APHID: Anomaly Processor in Hardware for Intrusion Detection

THESIS

Samuel Hart, Captain, USAF

AFIT/GCE/ENG/07-04

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCE/ENG/07-04

APHID:
Anomaly Processor in Hardware for Intrusion Detection

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Samuel Hart, B.S.C.E.

Captain, USAF

March 2007

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCE/ENG/07-04

APHID: Anomaly Processor in Hardware for Intrusion Detection

Samuel Hart, B.S.C.E.
Captain, USAF

Approved:

/signed/	08 Mar 2007
Maj Paul D. Williams, PhD (Chairman)	date
/signed/	08 Mar 2007
Dr. Barry E. Mullins (Member)	date
/signed/	08 Mar 2007
Dr. Yong C. Kim (Member)	date

AFIT/GCE/ENG/07-04

Abstract

The Anomaly Processor in Hardware for Intrusion Detection (APHID) is a step forward in the field of co-processing intrusion detection mechanism. By using small, fast hardware primitives APHID relieves the production CPU from the burden of security processing. These primitives are tightly coupled to the CPU giving them access to critical state information such as the current instruction(s) in execution, the next instruction, registers, and processor state information. By monitoring these hardware elements, APHID is able to determine when an anomalous action occurs within one clock cycle. Upon detection, APHID can force the processor into a corrective state, or a halted state, depending on the required response. APHID primitives also harden the production system against attacks such as Distribute Denial of Service attack and buffer overflow attacks. APHID is designed to be fast and agile, with the ability to create multiple monitors that switch in and out of monitoring with the context switches of the production processor to highly focused coverage over multiple devices and sections of code.

Acknowledgements

This thesis is dedicated to my family. I owe you all so much and I am grateful for every sacrifice you have made to allow this accomplishment.

Many thanks go out to all who helped with this thesis. To my advisor, for all of your extra hours dedicated to giving these ideas a sanity check. To our sponsor, AFRL/SNTA Anti Tamper Software Protection Initiative, thank you for the financial support. To my thesis committee, thank you for your time, inputs and suggestions.

Samuel Hart

Table of Contents

Page

Abstract iv

Acknowledgements v

List of Figures ix

List of Tables x

List of Abbreviations xi

I. Introduction 1
- 1.1 Proposed Solution 2
  - 1.1.1 Merits 2
  - 1.1.2 Costs 3
- 1.2 Document Organization 3

II. Related Research and Background Information 4
- 2.1 Definition of Intrusion 4
- 2.2 Classification of Intrusions 4
  - 2.2.1 Bugs. 4
  - 2.2.2 Flaws. 5
  - 2.2.3 Vulnerabilities. 5
  - 2.2.4 Design Vulnerabilities. 5
- 2.3 Detection Timeliness 7
  - 2.3.1 Measuring Time to Detection 7
- 2.4 Attack Surfaces 8
- 2.5 General Classification of Intrusion Detection Systems . . 8
  - 2.5.1 Network IDS 8
  - 2.5.2 Host IDS 9
  - 2.5.3 Host-based IDS Execution Classes 11
- 2.6 Related Research in IDS Co-processing 12
  - 2.6.1 Cryptographic Co-processing 12
- 2.7 CuPIDS 12
  - 2.7.1 CuPIDS Goals 13
  - 2.7.2 CuPIDS System Architecture 13
  - 2.7.3 Basic Capabilities 13
  - 2.7.4 Strengths and Weaknesses 15
- 2.8 Security Enhanced Chip Multiprocessor 16

Page

2.9 CoPilot . . . 16
2.10 Other Security Systems Using Methods Similar to APHID 17
2.10.1 Techniques for Monitoring Control Flow . . . 17
2.10.2 ESP: The Embedded Sensor Project . . . 18
2.10.3 Protecting the Stack with Hardware . . . 18
2.11 Classifying the Hardness of a Computing System . . . 19
2.12 Networking Background . . . 22
2.12.1 The Network Protocol Stack . . . 22
2.12.2 Internet Protocol Packets . . . 23
2.12.3 Denial of Service . . . 24
2.13 Reconfigurable Hardware . . . 26
2.13.1 Useful Hardware Primitives . . . 27
2.14 Chapter Summary . . . 27

III. APHID Model . . . 28
3.1 Problem Definition . . . 28
3.1.1 Research Hypothesis . . . 28
3.2 Solution Framework . . . 29
3.2.1 Research Goals . . . 29
3.3 Approach . . . 30
3.4 Device Driver Monitor . . . 31
3.4.1 Modular design . . . 31
3.4.2 Anomaly detection . . . 32
3.4.3 Justifications of Sizes . . . 32
3.4.4 Monitor Primitives . . . 34
3.4.5 Address Comparator . . . 36
3.4.6 State Machine . . . 37
3.5 Network Stack Monitor . . . 38
3.5.1 Limitation of the Network Stack Monitor . . . 40
3.5.2 Operation of the Network Stack Monitor . . . 40
3.5.3 Filter Design . . . 41
3.6 APHID System: Putting it All Together . . . 42
3.7 APHID Testing Model . . . 43
3.7.1 System Boundaries . . . 44
3.7.2 System Services . . . 44
3.7.3 Workload . . . 45
3.7.4 Performance Metrics . . . 46
3.7.5 Parameters . . . 46
3.7.6 Factors . . . 47
3.7.7 Evaluation Technique . . . 48
3.8 Chapter Summary . . . 48

Page

IV. APHID Implementation 50
4.1 Hardware and Software Platforms Used 50
4.2 APHID Network Stack Monitor Implementation 51
4.3 Integrating APHID with a Production Processor 52
4.3.1 Proposed Architecture 52
4.3.2 Successes 52
4.3.3 Roadblocks 54
4.3.4 Fallback Options 55
4.4 Intrusion Attacks on APHID 55
4.5 Distributed Denial of Service Attacks on APHID 55
4.6 Incomplete Implementation and its Effect on Testing . . 56

V. Results 58
5.1 Results of Tests on APHID Primitives 58
5.1.1 APHID Finite State Machine Simulation 58
5.1.2 APHID Address Comparator Discussion 60
5.1.3 Network Filter Simulation Discussion 60
5.2 The Benefits of Hardware Content Addressable Memory 60
5.3 Comparison of the APHID Network Filter to Existing Research 63
5.4 Total System Integration 64
5.4.1 In System Performance Estimate 64

VI. Concluding Remarks and Future Research 66
6.1 Concluding Remarks 66
6.2 Contributions 66
6.3 Future Research Opportunities 67

Appendix A. APHID Primitives in VHDL 68
A.1 APHID State Machine Primitive 68

Appendix B. VHDL MIPS Processor 72
B.1 Mips.vhd 72
B.2 MIPS_Piped.vhd 79
B.3 control.vhd 89
B.4 Ifetch.vhd 91
B.5 Idecode.vhd 94
B.6 Execute.vhd 97
B.7 Dmemory.vhd 100
B.8 Control.vhd 102
B.9 Decode.vhd 104

Bibliography 106

List of Figures

Figure		Page
2.1.	CuPIDS: Using SMP Architectures for Security Policy Compliance Monitoring	14
2.2.	A Buffer Overflow on the Stack	19
2.3.	Levels of Security	21
2.4.	The Network Protocol Stack and Packet Encapsulation	22
2.5.	A Distributed Denial Of Service Attack	25
3.1.	Monitor Cache	33
3.2.	APHID Flowchart	35
3.3.	Device Driver Monitor Architecture	36
3.4.	APHID State Machine	38
3.5.	Deploying APHID Network Stack Monitor	41
3.6.	Network Stack Monitor Flowchart	43
3.7.	The APHID System	44
3.8.	The System Under Test	45
4.1.	The APHID Network Filter	52
4.2.	The APHID Implementation on a RISC Processor	53
4.3.	The APHID Network Filter	56
5.1.	Results from Simulation of APHID State Machine	60

List of Tables

Table | Page

3.1. The Factors 47

List of Abbreviations

Abbreviation		Page
APHID	Anomaly Processor in Hardware for Intrusion Detection .	2
PPU	Production Processing Unit	2
API	Application Programming Interface	5
IDS	Intrusion Detection System	8
NIC	network interface card	9
VMM	Virtual Machine Monitor	11
OS	Operating System	11
VM	Virtual Machine	11
TPM	Trusted Platform Module	12
CuPIDS	Co-Processing Intrusion Detection System	12
SMP	Symmetric Multiprocessing	12
SPCM	Security Policy Compliance Monitoring	12
CPP	CuPIDS Production Process	13
CSP	CuPIDS Shadow Process	13
SECM	Security Enhanced Chip Multiprocessor	16
ESP	Embedded Sensor Project	18
SRAS	Secure Return Address Stack	19
TCP	Transmission control protocol	23
UDP	User Datagram Protocol	23
IP	Internet Protocol	23
IPv6	Internet Protocol version 6	23
DoS	Denial of Service	24
DDoS	Distributed Denial of Service	24
FPGA	Field Programmable Gate Array	26
LUT	Look Up Table	26

Abbreviation		Page
HDL	Hardware Description Language	26
CAM	Content Addressable Memory	27
KLA	Known Legitimate Actions	32
RTL	Register Transfer Logic	37
FSM	Finite State Machine	37

APHID:

Anomaly Processor in Hardware for Intrusion Detection

I. Introduction

Modern operating systems are becoming increasingly complex. Studies show that the average number of bugs per 1000 lines of code is between 1 and 16 [7]. Commodity operating systems (Linux, Unix, and Microsoft Windows XP) contain between 2 and 30 million lines of code, resulting in a conservative estimate of 15 thousand bugs in a typical operating system [42, 43]. One author claims that a medium-sized corporation could have as many as 5 million remotely exploitable security vulnerabilities when the problem is compounded over the corporation's network (approximately 30,000 nodes) [23]. Furthermore, device drivers make up a large portion of the operating systems and are often written by third parties. These drivers operate at elevated privilege levels (kernel mode) and have access to critical data structures and memory spaces that the operating system must use [38, 43]. These third party programs may not be developed with the same quality assurance and testing that is afforded by the developers of the operating system kernel. Even efforts in the area of driver certification are not enough because certification does not imply that the program is bug free, only that it meets certain minimum criteria, certified code may still contain software vulnerabilities and exploits [27]!

The spread of broadband to the masses compounds the problem of high bug count. While most users in the fields of computer science and engineering are concerned with security and take at least the minimal security precautions on the systems they use, the masses at large are notoriously bad at securing their systems. Home users have unsecured wireless routers, unpatched operating systems, and disabled

firewalls. Often useability trumps security. As a result of the large number of unsecured machines, distributed denial of service attacks become feasible. All an attacker needs to do is gain access to a sufficient number of machines (and turn them into zombies). The average home user would never know of this intrusion and can be an unwitting pawn in the schemes of attackers. Recent bot-net based denial of service attacks against the 13 root domain name servers exemplify the increasing severity and sophistication of the attacks [18].

1.1 Proposed Solution

Thesis Statement: *Intrusion Detection through the use of dedicated hardware running anomaly detection schemes at low levels (i.e. in hardware) will result in significant improvements in response time and reliability over software intrusion detection systems. Applying hardware primitives to the problem of denial of service attacks can increase a system's overall resistance to failure.*

The Anomaly Processor in Hardware for Intrusion Detection (APHID) is a method for creating a trusted computing environment while maintaining system performance. We accomplish this using hardware primitives and dedicated co-processing rather than a software only solution. The result is increased security with minimal performance impact as compared to a host-based intrusion detection system running as a separate software task on the main system production processing units (PPUs). Furthermore, the detection of intrusions can be accomplished in real-time, as the event occurs, rather than after some passage of time, giving rise to the potential for repair of intrusions (or even stopping the intrusion before damage occurs).

1.1.1 Merits. APHID hardware primitives offer the opportunity to capture bad events as they happen. Doing so can allow one to perform security checks at a very low level (resulting in high fidelity compliance checking) and at hardware speeds. Offloading the security monitoring overhead to dedicated processing allows the production system to operate without interruption when not under attack and to operate with minimal disruption when being actively attacked. Using APHID primitives also allows the administrator to harden the system against various forms

of attacks ranging from buffer overflows on device drivers to denial of service attacks on the network interface.

1.1.2 Costs. The benefits from APHID are not without their costs. APHID requires modification of commodity hardware platforms. Currently this is done with reconfigurable hardware, but for true performance benefits the primitives will need to be placed into the fabric of modern processors. Furthermore, the specific detection algorithms used by APHID are application specific and require the programmer to be intimately familiar with the devices and libraries they are protecting. Improper or naive implementations could result in a substantial performance penalty or an unstable system.

1.2 Document Organization

The remainder of this document is broken into five chapters. Chapter II discusses the past and current work in intrusion detection as well as important background material that the reader may find useful for understanding concepts presented later in the document. Chapter III presents the theoretical APHID architecture and testing models. Where possible, implementation specific details are omitted to keep the model as flexible as possible. Chapter IV presents our implementation of APHID and discusses the process taken to arrive at that implementation. Chapter V presents the results of the APHID implementation. Some of the results are empirical, gathered from simulation, and some are more theoretical, based on some simplifying assumptions that abstract away more complex issues to demonstrate the power of APHID where actual testing has not been possible to date. Chapter VI concludes the document with discussions about APHID and future research opportunities.

II. Related Research and Background Information

This chapter outlines research in the field of computer system security (specifically intrusion detection) and touches on some necessary background and fundamentals which may assist the reader with understanding topics discussed in later chapters. It is assumed that the reader has a general background in computers and information systems. This section provides a basis for further understanding of what intrusion detection involves and the work that has been done to address the problem.

2.1 Definition of Intrusion

This research is in the area of intrusion detection. For the sake of clarity we must define the term intrusion to remove ambiguity. An intrusion can be defined as: "An improper or unauthorized use of computer resources, with or without malicious intent." This definition encompasses the aspects of insider threat, and external attacks [5, 24, 45]. While we are concerned with insider threat activities, the bulk of this research deals with general protection of code while in execution. We do not address access policies in terms of who is allowed to be on the machine. We are specifically concerned with illegitimate code execution through the exploitation of vulnerabilities in software. Therefore, we can tighten the definition of an intrusion to: "Illegitimate execution of code on a system by exploiting software vulnerabilities." For the remainder of this document, the word *intrusion* is used with the more restricted definition, unless otherwise noted.

2.2 Classification of Intrusions

Intrusions take advantage of system weaknesses. Hoglund and McGraw place use the following categories to classify weaknesses: Bugs, Flaws, Vulnerabilities, and Design Vulnerabilities [23]. This classification is in order of increasing complexity and increasing effort for repair.

2.2.1 Bugs. Bugs are code level issues. Incorrect use of a function call such as `strcpy()` in C can result in a buffer overflow exploit if array bounds are

not checked. Unfortunately, this type of bug is one of the most common sources of system vulnerabilities. A common exploit can result in arbitrary code running at the privilege level of the process with the buffer overflow bug, compromising the system even to the point of being remotely controlled by an intruder. Most bugs can be resolved using simple scanning of software. Good programming practices reduce or eliminate these.

2.2.2 Flaws. Flaws run a deeper risk. A flaw is a problem with the implementation of a section of software, involving a more complex interaction than misuse of a system call or improper bounds checking. Flaws manifest from poor design decisions. A flaw may or may not be exploitable.

2.2.3 Vulnerabilities. Vulnerabilities are problems that can be exploited by an attacker. A bug or a flaw can result in a vulnerability. Vulnerabilities have to do with design decisions and access. Particular points of interest are interfaces between software modules (such as device drivers and system libraries). Vulnerabilities can either be directly exploited or used in combinations for gaining access to the system. A buffer overflow is a direct exploit of a bug. By running past the bounds of an array (buffer) the attacker can inject malicious code into the execution path, or can change the control flow of the program resulting in an intrusion. Examples of combination attacks involve exploitation of timing or interface flaws. Some error handling states can leave a system in an insecure state if a exception occurs, or certain application programming interface (API) calls can leave vulnerabilities exposed.

2.2.4 Design Vulnerabilities. Detecting a design vulnerability is more of an art than a science. Design vulnerabilities can take the form of improper configurations which expose data channels; improper error handling (as mentioned above), incorrect use of access control mechanisms (weak passwords, or unencrypted password files, for example), or complex interactions among distributed systems that "leak" information through unintended sources (message types and traffic analysis).

It is possible to classify intrusions according to the type of vulnerability exploited, however it is most common to classify them according to the type of activity performed. The following is a list of common classifications of intrusions. It is not exhaustive or complete. The information for this list is gathered from various sources [11, 13, 23].

Virus: A virus is malicious software that attaches itself to other software within the system, lacking the ability to self propagate outside the system.

Worm: A worm is software that performs malicious actions and also has the ability to propagate to other systems.

Trojans: A trojan is malicious software embedded in an innocuous package that can open security holes to create new vulnerabilities. Trojans can behave as a worm or as a virus.

Rootkit: Rootkits are a more recent addition to the list of vulnerabilities. The design of a malicous rootkit is particularly insidious in that it may become embedded in the operating system in such a manner that it can intercept and change any system messages. The result is that the rootkit maintains root level access for the intruder, but hides all such activities from the operating system. They can operate by several means. A masters thesis research effort by Nerenberg classifies rootkits according to their various strategies and implementations [32].

Timing Attacks: Mentioned above, timing attacks can take advantage of insecure system states achieved by sequences of system or function calls. Additionally, timing based attacks can take advantage of knowledge of scan intervals (by the IDS) or of other similar vulnerabilities in order to avoid detection and compromise a system.

Sequenced Attacks: Sequenced attacks involve using design vulnerabilities to build up an intrusion. These usually require extensive knowledge of the system, or some additional intrusion access through virus, worm, or trojan activities. The sequenced attacks often result from unintended consequences of library calls on the system state.

2.3 Detection Timeliness

One goal of this research is to achieve real-time detection using our IDS. To that end, it is necessary to explicitly define some terms for the remainder of the research. Rather than reinventing the wheel, we refer to definitions from Kuperman's Ph.D. dissertation [24].

Real-time: Kuperman defines real-time as: Detection of a bad event, $\boldsymbol{b}$, takes place while the system is operating, and before any event dependent upon $\boldsymbol{b}$ can occur.

Near Real-time: Near real-time is defined as: Bad event $\boldsymbol{b}$ is detected within some finite (and small) time δ of $\boldsymbol{b}$'s occurrence.

Periodic: A scan by the security system at a set interval p. Detection must take place within a worst case maximum of $2 * \boldsymbol{p}$ for the detection scheme to qualify for this name. This worst case occurs when b occurs immediately after the previous scan has finished and is not detected by the current scan until immediately before the next scheduled scan. If this were not the case, there would be a bottleneck in the detection system and the system could scan and process enough events to keep up with the incoming event stream.

Retrospective: An offline review of events to detect the bad events. This review can be as simple as a person reading the logs, to as complex as a full forensic analysis of a system.

2.3.1 Measuring Time to Detection. For the purposes of this research time is measured as the number of operations (defined as instructions at the assembly code level) executed from the time bad event b begins execution until b is detected. The reasons for this choice are twofold. First, on systems with similar instruction set architectures (such as Power PC and MIPS) one can make an "apples to apples" comparison of detection times, regardless of clocks rate. Likewise, on machines with differing instruction set architectures (such as MIPS and Intel x86), there is generally a

sense of the average ratio of instruction counts acquired from compiling the same code on both machines. Using the instruction counts and average number of instructions executed per clock cycle on each machine allows for a rough comparison between the two machines. The second reason is related to the overall effects of an intrusion. Logically, more instructions executed, from the time b occurs until it is detected, offers an intruder a greater opportunity to cause damage. The metric executed instructions between event occurrence and event detection gives a sense of the extent of damage that can be accomplished. Furthermore, intrusions involving relatively few bytes of executed code (like the SQL Slammer attack with a size of 376 bytes [19]) *could* execute and remove traces of the intrusion if the time to detection is large.

2.4 Attack Surfaces

Attack surfaces, defined by Manadhata and Wing, is the concept of the size of vulnerability presented to the attacker [28]. For instance, an entire operating system, with all of the services and processes running, presents an extremely large attack surface. It is not reasonable to expect a single monitoring unit to effectively protect this entire surface. By choosing a sufficiently small attack surface it is possible to create an monitor capable of protecting the code section in real time.

2.5 General Classification of Intrusion Detection Systems

An intrusion detection system (IDS) is generally classified as either a network IDS or a host IDS. Our research, while mainly focused on host based systems, has potential applications as a blended approach, incorporating both network and host-based IDS concepts.

2.5.1 Network IDS. A network IDS examines network traffic at the packet level and can be a stand-alone system which monitors all traffic on a specific network segment, or an internetworked set of intrusion detectors that share a private network (either physical, or virtual through a virtual private network) [8]. A network IDS

has visibility only of the traffic on the network. In a typical setup, the network IDS operates its network interface card (NIC) in promiscuous mode, capturing and examining all traffic on the network where it sits. This may require special configuration of the router or switch connected to the network IDS where the switch forwards all traffic to the IDS as well as to the destination. A prime example of a network IDS is the open source, self proclaimed de facto standard for intrusion detection/prevention, Snort [41].

