March 2019

An Efficient Run-time CFI Check for Embedded Processors to Detect and Prevent Control Flow Based Attacks

Srivarsha Polnati
University of South Florida, spolnati@mail.usf.edu

An Efficient Run-time CFI Check for Embedded Processors to Detect and Prevent Control Flow Based Attacks

by

Srivarsha Polnati

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Srinivas Katkoori, Ph.D.
Jay Ligatti, Ph.D.
Hao Zheng, Ph.D.

Date of Approval:
January 31, 2019

Keywords: Control Flow Integrity (CFI), Basic Blocks, Hamming Distance, SimpleScalar

DEDICATION

I dedicate this work to my family and all my beloved ones who helped and supported me.

ACKNOWLEDGMENTS

I would first like to formally acknowledge Dr. Srinivas Katkoori for providing the opportunity to work on this project. I always cherish his support and express my gratitude that he has helped me identify my strengths and use them in a best possible way. I am forever grateful to him for bringing me success with his guidance and constant support. I give thanks to Dr. Jay Ligatti and Dr. Hao Zheng for volunteering their precious time to serve as members on my thesis committee. I thank God for shielding me in every phase of my life and making me pursue a fruitful path. My special thanks to Sridhar Polnati and Uma Polnati (parents), Govind Vardhan Polnati (brother), Raghunath Bahunuthula (friend) and all my beloved ones for encouraging me in achieving my goals. I would also like to thank my friend Love Kumar Sah and all the colleagues I have acquainted for their help and assistance.

TABLE OF CONTENTS

LIST OF TABLES iii

LIST OF FIGURES iv

ABSTRACT vi

CHAPTER 1: INTRODUCTION 1
- 1.1 Importance of the Problem and Different Approaches 1
- 1.2 Proposed Approach Overview 2
- 1.3 Experimental Results Overview 3
- 1.4 Thesis Organization 3
- 1.5 Chapter Summary 3

CHAPTER 2: BACKGROUND AND RELATED WORK 4
- 2.1 Control Flow Integrity (CFI) 6
  - 2.1.1 Classification of Control-Flow Transfers 8
  - 2.1.2 Classification of Static Analysis Precision 10
  - 2.1.3 The Need for Hardware-based CFI 10
- 2.2 Security Attacks that Target Control Flow 12
- 2.3 Related Work and Existing Techniques 12
- 2.4 Detection of an Attack at Various Stages 14
  - 2.4.1 Detection at the Compilation Phase (GCC) 14
  - 2.4.2 Detection at the Assembler (GAS) 14
  - 2.4.3 Detection at the Linker (libBFD.a and GLD) 15
- 2.5 SimpleScalar 15
- 2.6 Instruction Set Architectures (ISA) 16
- 2.7 Different Simulations 18
  - 2.7.1 Advantages 19
  - 2.7.2 Drawbacks 19
- 2.8 Chapter Summary 20

CHAPTER 3: PROPOSED APPROACH 21
- 3.1 Instruction Annotation 21

3.2 Hamming Code Generating Algorithm 22
3.3 Attack Situations 25
3.4 Detection of Control Flow Violation in Hardware 26
3.5 Chapter Summary 27

CHAPTER 4: EXPERIMENTAL RESULTS 28
4.1 Scenarios Addressed 28
4.1.1 Annotating the First Instruction in a Basic Block 29
4.1.2 Annotating All the Instructions in a Basic Block 29
4.1.3 Annotating Instructions in a Basic Block by Adding a New Instruction 29
4.2 MiBench Suite Benchmarks 31
4.3 SimpleScalar Usage and Modifications 31
4.4 Modifications in the Simulator 33

CHAPTER 5: CONCLUSION 37

REFERENCES 38

ABOUT THE AUTHOR ..... END PAGE

LIST OF TABLES

Table 4.1 Instruction Overheads in MiBench Benchmark Programs 30

Table 4.2 SimpleScalar Configuration Used for Proposed Approach 31

LIST OF FIGURES

Figure 2.1 Simple C Program with if-else Condition 4

Figure 2.2 Flow Chart of the Code Shown in Figure 2.1 5

Figure 2.3 Simplified CFG for the Program Shown in Figure 2.1 6

Figure 2.4 Example of Over Approximation in Static Analysis 7

Figure 2.5 Classification of Control-Flow Transfers[1] 9

Figure 2.6 Static Analysis Precision Classification for Forward and Backward Control Flow . 10

Figure 2.7 Fields in an Instruction Word 14

Figure 2.8 Overview of Device Simulator 15

Figure 2.9 Software Architecture of SimpleScalar Simulator 16

Figure 3.1 C Program with a Condition, Loop, and a Function Call 22

Figure 3.2 Assembly Code for C Program in Figure 3.1 23

Figure 3.3 Basic Block Diagram of Code in Figure 3.1 24

Figure 3.4 Algorithm for Assigning Hamming Codes for the Basic Blocks 25

Figure 3.5 Proposed 5-Stage Hardware Pipeline with Run-time CFI Check 26

Figure 4.1 Snippet of Annotated Assembly Code 32

Figure 4.2 Snippet of Output File Without Encoding 33

Figure 4.3 Snippet of Annotated Instructions for Figure 4.1 33

Figure 4.4 Algorithm for Generating Hamming Codes 34

Figure 4.5 Output of Program in Figure 2.1 35

Figure 4.6 Output of Program in Figure 2.1 After Deviation of Path 35

Figure 4.7 Algorithm for Checking the Annotations and Creating an Illegal Path..... 36

Figure 4.8 Algorithm for Checking the Hamming Distance..... 36

ABSTRACT

A popular software attack on a program is by transferring the program control to malicious code inserted into the program. Control Flow Integrity (CFI) check has been proposed as a detection mechanism for control flow deviation. In the context of embedded processors, this thesis proposes a novel approach to implement CFI to detect and stall under a control flow attack. We exploit the unused bits in an instruction word to embed a label that can be used to check CFI during runtime. Given a control flow graph, we embed a unique label in each instruction in a basic block such that a given property is satisfied by labels along a valid control flow edge. For example, the hamming distance between any two basic blocks in a legal path is less than 5 and in illegal paths, it is greater than 5. In a five stage processor pipeline, when an instruction is fetched, its label is checked against prior instruction's label for the known property (i.e., hamming distance of 5). We implemented the proposed approach in the SimpleScalar toolset and validated on 7 embedded application benchmarks chosen from MiBench benchmark suite. To the best of our knowledge, this is the first time the control flow information is embedded in the executable binary that is used for CFI check during the runtime.

CHAPTER 1: INTRODUCTION

These days, there are several security concerns around detection and prevention of attacks that deviate the flow of program under execution. Control flow analysis offers detection mechanisms to detect the attacks that target to manipulate the program's flow of execution. Enforcing the Control flow integrity (CFI) ensures that the attacks called control-flow attacks, which manipulate the execution flow are detected. Specifically, this work addresses the vulnerability of manipulating the address of the succeeding instruction stored in the program counter (PC).

1.1 Importance of the Problem and Different Approaches

Software attacks these days are based on subverting the execution of machine code. CFI, which can withstand the attacks that target to control the behavior of the program can be employed, to enforce basic safety. It is simple to execute and it ensures to safeguard from attacks from adversaries. Its implementation is possible practically, due to its compatibility with the software that already exists. CFI could be implemented in the commodity systems just by rewriting their software. Security policies that direct the memory region's access control, can be enforced using CFI through the information it provides.

External attacks on computers are generally based on manipulating the behavior of the software. These attacks reach up to the channel being used for regular communication through software flaws. Then as a next step, these attacks aim at taking control of behavior of the software which

could include calling a sensitive function. Hence, these attacks are to be detected and necessary actions are to be taken to ensure desired flow in program execution.

