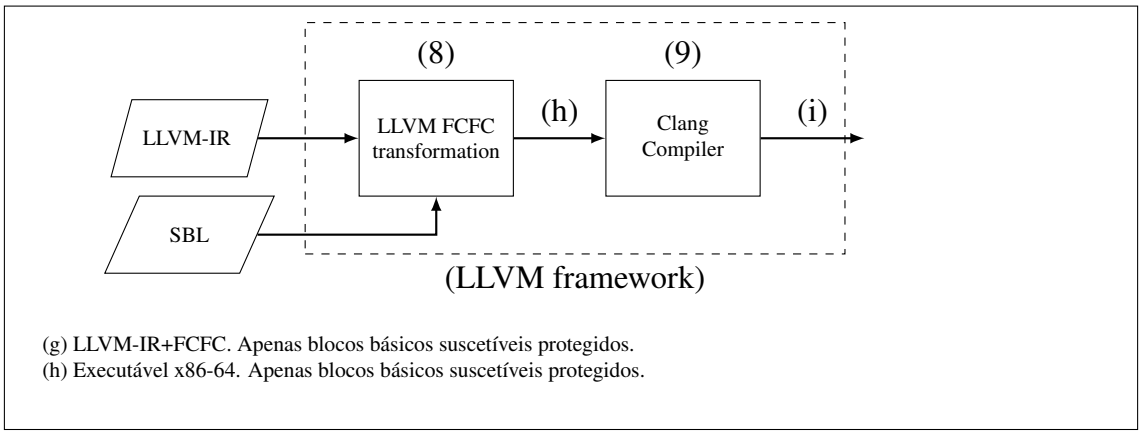
A Figura 5.3 mostra a terceira e última etapa do processo. As entradas para esta etapa

Figura 5.2: Detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*: Etapa 2



são o código intermediário LLVM (LLVM-IR) gerado do passo 1 da etapa 1, e a lista dos blocos básicos suscetíveis com o primeiro refinamento, ou com o segundo refinamento, gerada no passo 7 da etapa 2. No passo 7 o compilador, de posse da lista de blocos básicos suscetíveis, aplica a transformação FCFC gerando um arquivo LLVM-IR onde apenas os blocos básicos presentes na lista de blocos básicos suscetíveis são protegidos. Por fim, no passo 8, o arquivo LLVM-IR é compilado para um executável na arquitetura x86-64, onde apenas os blocos básicos suscetíveis estão protegidos.

Figura 5.3: Detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*: Etapa 3



É importante destacar que a técnica FCFC pôde ser utilizada porque ela possui as características necessárias para análise sistemática de *single bit-flip*. São elas:

1) Atualização local da assinatura. Isso significa que a função de atualização da assinatura não depende das assinaturas dos demais blocos básicos ao longo do fluxo de controle.

2) Ela não introduz novos blocos básicos ao grafo de fluxo de controle original, o que garante que análise sistemática de *single bit-flip* não será afetada.

Apesar da técnica FCFC ter sido escolhida para a proteção dos blocos básicos, qualquer outra técnica com as mesmas características poderia ser utilizada.

Na seção seguinte definimos precisamente o que são blocos básicos suscetíveis, e quais refinamentos podem ser realizados a fim de identificar quais blocos básicos devem, ou não, ser protegidos pela técnica de detecção de CFEs. A seguir são apresentados detalhes sobre a análise sistemática de *single bit-flip*. Por fim, mostramos como é gerada a versão do programa onde apenas os blocos básicos suscetíveis são protegidos.

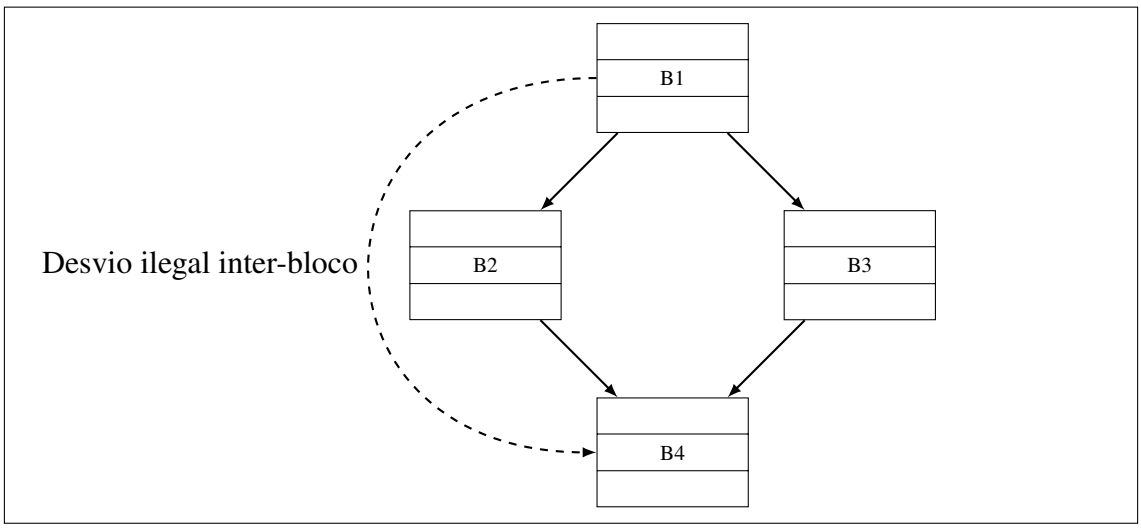
5.2 Blocos básicos suscetíveis e desvios errados

Para entender o objetivo da técnica proposta neste trabalho, precisamos, primeiro, compreender o que são *blocos básicos suscetíveis* e quais são os diferentes efeitos provocados por erros de fluxo de controle no software.

A Figura 5.4 mostra um fluxo de controle simples, com apenas quatro blocos básicos. Neste fluxo de controle podemos perceber que existem vários desvios legais, de B1 para B2 e B3, de B2 para B4 e de B3 para B4. Porém, na ocorrência de um erro de fluxo de controle, podemos ter diferentes desvios ilegais.

Podem ocorrer desvios ilegais *intra-bloco*, por exemplo, um desvio que parte do bloco básico B1 e atinge o próprio bloco básico. Desvios ilegais *intra-bloco* estão fora do escopo das técnica de detecção de erros de fluxo de controle, portanto, também não são considerados neste trabalho. Podem ocorrer desvios ilegais *inter-bloco*. Um desvio ilegal *inter-bloco* ocorre quando um erro de fluxo de controle criar um desvio (aresta) entre blocos básicos de um programa não existente no grafo de fluxo de controle original, por exemplo, entre os blocos básicos B1 e B4. Por fim, podem ocorrer desvios ilegais que tem como destino um endereço de memória fora do espaço de endereçamento do programa.

Figura 5.4: Bloco básico suscetível



Então, por exemplo, considerando como origem de falhas **apenas** o bloco básico B1, se pelo menos um desvio ilegal, que parte desse bloco, tiver como destino o bloco básico B4, chamamos o bloco básico B4 de *bloco básico suscetível*, ou seja, **os blocos básicos suscetíveis de um programa são os blocos básicos do programa que podem ser alvo de um desvio ilegal inter-bloco**. Mais precisamente, os blocos básicos suscetíveis possuem

endereços de memória que são alvos de um desvio ilegal inter-bloco.

Quando consideramos como destino de uma falha os blocos básicos B2 e B3, e como origem o bloco básico B1, **podemos ter um *desvio errado* quando o programa executa o bloco básico B2 em um momento em que o esperado era a execução do bloco básico B3**. FCFC não tem por objetivo detectar *desvios errados*, uma vez que os desvios (arestas) envolvidos na operação pertencem ao grafo de fluxo de controle do programa.

Ainda sobre a Figura 5.4, podemos observar que, se todos os blocos básicos estiverem protegidos por uma técnica de detecção de erros de fluxo de controle, e na hipótese de não ocorrer no mínimo um desvio ilegal *inter-blocos* entre quaisquer dois blocos básicos (todos os desvios ilegais podem ter como destino uma área de memória fora do espaço de endereçamento do programa, ou podem ocorrer apenas *desvios errados*, ou, ainda, desvios *intra-blocos*), o código extra inserido nesses blocos básicos será inútil para detectar qualquer erro de fluxo de controle e só irá acarretar em um *overhead* desnecessário.

5.3 Análise sistemática de *single bit-flip*

Para identificar os blocos básicos suscetíveis, aqueles que são alvos de um desvio ilegal inter-bloco e que por consequência precisam ser protegidos pela técnica de detecção de CFEs, é necessário realizar a análise sistemática de *single bit-flip*.

Quando uma falha transiente afeta o operando de um desvio, ou seja, o valor que define a posição para onde o fluxo de controle do programa irá seguir, esta falha manifesta-se como um *single bit-flip* no valor do operando do desvio. Tendo em vista que o operando possui um número fixo de bits, é possível simular todos os *single bit-flip* que podem ser ocasionados por falhas transientes neste operando. Portanto, é possível gerar todos os potenciais endereços para onde o fluxo de controle será direcionado caso ocorra um erro de fluxo de controle.

Também é importante observar que o esquema proposto considera que todas as instruções de desvio têm seus endereços de destino calculados de maneira estática, sendo assim todos os endereços são conhecidos antes da execução do programa, ou seja, não há desvios indiretos. Isso foi possível graças a utilização dos compilador LLVM, que permite gerar código sem desvios indiretos.

Então, por exemplo, levando-se em consideração a instrução de desvio ***jmpq 402475***, e tendo-se em vista que o operando possui 24 bits (x86-64), a Tabela 5.1 mostra alguns dos possíveis valores que o operando de desvio pode assumir caso ele seja afetado por um *single bit-flip*, ou seja, a tabela mostra uma simulação da análise sistemática de *single bit-flip*.

Para realizar a análise sistemática de *single bit-flip* o esquema proposto recebe como entrada o código *assembly* e o *dump* do programa. Através do *dump* é possível obter todos os endereços de memória do programa, todas as instruções de desvio, e todos os operandos dessas instruções. Com essa informação é feita a análise de *single bit-flip*, onde todos os endereços de destinos são gerados e salvos em um arquivo texto. Após, todos os endereços são avaliados e aqueles que forem inválidos são descartados. São considerados inválidos todos os endereços que estão fora do espaço de endereçamento do programa, ou seja, que correspondem a desvios ilegais para fora do programa. Com a lista dos endereços válidos é possível obter o nome do bloco básico de cada um deles, analisando-se o código *assembly*, que possui o nome de cada bloco básico do programa, gerando assim a lista de blocos básicos suscetíveis.

Porém, a lista de blocos básicos suscetíveis ainda não está finalizada, pois ela precisa

Tabela 5.1: Análise sistemática de single bit-flip

Original	Bit-flip	Resultante
jne,402475	0000 0000 0000 0000 0000 0001	jne, 402474
jne,402475	0000 0000 0000 0000 0000 0010	jne, 402477
jne,402475	0000 0000 0000 0000 0000 0100	jne, 402471
jne,402475	0000 0000 0000 0000 0000 1000	jne, 40247D
jne,402475	0000 0000 0000 0000 0001 0000	jne, 402465
jne,402475	0000 0000 0000 0000 0010 0000	jne, 402455
jne,402475	0000 0000 0000 0000 0100 0000	jne, 402435
...	...	...
...	...	...
jne,402475	0000 0010 0000 0000 0000 0000	jne, 422475
jne,402475	0000 0100 0000 0000 0000 0000	jne, 442475
jne,402475	0000 1000 0000 0000 0000 0000	jne, 482475
jne,402475	0001 0000 0000 0000 0000 0000	jne, 502475
jne,402475	0010 0000 0000 0000 0000 0000	jne, 602475
jne,402475	0100 0000 0000 0000 0000 0000	jne, 2475
jne,402475	1000 0000 0000 0000 0000 0000	jne, C02475

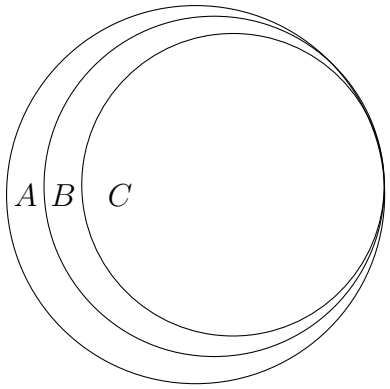
passar por dois refinamentos:

1. No primeiro refinamento são removidos todos os blocos básicos afetados apenas por desvios ilegais *intra-bloco*. Para identificar se um determinado bloco básico é afetado apenas por desvios ilegais *intra-bloco*, precisamos verificar na lista de endereços válidos se todos as falhas que têm como destinos esse bloco têm como origem o mesmo bloco básico.

2. No segundo refinamento são removidos todos os blocos básicos afetados apenas por *desvios errados*. Para identificar se um determinado bloco básico é afetado apenas por *desvios errados*, precisamos verificar na lista de endereços válidos se todos as falhas que têm como destinos esse bloco básico têm como origem um bloco básico que é o seu predecessor no grafo de fluxo de controle. Se um determinado bloco básico é afetado por falhas que tem como origem apenas seu predecessor, então esse bloco básico não precisa ser protegido. Para auxiliar na tarefa de detecção de *desvios errados*, o compilador LLVM gera um arquivo texto com a lista de todos os blocos básicos do programa e sues respectivos sucessores, que é utilizada pelo *framework* de análise.

