Quantitative Evaluation of Control Flow based Soft Error Protection Mechanisms

by

Abhishek Rhisheekesan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved March 2013 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Charles Colbourn
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

May 2013

ABSTRACT

Rapid technology scaling, the main driver of the power and performance improvements of computing solutions, has also rendered our computing systems extremely susceptible to transient errors called *soft errors*. Among the arsenal of techniques to protect computation from soft errors, Control Flow Checking (CFC) based techniques have gained a reputation of effective, yet low-cost protection mechanism. The basic idea is that, there is a high probability that a soft-fault in program execution will eventually alter the control flow of the program. Therefore just by making sure that the control flow of the program is correct, significant protection can be achieved. More than a dozen techniques for CFC have been developed over the last several decades, ranging from hardware techniques (CFCET [32], TTA [23], ASIS [8], W-D-P [22], OSLC [20]), software techniques (CFCSS [29], ECCA [2], CEDA [42], ACCE [43], YACCA [13], ACFC [44]), and hardware-software hybrid techniques (CFEDC [10], CSM [45], SIS [37], Watchdog assists [35]) as well.

Our analysis shows that existing CFC techniques are not only ineffective in protecting from soft errors, but cause additional power and performance overheads. For this analysis, we develop and validate a simulation based experimental setup to accurately and quantitatively estimate the architectural vulnerability of a program execution on a processor micro-architecture. We model the protection achieved by various state-of-the-art CFC techniques in this quantitative vulnerability estimation setup, and find out that software only CFC protection schemes (CFCSS [29], CFCSS+NA [5], CEDA [42]) **increase** system vulnerability by 18% to 21% with 17% to 38% performance overhead. Hybrid CFC protection (CFEDC [10]) **increases** vulnerability by 5%, while the vulnerability remains almost the same for hardware only CFC protection (CFCET [32]); notwithstanding the hardware overheads of design cost, area, and power incurred in the hardware modifications required for their implementations.

DEDICATION

To my parents, for their unconditional love and support.

ACKNOWLEDGEMENTS

I am indebted to my advisor, Dr. Aviral Shrivastava, without whose guidance and immense help, this thesis would not have been possible. I would like to express my sincere gratitude to him for keeping my spirits up when the muses failed me. His trenchant criticisms and probing questions enabled me to develop a solid understanding of the subject and instilled in me an attitude of not to settle for mediocrity. I am grateful for the valuable feedback, and the expertise provided by him.

I would like to make a special reference to my colleague, Dr. Reiley Jeyapaul for his invaluable guidance and the energetic discussions we had on the project. He was always there to help whenever I was detracting away from the end goal. I extend my heartfelt thanks to him for his critical reviews of my work.

I would also like to thank my committee members, Dr. Charles Colbourn for providing me an opportunity to be the teaching assistant for his course on theoretical computing, and Dr. Carole-Jean Wu. I also thank my labmates at Compiler Microarchitecture Laboratory for their invaluable feedback and helping in reviewing the literature during our "monday meetings". A special thanks goes to my colleague, Bryce Holton, for his suggestions and the valuable discussions we had in the lab.

I am extremely grateful to my parents, K. R. Rhisheekesan and Rema M. P, and my siblings, Abhijith and Aswathy, for their endless love, support and encouragement. A master's degree can be a lonely and painful journey, if not for my cheerful friends and batchmates (Sangeeta Balram, Anurag Joshi, Mahendra Rautela, Manmohan K, Rahul A. R, Manu Basil, Ramakirshna Prasad, and Anu Mercian, to name a few due to the restricted space), who make it rather an enjoyable one. And finally, I thank Almighty for leading me through the difficult times and for being a major source of strength.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Continuous and exponential technology scaling for the last 5 decades has enabled us to pack high-performance compute devices needing very little power, in small-size packaging at affordable cost. This has set in motion the information revolution, and the unprecedented integration of computing systems into our every-day life. However, a negative consequence of technology scaling this is that the transistors within modern (highly compact, and fast) processors have become extremely susceptible to *soft errors*. Among the many sources of transient faults in the system (e.g., electrical noise, external interference, cross-talk, etc.) charge carrying particles (alpha, low and high energy neutrons, etc.) cause the majority of soft errors in electronic devices [21]. At the current technology node, high-end embedded systems, e.g., smart-phones, tablets, etc., incur a Soft Error Rate (SER) of about *once-per-year*, but is expected to increase exponentially to once-per-day in a decade [18].

As a consequence, reliability is rapidly emerging as a primary design metric. Over the years, several schemes have been developed to protect computing systems from soft errors. Most protection schemes are built around time and/or space redundancy. The idea is the perform a computation twice, and see if the results match. If not, then must have been an error. These redundancy based protection schemes can and have been developed at various levels of system design abstraction – from transistor level [15, 27] to gate level [12, 40] to system level [41, 30, 26, 39, 17]. Although redundancy based methods have been considered effective in providing system reliability, they are generally considered to have high overhead. In particular, while it may be possible to minimize/hide the performance overhead of redundancy, the power overhead of redundancy cannot be hidden.

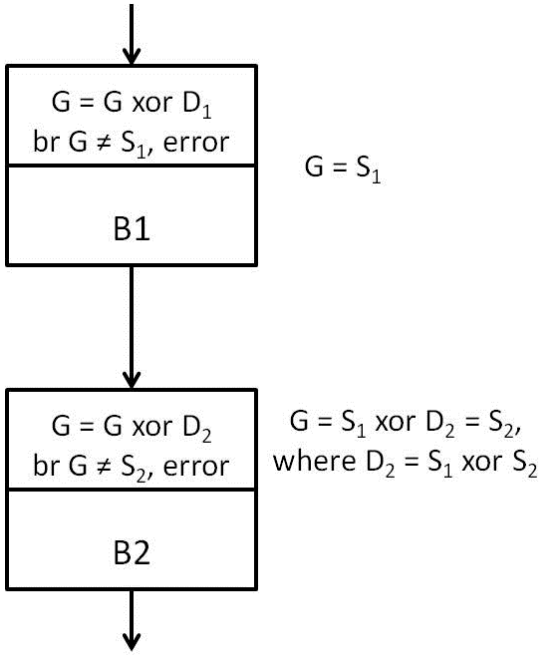


Figure 1: The mechanism of signature verification to detect control flow errors in CFCSS.

As an alternative, there is a whole class of techniques called Control Flow Checking (CFC) techniques. The key idea here is that the majority of soft errors affecting program behavior eventually manifest in the form of errors in the program execution sequence. Thus, by making sure that the control flow of the program is correct, significant protection can be achieved. CFC tecnniques can often be implemented with much less overhead than full scale redundancy. For example, as shown in Figure 1, the software-based CFC technique, CFCSS, adds some instructions to assign a variable to a unique value (signature) in each basic block, and also adds instructions in each basic block to check if the control flow is coming from a legitimiate basic block. This is achieved by comparing the value of the signature variable, held in the runtime signature register *G*. Thus a software CFC can be implemented by adding only a few instructions per basic block.

Owing to the belief that CFC will provide high levels of protection at relatively low overheads, more than a dozen proposals have been made on different ways to implement CFC. The arsenal of control flow based soft error protection techniques developed span across design layers from hardware [32, 23, 8, 22, 20], software [29, 42, 28, 2, 43, 13, 44], and hardware-software hybrid techniques [10, 45, 37, 35].

In this thesis, we develop a simulation based tool that can quantitatively and accurately estimate the architectural vulnerability of execution. We further model and estimate the protection achieved by the various CFC techniques. Our results reveal that CFC techniques do not protect execution from soft errors, but incur additional power and performance overheads. In particular, software only CFC protection schemes (CFCSS[29], CFCSS+NA[5], CEDA[42]) **increase** system vulnerability by 18% to 21% with 17% to 38% performance overhead. Hybrid CFC protection (CFEDC[10]) **increases** vulnerability by 5%, while the vulnerability remains almost the same for hardware only CFC protection (CFCET[32]); notwithstanding the hardware overheads of design cost, area, and power incurred in the hardware modifications required for their implementations.

Although the previous papers that proposed the CFC techniques demonstrated the effectiveness of their approach, their measurement was flawed. In all the previous papers, researchers have performed targeted fault injection – in the sense that they inject only control flow faults, and then perform the simulation to see if their technique can detect the injected fault or not. This targeted fault injection setup is perfect for debugging to see if the technique is working in all the cases, but it is not the right setup to estimate the effectiveness of the technique. This setup can measure, out of the control flow faults injected, what fraction of the faults can be caught by their technique, but control flow faults are only a small fraction of soft errors that happen.

From the analysis of the results, we are able to find out why existing CFC techniques are not very effective. Although it is true that many faults eventually translate to control flow errors in programs, there are two ways in which this happens. i) Direct: For example, a fault happens in the program counter, or PC. Then the control flow will be altered. Most existing CFC techniques are able to capture such faults, but as our analysis shows, only a small fraction of faults cause this. ii) Indirect: The fault, e.g., an error in the register file does not cause a control flow error immediately, but eventually this error propagates to variables that are used in deciding the branch outcome. While most faults suffer this fate, no existing CFC technique can detect these faults. Notably one of the latest techniques, CEDA makes a feeble attempt at this, but still falls far short.

Chapter 2

BACKGROUND

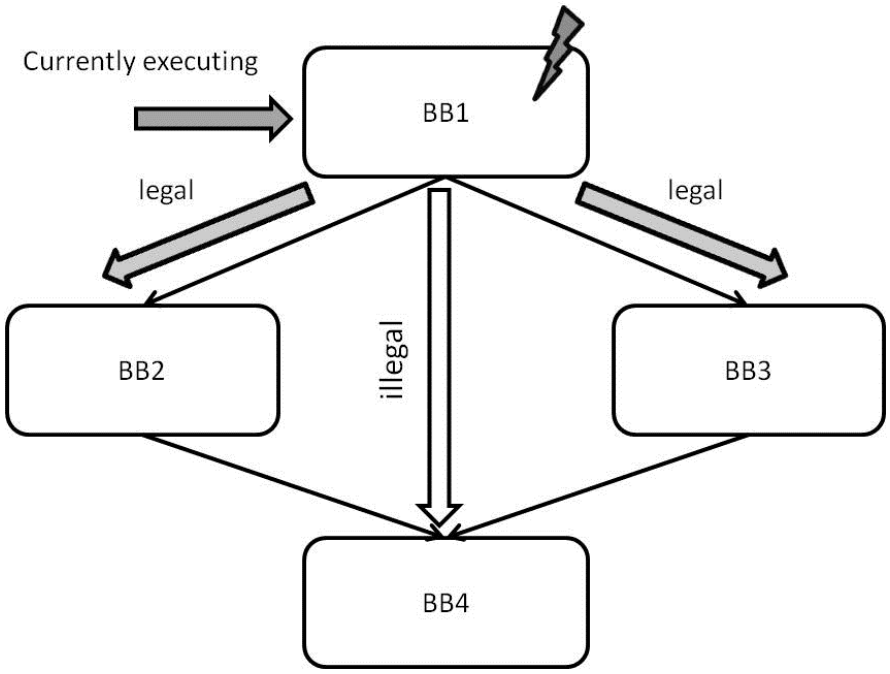


Figure 2: Control Flow Error.

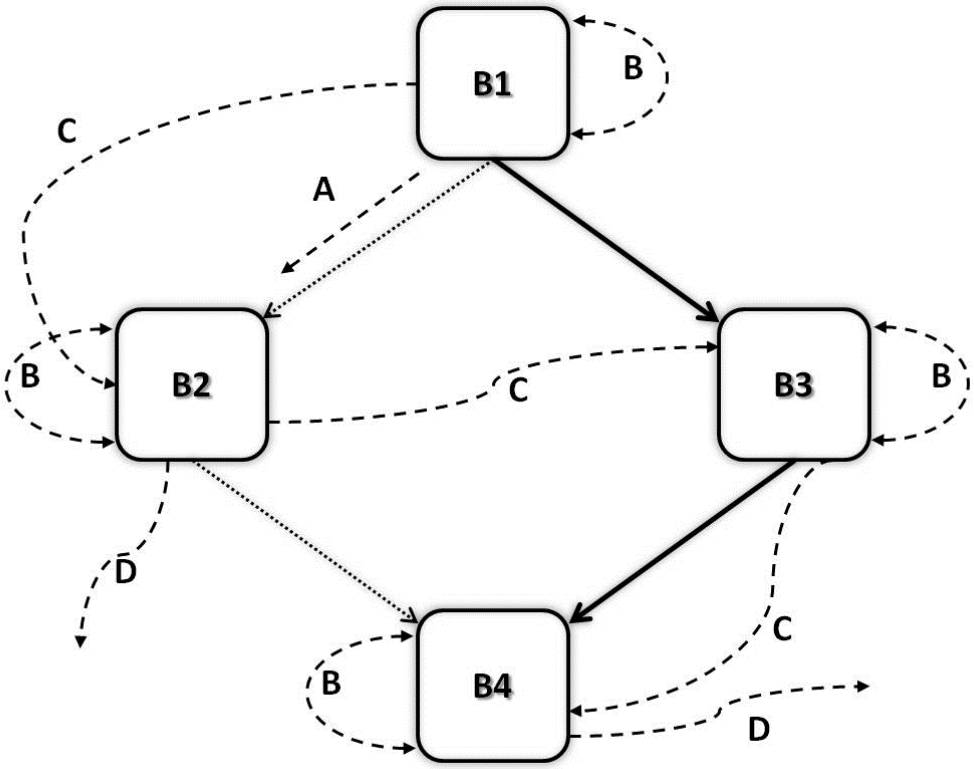
The intent of this research is to design, implement and validate a simulation based framework to evaluate the effectiveness of control flow checking mechanisms, specifically, the control flow error detection mechanisms. In order to evaluate the effectiveness, the domain of control flow checking needs to be explored to understand the concepts involved and the models of error detection. As a first step, the variant of soft errors which cause error in the control flow of applications, called the control flow error, is defined. Given a program input, the series of instructions executed is fixed. A control flow error (CFE) is defined as a deviation from the correct execution of the program. The presence of soft errors increase the probability of the occurrence of a control flow error. The Figure 2 shows a control flow graph of the program with basic blocks as nodes and branches as edges. A basic block is a portion of the code in a program with a single entry point and a single exit point or terminator instruction like

branch or return instruction. In the Figure 2, the execution is legal from basic block *BB1* to *BB2* or from *BB1* to *BB3*, but it is illegal to jump from *BB1* to *BB4*. If a soft error strikes during the execution of instructions in *BB1*, and causes an illegal jump to some address location in *BB4*, some of the instructions have been skipped. Such a deviation in control flow is termed as a control flow error. A control flow error can cause an erroneous program output, a system crash or a system hang, or may go silent and cause output data errors, and hence the correct control flow is critical to the correct execution of the program.

The second step is to derive a classification of CFEs. The various types of errors are illustrated in Figure 3. For example, type A shows a control flow error that can cause a change in legal direction of branch from taken direction (true condition for conditional branch instructions) to the not-taken direction (the fall through or the false condition for conditional branch instructions) or vice versa. The categories of CFEs are further mapped to the corresponding sources of errors that can trigger each class of CFEs. The correlation of these sources of control flow errors to the processor hardware components will be established in the following sections, which helps to establish the hardware-software interface at which the quantitative evaluation of soft errors is the most relevant. In the table in Figure 3, the sources of the respective CFE models are identified as follows (The classification of error models is taken from [34]):

- *Error in Branch Condition*– An error in the branch condition value computed before the execution of the `branch` instruction will result in the execution of incorrect legal branch (e.g., dotted arrow **A** in Figure 3).

- *Error in Program Counter (PC)*– An error in the PC, will result in an incorrect instruction being fetched and executed; which results in haphazard segment of the program being executed in random. Errors which skip instructions within a



	Error Type	Error Location
A	Erroneous legal branch is executed	- Error in Program Counter (A_1) - Error in branch condition (A_2) - Branch Deletion (A_3)
B	Jump to same basic block	- Error in Program Counter (B_1) - Error in branch offset (B_2) - Branch Insertion (B_3) / Deletion (B_4)
C	Jump to different basic block	- Error in Program Counter (C_1) - Error in branch offset (C_2) - Branch Insertion (C_3) / Deletion (C_4)
D	Jump to non-code memory region	- Error in Program Counter (D_1) - Error in branch offset (D_2) - Branch Insertion (D_3)

Figure 3: Control Flow Errors and their Sources - For the control flow graph (over program basic blocks) of a simple `if-then-else` kernel is shown, and the possible control flow errors that can occur have been identified and labelled by dotted arrows. The table explains each of the control flow error types, and the sources of these errors in the system.

basic block, or jump across basic blocks are examples of such errors (**B**, **C** & **D** in Figure 3).

- *Branch Insertion / Deletion*– When the `opcode` of the instruction, during its execution the processor pipeline, is affected by a soft error, a `branch` instruction can be formed or deleted from the execution. This will result in a change in the control flow of the program at the boundaries of the basic block transitions or will result in skipping of instructions within a basic block or jumping across different basic blocks (**B**, **C** & **D** in Figure 3).

- *Error in Branch Offset*– When the "branch offset" of the instruction is corrupted by soft errors during execution, the target address of the branch is corrupted. This will result in an illegal basic block, or random code, that crosses the program memory bounds, being executed after a branch instruction (**B**, **C** & **D** in Figure 3).

Chapter 3

MOTIVATION

The important question is why or why not CFC? There are several arguments in favour of why control flow checking came into being in the first place and the several benefits of control flow checking. 33% of all transient errors result in control flow error on RISC processors [31]. It can be as high as 77% for CISC processors [37]. In general, control flow checking techniques are known to be cost effective and/or provide high coverage with less overheads, when compared to redundancy based mechanisms. On an average, CFC techniques provide 90+% error coverage. The average additional hardware cost for hardware based CFC mechanisms is less than 10%. For the software based CFC techniques, the performance overhead is less than 100% which is comparatively less when compared to redundancy based techniques. For example, the errors detected by a representative redundancy based technique, EDDI [30], is 22.08%, whereas CFCSS, a typical control flow checking technique, can detect 35.26% of errors. Both techniques achieve high error coverage (98.5% for EDDI, and 96.9% for CFCSS), although a higher portion of it is contributed by correct results despite of fault injection, and the faults detected by OS. The execution time overhead for EDDI is 105.9%, due to duplication of instructions, compared to 43.14% for CFCSS. The impact of CFEs on application software and operating systems was evaluated in [42] and is provided here for reference, see Figure 4 and Figure 5. CFEs can cause programs to generate wrong answers or outputs in 15% of the cases. The study on Linux OS as given in Figure 5 shows 55% of CFEs result in system crashes, while 9% result in kernel panics, and a mere 3% in system hangs, causing a combined undesired system state in 67% cases. Essentially, control flow checking can be considered as a low overhead soft error protection scheme that has the potential to detect 33% to 77% soft errors.