Several mitigation techniques to fight back these control-flow attacks, were proposed in recent years. Some of these approaches include buffer overflow eliminations in the runtime, stack canaries, etc. Few of the prevention techniques from these are used widely, though the adversaries with knowledge of these can easily bypass them. Our approach uses a unique way of annotating the instructions to detect a control-flow attack. Converting the C code into assembly instructions and incorporating the annotation data within the instruction bits is the novelty of this approach. The property of using hamming codes for validation of a specific program is a very well known for several generic approaches. But validating control-flow paths by calculating the hamming distances between two basic blocks with given hamming codes in our work is an advancement in this regard.

1.2 Proposed Approach Overview

The proposed novel approach detects if the program under execution is following only the paths it could follow legally or if an attacker has manipulated its path. Hamming distances serve as the checking parameters to check if the path followed is legal or not. Theoretically, the C program under execution is first represented as a combination of basic building blocks. Then, this code is compiled to its binary equivalent [2], after which a set of instructions belonging to a specific basic block, is assigned a unique hamming code. Hamming codes are given in such a way that they follow some specific property that differentiates legal paths from illegal paths. In this work, hamming distance between any two basic blocks in a legal path is less than 5. If the hamming distance between two blocks is greater than or equal to 5, this implies that the flow between these blocks is illegal. The hamming distance can be varied by the user based on the number of basic blocks.

1.3 Experimental Results Overview

In this approach, CFI to detect a control flow attack in the context of embedded processors [3] is implemented. The experimental results of seven C programs from MiBench benchmark suite [4] performed on RISC-like architecture, shows that when there is any code injected by an attacker, a deviation of the path from its actual path is detected with instruction overhead of 0%. To the best of our knowledge, this is the first time the information about control flow is embedded in the executable binary that is used for CFI check during the runtime, instead of maintaining control flow graphs (CFG's) separately.

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, the control flow, CFI, security attacks that change the flow of control, and the existing techniques to detect change of control flow are discussed. In Chapter 3, we describe the static analysis of instruction level basic blocks, instruction annotation, hamming code generating algorithm, attack situations, and detection of control flow violation in hardware. In Chapter 4, we discuss the experimental flow, details of the MiBench suite benchmarks, SimpleScalar usage, and modifications, and present the experimental results. Chapter 5 concludes the thesis.

1.5 Chapter Summary

In this Chapter, we introduced the concept of control flow and discussed about the importance of the problem that is being addressed. We gave an overview of the proposed approach and the experimental results obtained.

CHAPTER 2: BACKGROUND AND RELATED WORK

The order of execution of statements in a program is called its control flow. Analyzing the control flow helps to determine the order in which the instructions are executed. This can be achieved by initially binding the appropriate sets of instructions to form basic blocks (BB). Each basic block represent a set of instructions in the program that have an entry point through head instruction and an exit point at the last instruction. CFG can be handy in representation of these blocks as nodes and the control flow paths could be represented as edges between them.

```
1 int main()
2 {
3     int number = 3;
4     if(number%2 == 0)
5      {
6        printf("%d is an even integer.",number);
7      }
8      else
9      {
10       printf("%d is an odd integer.",number);
11     }
12     return 0;
13 }
```

Figure 2.1: Simple C Program with if-else Condition

All the execution paths possible can be depicted in a control flow path which facilitates the identification of legal paths and illegal paths. The sequence in which the program's control flows from the entry node to exit node is called the path and any sub-sequence of the path is called path segment. Consider the program code shown in Figure 2.1. This program can be represented as basic block diagram as shown in Figure 2.2. The basic block or node on the top represents all the

instructions before the *if* statement. The block towards the left contains all the instructions that are to be executed if the condition is satisfied. The block on the right consists of set of instructions under *else* condition (i.e., the if condition evaluates to false). The bottom block contains the set of instructions which are to be executed for sure either after BB2 or BB3.

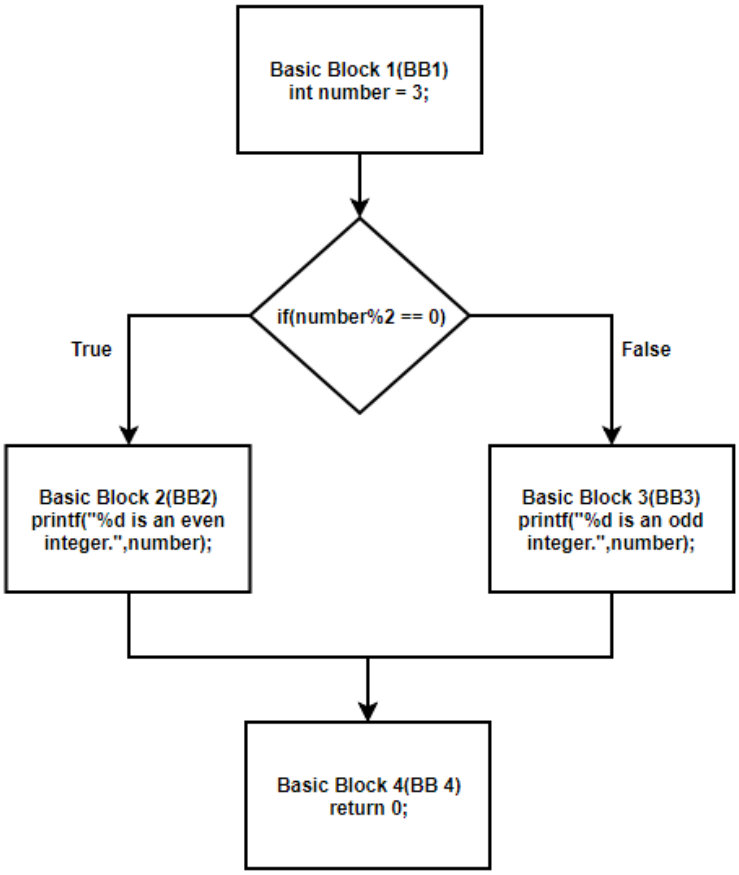

Figure 2.2: Flow Chart of the Code Shown in Figure 2.1

Further, Figure 2.3 is the CFG for the same program. The basic blocks are represented by nodes and the control flow from block to block is represented by the edges between the nodes. If there are two blocks between which the flow of control is not possible based on the program, there

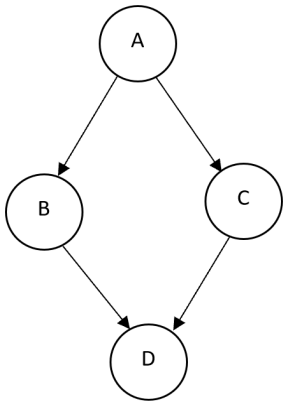

Figure 2.3: Simplified CFG for the Program Shown in Figure 2.1

will not be any edges joining the nodes corresponding to these two blocks. For e.g., in Figure 2.3 the transfer of control flow from top node to the bottom node directly is not legal. So, there is no edge joining these two nodes.

2.1 Control Flow Integrity (CFI)

CFI is the process that provide mechanisms to detect control flow based attacks that target to change the intended path of the program. The most highlighting features of CFI are efficiency, amenability to formal analysis, trustworthiness, deployability, scalability, data memory with complete control, and simplicity.

Though it is applicable for the programs to the extent of inner working, enforcement of the CFI cannot be bypassed. In CFI, indirect control-flow transfers are analyzed and the program's flow of execution is restricted during runtime based on a predetermined CFG. In the source language the targets are restricted to some constraints while at the machine level, mapped memory address that is executable is of main focus for the transfers characterized by indirect control flow.

The gap between the source code and the machine code is covered by CFI. This is done by permitting the transfer of control flow to a small target destination sets that are determined based on the location of the control flow that is indirect.

