November 2015

Securing Network Processors with Hardware Monitors

Kekai Hu

SECURING NETWORK PROCESSORS WITH HARDWARE MONITORS

A Dissertation Presented

by

KEKAI HU

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2015

Electrical and Computer Engineering

SECURING NETWORK PROCESSORS WITH HARDWARE MONITORS

A Dissertation Presented

by

KEKAI HU

Approved as to style and content by:

Russell G. Tessier, Chair

Tilman Wolf, Member

Weibo Gong, Member

Arjun Guha, Member

C. V. Hollot, Department Chair
Electrical and Computer Engineering

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest appreciation to my advisor Prof. Russell Tessier. Thank you for your kindness to bring me to the RCG family during the toughest time in my student life. I can't image that I could still finish my PhD without your help, you saved me. I have learnt so much during my PhD life. Not only the academic knowledge, but also the responsible, diligent, and enthusiastic attitude to work and life. I sincerely thank you for the opportunities and confidence you have given me all these years.

It's a great experience and privilege for me to be able to work with an accomplished researcher as Prof. Tilman Wolf. As my co-adviser, Prof. Wolf has given me valuable guidances on both my research and career development. I sincerely thank my committee members, Prof. Weibo Gong and Prof. Arjun Guha for providing constructive guidelines and brilliant comments to make this dissertation better.

I want to give special thanks to my family, my mom, and my dad. Your continuous support and encouragement are very important for me all the time. I would also like to thank all my RCG lab mates - Jia Zhao, Vishwas Vijayendra, Gayatri Prabhu, Akilesh Krishnamurthy, Murtaza Merchant, Harikrishnan, Ben Bovee, Cory Gorman, Xiaobin Liu, Justin Lu, Meha Kainth, Tedy Thomas, Sandesh Virupaksha, and Shrikant Vyas for making it a wonderful and enjoyable experience.

The six wonderful years of my life in the beautiful college town of Amherst will always be my precious memory in my life.

ABSTRACT

SECURING NETWORK PROCESSORS WITH HARDWARE MONITORS

SEPTEMBER 2015

KEKAI HU

B.Sc., WUHAN UNIVERSITY

M.Sc., WUHAN UNIVERSITY

Directed by: Professor Russell G. Tessier

As an essential part of modern society, the Internet has fundamentally changed our lives during the last decade. Novel applications and technologies, such as online shopping, social networking, cloud computing, mobile networking, etc, have sprung up at an astonishing pace. These technologies not only influence modern life styles but also impact Internet infrastructure. Numerous new network applications and services require better programmability and flexibility for network devices, such as routers and switches. Since traditional fixed function network routers based on application specific integrated circuits (ASICs) have difficulty keeping pace with the growing demands of next-generation Internet applications, there is an ongoing shift in the industry toward implementing network devices using programmable network processors (NPs).

While network processors offer great benefits in terms of flexibility, their reprogrammable nature exposes potential security risks. Similar to network end-systems,

such as general-purpose computers, software-based network processors have security vulnerabilities that can be attacked remotely. Recent research has shown that a new type of data plane attack is able to modify the functionality of a network processor and cause a denial-of-service (DoS) attack by sending a single malformed UDP packet. Since this attack relies solely on data plane access and does not need access to the control plane, it can be particularly difficult to control.

Hardware security monitors have been introduced to identify and eliminate these malicious packets before they can propagate and cause devastating effects in the network. However, previous work on hardware monitors only focus on single core systems with static (or very slowly changing) workloads. In network processors that use up to hundreds of parallel processor cores and have processing workloads that can change dynamically based on the network traffic, the realization of a complete multicore hardware monitoring system remains a critical challenge. Our research work in this thesis provides a comprehensive solution to this problem.

Our first contribution is the design and prototype implementation of a Scalable Hardware Monitoring Grid (SHMG). This scalable architecture balances area cost and performance overhead by using a clustered approach for multicore NP systems. In order to adapt to dynamically changing network traffic, a resource reallocation algorithm is designed to reassign the processing resources in SHMG to different network applications at runtime. An evaluation of the prototype SHMG on an Altera DE4 board demonstrates low resource and performance overheads. The functionality and performance of a runtime resource reallocation algorithm are tested using a simulation environment.

A second significant contribution of this work is a network system-level security solution for multicore network processors with hardware monitors. It addresses two key problems: (1) how to securely manage and reprogram processor cores and monitors in a deployed router in the network, and (2) how to prevent the large number of

identical router devices in the network from an attack that can circumvent one specific monitoring system and lead to Internet-scale failures. A Secure Dynamic Multicore Hardware Monitoring System (SDMMon) is designed based on cryptographic principles and suitable key management to ensure the secure installation of processor binaries and monitor graphs. We present a Merkle tree based parameterizable high-performance hash function that can be configured to perform a variety of functions in different devices via a 32-bit configuration parameter. A prototype system composed of both the SDMMon and the parameterizable hash is implemented and evaluated on an Altera DE4 board.

Finally, a fully-functional, comprehensive Multicore NP Security Platform, which integrates both the SHMG and the SDMMon security features, has been implemented on an Altera DE5 board.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Trends and Security Challenges in Internet Systems

The Internet is a critical infrastructure component in today's society. The profound impact of the Internet covers almost every aspect of modern life such as personal communication, business transactions, entertainment, digital government, etc. This impact is likely to continue to increase in the years ahead.

As the Internet evolves further, the underlying network infrastructure needs fundamental advancement to catch up with the growing requirements of the future Internet. Programmability of network devices such as routers and switches is a critical feature of the future Internet infrastructure.