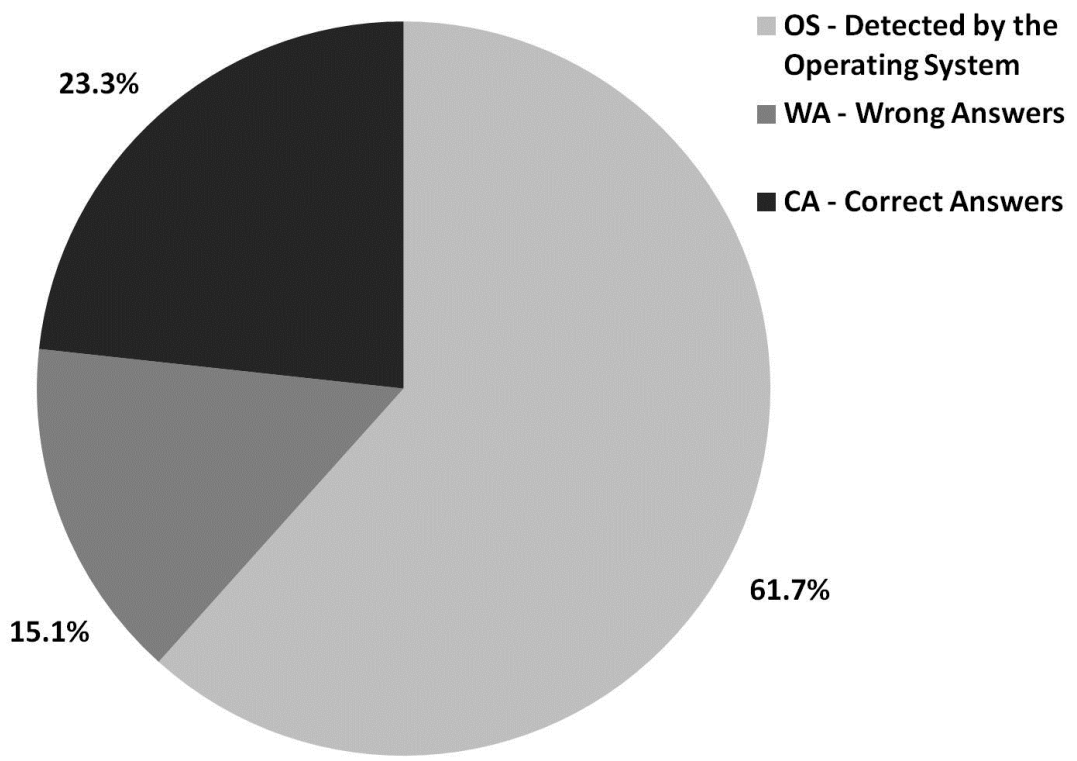


Figure 4: Effect of Control Flow Error on Application SW.

The above argument of why or why not CFC can be explained better if we can evaluate the level of protection provided by CFC techniques against soft errors. So that partly answers the next question - why evaluate CFC techniques? The response to this question can clear the air on validity of claims of protection made by different CFC techniques.The GemV-CFC toolset is used to evaluate purely hardware based, purely software based and hybrid CFC techniques and the conclusion derived from the results is that these techniques are ineffective in protecting the systems on which they are implemented, and generally tend to increase the vulnerability of the system making it more prone to transient errors. The quantitative evaluation of reliability before and after applying CFC techniques can provide an idea about the level of protection achieved by control flow checking techniques, whether it is positive or negative; negative as in reduction of reliability after applying the CFE detection technique due to

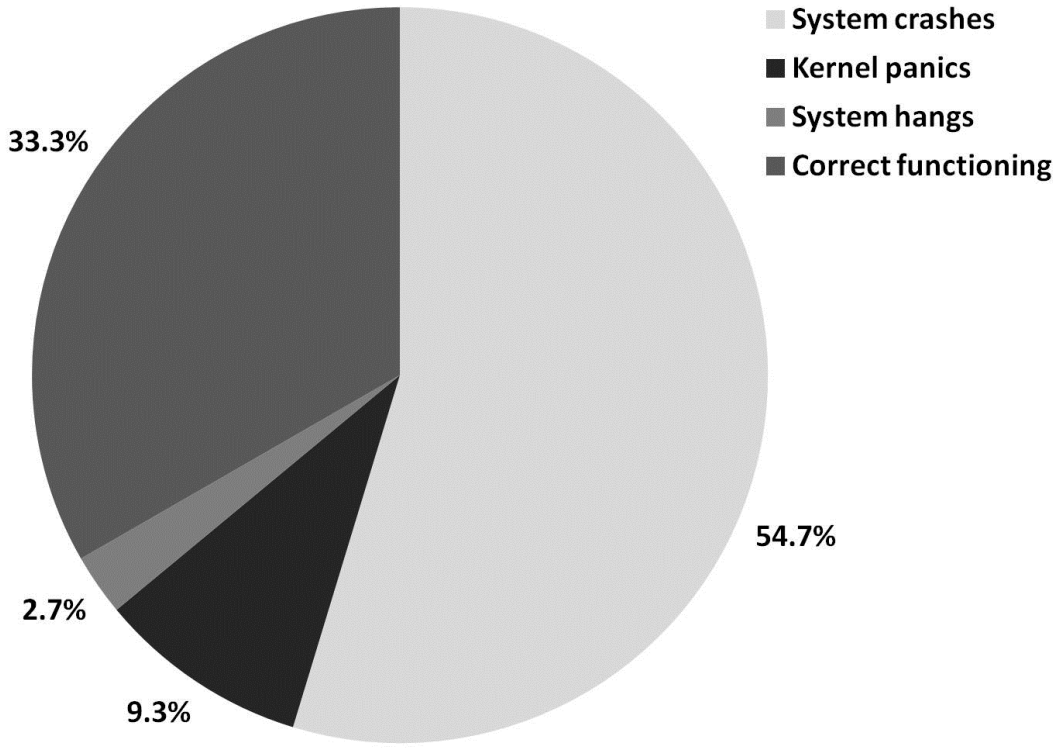


Figure 5: Effect of Control Flow Errors on Linux OS.

the additional instructions or overhead incurred in checking the correct control flow which leaves the system more vulnerable. In order to evaluate the effectiveness and the error coverage of existing CFC techniques, the level of protection acheived by these techniques is compared with an unprotected program running on an unprotected system. The introduction of the quantitative evaluation framework also makes possible the comparison of reliability of different CFC techniques. On a closer look, the evaluation can identify loopholes in existing CFC techniques by measuring the protection achieved for different processor hardware components which in turn gives an idea of the extent to which a CFC technique protects a particular category of control flow errors that are triggered by the hardware component. Thus, the component-wise evaluation can identify the processor components which are heavily prone to soft errors even after implementing the CFC scheme and can help the designer in exploring alternative cost-

effective techniques to protect such components. The framework also provides avenues to evaluate incremental modifications, and customizations of existing CFC techniques for different processor architectures, as well as assessment of novel CFC techniques against existing ones. It enables the designer to explore design tradeoffs between performance and reliability of a processor architecture, on implementation of control flow checking techniques, through simulations even before the RTL designs are available.

3.1 Mechanism of Control Flow based Protection

The philosophy behind control flow based techniques is that, when we analyze the data flow graph of a program, most of the data computations eventually result in one or more of the data involved in the execution of some control flow instruction (`branch` instruction). By verifying the execution of these control flow instructions, the computation can ensure correct execution of all the instructions involved in the data flow that lead to such control instructions. Chapter 4 discusses the error models and the sources of control flow errors. In this work, we develop a means to translate the implementation of a protection technique into its protection model for such control flow errors, and then translate the protection achieved in the form of system vulnerability reduction. The methods for deriving protection models are discussed in Chapter 6.

3.2 Need for Quantitative Evaluation

In the discussion of the control flow based error detection techniques provided in Chapter 4, we observe that relative comparisons between implementations are presented in the form of: (1) arguments for detection of increased control flow errors, thus resulting in improved error coverage, (2) experimental results showing differences in performance overhead between the implementations, (3) targeted fault injection experiments on a simulation model, demonstrating the effectiveness of the proposed technique. The inherent philosophy of control flow based error detection does not provide for a standard and unified method for evaluation of the proposed techniques. To the best of my

knowledge, no quantitative methodology has been proposed to accurately analyze the soft error protection achieved through the implementation of a control flow checking technique, or to compare with another design. The failure rate derived from the fault injection experiments, used here, though the closest measure of soft error rate, is plagued by the following disadvantages:

1. The fault injection experiments are all targeted fault injections, directed to analyze the error detection capabilities of the specific technique implementation, and are not representative of the error coverage provided to the processor as a whole. Through detailed analysis, the GemV-CFC framework contradicts the portrayal of the control flow based error detection techniques by targeted fault injection campaigns as effective.

2. The experiments cannot be easily ported across different processor architectures or configurations .

3. The targeted nature of the experiments used does not give an accurate measure for comparative analysis among different proposed techniques.

4. Though fault injection experiments result in an estimate of SER (Soft Error Rate) as a measure of system reliability, the significantly large number of simulation runs required make this metric unusable for quick and extensive design space explorations. The exhaustiveness of fault injection experiments required is shown in an example provided in Section 8.1.

3.3 Reliability Metric: Vulnerabilty

During the execution of an application in a processor, program data is temporarily stored on the numerous sequential elements in the processor (e.g., pipeline registers, buffers, cache blocks, register file, etc.). The data stored (even temporarily) on these elements is exposed to the effect of radiation induced transient errors and is therefore

vulnerable to soft errors. Mukherjee et al[25] defined the term *vulnerability* for the first time in this context and describe the same as the amount of time that sequential elements store active program data used by the program for its execution. System vulnerability is computed as the time period between component-accesses that expose vulnerable data in the hardware, which then can be accumulated over all the active components in the processor. In this work, the framework implements fine-grained bit-level vulnerability models in each of the architecture components in the processor (with the same implementation used for the cache in [38]); and compute system vulnerability in `bit-cycles`. To ascertain confidence of the reliability evaluations obtained using GemV-CFC, exhaustive fault injection experiments are performed on unit processor components to validate the vulnerability implementation of the simulation setup.

Vulnerability measurement tool	Availability	Vulnerability modeling for processor components	Validation	CFC protection modeling
Asim	Intel's proprietary tool	IQ, L1 data cache, DTLB, SQ, Execution Units	No	No
Sim-SODA (Sim-Alpha)	Publicly available	L1 data cache, DTLB, LQ, SQ, Victim Buffer, RF, ROB, FU, Instruction Window, Wake-up table	No	No
SS-SERA (SimpleScalar)	Publicly available	L1 data cache, L1 instruction cache, L2 unified cache, DTLB, ITLB, ROB, LSQ, IQ, FU, RF	No	No

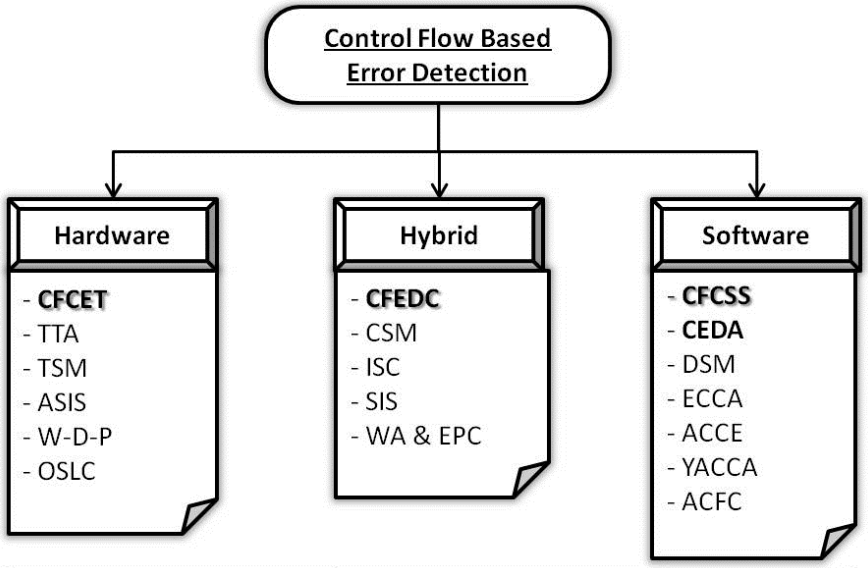
Table 1: The vulnerability measurement tools currently available, and their drawbacks.

Although there are couple of vulnerability estimation tools available, there are some drawbacks with these tools as shown in Table 1. The Asim simulator [9], equipped with vulnerability definitions and vulnerability measurements from [25, 4], is a proprietary tool and measures only the vulnerability of a selected set of processor components like Instruction Queue (IQ), L1 data cache, DTLB (Translation Lookaside Buffer for L1 data cache), Store Queue (SQ) and execution units. Compared to ASIM, the

Sim-SODA tool [11], based on Sim-Alpha simulator [7], models more processor components like L1 data cache, DTLB, LQ (Load Queue), SQ, Victim Buffer, RF, ROB (Reorder Buffer), FU (Functional Unit or Execution Unit), Instruction Window, and Wake-up Table (Instruction Window and Wake-up Table together form the IQ). The recently published SS-SERA tool [6], based on SimpleScalar simulator [3] models most of the processor components except pipeline registers like L1 data cache, L1 instruction cache, L2 unified cache, DTLB, ITLB, ROB, LSQ (Load Store Queue), IQ, FU, and RF (Register File). To the best of my knowledge, the vulnerability modeling in these tools is not validated. Moreover, none of these tools model the protection mechanisms of control flow checking techniques. To overcome these drawbacks, a new vulnerability estimation tool, ***GemV-CFC***, based on the popular cycle-accurate simulator, *gem5*, is developed and validated, and the modeling of protection mechanisms of control flow checking techniques is integrated into the tool.

Chapter 4

RELATED WORK



CFC technique	Error Model Coverage
CFCET	- A_3,B_2,B_3,C_2,C_3,C_4
CFEDC	- $A_3,B_1,B_2,B_3,C_1,C_2,C_3,C_4$
CSM	- $A_1,A_3,B_1,B_2,B_3,C_1,C_2,C_3,C_4$
SIS	- $A_1,A_3,B_1,B_2,B_3,C_1,C_2,C_3,C_4$
WA & EPC	- $A_1,A_3,B_1,B_2,B_3,C_1,C_2,C_3,C_4$
CFCSS	- C_1,C_2,C_3
CEDA & ACCE	- $A_1,A_2,A_3,B_1,B_2,C_1,C_2,C_3,C_4$
ECCA	- B_2,C_2,C_3
YACCA	- $A_1,A_2,A_3,B_1,B_2,C_1,C_2,C_3,C_4$

Figure 6: Classification of control flow based error detection techniques into purely hardware, purely software and hardware-software (hybrid) solutions, alongwith their detection coverage of error models.

With a vast variety of soft error detection and correction techniques, we need a classification of the underlying techniques. The techniques can be categorized into control flow checking techniques, redundancy based mechanisms, and a combination

CFC techniques	Evaluation methodology	Performance Overhead Evaluation	Area/Power Overhead Evaluation
Hardware based schemes			
CFCET	Analytical estimate	Time Stamp Counter	Synthesis (only area)
ASIS	Arguments	NM	Count additional bits
W-D-P	Arguments based on error models	Memory overhead	Count of gates and extra bits
OSLC	Fault injection	NM	NM
Hardware-software based (hybrid) schemes			
SIS	Fault injection	Code size, memory overhead	Count of gates and bytes of memory
CSM	Arguments and equations	Analytical estimates	Memory overhead (average block size and extra bits)
WA & EPC	Equations	NM	NM
CFEDC	Analytical equations based on error models	Simulations, Memory overhead	Synthesis (area and power)
Software based schemes			
CFCSS	Targeted fault injection	Actual execution time overhead, Code size	N/A
ECCA	Targeted fault injection	NM (but available from [44])	N/A
CEDA	Targeted fault injection	Simulations (Performance and Memory overheads)	N/A
ACCE	Targeted fault injection	Actual execution time and Memory overheads	N/A
YACCA	Targeted fault injection	Actual execution time and Memory overheads	N/A
ACFC	Targeted fFault injection	Analytical estimates (number of extra instructions)	N/A

Table 2: Evaluation methodologies used for measuring error coverage, performance, area and power overheads in various control flow checking based soft error protection techniques.

of both. Redundancy based techniques usually check the data flow by duplicating the instructions in a program and comparing the results from the original and duplicate stream of instructions. A classic example of redundancy based mechanisms is EDDI by Oh et al [30]. On the other hand, control flow checking techniques, which are the focus of this research, ensure the execution of correct sequence of instructions (which is fixed given a program input), even in presence of soft errors. Control flow techniques can be further classified into hardware based, software based, and hybrid which is a combination of both. The software approaches come with desirable advantages over hardware techniques, like cost effectiveness, reduction in design time, power savings, reduction in chip area, and scalability to future systems, but at the cost of performance overhead.

4.1 Hardware based Control Flow Checking techniques

Hardware control flow checking mechanisms generally rely on dedicated monitoring hardware like watchdog processor or additional hardware within the processor core to continuously monitor the control flow by comparing the runtime signatures with reference signatures, or instruction addresses with stored addresses, or by verifying the integrity of signatures using error correction codes. TTA [23] decomposes the application program into blocks and checks the execution time of blocks using timers in the watchdog processor. During runtime, the start address and size of a block are monitored to generate the block exit address, and a mismatch in the observed exit address and the generated exit address indicates an error. CFCET uses execution tracing to transmit the runtime branch instruction address and branch target address to an external watchdog processor, which compares them with the reference addresses stored in its associative memory. The coverage of the error models are shown in Figure 6, using the subscripted letters from the classification given in Figure 3. ASIS [8] allows control flow checking of several processors using a hardware signature generator, and a watchdog monitor.

The watchdog receives the cumulative signature of instructions in a block and can detect the start and end of the block whereas the monitor compares the signature using a reference signature graph and detects mismatches. Similarly, W-D-P [22] verifies the control flow using a reference control flow graph loaded in the watchdog processor. OSLC [20] also uses a similar mechanism but here the program is divided into segments which are further divided into blocks. As shown in Table 2, most of the hardware based techniques use analytical estimates (CFCET [33]) or functional arguments (ASIS [8]) or arguments based on error models (W-D-P [22]) to justify their respective error coverage, while OSLC [20] uses generic fault injection on address, control and data lines to obtain its error coverage. Hardware techniques, in general, provide area overhead information based on synthesis or simply by counting the additional hardware bits and gates required to generate runtime signatures and monitor them. To provide an idea of performance overhead, CFCET uses time stamp counters to measure execution time, while W-D-P provides memory overhead involved. Here, NM in the table means not measured.

4.2 Hybrid Control Flow Checking techniques

Hybrid control flow checking techniques generally involve code modifications using the compiler, and modifications to the processor hardware, to assist in monitoring the control flow. SIS [37] relies on signatured instruction streams for continuous monitoring of signatures of the sequence of executed instructions using a watchdog processor. Branch Address Hashing [36] (BAH) in SIS involves hashing of the branch instruction with its associated signature causing the branch address to be incorrect at compile time, and its restoration to the original value at runtime. Two dimensional signature in CSM [45] combines horizontal and vertical signatures where vertical signature is for an interval of instructions that constitute multiple blocks, and horizontal signature adds extra bits to each instruction word in the horizontal direction. Watchdog assist [35]

transmits the checksum of executed instructions to the watchdog processor at the start of each branch free execution block, where the watchdog starts subtracting the instructions and expects an all-zero result at the end of the block. CFEDC [10] modifies the fetch and decode pipeline stages with a combinational circuit to correct any errors in the control instruction preceded by a hamming code of the control instruction. SIS [37] performs generic fault injection on address, data and control buses to collect its fault coverage numbers. On the other hand, CSM [45] makes use of arguments based on error models and probability based equations to extract its error coverage information. Watchdog assists and extended precision checksums [35] use equations based on probability generating functions to estimate the error coverage, while CFEDC [10] uses simple analytical equations based on error models. CFEDC provides area and power overheads using RTL synthesis, whereas SIS counts the additional gates required in its hadrware portion of control flow checking and the extra bytes of memory added to the assembler and loader code. CSM provides the memory overhead using average block size information and the extra bits added to the instructions. To provide performance overhead, SIS resorts to code size and memory overheads, while CSM generates analytical estimates of performance loss using average block sizes and weighted-average performance overhead per control-flow construct, and CFEDC performs simulations using modelsim to obtain the execution times, and extracts the memory overhead incurred.