To determine the valid target sets for every indirect control-flow transfer, various mechanisms of CFI are used for the program's CFG computation. Even security promises of the CFI mechanism are dependent on the CFG accuracy. For non-trivial programs, we cannot have exact CFG.

```
1 void foo( int a ){
2  return;
3 }
4 void bar( int a ){
5  return;
6 }
7 void baz( void ){
8   int a = input();
9   void (*fptr)(int);
10  if(a){
11    fptr = foo;
12    fptr();
13  } else {
14    fptr = bar;
15    fptr();
16   }
17 }
```

Figure 2.4: Example of Over Approximation in Static Analysis

By looking at Figure 2.4, let us assume an analysis that is insensitive of flow, function foo() and function bar() will come under one class of equivalence. By looking at lines 12 and 15, any of these functions can be called. But if we look at the source code, only the foo() function has to be called at line 12 and only the bar() function should be called at line 15. Flow-sensitive analysis is a solution for this issue. But we have some over-approximations in all the static analysis of programs. As the targets are resistant to memory corruptions, there is no need for CFI to put constraints on control-flows that are caused because of direct branches. The main concentration is on branches

that can be corrupted by an attacker like jump statements, indirect calls, and returns. Mostly, it secures control-flow transfers where targets that are dependent on runtime are allowed, like the line 9 in Figure 2.4.

The main aim of CFI is to detect runtime attacks by observing the behavior of program for abnormalities. A CFG will be used to model normal program behavior. Each node in the CFG denotes a block (a group of instructions where control flows sequentially from the first instruction to the last). So at the first instruction, there is only one entry and one exit point, which is the only instruction that might cause a change in control flow. In a CFG, jumps and calls are responsible for occurrence of forward edges, while returns are responsible for occurrence of backward edges. At runtime, the dynamic control flow changes are confined to the static CFG.

Most of the CFI architectures only examine control flow changes caused by indirect branches, like calculated jumps, calculated calls, and returns. We assume that static branch targets need not be checked. By this, we can see that this cannot be used for self-modifying code.

Software-based CFI's generally look at all indirect branch targets before any indirect branch instruction is executed. Fine-grained CFI only allows control flow along some edges of a CFG. Coarse-grained CFI does not enforce a strict CFG. This depends on enforcing simple rules such as ensuring call instruction precede return targets and that the indirect branches can only target the first address in a function. They offer less security than fine-grained CFI policies.

2.1.1 Classification of Control-Flow Transfers

The two main kinds of control-flow transfers are forward *control-flow transfers*, where the control will be moved to a new location in a program and *backward control-flow transfers*, where

the control is returned to a prior location by the program. Jump and call are the two instructions given by the instruction-set architecture (ISA) of a CPU to drive the control flow forward. Each of this can either be indirect or direct. Forward control flow is of four types, one of which is direct jump where jump is to a target address that is constant determined in a static way. This is used by many local control-flows such as if-then-else statements or loops. Indirect jump is the second type in which, the jump is to a dynamically computed target address. For e.g., a dispatch table is used by statements such as switch cases. The third type of forward control type is direct call which is a call to a constant. For e.g., static function calls use this type. The fourth type is indirect call which is a call to a dynamically computed target address.

```
CF.1: Backward control-flow.
CF.2: Forward control-flow with direct jumps.
CF.3: Forward control-flow with direct calls.
CF.4: Forward control-flow with indirect jumps.
CF.5: Forward control-flow with indirect calls
      supporting function pointers.
CF.6: Forward control-flow with indirect calls
      supporting vtables.
CF.7: Forward control-flow with indirect calls
      supporting Smalltalk-style method dispatch.
CF.8: Complex control flow to support exception
      handling.
CF.9: Control-flow supporting language features
      like dynamic linking, separate compilation.
```

Figure 2.5: Classification of Control-Flow Transfers[1]

Control-flow transfers will be tough, if there are exceptions. The control-flows get locally complicated if there is exception Handling. For inter-procedural control-flows, stack frames are to be unwinded on the currently available stack until we find an exception handler that is suitable and matching. Other considerable issues by CFI mechanisms with respect to control flow are dynamic linking, separate compilation, and compilation of libraries.

During compilation, we may not know the complete CFG. This problem can be solved in two ways either by constructing a combined CFG or by relying on Link-time Optimization (LTO). Binary modules can help in recovering every control flow. Figure 2.5 shows the classification of control-flows [1].

2.1.2 Classification of Static Analysis Precision

The precision of the CFG computed governs the CFI mechanism guarantees in terms of the security. Based on static analysis precision, the classifications for forward control flow in ascending order and also the classifications that are critical for the precision in analysis of backward control flow transfer are shown in Figure 2.6.

```
SAP.F.0: Forward branch validation is absent
SAP.F.1a: ad-hoc algorithms and heuristics
SAP.F.1b: Analysis of context and insensitivity of the flow
SAP.F.1c: Equivalence classes are labeled
SAP.F.2: Analysis of class hierarchy
SAP.F.3: rapid-type analysis
SAP.F.4a: Analysis of flow sensitivity
SAP.F.4b: Analysis of context sensitivity
SAP.F.5: Analysis of context sensitivity and flow sensitivity
SAP.F.6: dynamic analysis (optimistic)
SAP.B.0: Backward branch validation is not present
SAP.B.1: Equivalence classes are labeled
SAP.B.2: Shadow stack
```

Figure 2.6: Static Analysis Precision Classification for Forward and Backward Control Flow

2.1.3 The Need for Hardware-based CFI

In most of the software-based CFI solutions, to perform CFI checks on indirect branches, we insert code into a program. This will be done as part of a compiler optimization step. But if the compiler does not know the security aspects concerning the CFI checks, the optimization step might

spill registers holding sensitive CFI data to the stack. Many software-based CFI architectures are dependent on runtime data structures which are stored in writable memory. Few attacks made use of this, by altering with runtime data structures to circumvent the security of the system. Instruction-level isolation techniques can be used to protect runtime data structures. For e.g., Software-based Fault Isolation (SFI) can be used. SFI ensures that each store instruction can only write into a specific memory space. Instruction-level isolation techniques rely on the integrity of code.

Runtime data structures and meta-data can be protected by hardware-based access control mechanisms. A little overhead will be incurred and it does not rely on integrity of code to protect sensitive information. The Trusted Computing Base (TCB) is a set of software components and hardware components which are critical to the system's security. The careful design and implementation of these components are the basics for the overall security of the system. The components of the TCB are made in a way that the device does not misbehave even if the other parts of the system are exploited. To guarantee its correctness, TCB should be as small as possible.

Hardware-based mechanisms can give security against strong attackers, such as attackers who have control on both code and data memory. Also, direct branches and unintended branches are allowed for verification along with indirect branches. The TCB size is usually small, as functionality related to security is mostly implemented in hardware with little or no trusted software.

In hardware-based mechanisms, fewer processor cycles need to be spent on performing security checks so that it provides better performance. Selection of components to be placed inside the TCB plays an important role in security-critical systems. To communicate runtime information with the CFI hardware or for configuring the CFI hardware, software components are needed. Care should be taken to ensure the following:

- Hardware/software interface cannot be exploited.
- Hardware-based security policies cannot be circumvented.
- TCB components are correct.
- Sensitive data stored in main memory is secure.

2.2 Security Attacks that Target Control Flow

There are several security attacks that aim at redirecting the program execution to the desired code that is already existing in the program or to an instruction in some other program. In general, the attackers can either inject some new instruction or they can also change the value of which the PC is currently pointing to. The CFG is violated during runtime when an attacker changes the address of the next instruction that the PC stores. The injected data corrupts the information in the PC and hence manipulates the control flow of the program. Any software is vulnerable to these control flow attacks during runtime due to the unavoidable dependency of the program under execution on the PC [5]. CFI deals with these attacks and provides mechanisms to detect if any malware was introduced into the program by an attacker.