In the classic TCP/IP network model (Figure 1.1), a network is divided into five layers: the application layer, transport layer, network layer, data link layer and physical layer. Various network protocols have been proposed and deployed in this hierarchy, and different network devices are implemented in different layers. For example, network end-systems such as general-purpose computers and workstations work in the application layer, network routers work in the network layer and network switches work in data link layer. Traditionally, numerous network protocols and applications (e.g., File Transfer Protocol (FTP) [73], Hypertext Transfer Protocol (HTTP) [39], Simple Mail Transfer Protocol (SMTP) [62], etc) are deployed in the application layer because of its software programmable nature. The network layer, on the other hand, has less protocol choices and has remained virtually unchanged for decades [10]. As a result, Internet Protocol (IP) [72], the dominant protocol in the

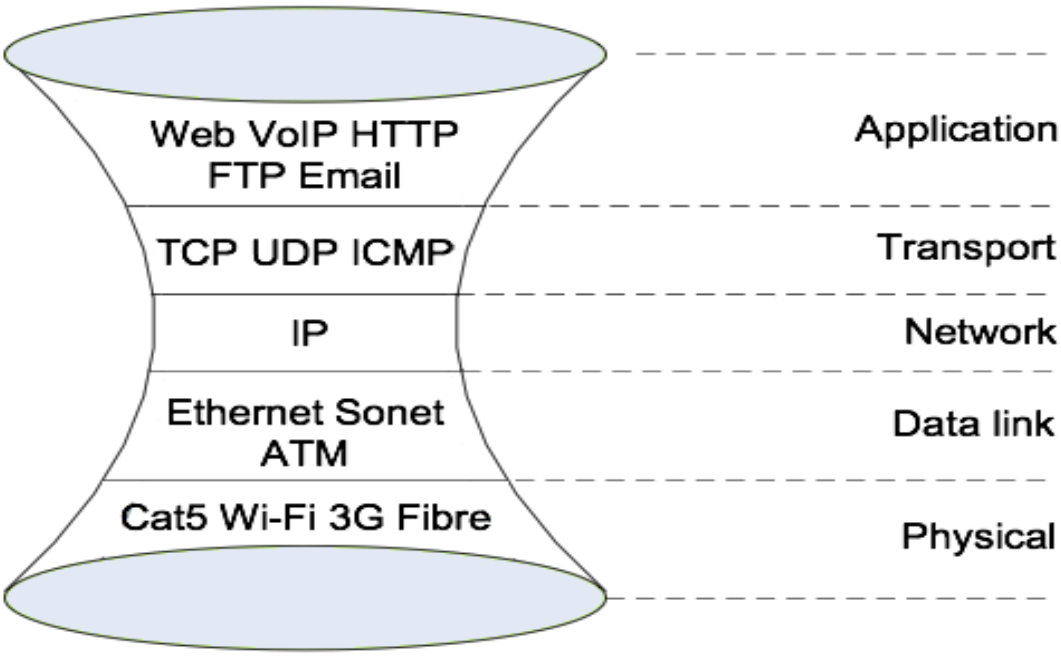

Figure 1.1: Layered networking in the Internet [10]

network layer, is implemented with application-specific integrated circuits (ASIC) in traditional routers [79]. Although the cost of developing ASICs is expensive, ASICs are able to achieve the necessary performance for multi-Gigabit per second traffic forwarding. ASIC routers are typically widely distributed in networks once designed.

In recent years, emerging novel network concepts and architectures, such as software defined network (SDN) [8,66] and network virtualization [24], have significantly expanded the protocol choices in the network layer. A large number of new tasks including security checks [34], data filtering [23], traffic management [18], resource management [88], etc, are implemented in the network layer to augment the basic functions of the Internet Protocol [17]. Network routers need to implement packet processing and data forwarding on a broad scale. However, ASIC-based routers typically have fixed functions that cannot be easily changed once designed, thus they are limited in their support of diverse network protocols.

General purpose network processors (NPs) offer more flexibility to adjust a router's functionality after production [36]. These devices have become the computing device of choice in a large fraction of contemporary network routers. Network processors often feature multiple software-programmable processor cores which give router manufacturers and network managers the ability to dynamically configure and update router functionality. A detailed introduction to network processors can be found in Section 2.1.

A side-effect of shifting from ASIC-based routers to routers with network processors is that it creates a new class of vulnerabilities and corresponding security issues. Traditional ASIC-based routers are generally secure since their functionality cannot be changed after manufacture. In contrast, just as general-purpose workstations and server processors have software vulnerabilities that can be attacked remotely, network processors have software with potential security vulnerabilities. Such security vulnerabilities can be exploited to change the behavior of a router. In particular, prior work has shown that an experimental network processor with a security vulnerability in packet processing code can be attacked by sending a single User Datagram Protocol (UDP) packet [22] (Figure 1.2). The result of the attack was the indefinite retransmission of the attack packet on the outgoing link at full data rate. This type of attack is particularly concerning since it can be launched through the data plane of the network (i.e., no access to the control interface of the router is necessary). Its effect can be devastating since routers in the network inherently have access to multiple high-bandwidth links. Thus, this type of attacks can trigger Gigabits of attack traffic with a single transmission.

While similar vulnerabilities and attacks have not yet been disclosed for current commercial router systems, there are no fundamental reasons why they cannot be found. In particular, network processor software complexity continues to grow as more features are deployed and thus the attack surface continues to increase. It