4.3 Software based Control Flow Checking techniques

Software tecnhiques are characterized by program modification by compilers or binary translators to insert software signatures that represent the control flow of the program, and verification of the runtime generated signatures by comparing them with the preassigned sinatures. CFCSS inserts a signature comparison at the start of each basic block after transforming the previous block's signature to that of the current block.

ECCA [2] fortifies the blocks with assertions and raises a divide by zero exception using a set of equations based on the block identifier and the product of permissible block IDs from the current block. CEDA [42] differs from CFCSS in the careful assignment of signatures to avoid the classic aliasing problem between legal and illegal branches when multiple nodes share multiple branch fan-in nodes as their destination nodes. ACCE [43] builds on the CEDA infrastucture by providing the correction capability. Local and global function error handlers detect illegal jumps from the current function and restore the control flow to the function where the error occured, respectively. YACCA [13] claims to cover even the errors not crossing the block boundaries, and avoids the branches in control flow checking code. Moreover, like CEDA, it repeats the condition checking of a conditional branch at the start of the target nodes for both true and false clauses. ACFC [44] reduces the execution overhead by combining the instrumentation of multiple basic blocks using one instruction and assigning a parity bit per basic block. The software techniques (CFCSS, ECCA [2], CEDA [42], ACCE [43], YACCA [13], and ACFC [44]), in general, perform targeted fault injection experiments to gather their fault coverage numbers. Also, most techniques provide performance overhead data using actual executions or simulations of benchmarks. Only ACFC provides its performance overhead using analytical estimates based on number of extra instructions. Since there is no additional hardware involved, these techniques don't provide area or power overheads.

Chapter 5

CONTRIBUTIONS

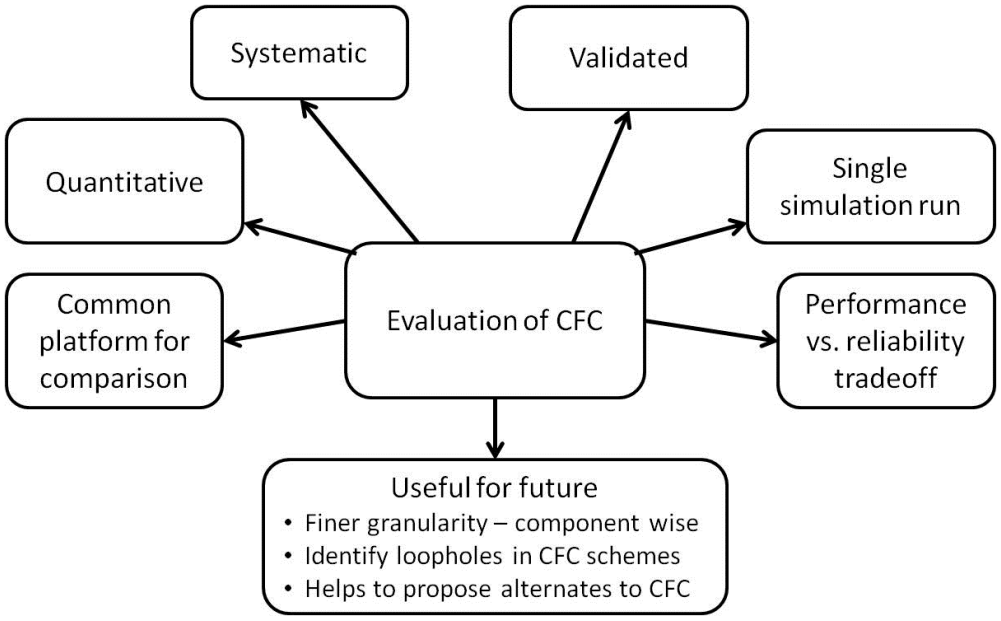


Figure 7: The requirements of a better vulnerability evaluation framework.

The major contributions from this work are:

1. The work develops and validates a comprehensive simulation based reliability estimation tool – **GemV-CFC**. The vulnerability measurements are based on the vulnerability and *Architecural Vulnerability Factor (AVF)* definitions from [25]. The fact that the tool is thoroughly validated, as detailed in Section 8.1, adds to the robustness of the framework. The core requirements of a vulnerability estimation tool for control flow checking, as shown in Figure 7, are satisfied by the GemV-CFC tool.

2. We develop a systematic methodology to model the protection mechanism of control flow checking tehcniques. The systematic approach is quite generic and can be applied to any control flow checking scheme.

Chapter 6

THE SYSTEMATIC APPROACH TO DERIVE PROTECTION MODEL FOR CFC

The systematic approach to derive the protection model for a control flow checking mechanism tries to answer the following questions:

1. What is a CFC technique trying to protect?
2. How is the CFC scheme achieving the protection?

In order to answer what CFC is trying to protect, we need a deifnition for a control flow error. A control flow error causes a deviation from an expected sequence of execution, given a program input.

1. This translates to an expected program counter (PC) transition from $PC = i$ at cycle t to $NPC = j$ at cycle $t+1$, where NPC implies the next instruction's PC.
2. Instead, due to a soft error, an erroneous $NPC = k$ is loaded at cycle $t+1$.

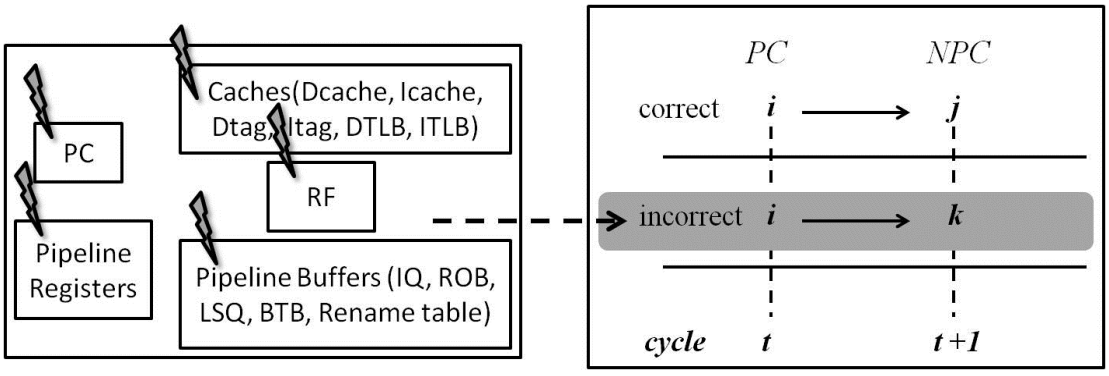


Figure 8: The error in various processor components can lead to an erroneous PC transition.

There are various components in the processor, as shown in Figure 8, which are exposed to bit flips and can possibly cause erroneous PC transitions. The obvious

question that arises is how can a bit flip cause an erroneous value to be loaded in PC. In order to provide a solution to the above questions, a systematic flow of steps to derive the protection model of CFC schemes is explained. The decision points in the flow will provide the required solutions.

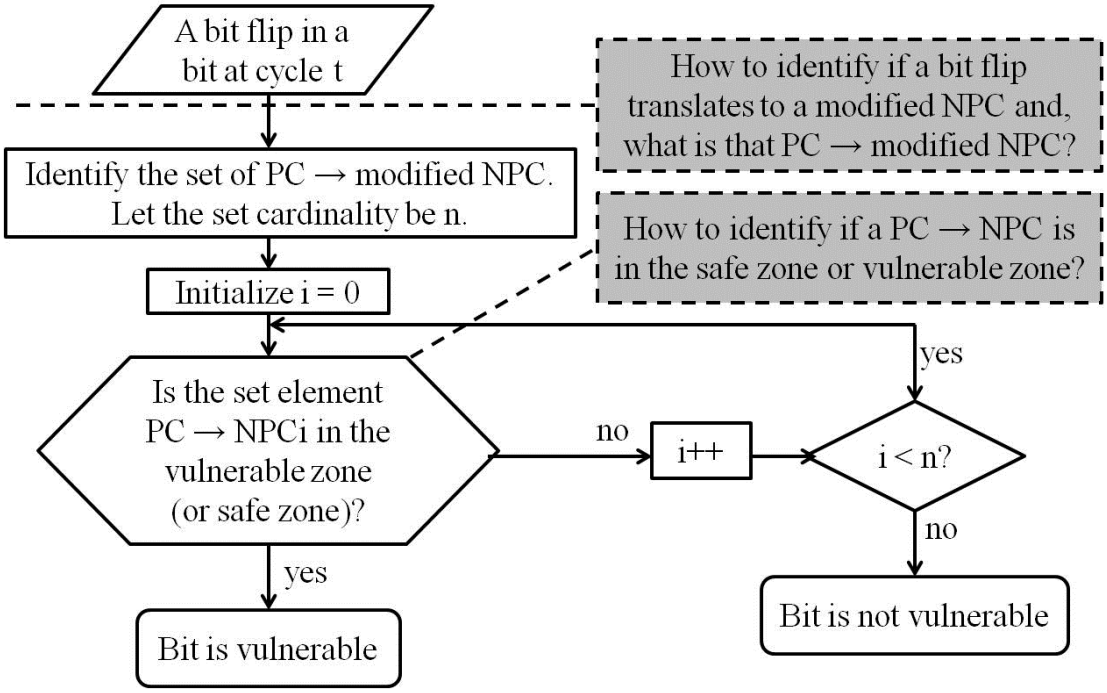


Figure 9: The systematic methodology to derive the protection model of control flow checking mechanisms.

As shown in Figure 9, we consider how a bit flip can translate to a erroneous PC to NPC transition and given such a transition, whether it can be detected by the CFC mechanism or not. The flow of the systematic methodology proceeds as follows:

1. A bit flip happens in a processor component in cycle *t*.

2. The bit flip translates to multiple erroneous PC to NPC transitions in later cycles. Let the cardinality of the set of such erroneous pairs of PC and NPC be *n*.

3. For each incorrect PC to NPC transition, given the value of the PC and the incorrect next PC, determine whether the control flow checking technique can detect

the error or not. If the CFC can detect the error, the PC to NPC transition is considered to be part of the *safe zone*. On the other hand, if it cannot be detected, it is considered to be in the *vulnerable zone*.

4. If at least one pair of PC and NPC is in the vulnerable zone, the bit is considered vulnerable in cycle *t*. If all the incorrect PC to NPC transitions due to a bit flip fall under the safe zone, the bit essentially cannot cause a control flow error and can be deemed not vulnerable in cycle *t*.

The flow poses two questions which require further analysis:

1. How to identify whether a bit flip can translate to erroneous PC to NPC transitions, and what are the corresponding PC to modified NPC values?

2. Given the values of PC and NPC in an incorrect PC to NPC transition, how does the CFC determine whether it is in the safe zone or vulnerable zone?

6.1 Identification of Safe Zone and Vulnerable Zone

To identify if an incorrect pair of PC and NPC can be detected, the CFC scheme has to spell out the categories of control flow errors that it can detect. For example, let us conisder the software based CFC scheme, CFCSS. As shown in Figure 10, RSR or *Runtime Signature Register* holds the identifying signature of a basic block during runtime. At the start of the basic block **BB3**, the signature of the predecessor basic block **BB2** is expected in RSR. The control flow checking code at the beginning of the basic block transforms this expected value in RSR to match the signature of BB3 using **XOR** operation. It compares the RSR with the signature value of BB3 and a mismatch will flag an error. Now, if an incorrect jump takes place from BB3 to BB2, the RSR will be holding the signature of BB3 while executing instructions in BB2. When it jumps to BB3 at the end of BB2, RSR holds an incorrect signature value of BB3, instead of

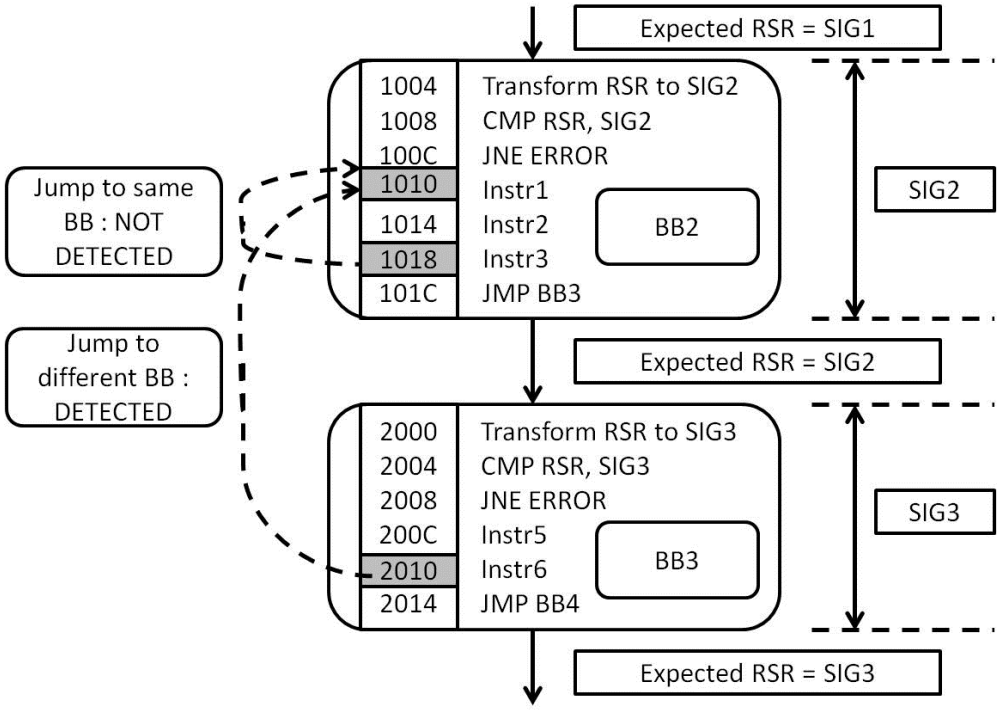


Figure 10: CFCSS can detect an incorrect jump from a basic block to a different basic block, but not to the same basic block.

the expected signature of BB2. The transformation will place an incorrect signature in RSR for comparison, triggering a call to flag the error. Therefore, an incorrect PC to NPC transition to a different basic block can be detected by CFCSS. On the other hand, if an incorrect branch happens within the basic block BB2, the RSR will still hold the signature of BB2 while executing BB2, and the transformation in BB3 will execute smoothly without flagging an error. This implies an erroneous PC to NPC transition to the same basic block cannot be detected by CFCSS.

After a thorough analysis of CFCSS, we generated a table which maps different categories of PC to NPC transitions to safe zone (SZ) or vulnerable zone (VZ), as shown in Table 3. Here, in Table 3 (b), corresponding PC and NPC categories from Table 3 (a) are marked as 1 and rest all categories are marked as 0. For example, for the first entry in Table 3 (a), since a PC to NPC transition from a basic block comprising

PC	NPC	SZ or VZ
O	OS	VZ
O	OD	SZ
O	CL	VZ
O	CA	VZ
O	CO	SZ
C	OS	SZ
C	OD	SZ
C	CL	SZ
C	CA	SZ
C	CO	SZ

O – Original Source Code
C – CFC Code

(a)

PC		NPC					SZ or VZ
O	C	OS	OD	CL	CA	CO	0 or 1
1	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	1
1	0	0	0	0	1	0	1
1	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	0	0	1	0	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	0	1	0

OS – Original Same BB
OD – Original Different BB

(b)

CL – CFC legal target
CA – CFC aliased target
CO – Other CFC code

Table 3: The CFCSS mapping of PC to NPC categories to safe or vulnerable zones.

original source code (excluding the additional control flow checking code) to the same original source code basic block cannot be detected by CFCSS, it falls under vulnerable zone. For the corresponding entry in Table 3 (b), the original source code sub-category (marked as O) for PC is shown as 1, and the same basic block original source code sub-category (marked as OS) is shown as 1 for NPC. Rest all sub-categories in the table entry are marked as 0. A 1 in the last column indicates vulnerable zone, whereas a 0 implies safe zone.

Given the mapping table and the set of incorrect PC to NPC values due to a bit flip in some processor component, the flow in Figure 11 can be used to identify the decision of a CFC scheme to place a PC to NPC transition in the safe or the vulnerable zone. First, it determines the category of a given PC to NPC transition. For the

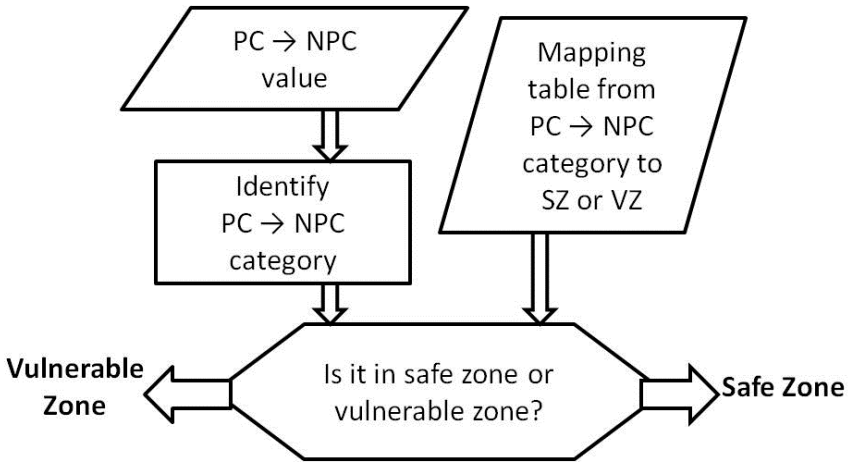


Figure 11: Control flow checking technique places an incorrect PC to NPC transition in safe or vulnerable zone based on its mapping table of PC to NPC transition categories to safe/vulnerable zones. For example, the mapping table for CFCSS is given in Table 3.

previous example of CFCSS (shown in Figure 10), given the PC and NPC values and the boundaries of basic blocks, we can determine whether the PC falls in a basic block comprising original source code or control flow checking code. We can also determine if the NPC falls in the same basic block of original source code or a different basic block, given the basic block boundaries. Once the category of PC to NPC transition is known, the mapping table in Table 3 can be used to determine whether the transition falls under the safe zone or vulnerable zone. For example, for the incorrect jump from a basic block of original source code (**BB3**) to a different basic block of original source code (**BB2**), shown in Figure 10, the second entry in Table 3 maps to safe zone. On the other hand, the incorrect jump from a basic block to the same basic block of original source code (**BB2**) is mapped to vulnerable zone in the first entry in the table. Similarly, the mapping tables for other CFC techniques like CFCSS+NA, CEDA, CFEDC and CFCET are shown in Table 4 a-d.