2.5.2 Host IDS. Like the integrated air defense system, a secure enterprise system should not rely on only one layer of defense, rather there should be multiple rings (layers) of protection. A network IDS is one component of such a plan. A Host IDS is a component that can be used to add an additional layer of security. Host IDSs monitor a particular single computing unit (what we would think of as a single server or desktop computer). The host IDS has access to internal information available to only that host, but has limited global awareness. The host IDS is used to monitor the processes and user activities on the single machine and can only monitor network traffic specifically destined for that machine. Dorothy Denning [16] provided the framework for much of the current work in general purpose IDS designs, and most host intrusion detection systems can be classified as derivatives of her work.

In the class of host based ID systems, three main approaches have been taken. The first uses anomaly detection (which we lean toward in our research), the second uses signature based detection, and the third uses specification based detection. We refer the reader to Axelsson's paper [3] for a thorough taxonomy of ID systems and the approaches taken, with the note that he specifies only two IDS classes, anomaly detection and signature detection. The third class, specification based detection, was noted by Williams in [45] as a reflection of the state of the art in intrusion detection.

Anomaly Detection: Anomaly detection works on the principle that a system knows what normal is. This can occur through either self learning, or through programming. Using this sense of normal, the IDS can ascertain when something

outside of those parameters occurs. Anomaly detectors can suffer from higher false positive rates, so a tradeoff must be made by setting thresholds of acceptable deviation from normal. Raising the thresholds can reduce the false positive rates (incorrectly identified intrusions), but can have an inverse effect on the rate of missed detections.

Signature-based Detection: Signature based systems use knowledge of previously encountered attacks to create a signature for a specific attack or class of attacks. This signature allows the IDS to search through events and detection occurs when a signature matches an event. The result is a highly reliable detector for events that have signatures. However, the detector can be easily fooled by even slight changes to the event, and will allow unknown intrusion events, for which there is no signature, unrestricted, undetected access to the system [45].

Specification-based Detection: Growing interest (and concern) in the arena of computer security is resulting in a change of perspective towards specification based systems. The concept of specification based systems is not new (especially in terms of the pace of computing system progress). Once thought to be too expensive (too hard to implement) and too restrictive, the realization of the need for tighter security models is bringing this concept to the forefront [10]. The Bell-Lapadula model [9] is one security model that is commonly applied to these type of systems. A specification-based system explicitly defines a condition of security for the system. Any state that does not match a state in this security specification is flagged or raises an alarm.

Hybrids: There are examples of IDSs that apply more than just one detection approach in an effort to improve reliability and accuracy. As mentioned before, a detection on a signature based system has a high probability of being correct, but it is trivial to manipulate many intrusions so that the signature is rendered invalid, while the attack remains a threat. Applying anomaly detection (or specification-based detection) in concert with a signature-based detector can improve the reliability and accuracy of the system.

2.5.3 Host-based IDS Execution Classes. We can classify host-based intrusion detectors by the manner in which they operate.

Interleaved: This IDS runs as a separate process, interleaved with other processes running on a multitasking machine and it relies on the operating system to schedule the IDS to run frequently enough to capture any intrusions. The best detection timeliness any interleaved IDS can hope for is *near real-time*, but the more likely case is a *periodic* detection timeliness (see Section 2.3 for the definition of near real-time).

Interposed: An interposed IDS works by adding a software layer to the system so that all system calls activate the IDS. The result is a finer granularity of detection, with the (potentially high) cost of added overhead to every system call.

Co-processing: A coprocessing based IDS offloads the overhead of running security tasks from the PPU to the security co-processor. This has the added benefit of providing parallel processing with the potential for *real-time* detection (see Section 2.3 for the definition of real-time). One hindrance to this approach is that the co-processor may not have necessary access to the PPU data structures and hardware without modification of the computer system architecture and operating system. This research is focused on defining new hardware primitives to advance co-processing IDS capabilities.

Virtual Machine Monitors: A virtual machine monitor (VMM) based IDS works by running the IDS in a software layer below the operating system (OS. Doing this allows the IDS to have access to virtual machine (VM resources and the IDS can perform detection on each guest OS running on that system. This method appears to have some merit and should prove to be very useful with the increasing emphasis on visualization technology [29].

2.6 Related Research in IDS Co-processing

This research is focused on using co-processing to implement a host-based IDS. The following sections highlight the main points of existing research in IDS co-processing. Co-processing systems stand in contrast to uniprocessor intrusion detection systems in that they are able to process in parallel with the production execution unit, allowing for a finer granularity of detection.

2.6.1 Cryptographic Co-processing.

While not a true IDS, a cryptographic co-processor provides the system with services that increase overall system security by encrypting data streams between functional units on the system. The resulting secured busses and channels allow for data integrity checking. Cryptographic co-processors do not necessarily prevent an intrusion, but they do minimize the attack surface presented to that intrusion source by securing the channels. In the event of an intrusion, the attacker is presented with a greater challenge in capturing meaningful data from the system. The trusted platform module used in Intel's LaGrande technology (TPM) is a mainstream example of a cryptographic co-processor [4].

2.7 CuPIDS

The co-processing intrusion detection system (CuPIDS), introduced by Williams, works on symmetric multiprocessing (SMP) systems. Most SMP systems attempt to balance the workload by spreading processes evenly across the processors. CuPIDS differers from this approach by making a conscious tradeoff between security and performance and dedicating one of the processors in the SMP architecture to security related processing tasks. This difference allows CuPIDS to perform parallel monitoring in real time (see Kuperman's definition in Section 2.3) [24, 45]. The parallel monitoring performed by CuPIDS supports the broader class of security policy compliance monitoring (SPCM) which includes intrusion detection as well as a number of related areas including error detection and computer forensics [10].

2.7.1 CuPIDS Goals. The goals of CuPIDS are:

- Harness the power of SMP systems to perform security monitoring.
- Allocate one processor to perform parallel monitoring at higher fidelity than possible in a uniprocessor intrusion detection system.
- Use the opportunities afforded by parallel monitoring to perform actions not available to a uniprocessor IDS, such as real time monitoring and self healing.

2.7.2 CuPIDS System Architecture. The foundation for the CuPIDS architecture is monitoring a production process by running the monitoring process in parallel (on a separate processor) as the production process executes. Designed around a shared resource, general purpose SMP architecture, CuPIDS separates protected processes into two components: A CuPIDS production process (CPP) and a CuPIDS shadow process (CSP). CuPIDS is event driven. The stream of events that CuPIDS operates on is generated by the CPP and used by the CSP. The shared memory resources are not shared symmetrically. The CSP is given insight into the memory space of the CPP but the CPP is not given any access to the memory space of the CSP. This allows the shadow process (CSP) to monitor the production process non-intrusively, while it executes. The CSP is also able to control the activities of the CPP through a control stream.

2.7.3 Basic Capabilities. Several characteristics define the capabilities and characteristics of CuPIDS. From Williams' dissertation [45], they are:

1. Hardware resources (a processor in this case) are dedicated solely to security tasks (See Figure 2.1).
2. All of the state of the monitored process (the CPP) is available to the monitoring process (the CSP).

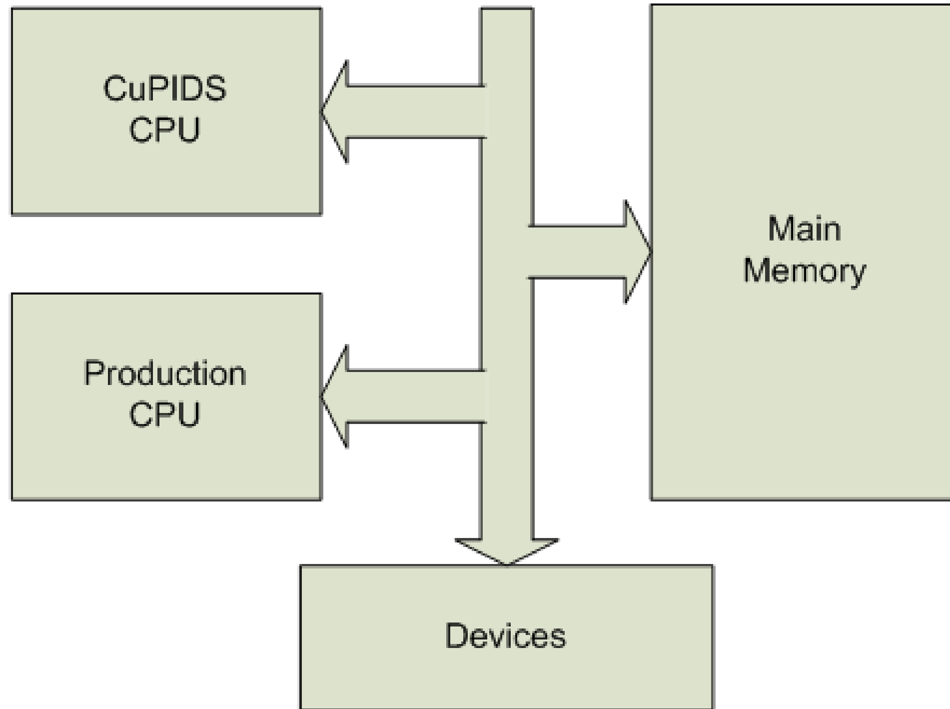

Figure 2.1: CuPIDS: Using SMP Architectures for Security Policy Compliance Monitoring

3. Primary detection tasks are specification-based. That is, explicit knowledge of how the protected application is intended to behave is used to detect incorrect or illegitimate behavior.

4. Security monitoring is done in parallel (concurrently) with the execution of the monitored task.

5. Attacks may be immediately acted upon because of the parallel monitoring. Any information or analysis gathered by the monitor can be used to halt the attack an keep the system from entering a damaged state, in contrast to an IDS running on the uniprocessor model (interposed or interleaved) where the attack may disable the IDS (or hide from it) before it has a chance to execute.

6. The parallel monitoring is not instruction-by-instruction. CuPIDS traces the execution path in parallel, keeping the appropriate monitor running with its production code.

7. CuPIDS is event based. The execution state of the monitored process is transmitted to the monitoring process via an event stream. The event stream can be automatically generated from some sources, and others require explicit instantiation by the programmer.
8. The monitoring unit is synchronized with the protected application by the event flows.
9. Monitoring can be performed asynchronously from the protected application, further complicating the task of the attacker.
10. CuPIDS resides and operates inside the host OS, giving the monitoring task complete visibility into the state of the OS and full control of a monitored process. This is a much higher level visibility than that provided by other approaches (virtual machine monitors and cryptographic processors).
11. CuPIDS can be implemented on commercially available hardware using modified commercially available software.

2.7.4 Strengths and Weaknesses. CuPIDS has strengths and weaknesses as pointed out by Williams.

Strengths: CuPIDS can detect intrusions in real time (again using Kuperman's definition). This is the main advantage of CuPIDS. Since it monitors in parallel, CuPIDS can also observe activity internal to the monitored process that a uniprocessor IDS simply cannot observe because it cannot run in parallel. The result is a higher fidelity detection model, using internal control flow events, programmer defined events, and interactions with external entities (e.g., libraries, drivers, system calls). Another strength of CuPIDS is that it removes the burden of security processing tasks from the production process. As security takes a larger role, the cost of security processing will also expand. CuPIDS uses some hardware that could have been used for production processing, but introduces minimal overhead on the production processor(s) it does not use.

Weaknesses: CuPIDS exists inside the production operating system. Because of this, its communications and processes are vulnerable if the host OS is compromised. Messages are passed through the kernel, and a compromised kernel could alter or suppress those messages, resulting in a broken event stream, and rendering CuPIDS inoperative. The tradeoff between visibility of processing state and protection of the security system is the source of this weakness.

2.8 Security Enhanced Chip Multiprocessor

The Security Enhanced Chip Multiprocessor (SECM) is another research effort in the field of parallel co-processing intrusion detection. Like CuPIDS, SECM uses commodity hardware and software (with modifications) to perform the monitoring tasks. SECM differs from CuPIDS in that it uses an asymmetric multiprocessing approach where the production processor uses a full blown kernel, with limited access to "riskier" system calls and the security processor runs a minimal kernel for higher performance and smaller attack surface. The security processor is given higher access to machine state and also access to the production kernel state for monitoring purpose, but the production kernel does not have the same privileges. State information about the production OS is gathered from the memory bus and shared memory resources. SECM, by having elevated privileges over the production OS is able to detect rootkit attacks on that OS (because it has visibility to layers "beneath" the OS where a rootkit would be embedded). Furthermore, given the elevated privileges, the security OS can dump a compromised production OS and reload a clean version of the production kernel in the event that an intrusion is detected. This allows the production OS to maintain some resemblance of robustness to attacks.

2.9 CoPilot

CoPilot approaches the co-processing intrusion detection problem from a different perspective [36]. Rather than dedicate a full CPU to the task of monitoring, the CoPilot system places a co-processor on the PCI bus. Here, the co-processor works

much like a graphics card works for acceleration of graphics libraries. Because the processor sits on the PCI bus, it can only gather system state information through that bus and it cannot directly impose control over the system. Residing on the main memory bus allows co-pilot to do a 1 to 1 mapping of memory addresses to memory space but limits its access to hardwired (non virtualized) kernel pages that do not move. CoPilot works by monitoring these memory locations for changes using anomaly detection. Due to overhead issues, CoPilot monitors on a periodic basis (see Section 2.3 for definition). Because of this, CoPilot cannot achieve real-time or near real-time detection. The idea of using a peripheral card holds promise despite the aforementioned timeliness and memory visibility issues.

2.10 Other Security Systems Using Methods Similar to APHID

This section briefly describes other security systems (not necessarily IDSs) that behave in ways similar to the desired behavior of APHID. These systems are related in that they either contain an idea similar to APHID concepts or use similar methods on different problems.

2.10.1 Techniques for Monitoring Control Flow. Zhang in [49] and Arora in [2] present independent methods for checking control flow at various levels of fidelity. Zhang's method modifies the hardware of a processor to allow for control flow checking at the instruction level. The checking is done by running anomaly detection routines in hardware. The anomaly detector is programmed statically with compile time data regarding branching and then the program is trained in a "sandbox", or a known secure environment. Doing this allows the hardware to record common branching and to gather a sense of normal behavior.

Arora's team presents three methods of integrity checking in an embedded environment. Her work provides: Intra-procedural control flow checking, where branches within a procedure are monitored for illegitimate activity, Inter-procedural control flow checking, where branches from one procedure to another are monitored for cor-

rectness, and finally, instruction stream integrity using hashing algorithms. Procedures are hashed at compile time and the hardware then hashes the procedure at run-time to check against any modifications that may have occurred.

2.10.2 ESP: The Embedded Sensor Project. The Embedded Sensor Project (ESP) is actually a software extension that embeds specific calls in sections of code (it is an interposing IDS in a sense) [48]. However, ESP aims at making the sensors small and fast, and optimized for the task of protecting a specific section of code. Synergistic coupling of the embedded sensors can allow ESP to provide intrusion detection at varying levels from the application level to the operating system level, even to the network level if deployed on several machines and given the ability to communicate with one another.

2.10.3 Protecting the Stack with Hardware. Buffer overflows continue to be a steady source of vulnerabilities, even though the solution to the problem is well known. Secure programming practices should eliminate this source of intrusions. Despite this, CERT continues to report vulnerabilities related to the technique of "stack smashing" on a fairly routine basis [13]. The majority of attacks involve the attacker injecting specially crafted code into a buffer with unchecked bounds. The attacker will write the bytecodes corresponding to a NOP (no operation in assembly code) creating what is known as a NOP sled. The attacker then replaces the return address on the stack with an address that is likely to be in the range of the NOP sled and the processor will then execute the NOPs until it reaches the malicious code at the end. If the buffer overflow vulnerability existed in code with elevated privileges the resulting intrusion can grant root level access to the attacker. Figure 2.2 shows a simple example of a buffer overflow in action. In this example, the buffer is in the program stack. The process reads input from the data source (user input, for example), the input exceeds the space allocated to the buffer and important stack data are overwritten. Here we see that the malicious data (from the array overflow) writes over the return address with the calling address of the malicious code. When

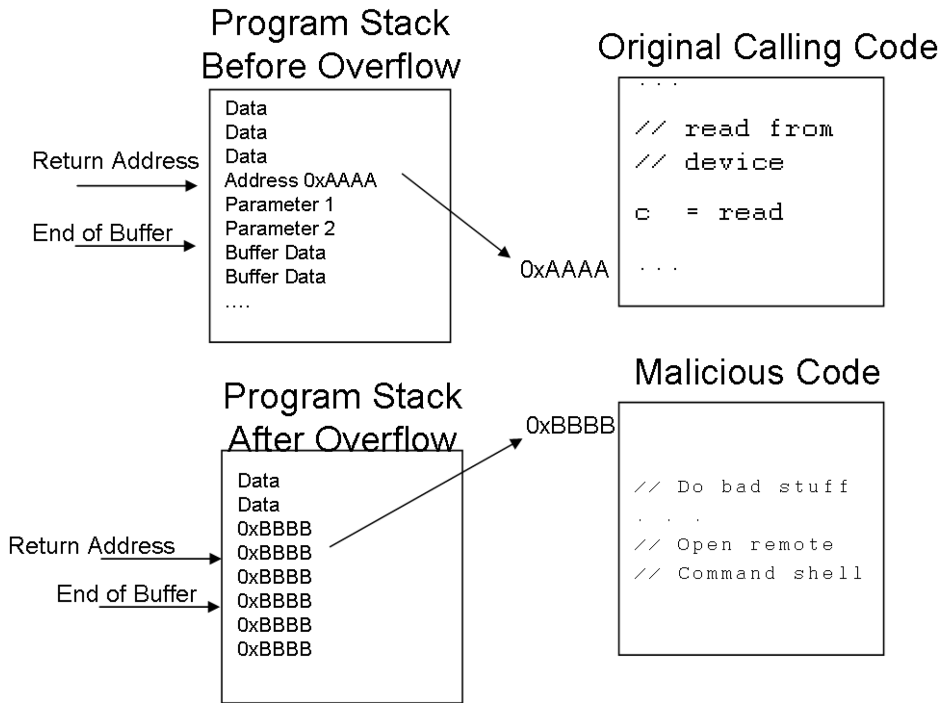

Figure 2.2: A Buffer Overflow on the Stack

the process executes the call and returns to what it thinks is the code section that made the original call, it will actually begin execution in the malicious code section where bad things (such as remote root-level access) can occur.

Several methods have been introduced to cope with this epidemic. Lee *et al.* propose a secure return address stack SRAS, which imposes additional hardware checking to stack calls. If SRAS detects a modification to the stack, then a flag is raised [26]. Smashguard employs similar techniques, but handles the problem of deep stack returns more elegantly than Lee [34].

2.11 Classifying the Hardness of a Computing System

It becomes necessary to classify computing systems according to their level of protection, or hardness. The term hardness is chosen to frame the concept in terms of military physical protection. For example, sandbags provide a level of protection

against light munitions, thereby improving the security of anything behind the sandbags. Further hardening could involve bunkers buried deep underground, with 12 foot thick blast doors. Obviously, there is a tradeoff between the cost of hardening and the level of protection required. It does not make sense to build a subterranean bunker to protect a units meal rations for the day. A truck or small building will provide adequate security. Likewise, it is not prudent to place critical command and control nodes in a canvas tent, that is what the hardened bunkers are for.

We use this analogy to make a point. The act of hardening a system imposes a cost on the system. The cost involves complexity (which can mean higher financial burden for designers) and usability/productivity/performance (which can result in higher financial burden for the users). There is an direct relationship between the cost of the system and the security of the system. This implies that there is a need to harden the system to the appropriate level (as defined by the purpose of the system). In keeping with the thesis statement in Chapter I, we aim to reduce the cost of the security mechanism by pushing the security aspects to a hardware monitor on a system, thereby increasing the security level of the system while minimizing the costs associated with that new security level.

We classify the hardness of a system using a ring topology. Our classification system has six levels numbered 0 to 5 (Figure 2.3).

Ring 0: Perfect Security. A useable system corresponding to this ring does not exist. A system with this level of security is a closed system. No information flows out of the system, nor can it interact with its environment. Unfortunately, this system is useless (notwithstanding its excellent properties as a paper weight).

Ring 1: Deeply Hardened. In the military analogy, this is the secure, hardened, underground facility. There is limited direct contact with systems in the outside world. Only trusted entities are allowed access (physical or remote). It can be compromised, but this requires an trusted insider with access to sabotage the system.

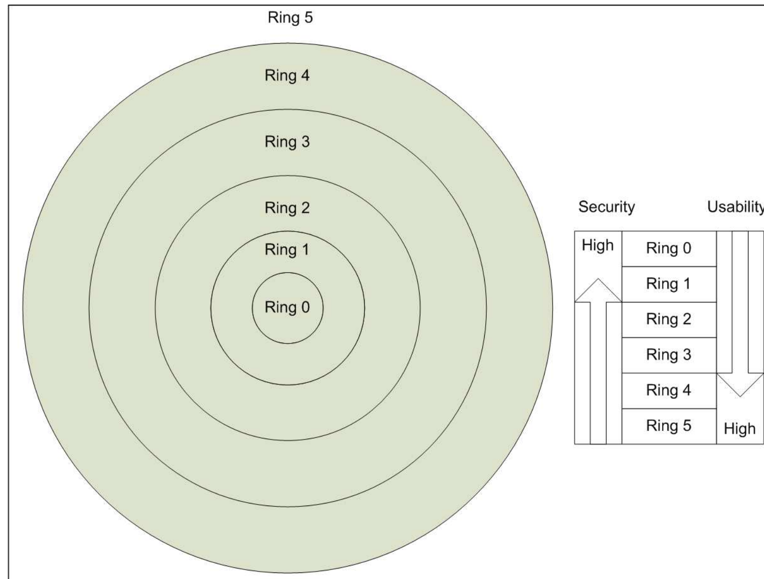

Figure 2.3: Levels of Security

This is the type of system you *want* your bank to have in place for managing deposits and withdrawals from your account.