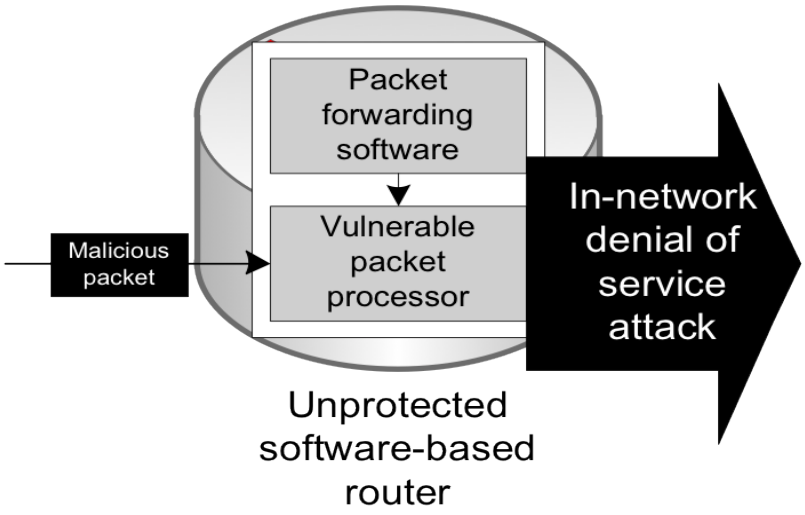

Figure 1.2: Attack on unprotected software-based router [21]

is important to note that even a single vulnerability can limit the operation of the Internet due to the homogeneity of the network equipment ecosystem. Currently, the network equipment market is dominated by a small number of vendors. If a vulnerability in deployed network processor code can be exploited, a large number of systems can be effected simultaneously. The ability to take down a significant fraction of all network devices in a short time would allow an attacker to drastically affect critical infrastructure. Such capabilities are particularly concerning in the context of cyber warfare (e.g., [61]).

Defenses against attacks on network processors need to match the system constraints of these devices. In particular, network processors use simple processor cores that typically do not run full operating systems. Thus, conventional software protection mechanisms (e.g., anti-malware software) are not suitable for this domain. In addition, network intrusion detection systems (e.g., Snort [77] or Bro [19]) are often only active on the ingress side of campus networks and thus do not protect the Internet core. Software based monitors (e.g., IRM [37]) are able to observe software execution and take remedial action on operations that violate a policy, but they are developed in high-level programming languages and require operating system sup-

port. Instead, hardware monitoring techniques have been proposed as an effective protection mechanism for network processors [16, 21, 65].

A *hardware security monitor* (Figure 3.5) is a module of specialized digital hardware that operates in parallel to the embedded network processor cores and keeps track of the processor behavior during runtime. It takes executed instructions from a processor core and compares them with a prestored "monitor graph", which is an binary that is generated from an offline analysis of the network application in the processor core to represent all possible valid program execution sequences. If any deviation from the expected instruction behavior is detected, the processor core can be reset by the monitor and continue operating without executing attack code. A comprehensive discussion of hardware monitors is presented in Chapter 3

1.2 Thesis Overview

Hardware monitors for *single core* network processor systems have been demonstrated in prior work [22, 57]. These solutions, however, do not address many critical problems that appear in practical network processor systems:

- Multiple cores: Practical network processors use multiple processor cores in parallel, and all of these cores need to be protected by hardware monitors.

- Multiple processing binaries: Network processors need to perform different packet processing functions on different types of network traffic. These operations are represented by different processing binaries on the network processing system. Thus, cores may need to execute different binaries and need to be monitored by hardware monitors that match these binaries.

- Dynamically changing workload: Due to changes in network traffic during runtime, the workload of processor cores may change dynamically [86]. Thus,

hardware monitors need to adapt to changing processing binaries during runtime.

- Homogeneity: Due to the large numbers of identical router devices in practical networks, a successful attack on one device can be applied to all the other devices, thus leading to a large-scale failure.

The goal of this dissertation work is to design, prototype and evaluate a multicore network processor security infrastructure that can accommodate all these requirements and protect routers based on network processors from data plane attacks.

Since one hardware monitor can only secure one processor core at a time, a multicore network processor needs a scalable processor-to-monitor interconnection architecture to balance the area cost and the performance overhead. A major contribution of our work is the design and prototype implementation of a clustered Scalable Hardware Monitoring Grid (SHMG) for multicore NP systems. In a multicore system that runs multiple processing binaries, dynamically changing network traffic requires the system to have the ability to reassign the processor cores to different binaries based on the network traffic at runtime. A resource reallocation algorithm is designed in SHMG for this purpose. An analytic analysis together with a simulation have validated the functionality and performance of this algorithm. A prototype SHMG on an Altera DE4 board [1] demonstrates low resource and performance overheads.

Although processors and monitors in SHMG design can dynamically switch from one network application to another, they can only run one application at a time. The secure installation of processor binaries and monitor graphs is indispensable during the application switch. A secure installation model based on cryptographic principles and suitable key management is presented to ensure invulnerability. When the control processor in the SHMG requests a new application from a remote server, the server first securely deploys the processor binaries and monitor graphs to a centralized mem-

ory in the network processor through an Ethernet port. Then the control processor installs the binaries and graphs to processor cores and monitors, if necessary.

Another problem in the network infrastructure is the *homogeneity* of routers, as we described in the early part of this chapter. To diversify the security of the routers, different hash functions are used with monitors in different NP-based routers. To support this security diversification, we present a Merkle tree based parameterizable high-performance hash function that can be configured to perform different functions in different devices by changing a 32-bit configuration parameter input. These hash values are then used by the monitor to evaluate network processor core performance on a cycle-by-cycle basis. A Secure Dynamic Multicore Hardware Monitoring System (SDMMon) that supports both secure installation and a parameterizable hash function has been prototyped and evaluated on an Altera DE4 board.

Eventually, a fully-functional, multicore network processor system which is protected by multiple hardware monitors is implemented on an Altera DE5 board. This system supports 10Gbps high speed Ethernet and has the capability to support all the security features in both SHMG and SDMMon. It has been evaluated at speed in a laboratory environment using a remote packet generator.

1.3 Thesis Outline

This thesis is organized into seven chapters:

Chapter 2 provides necessary background materials for this dissertation. We first describe network processors that are implemented with FPGAs in programmable network routers. Then, we provide an overview of network attacks and defenses in a computer network. Finally, FPGA technology is introduced.

Chapter 3 introduces data plane vulnerabilities in network processers, and presents and evaluates a memory based high performance hardware monitor to protect the NPs from this type of attack. We first start with an example of a data plane attack

in a network processor, then two previous hardware security monitoring techniques are introduced: address-based hardware security monitoring and hardware security monitoring based on instruction hashing. An enhanced hardware security monitor is designed and presented by constructing a deterministic finite automata (DFA) from a nondeterministic finite automata (NFA). The use of a DFA improves the efficiency of the monitor and reduces the monitor operation time to one clock cycle per instruction for all cases.