2.3 Related Work and Existing Techniques

There are already few software and hardware based solutions proposed by the early researchers and recent CFI enthusiasts. Software based defence mechanisms focus on enforcing CFI during runtime by validating the program's CFG. This enforcement has a minimal overhead and simple implementation. Recent research work is based on application of CFI to COTS [6] binaries. But they do not cover the case where there is a hand off of control flow to and fro between program

code of the application and that of the libraries. But rewriting binaries limit the functionality of generating the code dynamically. Return Oriented Programming attacks can be eliminated using G-Free which is an approach based on the compiler [7]. This approach fails to protect the control flow from getting redirected to an existing instruction rather, but can secure from code injections. Transfers of control can also be monitored during runtime using program shepherding. Though it resembles the control data isolation technique [8], it needs support during compilation to validate the branching. It uses the actual CFG to make a symbol of it. Since this symbol is not the actual CFG, dynamic verification is not possible. This is a drawback of program shepherding compared to control data isolation. The reason for this is the fact that, the CFG of an application identified, cannot be validated before the execution.

Compiler based approaches [9] gained lot of interest by the researchers and proved to be the foundation for several other proposals around the CFI space based on compilers. Compilers like *llvm* and *gcc* were attempted to incorporate CFI validations using forward edge CFI with compile-time labels [10]. However, this does not cover the vulnerabilities due to return instructions and code gadgets. Another CFI implementation called control flow restrictor targets to provide control flow security for iOS and it is also based on compiler [11]. But this work does not address the problem with complex applications but limits their range to an ARM controller running on iOS.

In addition to all the software based mechanisms, there are also solutions based on hardware. The root for all the hardware based solutions is to provide architectural support to the software based mechanisms to enforce CFI. Performance counter [12] is one such architectural support that could serve as a watchdog to check for CFI violation. Drawback of this work is that, it is vulnerable to heap spray attacks since it allows the execution of branches that are suspicious.

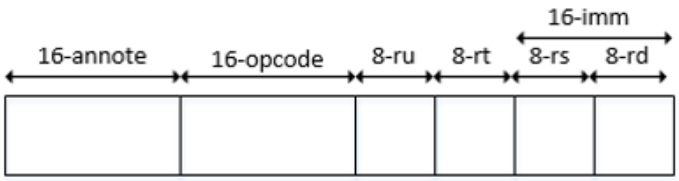

Figure 2.7: Fields in an Instruction Word

2.4 Detection of an Attack at Various Stages

Consider a user-written source code that is potentially the target for an attacker. There exist three possible phases to detect an attack. They are: the compiler, the assembler, and the linker.

2.4.1 Detection at the Compilation Phase (GCC)

It is the most feasible phase to detect an attack by using instruction annotations in the allocated bits. To avoid the painful difficulties in modifying GLD, GAS, and libBFD.a, care should be taken that no new types of linkages are added. In general, each instruction has two parts: inst.a (opcode) and inst.b (operand). There are 16 bits (0 to 15 bits) in inst.a to hold the annotations as shown in Figure 2.7. Annotations can be of two types. One of which is bit annotations represented as /a - /p which sets bits 0 to 15. For e.g., sw/a $r5, 4($r7). The other type is field annotations, represented as /s:e(v) which means it set bits s to e with value v. For e.g., sw/15:0(2) $r5, 4($r7).

2.4.2 Detection at the Assembler (GAS)

Dealing with the assembler should be avoided for the level of difficulty involved in this. Also, if new types of linkages are used, modification of libBFD.a and GLD is required which is very difficult.

2.4.3 Detection at the Linker (libBFD.a and GLD)

Due to the delicate nature of both of these tools, this process should be avoided.

2.5 SimpleScalar

For the validation of our idea, we used the SimpleScalar toolset. It is not easy to optimize all the three components: performance, flexibility, and detail. So, most of the implementations satisfy either one or two components. The SimpleScalar toolset acts as a base for simulation and architectural modeling. This provides different levels of hierarchies starting from simple unpipelined processors to detailed dynamic micro-architectures. In general, architectural simulator is a tool used for reproducing the behavior of a computing device. The overview of a device simulator is as shown in Figure 2.8.

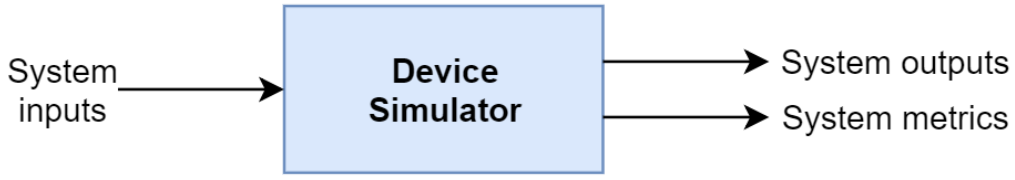

Figure 2.8: Overview of Device Simulator

There are various uses of a simulator. Some of the uses are that the system instrumentation can be enhanced, software development can be made more flexible, before the availability of the hardware itself the software can be validated and it allows in depth analysis of the design space. SimpleScalar simulators execute all the instructions in a program with the help of an interpreter and then create some computational operations. The SimpleScalar tool set also have visualization tools for performance, statistical analysis resources, and debug and verification infrastructure.

2.6 Instruction Set Architectures (ISA)

Instruction Set Architectures (ISA) which can be simulated by SimpleScalar are Alpha and PISA. The PISA instruction set is mainly used for instructional use. The binaries compiled for the SimpleScalar architecture are given as input to the tool set and their execution will be simulated on one of processor simulators. The machine on which SimpleScalar is running is called the Host and the ISA (Alpha configuration or PISA configuration) is referred as Target. Though the gcc cross-compiler suggests that the underlying machine executing the PISA instructions is a PISA machine, but it is actually not. Figure 2.9 illustrates this notion.

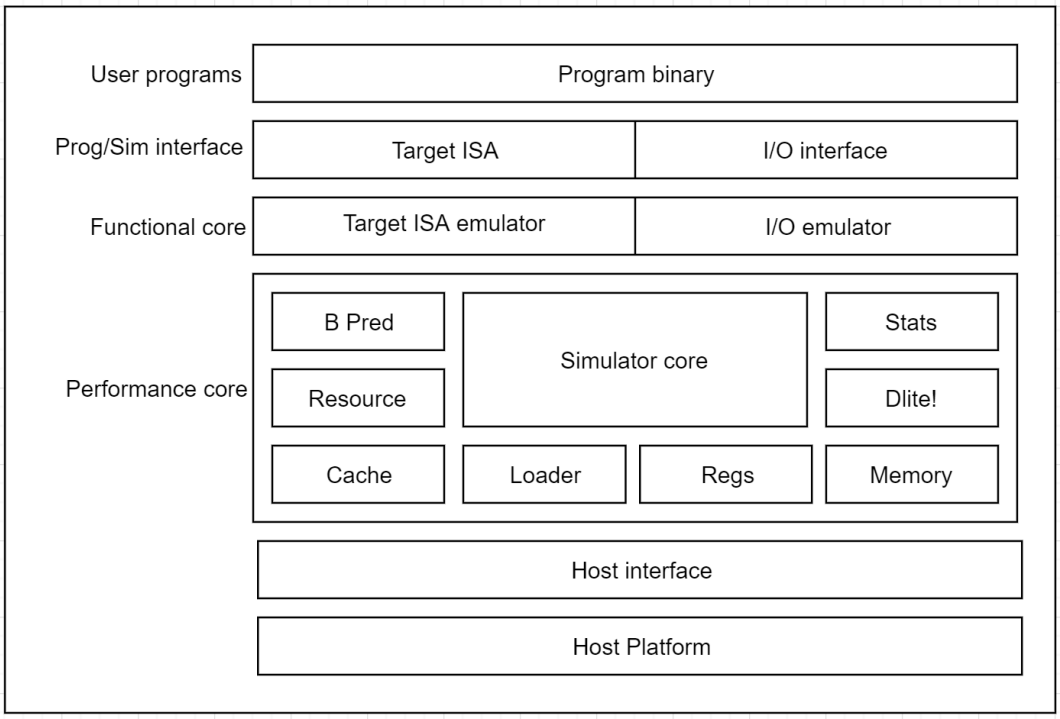

Figure 2.9: Software Architecture of SimpleScalar Simulator

Execution-driven simulation method is used for running applications. For this, an I/O emulation module and instruction-set emulator are required. Each instruction is interpreted by the instruction-set emulator, directing the hardware model's activities through callback interfaces. The

module for I/O emulation gives access to the programs simulated to external input and output facilities. For system-call emulation, when a system call is attempted to be executed by a program in the instruction set interpreter, the I/O module is invoked by the system. With the emulation of I/O to the device level, the portion of the OS for an application's execution can be analyzed. The simulator core, in the middle of the model, is responsible for defining the instrumentation and organization of the hardware model.