Ring 2: Hardened. The military analogy is a concrete building with access security and various levels of protection. From the information technology perspective this is like an enterprise server with physical access policies, running behind a firewall, up to date patching, and some type of intrusion detection systems in place.

Ring 3: Heavily Armored. This can be analogous to military heavy armor, like tanks and other vehicles. They are resistant to most attacks from similar platforms. In terms of computing systems we would classify this as a patched, firewalled, small business class network without automated intrusion detection capabilities, but with all appropriate traffic logging and security software (such as virus scanning and strong password login requirements) in place.

Ring 4: Lightly Armored. In the analogy we compare this level to a lightly armored vehicle. It can withstand many attack types, but cannot handle a direct attack from larger munitions and focused attacks by multiple sources. This

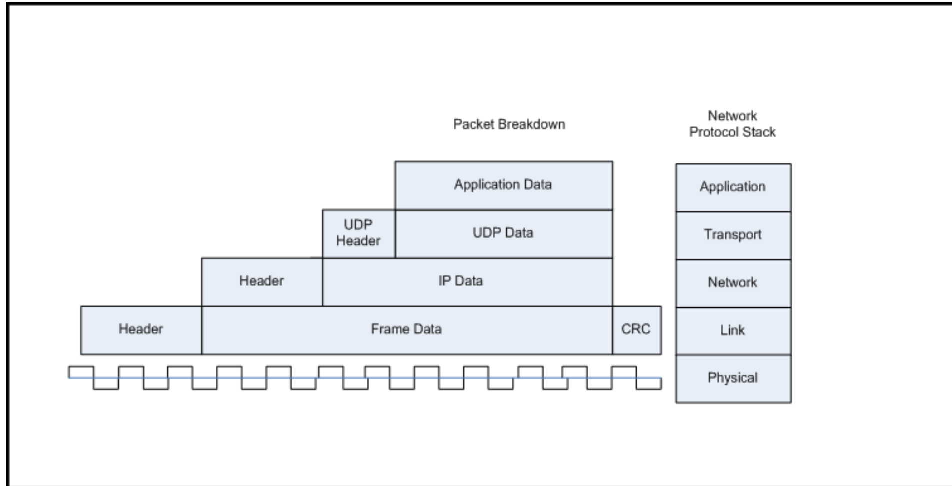

Figure 2.4: The Network Protocol Stack and Packet Encapsulation

would be the equivalent of a home user system, with up to date security patches and perhaps a personal firewall. Here the level of protection is good for very low cost.

Ring 5: Unsecured. Consider this as an unarmed soldier in an enemy camp. This is what you get when you install an operating system and without patching connect to the internet via broadband. Almost immediately, the system can become compromised. Moving from Ring 5 to Ring 4 is easy and has an excellent cost to benefit ratio.

2.12 Networking Background

Figure 2.4 shows both the protocol layering and the datagram packaging concepts discussed in the following sections.

2.12.1 The Network Protocol Stack. The Network Protocol stack is the Internet's layered service protocol stack that facilitates communication over networks. Using protocol layering allows different technologies to communicate by abstracting away complexity of the system. The Internet protocol stack has five layers: Application, Transport, Network, Link and Physical. Each layer provides a service to the layer directly above it in the stack. The following list contains examples of technologies that provide services in each layer, in order from top to bottom [25]:

- Application Layer: End user programs , such as instant messengers, maintain connections at this layer.
- Transport Layer: TCP (transmission control protocol) and UDP (user datagram protocol) provide datagram services at the transport layer level. TCP provides an end-to-end connection-oriented service with delivery guarantees, while UDP provides a connectionless service with no delivery guarantee. Both modes are used extensively and have merits depending on the particular application.
- Network Layer: The Internet Protocol (IP) provides the source and destination routing services to transport layer packets in the network layer.
- Link Layer: In the link layer, IP packets are provided with the point to point delivery services. Packets at this level are called frames. Ethernet, Wi-Fi, and the point-to-point protocol are examples of link layers.
- Physical Layer: The physical layer delivers bits and provides the mechanism for delivery of frames. This is where the hardware resides. We can think of this layers as the wires, or the radios involved in transmission and receipt of data.

2.12.2 Internet Protocol Packets. IP packets are the workhorses of the Internet. An IP Packet is wrapped in a frame and sent across links. Upon arriving at a host, the IP Packet is extracted from the frame and examined for destination address. If the host doing the examination is the destination, then the packet continues up to the transport layer. Otherwise the packet is ignored or forwarded, depending on the role of the receiving host. This research is primarily concerned with IPv6 (IP version 6) packets because they are (slowly) replacing IP version 4 packets as the new standard [25]. This choice provides several advantages. First, the amount of work that must be done to parse the packets is reduced because the specification removes variability of the header length. This was a problem in IPv4 where the header could have optional data resulting in more complex parsing algorithms [14]. Beyond that, the IPv6 packets also do not allow fragmented packets, which simplifies design choices in this research. As a result, the hardware that works with the IP

packets must handle larger addresses (128 bit versus 32 bit), and address ranges, than it would have handled with IPv4 packet specifications [15].

2.12.3 Denial of Service. Denial of Service (DoS) attacks are of particular importance to this research. Current events illustrate that this form of attack is still very much a threat [37]. A DoS may either originate from a single source or from many distributed sources. The attack from distributed sources is known as a distributed denial of service (DDoS) [25]. A system being attacked by a single source DoS needs only to block traffic from that source. A DDoS requires a more sophisticated approach, and there may not be any perfect solutions, depending on the intensity and spread of the attack. When a system comes under a DDoS, it may receive modest traffic from hundreds to thousands of unique nodes. The synergistic effect of this traffic can easily overwhelm most enterprise level servers [35]. Figure 2.5 shows a generic example of how a DDoS is initiated by the attackers. The zombies (or agents in some literature) generally participate unknowingly in the attack. The attack coordinators are far enough removed from the actual attack execution that it is difficult to prove their involvement in many cases. Handlers may also participate in the attacks, but typically this is avoided to keep the trail of evidence from being too strong.

Source address spoofing gives rise to on particular kind of DDoS attack that is particularly difficult to deal with. Legitimate sites are tricked into playing part in a DDoS by group of zombies. Rather than using the zombies to directly flood the target system, the zombies spoof their source address to that of the target and then send a TCP packet to a number of legitimate web sites. Because of TCP's three way handshake connection protocol, those sites respond with an acknowledgement. Unfortunately, the target system is suddenly inundated by the acknowledgements from these various sources. TCP's automatic retransmission further increases the problem.

Another interesting class of DDoS is the flash crowd. This DDoS is not a coordinated attack with malicious intent, but has serious consequences nonetheless.

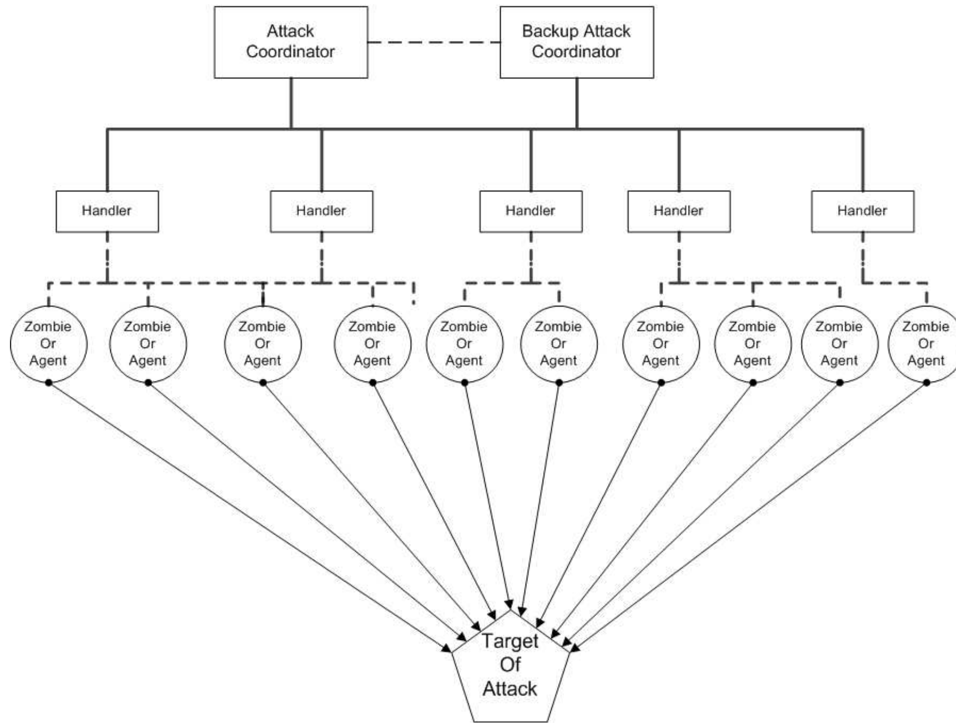

Figure 2.5: A Distributed Denial Of Service Attack

Flash crowds (also known as the slashdot effect [44]) occur when a particular web site becomes overwhelmed by legitimate requests for data because of a sudden increase in popularity or visibility stemming from being linked to by one of many popular article posting sites like Slashdot [39], digg [17], or Fark [20].

Much research has been in the area of protection against DDoS attacks using signature methods and or dropping specific types of traffic when under attack, using flow monitoring. For example, if the attack appears to be UDP based, then the firewall is directed to drop all UDP packets regardless of source. The problem with this 'solution' is that it gives the desired effect of a DoS for any legitimate sources communicating via UDP protocols.

One interesting direction of research is using history based IP filtering on edge routers of a particular network [35]. This research was discovered after the APHID proposal for doing a similar task in hardware, and it appears to have merit. Peng states that in a code red (a well known worm) attack the source IP address is spoofed

using a randomly generated IP address [35]. Only 0.06% to 14% of those spoofed IP addresses corresponded to addresses previously seen by the system. The authors describe a system of DDoS Protection where the edge routers of the network build up a history of IP addresses that have recently accessed the system [35]. They use two rules to define what a trusted IP address is. First the frequency of visits by an IP address, and second, the packet count generated by that IP address. Attack statistics show that most attacking IP sources generate a relatively low traffic volume compared to active users. Using this knowledge they build a system using the two rules to allow access (via a tuneable metric) to specific IP sources when an attack is taking place.

In Section 3.5 we discuss how APHID builds upon this concept and Section 5.3 shows how the hardware primitives improve the situation through increased speed of filtering and decreased operating system load.

2.13 Reconfigurable Hardware

Field programmable gate arrays (FPGAs) are devices that allow for rapid prototyping of digital designs [40]. Often described as a "sea of gates", an FPGA contains thousands of look up tables (LUTs) which can be used to implement arbitrary logic. Current generation FPGAs often contain embedded, optimized, hard wired, devices that are common fixtures in many designs. Examples of the hardwired blocks are multipliers, memories, and even processor cores. FPGAs are generally programmed using a hardware description language (HDL) which is compiled and translated to bitstreams by vendor specific software. Other approaches to programming involve writing code in a specialized C dialect that allows translation to the hardware. FPGA designs generally have limited clock speeds compared to integrated circuits. Applications where FPGAs are useful generally involve applications where using the parallel execution nature of hardware gives an advantage over serial execution of software commands. Pattern matching, vector operations, and memory to memory operations are examples of applications that see performance gains from FPGA implementations [47].

2.13.1 Useful Hardware Primitives. Using FPGA technology allows a designer to have access to (or to create) hardware primitives that have constant time performance, with low overhead, for problems that require multiple iterations or even polynomial time algorithms in software. By pushing the algorithm to hardware and exploiting parallelism, the designer can see huge performance improvements. One example of such a hardware primitive is the look up table (not the same as the tiny LUTs that make up FPGA fabric) . Look up tables have the same basic function as a hash table does in software. A key is associated with a memory location. The benefit that a hardware look up table has over a hash table is that the "hash" function in hardware requires very little time, whereas in software it can require significant overhead. Furthermore, access times for a look up table are often in the range of 1 to 2 clock cycles, depending on implementation. Of particular interest is the look up table known as a Content Addressable Memory (CAM). CAMs have the feature that the data to be stored is the key. This scheme is particularly useful in determining whether a piece of data is a member of a particular set. This research will make heavy use of CAM technology.

2.14 Chapter Summary

This chapter covered various topics including the concept of intrusions and intrusion detection. Other topics discussed include various methods and classifications of intrusion detection systems and current IDS implementations. A brief overview of the basics of network protocol stacks and network vulnerabilities is included. A short description of technologies used in this research closes out the chapter.

III. APHID Model

This chapter defines the APHID architectural model. The discussion includes a description of the generic architectural primitives as well as a proposed proof of concept application model. Finally, the chapter outlines the testing model that is used to examine the potential benefits of the hardware monitor enhancements.

3.1 Problem Definition

The current security climate is changing. Security was once an afterthought and performance was king, but we are now seeing a shift in priorities. Security is coming to the forefront as performance from current hardware exceeds the requirements of most systems. Attacks of increasing intensity, severity, and frequency on commercial operating systems with common vulnerabilities are forcing the community, at large, to examine other ways to combat the problems. APHID is one solution among many that should be part of a layered defense approach.

3.1.1 Research Hypothesis. **Hypothesis:** Intrusion Detection through the use of dedicated hardware running anomaly detection schemes at low levels (i.e., in hardware) will result in significant improvements in response time and reliability over software intrusion detection systems. Applying hardware primitives to the problem of denial of service attacks can increase a system's overall resistance to failure.

Software ID systems, running as a concurrent task on a multitasking operating system are forced to make a tradeoff between system overhead and detection granularity. In other words, a software IDS must be scheduled by the operating system to run. More frequent IDS scheduling by the OS results in a finer grained detection. That is, the chances of detecting an intrusion are greatly increased. The tradeoff is in terms of performance overhead. Every time the IDS gets scheduled, the system resources are being used for security processing rather than working on production processes.

By moving this overhead into hardware, we make the tradeoff at a different level. We sacrifice potential processing power (transistors, memory, etc...) to implement a hardware level detection mechanism that runs in parallel with the production process. The parallel monitoring frees the production processor from some of the burden of running IDS tasks, and therefore frees more resources for the actual production processes. Because we choose to protect a small attack surface at any given time, we are able to increase the reliability of results from anomaly detection.

3.2 Solution Framework

APHID builds off of the CuPIDS [45] model of using hardware to exploit parallelism and achieve faster detection and response to attacks on a system. CuPIDS uses a symmetric multiprocessor system where one processor (the security processing unit, SPU) monitors the operation of the other (production processing unit, PPU) processor. Special messages (execution traces) are passed through memory resulting in detection times orders of magnitude faster than a standard uniprocessor intrusion detection system [45]. This research extends and modifies the CuPIDS concept by further separating the SPU, departing from a symmetric multiprocessor to an asymmetric multiprocessor system with the SPU dedicated to the task of monitoring specific parts of the system. In the general case, any part of the system may be monitored. This application of APHID monitors the network protocols to include the OSI stack and system calls related to network operations. This scheme will provide general protection of device drivers as well as user processes.

3.2.1 Research Goals. This goal of this effort is to achieve significant reductions in response time to intrusions by asymmetric coprocessing to perform IDS functions. Response time, for the purposes of this paper, is defined as the time it takes the IDS to detect the attack and take some form of action, initially in the form of a simple notification.

To achieve the goals above, we define new hardware primitives and connect them to the PPU in ways that allow for novel detection schemes. We strive to answer the following questions:

1. Can a device driver be monitored by hardware to determine if malicious actions are occurring?
2. How fast can detection occur using the APHID scheme and how does this compare with existing methods?
3. What is the performance benefit gained by pushing the monitoring work into parallel hardware?

These questions can be summarized with the following goals.

1. Determine feasibility of detection through the use of hardware to monitor and protect device drivers.
2. Determine response time (time to detection in terms of instructions executed) of the APHID protected system and compare that to the software protected response times.
3. Develop an understanding of how APHID increased hardness on a system versus software only methods.

3.3 Approach

APHID is an asymmetric security co-processor that monitors device drivers and OS kernel code for anomalous behavior. Anomaly detection is facilitated by gathering statistics of normal behavior in a trusted environment. Then, when operating in an untrusted environment, APHID compares the current system state (or small portions thereof) with what it is given as normal activity.

The ideas behind APHID emphasize using hardware co-processing to enhance security instead of enhancing performance alone. In practice, the potential for performance enhancement exists because APHID is designed to reduce the burden of security on the production processing unit (PPU). In general, this design allows the PPU (which may consist of multiple processors) to operate as normal, without having to run a separate security task as overhead. The security task is offloaded to dedicated hardware where we take advantage of hardware parallelism to perform real-time security compliance checking. In our research we define real-time to mean monitoring

the PPU(s) as they operate without interrupting them to perform the checks. Only when a security event triggers APHID to take action will the PPU be affected by the existence of APHID. In contrast, a security process running on the PPU will always require system resources (memory, processor time, CPU cycles) regardless of attacks. Refer to Kuperman's definition of real-time in Section 2.3.

APHID is made up of hardware primitives, which are tightly coupled with the existing hardware platform. From the perspective of the OS, APHID does not exist. There is no need for OS support or connection to APHID. In fact, modifying the OS to be aware of APHID, and to have some control over it, would remove most of the security benefit in the event of many rootkit type of intrusions [23].

In the following sections we describe a notional system, composed of APHID primitives and their interfaces with the existing hardware.

3.4 Device Driver Monitor

The initial purpose for APHID is to monitor device drivers as they interact with the system. We have chosen device drivers because of their elevated privileges and generally limited interface with the operating system, that is, the attack surface is small (See Section 2.4). Because of this, we can create small and fast monitors to quickly detect intrusions. The same principles can be applied to system calls with some modification.

3.4.1 Modular design. APHID is designed to be modular, enabling the ability to monitor many types of devices. Additionally, APHID is designed to be useful for protecting other types of code. To facilitate these we propose the possibility of a Monitor Cache. The address comparator mentioned in Section 3.4.5 is used as a selector to a specific monitor based on which protected section of code (if any) is being executed.

Monitor Cache. A key function of the modular design is to allow for monitoring of many different devices and code types. These monitors are small enough to

reside in a reconfigurable memory cache. Figure 3.1 shows a monitor cache. Using the address comparator (shown as "state machine selector" in Figure 3.1), a specific monitor can be enabled when the section of code that it is designed to monitor enters execution on the PPU. The monitors execute custom state machines, tuned to the specific application. Enabling monitors in this way allows the monitors to be traded in and out in the same time it takes to to a process swap on the CPU, and the safety coverage can be seamless.

3.4.2 Anomaly detection. APHID uses anomaly detection to perform the monitoring tasks. To facilitate the anomaly detection, we use fast hardware look up tables to store a set of known legitimate actions (KLA). APHID monitors the behavior of a specific device driver and compares the actions it performs (jump to address x, return to address y, etc...) to the KLA set. If an action occurs that is not in the KLA set, the detector triggers and a flag is raised. The monitor then uses that information to perform corrective actions when possible. The KLA set is currently static and created ahead of time. Further enhancements to APHID can implement a dynamic reinforcement scheme where APHID is "trained" in safe mode, where intrusions are not allowed to occur. Then, using the dynamically collected KLA set, APHID can reinforce the static set and, additionally, learn from false positives and true positives through a feedback mechanism.