6.2 Determining the Set of Incorrect PC to NPC Values

To determine the incorrect PC to NPC values due to a bit flip in a processor component, we assume an inorder processor with no branch prediciton, an ECC protected L1 data

PC	NPC	SZ or VZ
O	OS	VZ
O	OD	SZ
O	CL	VZ
O	CA	SZ
O	CO	SZ
C	OS	SZ
C	OD	SZ
C	CL	SZ
C	CA	SZ
C	CO	SZ

PC		NPC					SZ or VZ
O	C	OS	OD	CL	CA	CO	0 or 1
1	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	1
1	0	0	0	0	1	0	0
1	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	0	0	1	0	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	0	1	0

O – Original Source Code
C – CFC Code

OS – Original Same BB
OD – Original Different BB

CL – CFC legal target
CA – CFC aliased target
CO – Other CFC code

(a) CFCSS+NA

PC	NPC	SZ or VZ
OB	OB	SZ
OB	OO	SZ
OB	C	SZ
OO	OB	VZ
OO	OO	VZ
OO	C	VZ
C	OB	SZ
C	OO	VZ
C	C	VZ

PC			NPC			SZ or VZ
OB	OO	C	OB	OO	C	0 or 1
1	0	0	1	0	0	0
1	0	0	0	1	0	0
1	0	0	0	0	1	0
0	1	0	1	0	0	1
0	1	0	0	1	0	1
0	1	0	0	0	1	1
0	0	1	1	0	0	0
0	0	1	0	1	0	1
0	0	1	0	0	1	1

OB – Original Source Code Branch
OO – Other Original Source Code

C – CFC Code

(b) CFEDC

PC	NPC	SZ or VZ
O	OS	VZ
O	OD	SZ
O	CL	VZ
O	CA	SZ
O	CO	SZ
C	OS	SZ
C	OD	SZ
C	CL	SZ
C	CA	SZ
C	CO	SZ

PC		NPC					SZ or VZ
O	C	OS	OD	CL	CA	CO	0 or 1
1	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0
1	0	0	0	0	1	0	0
1	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	0	0	1	0	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	0	1	0

O – Original Source Code
C – CFC Code

OS – Original Same BB
OD – Original Different BB

CL – CFC legal target
CA – CFC aliased target
CO – Other CFC code

(c) CEDA

PC	NPC	SZ or VZ
OB	O	SZ

PC	NPC	SZ or VZ
OB	O	0 or 1
1	1	0

OB – Original Source Code Branch
O – Original Source Code

(d) CFCET

Table 4: The mapping of PC to NPC categories to safe or vulnerable zones for different CFC schemes.

cache and a parity protected L1 instruction cache, as shown in Figure 12. The assumption simplifies the discussion, although the approach is not restrictive and can be easily extended to out-of-order processors with advanced branch prediction mechanisms.

Let us consider the vulnerability of PC bits. A bit flip in PC can cause an erroneous NPC value to be written into the PC if the MUX select bit in Figure 12 indicates a non-branch instruction. Let us assume a **non-branch** instruciton in the

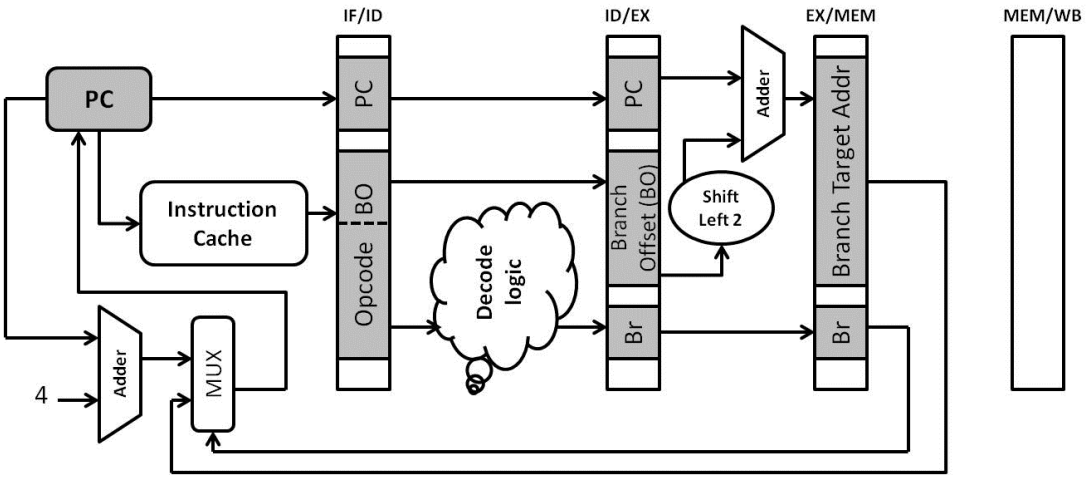


Figure 12: A simplistic 5-stage inorder pipeline.

current cycle. A bit flip in the PC causes a 1-bit hamming distance value to be held in the PC. The corresponding PC and NPC values in such an erroneous transition will be as follows:

$$PC = Current\ PC \tag{6.1}$$

$$NPC = (1\text{-}bit\ hamming\ distance\ Current\ PC) + 4 \tag{6.2}$$

Inorder to derive the vulnerability of PC for the current cycle, the next PC values for every 1-bit hamming distance values of current PC are calculated and these PC to NPC values are provided as inputs to the safe/vulnerable zone determination using the mapping table for the CFC mechanism under consideration. If the MUX select bit indicates a **branch** instruction, the erroneous PC values are overwritten in the next cycle with values from the **branch target address** field in the **execute-memory** pipeline register. In that case, since the erroneous PC value is not used to fetch instructions, the PC bits can be considered not vulnerable for the current cycle.

Now let us consider the vulnerability of the branch target address field in the execute-memory (EX/MEM) pipeline register. If the MUX select bit, which is derived from the **branch/non-branch control bit** in the execute-memory pipeline register, in-

dicates a branch instruction in the current cycle, a bit flip in the branch target address field can cause an erroneous value to be written in the next cycle. For this incorrect PC to NPC transition, the corresponding PC and NPC values are given as:

$$PC = Current\ PC \tag{6.3}$$

$$NPC = 1\text{-}bit\ hamming\ distance\ EX/MEM\ Branch\ Target\ Address \tag{6.4}$$

Applying similar logic to that of PC, the bits in branch target address field in the pipeline register are deemed vulnerable or non-vulnerable based on the safe/vulnerable zone determination of the corresponding PC to NPC values for every 1-bit hamming distance value. The bits can be considered non-vulnerable if the MUX select bit indicates a non-branch instruction for the current cycle.

To measure the vulnerability of the **branch/non-branch control bit** in the EX/MEM pipeline register, we consider two cases:

1. The bit indicates a branch instruction and is flipped to indicate a non-branch instruction in the current cycle.

2. The bit shows a non-branch instruction and is flipped to indicate a branch instruciton in the current cycle.

In the first case, the erroneous PC value is supplied to the adder from the PC itself to increment by 4, as the MUX select bit indicates the instruction in the current cycle to be non-branch, instead of taking the value from the branch target address field in the EX/MEM pipeline register for a normal branch instruction. The corresponding PC and NPC values are:

$$PC = Current\ PC \tag{6.5}$$

$$NPC = Current\ PC + 4 \tag{6.6}$$

On the other hand, in the second case, the erroneous PC value is read from the branch target address field in the EX/MEM pipeline register, as the MUX seelct bit indicates the instruction in the current cycle to be a branch instruction, instead of supplying the value of PC to the adder to increment by 4 for the actual non-branch instruction. The corresponding PC and NPC values are:

$$PC = Current\ PC \tag{6.7}$$

$$NPC = EX/MEM\ Branch\ Target\ Address \tag{6.8}$$

Now let us move to the fields in the decode-execute (ID/EX) pipeline register. For the **branch offset** field in the ID/EX pipeline register, a bit flip can cause an erroneous value to be used in the execution to calculate branch target address for the next cycle, provided the **branch/non-branch control bit** in the ID/EX pipeline register shows a **branch** instruction in the current cycle. The PC and NPC values for the corresponding incorrect PC to NPC transition can be derived as:

$$PC = PC\ field\ in\ ID/EX\ pipeline\ register \tag{6.9}$$

$$\begin{aligned} NPC = {} & (\text{1-}bit\ hamming\ distance\ ID/EX\ Branch\ Offset)_{shift\ left\ by\ 2} \\ & + (PC\ field\ in\ ID/EX\ pipeline\ register) \end{aligned} \tag{6.10}$$

For the PC field in the ID/EX pipeline register, the same equations can be used except that the 1-bit hamming distance is applied in that pipeline PC field. For the PC and branch offset fields in the fetch-decode (IF/ID) pipeline register, the same equations apply. For the opcode field in the IF/ID pipeline register, the same logic applies for PC to NPC calculation as the **branch/non-branch control bit** in the ID/EX pipeline register, except that the PC to NPC values for every 1-bit hamming distance opcode that can transform a branch instruction to non-branch instruction (and vice-versa) have to be considered. This completes the vulnerability calculations of PC and pipeline registers

in an inorder pipeline as the instruction field in the IF/ID is fetched from L1 instruction cache which is considered protected using parity.

6.3 Direct and Indirect Sources of Control Flow Errors

Until now, we considered control flow errors caused due to bit flips in processor components like PC and pipeline registers. The errors in these processor components are read only once and have direct impact on the NPC value, mostly in the next cycle or a subsequent cycle. The values in these registers will be overwritten every cycle during normal execution. Such processor components are classified as *direct* CFE sources. On the contrary, bit flips in the processor components like register file (RF), and pipeline buffers like instruction queue (IQ), load store queue (LSQ), branch target buffer (BTB), and register rename table, can be retained for multiple cycles and can be read multiple times. And these erroneous values can be passed on to other processor components before reflecting in a control flow error. Therefore, the bit flips in these components can cause multiple erroneous PC to NPC transitions in later cycles. These processor components are classified as *indirect* CFE sources. For the ARM architecture, under consideration, there are two cases that can cause such *indirect* control flow errors.

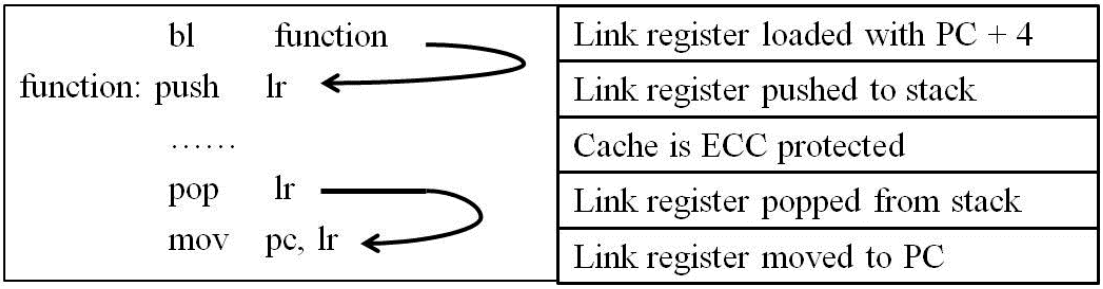


Figure 13: The register move to PC from link register causing an *indirect* control flow error.

The first case is an instruction involving a register move from one of the 16 architectural registers in ARM to the ARM PC register (register 15). For example, most of the compilers use the ARM **mov pc, lr** instruction to accomplish this where **lr**

denotes the link register. This instruction is used to restore the return address at the end of a function. The sequence of instructions associated with this register move to PC are shown in Figure 13. In the function call, the link register is loaded with the PC of the next instruction.The first instruction in the function body will push the link register onto the stack. Now the link register value is safe in ECC protected memory. Just before the register move to PC to restore the return address, the link register is popped from the stack. Note that the link register holds the vulnerable value that can corrupt PC, but the vulnerability interval for this link register usage is quite short. It involves a one cycle interval at the function call and another 1-cycle interval before restoring the return address. Although this is a minor portion of the link register usage, we give the benefit of the doubt to the control flow checking schemes and assume them to protect the link register completely.

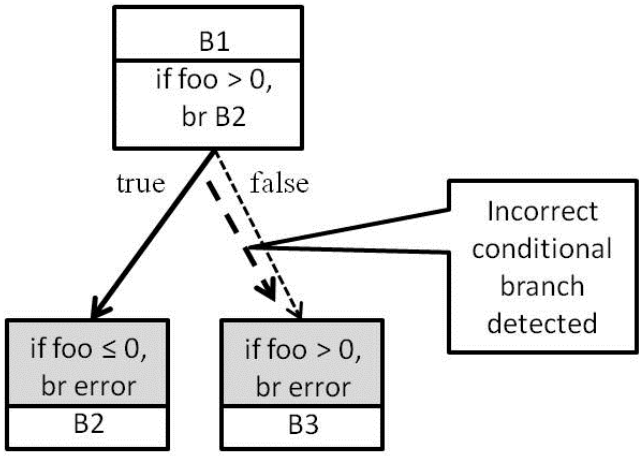


Figure 14: The jump check instructions in CEDA to provide protection against error in branch condition.

The second case involves a bit flip in a *indirect* source like a register in the RF causing an error somewhere in the data flow leading to the branch condition. Usually the branch direction in a conditional branch instruction is determined by a comparison which involves two registers or a register and an immediate value. These registers being

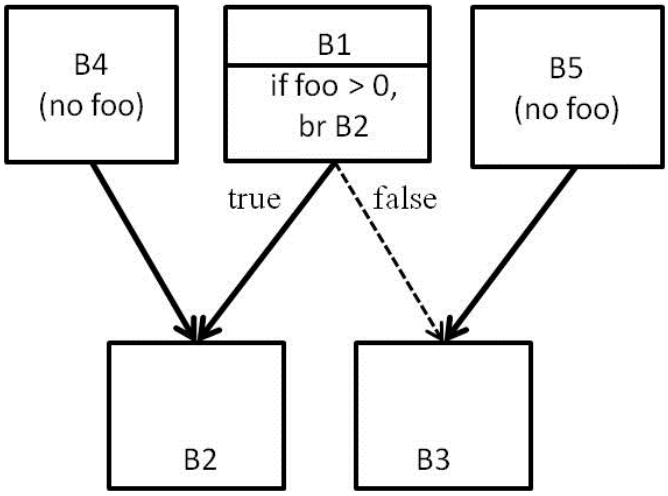


Figure 15: The jump check instructions in CEDA cannot be inserted if the legal branch target basic blocks have multiple basic blocks as their predecessors, which may not use the same branch comparison variables.

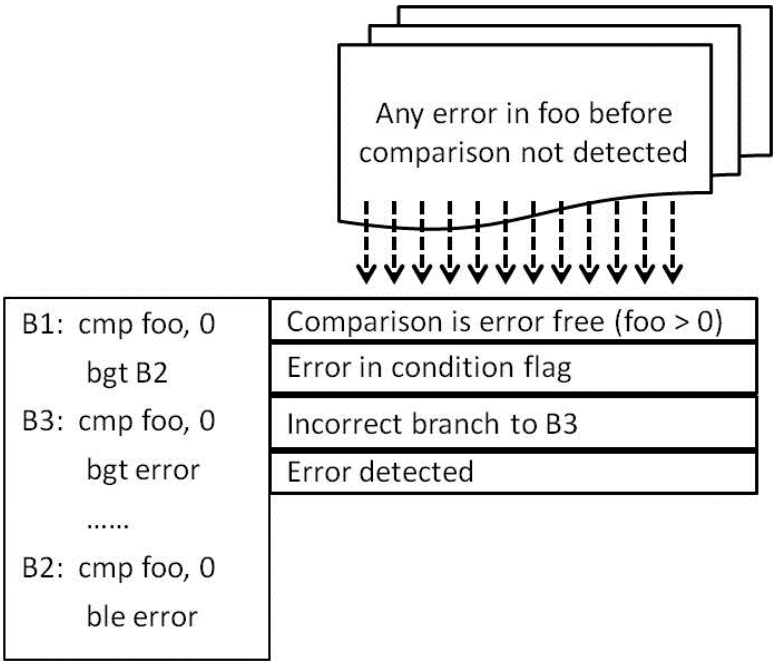


Figure 16: The jump check instructions in CEDA can only protect error in condition flag after the execution of comparison instruction and before the execution of conditional branch instruction.

compared may have derived their values from arithmetic, register move or memory load instructions executed before the comparison operation. The source registers used

in the arithmetic or register move operations have derived their values from arithmetic, register move or memory load instructions executed earlier, and this continues as a data flow chain. A bit flip in any processor component used in this data flow chain like the registers in the RF or the entries in the pipeline buffers can cause an error in the data values, leading to the branch condition to be erroneous. Therefore, instead of taking the expetced "true" direction of the conditional branch for the given input, it may be forced to fall through in the "false" direction or vice versa. To the best of my knowledge, there are only two control flow checking schemes that try to provide protection against control flow errors due to errors in branch condition - CEDA and YACCA. They use *jump check* conditions as shown in Figure 14. It shows a branch being taken from basic block $B1$ to $B2$ if the variable $foo > 0$ or the branch falls through from $B1$ to $B3$ if the condition is false. At the start of $B2$ and $B3$, the condition is checked again and if it deviates from the expected values, an error is flagged. The jump check is possible only if the branch target basic blocks have the basic block with the corresponding conditional branch as its only predecessor. If they have multiple predecessors as shown in Figure 15, as is mostly the case, the jump check will fail even in an error-free execution as these predecessor blocks may not generally use the *foo* variable. Even in the cases where jump check instructions can be inserted, it protects only the condition flag bits in the processor status register after the comparison and before the conditional branch is executed, as shown in Figure 16. Again, this represents a minor portion of the vulnerable period of the processor status register, and to give the benefit of the doubt to the CFC schemes employing the jump check instructions (like CEDA), the processor status register is considered not vulnerable.

6.4 Coverage of Control Flow Error Models

Based on the application of the systematic methodology on the CFC techniques used as case studies, Table 5 identifies the control flow errors that are covered (marked as **D**) by

	Detected Control Flow Errors				
Protection Technique	**PC**	**Branch Condition**	**Branch Insert**	**Branch Delete**	**Branch Offset**
CFCSS	D	-	D	-	D
CFCSS+NA	D	-	D	-	D
CEDA	D	D	D	D	D
CFEDC	D	-	D	D	D
CFCET	-	-	D	-	D

Table 5: Coverage of error models for the Case Study experiments.

these protection techniques. Here, we can observe that only a subset of the control flow errors are covered by each. From our analysis of protection models, we can observe that soft errors in only a few processor hardware components are actually detected by the control flow protection techniques proposed - PC, pipeline registers, link register and processor status register.

Chapter 7

THE GEMV-CFC FRAMEWORK

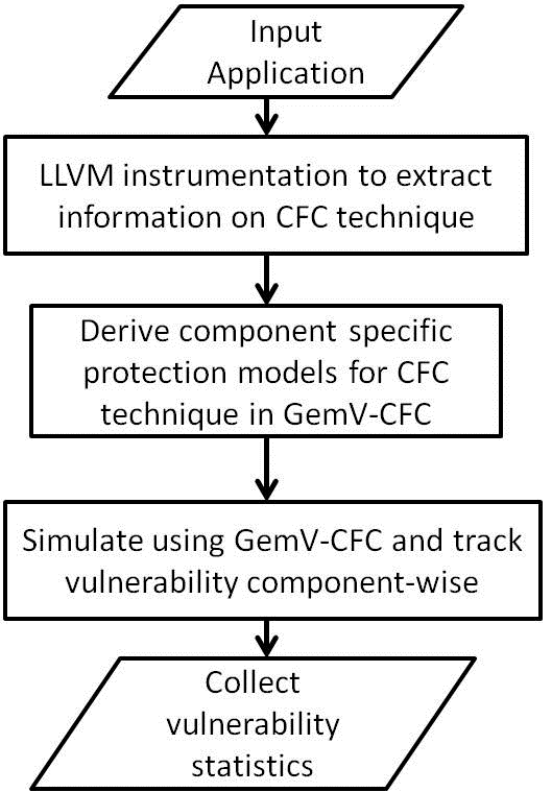


Figure 17: The approach in a flowchart.