Ao final da análise sistemática de *single bit-flip* duas listas são geradas: a) lista de blocos básicos suscetíveis após o primeiro refinamento, e b) a lista de blocos suscetíveis após o segundo refinamentos. Como podemos observar na Figura 5.5, cada refinamento representa um subconjunto da lista de blocos básicos suscetíveis.

Figura 5.5: Refinamentos



(A) Blocos básicos suscetíveis.
(B) Blocos básicos suscetíveis após o primeiro refinamento.
(C) Blocos básicos suscetíveis após o segundo refinamento.

5.4 Desalinhamento entre versões

Como dito anteriormente, o objetivo da técnica desenvolvida neste trabalho é proteger apenas os blocos básicos suscetíveis, removendo, portando, instruções de detecção de erros de fluxo de controle dos blocos básico não suscetíveis.

A primeira tentativa para alcançar este objetivo foi criar um passo no LLVM que, de posse da lista dos blocos básicos suscetíveis, protegia apenas esses blocos básicos. A Figura 5.6 exemplifica essa abordagem. Primeiro todos os blocos básicos do programa são protegidos. Posteriormente é feita a análise sistemática de *single bit-flip* no programa gerando a lista de blocos básicos suscetíveis. De posse da lista, o programa é compilado novamente, e é gerada uma nova versão do programa onde apenas o bloco básicos suscetíveis são protegidos, ou seja, nessa abordagem as instruções dos blocos básicos não suscetíveis são removidas.

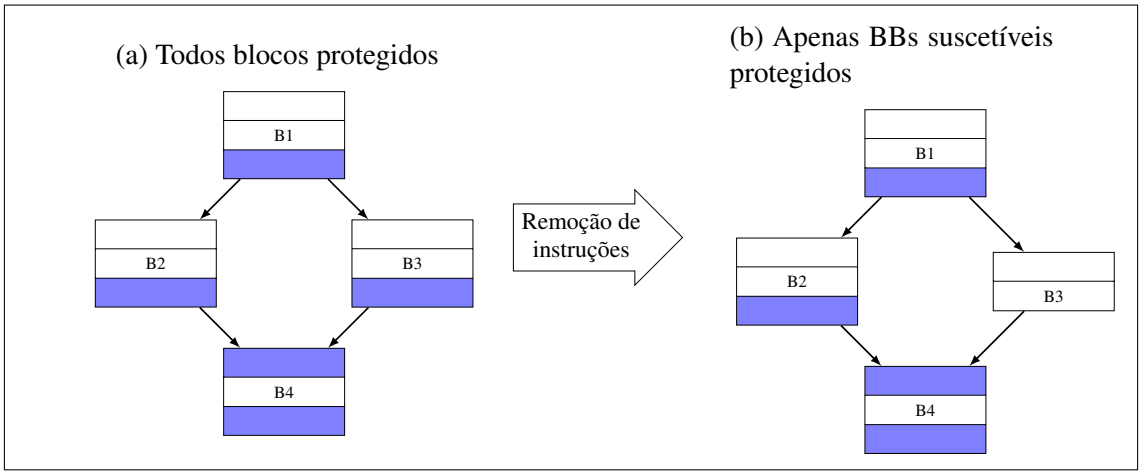
Essa abordagem mostrou-se falha devido a um problema fundamental: Ao removermos as instrução dos blocos básicos não suscetíveis modificamos o espaço de endereçamento do programa. Em outras palavra, modificamos o endereço de memória que cada instruções do programa ocupavam antes da alteração.

A análise sistemática de *single bit-flip* é feita sobre o código x86-64 com todos os endereços de memória resolvidos e com instruções de proteção em todos os blocos básicos. Essa análise permite saber o nome de todos os blocos básicos que precisam ser protegidos com instruções extras. Note, porém, que não é possível acionar novamente o LLVM para que ele, sabendo os nomes dos blocos básicos que não precisam de proteção, gere outro código x86-64 sem as instruções extras de proteção. Isso levaria a mudança nos endereços ocupados pelos blocos básicos e a mudanças nos valores dos operandos das instruções de desvio, o que invalida as informações sobre os blocos básicos suscetíveis obtidas na etapa de análise sistemática de *single bit-flip*

Quando isso acontece, dissemos que ocorreu um desalinhamento entre a versão onde todos os blocos básicos estão protegidos e a versão onde apenas os blocos básicos suscetíveis estão protegidos.

Com o objetivo de evitar o desalinhamento, ao invés de simplesmente removermos

Figura 5.6: Remoção de instruções de detecção de CFEs



as instruções dos blocos básicos não suscetíveis, os endereços de memória onde essas instruções se encontravam foram substituídas por instruções *NOP*.

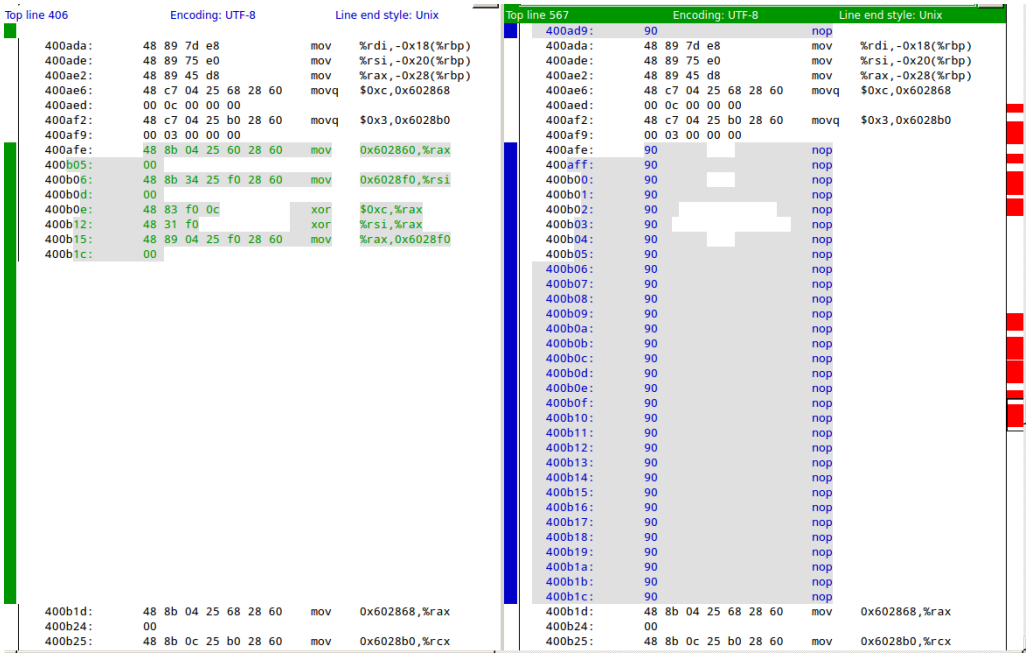
A Figura 5.7 exemplifica o resultado desta abordagem. Nela podemos observar a comparação do trecho de dois arquivos *dump* do mesmo *benchmark*, CRC, que correspondem a um bloco básico não suscetível. À esquerda temos o programa com todos os blocos básicos protegidos pela técnica FCFC. À direita temos a versão parcialmente protegida pela técnica FCFC, onde as instruções de detecção dos blocos básicos não suscetíveis foram substituídas por instruções *NOP*. Perceba que apenas as instruções de proteção (em destaque) foram alteradas, enquanto todas as outras instruções do programa permanecem em suas posições de memória original. Utilizando essa abordagem conseguimos eliminar as instruções de detecção de CFEs dos blocos básicos não suscetíveis e ao mesmo tempo manter o alinhamento entre as diferentes versões, o que garante a validade da análise sistemática de *single bit-flip*.

Como a arquitetura x86-64 é do tipo CISC, as instruções ocupam um espaço variável de memória, por consequência, foi necessário substituir, por exemplo, uma instrução *movq*, por doze (12) instruções *NOP*, pois um *movq* ocupa doze endereços de memória, enquanto um *NOP* ocupa apenas um endereço de memória.

Como dito anteriormente, o LLVM foi a ferramenta utilizada para desenvolver as transformações que permitiram inserir as técnicas de detecção de erros de fluxo de controle nos programas. Apesar de possibilitar a inserção de uma grande variedade de instruções, como instruções de transferência de dados, instruções aritmética e instruções de teste e desvio, o compilador LLVM não permitia a utilização da instrução *NOP* em suas transformações. Portanto, foi necessário modificar o compilador para que fosse possível a utilização desta instrução.

A partir desse ponto chamaremos de *FCFC total* a versão onde todos os blocos básicos estão protegidos pela técnica FCFC, de *FCFC parcial com primeiro refinamento* a versão onde apenas os blocos básicos suscetíveis com o primeiro refinamento estão protegidos, e de *FCFC parcial com segundo refinamento* a versão onde apenas os blocos básicos suscetíveis com o segundo refinamento estão protegidos.

Figura 5.7: Substituição de instruções por *NOPs*



5.5 Avaliação *FCFC parcial com primeiro refinamento*

Após ser realizada a análise sistemática de *single bit-flip* em todos os *benchmarks* utilizados neste trabalho, foi gerado para cada um dele uma versão *FCFC parcial com primeiro refinamento*. O resultado da análise sistemática de *single bit-flip* pode ser observado na Tabela 5.2. Ela mostra a quantidade de blocos básicos na versão original (*FCFC total*) e a versão onde apenas os blocos básicos suscetíveis foram protegidos (*FCFC parcial com primeiro refinamento*). Em média, 11,16% do blocos básico não precisaram ser protegidos, pois não são afetados por desvios ilegais *inter-bloco*.

Tabela 5.2: *FCFC parcial com primeiro refinamento*: Quantidade de blocos básicos suscetíveis protegidos

Benchmark	Blocos básicos protegidos		
	FCFC total	*FCFC parcial com primeiro refinamento*	% não protegidos
basicmath	63	62	1,59%
CRC	20	16	20,00%
dijkstra	60	56	6,67%
FFT	93	88	5,38%
patricia	179	136	24,02%
pbmsrch	34	34	0,0%
qsort	19	17	10,53%
rijndael	186	172	7,53%

Uma vez que todos os blocos básicos suscetíveis foram protegidos, a taxa de detecção de falhas nos *benchmarks* da versão *FCFC parcial com primeiro refinamento* deveria se manter a mesma em relação aos *benchmarks* da versão *FCFC total*. Para comprovar esta hipótese, e assim validar a técnica, foi realizado um novo conjunto de experimentos de injeção de falhas nos *benchmarks* da nova versão para avaliar a taxa de cobertura de falhas.

Mais do que isso, foram injetadas exatamente as mesmas falhas que aquelas injetadas nos *benchmarks* dos experimentos da Tabela 4.1 (*FCFC total*). Isso foi possível graças as características do injetor de falhas desenvolvido nesse trabalho, que permite a reutilização do arquivos de falhas para diferentes versões do mesmo *benchmark*. A Tabela 5.3 mostra o resultado desses experimentos. Com exceção dos *benchmarks* CRC e rijndael, a taxa de detecção de falhas ficou próxima a 100% quando comparada a versão *FCFC total*.

Tabela 5.3: *FCFC parcial com primeiro refinamento*: Taxa de cobertura

Benchmark	S	I	OS	T	EC	EW	D
basicmath	70,5%	14,5%	1,6%	0,0%	2,2%	3,0%	8,2%
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	4,5%	9,5%
dijkstra	79,2%	7,2%	2,2%	0,0%	4,2%	0,0%	7,2%
FFT	78,8%	7,2%	0,1%	0,0%	1,3%	5,7%	6,9%
patricia	73,8%	9,3%	0,1%	0,1%	6,4%	0,9%	9,4%
pbmsrch	76,3%	8,6%	0,1%	0,0%	4,2%	0,2%	10,6%
qsort	74,5%	9,4%	0,0%	0,0%	3,7%	2,7%	9,7%
rijndael	73,8%	9,1%	1,4%	0,0%	4,2%	3,2%	8,3%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

Em relação ao *benchmark* CRC, observamos que ocorreu uma diferença de 2,3 pontos percentuais entre as versões total e parcial. Analisando o *benchmark*, percebemos que as falhas que geravam essa diferença também não eram detectadas pelos blocos básicos suscetíveis na versão *FCFC total* do CRC. Nessa versão essas falhas eram detectadas por um bloco básico sucessor. O que ocorreu foi que nesse caso a falha afetou as instruções de detecção do bloco básico suscetível, inviabilizando a detecção pelo mesmo, mas foi detectada pelo bloco básico sucessor.

Para validar essa constatação foi gerada uma nova versão *FCFC parcial com primeiro refinamento* do *benchmark* CRC onde apenas mais um bloco básico foi protegido, o que reduziu a economia de blocos básicos para 13,04% no *benchmark* CRC, porém garantiu 100% de detecção de falhas em relação à versão original, como podemos ver na Tabela 5.4. No *benchmark* rijndael ocorreu um problema igual ao ocorrido no *benchmark* CRC. Então, com mais um bloco básico protegido foi possível igualar a taxa de cobertura entre as versões *FCFC total* e *FCFC parcial com primeiro refinamento* (resultado na Tabela 5.4). Porém, a taxa de blocos básicos não protegidos caiu para 6,98%.