3.4.3 Justifications of Sizes. Several assumptions have been made that are important to note at this point. APHID monitors use control flow changes (branches, jumps, returns and calls) to generate the KLA set. Hennesey and Patterson [22] present profiles of several benchmarks, in Appendix B of their book. From these profiles we gather that 17% of all instructions in these benchmarks affect control flow. We will assume that most device drivers are comparable in instruction profile to these benchmarks. Ball *et al* have done a static analysis of Windows device drivers for the sake of error detection [6]. In their paper they list the lines of code for the twenty-six device drivers included in the Windows device driver kit. The sizes range from 24536

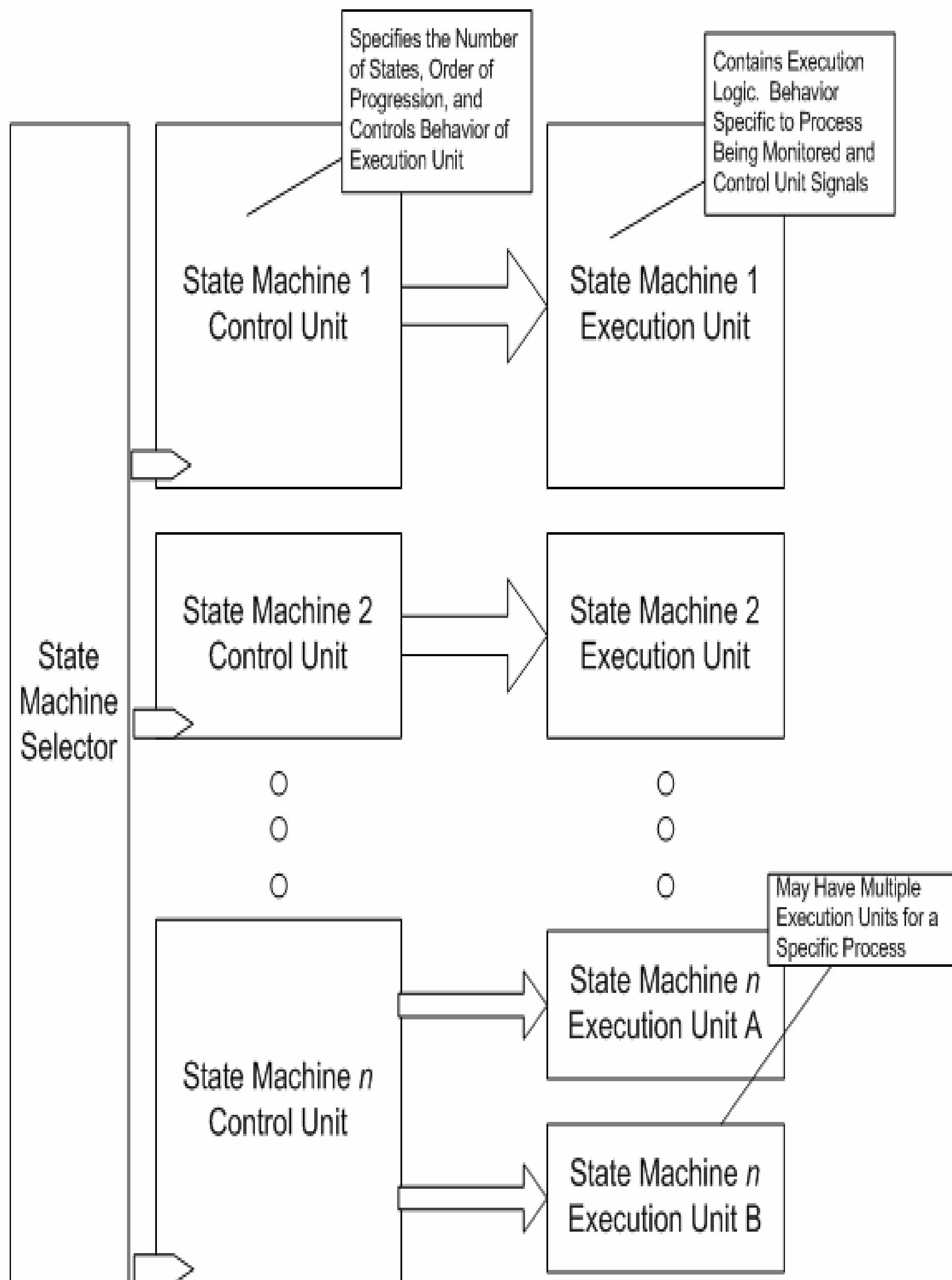

Figure 3.1: Monitor Cache

lines of code down to 304 lines of code. Using the metric of 17% of all instructions as branches we can generate KLA sets ranging from 51 entries up to 4172 entries. Based on several of compilations of C source code, the average line of C translates to somewhere between 4 and 5 assembly instructions. This is highly dependent on the number of function calls in the C code, but as a general rule, assume 5 assembly instructions per line of C, which means we should inflate the KLA set sizes by a factor of at least 5. The result is a KLA set of 255 entries to 20000 entries. Each entry takes 4 bytes resulting in a table size of 1200 to 80000 bytes. This means that a table larger than 64 kilobytes will require caching off of the Monitors memory, and the response time will be longer than that of a table smaller than 64 kilobytes.

3.4.4 Monitor Primitives. The driver monitor must be able to:

- Determine whether the current section of code in execution is protected.
- Decode the type of instruction being executed and any target addresses in that instruction.
- Compare the action to the KLA set.
- Take appropriate measures when the current action is not in the KLA set.

Corresponding to the list of requirements: The monitor has a look up table of protected addresses (or address ranges). There is hardware dedicated to decoding the instruction (similar to what exists for the PPU). Additionally a look up table for the KLA set exists. Finally, there is a mechanism for executing to repair damage from illegitimate actions. The mechanism could be as simple as a state machine (different from APHID's state machine) in hardware or as complex as a dedicated microprocessor running software (separate from the OS and PPU). Figure 3.2 shows a flowchart of the decisions that APHID must make and some actions that must be performed.

The monitor implements a Moore type state machine (see Section 3.4.6). For the sake of modularity, we separate the state machine into two parts; the control unit and the execution unit. The control unit takes an input of the system clock and

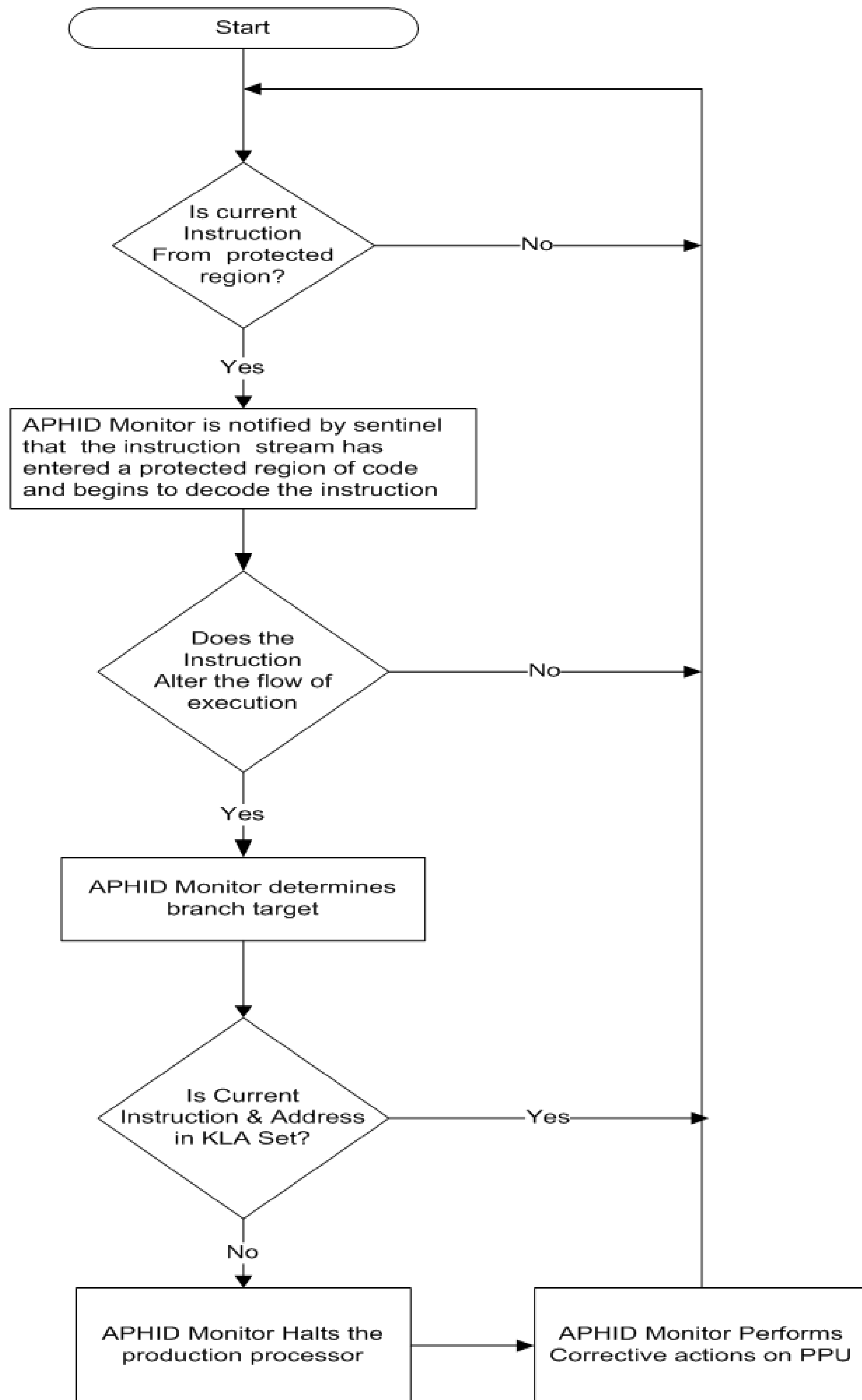

Figure 3.2: APHID Flowchart

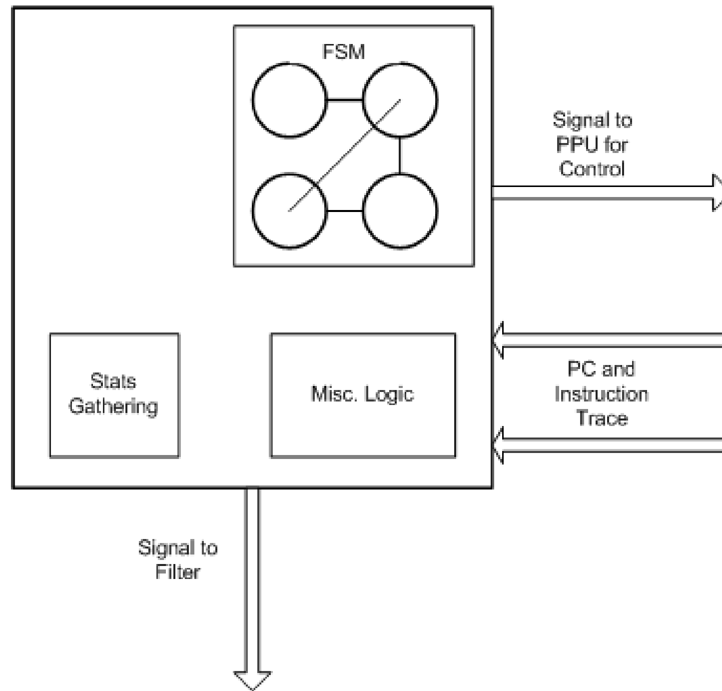

Figure 3.3: Device Driver Monitor Architecture

system state (current instruction in execution and program counter). As the control unit moves through the states, based on the inputs, it produces output signals to the execution unit. The execution unit then acts on the control unit output signals. See Figure 3.3 for representation of the system.

3.4.5 Address Comparator. APHID monitors must be able to detect when they are operating in a protected code section at hardware speeds. This is achieved by hardware look up tables where the current address on the program counter (PC) is checked against a table of protected addresses. With the proper implementation, this lookup occurs in constant time (O(1) in big O notation) with a low constant factor. The entire KLA set for a device driver (and potentially many device drivers) fits in a 64Kbyte block of ram which is adjacent to the monitors. Williams notes that the whitelist for WU-ftp (which is notably larger than most device drivers) fits in a cache of similar size [45].

3.4.6 State Machine. The APHID model can be created using various types of logic as needed. For example, an implementation may use gate level logic to create a detector using only combinational logic. Other implementations may use flip flops or registers and register transfer logic (RTL) to create a detector that is synchronized to the PPU and has guarantees on when detections occur. We propose using RTL and some combinational logic to implement a finite state machine (FSM) in hardware that allows the monitor to be easily modified and extended.

As shown in Figure 3.4, APHID's state machine consists of four states when monitoring device drivers. The initial state, WAIT, is where the state machine begins operation. While in WAIT, APHID watches for the section of code that it is designed to monitor. When the processor begins executing the protected code, APHID moves to the MONITOR state. While in MONITOR, APHID watches for any commands that alter the flow of execution, like jump or branch instructions. When one of these instructions is encountered, the address is checked using the comparator as described above. If the combination is not found in the KLA set, the state machine moves to the HALT state. In HALT, APHID signals the processor to stop and marks the anomaly in the instructions log. Once all actions in HALT are completed, the state machine moves to the RESUME state. RESUME makes any corrections or repairs and returns control to the processor. In future work, RESUME, will likely be expanded to several smaller states to accomplish more detailed repair tasks. Because we implement the state machine in hardware, APHID is able to execute the logic and transitions during the clock period of the production processor, allowing rapid detection of illegitimate behavior. It is important to note that only the transitions from WAIT to MONITOR and MONITOR to HALT require the single clock period implementation to keep up with the instruction stream. When illegitimate behavior is detected, the HALT and RESUME states have more flexibility in the number of cycles required to correct the situation.

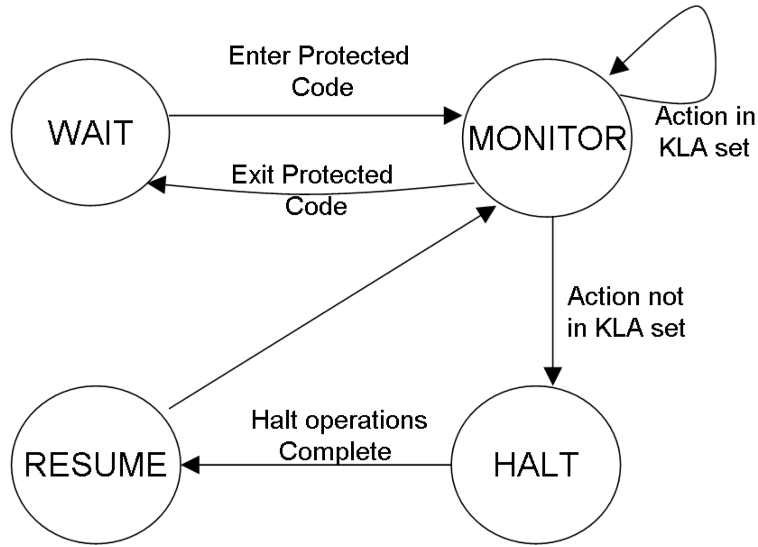

Figure 3.4: APHID State Machine

3.4.6.1 Corrective Action. Intrusion *detection* is not the only goal of APHID. A useful feature is the ability to repair damage from intrusions and resume normal operation. We have designed APHID to deal with this capability. The final state in the APHID state machine is the repair state. In practice this state will be made up of multiple states, or even another state machine. The implementation details will then follow some algorithm to repair or nullify any effects that the action may have had and return control to the production processor. For this system to work, one must assume that the architect of the monitor understands *all* of the effects that the driver has on the system to avoid stability issues. This is an area of future work. The possibilities for repair are depend greatly on the device and application being protected.

3.5 Network Stack Monitor

One application of interest for the hardware primitives is protection of the network interface to the system. APHID is not designed to replace the current security mechanisms (firewalls, secure network designs, etc...). However, we can use it to en-

hance the overall system security by adding an additional protection mechanism. We apply the APHID primitives to the network stack (from the network interface card, up to the OS). By doing this we can reduce the effect that a Denial of Service type of attack can have on a system like a web server, as well as protecting the system from intrusions. While protecting against a DoS, APHID is designed to maintain service (at a reduced capacity) by allowing access from trusted sources while denying access to untrusted sources. This may result in a denial of service to some legitimate users who do not happen to be in the trusted sources list (such as new customers), but should maintain service for the trusted sources. Additionally, APHID systems could be configured to communicate with the enterprise level firewall (as well as protecting the firewall) with trusted address lists. The firewall can then filter traffic as necessary to maintain the system integrity on a corporate/enterprize level.

For general intrusion detection purposes, we can set the appropriate code protection ranges in the APHID monitor and populate the KLA set for the monitor. By applying additional hardware primitives to the system, we can also add protection against denial of service attacks by using selective filtering of incoming network traffic. The network stack monitor is actually a device driver monitor with additional primitives to assist in monitoring the network stack.

The APHID network monitor sits (in logical terms) as a filter between the link layer (Layer 2) and the network layer (Layer 3) in the Network Protocol stack (for reference see Section 2.12.1). APHID needs to be as close to the hardware as possible to get the benefits of hardware speeds and to minimize or eliminate changes required to the protected operating system. The actual system implementation will dictate exactly where APHID needs to interact, but generally speaking, we can think of APHID as a thin layer between the traditional layers 2 and 3. Inserting APHID here gives APHID the ability to operate on raw (untouched by the Operating System) packets, but removes the requirements of knowing the details of the physical medium and link layer parameters such as error handling and retransmissions.

3.5.1 Limitation of the Network Stack Monitor. The network stack monitor works on the principle that we can remove the negative effects of a denial of service by refusing to operate on untrusted network traffic in an attack scenario. By pushing this filtering down to the hardware level, APHID can selectively drop packets and their interrupts/notifications at hardware speeds without involving the production software/hardware. Of course this is no substitute for secure network designs, and normal firewall techniques. One must assume that all nodes/links in the route(s) between the attacker(s) and the protected target have sufficient resources to handle the volume of traffic being presented by the DoS. If one of those nodes/links is overwhelmed, then the DoS succeeds at that point, the "weakest link". What APHID is proposing makes the end system a hardened system. By doing this, it is no longer the weakest link. In practice, the APHID system should be applied on all levels of the target infrastructure to push the attack resistance to the farthest distance from the production servers as possible (see Figure 3.5). In most cases, filtering will eliminate the majority of the attacking packets. Only in the event that the attacker knows the trusted sources list could an effective attack take place.

3.5.2 Operation of the Network Stack Monitor. During normal operation, the network stack monitor acts passively. That is, only the monitor is active, but the filter is disabled. Packets pass through the filter unaltered, and there is no difference (from the OS perspective) between this mode and an unprotected machine. The APHID driver monitor (described in Section 3.4) watches for intrusions as described above, and also keeps track of system workload parameters that are related to network traffic. For example, the driver monitor may keep track of the processor usage statistics by the protected driver process, and/or it may keep a record of the rate that the particular memory addresses associated with receiving a packet are accessed. When a threshold is crossed for a sustained period of time, APHID concludes that it is under a denial of service type of attack. This threshold is set based on past usage metrics. When the APHID driver monitor detects this threshold breach, it activates

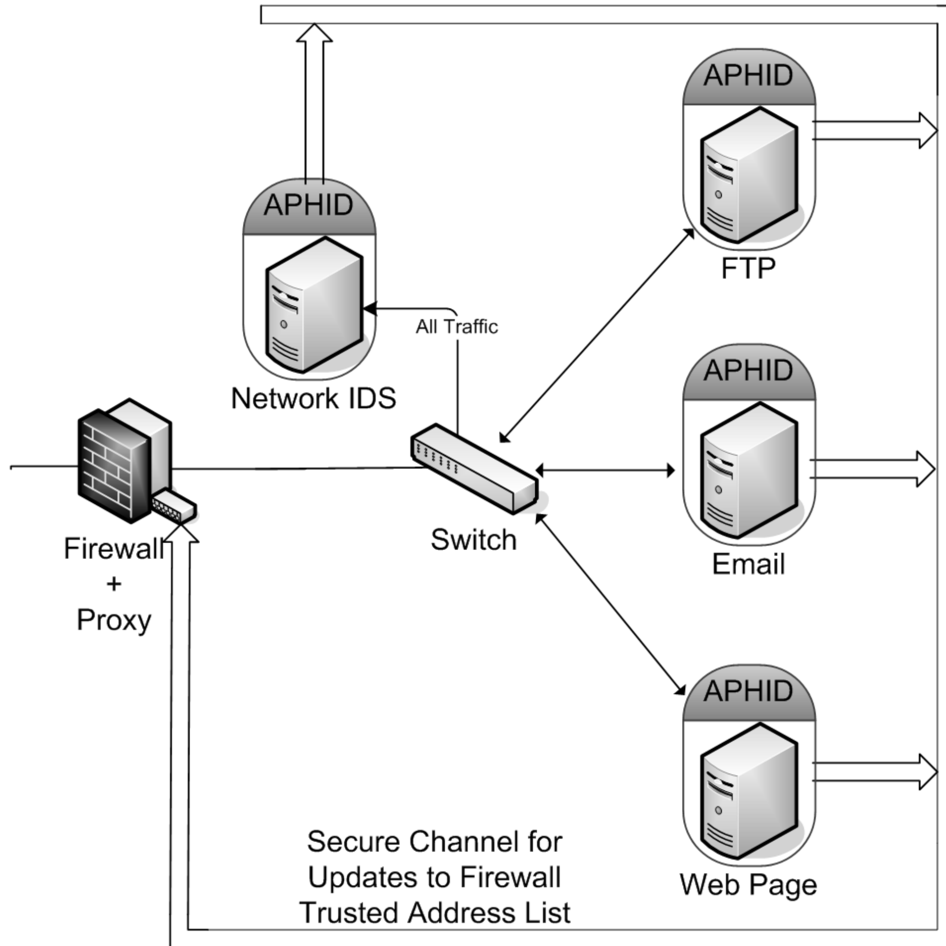

Figure 3.5: Deploying APHID Network Stack Monitor

the filter on the Stack Monitor. When the attack ceases, as determined by sustained sub-threshold workloads, then APHID resumes full, filter-free, service. In this way, access is restored to all sources, regardless of trust.

3.5.3 Filter Design. When activated, the filter works by examining a packet as it is received from the network. If the source of the packet is untrusted, then the filter simply drops the packet without signaling the operating system in any way. This can be accomplished in several ways, depending on the architecture of the system. Generally, whether the system is based on polling or interrupts, the filter keeps the packet from queueing to the operating system and drops the signal that lets the

system know that a packet has arrived. This filtering mechanism allows trusted users full access while muting the effects of untrusted packets.

3.5.3.1 Filtering the Packet. APHID Network Stack Monitor is only concerned with the IP packets it receives over the network. Other packet types are allowed to pass, since these packets usually originate from within the local network and are required for housekeeping purposes, one example is the address resolution protocol, which facilitates communication on a local area network [25]. As the packet arrives to the system, it passes through the physical and link layers of the Network Protocol stack (see Section 2.12.1 for details) and is then captured by the filter. The filter must extract the packet type from the frame and only if it is an IP packet does the filter continue processing. If the type is other than IP, it is passed on without filtering. The IP packet headers are parsed to determine the source, and the source is checked against a trusted sources hardware lookup table (similar to the address comparator and KLA set lookup tables). If the source is in the table, the packet is allowed to proceed. If the source is not in the table, then the packet is dropped and notification (interrupt or flag) of packet receipt to the PPU is suppressed. Figure 3.6 shows a flowchart of the decisions in filtering.