The implementation framework for quantitative evaluation of control flow checking techniques is called ***GemV-CFC*** to emphasize the usage of the modular, discrete event driven and cycle-accurate computer system simulator, ***gem5***, to track the vulnerability of processor components when programs are hardened using control flow checking mechanisms. The ***V*** in ***GemV*** stands for vulnerability. The working of this framework is described in the following steps and the flow is shown in Figure 17:

1. Given an application program as the input, instrument the compiler to extract information on the control flow checking technique to assist the simulator in tracking vulnerability of processor components protected by the CFC mechanism. The

purely software based and software portion of the hybrid CFC mechanisms are generally implemented in the compiler.

2. Derive the component specific protection models as applicable to the CFC technique in the cycle accurate microarchitectural simulator, *gem5*. Note that the purely hardware based and hardware portion of the hybrid CFC techniques are implemented in the gem5 simulator.

3. Along with the protection models for control flow checking mechanisms, implement vulnerability tracking functionality in gem5 simulator to form the GemV-CFC framework, and simulate the input application using the GemV-CFC.

4. Collate the vulnerability statistics provided by the GemV-CFC.

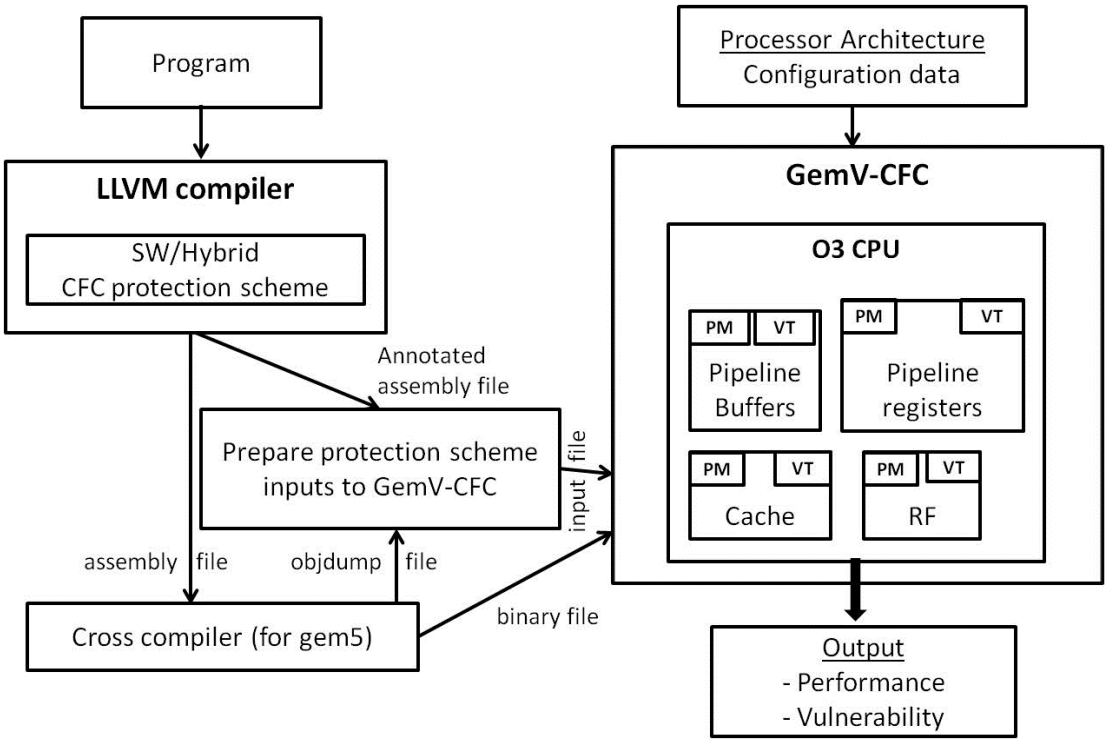


Figure 18: The overview diagram of the infrastructure.

The overall framework as shown in Figure 18 illustrates the flow of the input program through different components of the framework. The input application source

code is fed to the compiler. LLVM compiler framework[19] is used to instrument the CFC techniques and to gather information required to aid vulnerability tracking of the processor components in the GemV-CFC simulator to derive the protected component of vulnerability. For example, the control flow checking instructions can be marked, or the boundaries of basic blocks and control flow checking instructions can be marked in order to assist vulnerability tracking as per the protection model of the CFC used in the simulator. The significance of such inputs from LLVM compiler to GemV-CFC simulator, along with other components in the overview diagram will be explained in detail shortly in the next few sections as we go through the simulator framework and the case study of different control flow checking mechanisms where the GemV-CFC toolset is applied to analyse the protection achieved.

As part of the GemV-CFC framework, a protection model or PM is derived for the control flow checking for each hardware component intended to be protected by the CFC mechanism, as explained in Chapter 6. The protection model encapsulates the level of protection achieved by the control flow checking method when applied to a processor component. A control flow error in a processor component can be protected by the control flow technique to some extent or can be protected completely. The extent to which the processor component is protected can involve coverage for only some of the sub-components or some bits in the register or the sequential storage element being considered.

The vulnerability tracker module or VT in the GemV-CFC framework provides the capability and interfaces to track vulnerability of the processor components. The VT design is quite generic and scalable, and is customizable for each processor component as the functionality entails. It can calculate the total vulnerability and the protected vulnerability of the corresponding processor component. The *effective vulnerability* of an architectural component upon implementing the CFC technique is derived by de-

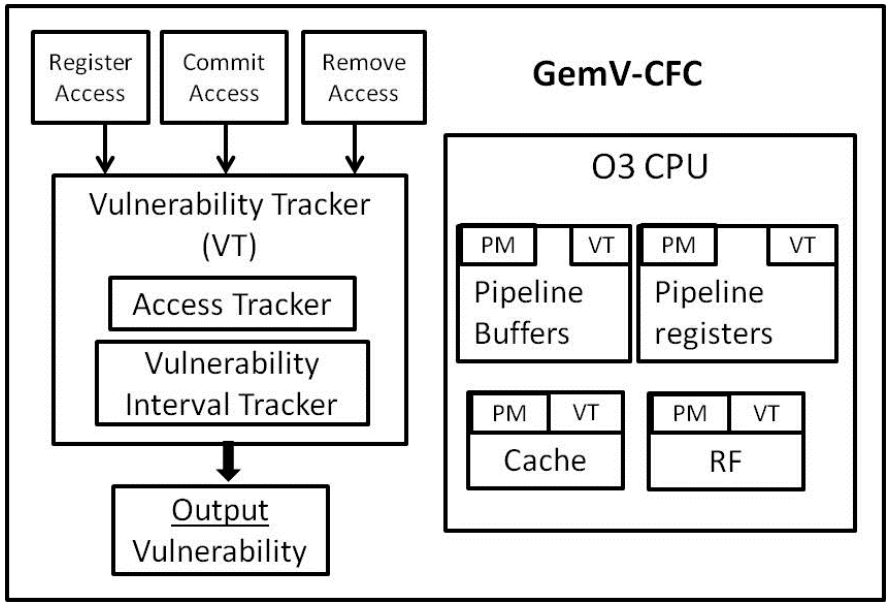


Figure 19: The GemV-CFC framework.

ducting the protected vulnerability from the total vulnerability of the component. The effective vulnerability of all components are accumulated to obtain the effective vulnerability of the system. All that is required in the simulation code is to attach an instance of VT to each processor component as shown in Figure 19, with the component specific customizations applicable, if any. The vulnerability tracker consists of two main components.

1. Access Tracker.

2. Vulnerability Interval Tracker.

The access tracker tracks the accesses to the attached processor component. For example, when the VT is attached to a register in the register file (RF), the access tracker in the VT tracks the reads and writes to the register. The vulnerability interval tracker tracks the vulnerability intervals or the durations when the processor components are exposed to control flow errors. For example, the vulnerable and non-vulnerable inter-

vals for the read-write access pattern to a register in the register file are demonstrated in Figure 20. The first write (leftmost) to read is vulnerable whereas the first read to write to is non-vulnerable as the register data gets overwritten. Accordingly, the second write to the last read (rightmost) is also vulnerable.

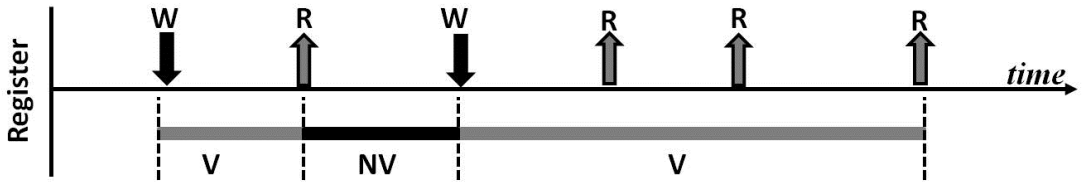


Figure 20: The vulnerability tracking for a register in the register file.

The vulnerability tracker has three interfaces to selectively track the relevant accesses. The register access interface registers the access time (tick in simulation jargon), the access mode (read or write), and the number of bits accessed for each access to the processor component, and associates these access details to an access entry with an access ID for tracking purposes. For out of order processors, the instructions in flight or dynamic instructions may be **committed** for the usual sequence of executed instructions or discarded (**squashed**), for example, in case of a branch misprediction. The commit access interface commits the component access entries that are part of the committed dynamic instructions whereas the remove access interface deletes the access entries that are part of the **squashed** dynamic instructions.

Chapter 8

EXPERIMENTS AND RESULTS

8.1 GemV-CFC Validation

Inorder to verify the correctness of GemV-CFC implementation, a correlation between the *vulnerability* metric used in GemV-CFC and *soft error rate* (**SER**) measured by fault injection campaigns is established. As defined in section(Section 3.3), the term *vulnerability* represents the time period between accesses that expose vulnerable data in the hardware component. The vulnerability of all bits in a hardware component are accumulated for all active `bit-cycles` and further the accumulation is extended to all active components in the processor. Fault injection campaigns inject the fault in specific bits at specific locations in hardware on predetermined clock cycles and count the number of simulations which generate erroneous output and calculate **SER** as the ratio of erroneous simulations to the total number of simulations. Comprehensive fault injection in a microarchitectural simulator, although prohibitively time consuming, provides coverage for almost all soft errors that can occur at the microarchitectural level. Vulnerabiliy overestimates the effect of a certain set of soft errors masked by a phenomenon called *software masking* or data-value masking by counting the corresponding bits as vulnerable, whereas fault injection may take into account the masking effect. For example, an AND instruction with an operand 0 will mask the possible errors in the other operand. Since the GemV-CFC implementation is based on the vulnerability metric, the input program for validation has to be devoid of data values that can cause software masking. Otherwise, the injected fault will be masked and will not manifest as an error in the program output, whereas the bit-cycle will still be counted as vulnerable by the vulnerability evaluation framework.

To provide a glimpse of the practicality of fault injection experiments, a MiBench benchmark takes 39 billion cycles on an average to execute on gem5 simulator, and it

takes 1121 seconds to simulate the benchmark on the host processor. The host processor considered here is a compute node on our 22-node linux cluster (21 Dual Quad-Core Intel Xeon E5620 2.4GHz processors with 24GB RAM, 1 NVIDIA Fermi M2050 GPU with 1.5GB RAM). In the case of a 32-bit register, the total number of fault injection simulations required are *Number of bits in the register* $\times$ *average execution cycles* $=$ *1.25 trillion*. So, the total host simulation time required for the fault injection campaign on the 32-bit register is 1400 trillion seconds and it takes 252 years to complete on the 22-node cluster.

A component specific validation infrastructure based on fault injection is designed and implemented in gem5 framework. Architecture components like PC and entries from the integer register file (RF) are selected inorder to provide a reasonable coverage of the processor. Since it is practically infeasible to conduct fault injection campaigns on larger benchmarks, a few small programs like matrix multiplication, dot product of vectors, vector addition, and matrix determinant are selected for ease of analysis and all input values are forced to be non-zero to avoid software masking. Initially, the input program is simulated without injecting any faults and the error-free output is saved for reference. The execution time of the error-free simulation is also noted. An error is injected in a selected bit position in one of the representative 32-bit temporary integer registers in ARM architecture on a selected simulation cycle during the execution of the input binary. If the simulation completes successfully, the resulting output is compared with the reference output. A mismatch is counted as a vulnerable bit-cycle. If the output is an exact copy of the reference output, the bit is deemed not vulnerable for the specific cycle. If the injected fault results in a segmentation fault, or similar architectural exceptions, the simulation run is tallied under the vulnerable bit-cycle bucket. An infinite loop is detected if the execution time exceeds the reference execution time plus an additional fixed buffer time, and the corresponding simulation

is enumerated as a vulnerable bit-cycle. Both vulnerable and non-vulnerable bit-cycles are counted towards the total simulation bit-cycles. The experiment is repeated for every bit in the register for every simulation cycle. The ratio of the vulnerable bit-cycles to the total simulation bit-cycles gives the component SER for the paritcular register in the register file. Similar experiments are carried out for PC, and four registers in the integer RF. Further, the component level soft error rates are compared with the total vulnerability values derived from the GemV-CFC framework. As shown in Table 6, the results show almost 100% correlation between the vulnerability values of architecture components from the GemV-CFC implementation and the soft error rates obtained from fault injection based validation experiments. The vulnerability calculation of most other processor components are modelled on similar lines and should show the same trend.

Benchmark	Correlation % (SER-GemV/SER-FI)	
	PC	Registers 0 to 3
Matrix multiplication	105.6	100.8
Dot product of vectors	100.7	100.2
Vector addition	104.4	99.6
Matrix Determinant	105.9	100

Table 6: GemV-CFC fault injection validation results.

As the current vulnerability estimation tools don't model the vulnerability of pipeline registers accurately, we obtain the number of vulnerable bits per instruciton per pipeline stage from an open source ARM AMBER processor RTL design [1]. These pipeline register vulnerability numbers are validated in RTL using fault injection.

8.2 Case Study: Application of GemV-CFC in analysis of CFC Techniques

To demonstrate the applications of GemV-CFC, experiments are performed to evaluate the protection achieved on embedded systems for four state-of-the-art control flow

Compilation Environment	
Compiler	LLVM (ARM v7-a)
Cross-compiler	CodeBench gcc (ARM v7-a)
Simulation Environment	
Mode	System Emulation mode
Architecture	ARM v7-a
Pipeline	5-stages (Out-Of-Order)
L1 D-Cache L1 I-Cache	64KB (2-way) 32KB (2-way)
D-TLB / I-TLB	64 entries

Table 7: Experimental setup for the case study demonstration of the GemV-CFC framework.

based techniques proposed recently, on 14 benchmarks from the MiBench suite [14]. The setup of the experiments is shown in Table 7. The L1 instruction cache and data cache are assumed to have parity protection and ECC protection, respectively. For the benchmark programs used in the experiments, the additional control flow checking instructions for the software and hybrid techniques are inserted during the late code generation phase of the LLVM compiler. In addition, the generated code is annotated (see Figure 18) with the information required by the GemV-CFC simulator for accurate vulnerability and protection analysis on the system. Similarly, hardware changes to the processor required by the hardware and hybrid techniques are implemented in the GemV-CFC simulator. In addition, the vulnerability tracker and protection models for the hardware changes are updated wherever required. The systematic methodology to derive the protection models are discussed in Chapter 6. This section briefly describes the mechanism of the implemented technique, and discuss the evaluation results derived from the GemV-CFC framework.

8.3 Control Flow Protection by Software Signatures (CFCSS)

Control Flow Checking by Software Signatures (CFCSS) [29] is a software only control flow protection (CFP) technique. Figure 21 describes the mechanism of the pro-

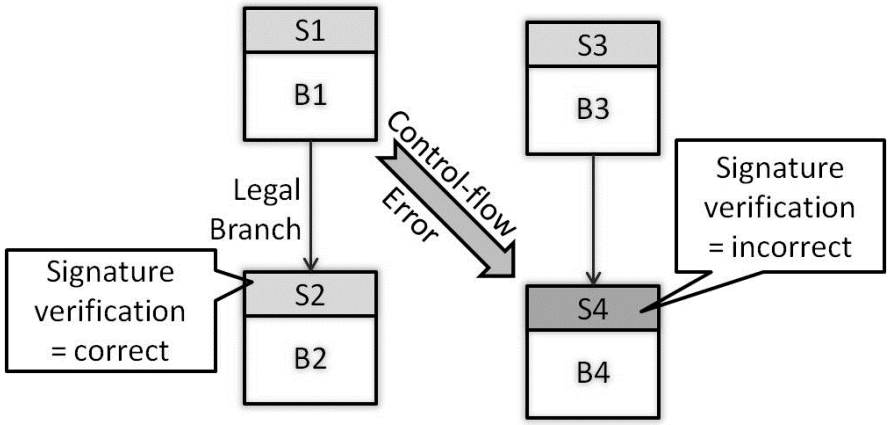


Figure 21: Working of control flow error detection through software signatures. The original program with four basic blocks (B1∼B4) is shown with the CFCSS implementation of *signature-checking code* (S1∼S4) to detect control flow errors during its execution.

posed method by means of a simplified example. For a given program when implemented with the CFCSS compiler technique, *signature-checking code* is inserted at the beginning of each of the basic-blocks in the program. The idea here is that, by means of software signatures, the basic blocks of the program are marked. During compilation, the legal successors to a basic block are identified, and the *signature-checking code* is installed to verify the transitions during execution. The checker code is composed of `xor` logic over the signature values. For example, in Figure 21, such a legal transition is verified by the S2 header in the basic block B2. In the case of a soft error that triggers a control flow error, any other basic block of the program could be executed. In this case, the *signature-checking code* of that basic block (say for example S4 of basic block B4 in Figure 21), check the signature values, and identifies that this block was executed from an illegal predecessor. Therefore, a control flow error is detected in the system.

The working of control flow error detection through software signatures hinges on the fact that each of the basic blocks in a program are assigned unique 32-bit signatures. In the case when a block has two predecessors, the CFCSS mechanism is

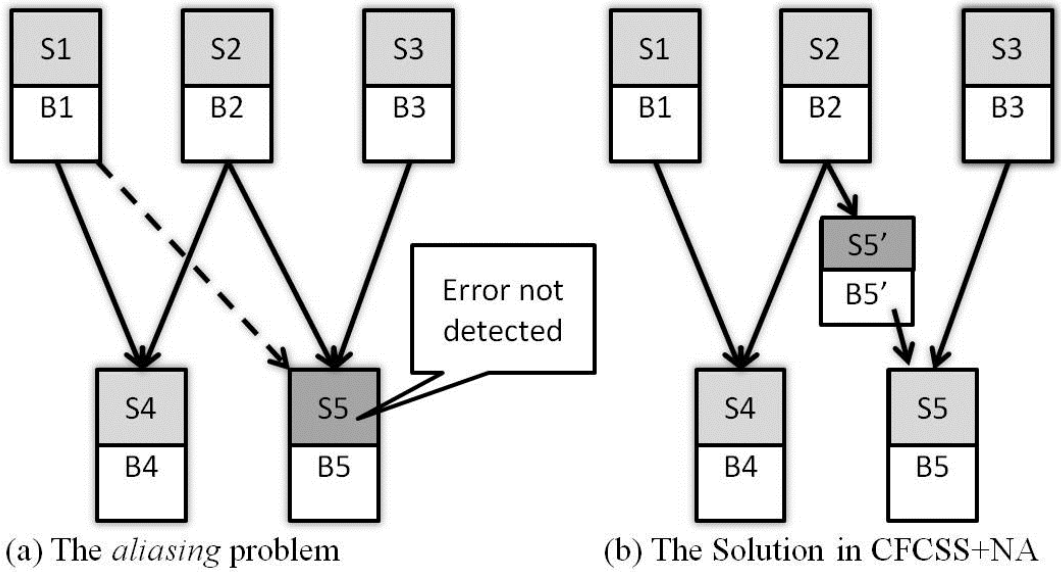


Figure 22: Demonstrating the aliasing problem in CFCSS implementation, with an example CFG.

such that either of the predecessors is chosen at random as the base, and the *signature-checking code* is designed to verify transitions from either of the legal predecessors using `xor` logic operations on the signatures. In the case when multiple blocks share multiple successor nodes with the successor nodes having multiple predecessor nodes, owing to the mechanism of signature assignment and verification in CFCSS, aliasing of the signatures renders certain control flow errors undetectable. Figure 22(a) describes the aliasing problem with an example. In the example B4 and B5 share B2 as a common predecessor, and therefore instead of a random selection of predecessors for signature verification, CFCSS restricts the base for runtime adjusting signature calculations to the common predecessor B2, and the signature-checking code is generated [29]. In the case of a control flow error that causes B5 to execute after B1, the aliasing problem renders such errors undetectable. Chao et al [5] in their work develop a method to overcome this aliasing problem, which is described in Figure 22. In this, between blocks that form a "W" structure (introducing the aliasing problem), one of the transition edges is interrupted by a dummy block with a unique signature of its own. This method ensures

that no two basic blocks in the program share a common predecessor, thus improving on soft error coverage.