In 1992, SimpleScalar was initially a part of the Multiscalar project. This toolset was made available to academic users as open source distribution with the assistance of Doug Burger in 1995. The tool is now maintained by SimpleScalar LLC and is available at http://www.SimpleScalar.com.

To create computer system models, SimpleScalar can be used as a base infrastructure, which simplifies implementation of hardware models that are capable of simulating complete applications. The tool set consists of many simulators which follow the microprocessor at various levels. Some of these are as listed below:

- *sim-fast*: It is a fast and speed optimized interpreter. The behavior of the caches, pipelines, or any other micro-architecture parts is not accounted in this. Only functional simulations are performed and in-order execution of the instructions is used. 780 lines of code can be simulated with a speed of 7 MIPS.

- *sim-safe*: It is slightly slower instruction interpreter when compared to sim-fast. On all memory operations, memory alignment, and memory access permission will be checked. If sim-fast crashes without any information, then this can be used. 320 lines of code can be simulated with a speed of 6 MIPS.

- *sim-profile*: It acts as a dynamic program profiler. It has a track of text and data segment profiles, adress modes usage, counts of instruction class, and dynamic instruction counts. 1300 lines of code can be simulated with a speed of 4 MIPS.

- *sim-cache*: It is a multi-level memory cache simulator. Various levels of caches of data and instruction can have different sizes and organizations. If cache performance effect is not necessary, this simulator can be used for fast cache simulation. 1400 lines of code can be simulated with a speed of 4 MIPS.

- *sim-bpred*: It is a branch predictor simulator. Many branch prediction schemes will be simulated and gives a result as miss rates and hit rates. Even with this, the impact of the branch prediction on the execution time is not simulated precisely. 1200 lines of code can be simulated with a speed of 5 MIPS.

- *sim-outorder*: It is a detailed micro-architectural timing simulator. This is a highly parameterized simulator which may match systems with varying numbers of the execution units. 3900 lines of code can be simulated with a speed of 0.3 MIPS.

2.7 Different Simulations

In one of the situation, execution-driven simulation is used in all SimpleScalar models. For computer system models, instruction execution should be reproduced on the simulated machine. A different situation for the above one is the trace driven simulation, in which, a hardware timing model receives an input of a set of instructions that are prerecorded. For collection of instruction traces, many techniques (software and hardware) such as trace synthesis, binary instrumentation, or hardware monitoring are used.

2.7.1 Advantages

Execution-driven simulation has many benefits when compared to trace-based techniques. While execution of a program, access is given to all the consumed data and produced data, which is helpful for optimization research such as analysis of dynamic power, compressed memory systems, and value prediction.

Simulation driven by execution also facilitates higher accuracy in speculation modeling. Till the simulation finds an incorrect prediction, speculative execution will continue at a higher throughput, as it restarts with corrected values by flushing the processor pipeline. Before this recovery, the program runs slowly as the mis-speculated code execution and non-speculative execution compete with one another for resources. In trace-driven techniques, only correct program execution will be recorded, so mis-speculated code execution cannot be modeled.

2.7.2 Drawbacks

The two main drawbacks of execution-driven simulation are: (a) complexity in the model is high; and (b) reproducing the experiments is difficult.

Since the instruction and I/O emulators are required by the execution-driven models, for calculation, there will be increase in model complexity. But, that is not the case with the simulators that are driven by trace. This complexity can be reduced using SimpleScalar, by leveraging the set of internal tools with which the emulators are synthesized and defined target-language.

This target definition gives a base for determining the complexities of the execution of an instruction, that include operational semantics, side effects of memory and register, and faulting semantics of an instruction. Replicating experiments that rely upon real world events might be

difficult, as simulators which are execution-driven are directly connected with I/O emulation to the input devices.

In trace-driven experiments, effects of external inputs are noted by instruction traces within the instruction trace file and hence they do not experience the difficulties as above. To handle this issue of reproducibility, SimpleScalar provides an external input tracing feature. During the execution, the inputs to a program by some external devices will be recorded by these traces. Then the traced inputs are replayed and the original execution is recreated.

2.8 Chapter Summary

To summarize, most of the previous approaches fall under mitigation techniques that initially try to detect a control flow attack and protect the program code against it. There are several assumptions involved from the security perspective which could increase the vulnerabilities in the program [13]. Our work is also a mitigating technique but does not involve any assumptions that make the CFI more vulnerable. Incorporating the annotations in binary instructions allows us to detect if any control flow attack has happened during runtime. This is a novel approach as it binds the annotations that could be used to stall the execution of instructions that violate the annotation criteria.

CHAPTER 3: PROPOSED APPROACH

One interesting aspect of the proposed approach is that we do not need external CFGs to check the control flow. We only take into account the branch and jump type instructions. Let us consider a simple C program as shown in Figure 3.1, which consists of different control statements (if condition, for loop, and a function call) vulnerable to attack. In order to extract the BBs in instruction level, this C program is assembled to generate instruction file as shown in Figure 3.2.

In this figure, control instructions such as branch, jump help in dividing the instructions into basic blocks. Control instruction have address to jump to. We allot a new BB only if such addresses are forward addresses. In case there is a backward address, the control is passed to a BB that already exists.

In general, Number of BBs is determined by the number of forward addressed instructions (branch + jump). In Figure 3.3, we show the BBs snippet for the C program in Figure 3.1. For each BB, it ends with a control that diverts instruction to a new block. Notice that we have a backward addressed jump instruction in BB5 that goes to BB3. Here, as discussed, the control goes back to an existing block rather than generating a new basic block.

3.1 Instruction Annotation

We make use of the annotations fields as shown in the Figure 2.7, or some labels to encode the instruction. For the purpose of encoding, we assumed a known property i.e., the hamming distance.

```
1. int incr(int i)
2. {
3.     printf("%d\n",i++);
4. }
5. int main()
6. {
7.  int i, a=10;
8.  if(a<1)            //condition
9.     a++;
10.  else
11.     a--;
12.  for(i=0;i<10;i++)  //loop
13.     a++;
14.  incr(a);           //function call
15.  return 0;
16. }
```

Figure 3.1: C Program with a Condition, Loop, and a Function Call

Then, we assign hamming codes to each of the basic blocks by following a specific property of hamming distance. Based on this, we manually assigned the hamming codes to basic blocks. In a five stage processor pipeline as we considered, we fetch an instruction and we check its label against prior instruction's label for the hamming distance. Assignment of hamming codes is discussed in Section 3.2.