3.6 APHID System: Putting it All Together

The primitives outlined in Sections 3.4 and 3.5 must be combined with a production system to be of real benefit. Figure 3.7 shows a simplified computer architecture with the APHID primitives included. This layout is a notional design. Specific hardware platforms will differ in design due to architectural decisions and performance requirements. Generally, APHID requires access to the program counter (PC) and the current instruction under execution. APHID also needs to be able to capture interrupts from the device under protection (the network interface in our case) and the incoming data on the device. Finally, APHID needs to be able to halt the processor. Future work needs to be focused on what access is required to data structures

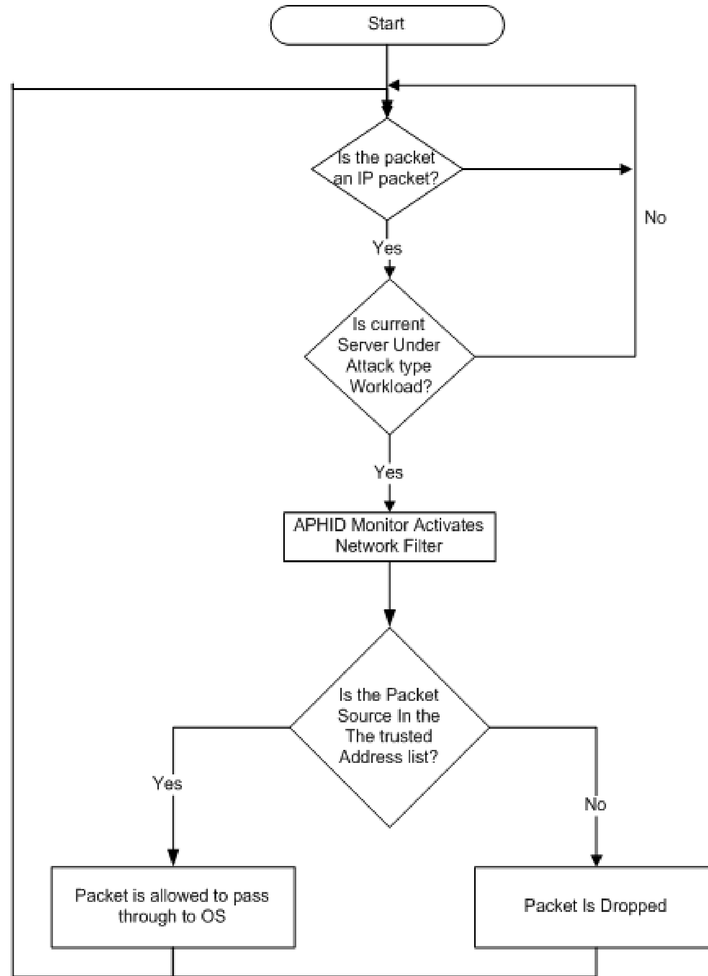

Figure 3.6: Network Stack Monitor Flowchart

and memory components for the purpose of repairing damage caused by intrusions. APHID aims to minimize the required access by keeping the time to detection very short, thereby limiting the potential for damage to the system.

3.7 APHID Testing Model

The Design of Experiments model is used to set up the framework for the APHID testing model. For reference on this technique see the NIST handbook [33]. The APHID testing model below is designed to represent testing of a critical parts of the APHID system in a white box environment. By supplying test vectors to

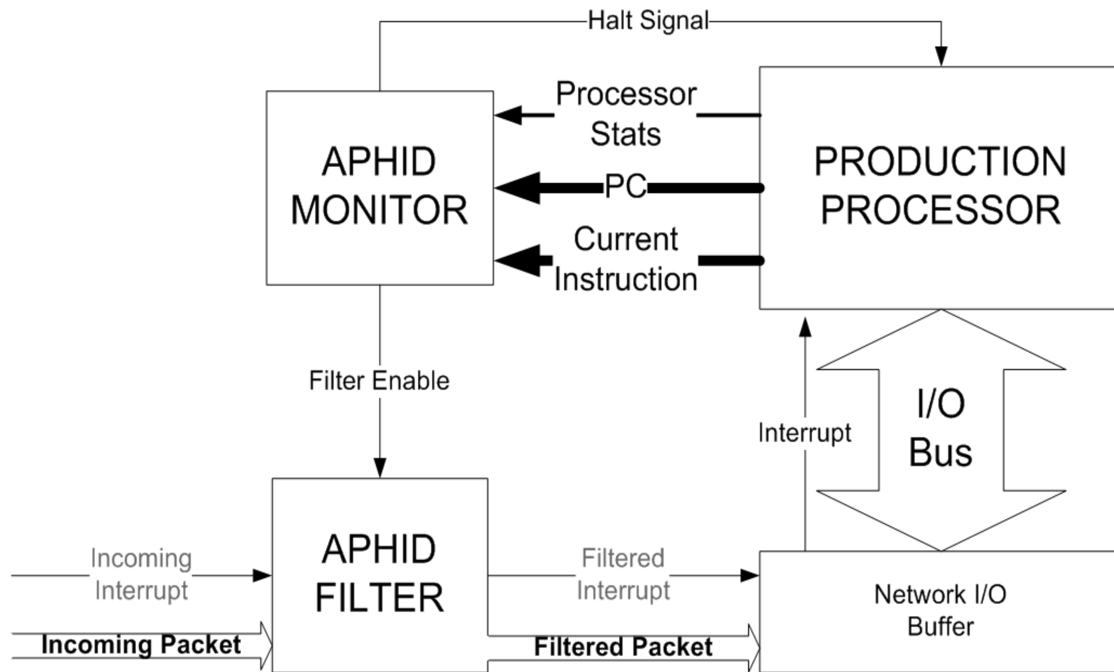

Figure 3.7: The APHID System

the system it is possible to determine whether the primitive being tested operates correctly and data about the latencies involved (in terms of number of clock cycles elapsed) is gathered.

3.7.1 System Boundaries. The Anomaly Processor In Hardware (AHPID) is the system under test. The scope of this research to detection of a DDoS attack with APHID enabled and with a software Intrusion Detection System. Therefore, only the network stack and drivers are monitored with the APHID system. Figure 3.8 is a block diagram of the system under test .

3.7.2 System Services. APHID provides the following services and outcomes. First, APHID provides intrusion detection. This is shown as Attack Detected, in Figure 3.8. The outcome of this service is binary. Either the attack is detected, or it is not. The final output of the system is a metric of the time until detection.

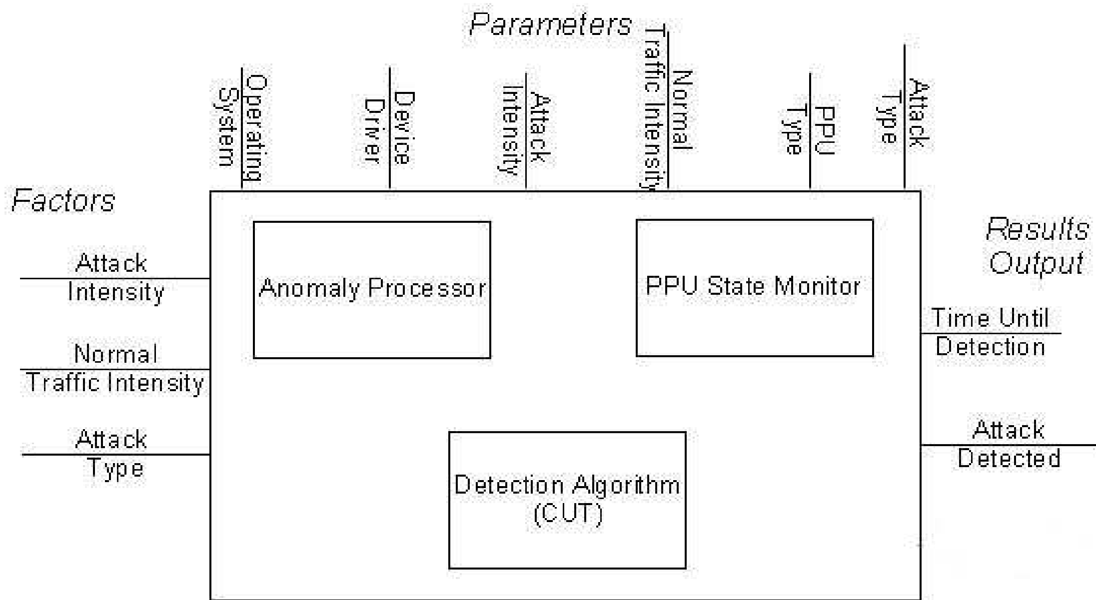

Figure 3.8: The System Under Test

As stated before, time until detection is measured in terms of number of instructions executed. Despite the fact instruction execution time is variable, the number of instructions executed is chosen because each instruction is a potential intrusion or a part of an intrusion.

3.7.3 Workload. In this scenario, APHID is protecting a web server. The workload presented to APHID is in terms of legitimate traffic intensity, attack traffic intensity, and the type of attack. Legitimate traffic intensity has three levels. Low intensity traffic, on the order of a few (less than 10) accesses to the web page per minute, moderate intensity traffic (2 to 5) accesses per second, and high intensity traffic (20+ access per second). These levels were chosen based on the capabilities of the hardware being used. In production class systems, the intensity levels may need to be adjusted to accommodate the higher performance capabilities of the systems. An empirical study on the capacity of the hardware would provide a more accurate

assessment of the requisite intensity levels. Attack intensity comes in 3 levels as well, corresponding to the same number of attacks per second as the legitimate traffic intensity. Attack type, the final workload parameter, is defined as either a buffer overflow attack on the network interface device (intrusion only), or a Denial of Service on the web server, or a combination, where the intrusion is masked by the DoS.

3.7.4 Performance Metrics. System performance, in terms of the SUT, is measured in time to detection of the intrusion, and correct filter operation. The second metric is a binary result. That is, the filter either works or does not work.

3.7.4.1 Time to Detection of an Intrusion. Time to detection is measured by counting the number of instructions executed since the injection of the intrusion to the system. This is not a true measure of time, rather it is a measure of the potential for damage to the system. Given that APHID is expected to detect attacks on the order of a few instructions, it is reasonable to assume that an attack may be captured prior to completion, allowing for damage repair. This stands in stark contrast to a software IDS which has no guarantees on when the multitasking operating system will return control to the monitor. It is conceivable that an attack could occur and never be noticed by a software IDS, while dedicated hardware can capture the same attack rather quickly.

3.7.5 Parameters. The system parameters that affect the SUT performance are the Operating System, PPU, and device driver being monitored. These parameters are fixed for these experiments because of the hardware platform we are using to conduct the research.

3.7.5.1 Workload Parameters. The workload parameters that affect performance of APHID testbed are normal (legitimate) network traffic intensity, attack type, and attack intensity. Normal network traffic intensity affects SUT because IDS algorithms can be thought of as a sensor, and if the 'noise level' is high, then it

	Factors		
	Attack Intensity	Attack Type	Normal Traffic Intensity
Levels	High	Intrusion	High
	Med	DDoS	Med
	Low	Both	Low

Table 3.1: The Factors

may be more difficult to detect a particular 'signal', or attack in our case. It would seem logical that an attack is more easily detected if the attack intensity is significantly greater than the normal network intensity. In reality, a DDoS attack works only because the attack overloads the server under attack. The attack type is a parameter of the workload because APHID is designed to deal with both intrusions and denial of service attacks.

3.7.6 Factors. The factors selected are normal network traffic intensity, attack intensity, and hardware/software detector. Normal (legitimate) network traffic intensity has three levels, low, medium, and high. Section 3.7.3 defined three levels of intensity for both attacks and legitimate traffic. Low intensity is on the order of 10 hits per minute. Moderate intensity is on the order of 2 to 4 hits per second. High intensity is on the order of 20+ hits per second. Moderate and highs levels were chosen to study the 'interesting' parts of the system performance. We assume that low intensity normal network traffic is of minimal interest because software detection schemes should be able to manage under this scenario. Attack intensity also has three levels, low, moderate and high. While, a low intensity DDoS attack may not even be successful as a DDoS, a low intensity intrusion is the most likely type of intrusion. The attack type has three levels; DDoS only, intrusion only, or both. Refer to Table 3.1 for the factors.

3.7.7 Evaluation Technique. This research evaluates performance using empirical measurement based on simulation of VHDL and direct measurements of implementation on limited parts of the system under test. There are no current analytical models for this type of study, and currently there is no implementation of a full APHID system. FPGA technology allows rapid hardware design turnaround required, and gives us a mechanism to attach our hardware primitives to a soft-core (VHDL based) processor. The main evaluation technique is simulation of the VHDL for proper operation. Direct measurements on the hardware are possible using embedded debugging components in the reconfigurable FPGA fabric. However, lacking a working implementation of APHID, we fall back to evaluating the hardware primitives using VHDL simulation of the design that is eventually compiled into the full system. From the simulation we can obtain reliable data about the operation of the hardware primitives. These data are the number of clock cycles for the particular operation of a primitive, and correct operation of that primitive, based on the inputs given.

3.8 Chapter Summary

Chapter III covered the APHID design and testing models. APHID design consists of several components. The heart of APHID is a state machine designed to execute a monitoring algorithm, based on the flowchart shown in Figure 3.2. Additional primitives are required to accomplish the required monitoring and security functions. These primitives are the KLA set, which is made up of a hardware look up table, the network filter, which eliminates untrusted packets from the network, and a monitor cache to enable the capability of protecting more than one device.

The APHID testing model defines the proposed method of testing a full APHID implementation. The tests include running the system under a DDoS, and attacking the APHID protected system with buffer overflow class intrusions over the network. Varying intensity levels are defined to test the system under different load levels.

Finally the methods of simulating APHID components are described as a backup plan to testing the full implementation.

IV. APHID Implementation

This chapter describes the implementation and testing of APHID. Implementation of the Chapter III models are detailed, giving more information regarding how APHID connects to the test production system. The reader must note that there is a distinction between the planned implementation and the current state of the implementation. Where necessary, a comment regarding these differences is inserted, and finally a section is dedicated to the impact of these differences on the testing model provided in Section 3.7.

4.1 Hardware and Software Platforms Used

The APHID platform will be entirely contained on a Digilent XUP2VP FPGA evaluation board based on a Virtex II Pro FPGA. The initial test configuration uses a Microblaze Xilinx IP core is instantiated on the FPGA as the production processor. Using Xilinx debugging tools gives some capability for metric gathering purposes, and simulation provides the mainstay of the results at this time. The APHID FPGA configurations are designed using the Xilinx ISE 8.1 Professional Edition and Xilinx EDK 8.1 Professional Edition. uClinux is the operating system being run on one Microblaze core, while a stand alone process runs on another Microblaze core. This stand alone process is created to reduce the complexity involved in writing a device driver for the embedded uClinux operating system, given the time constraints. The network is set up using a network traffic generator as the aggressor machine. The traffic generator is a LANForge Appliance by Candela Technologies [12]. The generator is capable of emulating 2000 separate network connections through multiplexing 4 data generator ports, and can generate 250 Mbps worth of UDP/TCP traffic per CPU, resulting in 500 Mbps traffic. The aggressor and APHID are connected using a 8 port switch to allow the aggressor to behave as a Distributed DoS. The traffic generator comes with scripts that execute DDoS activities. A custom payload in a single message serves as the test for the intrusion detection component.

An alternate implementation uses the Altera Quartus II 6.1 WebPack HDL design envionment. This environment is used to implement the APHID device driver monitor on a simplified MIPS processor. The choice to use this platform was made because the MIPS processor was already working in that environment and rather than dealing with porting the VHDL to the Xilinx environment, we chose to port the APHID primitives (which are smaller) to the Altera environment. The functionality provided by each environment is essentially equivalent.

4.2 APHID Network Stack Monitor Implementation

Figure 4.1 shows the implementation details of the APHID filter. The filter takes the incoming packet (and its associated interrupt) as inputs. An additional input is the activation control from the APHID monitor. If the monitor does not enable the filter, then the packet and associated interrupt pass through the filter with no processing or delays and proceed directly to the I/O buffer. When the APHID monitor asserts the activation control to True, then the incoming packets are filtered by placing them into a register. The source address of the packet is extracted from the packet itself by a set of fast combinational logic gates and that address is used as input to a content addressable memory. The CAM responds with a boolean result in 1 clock cycle. We use the result of the CAM lookup to enable the output on the register. If the address does not exist in the CAM, then the CAM output is logic low resulting in no output from the register. If the address does exist, the CAM responds with logic high and the register output is enabled, so the packet and its interrupt are allowed to pass through the filter. Note that this introduces one additional clock delay to the transmission of packets up the stack, but this is not a concern because the filter is clocked at the same rate as the rest of the memory. It does not cause a bottleneck, only 1 additional clock latency.

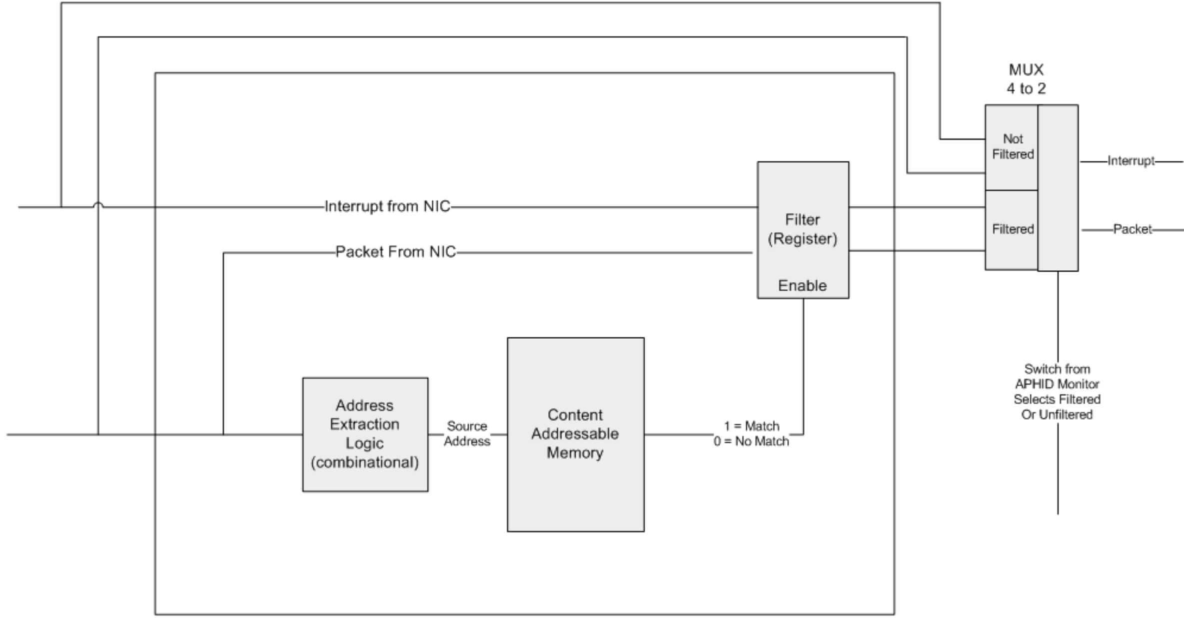

Figure 4.1: The APHID Network Filter

4.3 Integrating APHID with a Production Processor

This section describes the attempts at integrating APHID with a working production class processor. We were able to make some progress, but due to time limitations, there is not a final implementation of APHID working in concert with the real processor.

4.3.1 Proposed Architecture. In designing APHID, a conscious choice to target the RISC family of architectures was made. This is mainly due to the fact that the processors on the FPGA platforms are RISC based (both the Power PC and Microblaze are RISC machines) and also due to the easier implementation of hardware when using register-register architectures. Figure 4.2 is a component level diagram of APHID working with a RISC processor. Here MIPS is represented, but Microblaze and PowerPC architectures are very similar.

4.3.2 Successes. Related research proposed by Stephen Mott [31] shows that this is feasible. While the APHID model is fundamentally different from his

Figure 4.2: The APHID Implementation on a RISC Processor

work, the concepts and techniques used are similar. Building from that research, it is possible to connect APHID primitives to the Microblaze processor.

The Xilinx tools allow for two distinct modes of creating systems with embedded processors and custom peripherals. In this case, the APHID primitives are custom peripherals. In the first mode, one must create the peripheral using the ISE development environment, and then create a custom wrapper to create an IP core. An IP core is a 'drop-in' component used by the Xilinx design tools. This IP core is then accessed through command options in the XPS design environment and the bus architectures are connected inside the tool. The second method is an inverse of the first method. First the embedded system including processors and memories is created in the XPS design tool, and the necessary interfaces are made visible. Then the design is exported to the ISE development environment where the additional architectures are attached as needed. The benefit to this method is that the XPS component does not require drivers for the new hardware, whereas the previous method does require custom drivers regardless of their necessity.