Tabela 5.4: *FCFC parcial com primeiro refinamento*: Taxa de cobertura - CRC e rijndael corrigidos

Benchmark	S	I	OS	T	EC	EW	D
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	2,2%	11,8%
rijndael	74,0%	9,1%	0,0%	1,4%	4,2%	2,8%	8,5%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

5.6 Avaliação *FCFC parcial com segundo refinamento*

Nas seções anteriores a análise sistemática de *single bit-flip* foi utilizada para identificar todos os blocos básicos suscetíveis e assim diminuir a quantidade de blocos básicos que precisavam ser protegidos pela técnica FCFC, gerando as versões *FCFC parcial com primeiro refinamento*. Porém, além dos blocos básicos não suscetíveis, um outro tipo de bloco básico não precisa, em teoria, ser protegido pela técnica de detecção de erros de fluxo de controle. São os blocos básicos suscetíveis a apenas *desvios errados*.

Como explicado nas Seções 2.1 e 5.2, um erro de fluxo de controle errado ocorre quando ao invés de executar o desvio de um bloco A para um bloco B, sucessor de A, executa um desvio do bloco A para C, também sucessor de A, o qual não era esperado para esse momento da execução. Assim temos o que chamamos de *desvios errados*. Pelo fato do desvio executado estar no fluxo de controle do programa, em teoria, não é tarefa da técnica de detecção de erros de fluxo de controle detectá-lo. FCFC, assim como outras técnicas, não pode detectar este tipo de erro. Portanto, o código protetor para este caso é totalmente desnecessário.

Para evitar estes código desnecessário, a ferramenta que realiza a análise sistemática de *single bit-flip* foi modificada para identificar, além dos blocos básicos suscetíveis, também os *desvios errados*. Com essa nova otimização foi possível diminuir a quantidade de blocos básicos que necessitam ser protegidos. As versões com essa otimização receberam o nome de *FCFC parcial com segundo refinamento*.

A Tabela 5.5 mostra o resultado da análise sistemática de *single bit-flip* e eliminação de blocos básicos afetados por *desvios errados*. Em média, 15,74% do blocos básico não precisaram ser protegidos. Um acréscimo de 4,58 pontos percentuais em relação a versão *FCFC parcial com primeiro refinamento*.

Tabela 5.5: *FCFC parcial com segundo refinamento*: Quantidade de blocos básicos suscetíveis protegidos

Benchmark	Blocos básicos		
	FCFC total	*FCFC parcial com segundo refinamento*	% não protegidos
basicmath	63	60	4,76%
CRC	20	15	25,00%
dijkstra	60	54	10,00%
FFT	93	82	11,83%
patricia	179	129	27,93%
pbmsrch	34	31	8,82%
qsort	19	16	15,79%
rijndael	186	164	11,83%

Para avaliar a taxa de cobertura dos *benchmarks* da versão *FCFC parcial com segundo refinamento*, foi realizado um novo conjunto de experimentos aos mesmos moldes daqueles realizados na Seção 5.5. A Tabela 5.6 mostra o resultado dos experimentos realizados com essa versão e a Tabela 5.7 mostra o resultado das versões CRC e rijndael corrigidas.

Tabela 5.6: *FCFC parcial com segundo refinamento*: Taxa de cobertura

Benchmark	S	I	OS	T	EC	EW	D
basicmath	69,9%	14,5%	1,6%	0,0%	2,4%	3,0%	8,6%
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	4,5%	9,5%
dijkstra	79,2%	7,2%	2,2%	0,0%	4,2%	0,0%	7,2%
FFT	78,5%	7,2%	0,1%	0,0%	1,3%	5,7%	7,2%
patricia	73,3%	8,4%	0,1%	0,1%	8,4%	1,7%	8,0%
pbmsrch	76,3%	8,6%	0,1%	0,0%	4,2%	0,2%	10,6%
qsort	73,9%	8,0%	0,0%	0,0%	3,7%	2,7%	11,7%
rijndael	73,8%	9,1%	1,4%	0,0%	4,2%	3,3%	8,2%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

Tabela 5.7: *FCFC parcial com segundo refinamento*: Taxa de cobertura - CRC e rijndael corrigidos

Benchmark	S	I	OS	T	EC	EW	D
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	2,2%	11,8%
rijndael	74,0%	9,1%	0,0%	1,4%	4,2%	2,9%	8,4%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

5.7 Reposicionamento das instruções de desvio

Os resultados das seções anteriores mostram que a análise sistemática de *single bit-flip* é capaz de detectar os blocos básico suscetíveis e que protegendo apenas esses blocos básicos não perdemos capacidade de detecção de falhas. Entretanto, não foi possível simplesmente remover as instruções dos blocos básicos não suscetíveis. As instruções desses blocos básicos tiveram que ser substituídas por instruções *NOP* para garantir o alinhamento entre as versões. Levando-se em consideração que uma instrução *NOP* executa mais rápido que um *movq*, por exemplo, já teríamos um ganho de desempenho no nosso programa se cada instrução de detecção fosse substituída exatamente por um *NOP*, contudo, por estarmos trabalhando com em uma arquitetura CISC, não foi isso que aconteceu. Foi necessário substituir cada instrução de detecção por várias instruções *NOP*. Esse número variou de instrução para instrução.

Portanto, como apresentada até agora, a técnica não garante a diminuição do *overhead*, pois trocamos a execução de algumas instruções complexas pela execução de muitas instruções *NOP*. A ideia original era minimizar esse problema já na fase de compilação do programa colocando os *NOPs* após as instruções de desvio que caracterizam o fim do bloco básico, assim evitando a execução dos *NOPs* e garantindo o alinhamento entre as versões total e parcial. Porém, isso não foi possível devido a restrições do compilador LLVM, que não permite que instruções *NOP* terminem um bloco básico.

Para contornar essa limitação optamos por trabalhar com o código *assembly* dos programas, onde seria possível simplesmente reposicionar (inverter) os *NOPs*, colocando-os após as instruções de desvio. Desta forma, poderíamos diminuir, ou até mesmo zerar, a quantidade de *NOPs* executados. Para tal, criamos um script em *bash* que simplesmente percorre o código *assembly* de cada *benchmark* e coloca todos os *NOPs* de um determinado bloco básico após a instrução de desvio (*jump*) do bloco básico. Assim, geramos

para cada *benchmark* duas novas versões: 1) *FCFC parcial com primeiro refinamento e NOPs invertidos*, e 2) *FCFC parcial com segundo refinamento e NOPs invertidos*.

Nas seção seguinte apresentamos a avaliação da taxa de cobertura dessas duas novas versões.

5.7.1 Avaliação *FCFC parcial com primeiro refinamento e NOPs invertidos* e *FCFC parcial com segundo refinamento e NOPs invertidos*

Mais uma vez cada nova versão dos *benchmarks* foi submetida a injeção de falhas para verifica a taxa de cobertura aos mesmos moldes das Seções anteriores.

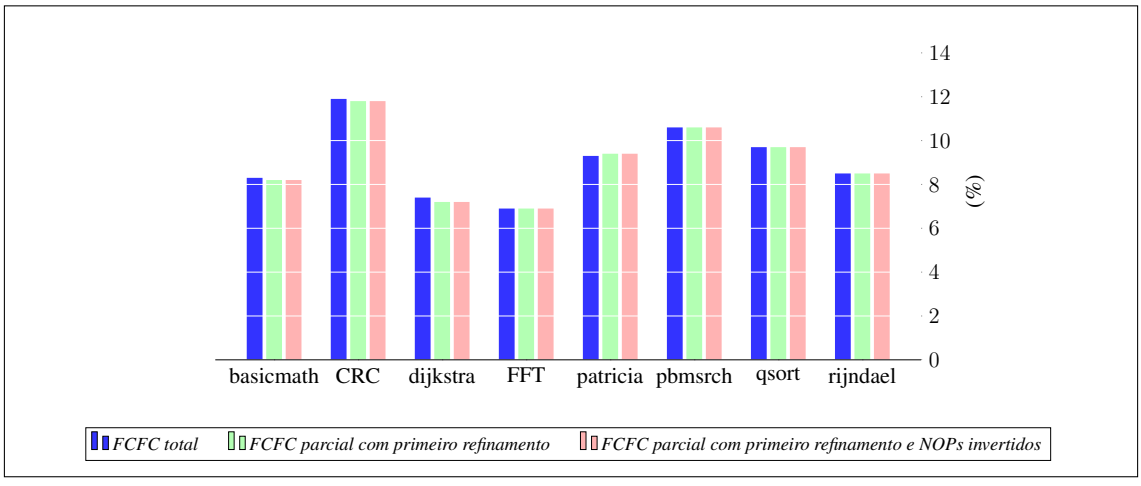
A Tabela 5.8 mostra o resultado dos experimentos com a versão *FCFC parcial com primeiro refinamento e NOPs invertidos*. Já o gráfico da Figura 5.8 mostra um comparativo da taxa de cobertura entre as versões *FCFC total*, *FCFC parcial com primeiro refinamento* e *FCFC parcial com primeiro refinamento e NOPs invertidos*. Como podemos perceber, a taxa de detecção de falhas manteve-se praticamente a mesma entre as diferentes versões.

Tabela 5.8: *FCFC parcial com primeiro refinamento e NOPs invertidos*

Benchmark	S	I	OS	T	EC	EW	D
basicmath	70,5%	14,6%	1,6%	0,0%	2,1%	3,0%	8,2%
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	2,2%	11,8%
dijkstra	79,2%	7,2%	2,2%	0,0%	4,2%	0,0%	7,2%
FFT	79,1%	6,9%	0,1%	0,0%	1,3%	5,7%	6,9%
patricia	73,8%	9,3%	0,1%	0,1%	6,4%	0,9%	9,4%
pbmsrch	76,3%	8,6%	0,1%	0,0%	4,2%	0,2%	10,6%
qsort	74,5%	9,4%	0,0%	0,0%	3,7%	2,7%	9,7%
rijndael	74,0%	9,1%	1,4%	0,0%	4,2%	2,8%	8,5%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

Figura 5.8: Comparação da taxa de cobertura de falhas



A Tabela 5.9 mostra o resultado dos experimentos com a versão *FCFC parcial com segundo refinamento e NOPs invertidos*. Já o gráfico da Figura 5.9 mostra um comparativo

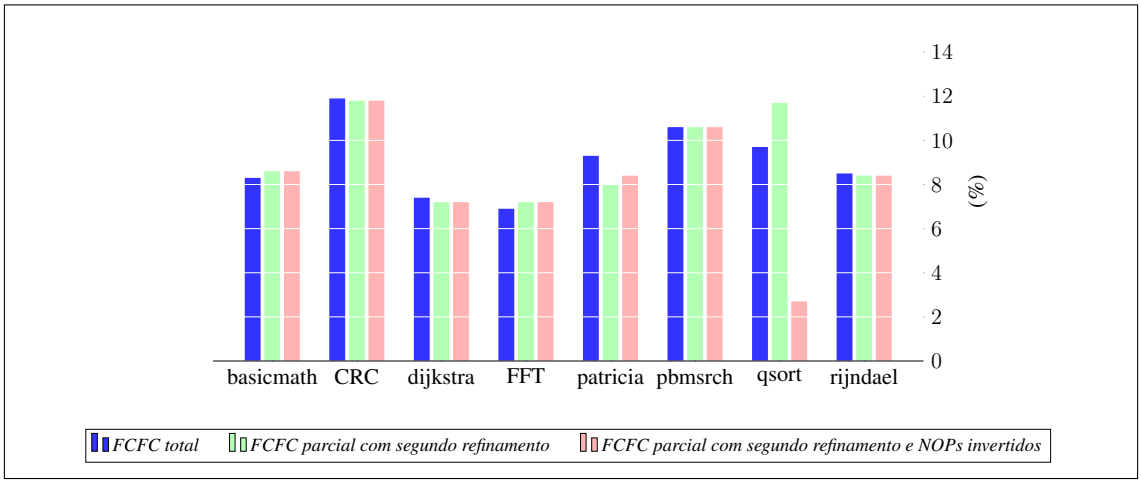
da taxa de cobertura entre as versões *FCFC total*, *FCFC parcial com segundo refinamento* e *FCFC parcial com segundo refinamento e NOPs invertidos*. Como podemos perceber, a taxa de detecção de falhas manteve-se praticamente a mesma entre as diferentes versões.

Tabela 5.9: *FCFC parcial com segundo refinamento e NOPs invertidos*

Benchmark	S	I	OS	T	EC	EW	D
basicmath	69,9%	14,6%	1,6%	0,0%	2,3%	3,0%	8,6%
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	2,2%	11,8%
dijkstra	79,2%	7,2%	2,2%	0,0%	4,2%	0,0%	7,2%
FFT	78,8%	6,9%	0,1%	0,0%	1,3%	5,7%	7,2%
patricia	73,2%	8,6%	0,1%	0,1%	8,0%	1,6%	8,4%
pbmsrch	76,3%	8,6%	0,1%	0,0%	4,2%	0,2%	10,6%
qsort	73,9%	8,0%	0,0%	0,0%	8,4%	7,0%	2,7%
rijndael	73,6%	9,1%	1,4%	0,0%	4,7%	2,8%	8,4%

S - Segmentation fault. I - Illegal instrucion. OS - Other faults. T - Timeout.
EC - Exited correct. EW - Exited wrong. D - Detected

Figura 5.9: Comparação da taxa de cobertura de falhas



5.8 Problemas encontrados

Nesta seção descrevemos problemas encontrados ao realizar a inserção de *NOPs* nos programas para remover instruções de detecção e manter o alinhamento, e ao inverter a posição das instruções de desvios a fim de evitar a execução das instruções *NOP*. É importante ressaltar que os experimentos realizados ao longo deste trabalho mostram os resultados das versões já corrigidas, ou seja, os problema foram detectados e corrigidos antes da realização dos experimentos.