Chapter 4 includes the first major contribution of this thesis: a Scalable Hardware Monitoring Grid (SHMG) design for multicore network processors. By keeping a good balance between multicore interconnectivity and resource cost, the SHMG provides a scalable multicore hardware solution by clustering processor cores and monitors. Inside each cluster, a scalable processor-to-monitor interconnection is used. A runtime resource reallocation algorithm is presented to allocate the resources in SHMG, such as processors and monitors, to different network applications based on network traffic load. A Java-based simulator is designed to evaluate the performance and effectiveness of this algorithm. SHMG is prototyped and evaluated using four processor cores and six hardware monitors in an Altera Stratix IV FPGA on an Altera DE4 board.

Chapter 5 focuses on the system-level security issues in multicore hardware monitoring systems. We present a solution to the problem of secure, dynamic installation of hardware monitoring graphs on the network processors, and then address the problem of how to overcome the homogeneity of a network with many identical devices, where a successful attack, albeit possible only with small probability, may have devastating effects.

Chapter 6 introduces the final result of this research, a complete multicore network processor security platform on an Altera DE5 board. It consists of four 10Gbps SFP+ Ethernet ports, a four NP core six monitor SHMG cluster, a Nios II control processor running the μClinux operating system, and a hardware trusted platform module

(TPM). The platform also includes system software that supports SHMG runtime reconfiguration, security key management and secure installation of program binaries and monitoring graphs. Our system is evaluated experimentally by connecting to another DE5 board that operates as a packet generator and packet collector.

Chapter 7 summarizes this dissertation and provides directions for future work.

Results from the research outlined in this thesis have been published in the following conference proceedings and journals:

1. Tilman Wolf, Kekai Hu, Harikrishnan Chandrikakutty, and Russell Tessier, "Securing Network Processors with High-Performance Hardware Monitors", *IEEE Transactions on Dependable and Secure Computers*, vol. pp, issue 99, Nov 2015. [85]

2. Kekai Hu, Harikrishnan Chandrikakutty, Russell Tessier, and Tilman Wolf, "Scalable Hardware Monitors to Protect Network Processors from Data Plane Attacks", *Proc. of First IEEE Conference on Communications and Network Security (CNS)*, Oct 2013. **Best paper award**. [49]

3. Kekai Hu, Thiago Teixeira, Russell Tessier and Tilman Wolf, "System-Level Security for Network Processors with Hardware Monitors", *Proc. of 51th Design Automation Conference (DAC)*, San Francisco, CA, June 2014. [50]

4. Kekai Hu, Harikrishnan Chandrikakutty, Zachary Goodman, Russell Tessier, and Tilman Wolf, "Scalable Hardware Monitors to Protect Network Processors from Data Plane Attacks", *IEEE Transactions on Computers* , vol. pp, issue 99, May 2015. [48]

5. Russell Tessier, Tilman Wolf, Kekai Hu, and Harikrishnan Chandrikakutty, "Reconfigurable Network Router Security", in Reconfigurable Logic: Architecture, Tools and Applications, Pierre Gaillardon, ed., CRC Press, 2015 [82]

CHAPTER 2

BACKGROUND

2.1 Network Processor

Network processors (NPs) are programmable devices that can process network packets (up to hundreds of millions of them per second) at wire-speeds of multi-Gbps [42]. In contemporary routers, network processors (NPs) typically contain simple RISC-based processing cores which can efficiently manipulate data packets. The functionality of these programmable processors can easily be updated via software updates to provide a broad range of router functionality. Some of the typical applications implemented as software running on network processors include [26]:

- Packet or frame discrimination and forwarding: the basic function of a router or a switch.
- Quality of service (QoS) enforcement: identify and balance the network traffic of different types or classes of network packets, manage congestions, reduce error rate, etc.
- Access control functions: identify the authentication of packets and allow only the authenticated packets to traverse the router or switch.
- Encryption of data streams: Encrypt the data flow with hardware-based encryption engines in the network processor.

Network processors have been used increasingly widely in all kinds of contemporary network devices: workstation-based routers [54] [51], programmable routers [78],