4.3.3 Roadblocks. Several roadblocks were encountered during the pursuit of this research. The baseline design required an embedded Linux kernel running on the Xilinx Microblaze processor core. Due to tool chain complications it became evident early in the process that the task of integration with the Microblaze core would be more difficult than initially expected. The implementation chain chosen required that drivers exist for the hardware primitives, and this was an error in design. Reverting to the design flow of creating a project in XPS and then augmenting the hardware in ISE required us to redo much of the work. The compiling tool chain proved difficult to work with because of the degree of customization available and lack of relevant documentation for a multi-core example. When the baseline Microblaze running uClinux was finally operational, there was little time to tinker with adding new functionality to the core. We chose instead to fall back to a simpler test system.

4.3.4 Fallback Options. Starting small, the APHID primitives were implemented in VHDL and tested in simulation only. Our next step was to attach APHID to a minimal VHDL implemenation of a MIPS processor (Appendix B contains the full VHDL source code for the processor). This processor is available with the Altera DE2 evalution board [1] course software and is described in the book [21]. The minimal MIPS implementation has only a small subset of the instruction set architecture. This limits the ability to test APHID with real software. Fortunately it is possible to create a small set of processes that can test the ability to catch actions not in the KLA set because simple PC relative branching exists.

4.4 Intrusion Attacks on APHID

Using the minimal MIPS architecture we are able to create several small procedures to run in a simulation. Since the waveform screen is small, and all of the instructions must be 'compiled' by hand to opcode, the procedures are kept to a few instructions each. To test APHID, the processor operates on a sequence of actions where the entries (branch + address) exist in the KLA set, and confirm that legitimate actions are unaltered. Then a sequence is executed where some of the actions are not entries in the KLA set. The processor simply halts and control is shifted to a "fail safe" loop upon attempted execution of an illegitimate action (see Figure 4.3).

4.5 Distributed Denial of Service Attacks on APHID

Because we have been unable to get a working prototype to hardware, we cannot test the system using real network traffic. As a substitute we have created a limited set of addresses that are fed to the APHID Network Filter and we monitor the results of the trusted source set output when presented with these addresses. Doing this verifies correct functionality at the behavioral level. Direct compilation of the VHDL to hardware should have equivalent operation.

To keep the test as realistic as possible we use IPv6 length addresses (128 bits) and we create packets the size of the maximum transmission unit defined by the

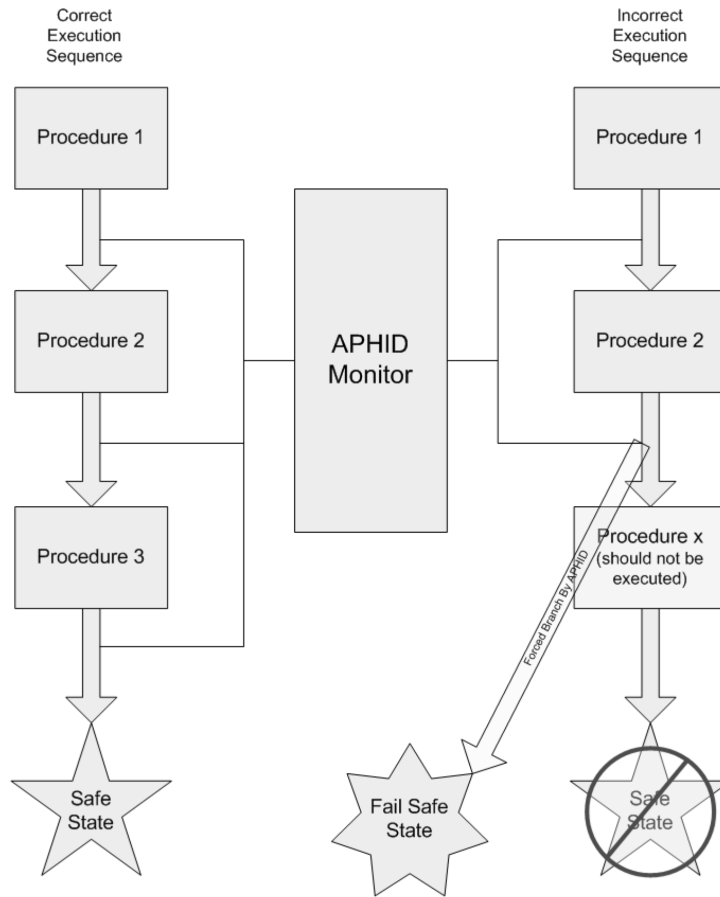

Figure 4.3: The APHID Network Filter

IPv6 protocol. In so doing, future applications should require minimal rework to incorporate the APHID primitives into a real system.

4.6 Incomplete Implementation and its Effect on Testing

Because of the difficulties in implementation on hardware, we have not been able to test APHID according to the test plan in Section 3.7. Work towards future publications, beyond this document, will make concerted efforts to get APHID running on a real hardware testbed. In the mean time, we have been limited to running simulations on small components of APHID and to connecting APHID to minimal processor implementations. There is a potential for instantiating the APHID primitives in the

FPGA hardware and testing performance on hardware, however, the benefit of this over simulation is minimal, and the amount of work required would be significant. The hardware instantiation and testing would definitely be of great use for follow on research. Not only would it expose the researchers to the tool chain early on, but it would also provide hands on insight into how the hardware primitives work.

V. Results

This section shows the results of simulation, numerical calculation and limited hardware instantiation of APHID. The test model from Chapter III is followed to the degree that it is applicable to the current limited implementation of APHID. Many of the tests proposed in Chapter III are not feasible with the current lack of networking support. Simulations are substituted where possible. We simulate the components first and then build up to a more robust simulation involving a MIPS processor in VHDL as described in Chapter IV.

5.1 Results of Tests on APHID Primitives

5.1.1 APHID Finite State Machine Simulation. The first test is a simulation of the APHID state machine. Refer to Section 3.4.6 for a discussion on the modeling of the state machine. The state machine has the following inputs and outputs. Figure 5.1 shows the results of the simulation.

Inputs:

- CLK – The system clock.
- Protected – Flag to indicate that CPU is operating in protected code.
- Legitimate – Flag to indicate that the current action is in the KLA set.
- Ready – Flag to indicate that HALT operations are complete.

Outputs:

- halt_out – Halt signal sent to APHID monitor which then forces the CPU to halt. In this early implementation the halt is accomplished by forcing the CPU to execute a NOP repeatedly. The PC is not allowed to advance. Alternative implementations could force the CPU to a ‘dump’ routine where the state is quickly saved for forensics and then damage could be repaired before returning the processor to normal operation.
- resume_out – A signal sent to the processor from the RESUME state to return execution to the protected code section again.
- current_state – For simulation purposes only, this displays the current state the FSM is in. Note that there is a transient “glitch” in the waveform output when

both lines change (i.e., from state 1 to 2 which translates as 01 to 10 in binary). This is not an error in the FSM, rather it is a result of displaying the state with combinational logic rather than clocked logic.

The reader can observe in Figure 5.1 that the state machine behaves as desired. After reset, the state machine sits in state WAIT (FSM_WAIT or S0 in the source code) and will remain in WAIT until the address comparator (as described in Section 3.4.5) signals that the processor is entering a protected section of code by raising the input `Protected` to logic 1. While `Protected` is asserted, the FSM moves to the MONITOR state (also known as S1). The FSM will remain in MONITOR while `Protected` is asserted and `Legitimate` is stays at logic 1. If `Protected` stays at logic 1 and `Legitimate` is set to logic 0, then the FSM enters the HALT state (S2) and raises the `halt_out` signal. This signal is attached to the execution unit of the monitor as described in Section 3.4.6. The execution unit (not shown in this simulation) then performs the tasks necessary to halt the PPU and perform corrective actions (as needed). The FSM remains in HALT until the execution unit responds with `Ready` asserted to logic 1. When the FSM received the `Ready` signal, it proceeds to the RESUME state (S3) where the `resume_out` flag is pulsed for 1 clock cycle (until the FSM leaves RESUME). The `resume_out` signals the execution unit that the FSM is now entering MONITOR and that the PPU should resume operation. Note: There is a 2 clock cycle delay on the `current_state` output. Since the FSM is a moore type state machine, where the outputs occur only at states, one should notice only a 1 cycle delay from the time the input causes transition until the output reflects the change. In post simulation analysis, we determined that the cycle shows up because the outputs in the simulation are registered (that is, the output is stored in a register), causing an additional one cycle delay. This registering is an artifact of a configuration option. When the FSM is part of a larger system, the outputs are not registered because they are inputs to other synchronous systems (e.g., memory, processor, execution units) and the additional register would cause a redundant delay.

Appendix A contains the full source code for the APHID state machine.

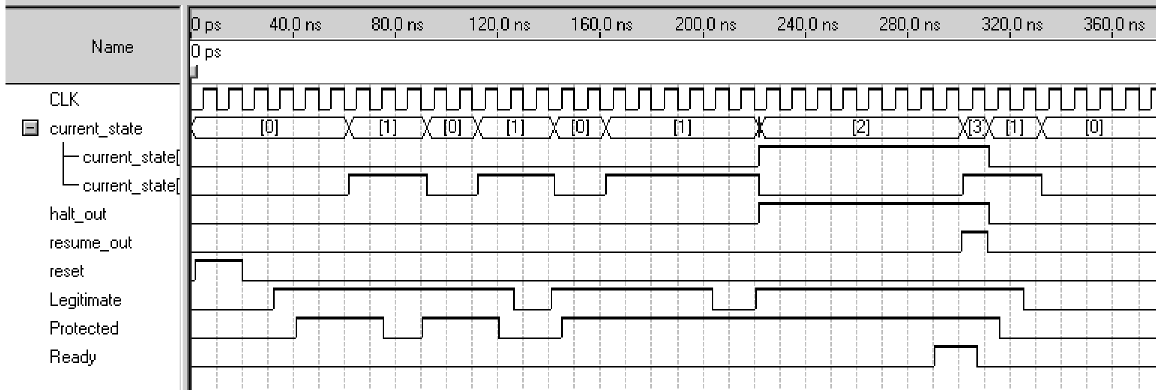

Figure 5.1: Results from Simulation of APHID State Machine

5.1.2 APHID Address Comparator Discussion. The APHID address comparator uses a content addressable memory (CAM) to achieve fast determination of set membership (KLA set or Trusted Address set). Rather than recreating the hardware, we rely on Xilinx (or Altera) prefabricated cores. The benefit: The design is optimized and tested. In lieu of simulation, we refer the reader to the Xilinx user guide and data sheet [46] for the CAM.

5.1.3 Network Filter Simulation Discussion. The network filter uses a CAM to test whether an address belongs to the trusted address list. There is some additional logic associated with the filter that switches between filtered and unfiltered address data. The actual filter is a large register with an output enable which is triggered by the "Match" signal from the CAM. The filter has not been simulated at the time of this writing.

5.2 The Benefits of Hardware Content Addressable Memory

APHID uses look up tables to perform address checking, KLA set lookup, and network source address lookups. In this section, a hardware CAM is compared to a software hash table as implementations of the look up tables. Both units provide O(1) data lookup/retrieval in the average case, and both can be used to perform

set membership tests. This comparison shows how a hardware implementation can reduce the burden of security by exploiting hardware parallelism.

The Xilinx implementation of a content addressable memory described in Section 2.13 has a latency of one clock cycle. That is, once the item for lookup (in our case the address on the PC) is ready, the CAM furnishes a result of present or not present in 1 clock cycle. For very small table sizes, one can assume that the entire look up table resides in memory. As the table size increases, one must take into account the access latencies incurred as hierarchical memory (from cache to ram to disk) is accessed.

Now, examine the following hash function written in C. We assume the machine is a 32 bit architecture, resulting in the macro `INT_SIZE` being equal to 32.

```
00 /* Bitwise hash function. */
01 unsigned int bitwisehash(int address, int tsize, unsigned int seed)
02 {  char c;
03    unsigned int h;
04    int i;
05
06    h = seed;
07
08    for(i=0; i<INT_SIZE;i++)  //INT_SIZE = 32 (bits)
09    {
10     h^=((h<<5)+c+(h>>2));
11    }
12    return((unsigned int)((h&0x7fffffff) % tsize));
13 }
```

From this code we can get an instruction count per function call. Calling a function requires the system to load all of the operands. For this function we can assume that the function call results in five operations. For the purposes of this exercise, variable declarations incur no operations. The following is a list of the operations associated with each line.

- Line 01: 5 operations (per call) 3 loads + 2 overhead
- Line 06: 1 operation

- Line 08: 2 operations (per loop iteration) + 1 operation for loop initialize
- Line 10: 4 operations (per loop iteration)
- Line 12: 4 operations (for the exit function)

To summarize, the function call overhead (call + return + miscellaneous non loop instructions) is 10 operations. The loop initialization is 1 operation and each loop iteration requires 6 operations. The total instruction count for one hash function calculation is:

Total Instruction Count $= 32 \times 6 + 11 = 203$ instructions.

Once the hash function is calculated, the hash table lookup takes around 20 operations to fetch from an address in memory. In a modern, superscalar processor, we can safely assume an instruction issue rate of 2 instructions per clock cycle, a hash table lookup, which includes the hash function plus the retrieval from the table, takes $\lceil 1/2(32 * 6 + 11 + 20) \rceil = 112$ clock cycles (assuming the ideal case where no cache misses occur). The resulting comparison is one cycle in hardware (every time) to a minimum of 112 cycles in a software implementation, showing significant improvement. Every branch type instruction executed in a protected code section incurs this cost. If we were to do this check in software, on the production machine, the overhead would be prohibitive. In hardware, the lookup on the instruction can occur in parallel with the then next instruction fetch so the single clock cycle cost is absorbed by the parallelism.

To be fair, the single clock cycle look up time in hardware is correct for most driver monitors. Refer to Section 3.4.3 for justification of the KLA concepts of size. If the monitor is larger than 15,000 lines of C code, it is likely that a 64 kByte block ram will not be large enough to hold all of the entries in of the KLA set for the driver. A more accurate *average* hardware lookup time includes the extra delay of accessing a larger memory at a slower speeds, similar to the performance metrics of a cache miss penalty calculation. Below is a simple equation for the average access time in hardware, using variables.

$$A_{Tavg} = p + (1-p)[\frac{M-C}{C} * 10 + (1 - \frac{M-C}{C})] \quad (5.1)$$

Where `p` is the percentage of monitored device drivers greater than 15,000 lines of C code, `M` is the size of the monitor KLA set and `C` is the size of the cache memory.

5.3 Comparison of the APHID Network Filter to Existing Research

Peng's research shows that 90% of legitimate traffic is protected using only 4 megabytes of memory and 80% can be protected using only 800 kilobytes of memory [35]. Using these numbers, we can estimate the size of the memory associated with the monitor. Using APHID in a manner shown in Section 3.5.1 can allow us to minimize the required memory of a specific APHID monitor by combining the histories of several monitors protecting different sources. Each monitor then updates the firewall rules as necessary.

In hardware we can filter out the untrusted messages in 1 clock cycle. This is the same argument used in the hash table discussion in Section 5.2. The additional clock cycle is incurred because the messages must wait for the CAM to trigger the register output (Refer to Figure 4.1 for architecture).

Filtering, as done in Peng's research, is performed in software using firewall rules. To accomplish the filtering, the firewall must:

1. Parse the packet: Assume this can be done in constant time requiring 20 clock cycles to find the source address. (Cost 20 clock cycles)
2. Do a rule table lookup on the address: We will assume this is similar to the discussion of the ideal case in Section 5.2. (Cost > 120 clock cycles)
3. Perform the filtering: Assume 10 clock cycles for the decision making. (Cost < 10 cycles)

We see that the hardware version has even more of an advantage over a firewall. In this case the difference is around 170 to 1 in the case where the firewall does not

have to access main memory (which is an unlikely case). The burden of processing the firewall rules for a system already under duress (as in a DDoS) may be the final blow to the system.

Fortunately, APHID's layered approach allows the particular services to be protected from the immediate surge, and then the enterprise firewall can be updated via a secure channel to include the trusted lists from the APHID monitors to reduce the impact on the enterprise network infrastructure. This could be thought of as buffering. The APHID monitors initially absorb the early brunt of the attack until the heavy duty enterprise level IDS and firewall can take up the slack.

5.4 Total System Integration

Currently APHID is not integrated to a CPU. Integration with the minimal MIPS processor is very close to completion, there are still some problems with memory interfaces and programming. Because of this, a simulation of the system is not available. We rely on the architecture and the timing analysis to come up with an estimate of the performance.

5.4.1 In System Performance Estimate. See Section 4.3.1 and Figure 4.2 for the description of how APHID interacts with the processor.

APHID is not active when the PPU is operating on unprotected code. There is no performance overhead in this mode. When APHID activates, there is no performance overhead while the actions are legitimate (no anomalous behavior, all actions in KLA set). However, as soon as an intrusion (illegitimate action) occurs, APHID forces the PPU into a HALT loop (for a fixed number of iterations in this context) where state information is gathered and processed, and then pushes execution back to the RESUME and MONITOR States.

The following equation shows the overhead as a function of the intrusion rate and Halt State cycles, measured in terms of the ideal processor performance.

Let A be the attack rate in terms of illegitimate actions per 100 lines of code.

Let C be the number of cycles dedicated to the Halt state.

Let I be the ideal performance.

The equation for performance under attack is:

$$P_A = \frac{I(1-A)}{100} + \frac{A(C+1)}{100}. \tag{5.2}$$

The (+1) component of $(C+1)$ is the single cycle cost for the RESUME state. Observing the equation, it is important to keep C as small as possible and, A should not be large. If A becomes too large then the system could experience a DoS because APHID will constantly be processing in the HALT state. Future work could look at the possibility of task switching to an unprotected section to perform the APHID state saving and state correction in parallel with regular unprotected code.

VI. Concluding Remarks and Future Research

6.1 Concluding Remarks

In this research, it has been shown that APHID primitives provide the a system with increased security, hardening that system against attacks. Using small, fast hardware sensors allows APHID to achieve detection of intrusions in real time by capturing the intrusion as it executes. The APHID network monitor gives a robust protection against DDoS attacks with the caveat that APHID can only protect the link that it sits on. Because of this, it makes sense for the deployment of APHID to be in place on all links in the system. Intrusions can be detected as they occur because of the hardware level visibility made possible by APHID's primitives.

APHID is not implemented (currently) as a full system. Many of the pieces have been tested and simulated, and the architectures have been laid out in detail.

6.2 Contributions

This research advances the field of intrusion detection by pushing the overhead of security monitoring down into low level hardware primitives which operate at the same rate as the system clock. In doing so, we have shown that tremendous speedup is possible over a notional ideal operation of key functions (Hash table lookups). The total overhead introduced by APHID when in system is dependent on the rate of attacks and the type of correction desired. Simple intrusion reporting will introduce very low overhead, while intrusion correction can be significantly more expensive (per intrusion). However, capturing the intrusion in a single clock cycle keeps the potential damage to a minimum because the register state of the machine is not altered until the legitimacy of the action is checked by APHID. Keeping this in mind, we can create an implementation of APHID where the repair overhead is accomplished while known unprotected code is in execution.

We have shown that APHID can be used to detect intrusions and also to protect against DDoS attacks, by keeping a history in the filter memory and dropping packets not in the history when APHID is under attack from a DDoS.

6.3 Future Research Opportunities

The nature of this research provides many opportunities for further efforts in this field. One area of further pursuit would be to implement and test multiple monitors and a reconfigurable cache of monitors. Using the cache, and reconfiguration of the cache, it may be possible to provide coverage for many of the vulnerabilities in an operating system while keeping the overhead to a minimum. Leveraging work by Montminy in dynamic reconfiguration of bitstreams in FPGAs may be applicable to this work [30].

APHID's current implementation is focused on protection of device drivers. Future work may make the application of APHID more general and allow APHID monitors to secure protections such as system calls, library functions, or even services (such as a mail server) running on a processor.

One question that comes up when working with the hardware primitives is: Can APHID state machines be used for security policy compliance monitoring relatively small sections of code, and can these state machines work together to create a more robust SPCM system.

Other research has proposed hashing code segments to provide verification that the code has not been changed [2]. APHID primitives may provide a way to perform those hashes in real time without burdening the production system. Doing this can provide assurance that the processes have not been modified by an intrusion, or modified themselves illegitimately.

Pipelined production processors with high issue rates may pose a difficulty for monitoring with the current APHID implementations. More effort could be directed to determining how pipelining affects APHID monitors, and can the APHID monitors be pipelined themselves to maintain pace with the production system.

Finally, the concept of networking APHIDs with other APHIDs to form a hybrid distributed network intrusion detection system (as alluded to in Section 3.5.1).

Appendix A. APHID Primitives in VHDL

This appendix contains the VHDL source for the APHID primitives as they exist at the time of this writing. Because this research is intended to survive beyond this document, the most current design configurations and documentation will be made available upon request. The APHID Network filter is currently in block diagram/schematic format. Using Xilinx proprietary custom IP cores does not allow one to create full VHDL sources. It is possible to create a VHDL wrapper and entity declarations to show the VHDL interconnect. The filter was created in a schematic form, and has not been compiled to VHDL at this point.