5.8.1 *FCFC parcial com primeiro refinamento*

A remoção de instruções dos blocos básicos e substituição por *NOPs* provocou um efeito não previsto. Pelo fato de haver menos instruções em alguns blocos básicos o compilador modificou a alocação de registradores entre a versão da técnica *FCFC total*, onde todos os blocos básicos estão protegido, e a versão *FCFC parcial com primeiro refinamento*, onde apenas os blocos básicos suscetíveis estão protegidos. Com isso uma instrução que na versão *FCFC total* utilizava, por exemplo, o registrador *edx*, e ocupava 6 posições de memória, na versão parcial passou a utilizar o registrador *eax*, e a utilizar 5 posições de memória. Como consequência, ocorre um desalinhamento entre as versões, e a análise sistemática de *single bit-flip* é invalidada.

A Figura 5.10 mostra a comparação do *dump* de duas versões do *benchmark basicmath* para exemplificar este problema. Observe que na posição de memória *40204b* do arquivo à esquerda (versão *FCFC total*) a instrução *cmp* utiliza o registrador *edx* e ocupa 6 posições de memória. Já na versão à direita (*FCFC parcial com primeiro refinamento*) a instrução na posição de memória 40204b é a mesma, *cmp*, porém ela utiliza o registrador *eax* e 5 posições de memória. Por consequência todas as instruções posteriores à instrução *cmp* ficaram desalinhadas entre as duas versões, ou seja, a análise sistemática de *single bit-flip* é invalidada.

Sendo assim, foi necessário identificar todos os *benchmarks* onde esse problema ocorreu e gerar novas versões corrigidas. Para gerar essas novas versões criamos um arquivos com informações que diziam ao compilador quantos *NOPs* a mais (além daqueles necessários para substituir as instruções de detecção de erros de fluxo de controle que estavam sendo eliminadas) deveriam ser inseridos em um determinado bloco básico para garantir o alinhamento. De posse deste arquivos o compilador gerou as novas versões, agora corretamente alinhadas.

A Figura 5.11 mostra o resultado para o *benchmark basicmath*. Mais uma vez temos a comparação entre as duas versões. À esquerda a versão *FCFC total* e à direita a versão *FCFC parcial com primeiro refinamento*. Como podemos perceber, a partir da instrução no endereço de memória *40204b* todas as demais instruções ficaram alinhadas entre as duas versões. A instrução do endereço *40204b*, propriamente dita, ficou desalinhada. Mas isso não representa um problema , visto que este é um bloco básico não suscetível.

O problema relatado também ocorreu na versão *FCFC parcial com segundo refinamento*, onde a correção foi feita nos mesmos moldes da versão *FCFC parcial com primeiro refinamento*.

Figura 5.10: Desalinhamento causado pela realocação de registradores

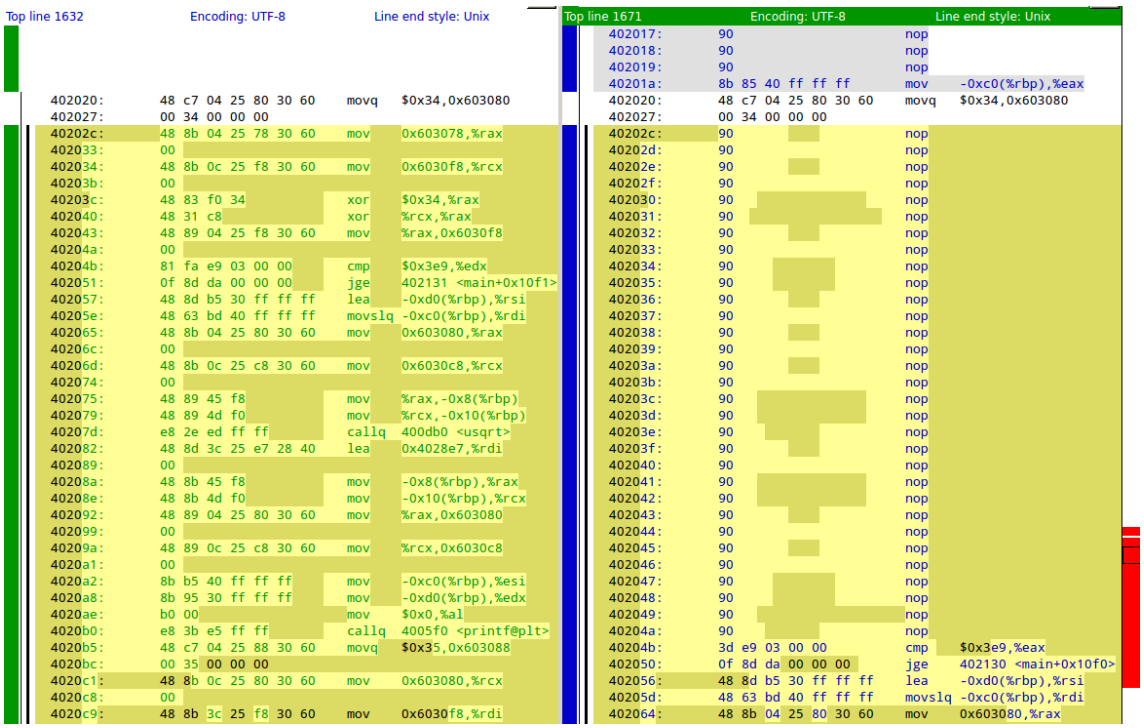
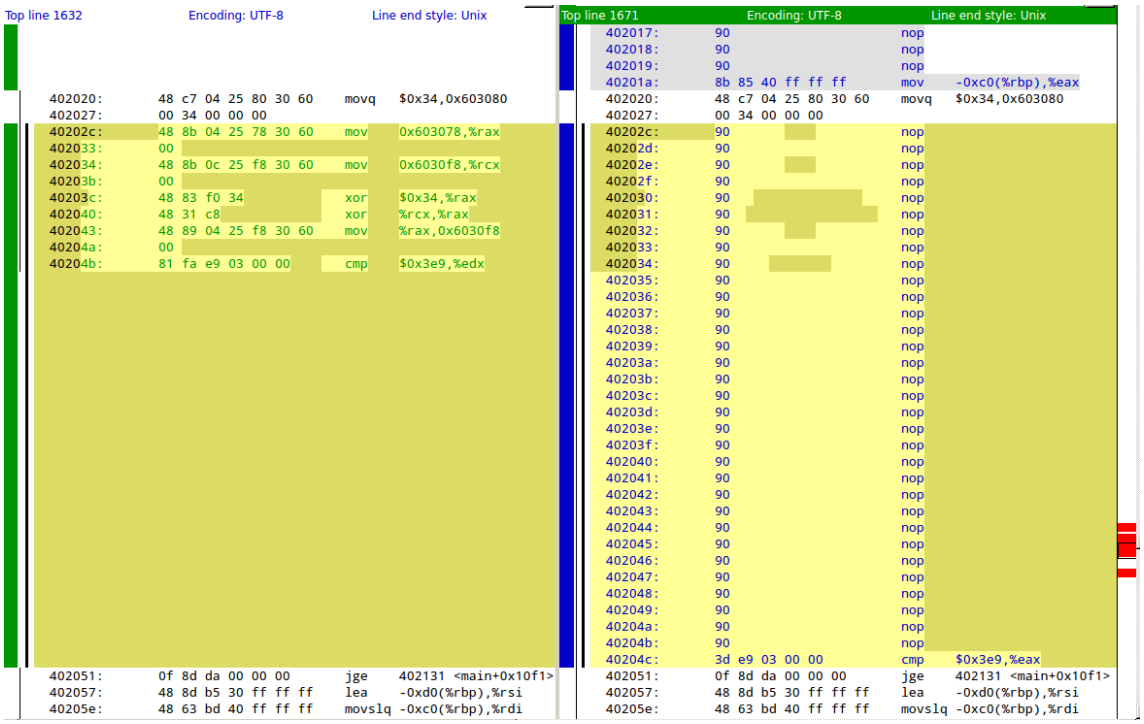


Figura 5.11: Correção do desalinhamento causado pela realocação de registradores



5.8.2 *FCFC parcial com primeiro refinamento e NOPs invertidos*

Mais uma vez a alteração no código - nesse caso o reposicionamento das instruções de desvio - provocou um desalinhamento entre as diferentes versões dos *benchmarks*. Aqui não houve alteração da alocação de registradores, pois no *assembly* do programa todos

os registradores já estão definidos. O caso aqui foi que, para uma mesma instrução de desvio, foi utilizada uma quantidade diferente de memória.

Por exemplo, no *benchmark* rijndael, uma instrução de desvio condicional (JE) que ocupava 2 endereços de memória passou a ocupar 6 endereços de memória. Ou seja, deixou de ser um *Jump short* e passou a ser um *Jump near*. Isso ocorre porque a instrução de desvio é sensível à posição do bloco básico destino do desvio. Se este estiver mais distante, no que diz respeito ao endereço de memória, será utilizada um instrução que utiliza mais memória para o endereçamento, caso contrário, será utilizada uma instrução que utiliza menos memória. O interessante é que no código *assembly* não há diferenciação entre *Jump short* e *Jump near*, pois ele trabalha apenas com mnemônicos. Apenas quando o código é traduzido para a arquitetura alvo é que os endereços de memória são definidos, assim como a utilização da instrução de desvio adequada.

Para solucionar esse problema foi necessário analisar cada *benchmark* individualmente, detectar onde o problema ocorria, e inserir mais ou menos *NOPs*, de acordo com a necessidade, no código *assembly* de cada *benchmark* da versão *FCFC parcial com primeiro refinamento e NOPs invertidos* antes que ele fosse compilado para arquitetura alvo. Esse correção foi deita manualmente, pois ocorreu em apenas alguns *benchmarks*.

O problema relatado também ocorreu na versão *FCFC parcial com segundo refinamento e NOPs invertidos*, onde a correção foi feita nos mesmos moldes da versão *FCFC parcial com primeiro refinamento e NOPs invertidos*.

6 AVALIAÇÃO DE DESEMPENHO

No Capítulo 5 mostramos que através da análise sistemática de *single bit-flip* é possível identificar os blocos básicos suscetíveis e que ao protegermos apenas esses blocos não perdemos capacidade de detecção de falhas, quando comparamos as diferentes versões dos *benchmarks* protegidos pela técnica FCFC. Mas além de manter a taxa de detecção de erros de fluxo de controle, o objetivo da técnica desenvolvida neste trabalho é reduzir o *overhead* imposto pelas técnicas de detecção de erros de fluxo de controle. Neste capítulo iremos mostrar os dados referentes ao desempenho dos *benchmarks* utilizado neste trabalho protegidos pelas diferentes versões da técnica FCFC, assim como uma comparação de desempenho entre essas versões. No final do capítulo apresentamos uma perspectiva em relação à utilização da técnica em um ambiente RISC.

6.1 Metodologia para avaliação de desempenho

Neste trabalho todos os *benchmarks* utilizados foram protegidos com a mesma técnica de detecção de erros de fluxo de controle, FCFC, variando a quantidade de blocos básicos protegidos e a forma que eles foram protegidos, o que por sua vez gerou diferentes versões da técnica. Neste capítulo o objetivo é avaliar o desempenho destas diferentes versões e compará-las. Para a melhor compreensão das versões existentes, na Tabela 6.1 apresentamos um breve resumo de cada uma delas.

Como veremos a seguir, a diferença de desempenho entre algumas versões dos *benchmarks* foi muito pequena. Devido a isso, medir o desempenho em termos de unidade de tempo mostrou-se infrutífero, pois não foi possível demostrar de forma precisa qualquer ganho ou perda utilizando-se desse expediente. Sendo assim, optamos por avaliar o desempenho dos *benchmarks* em termos do número de instruções executadas por cada um deles.

Para obter todas as instruções executadas por uma determinada versão de um *benchmark* e para contabilizar a quantidade de vezes que cada instrução foi executada, foi realizada a extração do *trace* de todas as versões de todos os *benchmarks*. O *trace* é um arquivo texto que guarda os dados referente a todas as instruções executadas e a quantidade de vezes que cada uma delas foi executada. Para extrair o *trace* foi utilizado o gdb (GNU debugger). Maiores detalhes sobre como o trace dos programas é extraído podem ser obtidos na Seção B.1 do Apêndice B.

Tabela 6.1: Diferentes versões da técnica *FCFC*

Nome	Descrição
FCFC total	Todos os blocos básicos protegidos
FCFC parcial com primeiro refinamento	Apenas blocos básicos suscetíveis com primeiro refinamento protegidos
FCFC parcial com primeiro refinamento e NOPs invertidos	Apenas blocos básicos suscetíveis com primeiro refinamento protegidos e instruções *NOP* invertidas
FCFC parcial com segundo refinamento	Apenas blocos básicos suscetíveis com segundo refinamento protegidos
FCFC parcial com segundo refinamento e NOPs invertidos	Apenas blocos básicos suscetíveis com segundo refinamento protegidos e instruções *NOP* invertidas

6.2 *FCFC total* vs *FCFC parcial com primeiro refinamento*

O primeiro *trace* extraído dos *benchmarks* foi o da versão *FCFC total*. A Tabela 6.2 mostra o resultado da extração do *trace*, onde podemos observar o total de instruções executadas por cada um dos *benchmarks*.