3.2 Hamming Code Generating Algorithm

In general, we need to consider the number of bits a hamming code should be and then we assign the hamming codes to each basic block manually by following a specific algorithm as shown in Figure 3.4. As shown in this figure, we first assign hamming code to the root node and then proceed to the nodes in the succeeding level. In the process of assigning hamming codes to nodes in each level, let us suppose that we have a backward jump from a node (say in level 3) to a node (say in level 2). In this case, we encounter a conflict by following Breadth First Search (BFS) as

```
400298:  addiu $2,$0,10
4002a0:   sw $2,20($30)
4002a8:   lw $2,20($30)
4002b0:   bgtz $2,4002e0
4002b8:   lw $3,20($30)
4002c0:   addiu $2,$3,1
4002c8:   addu $3,$0,$2
4002d0:   sw $3,20($30)
4002d8:   j 400300
4002e0:   lw $3,20($30)
4002e8:   addiu $2,$3,-1
4002f0:   addu $3,$0,$2
4002f8:   sw $3,20($30)
400308:   sw $0,16($30)
400310:   lw $2,16($30)
400318:   slti $3,$2,10
400320:   bne $3,$0,400330
400328:   j 400378
400330:   lw $3,20($30)
400338:   addiu $2,$3,1
400340:   addu $3,$0,$2
400348:   sw $3,20($30)
400350:   lw $3,16($30)
400358:   addiu $2,$3,1
400360:   addu $3,$0,$2
400368:   sw $3,16($30)
400370:   j 400310
400378:   lw $4,20($30)
400380:   jal 4001f0     <incr>
400388:   addu $2,$0,$0
```

Figure 3.2: Assembly Code for C Program in Figure 3.1

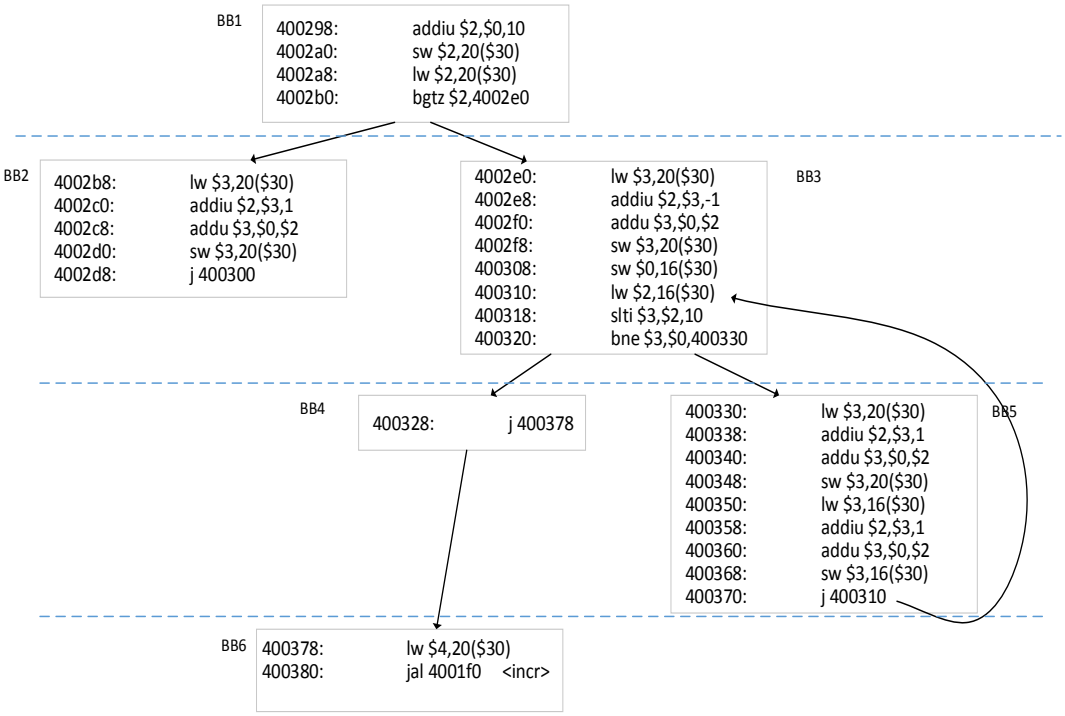

Figure 3.3: Basic Block Diagram of Code in Figure 3.1

the node (in level 2) will already be marked as visited. For this reason, we check if the node is visited or not in the algorithm. Also we need to have a record of all the hamming codes which we are using, as we do not want any hamming code to be repeated. At each level, by considering the legal paths, we assign one hamming code for a basic block from a set of possible codes and discard the rest. Our assumption is that hamming distance is less than 'k' for legal paths and greater than or equal to 'k' for illegal path. So, for this we need to check the hamming codes of each block to others. Extra bits needed for annotating instruction is given by *log(number of BBs)*.

```
Data: V = Basic Block, E = control flow, h is predefined hamming
      distance
Result: Hamming codes assigned to a CFG
// Step 1 - Assign the first node of CFG with some random hamming
 code and keep record of it.
 // Step 2 - Traverse the graph in Breadth First Search manner.
 for node N in graph do
    if N = not visited then
       Assign hamming code such that the hamming distance between
        N and its parent node is less than h. Keep record of this
        hamming code.
    else
       // Node has more than one parent.
        1. Assign hamming code such that the hamming distance
        from N to all of its parent nodes should be less than h. Keep
        record of this hamming code.
        // As the hamming code of N is changed, its child nodes get
        different hamming codes.
        2. Re-assign the hamming codes for the child nodes of N and
        keep record of these codes.

    end
end
```

Figure 3.4: Algorithm for Assigning Hamming Codes for the Basic Blocks

3.3 Attack Situations

These days, we come across various advanced attacks. Handling all of them is very difficult with a single approach. One such common runtime attack is buffer overflow attack, due to which the flow of the program gets diverted to a different path, generating wrong outputs. In our approach, we restrict that the address modified by an attacker would divert the program path to an illegal path rather than its actual path. For e.g., if an attacker adds some value to the current PC value, the PC now points to an instruction in some other basic block (which creates an illegal path). Since different blocks are encoded with different hamming codes and as the hamming distance will be greater than 5 here, we can understand that there is some malicious attack.

One of the limitation for our approach is that, the PC value modified by an attacker should not point to some other instruction in the same block. That means the modified PC value should point to some other instruction that is encoded with a different hamming code, or the encoding property should be different from ours or it should not be encoded at all. Also since CFG is a directed graph (an edge from u to v is allowed but an edge from v to u may not be), in our approach it is an assumption that backward flow is not allowed to its immediate parent node.

3.4 Detection of Control Flow Violation in Hardware

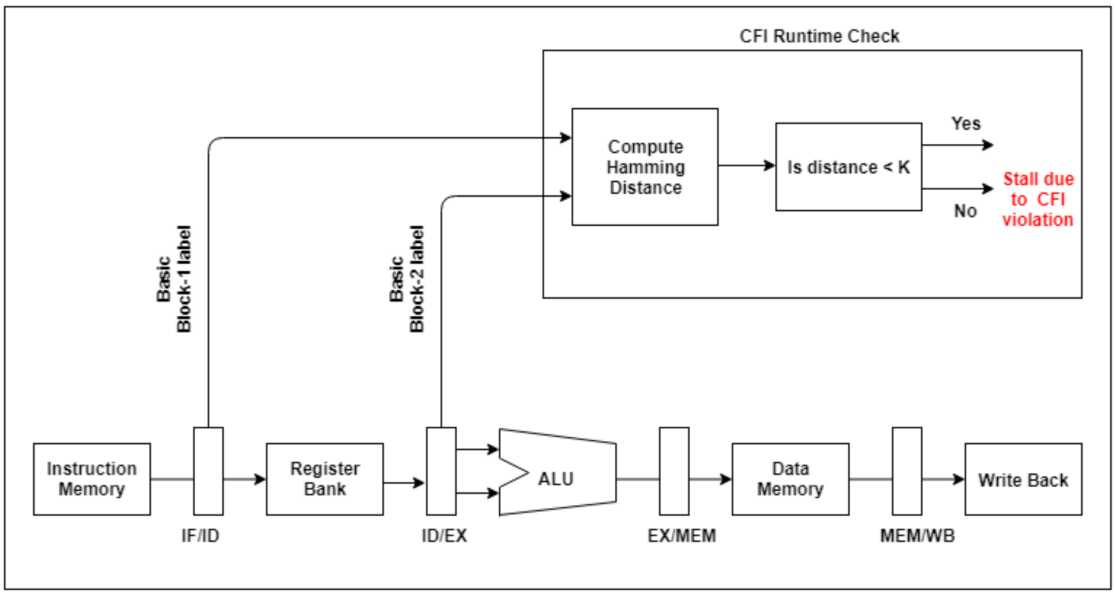

Figure 3.5: Proposed 5-Stage Hardware Pipeline with Run-time CFI Check

Detection of control flow violation in hardware is demonstrated in Figure 3.5. The control instructions such as branch and jump instructions trigger the hamming distance comparator. When the control encounters the first control statement instruction, it stores that instruction into the annotate register. The reason why it should be stored is that, the hamming distance comparator requires two hamming codes to calculate the hamming distance between them. Now, a check of

hamming distance for a given property is done in the hamming distance comparator (for e.g., in our approach it checks if the distance is less than 5, or greater than or equal to 5) and it gives an interrupt if it finds that there is some deviation of path in our program due to some malicious activity (i.e., when hamming distance is generated as 5 or greater than 5 in our case). In the hamming distance comparator, XOR operation is first performed for the given two hamming codes and the resultant output is given as input to a 1-counter which counts the number of 1's in the code, thus finding hamming distance. Depending on this hamming distance, control flow violation will be detected in the hardware pipeline.