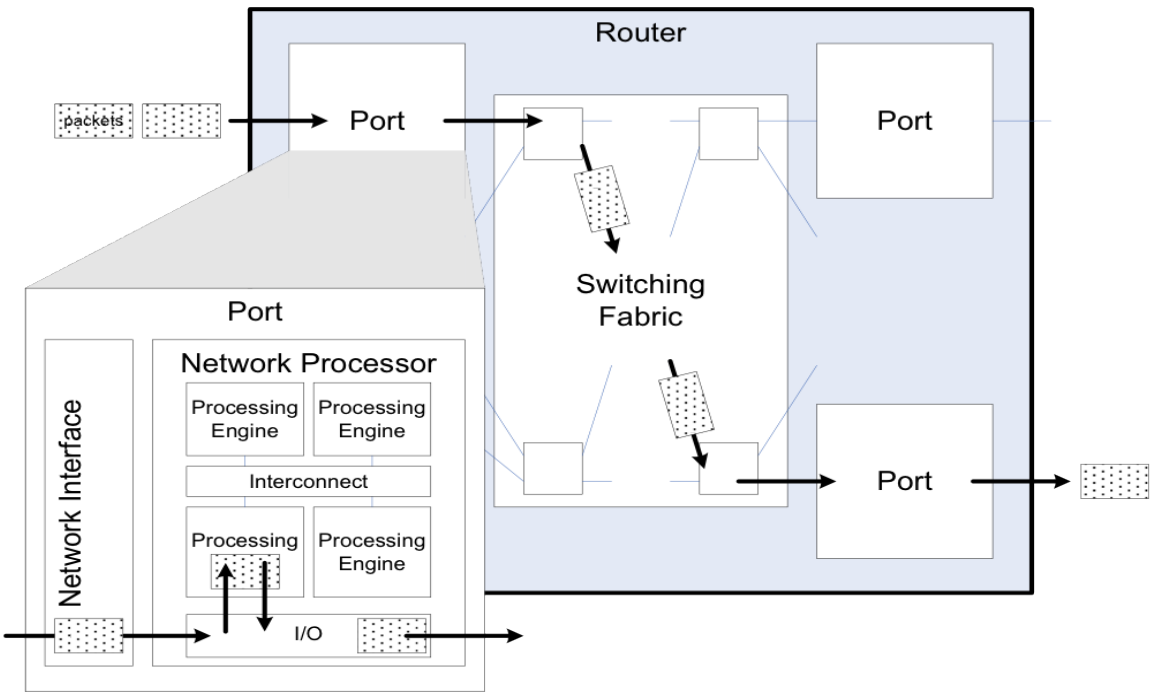

Figure 2.1: Network processor implemented in a programmable router

and virtualized router platforms [14] over the past decade. Commercial examples of network processors include Cisco QuantumFlow [25], Cavium Octeon [20], and EZchip NP-5 [38] with data rates in hundreds of Gigabits per second. Figure 2.1 shows a multicore network processor that is implemented in a programmable router.

2.2 Overview of Network Attacks and Defenses

In this age of universal electronic connectivity, the explosive growth in computer systems and their interconnections via the Internet has increased the dependence of both organizations and individuals on the information stored and communicated using these systems. This, in turn, has led to a heightened awareness of computer and network security [81] [15].

The NIST *Computer Security Handbook* [45] defines *computer security* as

> The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability,

and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications).

This definition introduces three key concepts that form the fundamental objectives of computer security:

- **Integrity:** This term includes two concepts:

  Data integrity: Make sure that no unauthorized modifications or destructions are made to the information and data.

  System integrity: Make sure that a system performs its intended function in an unimpaired manner [81].

- **Confidentiality:** This term includes two concepts:

  Data confidentiality: Make sure that no unauthorized individuals have access to the confidential information in the data.

  Privacy: Make sure that the information about the individuals who manage the data won't be disclosed to unauthorized individuals.

- **Availability:** Make sure that the systems work properly and the services are available to the authorized users.

2.2.1 Network Attacks

By the definition of the open systems interconnection (OSI) security architecture [31], any action that compromises the security of information or makes unauthorized use of an asset in computer networks can be defined as a network attack. Numerous network attacks [3, 41, 52, 68] can be classified as *passive attacks* or *active attacks*.

Passive attacks

A passive attack attempts to learn or make use of information from the system but does not affect system resources. Two types of passive attacks are the release

of message contents and traffic analysis. By definition, message content attacks are attacks that eavesdrop and monitor network transmissions to get confidential information in the messages. In traffic analysis, the attacker determines the location and identity of communicating hosts and observes the frequency and length of messages being exchanged. With this information, the attacker tries to predict the nature of the communication that is taking place. Passive attacks are very difficult to detect because they do not involve any data modification. Neither the message sender nor the receiver is aware that a third party has read the messages because the data is unaffected. As a result, a typical security solution for passive attacks is not detection, but prevention, usually by means of encryption [81].

Active attacks

An active attack attempts to alter system resources or affect their operation. Active attacks involve some manipulation of the data stream or the creation of a malicious stream and can be subdivided into four categories: modification of messages, masquerade, replay, and denial of service. Modification of messages includes hackers manipulating some part of a legitimate message to produce unauthorized effects. Masquerade attacks usually happen together with other active attacks, for example, modification of message attacks. After capturing a message from the sender, a hacker modifies the message and masquerades as the sender or a different entity, such as authorized entities, to transmit the malicious message to the receiver. Replay attacks happen when an attacker passively captures a message, and retransmits this message to produce an unauthorized effect. Denial of service (DoS) attacks prevent the authorized use or management of network facilities such as websites, servers, or routers. An attacker takes control of a large number of devices in the network and bombards a target facility with a large number of messages to overload it, thus disrupting the normal operation of the target.

In contrast to passive attacks that are difficult to detect but mostly reasonable to prevent, active attacks are difficult to prevent because of the variety of potential network vulnerabilities. Instead, the defense objective of active attacks is to detect the attacks and recover from any negative effects. These types of attacks are the focus of this research.

2.2.2 Security Defense Mechanisms

Firewalls and virus scanners are the most common security solutions on network end-systems [64]. Firewalls [70] [89] are software or hardware-based network security systems that examine incoming and outgoing packets and filter out malformed packets. Based on a variety of applied rule sets, firewalls can be implemented at different network layers including the application, transport and network layers. They are useful against masquerade attacks and denial of service attacks that originate from outside the firewalls. Virus scanners (i.e., antivirus) are examples of intrusion detection systems [40]. These systems detect and remove malicious viruses and all kinds of malware, such as browser hijackers, backdoors, rootkits, Trojan horses, worms, fraudtools, adware and spyware to protect host computers from hacking, phishing, espionage, etc.

Many attackers target the network control plane in applying attacks such as malicious route announcement and DNS cache poisoning. Defenses to these attacks rely on secure routing (with cryptographic authentication) [53,80] or secure DNS protocols (DNSSEC) [55] in performing network communication.