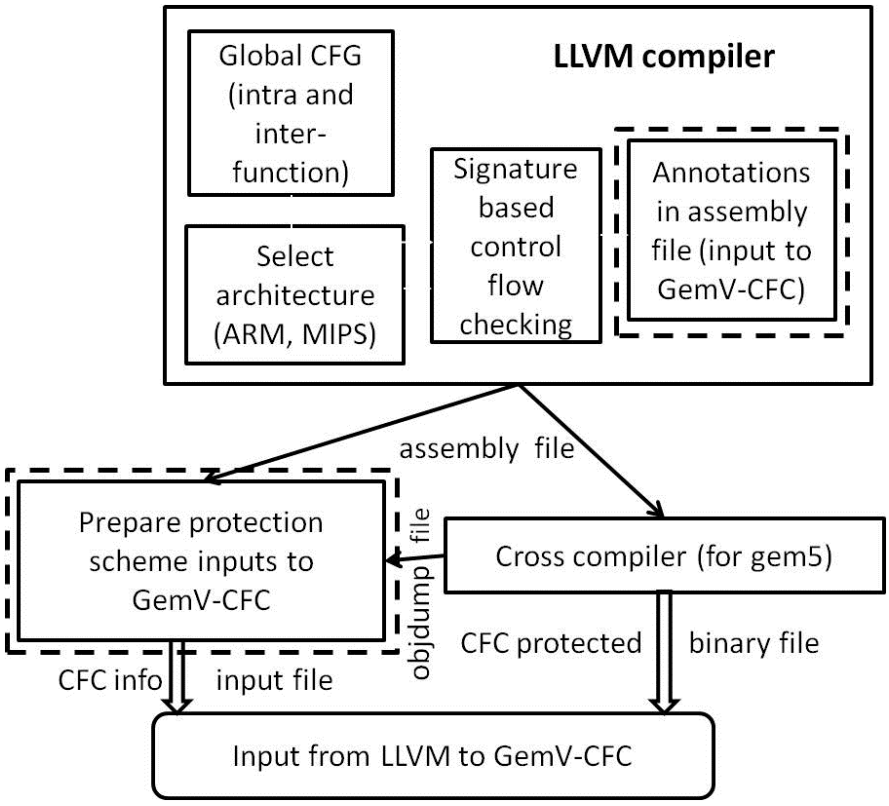


Figure 23: The implementation of CFCSS in LLVM.

The implementation of CFCSS in the popular LLVM compiler [19] consists of the generation of a global control flow graph, applying the CFCSS algorithm to the input program using the graph, and extracting the information on the technique to provide inputs to GemV-CFC, as shown in Figure 23. A global control flow graph (GCFG) incorporates the complete control flow transitions in the program. The nodes in the graph are basic blocks in the program, and the edges in the graph are the control flow transitions in the program, as defined in the previous section. The control flow transitions include branches, subroutine calls, and return statements. In effect, the GCFG incorporates the intra-function Control Flow Graph (CFG), which is readily available in LLVM, along with the inter-function CFG, which adds subroutine calls as

transition edges to the graph. The call and return statements are parsed to identify the predecessor, and successor nodes in the inter-function CFG. The GCFG is generated at the machine code level after the code generation step in the LLVM compiler backend. The GCFG generation and CFCSS algorithm blocks are implemented as passes in the LLVM compiler. The CFCSS implementation in the compiler provides the flexibility to choose two processor architectures, ARM and MIPS. The program is fed to the CFCSS pass which applies the CFCSS algorithm by adding the CFCSS header instructions to the basic blocks. The LLVM CFCSS instrumentation module tags the CFCSS header boundaries and the original basic block boundaries using comments in the assembly file generated during compilation.

The assembly file is cross compiled using the gcc cross compiler for ARM to generate the program binary. The object dump from the binary along with the assembly file are parsed to correlate the assembly comment tags for block boundaries to the instruction addresses in the object dump file. This correlation information serves as the link between LLVM and GemV-CFC.

8.3.1 Experimental Evaluation

Over the years, researchers have motivated for, and also demonstrated the effectiveness of the proposed control flow protection techniques, for commodity processors with the help of targeted fault injection experiments and relative comparisons. Through accurate and systematic modeling of system vulnerability and the protection achieved through the implementation of such protection techniques, GemV-CFC is able to analyze for the first time the effectiveness of the techniques in embedded systems. Figure 24 and Figure 25 present the results obtained from the experiments with the GemV-CFC framework upon implementation of the software based control flow protection techniques, CFCSS [29] and CFCSS+NA [5], respectively. The graphs show the effective system vulnerability (with protection) normalized over that of the original program.

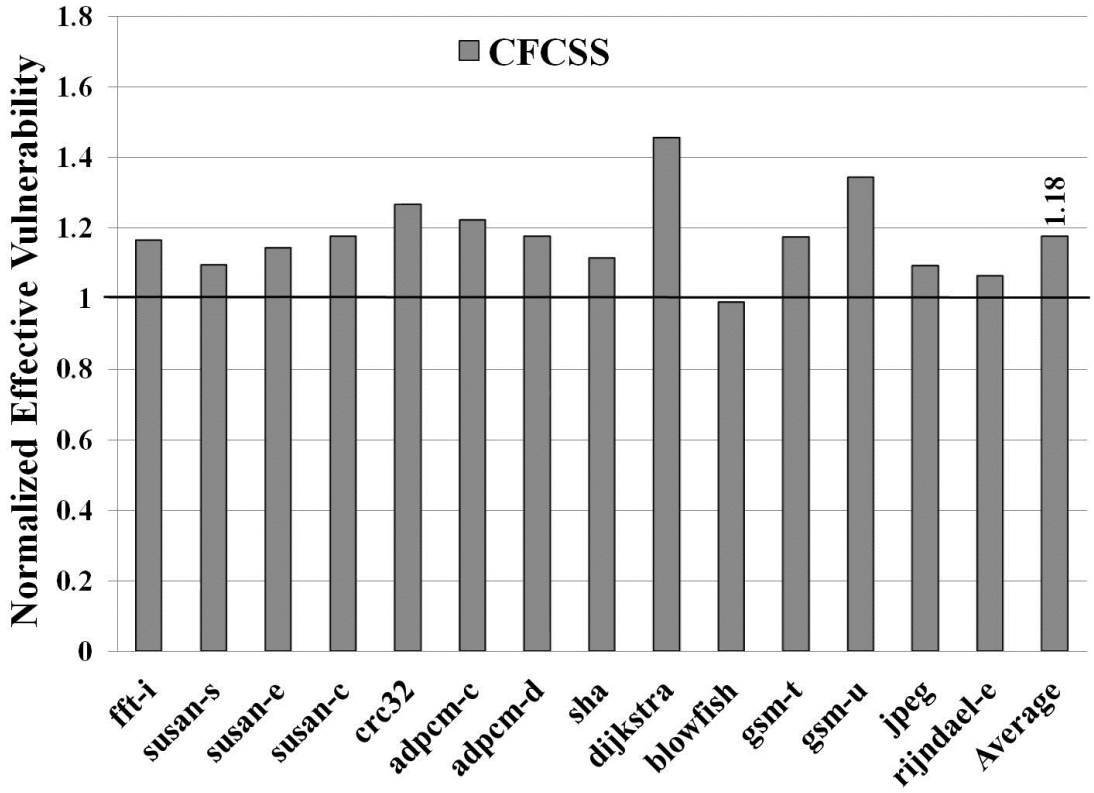


Figure 24: The effective system vulnerability (normalized over original program vulnerability) with CFCSS.

8.3.1.1 CFCSS Ineffective for Processor

1. When CFCSS is implemented on embedded processors, control flow protection only tends to increase system vulnerability after protection for all but one benchmark. In the case of an improved technique CFCSS+NA, we can observe that the vulnerability increases for all the benchmarks.

2. On an average, CFCSS and CFCSS+NA increase system vulnerability by 18%. The reason behind this behavior can be attributed to the fact that CFCSS incurs (on average) 22% performance penalty for software protection, by means of 48% increased number of instructions executed, thus increasing the residency of exe-

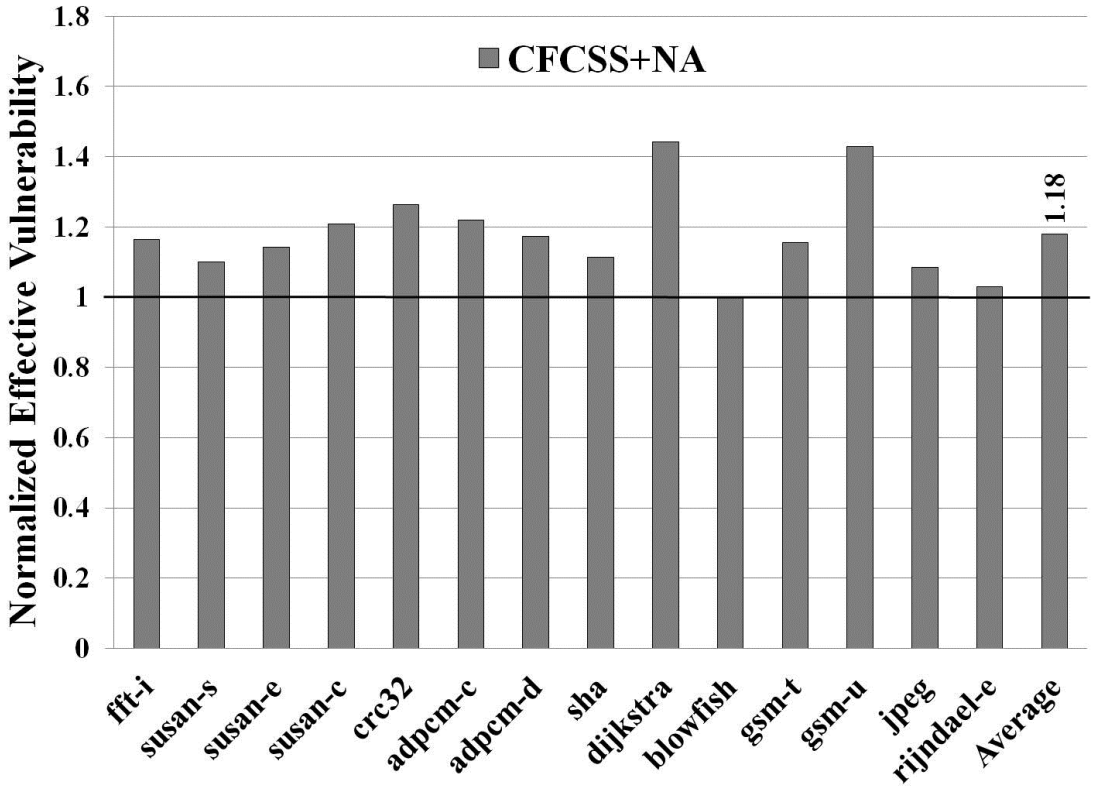


Figure 25: The effective system vulnerability (normalized over original program vulnerability) with CFCSS+NA (CFCSS with no aliasing).

cution data in vulnerable blocks. CFCSS+NA incurs a performance overhead of 21% and executes an additional 49% instructions to verify the control flow.

3. One startling fact to note here is that, the control flow protection technique implemented to protect the execution of `branch` instructions requires 84% additional `branch` executions.

We further looked into the contribution of the control flow checking code towards the effective system vulnerability and the execution time, as shown in Figure 26. On an average. the control flow checking code contributes 32.5% of the effective system vulnerability, and 19.5%*oftheexecutiontime*.

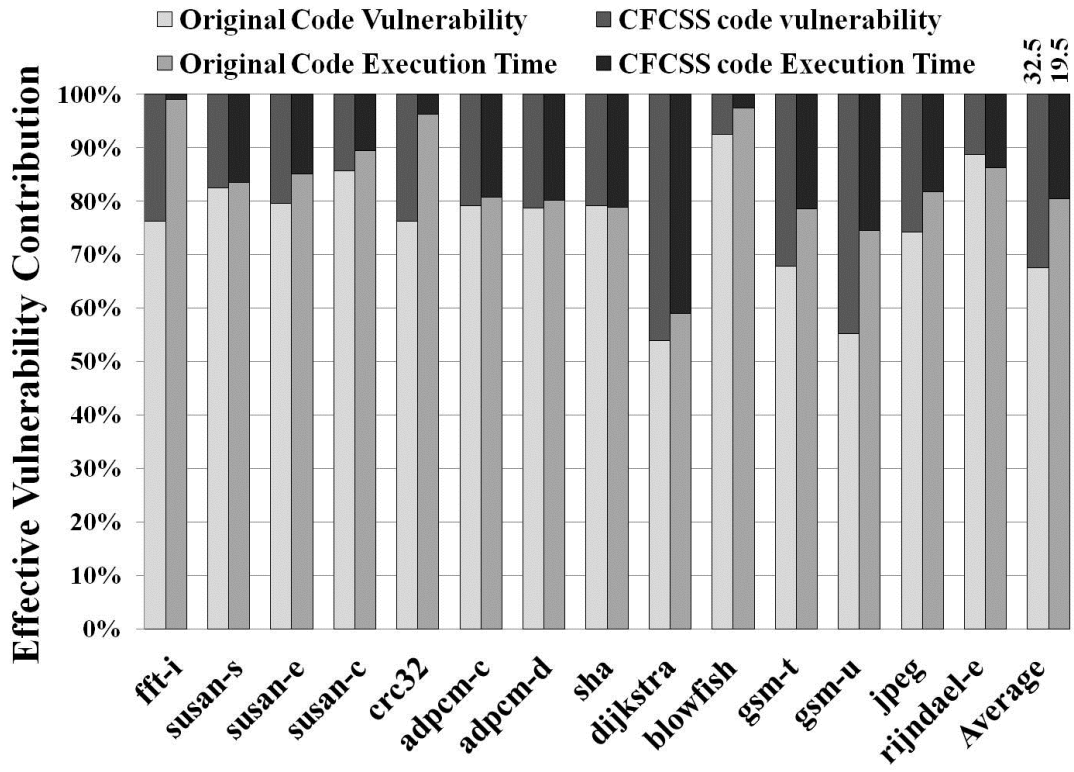


Figure 26: The percentage contribution of original code and the control flow checking code towards the effective system vulnerability and the execution time with CFCSS protection.

8.4 Control-Flow Error Detection Using Assertions (CEDA)

Control-Flow Error Detection Using Assertions (CEDA) [42] is the state-of-the-art technique in the field of software only control flow protection. Figure 27 illustrates the CEDA detection mechanism. The signature assignment, alongwith the control flow verification at the start and end of basic blocks, enables CEDA to detect most of the control flow errors. Through careful selection of software signatures, the techqniue detects the aliasing errors by maintaining unique signatures for even the aliased basic blocks. For example, aliasing from B1→B5 is detected by header S5, as shown in Figure 27(a). Furthermore, the jump check instructions in the signature checking code header detects the incorrect execution of a conditional branch. Figure 27(b) provides

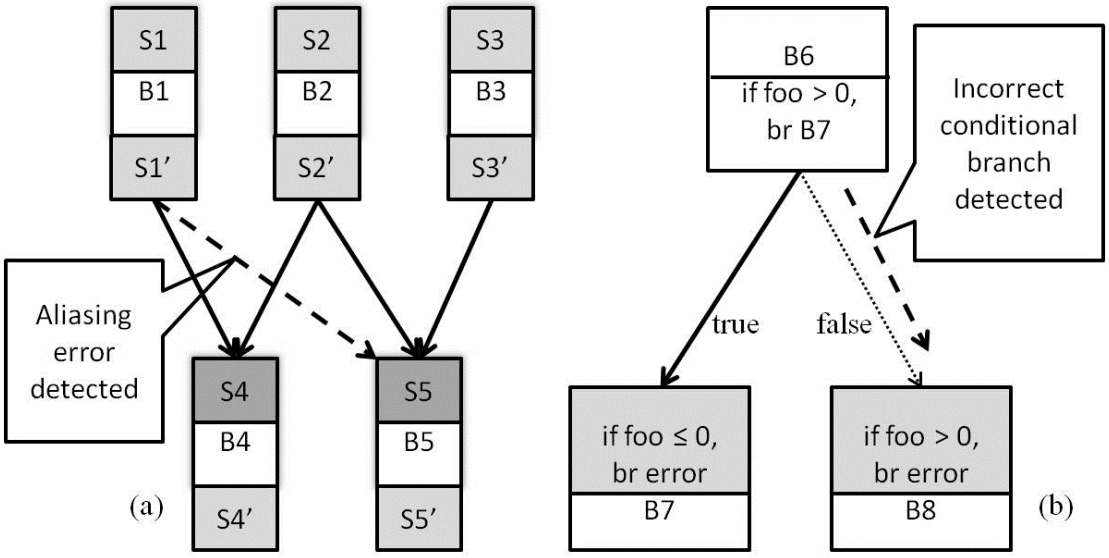


Figure 27: Working of control-flow error detection using assertions. The original program with five basic blocks (B1∼B5) is shown with the CEDA implementation of *signature-checking code headers and footers* (S1∼S5 and S1′ ∼S5′) to detect control flow errors during its execution. Detection of (i) aliasing errors and (ii) incorrect conditional executions are shown.

an example where an instance of the conditional branch in B6 is supposed to direct the branch execution to B7, as the condition is true for the particular instance. Instead, due to soft error, B6→B8 branch is taken. The jump check instructions in the header of B8, which further checks the condition, will detect such an incorrect branch. Since the condition in B6 is true for the instance, the variable $foo > 0$, and the jump check instruction in B8 which validates if $foo > 0$ will flag the error.

8.4.1 Experimental Evaluation

Figure 28 shows the results obtained from the experiments with the GemV-CFC framework upon implementation of CEDA.