Tabela 6.2: Desempenho: *FCFC total*

Benchmarks	Número de instrução executadas
basicmath	7,422,666
CRC	54,754,761
dijkstra	192,411,876
FFT	7,661,798
patricia	16,667,614
pbmsrch	542,441
qsort	6,854,311
rijndael	47,002,968

A Tabela 6.3 mostra o resultado do trace da versão *FCFC parcial com primeiro refinamento*. Nessa tabela, além do total de instruções executadas por cada *benchmark*, observamos o total de instruções *NOP* executadas. Ou seja, dentre o total de instruções executadas, quantas execuções representam instruções *NOP*.

A primeira observação que podemos fazer é que o total de instrução executadas aumentou em relação à versão *FCFC total*. Isso já era previsto e se explica pelo fato de que a substituição de instrução por *NOPs* não foi do tipo "um-para-um". Como explicado na Seção 5.4, por estarmos trabalhando em um ambiente CISC, foi necessário substituir as instruções de detecção de falhas por vários *NOPs*, por exemplo, uma instrução *movq* foi substituída por 12 *NOPs*, para garantir o alinhamento.

A segunda observação diz respeito a quantidade de blocos básicos não suscetíveis executados. Na Tabela 5.2 observamos que, por exemplo, o *benchmark rinjdael* possui 172

Tabela 6.3: Desempenho: *FCFC parcial com primeiro refinamento*

Benchmarks	Instrução executadas	*NOPs* executados	BB *não suscetíveis* executados
basicmath	7.497.816	87.174	1
CRC	54.754.798	43	1
dijkstra	304.335.026	130.072.850	2
FFT	7.661.798	0	0
patricia	16.758.042	105.092	2
pbmsrch	542.441	0	0
qsort	11.310.369	5.178.662	1
rijndael	48.445.256	1.676.173	3

blocos básicos suscetíveis, de um total de 186 blocos básicos, portanto, 14 blocos básicos são nãos suscetíveis e não precisaram ser protegidos pela técnica FCFC. Porém, desses 14 blocos básicos apenas 3 foram executados, devido a entrada utilizada pelo *benchmark*, ou seja, 21,42% dos blocos básicos não suscetíveis. No caso do *benchmark patricia* a diferença foi ainda mais gritante. Dos 43 blocos básicos não suscetíveis, apenas 3 foram executados, 6,97% somente. Já o *benchmark* FFT não teve nenhum bloco básico não suscetível executado, o que fez com que o não houvesse mudança no número de instruções executadas.

A obtenção da informação sobre a quantidade de blocos básicos executados foi possível graças a extração do trace, que, além da quantidade de instruções, informa o endereço de memória de cada uma delas, o que possibilita identificar os blocos básicos executados.

6.3 *FCFC parcial com primeiro refinamento* vs *FCFC parcial com primeiro refinamento e NOPs invertidos*

Depois de avaliado o desempenho da versão *FCFC parcial com primeiro refinamento*, o próximo passo foi avaliar a versão *FCFC parcial com primeiro refinamento e NOPs invertidos* comparando-a com a primeira. O resultado do trace extraído dos *benchmarks* dessa versão podem ser visto na Tabela 6.4.

Como dito na Seção 5.7, na versão *FCFC parcial com primeiro refinamento e NOPs invertidos* as instruções *NOP* foram reposicionadas e colocadas após a instrução de fim do bloco básico (instrução de desvio - *jump*), com o objetivo de reduzir, ou até mesmo eliminar, a execução dos *NOPs*. Como sabemos, o comportamento de uma instrução de desvio é da seguinte forma:

Se for um desvio incondicional, toda vez que o *programa counter* atinge essa instrução, o fluxo de controle passa a apontar para o endereço de memória que está no operando da instrução. Se for um desvio condicional, se a condição for falsa/verdadeira, isso irá dependendo do tipo do desvio condicional, o fluxo de controle passa a apontar para o endereço de memória que está no operando da instrução, senão, o fluxo de controle aponta para a instrução imediatamente posterior ao desvio.

Como podemos deduzir a partir do comportamento das instruções de desvio condicional e pelos resultados da Tabela 6.4 não foi possível eliminar totalmente a execução dos *NOPs*. Apesar disso, o fato é que ocorreu a redução de execução de *NOPs* em todos os *benchmarks*, mesmo que em alguns casos a redução tenha sido baixa, quando comparamos com a versão *FCFC parcial com primeiro refinamento*.

Tabela 6.4: Desempenho: *FCFC parcial com primeiro refinamento e NOPs invertidos*

Benchmarks	Instrução executadas	*NOPs* executados	BB não suscetíveis executados
basicmath	7.497.753	87.111	1
CRC	54.754.755	0	1
dijkstra	303.406.576	129.144.400	2
FFT	7.661.798	0	0
patricia	16.744.092	91.142	2
pbmsrch	542.441	0	0
qsort	8.763.264	2.631.557	1
rijndael	47.236.876	467.793	3

O *benchmark* CRC, por exemplo, que na versão *FCFC parcial com primeiro refinamento* executou 43 *NOPs*, já na versão *FCFC parcial com primeiro refinamento e NOPs invertidos* não executou *NOPs*. Por sua vez o *benchmark basicmath* teve uma redução de apenas 0,072% no número de *NOPs* executados, enquanto que o *benchmark rijndael* teve uma redução de 72,09%, a maior redução percentual se não considerarmos o CRC. Em valor absoluto, a maior redução foi do *benchmark qsort*, onde ocorreu a redução de 2.547.105 execuções de *NOPs*.

Quando consideramos o número total de instruções executadas, percebemos que o ganho em termos de redução de execuções foi baixo para a maioria dos *benchmarks*. O *benchmark* CRC apresentou o menor ganho (0,000078%), seguido dos *benchmarks basicmath* (0,00084%), *patricia* (0,083%), *dijkstra* (0,30%), *rijndael* (2,49%) e *qsort* (22,52%), que apresentou o maior ganho. Os *benchmarks* FFT e *pbmsrch* executaram exatamente as mesmas instruções, pois para o primeiro não ocorreu a execução dos blocos básicos *não suscetíveis*, já o segundo não possuía esse tipo de bloco básico. A Tabela 6.5 mostra o resumo dos valores.

Tabela 6.5: Percentual de redução de instruções executadas

Benchmarks	% em relação ao total de *NOPs*	% em relação ao total de instruções
basicmath	0,0722%	0,0008%
CRC	100,00%	0,00007%
dijkstra	0,7137%	0,305%
FFT	-	-
patricia	13,274%	0,0832%
pbmsrch	-	-
qsort	49,1846%	22,52%
rijndael	72,0916%	2,4943%

6.4 *FCFC parcial com segundo refinamento* vs *FCFC parcial com segundo refinamento e NOPs invertidos*

A versão *FCFC parcial com segundo refinamento* aumentou a quantidade de blocos básicos *não suscetíveis* conforme visto na Tabela 5.5, por consequência aumentou a quantidade de blocos básicos que não precisam de proteção contra erros de fluxo de controle, o que potencialmente aumentaria a quantidade de blocos básicos *não suscetíveis* executados, o que, por sua vez, poderia melhorar o desempenho em relação à versão *FCFC parcial com primeiro refinamento*.

Com podemos ver na Tabela 6.6 confirmou-se a expectativa de execução de mais blocos básicos não suscetíveis, porém o número total de instruções (*NOP* ou não-*NOP*) executadas aumentou em todos os casos. Mais uma vez, isso se explica pelo fato de ser necessário trocar uma instrução de detecção por vários *NOPs*, como mais instruções foram trocadas por *NOPs*, o número total de instruções executadas aumentou, assim como o número total de *NOPs* executados.

Tabela 6.6: Desempenho: *FCFC parcial com segundo refinamento*

Benchmarks	Instrução executadas	*NOPs* executados	BB *não suscetíveis* executados
basicmath	8,247,656	981,214	3
CRC	54,754,824	74	2
dijkstra	304,335,572	130,073,501	4
FFT	7,874,842	254,014	5
patricia	25,302,470	10,047,395	7
pbmsrch	544,992	2,978	3
qsort	12,901,543	7,075,831	2
rijndael	48,445,334	1,676,266	5

A Tabela 6.7 mostra o resultado do avaliação de desempenho da versão *FCFC parcial com segundo refinamento e NOPs invertidos*. Como podemos perceber, houve um ganho em relação à versão *FCFC parcial com segundo refinamento*, pois ocorreu uma redução no total de instruções executadas em todos os *benchmarks*.

Tabela 6.7: Desempenho:*FCFC parcial com segundo refinamento e NOPs invertidos*

Benchmarks	Número de instrução executadas	Número de *NOPs* executados
basicmath	8,247,593	981,151
CRC	54,754,750	0
dijkstra	303,406,471	129,144,400
FFT	7,874,811	253,983
patricia	24,544,892	9,289,817
pbmsrch	544,124	2,110
qsort	8,457,269	2,631,557
rijndael	47,236,892	467,824

A Tabela 6.8 mostra o ganho em relação à quantidade de *NOPs* e em relação ao total de instruções executadas.

Tabela 6.8: Percentual de redução de instruções executadas

Benchmarks	% em relação ao total de *NOPs*	% em relação ao total de instruções
basicmath	0,0064%	0,0007%
CRC	100,00%	0,0001%
dijkstra	0,7142%	0,3052%
FFT	0,0122%	0,0003%
patricia	7,5400%	2,994%
pbmsrch	29,147%	0,1592%
qsort	62,8092%	34,4476%
rijndael	72,0913%	2,4944%

6.5 Estimativa para utilização em um sistema RISC

Como pudemos observar nos resultados das seções anteriores foi possível substituir instruções complexas por instruções simples do tipo *NOP*, tanto no primeiro quanto no segundo refinamento. E quando os *NOPs* foram invertidos evitamos a execução de diversas instruções desse tipo. Porém, o ganho não foi tão grande quanto o esperado. E quando compararmos com a versão *FCFC total*, onde todos os blocos básicos estão protegidos, não houve redução no total de instruções executadas. Pelo contrário, houve aumento em todos os casos. Isso se deu porque trabalhamos em um ambiente CISC, onde as instruções não possuem um tamanho regular. Sendo assim, sempre que substituímos uma instruções por *NOPs* foi necessário substituir essas instruções não por um, mas por muitos *NOPs*.

Para estimar as possibilidades de utilização da proteção de blocos básicos suscetíveis, através da análise sistemática de *single bit-flip*, em um ambiante RISC, calculamos a substituição de instruções por *NOPs*, como se estivéssemos trabalhando nesse ambiente, ou seja, calculamos como seriam os resultados se cada instrução pudesse ser substituída por um único *NOP*.

A Tabela 6.9 mostra o resultado das estimativa da substituição de instruções por *NOPs*. A segunda coluna mostra a quantidade de instruções executadas na versão *FCFC total*. A terceira coluna mostra quantas destas instruções (execuções) poderiam ser trocadas por execuções de *NOPs* na versão *FCFC parcial com primeiro refinamento*. Já a quarta coluna mostra quantos *NOPs* seriam executados na versão *FCFC parcial com primeiro refinamento e NOPs invertidos*.

Tabela 6.9: Estimativa para um ambiente RISC, *FCFC parcial com primeiro refinamento*e *FCFC parcial com primeiro refinamento e NOPs invertidos*

Benchmark	Instruções executadas na versão *FCFC total*	Estimativa de *NOPs* executados na versão *FCFC parcial com primeiro refinamento*	Estimativa de *NOPs* executados na versão *FCFC parcial com primeiro refinamento e NOPs invertidos*
basicmath	7,422,666	12.024	12.014
CRC	54,754,761	6	0
dijkstra	192,411,876	18.149.700	17.999.950
FFT	7,661,798	0	0
patricia	16,667,614	14.664	12.414
pbmsrch	542,441	0	0
qsort	6,854,311	722.604	426.429
rijndael	47,002,968	233.885	38.985

Por exemplo, para o *benchmark qsort*, das 7.422.666 instrução executadas na versão *FCFC total*, 722.604 poderiam ser substituídas por *NOPs* na versão *FCFC parcial com primeiro refinamento*, um total de 10,54% das instruções. Já na versão *FCFC parcial com primeiro refinamento e NOPs invertidos* 426.429 *NOPs* seriam executados, ou seja, do

total de 7.422.666, 296.175 não seriam executadas, um ganho de 4,32%, enquanto que 6,21% do total de instruções seriam trocadas por *NOPs*. A Tabela 6.10 mostra o resumo da estimativa do percentual de instruções salvas e percentual de instruções trocadas por *NOPs* na versão *FCFC parcial com primeiro refinamento e NOPs invertidos*.