In the network data plane, where traditional passive attacks such as sniffing and snooping take place [84], data encryption is used to protect data confidentiality and integrity. Encipher algorithms (e.g., Data Encryption Standard (DES) [27] and Advanced Encryption Standard (AES) [30]) transform data into a scrambled form that is not readily intelligible. Encryption keys are used during the transformation and sub-

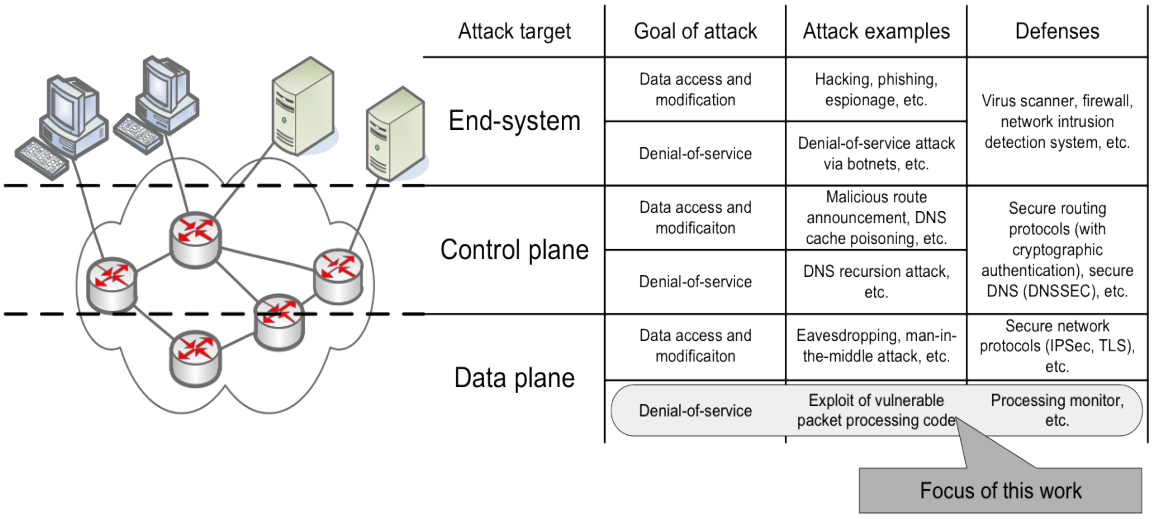

Figure 2.2: Network attacks and defenses classification

sequent recovery of the data. The encryption algorithms are designed in a way that attackers cannot recover the data without the keys. Secure network protocols, such as Internet protocol security (IPSec) [34] and transport layer security (TLS) [32], employ data encryption as well as well-established handshake processes to prevent data plane attacks like eavesdropping and man-in-the-middle attacks.

Recently, a new class of attacks has emerged targeting the network data plane. This type of attack aims to disrupt or modify the operation of routers to achieve denial of service attacks or to use routers to actively launch denial of service attacks. Unlike the attacks that target the control interface of routers [29] and can be prevented using standard security mechanisms as for end-systems, this new attack on general-purpose network processor cores can be launched through the data plane by simply sending malformed data packets [22] and cannot be easily prevented with conventional mechanisms. This attack is particularly problematic since it targets the high-performance forwarding component of critical infrastructure. Our work in this thesis focuses on preventing this type of attack.

2.3 Overview of FPGA Technology

Although the multicore network processors we protect with monitoring in this work are initially targeted to field-programmable gate arrays (FPGAs) for prototyping, the entire system could eventually be implemented in an ASIC. To assist the reader in understanding our prototyping technology, we provide a brief discussion of FPGAs. An FPGA is an integrated circuit that can be reprogrammed by a user to perform any digital logical function. In contrast to an ASIC that is customized for a particular use, an FPGA can be used to implement virtually any digital logic function. The FPGA implementation sacrifices area, delay and power versus an ASIC implementation to achieve the flexibility [58].

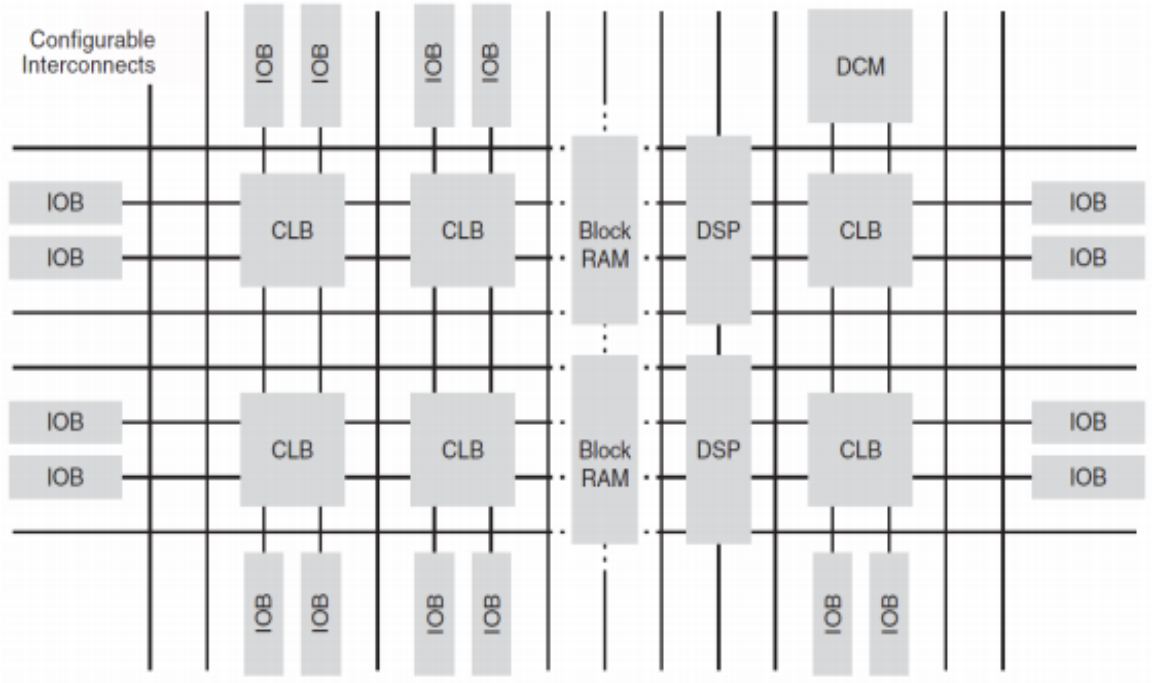

Figure 2.3: FPGA architecture [71]