8.4.1.1 CEDA Ineffective for Processor

CEDA is one of the state-of-the-art techniques in the software based control flow protection domain and provides mechanisms to detect almost all categories of control flow

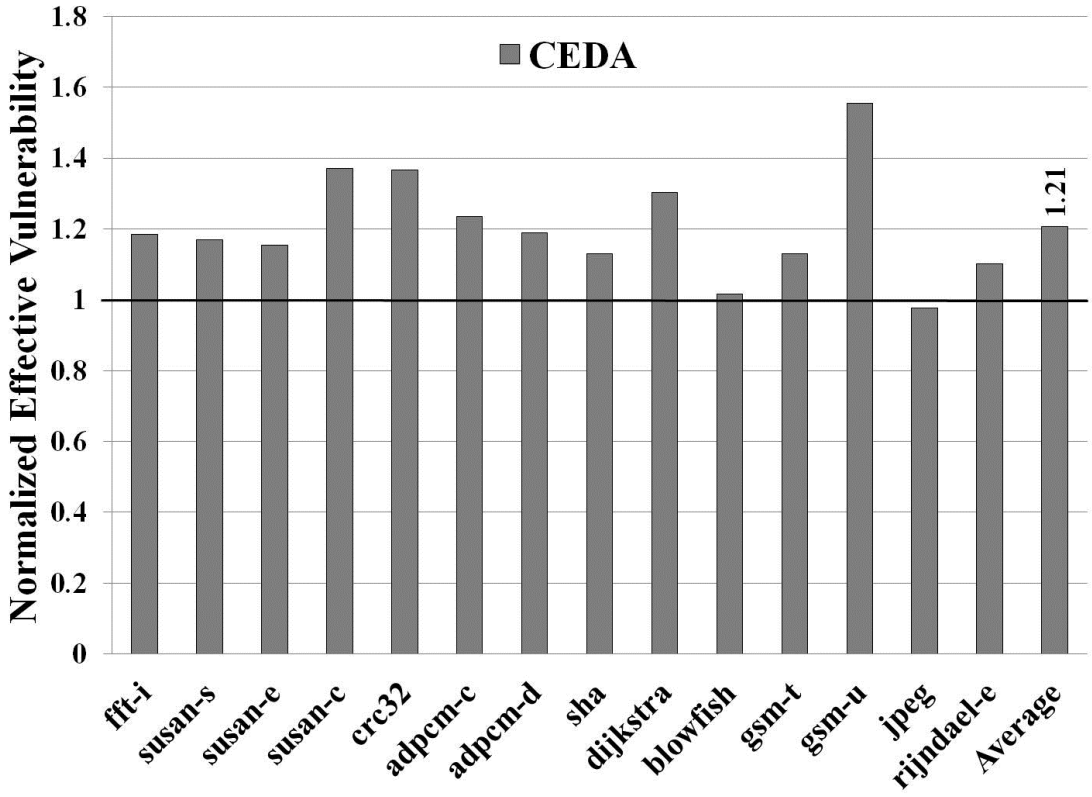


Figure 28: The effective system vulnerability (normalized over original program vulnerability) with CEDA.

errors. But GemV-CFC contradicts this claim by showing that CEDA increases vulnerability of the system considerably and the situation is worse when compared to CFCSS or CFCSS+NA.

1. With the implementation of CEDA on commodity processors, system vulnerability increases after protection for most benchmarks (13/14). In fact for *gsm-u* benchmark, CEDA increases vulnerability by 55%.

2. On an average, CEDA increases system vulnerability by 21%. The behavior can be attributed to the fact that CEDA incurs (on average) 38% performance penalty for software protection, by means of 77% of extra instructions executed.

3. Here again, the control flow protection technique implemented to protect the execution of `branch` instructions requires 236% additional `branch` executions.

8.5 Control Flow Error Detection and Correction (CFEDC)

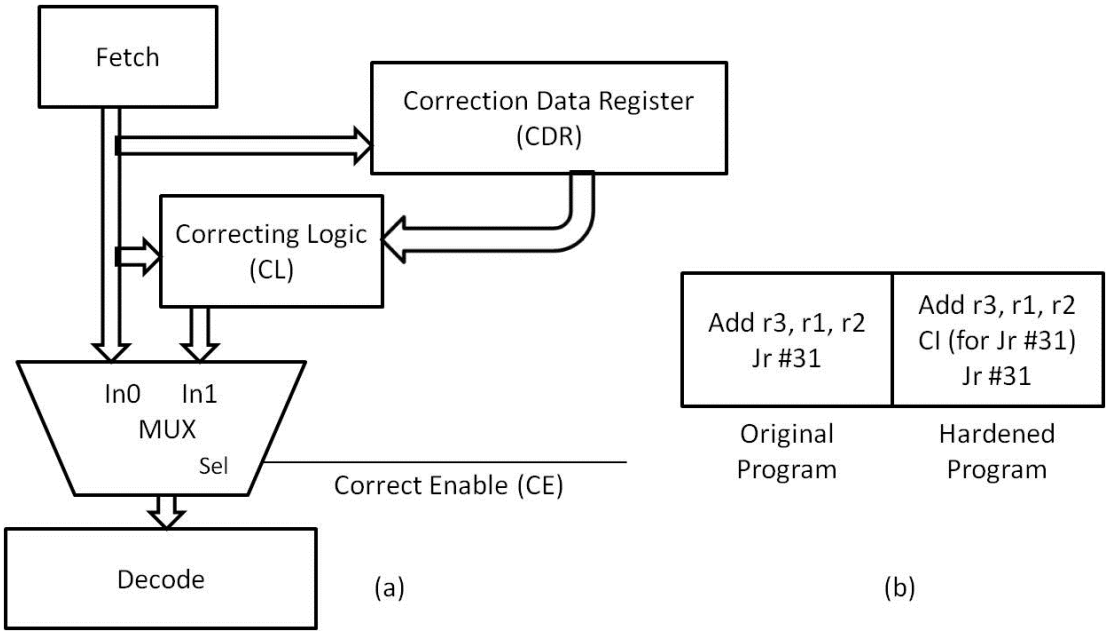


Figure 29: Working of CFEDC.

Farazmand et al [10] in their work propose a simple hybrid technique to detect and correct soft errors that occur in the processor pipeline affecting their correct execution. This process involves two stages: (1) **Program Hardening**– During compilation, *correction data* (CD), which is a correction code for the control instruction CI), is inserted before each control instruction in the program, as shown in Figure 29(b). The CI is a hamming code of the subsequent branch instruction. The CD is set to be introduced into the pipeline before the decoding of the control instruction, as shown in Figure 29(a). (2) **Correcting Hardware**– The CD from the program is processed by specialized hardware within the pipeline to detect any errors during the `fetch` stage of the control instruction. In the case of a detected error, the *Correcting Logic* (CL) together with the *Correction Data Register* (CDR) is used to introduce the correct control instruction into the pipeline register of the `decode` stage.

8.5.1 Experimental Evaluation

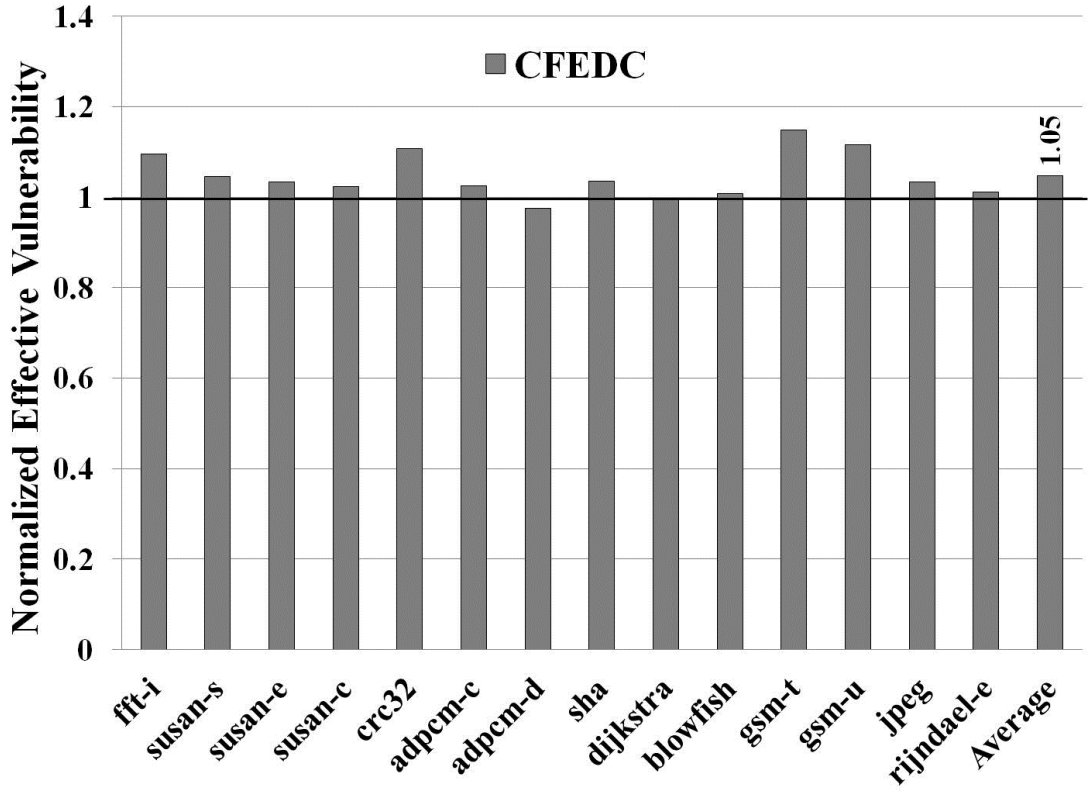


Figure 30: The effective system vulnerability (normalized over original program vulnerability) with CFEDC.

Figure 30 shows the results obtained from the experiments using the GemV-CFC framework upon implementation of CFEDC. On an average, CFEDC increases the vulnerability of the system by 5%, due to the additional 5% control instructions required to detect and correct the errors in the fetched branch instructions, before entering the `decode` stage.

8.6 Control Flow Checking by Execution Tracing (CFCET)

Rajabzadeh et al [32] propose a method to verify the control flow execution of the system through the use of an external watchdog processor together with the internal execution tracing feature (Branch Trace Messaging or BTM [16]) available in commer-

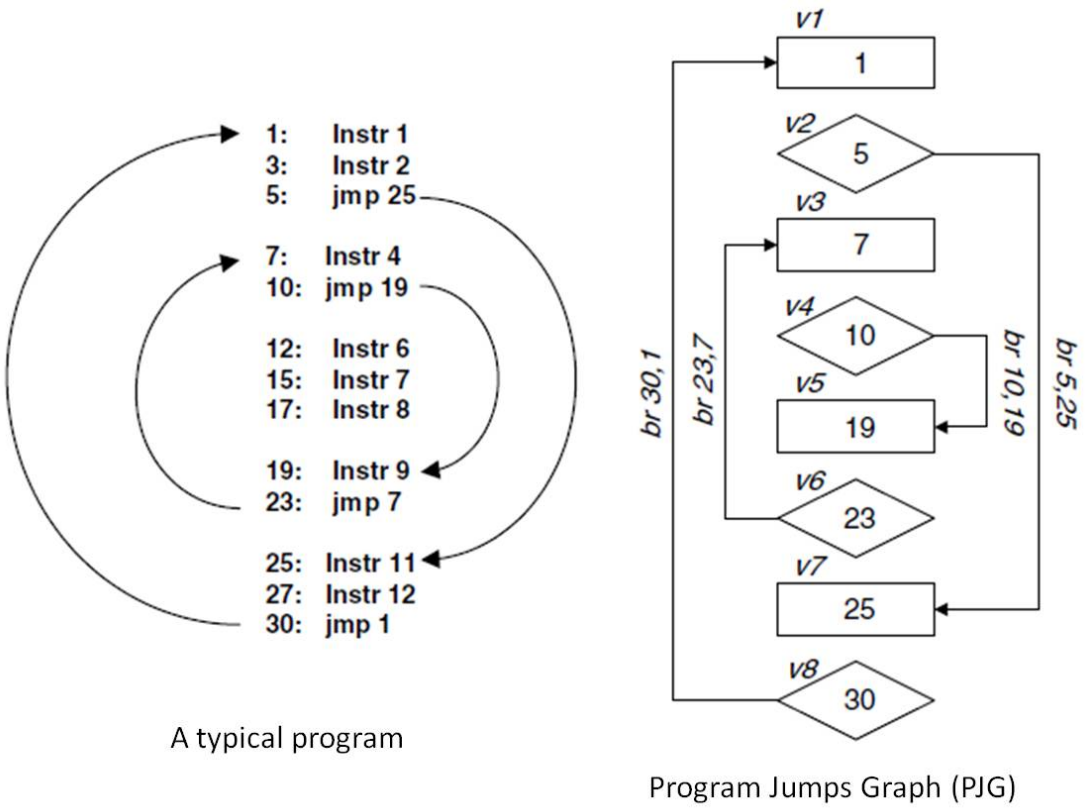


Figure 31: A typical program with its Program Jumps Graph (PJG) is shown. PJG is used as a reference graph by the watchdog processor in CFCET.

cial off-the-shelf (COTS) processors (Intel Pentium Family [16]). In this, the technique traces the program jumps graph (PJG) at run-time and compares with the reference jumps graph to detect possible violation caused by transient faults. PJG represents the jump instructions in the program as shown in Figure 31. This graph information is loaded in the associative memory of the watchdog processor for reference during execution. An insertion of a branch or a branch target modification causes a mismatch when compared to this reference PJG, as shown in Figure 32, which leads to the detection of the control flow error. Note that the branch deletion cannot be detected using CFCET since the branch deletion causes a non-branch instruction to execute and hence, BTM cycles wont be generated as BTM cycles are generated only for branch instructions.

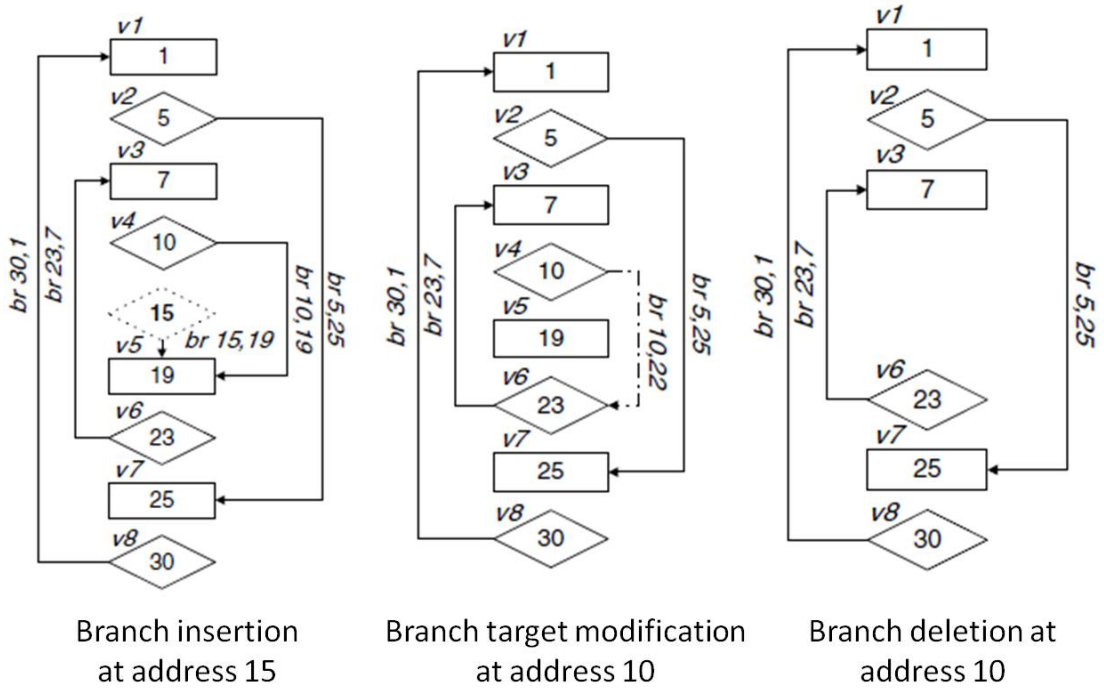


Figure 32: Working of CFCET. Different CFEs and their effects on PJG are shown here, for example, (a) branch insertion at address 15, (b) branch target modification at address 10, and (c) branch deletion at address 10.

8.6.1 Experimental Evaluation

Figure 33 shows the results obtained from the GemV-CFC simulations upon implementation of CFCET. The system vulnerability remains almost the same even after applying the protection through the execution tracing feature. Although the number of instructions is unchanged, the extra BTM cycles contribute to the increase in effective vulnerability, for some of the benchmarks.

8.7 Vulnerability Per Cycle and Vulnerability Per Instruction

Figure 34 presents the effective system vulnerability after protection per cycle and per instruction normalized over original program vulnerability per cycle and per instruction respectively, upon implementation of different CFC techniques, averaged over all the MiBench benchmarks. On an average, there is not much change in vulnerability

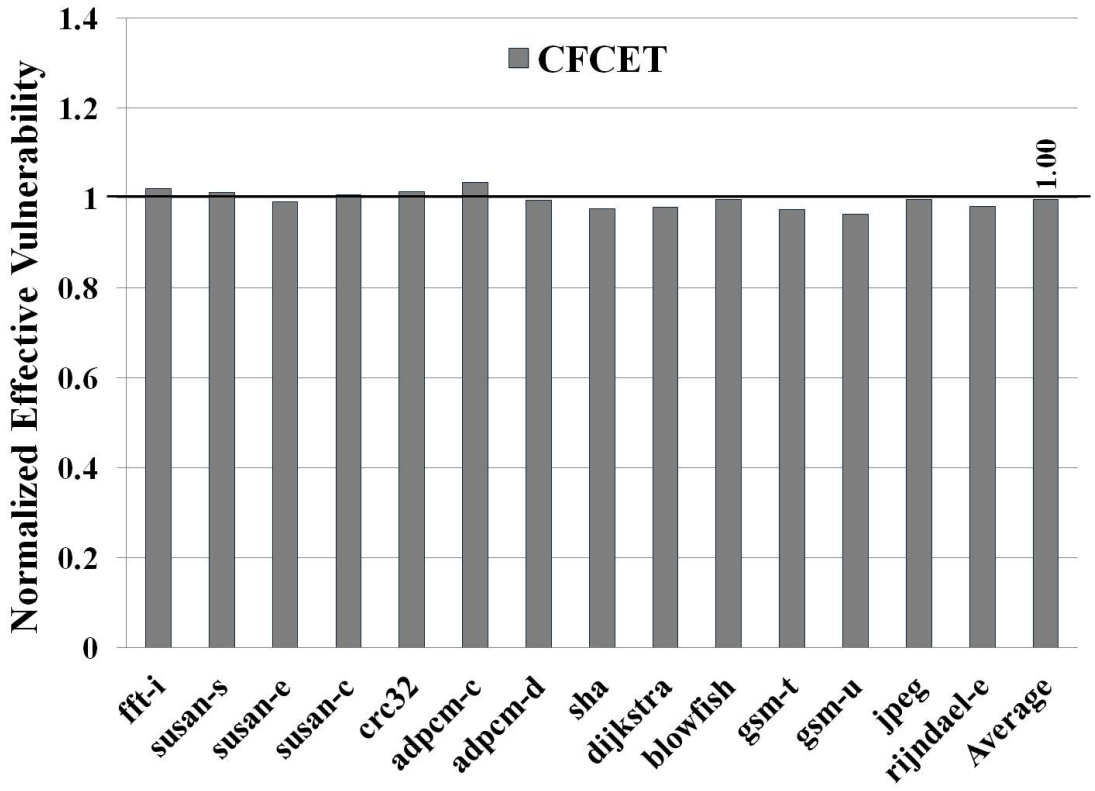


Figure 33: The effective system vulnerability (normalized over original program vulnerability) with CFCET.

per cycle compared to the original program for any of the techniques. The experiment shows that the increase in system vulnerability is proportionate to the increase in execution cycles for all the CFC techniques, due to the corresponding increased residency of vulnerable data in processor components.