Tabela 6.10: Estimativa *FCFC parcial com primeiro refinamento e NOPs invertidos*

Benchmarks	% de instruçõs trocas por *NOPs*	% instruções salvas
basicmath	0,162%	0,00013%
CRC	0,00%	0,00001%
dijkstra	9,355%	0,0778%
FFT	0,00%	0,00%
patricia	0,074%	0,0135%
pbmsrch	0,00%	0,00%
qsort	6,221%	4,321%
rijndael	0,083%	0,414%

A Tabela 6.11 mostra o resultado das estimativa da substituição de instruções por *NOPs* para as demais versões. A segunda coluna mostra a quantidade de instruções executadas na versão *FCFC total*. A terceira coluna mostra quantas destas instruções (execuções) que poderiam ser trocadas por execuções de *NOPs* na versão *FCFC parcial com segundo refinamento*. Já a quarta coluna mostra quantos *NOPs* seriam executados na versão *FCFC parcial com segundo refinamento e NOPs invertidos*. Já a Tabela 6.12 mostra o resumo da estimativa do percentual de instruções salvas e percentual de instruções trocadas por *NOPs* na versão *FCFC parcial com segundo refinamento e NOPs invertidos*.

Tabela 6.11: Estimativa para um ambiente RISC, *FCFC parcial com segundo refinamento*e *FCFC parcial com segundo refinamento e NOPs invertidos*

Benchmark	Instruções executadas na versão *FCFC total*	Estimativa de *NOPs* executados na versão *FCFC parcial com segundo refinamento*	Estimativa de *NOPs* executados na versão *FCFC parcial com segundo refinamento e NOPs invertidos*
basicmath	7.422.666	156.224	156.214
CRC	54.754.761	11	0
dijkstra	192.411.876	18.149.805	17.999.950
FFT	7.661.798	40.970	40.965
patricia	16.667.614	1.412.539	1.290.349
pbmsrch	542.441	427	287
qsort	6.854.311	1.028.599	367.194
rijndael	47.002.968	233.900	38.995

Tabela 6.12: Estimativa *FCFC parcial com segundo refinamento e NOPs invertidos*

Benchmarks	% de instruçõs trocas por *NOPs*	% instruções salvas
basicmath	2,105%	0,00013%
CRC	0,00%	0,00002%
dijkstra	9,355%	0,07788%
FFT	0,535%	0,00007%
patricia	7,742%	0,73310%
pbmsrch	0,053%	0,02581%
qsort	5,357%	9,64947%
rijndael	0,083%	0,41467%

7 CONCLUSÃO

Sistemas computacionais estão constantemente expostos a eventos externos, de ordem natural ou artificial, que podem provocar comportamentos inesperados devido à ocorrência de falhas. Parte dessas falhas não persistem durante toda a execução do sistema, sendo conhecidas como falhas transientes. Falhas transientes podem causar uma sequência incorreta na execução das instruções do programas, o que chamamos de erros de fluxo de controle (*Control Flow Errors* - CFEs).

Para a detecção de erros de fluxo de controle, existem técnicas baseadas em hardware, voltadas para aplicação onde os custos para o desenvolvimento de *chips* não são considerados um fator de impedimento, pois o principal objetivo é obter a maior confiabilidade possível, tais como sistemas bancários e sistemas voltados para a aviação. Também existem técnicas baseadas em software voltadas para aplicações de baixo custo em sistemas embarcados. Essas técnicas buscam atender aplicações onde o custo e desempenho são importantes. Essas técnicas são conhecidas como *Software Implemented Hardware Fault Tolerance* - SIHFT, ou ainda, *Algorithm Based Fault Tolerance* - ABFT. (GOLOUBEVA; REBAUDENGO; REORDA; VIOLANTE, 2006)

As técnicas baseadas em software para a detecção e erros de fluxo de controle consistem geralmente na manutenção e atualização, em tempo de execução, de variáveis que refletem o fluxo de controle do programa. Através do monitoramento dessas variáveis é possível detectar violações no fluxo de controle do programa. Essas técnicas são conhecidas como *signature monitoring* ou *signature checking*. Porém, a adoção de uma técnica baseada em *signature checking* implica na introdução de código adicional específico (*overhead*) no programa. Esse código consiste de instruções para o monitoramento e detecção de qualquer violação que ocorra no fluxo de controle. Todavia, as técnicas para detecção de erros de fluxo de controle introduzem, em tempo de compilação, código extra em todos os blocos básicos do programa. Esse código extra em todos os blocos básicos é responsável pela detecção dos CFEs e também é responsável pelo *overhead* associados às técnicas de detecção de CFEs.

Este trabalho teve como objetivo principal mostrar que, considerando o modelo de falhas de um *bit-flip* no operando de uma instrução de desvio, é possível, através da análise sistemática de *single bit-flip*, detectar os blocos básicos suscetíveis do programa e restringir as instruções de detecção de erros de fluxo de controle para apenas esses blocos básicos, e assim reduzir o *overhead* imposto pela técnica de detecção de erros de fluxo de controle.

Para alcançar esse objetivo, primeiramente adaptamos a técnica *Control Flow Checking by Software Signatures (CFCSS)*, de tal forma que ela tivesse as propriedades necessárias para a realização da análise sistemática de *single bit-flip*, gerando a técnica *Flexible Control Flow Check (FCFC)*. As propriedades em questão são: a) atualização local de

assinaturas, e b) manter o grafo de fluxo de controle do programa inalterado. Além de ter as propriedades necessárias, a técnica FCFC mostrou ter eficiência para detecção de erros de fluxo de controle comparável às técnicas CFCSS e *Control-Flow Error Detection Using Assertions (CEDA)*, o que possibilitou sua utilização na etapa seguinte.

Na etapa seguinte, foi desenvolvida a detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip*. A técnica possui diferentes passos que têm por objetivo detectar os blocos básicos suscetíveis do programa para que apenas esses blocos básicos sejam protegidos. Na passo da análise sistemática de *single bit-flip* é possível detectar todos os blocos básicos suscetíveis do programa. No passo seguinte, esses blocos básicos passam por dois refinamentos. No primeiro refinamento (*FCFC parcial com primeiro refinamento*) são removidos da lista os blocos básicos afetados apenas por CFEs *intra-bloco*. No segundo refinamento (*FCFC parcial com segundo refinamento*) são removidos os blocos básicos afetados apenas por *desvios errados*. Uma vez detectados os blocos básicos suscetíveis, foi possível restringir as instruções de detecção de erros de fluxo de controle para apenas esses blocos básico. Desta forma, reduzindo a quantidade de blocos básicos protegidos no programa.

Na etapa de avaliação da taxa de cobertura, cada refinamento foi avaliado individualmente, tanto no que diz respeito a capacidade de detectar falhas (taxa de cobertura de erros) quanto no ganho de desempenho.

No primeiro refinamento, foi possível reduzir a quantidade de blocos básicos protegidos em uma taxa que variou entre 1,59% e 24,02%, o que nos deu uma média de 11,16% de blocos básicos que não precisaram ser protegidos. Com relação à taxa de cobertura do primeiro refinamento, os experimentos mostraram que todos os *benchmarks* tiveram uma taxa de cobertura de falhas próxima a 100% em relação ao *baseline FCFC total* (versão com todos os blocos básicos protegidos).

No segundo refinamento, foi possível reduzir a quantidade de blocos básicos protegidos em uma taxa que variou 4,76% e 27,93%, o que nos deu uma média de 15,74% de blocos básicos que não precisaram ser protegidos. No segundo refinamento, os experimentos mostraram que a taxa de cobertura de falhas também ficou próxima a 100% em relação ao *baseline FCFC total*.

Com a detecção dos blocos básicos suscetíveis, foi possível diminuir a quantidade de blocos básicos protegidos no programas. Porém, para garantir o alinhamento, não foi possível simplesmente remover as instruções de detecção dos blocos básicos não suscetíveis, essas instruções tiveram que ser substituídas por instruções *NOP*. Isso afetou o objetivo inicial do trabalho, que era o de diminuir o *overhead*, pois trocamos a execução de algumas instruções pela execução de vários *NOPs*. Para tentar evitar a execução das instruções *NOP* e assim e assim tentar diminuir o *overhead* decidimos inverter a posições dos *NOPs* em relação às instruções de desvio do programa, criando duas novas versões: *FCFC parcial com primeiro refinamento e NOPs invertidos*á e *FCFC parcial com segundo refinamento e NOPs invertidos*.

Com relação a taxa de cobertura de falhas a versão *FCFC parcial com primeiro refinamento e NOPs invertidos* teve resultados iguais aos da versão *FCFC parcial com primeiro refinamento*, ficando com a taxa de cobertura inalterada. Já a taxa de cobertura da versão *FCFC parcial com segundo refinamento e NOPs invertidos* teve uma pequena variação em relação à versão *FCFC parcial com segundo refinamento*. Essa variação ocorreu nos *benchmarks patricia* e *qsort*. Nos demais *benchmarks* a taxa de cobertura ficou igual.

Na etapa seguinte, avaliou-se o desempenho de cada uma das quatro versões dos *benchmarks* com relação ao *baseline*, versão *FCFC total*. A diferença de desempenho entre

algumas versões dos *benchmarks* foi muito pequena. Devido a isso, medir o desempenho em termos de unidade de tempo mostrou-se infrutífero, pois não foi possível demostrar de forma precisa qualquer ganho ou perda de desempenho utilizando-se desse expediente. Sendo assim, optamos por avaliar o desempenho dos *benchmarks* em termos do número de instruções executadas por cada um deles.

Os resultados da avaliação de desempenho mostraram que o ganho não foi na mesma proporção da quantidade de blocos básicos não protegidos, na realidade, ele ficou abaixo do esperados.

Quando comparamos a versão *FCFC total* com as versões parciais, tanto primeiro quanto o segundo refinamento, percebemos que a quantidade de instruções executadas aumentou nessas duas últimas versões. Mesmo quando comparamos *FCFC total* com as versões onde os *NOPs* foram invertidos (com o objetivo de evitar a execução dos mesmos) também percebemos que o número de instruções executadas aumenta.

Como dito anteriormente, isso ocorreu pelo fato de termos substituídos as instruções de detecção por instruções *NOP*. Em alguns casos uma única instrução teve que ser substituída por 12 instruções *NOP*. Isso ocorreu porque trabalhamos em um ambiente CISC onde as instruções não possuem um tamanho regular.

Porém, em um ambiente RISC, onde as instruções possuem tamanho regular, por exemplo, no MIPS, existe a perspectiva de ganho de desempenho, como mostrado na Seção 6.5. Nesse caso cada instrução poderia ser trocada por exatamente um *NOP*. Por si só isso poderia melhorar o desempenho do sistema, pois a execução de um *NOP* é relativamente mais rápida que a execução de um *mov*, por exemplo. Mas além disso, poderia ser possível diminuir a quantidade de instruções executadas. Adicionalmente, a utilização da técnica proposta neste trabalho em um ambiente RISC evitaria os problemas relatado nas Seção 5.8, o desalinhamento das instruções devido a realocação de registradores e devido a utilização de *Jump short* e *Jump near*.

Então, como um trabalho futuro, o principal objetivo é avaliar a técnica de detecção e proteção de blocos básicos suscetíveis através da análise sistemática de *single bit-flip* em um ambiente RISC. Para tal, será necessário modificar o passos implementados no LLVM e configurar o compilador de tal forma que ele gere o código para a arquitetura RISC escolhida. Também será necessário modificar o injetor de falhas, pois esse utiliza vários componentes presente no Linux e que talvez não funcionem na arquitetura escolhida.

REFERÊNCIAS

SCHUETTE, M. A.; SHEN, J. P. Processor Control Flow Monitoring Using Signatured Instruction Streams. **IEEE Transactions on Computers**, v. C-36, n. 3, p. 264-276, mar. 1987.

ALKHALIFA, Z.; NAIR, V. S. S.; KRISHNAMURTHY, N; ABRAHAM J. A. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. **IEEE Transactions on Parallel and Distributed Systems**, v. 10, n. 6, p. 627-641, jun. 1999.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Control-flow checking by software signatures. **IEEE Transactions on Reliability**, v. 51, n. 1, p. 111-122, mar. 2002.

VEMU, R.; ABRAHAM, J. A. CEDA: Control-Flow Error Detection Using Assertions. **IEEE Transactions on Computers**, v. 60, n. 9, p. 1233-1245, set. 2011.

MAHMOOD, A.; MCCLUSKEY, E. J. Concurrent error detection using watchdog processors-a survey. **IEEE Transactions on Computers**, v. 37, n. 2, p. 160-174, fev. 1988.

LU, D. J. Watchdog Processors and Structural Integrity Checking. **IEEE Transactions on Computers**, v. C-31, n. 7, p. 681-685, jul. 1982.

AVIZIENIS, A.; KELLY, J. P. J. Fault Tolerance by Design Diversity: Concepts and Experiments. **IEEE Transactions on Computers**, v. 17, n. 8, p. 67-80, ago. 1984.

BAGCHI, S.; SRINIVASAN, B.; WHISNANT, K.; KALBARCZYK, Z.; IYER, R. K. Hierarchical error detection in a software implemented fault tolerance (SIFT) environment. **IEEE Transactions on Knowledge and Data Engineering**, v. 12, n. 2, p. 203-224, mar. 2000.