The most common FPGA architecture consists of an array of configurable logic blocks (CLBs), configurable interconnections (wires) and input/output banks (IOB). (Figure 2.3). During FPGA configuration, digital circuits are implemented by customizing the CLBs and the routing circuitry using computer-aided design (CAD)

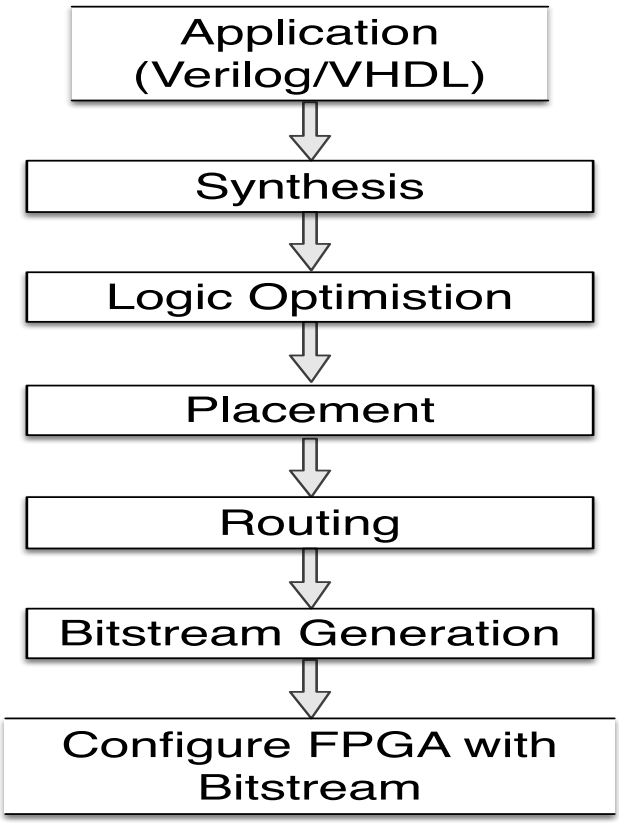

Figure 2.4: FPGA development flow

software (e.g., Quartus [11] for Altera FPGAs and ISE [87] for Xilinx FPGAs). In addition to millions of CLBs and wires, state-of-art FPGAs also integrate Block RAMs, digital signal processing blocks (DSPs) and digital clock managers (DCMs).

Hardware description languages (HDL), such as Verilog and VHDL, are used to specify programs for FPGAs. After a user provides an HDL design to the CAD tools, the tools translate the hardware description to an optimized technology-mapped netlist. Mapping to the target FPGA takes place under constraints of resources, area, clock speed, and power. A typical FPGA development flow is illustrated in Figure 2.4. This physical design process includes placement and routing steps which assign design CLBs and other resources to physical resources on the chip and interconnect them using wires with programmable connections. After assignment, a bitstream is

generated for a target FPGA. Downloading this bitstream to the FPGA configures the device to its appropriate functionality.

In recent years, CAD tools have greatly simplified FPGA application development. Compared with the long time period required to design, tape out and verify an ASIC design, the compilation and verification process of an FPGA design is simple and fast. The typical compilation time of an FPGA design ranges from a few minutes to a couple of hours. This feature makes FPGAs attractive for rapid prototyping applications. Since FPGAs typically lead ASICs in process technology, they are used extensively to prototype new technologies before fabricating custom ASICs [35, 59].

CHAPTER 3

DATA PLANE ATTACK AND HIGH-PERFORMANCE HARDWARE MONITOR

This chapter explores the vulnerabilities in network processors and gives an example of a potential attack that can lead to network denial of service. Hardware security monitoring techniques are applied to prevent this type of attack.

Sections 3.1 and 3.2 introduce data plane vulnerabilities identified in previous research [22] and review two types of existing hardware monitor designs [16, 22]. Section 3.3 introduces a high-performance hardware monitor design from [57]. In Sections 3.4 and 3.5, we provide a comprehensive evaluation of a high-performance hardware monitor design using nine NpBench [60] network benchmarks.

3.1 Vulnerabilities in Network Processors

As discussed in Section 2.1, network processors are located at router ports, where they process traffic that is traversing the router. The typical system architecture and operation of a network processor is illustrated in Figure 2.1.

Due to the very high data rates at the edge and the core of the network, network processors typically need to achieve throughput rates on the order of tens to hundreds of Gigabits per second. To provide the necessary processing power, network processors are implemented as multi-processor systems-on-chip (MPSoC) with tens to hundreds of parallel processor cores [25] [20] [38]. Each processor has access to local and shared memory and is connected through a global interconnect. Depending on the software configuration of the system, packets are dispatched to a single processor