The software based CFC schemes bring down the vulnerability per instruction by 16 - 20%, and the hybrid CFP achieves a reduction of 5%, whereas the hardware based CFP increases the vulnerability per instruction by 9%. The system vulnerability of the hardware based CFCET technique increases due to the additional BTM cycles required for the execution tracing, although the number of instructions remain the same since the tehcnique is hardware based. The reduction in system vulnerability per instruction for the software based techniques is due to the fact that the increase in system

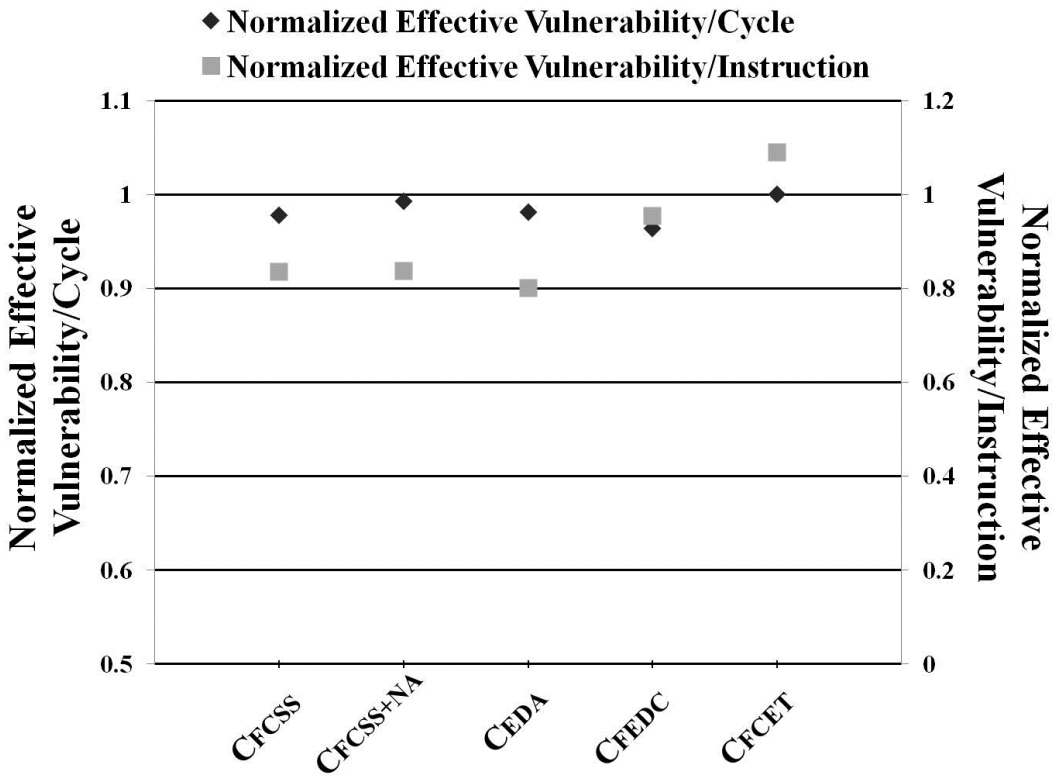


Figure 34: The normalized effective system vulnerability per cycle and per instruction upon implementation of (a) CFCSS, (b) CFCSS+NA, (c) CEDA, (d) CFEDC, and (e) CFCET.

vulnerability is offseted by the comparatively higher increase in number of instructions executed.

8.8 Error Coverage

In the analysis of a soft error protection technique, error coverage is a key metric used in the justification of its applicability and efficacy. In the case of control flow based soft error protection techniques, we observe that there does not exist any quantitative mechanism to measure the error coverage of the proposed technique. Through implementation of vulnerability measurement of all processor components, protection model interpretation, and cycle-accurate simulations in our GemV-CFC framework, we are able to readily analyze the error coverage with quantitative numbers. Figure 35 shows

the distribution of vulnerability among the processor components for the system core. In the case of control flow protection techniques, the cache and its components are always considered protected and beyond the scope of protection. Therefore, for our coverage analysis, we present the distribution of vulnerability in the processor core alone. In Figure 35 we see that the pipeline registers (89%) constitute the major vulnerability proportion followed by the register file (6.62%), rename table (3.16%) and then the PC (0.54%).

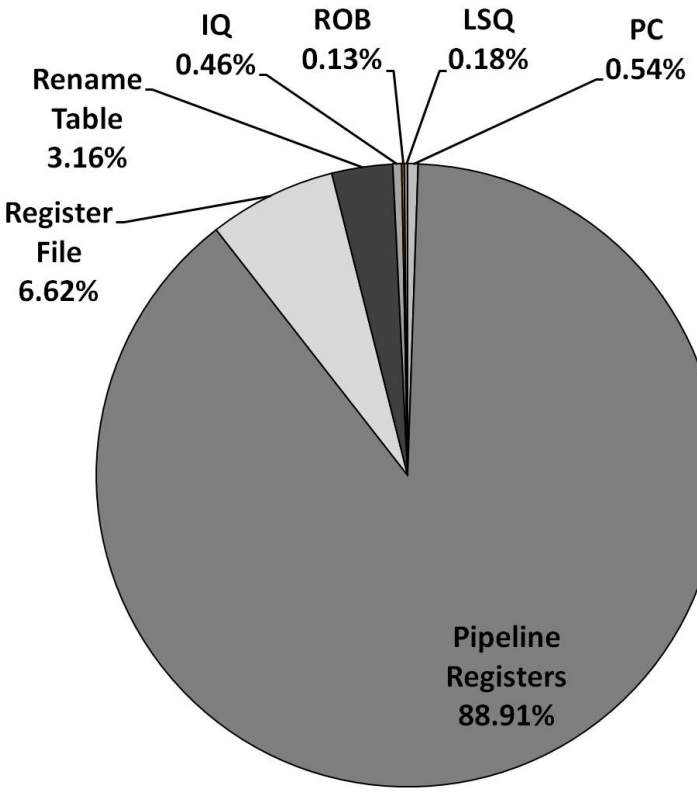


Figure 35: Vulnerability distribution in the processor components.

As discussed earlier, we observe that control flow based protection techniques can only protect (or in other words detect errors in) only the PC, the pipeline registers, the link register and the processor status register. From the vulnerability distribution graph (Figure 35), we see that the importance of control flow protection techniques to protect the major vulnerable components in the processor core is valid. However,

	Protection Achieved			
Protection Technique	**PC**	**Pipeline Register**	**Link Register**	**Processor Status Register**
CFCSS	91.03%	0.72%	100%	0%
CFCSS+NA	91.03%	0.73%	100%	0%
CEDA	93.19%	1.07%	100%	100%
CFEDC	12.8%	0.21%	0%	0%
CFCET	N/A	0.4%	0%	0%

Table 8: Protection achieved in processor components in the GemV-CFC framework for the Case Study experiments.

careful quantitative analysis of the said protection techniques reveals a startling revelation. In Table 8 we see that the proportion of protection applied over these vulnerable components is negligible and contradictory to the requirements of the system. The protection provided to pipeline registers, across the spectrum of CFC techniques, is minimal. Only software based CFC schemes provide reasonable protection to PC. Note that the software based CFC techniques are given the benefit of the doubt in case of 100% protection for link register and CEDA's 100% protection of processor status register. Considering the huge proportion of total vulnerability from the pipeline registers, we see that the effective protection applied here is negligible.

Quantitative analysis of error coverage in the control flow based protection techniques proposed reveals that only a meagre fraction of the total system vulnerability is actually protected by the said techniques, which is in short in-effective in achieving any level of system level protection from soft errors.

Chapter 9

DISCUSSION

9.1 Why are Control Flow Checking Mechanisms Ineffective?

We have seen that 33 to 77% of faults can eventually cause a control flow error in the system. We have identified two cases where faults can trigger control flow errors:

1. Direct sources - Faults in certain processor elements can directly cause a control flow error. For example, a bit flip in the PC register or in the branch offset field in the pipeline registers can cause different NPC value to be loaded into the PC in the next cycle or a subsequent cycle, which can cause a control flow error.

2. Indirect sources - These are faults that occur in some processor elements and may pass the erroneous value through different components or registers before eventually turning up as a control flow error. For example, a fault in a register in the integer RF may cause the value to be used for a comparison, before the conditional branch, in the third read of the register after the fault and the comparison might evaluate to false instead of the expected value of true (or vice versa). This erroneous comparison result will trigger a control flow error. Here, the fault was stored for some time and passed through couple of processor components before showing up as control flow error and therefore, such processor components are considered as indirect sources of control flow errors.

Control flow errors due to indirect sources are more common compared to direct sources. None of the CFC techniques offer protection from control flow errors due to indirect sources. In order to detect such errors, we believe the redundancy based soft error detection techniques need to be employed as they verify the computation results at regular intervals by comparing with redundant duplicate values. The exisitng control flow checking techniques concentrate mostly on detecting errors due to direct sources and as

we have seen, they increase the vulnerability of the system and come with performance overheads. To provide a glimpse of the percentage of control flow errors protected by CFC techniques, let us consider the case of CFCSS. Let us consider a RISC system, for which 33% of errors can eventually cause control flow errors. On an average, the total vulnerability of CFCSS protected applications was found to be 1.55×10^{14} bit-cycles. In this case, 33% or 5.1×10^{13} bit-cycles can cause control flow errors. Out of the total vulnerable cycles, only 2.3×10^{12} bit-cycles are protected by CFCSS, which constitutes a meagre 1.34%. Therefore, only 4.04% of possible control flow errors are protected by CFCSS. This shows the ineffectiveness of CFCSS and attributes it to the fact that it doesn't provide error coverage for the indirect sources of control flow errors.

9.2 Alternative Protection Methods

As we can see from Figure 35, the major contributors to the vulnerability of the processor core are pipeline registers and register file. There are several existing hardware based mechanisms that provide better error coverage for these processor components at the cost of comparable area and power overheads. For example, the C-element latch scheme [12] provides 99.88% protection for pipeline latches, with an area overhead of 6.4 to 15%, by duplicating the pipeline latches. The Shield technique [24] can selectively protect most vulnerable registers using ECC. Shield reduces the AVF of integer register file by up to 84% and the floating point register file by up to 100%, with a minimal area overhead of 10% and a power overhead of 45%. The shielding technique doesn't incur any performance penalty. Although these techniques come with some area and power overheads, they can together provide better protection to the system than the control flow checking techniques.

Chapter 10

CONCLUSIONS

Over the years, researchers have motivated for, and also demonstrated the effectiveness of the proposed control flow protection techniques, for commodity processors with the help of targeted fault injection experiments and relative performance comparisons. In our work, we analyze system reliability by means of an accurate, and efficient metric - *vulnerability*. We model vulnerability at the system level in a cycle-accurate simulation environment (gem5), and also integrate methodologies to quantitatively evaluate the vulnerability of all the components in a processor in the presence of control flow based protection techniques in our framework, GemV-CFC. For the first time, we derive and implement a systematic approach to model the protection offered by CFC techniques. Our evaluation experiments in the form of case studies over state-of-the-art CFC techniques at all levels of computation (hardware, software and hybrid layers) indicate that control flow based protection techniques are highly ineffective for modern embedded systems. Although these CFC techniques claim high error coverage and protection, they fail to provide quantitative evidence to support their claims. Our quantitative analysis reveals the loop-holes in such implementations and indicates that the proposed control flow based protection techniques (at the software, hybrid and hardware layers) are insufficient to protect a system from soft errors. On an average, software based control flow protection techniques increase the vulnerability of the system by 18 to 21% and comes with a performance overhead of 17 to 38%. The hybrid techniques increase the system vulnerability by 5% while incurring a performance penalty of 3.5%. For the hardware techniques, the system vulnerability remains almost the same, but it comes with a performance overhead of 9%, notwithstanding the vulnerability of additional hardware added for their implementations. Our component-wise vulnerability analysis shows that pipeline registers and register file are the major contributors to the over-

all system vulnerability, and hence alternative techniques like C-element latch scheme and Shield together can provide better protection to the processor, with nominal area overheads and reasonable power overheads. The main drawback of the control flow checking techniques is that they don't detect control flow errors caused due to indirect sources like register file, and provide protection only to errors in direct sources like pipeline registers, and are ineffective even in providing reasonable protection to these direct sources.

REFERENCES

[1] Amber ARM-compatible core :: Overview, http://opencores.org/project,amber, 2010.

[2] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):627–641, June 1999.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, Feb. 2002.

[4] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 532 – 543, june 2005.

[5] W. Chao, F. Zhongchuan, C. Hongsong, B. Wei, L. Bin, C. Lin, Z. Zexu, W. Yuying, and C. Gang. CFCSS without Aliasing for SPARC Architecture. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2094 –2100, 29 2010-july 1 2010.

[6] Y. Cheng, A.-g. Ma, Y.-w. Wang, Y.-x. Tang, and M.-x. Zhang. SS-SERA: An improved framework for architectural level soft error reliability analysis. *Journal of Central South University*, 19:3129–3146, 2012.

[7] R. Desikan, D. Burger, S. W. Keckler, and T. M. Austin. Sim-alpha: a Validated Execution Driven Alpha 21264 Simulator. Technical Report TR-01-23, Department of Computer Sciences, University of Texas at Austin, 2001.

[8] J. Eifert and J. Shen. Processor Monitoring Using Asynchronous Signatured Instruction Streams. In *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, page 106, jun 1995.

[9] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: a performance model framework. *Computer*, 35(2):68 –76, feb 2002.

[10] N. Farazmand, M. Fazeli, and S. Miremadi. FEDC: Control Flow Error Detection and Correction for Embedded Systems without Program Interruption. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 33 –38, march 2008.

[11] X. Fu, T. Li, and J. A. B. Fortes. Sim-SODA: A Unified Framework for Architectural Level Software Reliability Analysis. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation (Held in conjunction with International Symposium on Computer Architecture)*, 2006.

[12] K. Gardiner, A. Yakovlev, and A. Bystrov. A C-element Latch Scheme with Increased Transient Fault Tolerance for Asynchronous Circuits. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, pages 223 –230, july 2007.

[13] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 581 – 588, nov. 2003.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[15] P. Hazucha, T. Karnik, S. Walstra, B. Bloechel, J. Tschanz, J. Maiz, K. Soumyanath, G. Dermer, S. Narendra, V. De, and S. Borkar. Measurements and analysis of SER tolerant latch in a 90 nm dual-Vt CMOS process. In *Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003*, pages 617 – 620, sept. 2003.

[16] Intel Corporation. *Pentium Processor Family Developer's Manual*. Intel Corporation, 1997.

[17] R. Jeyapaul, F. Hong, A. Rhisheekesan, A. Shrivastava, and K. Lee. UnSync: A Soft Error Resilient Redundant Multicore Architecture. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 632 –641, sept. 2011.

[18] S. Kayali. Reliability Considerations for Advanced Microelectronics. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, PRDC '00, page 99, Washington, DC, USA, 2000. IEEE Computer Society.

[19] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[20] H. Madeira and J. Silva. On-line signature learning and checking: experimental evaluation. In *CompEuro '91. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference. Proceedings.*, pages 642 –646, may 1991.

[21] T. May. Soft Errors in VLSI: Present and Future. *Components, Hybrids, and Manufacturing Technology, IEEE Transactions on*, 2(4):377 – 387, dec 1979.

[22] T. Michel, R. Leveugle, and G. Saucier. A new approach to control flow checking without program modification. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 334 –341, jun 1991.

[23] G. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson. Use of time, location and instruction signatures for control flow checking. In *Proc. of the DCCA-5 Intl Conf*, 1998.

[24] L. W. Montesinos, Pablo and J. Torrellas. Shield: Cost-effective soft-error protection for register files. *Third IBM TJ Watson Conference on Interaction between Architecture, Circuits and Compilers (PAC2)*, 2006.

[25] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. *Microarchitecture, IEEE/ACM International Symposium on*, 0:29, 2003.

[26] J. B. Nickel and A. K. Somani. Reese: A method of soft error detection in microprocessors. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pages 401–410, Washington, DC, USA, 2001. IEEE Computer Society.

[27] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pages 86 –94, 1999.

[28] B. Nicolescu, Y. Savaria, and R. Velazco. Software detection mechanisms providing full coverage against single bit-flip faults. *Nuclear Science, IEEE Transactions on*, 51(6):3510 – 3518, dec. 2004.

[29] N. Oh, P. Shirvani, and E. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, 51(1):111 –122, mar 2002.

[30] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63 –75, mar 2002.

[31] J. Ohlsson, M. Rimen, and U. Gunneflo. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 316 –325, jul 1992.

[32] A. Rajabzadeh and S. Miremadi. CFCET: A hardware-based control flow checking technique in COTS processors using execution tracing. *Microelectronics Reliability*, 46(5):959–972, 2006.

[33] A. Rajabzadeh and S. G. Miremadi. A Hardware Approach to Concurrent Error Detection Capability Enhancement in COTS Processors. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, PRDC '05, pages 83–90, Washington, DC, USA, 2005. IEEE Computer Society.

[34] A. Rajabzadeh and M. Vosoughifar. A Dependable Microcontroller-based Embedded system. In *The Fourth International Conference on Dependability (DEPEND 2011)*, pages 24–29, aug 2011.

[35] N. R. Saxena and W. K. McCluskey. Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums. *IEEE Trans. Comput.*, 39(4):554–559, Apr. 1990.

[36] M. A. Schuette and J. P. Shen. On-line monitoring using signatured instruction streams. In *Proc. 13th Int. Test Conf.*, pages 275–282, oct 1983.

[37] M. A. Schuette and J. P. Shen. Processor Control Flow Monitoring Using Signatured Instruction Streams. *IEEE Trans. Comput.*, 36(3):264–276, Mar. 1987.

[38] A. Shrivastava, J. Lee, and R. Jeyapaul. Cache vulnerability equations for protecting data in embedded processor caches from soft errors. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 143–152, New York, NY, USA, 2010. ACM.

[39] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. In *Proceedings of the 39th Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO 39, pages 223–234, Washington, DC, USA, 2006. IEEE Computer Society.

[40] D. D. Thaker, F. Impens, I. L. Chuang, R. Amirtharajah, and F. T. Chong. On using recursive tmr as a soft error mitigation technique, 2008.

[41] R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier. Multicore soft error rate stabilization using adaptive dual modular redundancy. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 27–32, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[42] R. Vemu and J. Abraham. CEDA: Control-Flow Error Detection Using Assertions. *IEEE Trans. Comput.*, 60(9):1233–1245, Sept. 2011.

[43] R. Vemu, S. Gurumurthy, and J. Abraham. ACCE: Automatic correction of control-flow errors. In *Test Conference, 2007. ITC 2007. IEEE International*, pages 1 –10, oct. 2007.

[44] R. Venkatasubramanian, J. Hayes, and B. Murray. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pages 137 – 143, july 2003.

[45] K. Wilken and J. P. Shen. Continuous signature monitoring: efficient concurrent-detection of processor control errors. In *Proceedings of the 1988 international conference on Test: new frontiers in testing*, ITC'88, pages 914–925, Washington, DC, USA, 1988. IEEE Computer Society.