MAY, T. C.; WOODS, M. H. Alpha-particle-induced soft errors in dynamic memories. **IEEE Transactions on Electron Devices**, v. 26, n. 1, p. 2-9, jan. 1979

OHLSSON, J; RIMEN, M; GUNNEFLO, U. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 22., 1992. **Proceedings...** [S.l:s.n], 1992. p. 316-325

VENKATASUBRAMANIAN, R.; HAYES, J.P.; MURRAY, B.T. Low-cost on-line fault detection using control flow assertions. In: IEEE ON-LINE TESTING SYMPOSIUM, 9., 2003. **Proceedings...** [S.l:s.n], 2003. p. 137-143

GOLOUBEVA, O.; REBAUDENGO, M.; REORDA, M.S.; VIOLANTE, M. Soft-error detection using control flow assertions. In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 18., 2003. **Proceedings...** [S.l:s.n], 2003. p. 581-588

KHUDIA, D. S.; MAHLKE, S. Low cost control flow protection using abstract control signatures. In: ACM SIGPLAN/SIGBED CONFERENCE ON LANGUAGES, COMPILERS AND TOOLS FOR EMBEDDED SYSTEMS, 14., 2013. **Proceedings...** New York, NY: ACM, 2003. p. 3-12.

JEONG, K.; KAHNG, A. B. A power-constrained MPU roadmap for the International Technology Roadmap for Semiconductors (ITRS). In: INTERNATIONAL SOC DESIGN CONFERENCE, 2009. **Proceedings...** [S.l:s.n], 2009. p. 49-52

AUSTIN, T.M. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In: Annual International Symposium on Microarchitecture, 32., 1999. **Proceedings...** [S.l:s.n], 2009. p. 196-207.

RAGEL, R.G.; PARAMESWARAN, S. Hardware assisted pre-emptive control flow checking for embedded processors to improve reliability. In: International Conference Hardware/Software Codesign and System Synthesis, 4., 2006. **Proceedings...** [S.l:s.n], 2006. p. 100-105

KANAWATI, G.A.; NAIR, V.S.S.; KRISHNAMURTHY, N.; ABRAHAM, J.A. Evaluation of integrated system-level checks for on-line error detection. In: IEEE International Computer Performance and Dependability Symposium, 1996. **Proceedings...** [S.l:s.n], 1996. p. 292-301

GUTHAUS, M. R.; RINGENBERG, J. S.;ERNST, D.; AUSTIN, T. M.; MUDGE, T.; BROWN, R. B. MiBench: A free, commercially representative embedded benchmark suite. In: International Workshop on Workload Characterization, 4., 2001. **Proceedings...** [S.l:s.n], 2001. p. 3-14

CONSTANTINESCU, C. **Impact of deep submicron technology on dependability of VLSI circuits**. In: International Conference on Dependable Systems and Networks, 2002. **Proceedings...** [S.l:s.n], 2002. p. 205-209

LATTNER, C. **LLVM: An Infrastructure for Multi-Stage Optimization**. 2002. 68 f. Dissertação (Mestrado em Ciência da Computação) — Computer Science Dept., University of Illinois at Urbana-Champaign, 2002. Disponível em <http://llvm.cs.uiuc.edu>. Acesso em: 11 out. 2014.

PARIZI, B, R. **Implementação e avaliação da técnica ACCE para detecção e correção de erros de fluxo de controle no LLVM**. Dissertação de Mestrado. Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação.

TIMMER, J. A fast look at Swift, Apple's new programming language. **Ars Technica**. [S.l:s.n]. Disponível em <http://http://arstechnica.com/apple/2014/06/a-fast-look-at-swift-apples-new-programming-language/>. Acesso em: 11 out. 2014.

INTEL CORPORATION. **Intel® 64 and IA-32 Architectures Optimization Reference Manual. Volume 2 (2A, 2B and 2C): Instruction Set Reference, A-Z**. [S.l.]:Intel, 2015.

ZIEGLER, J. F.; PUCHNER, H. **SER-History, Trends, and Challenges: A Guide for Designing with Memory ICs**. [S.l.]:Cypress Semiconductor Corp., 2004.

MUKHERJEE, S. **Architecture Design for Soft Errors**. San Francisco, CA:Morgan Kaufmann Publishers Inc., 2008.

GOLOUBEVA, O.; REBAUDENGO, M.; REORDA, M. S.; VIOLANTE, M. **Software Implemented Hardware Fault Tolerance**. Torino, Italy: Springler, 2006.

APÊNDICE A LLVM

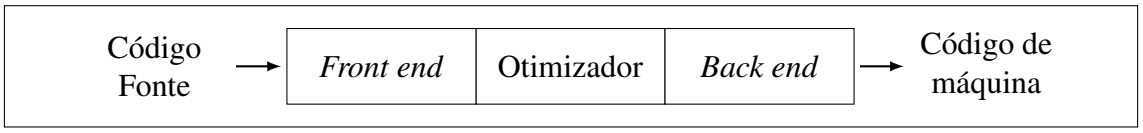
O LLVM (*Low Level Virtual Machine*) é um *framework* de compilação formado por diferentes ferramentas de baixo nível (e.g: compiladores, assemblers e debbugers), projetadas para serem compatíveis com ferramentas utilizadas por sistemas Unix. Seus componentes são escritos na linguagem C++ e foram projetados para permitir otimizações em tempo de compilação, ligação e execução.

Originalmente implementado para as linguagens C e C++, um dos fatores que garantiu seu sucesso foi seu projeto independente de linguagem, o que gerou um grande número de *front end* para diferentes linguagens, como por exemplo: Common Lisp, Haskell, Python, Ruby, Rust, Scala, C#, Lua, etc. O projeto teve início na Universidade de Illinois em Urbana-Champaign por Chris Lattner, sob orientação de Vikram Adve, em 2000 (LATTNER, 2002).

A versão inicial do LLVM, lançada em 2003, foi focada em pesquisas acadêmicas em compiladores e otimização de código. A partir de 2005, o LLVM passou a ser usado na indústria, mais especificamente pela Apple, que contratou Lattner para incorporar o LLVM no desenvolvimento de vários produtos comercializados pela empresa, entre eles o XCode, uma IDE *(Integrated Development Environment)* para desenvolvimento de software, e o MAC OS X, sistema operacional dos sistemas computacionais comercializados pela Apple. O LLVM também dá base a nova linguagem de programação Swift (TIMMER, 2014), introduzida pela Apple na Worldwide Developers Conference 2014.

A estrutura mais popular para um compilador estático tradicional é o de três fases, cujos componentes principais são o *front ends*, o otimizador e *back end* (Figura A.1). O *front end* analisa o código fonte, verificando se há erros, e constrói a Árvore de sintaxe abstrata (AST, para *Abstract Syntax Tree*) para representar o código de entrada. A AST é eventualmente convertida em uma nova representação para a otimização.

Figura A.1: Arquitetura clássica de compiladores

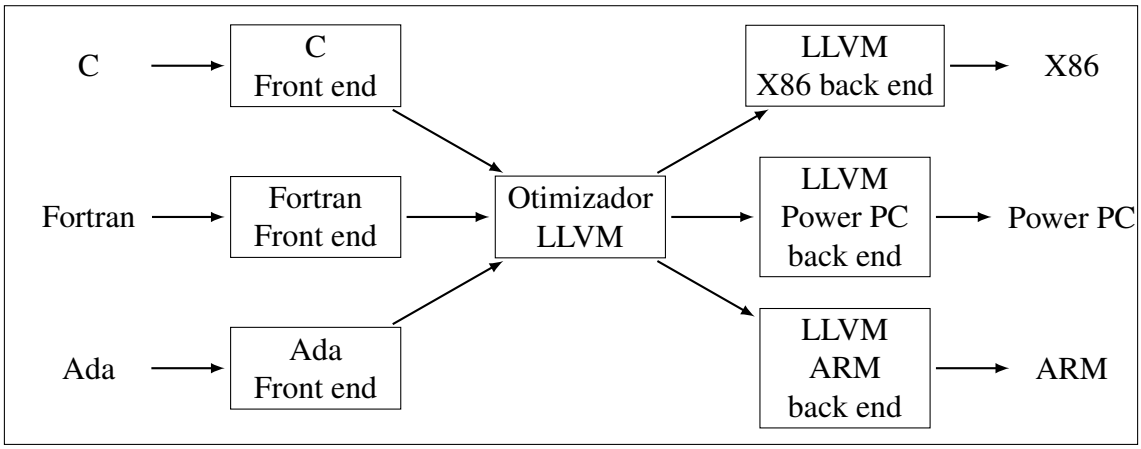


O otimizador é responsável por fazer uma ampla variedade de transformações para tentar melhorar a performance do código, tais como a eliminação de cálculos redundantes. Ele é geralmente independente, com maior ou menor grau, da linguagem alvo. O *back end* (também conhecido como o gerador de código) mapeia então o código para o conjunto de instruções da linguagem alvo. Além de tornar o código correto, ele é responsável pela

geração de um bom código que tire proveitos das principais características da arquitetura suportada.

Por sua vez, o LLVM expande esta estrutura clássica permitindo múltiplas linguagens de programação e múltiplas arquiteturas de destino. A Figura A.2 ilustra esta ideia.

Figura A.2: Estrutura LLVM



Com esta estrutura o LLVM pode suportar novas linguagens programação facilmente, pois só o que precisa ser modificado é o *front end*. O otimizador e o *back end* podem ser utilizados sem modificação. Se este componentes não fossem independentes, para qualquer nova linguagem que se criasse, seria necessário reimplementar todos os componentes.

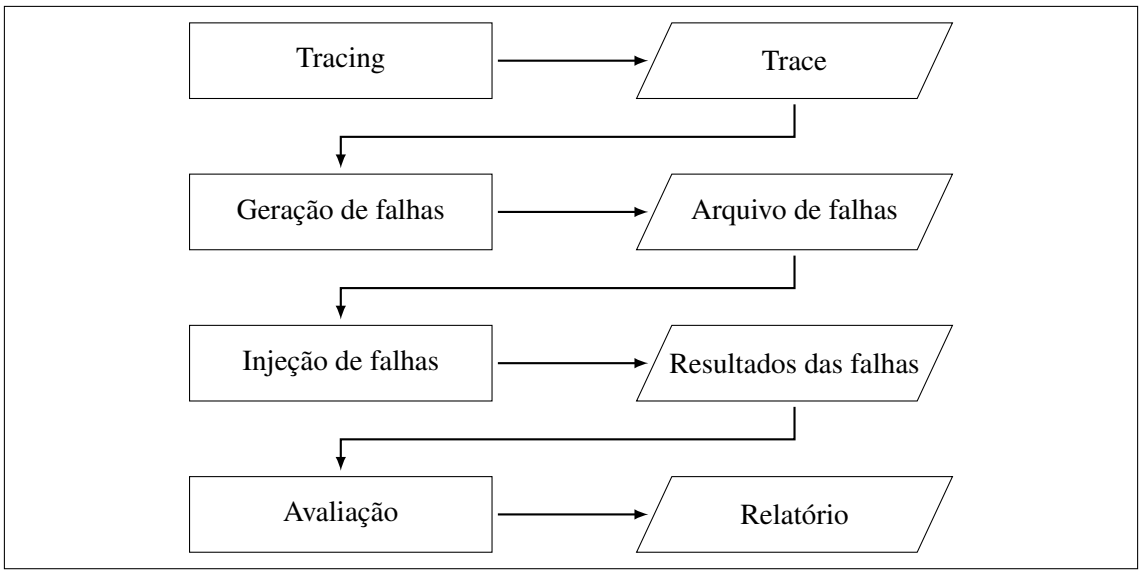
Uma das características mais importantes do LLVM é a sua linguagem intermediária LLVM IR, que é independente de linguagem de programação e também independente de arquitetura de máquina. LLVM IR foi projetada para de tal forma a facilitar analises e transformações de nível intermediário que podem ser encontradas nas etapas de otimização de um compilador comum. LLVM IR provê informações de alto nível sobre programas, como tipos, para suportar sofisticadas transformações e análises, e ao mesmo tempo é próxima ao Assembly contendo instruções no formato de 3 endereços, facilitando a tradução para código objeto. A linguagem LLVM-IR é representada no formato *Single Static Assignement (SSA)* onde cada variável é atribuída apenas uma única vez. O formato SSA facilita a implementação de inúmeras otimizações no código de forma mais eficiente. Para alcançar a forma SSA, o LLVM possui um conjunto infinito de registradores, que são numerados sequencialmente em um programa.

APÊNDICE B INJETOR DE FALHAS

Diferentes trabalhos fazem referencia a injetores de falhas baseados em software que permitem a simulação de erros de fluxo de controle. Porém, mesmo depois de muita pesquisa, nenhum desses softwares foi encontrado através das referências disponíveis. Sendo assim, fez-se necessário a criação de um injetor de falhas que permitisse a simulação de erros de fluxo de controle para a continuidade do trabalho. O injetor de falhas descrito a seguir foi inspirado fortemente na descrição do injetor de falhas utilizado por (VEMU; ABRAHAM, 2011). O resultado do trabalho está disponível, através do seguinte endereço eletrônico `http://sourceforge.net/projects/faultsinjectionframework/`.

O injetor de falhas desenvolvido neste trabalho é do tipo SWIFI (*Software-Implemented Fault Injection*). Ele foi desenvolvido para plataforma Linux rodando em um processador com arquitetura AMD64 (x86-64). As principais ferramentas utilizadas foram o GDB (*GNU Projet debugger*) e *scripts* escritos na linguagem Bash (*GNU Project's shell*). A Figura B.1 mostra a arquitetura geral da ferramenta e suas principais etapas. Em cada etapa temos um *script* na linguagem Bash responsável por uma determinada tarefa. Cada etapa gera um tipo de arquivo texto diferente, que serve como entrada para a etapa seguinte.