3.5 Chapter Summary

In this chapter, the proposed approach to implement runtime CFI check with instruction encoding is presented. Static analysis on the assembled instruction code of one of the C program is done in order to find Basic Blocks (BB) at the instruction level. Further, hamming code assignment algorithm is presented to encode each BB node. Later, is the demonstration of an attack model that will deviate the actual path of the program. The detection of control flow violation in hardware is also demonstrated in this chapter.

CHAPTER 4: EXPERIMENTAL RESULTS

To start with the discussion of our experimental approach, we first compiled the C program with the gcc of SimpleScalar toolset. From this, an object file is generated, which is then used for obtaining the assembly level code as shown in Figure 3.2. Considering the number of branch and jump instructions, we will divide the code into different Basic Blocks as in Figure 3.3.

Later, we assigned different hamming codes for each basic block by following certain property. In our approach, for most of the programs we followed a property that all the basic blocks in a legal path should have hamming distance of less than 5 and all those basic blocks which are in the illegal path should have a hamming distance greater than or equal to 5.

4.1 Scenarios Addressed

In our approach we address two different scenarios. In the first case, the bits are sufficient in an instruction word to embed the label. In the second case, the bits are insufficient in an instruction word to embed the label. As discussed below, we propose three different approaches for label embedding. Approaches 4.1.1 and and 4.1.2 come under the first scenario while approach 4.1.3 is for the second scenario. Section 4.1.1 describes embedding the first instruction in a basic block. In Section 4.1.2, we embed all instructions in a basic block. Inserting a new instruction at the beginning of the basic block is described in Section 4.1.3. Approaches described in section 4.1.1 and 4.1.2 will not have code-bloat while approach in section 4.1.3 will have a code-bloat.

4.1.1 Annotating the First Instruction in a Basic Block

The approach here is to annotate the first instruction in all the basic blocks. One of the limitation that we encounter here is that we need to assume that the new PC value modified by the attacker should always point to the first instruction in some basic block. As none of the other instructions in the block are annotated, error can not be detected if the control points to some other instruction in a basic block other than the first instruction. One extension which can be done to this is annotation of all the instructions in a basic block as in the next section (Section 4.1.2).

4.1.2 Annotating All the Instructions in a Basic Block

In this method, we annotate all the instructions in a basic block with the hamming code of that particular block (for e.g., if BB1 is assigned a hamming code of 111111, each and every instruction belonging to that particular BB will be annotated with this hamming code). In this case, an attack can be detected even if the attacker modifies the PC value such that the control points to any instruction in the block. But in this scenario, we should limit that the diverted path should be a part of your program. Or, if we consider that the attacker changes the path to some other program, an assumption is that, the instruction to which the control is diverted should not be annotated or it should be annotated with some other property that is different from ours. This approach has higher probability for detecting an attack than the first approach.

4.1.3 Annotating Instructions in a Basic Block by Adding a New Instruction

For some instructions in real implementation, all the bits will be used by the instruction such that we do not have any unused bits for annotation. In such cases, we add a new instruction to perform encoding and we assign it with some label. In the first two approaches the new instruction

overhead will be 0%, as we are not adding any new instructions. Though there is hamming distance check being done, no performance overhead is incurred because it is being done in parallel and transparent to the user during execution stage as shown in Figure 3.5. As this check is not on the critical path, it will not result in any increase in clock period. But in the third case, we have some percent of overhead depending on the percentage of number of branch and jump instructions to the total number of instructions in a program.

The circuit overhead comes from the CFI check block shown in Figure 3.5. It consists of an 8-bit XOR and 8 bit comparator. When compared to the number of gates of blocks in the pipeline (Pipeline registers, ALU, Instruction, and Data Memory, etc.), this overhead is minimal (<0.1%).

Table 4.1: Instruction Overheads in MiBench Benchmark Programs

MiBench Benchmark	Program Name	Number of branch (b) instructions	Number of jump (j) instructions	Total (b) + (j)	Total instruction count	% of instructions responsible for new BBs	Detection
Automotive/ qsort	qsort_small	6	19	25	178	14	yes
Automotive/ basicmath	cubic	2	19	21	262	8	yes
Network/ dijkstra	dijkstra_small	16	40	56	414	13.5	yes
Office/ ispell	sq.c	3	11	14	72	19.4	yes
Security/ sha	sha_driver	3	13	16	579	2.8	yes
Telecom/ adpcm	timing	3	18	21	138	15.2	yes
Telecom/ adpcm	rawcaudio	3	12	15	59	25.4	yes

4.2 MiBench Suite Benchmarks

In a design process that considers performance as metric, evaluation against well known benchmarks is critical. We used MiBench benchmark suite to evaluate our results. MiBench benchmark suite is categorized into six different suites, where each suite represents a particular area in the embedded market. The source code in this benchmark is freely available. The percentage of control instructions (due to which we have small percent of overhead), for some of the benchmark programs from MiBench suite is listed in Table 4.1.

4.3 SimpleScalar Usage and Modifications

Table 4.2: SimpleScalar Configuration Used for Proposed Approach

Execution Type	In-order
Fetch queue	1
issue width	1
Decode Width	1
Commit Width	1
Issue in order	True
Functional Units	1 integer ALU, 1 FP ALU, 1 FP Mult
L1 I-cache	32KB, 1-way
L1 D-cache	32KB, 1-way
L2 cache	None
Memory bus width	4
Memory latency	12
RUU size	2
LSQ size	8

From the two available instruction sets (Alpha and PISA) in SimpleScalar, we used PISA that has a RISC architecture and consists of MIPS type of instruction set. Out of many micro-architectural simulators provided by the SimpleScalar toolset, we used Sim-outorder simulator. Sim-outorder simulator is a detailed micro-architectural timing simulator that is trace-driven dy-

namically. This is a highly parameterized simulator which may match systems with varying numbers of the execution units. The issue of out-of-order is supported by this simulator. It has the ability to model an out-of-order microprocessor that has caches, external memory, and branch predictions. In general, 3900 lines of code can be simulated with a speed of 0.3 MIPS by this simulator. Sim-outorder simulator's configuration on which we generated results is as shown in Table 4.2.