A.1 APHID State Machine Primitive

The code below corresponds to the first cut of the APHID state machine. It stands alone and contains no vendor specific commands. Using this code as an example is recommended. Make sure to name the file APHIDFSM.vhd. Alternatively, request an electronic copy of the source files.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY APHIDFSM IS
PORT
(
clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
Ready, Protected, Legitimate : IN STD_LOGIC;
halt_out, resume_out : OUT STD_LOGIC;
--cs : OUT STD_LOGIC_VECTOR(3 downto 0);
current_state : OUT integer range 0 to 3
);
END APHIDFSM;
```

```
ARCHITECTURE rtl OF APHIDFSM IS
TYPE state_type IS (FSM_WAIT, MONITOR, HALT, RESUME);
SIGNAL state  : state_type;
BEGIN

-- Sequential block to create state registers and state transitions
PROCESS (clk, reset)
BEGIN
IF reset = '1' THEN
state <= FSM_WAIT;
ELSIF clk'EVENT AND clk = '1' THEN

CASE state IS
WHEN FSM_WAIT =>
IF Protected = '1' THEN
state <= MONITOR;
ELSE
state <= FSM_WAIT;
END IF;

WHEN MONITOR =>
IF Protected = '1' THEN
IF Legitimate = '1' THEN
state <= MONITOR;
ELSE
state <= HALT;
END IF;
ELSE
```

```
state <= FSM_WAIT;
END IF;

WHEN HALT =>
IF Ready = '1' THEN
state <= RESUME;
END IF;

WHEN RESUME =>
state <= MONITOR;
END CASE;
END IF;
END PROCESS;

-- Combinational logic to create outputs for each state
WITH state SELECT
halt_out <= '0' WHEN    FSM_WAIT,
'0'WHEN MONITOR,
'1'WHEN HALT,
'1'WHEN RESUME;
WITH state SELECT
resume_out <= '0'   WHEN  FSM_WAIT,
'0'  WHEN MONITOR,
'0'  WHEN HALT,
'1'  WHEN RESUME;

WITH state SELECT
 current_state   <= 0  WHEN FSM_WAIT,
1  WHEN MONITOR,
```

```
2  WHEN HALT,
3  WHEN RESUME;
END rtl;
```

Appendix B. VHDL MIPS Processor

This appendix holds the VHDL MIPS processor as presented in the Summer 2006 AFIT course CSCE687. This processor provides a good, simple starting point to expand and enhance APHID or other research efforts.

The source code is spread across several files, and the thesis document is not the most useful medium to display it. Nevertheless, for the sake of completeness, here it is. There are two versions, the first one is not pipelined, the second is pipelined.

B.1 *Mips.vhd*

The unpipelined version.

```
-- Top Level Structural Model for MIPS Processor Core
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY MIPS IS

PORT( reset, clock : IN  STD_LOGIC;
-- Output important signals to pins for easy display in Simulator
PC : OUT  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
ALU_result_out, read_data_1_out, read_data_2_out, write_data_out,
      Instruction_out : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Branch_out, Zero_out, Memwrite_out,
Regwrite_out : OUT  STD_LOGIC );
END  MIPS;

ARCHITECTURE structure OF MIPS IS

COMPONENT Ifetch
```

```
        PORT( Instruction : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        PC_plus_4_out  : OUT   STD_LOGIC_VECTOR( 9 DOWNTO 0 );
        Add_result  : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
        Branch  : IN  STD_LOGIC;
        Zero  : IN  STD_LOGIC;
        PC_out  : OUT  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
        clock,reset  : IN  STD_LOGIC );
END COMPONENT;

COMPONENT Idecode
      PORT( read_data_1  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        read_data_2  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        Instruction  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        read_data  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        ALU_result  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        RegWrite, MemtoReg  : IN  STD_LOGIC;
        RegDst  : IN  STD_LOGIC;
        Sign_extend  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        clock, reset : IN  STD_LOGIC );
END COMPONENT;

COMPONENT control
     PORT(  Opcode  : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 );
             RegDst  : OUT  STD_LOGIC;
             ALUSrc  : OUT  STD_LOGIC;
             MemtoReg  : OUT  STD_LOGIC;
             RegWrite  : OUT  STD_LOGIC;
             MemRead  : OUT  STD_LOGIC;
             MemWrite  : OUT  STD_LOGIC;
```

```
              Branch  : OUT  STD_LOGIC;
              ALUop  : OUT  STD_LOGIC_VECTOR( 1 DOWNTO 0 );
              clock, reset : IN  STD_LOGIC );
END COMPONENT;

COMPONENT  Execute
         PORT( Read_data_1  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                Read_data_2  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                Sign_Extend  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                Function_opcode : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 );
                ALUOp  : IN  STD_LOGIC_VECTOR( 1 DOWNTO 0 );
                ALUSrc  : IN  STD_LOGIC;
                Zero  : OUT STD_LOGIC;
                ALU_Result  : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                Add_Result  : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                PC_plus_4  : IN  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
                clock, reset : IN  STD_LOGIC );
END COMPONENT;


COMPONENT dmemory
     PORT( read_data  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         address  : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         write_data  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         MemRead, Memwrite  : IN  STD_LOGIC;
         Clock,reset : IN  STD_LOGIC );
END COMPONENT;

-- declare signals used to connect VHDL components
```

```
SIGNAL PC_plus_4  : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
SIGNAL read_data_1  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_data_2  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Sign_Extend  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Add_result  : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL ALU_result  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_data  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ALUSrc  : STD_LOGIC;
SIGNAL Branch  : STD_LOGIC;
SIGNAL RegDst  : STD_LOGIC;
SIGNAL Regwrite  : STD_LOGIC;
SIGNAL Zero  : STD_LOGIC;
SIGNAL MemWrite  : STD_LOGIC;
SIGNAL MemtoReg  : STD_LOGIC;
SIGNAL MemRead  : STD_LOGIC;
SIGNAL ALUop  : STD_LOGIC_VECTOR(  1 DOWNTO 0 );
SIGNAL Instruction : STD_LOGIC_VECTOR( 31 DOWNTO 0 );

BEGIN
-- copy important signals to output pins for easy
-- display in Simulator
   Instruction_out  <= Instruction;
   ALU_result_out  <= ALU_result;
   read_data_1_out  <= read_data_1;
   read_data_2_out  <= read_data_2;
   write_data_out   <= read_data WHEN MemtoReg = '1' ELSE ALU_result;
   Branch_out  <= Branch;
   Zero_out  <= Zero;
   RegWrite_out  <= RegWrite;
```

```
   MemWrite_out  <= MemWrite;
-- connect the 5 MIPS components
  IFE : Ifetch
PORT MAP ( Instruction  => Instruction,
          PC_plus_4_out  => PC_plus_4,
Add_result  => Add_result,
Branch  => Branch,
Zero  => Zero,
PC_out  => PC,
clock  => clock,
reset  => reset );

   ID : Idecode
    PORT MAP ( read_data_1  => read_data_1,
         read_data_2  => read_data_2,
         Instruction  => Instruction,
         read_data  => read_data,
ALU_result  => ALU_result,
RegWrite  => RegWrite,
MemtoReg  => MemtoReg,
RegDst  => RegDst,
Sign_extend  => Sign_extend,
         clock  => clock,
reset  => reset );


   CTL:   control
PORT MAP (  Opcode  => Instruction( 31 DOWNTO 26 ),
RegDst  => RegDst,
```

```
ALUSrc  => ALUSrc,
MemtoReg  => MemtoReg,
RegWrite  => RegWrite,
MemRead  => MemRead,
MemWrite  => MemWrite,
Branch  => Branch,
ALUop  => ALUop,
                clock  => clock,
reset  => reset );

   EXE:  Execute
    PORT MAP ( Read_data_1  => read_data_1,
              Read_data_2  => read_data_2,
Sign_extend  => Sign_extend,
                Function_opcode => Instruction( 5 DOWNTO 0 ),
ALUOp  => ALUop,
ALUSrc  => ALUSrc,
Zero  => Zero,
                ALU_Result => ALU_Result,
Add_Result  => Add_Result,
PC_plus_4 => PC_plus_4,
                Clock => clock,
Reset => reset );

   MEM:  dmemory
PORT MAP ( read_data  => read_data,
address  => ALU_Result (7 DOWNTO 0),
write_data  => read_data_2,
MemRead  => MemRead,
```

```
Memwrite  => MemWrite,
                clock  => clock,
reset  => reset );
END structure;
```

B.2 MIPS_Piped.vhd

The Pipelined version of MIPS.

```
-- Top Level Structural Model for MIPS Processor Core
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY MIPS IS

PORT( reset, clock : IN  STD_LOGIC;
-- Output important signals to pins for easy display in Simulator
PC : OUT  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
ALU_result_out, read_data_1_out, read_data_2_out, write_data_out,
      Instruction_Fetch : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Instruction_Decode : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Instruction_Execute : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Instruction_Mem : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
WriteBackRegister   : OUT   STD_LOGIC_VECTOR( 4 downto 0  );
Branch_out, Zero_out, Memwrite_out,
Regwrite_out : OUT  STD_LOGIC );
END  MIPS;

ARCHITECTURE structure OF MIPS IS

COMPONENT Ifetch
         PORT(
Instruction : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
PC_plus_4_out  : OUT   STD_LOGIC_VECTOR( 9 DOWNTO 0 );
```

```
        Add_result  : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
        Branch  : IN  STD_LOGIC;
        Zero  : IN  STD_LOGIC;
        PC_out  : OUT  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
        clock,reset  : IN  STD_LOGIC );
END COMPONENT;

COMPONENT Idecode
      PORT( read_data_1  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        read_data_2  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        Instruction  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        read_data  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        ALU_result  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
        RegWrite, MemtoReg  : IN  STD_LOGIC;
        RegDst  : IN  STD_LOGIC;
        Sign_extend  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
write_register_address  :  IN STD_LOGIC_VECTOR( 4 DOWNTO 0 );
reg_dest_mux_output : OUT STD_LOGIC_VECTOR( 4 DOWNTO 0);
        clock, reset : IN  STD_LOGIC );
END COMPONENT;

COMPONENT control
     PORT(  Opcode  : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 );
             RegDst  : OUT  STD_LOGIC;
             ALUSrc  : OUT  STD_LOGIC;
             MemtoReg  : OUT  STD_LOGIC;
             RegWrite  : OUT  STD_LOGIC;
             MemRead  : OUT  STD_LOGIC;
             MemWrite  : OUT  STD_LOGIC;
```

```
              Branch  : OUT  STD_LOGIC;
              ALUop  : OUT  STD_LOGIC_VECTOR( 1 DOWNTO 0 );
              clock, reset : IN  STD_LOGIC );
END COMPONENT;

COMPONENT  Execute
         PORT( Read_data_1  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                Read_data_2  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                Sign_Extend  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                Function_opcode : IN  STD_LOGIC_VECTOR(  5 DOWNTO 0 );
                ALUOp  : IN  STD_LOGIC_VECTOR(  1 DOWNTO 0 );
                ALUSrc  : IN  STD_LOGIC;
                Zero  : OUT STD_LOGIC;
                ALU_Result  : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
                Add_Result  : OUT STD_LOGIC_VECTOR(  7 DOWNTO 0 );
                PC_plus_4  : IN  STD_LOGIC_VECTOR(  9 DOWNTO 0 );
                clock, reset : IN  STD_LOGIC );
END COMPONENT;


COMPONENT dmemory
     PORT( read_data  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         address  : IN  STD_LOGIC_VECTOR(  7 DOWNTO 0 );
         write_data  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         MemRead, Memwrite  : IN  STD_LOGIC;
         Clock,reset : IN  STD_LOGIC );
END COMPONENT;

-- declare signals used to connect VHDL components
```

```
--SIGNAL PC_plus_4  : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
--SIGNAL read_data_1  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
--SIGNAL read_data_2  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
--SIGNAL Sign_Extend  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
--SIGNAL Add_result  : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
--SIGNAL ALU_result  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
--SIGNAL read_data  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
--SIGNAL ALUSrc  : STD_LOGIC;
--SIGNAL Branch  : STD_LOGIC;
SIGNAL RegDst  : STD_LOGIC;
--SIGNAL Regwrite  : STD_LOGIC;
--SIGNAL Zero  : STD_LOGIC;
--SIGNAL MemWrite  : STD_LOGIC;
--SIGNAL MemtoReg  : STD_LOGIC;
--SIGNAL MemRead  : STD_LOGIC;
--SIGNAL ALUop  : STD_LOGIC_VECTOR(  1 DOWNTO 0 );
--SIGNAL Instruction : STD_LOGIC_VECTOR( 31 DOWNTO 0 );


 -- Pipeline Signals
SIGNAL Instruction_IFID_IN : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Instruction_IFID_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Instruction_IDEX_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Instruction_EXMEM_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL PC_plus_4_IFID_IN : STD_LOGIC_VECTOR(9 downto 0);
SIGNAL PC_plus_4_IFID_OUT : STD_LOGIC_VECTOR(9 downto 0);
SIGNAL PC_plus_4_IDEX_IN : STD_LOGIC_VECTOR(9 downto 0);
SIGNAL PC_plus_4_IDEX_OUT : STD_LOGIC_VECTOR(9 downto 0);
SIGNAL Add_result_EXMEM_OUT : STD_LOGIC_VECTOR(  7 DOWNTO 0 );
```

```
SIGNAL M_EXMEM_OUT : STD_LOGIC_VECTOR( 2 downto 0);
SIGNAL Zero_EXMEM_OUT : STD_LOGIC;
SIGNAL read_data_1_IDEX_IN : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_data_2_IDEX_IN : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_data_MEMWB_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ALU_result_MEMWB_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL WB_MEMWB_OUT : STD_LOGIC_VECTOR( 1 DOWNTO 0 );
SIGNAL Sign_extend_IDEX_IN : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL reg_dest_mux_output_MEMWB_OUT: STD_LOGIC_VECTOR(4 downto 0);
SIGNAL reg_dest_mux_output_IDEX_IN  : STD_LOGIC_VECTOR(4 downto 0);
SIGNAL EX_IDEX_IN : STD_LOGIC_VECTOR(2 downto 0);
SIGNAL WB_IDEX_IN : STD_LOGIC_VECTOR(1 downto 0);
SIGNAL M_IDEX_IN  : STD_LOGIC_VECTOR(2 downto 0);
SIGNAL read_data_1_IDEX_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_data_2_IDEX_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Sign_extend_IDEX_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL EX_IDEX_OUT  : STD_LOGIC_VECTOR( 2 DOWNTO 0 );
SIGNAL Zero_EXMEM_IN : std_logic;
SIGNAL ALU_Result_EXMEM_IN : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Add_Result_EXMEM_IN : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL read_data_MEMWB_IN :  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ALU_Result_EXMEM_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_data_2_EXMEM_OUT : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL reg_dest_mux_output_IDEX_OUT : STD_LOGIC_VECTOR(4 downto 0);
SIGNAL reg_dest_mux_output_EXMEM_OUT : STD_LOGIC_VECTOR(4 downto 0);
SIGNAL M_IDEX_OUT    :  STD_LOGIC_VECTOR(2 downto 0);
SIGNAL WB_IDEX_OUT   :  STD_LOGIC_VECTOR(1 downto 0);
SIGNAL WB_EXMEM_OUT   :  STD_LOGIC_VECTOR(1 downto 0);
```

```
BEGIN
-- copy important signals to output pins for easy
-- display in Simulator
   Instruction_Fetch <= Instruction_IFID_IN;
   Instruction_Decode <= Instruction_IFID_OUT;
   Instruction_Execute <= Instruction_IDEX_OUT;
   Instruction_Mem <= Instruction_EXMEM_OUT;
   ALU_result_out   <= ALU_Result_EXMEM_IN;
   read_data_1_out   <= read_data_1_IDEX_OUT;
   read_data_2_out   <= read_data_2_IDEX_OUT;
   write_data_out    <= read_data_MEMWB_OUT WHEN WB_MEMWB_OUT(1) = '1' ELSE ALU_resul
   Branch_out   <= M_EXMEM_OUT(0);
   Zero_out   <= Zero_EXMEM_OUT;
   RegWrite_out   <= WB_IDEX_IN(1);
   MemWrite_out   <= M_IDEX_IN(1);
   WriteBackRegister <= reg_dest_mux_output_MEMWB_OUT;
--Branchne <= M_EXMEM_OUT(2);

-- connect the 5 MIPS components
  IFE : Ifetch
PORT MAP ( Instruction  => Instruction_IFID_IN,
          PC_plus_4_out  => PC_plus_4_IFID_IN,
Add_result  => Add_result_EXMEM_OUT,
Branch  => M_EXMEM_OUT(0),
Zero  => Zero_EXMEM_OUT,
PC_out  => PC,
clock  =>  clock,
reset  => reset );
```

```
   ID : Idecode
    PORT MAP ( read_data_1  => read_data_1_IDEX_IN,
         read_data_2  => read_data_2_IDEX_IN,
         Instruction  => Instruction_IFID_OUT,
         read_data  => read_data_MEMWB_OUT,
ALU_result  => ALU_result_MEMWB_OUT,
RegWrite  => WB_MEMWB_OUT(1),
MemtoReg  => WB_MEMWB_OUT(0),
RegDst  => RegDst,
Sign_extend  => Sign_extend_IDEX_IN,
         clock  =>  clock,
write_register_address => reg_dest_mux_output_MEMWB_OUT,
reg_dest_mux_output => reg_dest_mux_output_IDEX_IN,
reset  => reset );


   CTL:   control
PORT MAP (  Opcode  => Instruction_IFID_OUT(31 DOWNTO 26),
RegDst  => RegDst,
ALUSrc  => EX_IDEX_IN(2),
MemtoReg  => WB_IDEX_IN(0),
RegWrite  => WB_IDEX_IN(1),
MemRead  => M_IDEX_IN(2),
MemWrite  => M_IDEX_IN(1),
Branch  => M_IDEX_IN(0),
ALUop  => EX_IDEX_IN(1 downto 0),
                clock  =>  clock,
```

```
reset  => reset );

   EXE:  Execute
    PORT MAP ( Read_data_1  => read_data_1_IDEX_OUT,
              Read_data_2  => read_data_2_IDEX_OUT,
Sign_extend  => Sign_extend_IDEX_OUT,
                Function_opcode => Sign_extend_IDEX_OUT( 5 DOWNTO 0 ),
ALUOp  => EX_IDEX_OUT(1 downto 0),
ALUSrc  => EX_IDEX_OUT(2),
Zero  => Zero_EXMEM_IN,
                ALU_Result => ALU_Result_EXMEM_IN,
Add_Result  => Add_Result_EXMEM_IN,
PC_plus_4 => PC_plus_4_IDEX_OUT,
                Clock =>  clock,
Reset => reset );

   MEM:  dmemory
PORT MAP ( read_data  => read_data_MEMWB_IN,
address  => ALU_Result_EXMEM_OUT (7 DOWNTO 0),
write_data  => read_data_2_EXMEM_OUT,
MemRead  => M_EXMEM_OUT(1),
Memwrite  => M_EXMEM_OUT(0),
                clock  =>  clock,
reset  => reset );

Pipeline_Updates: PROCESS ( clock, reset)
BEGIN
  if reset='1' then
              Instruction_IFID_OUT <=  (others=>'0');
```

```
          elsif  clock'event and  clock='1' then

Instruction_IFID_OUT  <= Instruction_IFID_IN;
Instruction_IDEX_OUT  <= Instruction_IFID_OUT;
Instruction_EXMEM_OUT <= Instruction_IDEX_OUT;

-- reg dest mux output trail
reg_dest_mux_output_IDEX_OUT <= reg_dest_mux_output_IDEX_IN;
reg_dest_mux_output_EXMEM_OUT <= reg_dest_mux_output_IDEX_OUT;
reg_dest_mux_output_MEMWB_OUT <= reg_dest_mux_output_EXMEM_OUT;

-- pc_plus4_ trail
PC_plus_4_IFID_OUT <= PC_plus_4_IFID_IN;
PC_plus_4_IDEX_OUT <= PC_plus_4_IFID_OUT;

-- Control signals

M_IDEX_OUT   <= M_IDEX_IN;
M_EXMEM_OUT  <= M_IDEX_OUT;
WB_IDEX_OUT  <= WB_IDEX_IN;
WB_EXMEM_OUT <= WB_IDEX_OUT;
WB_MEMWB_OUT <= WB_EXMEM_OUT;
EX_IDEX_OUT  <= EX_IDEX_IN;

--
   read_data_2_IDEX_OUT <= read_data_2_IDEX_IN;
read_data_2_EXMEM_OUT <= read_data_2_IDEX_OUT;
read_data_1_IDEX_OUT <= read_data_1_IDEX_IN;
```

```
  --
  Sign_extend_IDEX_OUT <=Sign_extend_IDEX_IN;

--
   ALU_Result_MEMWB_OUT <= ALU_Result_EXMEM_OUT;
   ALU_Result_EXMEM_OUT <= ALU_Result_EXMEM_IN;

--
   read_data_MEMWB_OUT <= read_data_MEMWB_IN;
--
Add_result_EXMEM_OUT <= Add_result_EXMEM_IN;
Zero_EXMEM_OUT <= Zero_EXMEM_IN;

            end if;


END PROCESS;
END structure;
```

B.3 control.vhd

```
-- control module (implements MIPS control unit)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY control IS
   PORT(
Opcode  : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 );
RegDst  : OUT  STD_LOGIC;
ALUSrc  : OUT  STD_LOGIC;
MemtoReg  : OUT  STD_LOGIC;
RegWrite  : OUT  STD_LOGIC;
MemRead  : OUT  STD_LOGIC;
MemWrite  : OUT  STD_LOGIC;
Branch  : OUT  STD_LOGIC;
ALUop  : OUT  STD_LOGIC_VECTOR( 1 DOWNTO 0 );
clock, reset : IN  STD_LOGIC );

END control;