Figura B.1: Arquitetura do injetor de falhas



B.1 Geração do arquivo de *trace*

Na primeira etapa do injetor de falhas é gerado o *trace log* do programa, ou simplesmente *trace*. O *trace* é um arquivo texto que guarda os dados referente a todas as instruções executadas. O trabalho deste etapa é realizado pelo *script tracing.sh*.

O primeiro passo para a geração do *trace* consiste em executar, através do GDB, o programa do qual queremos extrair o *trace*. Durante a execução do programa, o GDB permite que as instrução executadas sejam salvas em um arquivo texto. Então, ao final da execução do programa temos um arquivo texto com todas as instruções executadas pelo programa, linha após linha.

No segundo passo, o *trace* passa por uma etapa de refinamento que consiste em agrupar e contabilizar o número de instruções executadas. O resultado final após o refinamento pode ser visto no exemplo da Figura B.2, onde em cada linha temos, respectivamente, o endereço da instrução executada, a instrução executada, a quantidade de vezes que a instrução foi executada e, se for uma instrução de desvio, o endereço da instrução imediatamente posterior à instrução de desvio.

Figura B.2: Exemplo de *trace*

```
1 0x401d29 movq $0x2a,0x6030a8 19986
2 0x401d35 movq $0x0,0x6030f0 19986
3 0x401d41 mov 0x603070,%rcx 19986
4 0x401d49 mov 0x6030f8,%rdx 19986
5 0x401d51 xor $0x29,%rcx 19986
6 0x401d55 xor %rdx,%rcx 19986
7 0x401d58 mov %rcx,0x6030f8 19986
8 0x401d60 jmpq 0x401c6c 19986 401d65
9 0x401d65 lea 0x40292f,%rdi 18000
10 0x401d6d mov $0x0,%al 18000
11 0x401d6f callq 0x4005f0 18000
12 0x401d74 movq $0x2b,0x603070 18000
13 0x401d80 mov 0x603068,%rdi 18000
14 0x401d88 mov 0x6030f8,%rcx 18000
15 0x401d90 xor $0x28,%rdi 18000
16 0x401d94 xor %rcx,%rdi 18000
17 0x401d97 mov %rdi,0x6030f8 18000
18 0x401d9f mov %eax,-0x18c(%rbp) 18000
```

O *script tracing.sh* tem como entrada, além do programa do qual queremos extrair a *trace*, um arquivo texto com o nome das funções presentes no programa. Durante este trabalho, a geração desses arquivo foi feita com o auxílio de um passo de análise LLVM. O passo de análise simplesmente percorre o arquivo do programa na linguagem LLVM-IR, obtém o nome de todas as funções do programa e coloca estes nome em um arquivo texto. A Figura B.3 exemplifica o formato do arquivo.

Figura B.3: Formato do arquivo com o nome das funções do programa

```
1 function_name_2
2 function_name_3
3 function_name_4
4 function_name_5
```

B.2 Geração do arquivo de falhas

De posse do arquivo *trace* é possível ir para a próxima etapa, que consiste na geração do arquivo de falhas. O arquivo de falhas contém as instruções para a próxima etapa do processo, a injeção de falhas. Essas instruções dizem quantas falhas serão injetadas, e para cada falha, o endereço da instrução em que a falha será injetadas e o número de execução da instrução em que a falha será executada. O trabalho desta etapa é realizado pelo *script faults_generator.sh*. Nestas etapa os seguintes procedimentos são feitos pelo *script*.

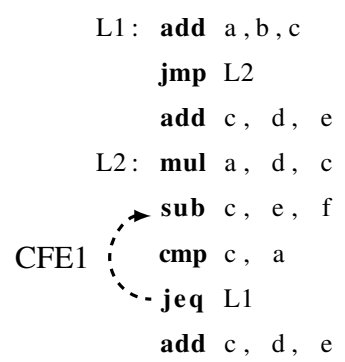
ii De forma aleatória, seleciona uma instrução de desvio na qual o erro será inserido. O *script* seleciona as instruções levando-se em conta o número de vezes que cada instrução foi executada. Por exemplo, se uma instrução que foi executada mil vezes ela terá maior probabilidade de ser selecionada do que uma instrução que foi executada apenas dez vezes.

iii De forma aleatória, seleciona uma das execuções da instrução onde a falha será injetada.

iv Por fim, gera um arquivo com todas as informações necessárias para o injetor de falhas.

O injetor de falhas implementa o modelo de falhas alteração no operando de desvio. A Figura B.4 ilustra esse modelo. *CF1* representa um alteração do operando de desvio onde o campo de destino da instrução *jeq L1* é corrompido e a instrução resulta em um desvio para um endereço diferente do especificado originalmente.

Figura B.4: Exemplo simulação erro de fluxo de controle



(a) Fonte: (VEMU; ABRAHAM, 2011), p. 1235

A Figura B.5 mostra um exemplo de um arquivo gerado por esta etapa. Cada linha representa uma falha a ser injetada. Por exemplo, a linha 2 mostra que uma falha será inserida na instrução de endereço `0x40100b` (instrução `jmpq` da Figura B.2, linha 9), o modelo da falha é do tipo `change` (alteração do operando de desvio), ou seja, o endereço de destino original do desvio será alterado através de um operação `XOR` entre seu próprio valor e um valor aleatório, neste caso o valor `2097152`. O próximo valor, `12640`, diz que a falha será injetada na 12640 execução da instrução. Por fim, o valor `0x400e1e` representa o destino original da instrução de desvio.

Figura B.5: Arquivo de falhas

```
1 change 0x401d60 2097152 12640 0x401c6c
2 change 0x400e6e 64 31277 0x401010
3 change 0x401bb9 8 11017 0x401e09
4 change 0x401e04 4194304 10954 0x401b4c
5 change 0x401af0 32 109 0x401e87
6 change 0x40088f 16384 13724 0x400b2c
7 change 0x401d60 4096 8798 0x401c6c
8 change 0x40088f 8192 11848 0x400b2c
9 change 0x401cc4 32 32729 0x401d65
10 change 0x401bb9 32 18541 0x401e09
11 change 0x40212c 32768 308 0x401ffb
12 change 0x401d60 4096 14992 0x401c6c
13 change 0x40100b 16 7057 0x400e1e
14 change 0x400f2d 8 5180 0x400f81
15 change 0x401bb9 256 1948 0x401e09
16 change 0x400ef4 16 22526 0x400f2f
17 change 0x401d60 256 9302 0x401c6c
18 change 0x401bb9 4 3905 0x401e09
19 change 0x400e6e 65536 9434 0x401010
```

B.3 Injeção de falhas

Uma vez que o arquivo de falhas esteja pronto, podemos passar para o próximo passo, ou seja, inserir as falhas propriamente ditas. O trabalho nesta etapa é realizado pelo *script faults_injector.sh*. Este *script* lê o arquivo de falhas e para cada falha (linha) deste arquivo ele injeta uma falha no programa da seguinte forma:

i Inicializa-se o GDB e carrega-se o programa onde as falhas serão injetadas.

ii Insere-se um *breakpoint* na instrução onde a falha será injetada.

iii Executa-se o programa. O programa irá parar a execução em cada execução da instrução selecionada.

iv Continua a execução do programa (n-1) vezes. Na *nth* execução da instrução, injeta a falha previamente selecionada.

v Continua a execução do programa e observa o efeito do erro. Se uma exceção é ativada, então o hardware/sistema operacional irá detectar o erro. Se nenhuma exceção é disparada, deve-se comparar a saída do programa com a saída do programa livre de erro. Cada execução do programa gera uma saída que poderá, ou não, ser afetada pela falha injetada. Além disso, o *script faults_injector.sh* coloca em um arquivo de saída todos os resultados gerados pelo GDB. Os resultados gerados pelo GDB são diferentes da saída produzida pelo programa onde estamos injetando as falhas, por isto estas informações devem ficar separadas.

B.4 Avaliação

O último passo do injetor de falhas consiste em avaliar os error gerados pela injeção de falhas. Para tal, deve-se avaliar o arquivo de saída gerado pelo *script faults_injector.sh* e todas as saídas produzidas pelo programa onde as falhas foram injetadas. Os erros gerados são categorizados de acordo com o efeito deles no software. As categorias consideradas são as seguintes.

i OS: Caso em que os erros são detectados pelo sistema operacional ou pelo hardware. São erros do tipo *segmentation fault*, *illegal instructions*, etc.

ii AD: Caso em que os erros são detectados pelo mecanismo de detecção embutido no programa em execução. Erros detectados por técnicas para detecção de erro de fluxo de controle encaixam-se neste tipo.

ii WA: Caso em que os erros não são detectados e o programa gera uma saída erra. Este tipo de erro precisa ser minimizado para aumentar a tolerância a falhas do software.

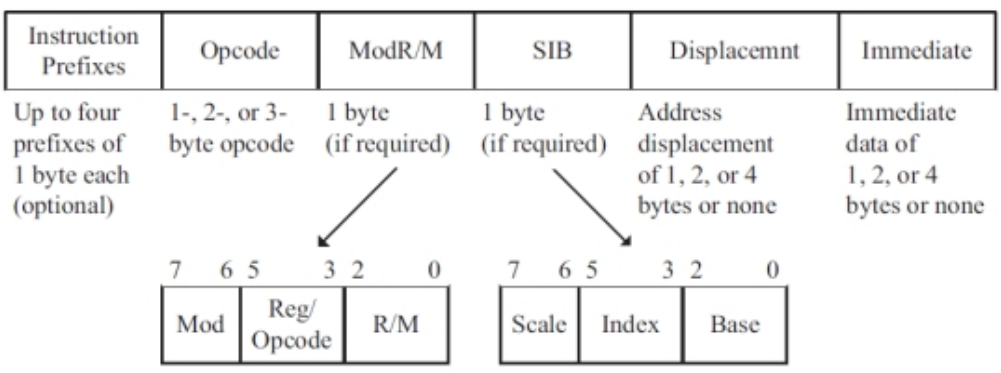
iv CA: Caso em que o programa gera saída corretas apesar do erro e o erro não é detectado por nenhum tipo de técnica de detecção empregada, tampouco pelo sistema operacional ou hardware.

APÊNDICE C ANÁLISE DA TRANSFORMAÇÃO DE OP-CODES

Os modelos de falhas *criação de desvio* pode ocorrer se uma instrução que não é de desvio se transforma em um desvio e o modelo de falhas *remoção de desvio* pode ocorrer se um desvio se transforma em uma operação que não é de desvio. Porém, a probabilidade de se transformar o *opcode* de uma instrução que não é de desvio em um *opcode* de uma instrução de desvio e vice-versa, devido a um único *bit-flip*, é extremamente baixa. Além de depender da codificação dos *opcode* e do conjunto de instruções da arquitetura.

A Figura C.1 mostra o formato das instruções na arquitetura Intel 64 e IA-32 (INTEL, 2015). Como podemos perceber, existem diferentes campos que formam a instruções, sendo um deles o *opcode*. Então, por exemplo, considerando as instruções onde o *opcode* é formado por um byte, para que a instrução ADD, *opcode* 00 (0000 0000), se transforme em um JO, *opcode* 70 (0111 0000), são necessários três *bit-flips* no *opcode*, em posições específicas.

Figura C.1: Formato das instruções na arquitetura Intel 64 e IA-32



(INTEL, 2015) (página 31)

A Tabela C.1 mostra um pequeno trecho da análise realizada entre os diferentes *opcode* com apenas um byte. A análise consistiu em realizar a diferença (ou exclusivo) entre os *opcode* de uma instrução JUMP e o *opcode* das demais instruções, e então contabilizar quantos estavam distantes por mais de um bit e quantos estavam distantes por apenas um

bit. Na Tabela C.1 - que contém apenas alguns valores da análise realizada - podemos verificar que apenas a instrução XOR, *opcode* 30, está distante apenas um bit das instrução JO, *opcode* 70.

O resultados final da análise mostrou que em apenas 1,60% dos casos existia a diferença de apenas um bit entre o opcode de uma instrução que não é JUMP e uma instrução JUMP e vice-versa. Em 4,66% dos casos existia diferença de um bit entre uma instrução JUMP e outra instrução que também é um JUMP. Nos demais casos, 93,84% a diferença foi de mais de um bit.

Portanto, para os *opcode* com um byte, existe a 1,60% possibilidade de um *opcode* de uma instrução não JUMP virar um *opcode* de uma instrução JUMP, e vice-versa, devido a um único bit-flip. Além disso, o bit-flip deve ocorrer em um bit específico dentre os oito possível do *opcode*, o que reduz as chances para 0,18%.

Tabela C.1: Diferenças em relação ao *opcode* 70 da instrução JO (Jump short)

Opcode	Mnemonic	Resultado do Xor
00	ADD	1110000
01	ADD	1110001
02	ADD	1110010
03	ADD	1110011
04	ADD	1110100
05	ADD	1110101
06	PUSH	1110110
07	POP	1110111
...	...	...
2A	SUB	1011010
2B	SUB	1011011
2C	SUB	1011100
2D	SUB	1011101
2F	DAS	1011111
30	XOR	1000000
31	XOR	1000001
32	XOR	1000010
33	XOR	1000011
...	...	...