To start with our actual implementation, we have annotated the instructions by making use of the 16-bit annotation field. For this, we first convert C code to assembly level code by using *sslittle-na-sstrix-gcc -S sample.c* command. Now, sample.s file will be generated by default. It is in this file where we annotate the instructions using field annotations as shown in Figure 4.1.

```
BB1:  ....
      bne/15:0(136)     $3,$0,$L2
BB2:  la/15:0(155)      $4,$LC0
      ....
BB3:  la/15:0(104)     $4,$LC
      ....
BB4:  move/15:0(123)   $2,$0
      ....
```

Figure 4.1: Snippet of Annotated Assembly Code

If we compile the assembly level code of Figure 4.1, using *sslittle-na-sstrix-gcc sample.s* command, a non-readable output file will be generated. In order to convert it to a readable file and check if the instructions are encoded, we dump this code into a file named sample-output.txt, using the command: *sslittle-na-sstrix-objdump -x -d -l a.out > sample-output.txt*

If we generate the output without annotating the instructions in sample.s file, then the output would be as shown in Figure 4.2. In this Figure, the bits that are represented in blue, are the bits reserved for the purpose of annotation. We can observe that all those bits in annotation field were initially 0. We annotate the instructions like those shown in Figure 4.1 the output file looks like the

```
.........
400238:     06 00 00 00    bne $3,$0,400268
400240:     a2 00 00 00    lui $4,4096
.........
400268:     a2 00 00 00    lui $4,4096
.........
400288:     42 00 00 00    addu $2,$0,$0
.........
```

Figure 4.2: Snippet of Output File Without Encoding

```
.........
400238:     06 00 88 00    bne $3,$0,400268
400240:     a2 00 9b 00    lui $4,4096
.........
400268:     a2 00 68 00    lui $4,4096
.........
400288:     42 00 7b 00    addu $2,$0,$0
.........
```

Figure 4.3: Snippet of Annotated Instructions for Figure 4.1

snippet shown in Figure 4.3. Now, if we focus on the bits in blue, we can see that there is a change in the bits.

As it is very difficult for us to form the hamming codes for each and every block, we followed a program to generate different hamming codes within different hamming distances as shown in Figure 4.4. In the program, we initially give one hamming code as input, for which we want other hamming codes within the hamming distances from 0 to 8 (since we used 8-bit hamming code). All we need to do is to check for a code that satisfies our property and allocate it for that block.

4.4 Modifications in the Simulator

The output of the program in Figure 2.1 will be as shown in Figure 4.5. For validating our approach we intentionally injected a line of code that manipulates the PC value such that the flow

```
Data: str = character pointer, i = len - 1, changesLeft = i
Result: Hamming codes within different hamming distances.
//Input a hamming code for which you want other hamming codes
 within different hamming distances
 //len = maxDistance = length of string
 for i = 1 to maxDistance do
 |  //call hamming function
 |
end
//start of hamming function
 if changesLeft = 0 then
 |  //print the string and return
 |
end
if changesLeft < 0 then
 |  return;
end
Flip the current bit
 hamming(str, i-1, changesLeft-1)
 Flip the bit again to undo
 hamming(str, i-1, changesLeft)
 //End of hamming function
```

Figure 4.4: Algorithm for Generating Hamming Codes

of program becomes illegal. The output of the program in Figure 2.1 after code injection will be as shown in Figure 4.6. We can observe that the hamming distance between Basic Blocks in Figure 4.5 is 3 (less than 5, which implies that the path is legal). In Figure 4.6, we observe that the hamming distance is 6 (greater than 5, which means that the path is illegal).

With the proposed approach we were able to detect the control flow based attacks successfully. The changes made to the simulator in our approach is as shown in Figure 4.7. We initially retrieve the annotated value of the instruction in one BB and store it in the variable "anno1". We then retrieve the annotated value of the instruction in next BB and store it in the variable "anno2." We compare the hamming distance between "anno1" and "anno2" as shown in Figure 4.8.

```
sim: ** starting functional simulation w/ caches **
Actual inst.b: 0x300000a
Hamming code 1: 136, PC value: 0x400238
Hamming code 2: 104, PC value: 0x400268
Hamming distance: 3
3 is an odd integer.
End of program
Sim: ** simulation statistics **
Sim_num_insn            10215 # total number of instructions executed
Sim_num_refs             5078 # total number of loads and stores executed
```

Figure 4.5: Output of Program in Figure 2.1

```
sim: ** starting functional simulation w/ caches **
actual inst.b    : 0x300000a
modified inst.b: 0x3000012
hamming code 1: 136, PC value: 0x400238
hamming code 2: 123, PC value: 0x400288
Hamming distance: 6
panic: illegal path
sim: ** simulation statistics **
sim_num_insn             178 # total number of instructions executed
sim_num_refs              49 # total number of loads and stores executed
```

Figure 4.6: Output of Program in Figure 2.1 After Deviation of Path

Coming to the overheads, when there is protection for return addresses in the original CFI implementation [14] the overhead is 21%. Also, when there is protection for return addresses and function pointers, in one of the powerful CFI implementations [15] the average performance overhead is 10%. But when we consider our approach, we have a performance overhead of 0% (in the approaches 4.1.1 and 4.1.2 where labels were not used), which is very effective.

```
Data: a = PC value at start of program, b = PC value at end of
      program
Result: Diversion of the program flow to an illegal path
// Firstly check if the PC value is pointing to the current program.
 if regs.regs_PC ≥ a and regs.regs_PC ≤ b then
    if set = 1 then
       set=0;
        //Retrieve second annotated value
        anno2 = (int)((inst.a » 16) & 0xff);
        // call function for checking hamming distance shown in 4.8
        dist = hammingDistance(anno1, anno2);
    end
    if operator ≠ "bne" then
       //Retrieve first annotated value
        anno1 = (int)((inst.a » 16) & 0xff);
        //manipulating PC value
        inst.b = inst.b + 8;
        set=1;
    end
end
```

Figure 4.7: Algorithm for Checking the Annotations and Creating an Illegal Path.

```
Data: anno1 = value of annotated bit in a BB, anno2 = value of
      annotated bit in next BB, x = XOR of anno1 and anno2,
      setBits = 0
Result: Hamming distance between two Basic Blocks.
while x >0 do
    setBits = setBits + (x & 1);
     x = x » 1;
end
//return setBits
```

Figure 4.8: Algorithm for Checking the Hamming Distance.

CHAPTER 5: CONCLUSION

In our work, we present an approach to implement CFI to detect and stall under a control flow attack. We demonstrated that control flow information can be successfully embedded such that it can be used for runtime CFI check. We redesigned the hardware pipeline such that it has the ability to detect attack in run-time. We proposed three approaches in which there is no instruction and execution overhead in two of the approaches (when we have sufficient bits to embed a label). Additionally, we also prevented the control flow attack on 7 embedded benchmarks chosen from MiBench benchmark suite.

In our approach, we assigned the hamming codes manually which will be difficult for larger programs. In such cases, the number of hamming code bits required to maintain the chosen hamming distances for legal vs. illegal paths will be huge. As an extension to our work, this limitation can be addressed by choosing some other property that distinguishes the legal paths from the illegal ones. Our work is applicable to light-weight processors running dedicated programs.

REFERENCES

[1] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *CoRR*, abs/1602.04056, 2016.

[2] W. Chunlei, Z. Gang, and D. Yiqi. An efficient control flow security analysis approach for binary executables. In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 272–276, Aug 2009.

[3] O. Qingyu, H. Kai, and W. Xiaoping. Research on the embedded security architecture based on the control flow security. In *2009 Second International Workshop on Computer Science and Engineering*, volume 1, pages 133–137, Oct 2009.

[4] 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 Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, Dec 2001.

[5] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168. Springer-Verlag, 2006.

[6] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352. USENIX, 2013.

[7] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58. ACM, 2010.

[8] W. Arthur, S. Madeka, R. Das, and T. Austin. Locking down insecure indirection with hardware-based control-data isolation. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 115–127. ACM, 2015.

[9] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308. ACM, 2013.

[10] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955. USENIX Association, 2014.

[11] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based cfi for ios. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 309–318. ACM, 2013.

[12] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012.

[13] A. A. Younis, Y. K. Malaiya, and I. Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 1–8, Jan 2014.

[14] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353. ACM, 2005.

[15] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277, May 2008.

ABOUT THE AUTHOR

Srivarsha Polnati completed her Bachelors of Technology in Computer Science and Engineering from Gandhi Institute Of Technology And Management (GITAM), Visakhapatnam, India in 2017. In 2017, she joined Department of Computer Science and Engineering at University of South Florida, Tampa to pursue her Master's degree. She worked under the supervision of Dr. Srinivas Katkoori and completed her thesis. She is always motivated to take up any new and challenging tasks.