ARCHITECTURE behavior OF control IS

SIGNAL  R_format, Lw, Sw, Beq, Bne  : STD_LOGIC;

BEGIN
-- Code to generate control signals using opcode bits
R_format  <=  '1'  WHEN  Opcode = "000000"  ELSE '0';
```

```
Lw          <=  '1'  WHEN  Opcode = "100011"  ELSE '0';
  Sw          <=  '1'  WHEN  Opcode = "101011"  ELSE '0';
    Beq         <=  '1'  WHEN  Opcode = "000100"  ELSE '0';
Bne <=  '1'  WHEN  Opcode = "000101"  ELSE '0';  -- Added for BNE
   RegDst     <=  R_format;
  ALUSrc   <=  Lw OR Sw;
MemtoReg  <=  Lw;
   RegWrite  <=  R_format OR Lw;
   MemRead  <=  Lw;
    MemWrite  <=  Sw;
  Branch      <=  Beq OR Bne;  -- Added BNE
ALUOp( 1 )  <=  R_format;
ALUOp( 0 )  <=  Beq OR Bne;  -- Added BNE

   END behavior;
```

B.4 Ifetch.vhd

```
-- Ifetch module (provides the PC and instruction
--memory for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY Ifetch IS
PORT( SIGNAL Instruction  : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         SIGNAL PC_plus_4_out  : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 );
         SIGNAL Add_result  : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         SIGNAL Branch  : IN  STD_LOGIC;
         SIGNAL Zero  : IN  STD_LOGIC;
       SIGNAL PC_out  : OUT STD_LOGIC_VECTOR( 9 DOWNTO 0 );
         SIGNAL clock, reset  : IN  STD_LOGIC);
END Ifetch;

ARCHITECTURE behavior OF Ifetch IS
SIGNAL PC, PC_plus_4   : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
SIGNAL next_PC, Mem_Addr : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN
--ROM for Instruction Memory
inst_memory: altsyncram

GENERIC MAP (
operation_mode => "ROM",
```

```
width_a => 32,
widthad_a => 8,
lpm_type => "altsyncram",
outdata_reg_a => "UNREGISTERED",
init_file => "program.mif",
intended_device_family => "Cyclone"
)
PORT MAP (
clock0     => clock,
address_a  => Mem_Addr,
q_a  => Instruction );
-- Instructions always start on word address - not byte
PC(1 DOWNTO 0) <= "00";
-- copy output signals - allows read inside module
PC_out  <= PC;
PC_plus_4_out  <= PC_plus_4;
-- send address to inst. memory address register
Mem_Addr <= Next_PC;
-- Adder to increment PC by 4
       PC_plus_4( 9 DOWNTO 2 )  <= PC( 9 DOWNTO 2 ) + 1;
        PC_plus_4( 1 DOWNTO 0 )  <= "00";
-- Mux to select Branch Address or PC + 4
Next_PC  <= X"00" WHEN Reset = '1' ELSE
Add_result  WHEN ( ( Branch = '1' ) AND ( Zero = '1' ) )
ELSE   PC_plus_4( 9 DOWNTO 2 );
PROCESS
BEGIN
WAIT UNTIL ( clock'EVENT ) AND ( clock = '1' );
IF reset = '1' THEN
```

```
   PC( 9 DOWNTO 2) <= "00000000" ;
ELSE
   PC( 9 DOWNTO 2 ) <= next_PC;
END IF;
END PROCESS;
END behavior;
```

B.5 Idecode.vhd

```
--  Idecode module (implements the register file for
LIBRARY IEEE;  -- the MIPS computer)
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Idecode IS
  PORT( read_data_1 : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
read_data_2 : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Instruction : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
read_data  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
ALU_result : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
RegWrite  : IN  STD_LOGIC;
MemtoReg  : IN  STD_LOGIC;
RegDst  : IN  STD_LOGIC;
Sign_extend : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
clock,reset : IN  STD_LOGIC );
END Idecode;


ARCHITECTURE behavior OF Idecode IS
TYPE register_file IS ARRAY ( 0 TO 31 ) OF STD_LOGIC_VECTOR( 31 DOWNTO 0 );

SIGNAL register_array : register_file;
SIGNAL write_register_address  : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
SIGNAL write_data : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL read_register_1_address : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
SIGNAL read_register_2_address : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
```

```
SIGNAL write_register_address_1 : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
SIGNAL write_register_address_0 : STD_LOGIC_VECTOR( 4 DOWNTO 0 );
SIGNAL Instruction_immediate_value : STD_LOGIC_VECTOR( 15 DOWNTO 0 );

BEGIN
read_register_1_address  <= Instruction( 25 DOWNTO 21 );
    read_register_2_address  <= Instruction( 20 DOWNTO 16 );
    write_register_address_1 <= Instruction( 15 DOWNTO 11 );
    write_register_address_0  <= Instruction( 20 DOWNTO 16 );
    Instruction_immediate_value <= Instruction( 15 DOWNTO 0 );

-- Read Register 1 Operation
read_data_1 <= register_array(
      CONV_INTEGER( read_register_1_address ) );

-- Read Register 2 Operation
read_data_2 <= register_array(
      CONV_INTEGER( read_register_2_address ) );

-- Mux for Register Write Address
     write_register_address <= write_register_address_1
WHEN RegDst = '1'   ELSE write_register_address_0;

-- Mux to bypass data memory for Rformat instructions
write_data <= ALU_result( 31 DOWNTO 0 )
WHEN ( MemtoReg = '0' )  ELSE read_data;

-- Sign Extend 16-bits to 32-bits
     Sign_extend <= X"0000" & Instruction_immediate_value
```

```
WHEN Instruction_immediate_value(15) = '0'
ELSE X"FFFF" & Instruction_immediate_value;

PROCESS
BEGIN
WAIT UNTIL clock'EVENT AND clock = '1';
IF reset = '1' THEN
-- Initial register values on reset are register = reg#
-- use loop to automatically generate reset logic
-- for all registers
FOR i IN 0 TO 31 LOOP
register_array(i) <= CONV_STD_LOGIC_VECTOR( i, 32 );
  END LOOP;
-- Write back to register - don't write to register 0
   ELSIF RegWrite = '1' AND write_register_address /= 0 THEN
      register_array( CONV_INTEGER( write_register_address)) <= write_data;
END IF;
END PROCESS;
END behavior;
```

B.6 Execute.vhd

```
--  Execute module (implements the data ALU and Branch Address Adder
--  for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY  Execute IS
PORT( Read_data_1  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Read_data_2  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Sign_extend  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Function_opcode : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 );
ALUOp  : IN  STD_LOGIC_VECTOR( 1 DOWNTO 0 );
ALUSrc  : IN  STD_LOGIC;
Zero  : OUT STD_LOGIC;
ALU_Result  : OUT STD_LOGIC_VECTOR( 31 DOWNTO 0 );
Add_Result  : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
PC_plus_4  : IN  STD_LOGIC_VECTOR( 9 DOWNTO 0 );
clock, reset : IN  STD_LOGIC );
END Execute;

ARCHITECTURE behavior OF Execute IS
SIGNAL Ainput, Binput  : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL ALU_output_mux : STD_LOGIC_VECTOR( 31 DOWNTO 0 );
SIGNAL Branch_Add  : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL ALU_ctl : STD_LOGIC_VECTOR( 2 DOWNTO 0 );
BEGIN
Ainput <= Read_data_1;
```

```
-- ALU input mux
Binput <= Read_data_2
WHEN ( ALUSrc = '0' )
   ELSE  Sign_extend( 31 DOWNTO 0 );

-- Generate ALU control bits
ALU_ctl( 0 ) <= ( Function_opcode( 0 ) OR Function_opcode( 3 ) ) AND ALUOp(1 );
ALU_ctl( 1 ) <= ( NOT Function_opcode( 2 ) ) OR (NOT ALUOp( 1 ) );
ALU_ctl( 2 ) <= ( Function_opcode( 1 ) AND ALUOp( 1 )) OR ALUOp( 0 );

-- Generate Zero Flag
Zero <= '1'
WHEN ( ALU_output_mux( 31 DOWNTO 0 ) = X"00000000"  )
ELSE '0';

-- Select ALU output
ALU_result <= X"0000000" & B"000"  & ALU_output_mux( 31 ) -- When CTL is 111
WHEN  ALU_ctl = "111"
ELSE   ALU_output_mux( 31 DOWNTO 0 );

-- Adder to compute Branch Address
Branch_Add <= PC_plus_4( 9 DOWNTO 2 ) +  Sign_extend( 7 DOWNTO 0 ) ;
Add_result  <= Branch_Add( 7 DOWNTO 0 );

PROCESS ( ALU_ctl, Ainput, Binput )
BEGIN
-- Select ALU operation
  CASE ALU_ctl IS
-- ALU performs ALUresult = A_input AND B_input
```

```
WHEN "000"  =>ALU_output_mux  <= Ainput AND Binput;

-- ALU performs ALUresult = A_input OR B_input
      WHEN "001"  =>ALU_output_mux  <= Ainput OR Binput;

-- ALU performs ALUresult = A_input + B_input
  WHEN "010"  =>ALU_output_mux  <= Ainput + Binput;

-- ALU performs --Unwritten  code?
    WHEN "011"  =>ALU_output_mux <= X"00000000";

-- ALU performs --Unwritten  code?
    WHEN "100"  =>ALU_output_mux  <= X"00000000";

-- ALU performs --Unwritten  code??
    WHEN "101"  =>ALU_output_mux  <= X"00000000";

-- ALU performs ALUresult = A_input -B_input
    WHEN "110"  =>ALU_output_mux  <= Ainput - Binput;

-- ALU performs SLT
     WHEN "111"  =>ALU_output_mux  <= Ainput - Binput ;

WHEN OTHERS =>ALU_output_mux  <= X"00000000" ;
   END CASE;
  END PROCESS;
END behavior;
```

B.7 *Dmemory.vhd*

This is the data memory module. The current implementation uses Altera specific memory devices. It will need to be rewritten for Xilinx specific memories.

```
--  Dmemory module (implements the data
--  memory for the MIPS computer)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY dmemory IS
PORT( read_data  : OUT  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
         address  : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         write_data  : IN  STD_LOGIC_VECTOR( 31 DOWNTO 0 );
    MemRead, Memwrite  : IN  STD_LOGIC;
            clock,reset : IN  STD_LOGIC );
END dmemory;

ARCHITECTURE behavior OF dmemory IS
SIGNAL write_clock : STD_LOGIC;
BEGIN
data_memory : altsyncram
GENERIC MAP  (
operation_mode => "SINGLE_PORT",
width_a => 32,
```

```
widthad_a => 8,
lpm_type => "altsyncram",
outdata_reg_a => "UNREGISTERED",
init_file => "dmemory.mif",
intended_device_family => "Cyclone"
)
PORT MAP (
wren_a => memwrite,
clock0 => write_clock,
address_a => address,
data_a => write_data,
q_a => read_data );
-- Load memory address register with write clock
write_clock <= NOT clock;
END behavior;
```

B.8 Control.vhd

```
-- control module (implements MIPS control unit)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY control IS
   PORT(
Opcode  : IN  STD_LOGIC_VECTOR( 5 DOWNTO 0 );
RegDst  : OUT  STD_LOGIC;
ALUSrc  : OUT  STD_LOGIC;
MemtoReg  : OUT  STD_LOGIC;
RegWrite  : OUT  STD_LOGIC;
MemRead  : OUT  STD_LOGIC;
MemWrite  : OUT  STD_LOGIC;
Branch  : OUT  STD_LOGIC;
ALUop  : OUT  STD_LOGIC_VECTOR( 1 DOWNTO 0 );
clock, reset : IN  STD_LOGIC );

END control;

ARCHITECTURE behavior OF control IS

SIGNAL  R_format, Lw, Sw, Beq, Bne  : STD_LOGIC;

BEGIN
-- Code to generate control signals using opcode bits
R_format  <=  '1'  WHEN  Opcode = "000000"  ELSE '0';
```

```
Lw          <=  '1'  WHEN  Opcode = "100011"  ELSE '0';
  Sw          <=  '1'  WHEN  Opcode = "101011"  ELSE '0';
    Beq         <=  '1'  WHEN  Opcode = "000100"  ELSE '0';
Bne <=  '1'  WHEN  Opcode = "000101"  ELSE '0';  -- Added for BNE
   RegDst     <=  R_format;
  ALUSrc   <=  Lw OR Sw;
MemtoReg  <=  Lw;
   RegWrite  <=  R_format OR Lw;
   MemRead  <=  Lw;
    MemWrite  <=  Sw;
  Branch      <=  Beq OR Bne;  -- Added BNE
ALUOp( 1 )  <=  R_format;
ALUOp( 0 )  <=  Beq OR Bne;  -- Added BNE

   END behavior;
```

B.9 Decode.vhd

```
--  Dmemory module (implements the data
-- memory for the MIPS computer)

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY dmemory IS
PORT( read_data  : OUT  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         address  : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         write_data  : IN  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
    MemRead, Memwrite  : IN  STD_LOGIC;
             clock,reset : IN  STD_LOGIC );
END dmemory;

ARCHITECTURE behavior OF dmemory IS
SIGNAL lpm_write : STD_LOGIC;
BEGIN
data_memory: lpm_ram_dq
GENERIC MAP (
lpm_widthad  =>  8,
lpm_outdata  =>  "UNREGISTERED",
lpm_indata  =>  "REGISTERED",
lpm_address_control  =>  "UNREGISTERED",
-- Reads in mif file for initial data memory values
```

```
lpm_file  =>  "dmemory.mif",
lpm_width  =>  8 )

      PORT MAP (
data =>write_data,  address  =>  address,
we  =>lpm_write,inclock  =>  clock,      q  =>  read_data );
-- delay lpm write enable to ensure stable address and data
lpm_write <= memwrite AND ( NOT clock );
END behavior;
```

Bibliography

1. Altera. "Alera University Program", Februray 2007. URL `http://www.altera.com/education/univ/unv-index.html`.
2. Arora, Divya, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. "Secure Embedded Processing through Hardware-assisted Run-time Monitoring". *Design, Automation and Test in Europe Conference and Exhibition*. 2005.
3. Axelsson, Stefan. *Intrusion Detection Systems: A Survey and Taxonomy*. Technical Report 99-15, Chalmers Univ., March 2000. URL `citeseer.nj.nec.com/axelsson00intrusion.html`.
4. Bajikar, S. "Trusted Platform Module (TPM) based Security on Notebook PCs-White Paper". *Intel Corporation–Mobile Platforms Group*, 2002.
5. Balasubramaniyan, J. S., J. O. Garcia-Fernandez, D. Isacoff, Eugene H. Spafford, and Diego Zamboni. "An Architecture for Intrusion Detection Using Autonomous Agents". *ACSAC*, 13–24. 1998. URL `citeseer.ist.psu.edu/balasubramaniyan98architecture.html`.
6. Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. "Thorough static analysis of device drivers". *EuroSys06: European Systems Conference*, 73–85, 2006.
7. Basili, V.R. and B.T. Perricone. "SOFTWARE ERRORS AND COMPLEXITY: AN EMPIRICAL INVESTIGATION". *Communications of the ACM*, 27(1):42–49, 1984.
8. Beale, Jay, Andrew Baker, Brain Caswell, and Mike Poor. *Snort 2.1 Intrusion Detection*. Syngress, 2004.
9. Bell, D. E. and L. J. LaPadula. *Secure Computer Systems: Mathematical Foundations and Model*. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.
10. Bishop, Matt. *Computer Security, Art and Science*. Addison Wesley, 2003.
11. CACI. "Computer Security Threats: Malicious Threats". Web Page, 2007. URL `http://www.caci.com/business/ia/threats.html`.
12. Candela Technologies. "LANforge v4.4.9 - Candela Technologies", 2007. URL `http://www.candelatech.com/`.
13. Computer Emerengy Response Team (CERT). "CERT/CC Statistics 1988-2006". URL `www.cert.org/stats/cert_stats.html`. www.cert.org/stats/cert_stats.html.

14. DARPA. "RFC 791: Internet Protocol - DARPA Internet Program Protocol Specification". Web Page, September 1981. URL http://tools.ietf.org/html/rfc791. Accessed Feb 10, 2007.

15. Deering, S. and R. Hinden. "Internet Protocol, Version 6 (IPv6) Specification". Web Page, December 1998. URL http://tools.ietf.org/html/rfc2460. Accessed Feb 19, 2007.

16. Denning, D.E. "An Intrusion-Detection Model". *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

17. digg. "Digg.com". URL http://www.digg.com/.

18. Evers, Joris. "Internet backbone suffers suspected attack". web article, Feb 2007. URL http://news.zdnet.co.uk/security/0,1000000189,39285836,00.htm.

19. F-Secure Corp. "F-Secure Visrus Descriptions: Slammer". Http://www.f-secure.com/v-descs/mssqlm.shtml.

20. Fark. "Fark.com", Feb 2007. URL http://www.fark.com.

21. Hamblen, James O., Tyson S. Hall, and Michael D. Furman. *Rapid Prototyping of Digital Systems, 3rd Ed.* Springer, 2005.

22. Hennessy, John L. and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kauffman, 4 edition, 2006.

23. Hoglund, Greg and Gary McGraw. *Exploiting Software-How to Break Code*. Addison Wesley, 2004.

24. Kuperman, Benjamin A. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources*. Ph.D. thesis, Purdue University, West Lafayette, IN, 08 2004. CERIAS TR 2004-26.

25. Kurose, J. and K. Ross. *Computer Networking*. Addison Wesley, 2005.

26. Lee, Ruby B., David K. Karig, John P. McGreggor, and Zhijie Shi. *Security in Pervasive Computing*, chapter Enlisting Hardware Architecture to Thwart Malicious Code Injection, 237–252. Springer Berlin, 2003.

27. Leyden, John. "Security firm punctures Vista's Patchguard". URL http://www.regdeveloper.co.uk/2006/10/27/patchguard_row_analysis/.

28. Manadhata, Pratyusa and Jeannette M. Wing. *Measuring a System's Attack Surface*. Technical Report CMU-CS-04-102, Canegie Mellon University, Pittsburgh, PA, January 2004.

29. Medley, Douglas. *Virtualization Technology Applied to Rootkit Defense*. Master's thesis, Air Force Institute of Technology, 2007.

30. Montminy, D. *Using Relocatable Bitstreams for Fault Tolerance*. Master's thesis, Air Force Institute of Technology, 2007.

31. Mott, Stephen. *Exploring Hardware-based Primitives to Enhance Parallel Security Monitoring in a Novel Computing Architecture*. Master's thesis, Air Force Institute of Technology, 2007.

32. Nerenberg, Daniel. *A Study of Rootkit Stealth Techniques and Associated Detection Methods*. Master's thesis, Air Force Institute of Technology, 2007.

33. NIST and SEMATECH. "NIST/SEMATECH e-Handbook of Statistical Methods". e-Book Web Page, June 2003. URL `http://www.itl.nist.gov/div898/handbook`.

34. Ozdoganoglu, H., CE Brodley, TN Vijaykumar, B.A. Kuperman, and A. Jalote. "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address". *Purdue University TR-ECE*, 03–13.

35. Peng, T., C. Leckie, and K. Ramamohanarao. "Protection from distributed denial of service attacks using history-based IP filtering". *Communications, 2003. ICC'03. IEEE International Conference on*, 1, 2003.

36. Petroni, Nick L., Timothy Fraser, Jesus Molina, and William A. Arbaugh. "Copilot - A Coprocessor-based Kernel Runtime Integrity Monitor". *Proceedings of the 13th USENIX Security Symposium*. USENIX, August 2004.

37. Rogin, Josh. "Cyber officials: Chinese hackers attack 'anything and everything"'. web article, Feb 2007. URL `http://www.fcw.com/article97658-02-13-07-Web&printLayout`.

38. Rubini, Alessandro and Jonathan Corbet. *Linux Device Drivers, 2nd Ed.* O'Reilly, 2001.

39. Slashdot. "Slashdot.org", Feb 2007. URL `http://slashdot.org`.

40. Smith, Douglas J. *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs adn FPGAs using VHDL or Verilog*. Doone Publishings, 1996, 2001.

41. Snort.org. "Snort - the de facto standard for intrusion detection/prevention". Web Page. URL `http://www.snort.org/`.

42. Swift, M.M., B.N. Bershad, and H.M. Levy. "Improving the reliability of commodity operating systems". *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, 2005.

43. Tanenbaum, AS, JN Herder, and H. Bos. "Can we make operating systems reliable and secure?" *Computer*, 39(5):44–51, 2006.

44. Wikipedia. "Slashdot Effect", Feb 2007. URL `http://en.wikipedia.org/wiki/Slashdot_effect`.

45. Williams, Paul D. *CuPIDS: Co-Processor based Intrusion Detection System*. Ph.D. thesis, Purdue University, West Lafayette, IN, August 2005.

46. Xilinx. “Content Addressable Memory v5.1”. Datasheet, Nov 2004. URL `http://www.xilinx.com/ipcenter/catalog/logicore/docs/cam.pdf`.

47. Xilinx. “Virtex-4 Configuration Guide”, 2004.

48. Zamboni, Diego. *Doing Intrusion Detection Using Embedded Sensors*. Ph.D. thesis. URL `https://www.cerias.purdue.edu/papers/archive/2000-21.pdf`. CERIAS TR 2000-21.

49. Zhang, Tao, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. “Anomalous Path Detection with Hardware Support”. *CASES’05*. September 2005.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*