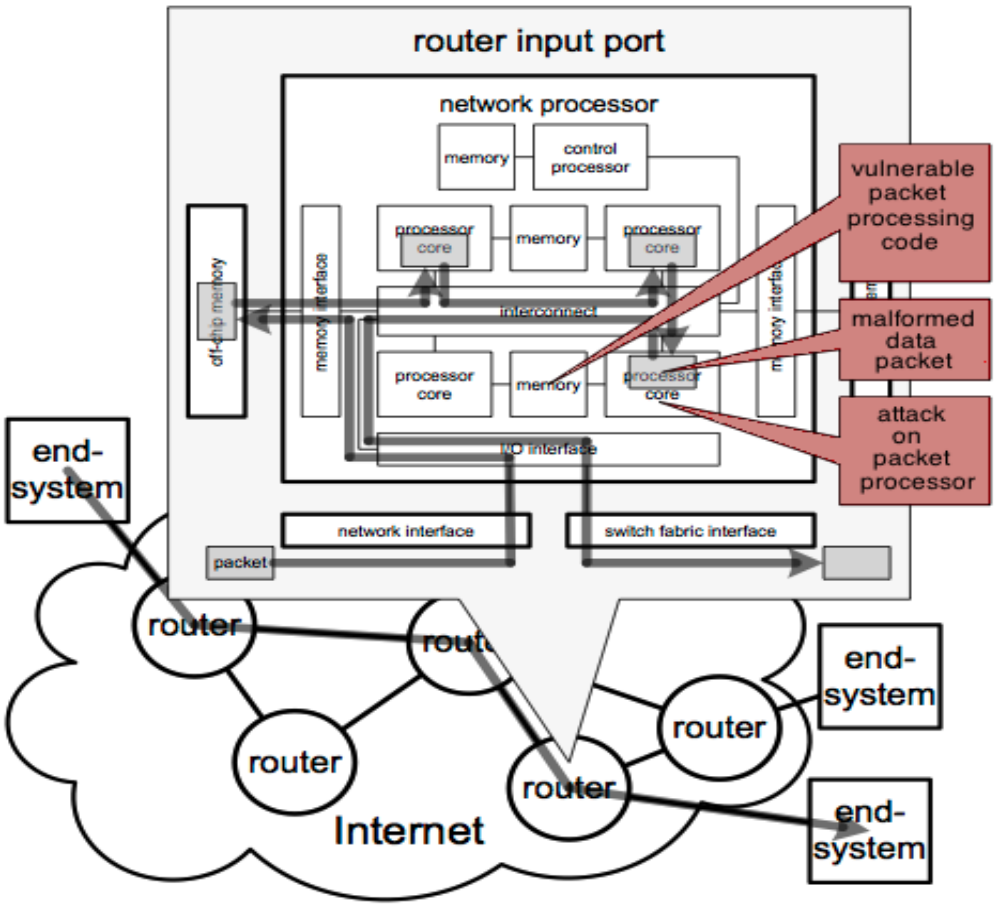

Figure 3.1: Attack on network processor

core for processing (run-to-completion processing) or passed between processor cores for different processing steps (pipelined processing). An on-chip control processor performs runtime management of processor core operation.

In order to fit such a large number of processor cores onto a single chip, each processor core can only use a small amount of chip real-estate. Therefore, network processor cores are typically implemented as very simple reduced instruction set computer (RISC) cores with only a few kilobytes of instruction and data memory. These cores support a small number of hardware threads, but are not capable of running an

operating system. Therefore, conventional software defenses used for workstation and server processors cannot be employed. Nevertheless, these cores are general-purpose processors and can be attacked just like more advanced processors on end-systems.

An attack scenario for network processors is illustrated in Figure 3.1. The premise for this attack is that the processing code on the network processor exhibits a vulnerability. It was shown in prior work that such a vulnerability can be introduced due to an uncaught integer overflow in an otherwise benign and fully functional packet processing function [22].

Here we give an example network application that has a vulnerability in packet processing code and show that if this vulnerability is matched with a suitable attack packet (e.g., a malformed UDP packet), an attack on a processor core can be launched. The attack packet smashes the processor stack and leads to the execution of code that is carried in the packet payload. The processor ends up re-transmitting the attack packet at full data rate on all its outgoing ports without recovering until the network processor is reset.

The vulnerable application (Figure 3.2) is a congestion management (CM) protocol application [18] that inserts a custom protocol header in the packet header space between the IP header and the UDP header.

After inserting the CM header, the application checks the new packet size ($len1 + len2$) to make sure it does not exceed the maximum datagram length. Exploiting an integer overflow vulnerability, the boundary check in the CM code can be circumvented and the stack can be smashed. The vulnerable code is shown in Figure 3.3.

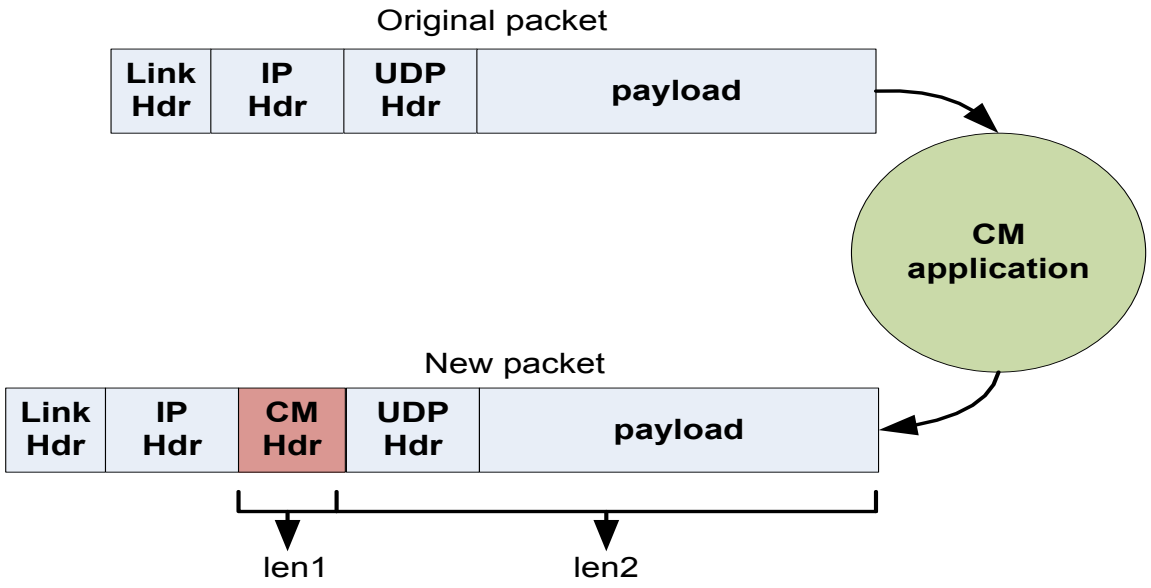

Figure 3.2: IPv4 packet with congestion management header

```
int mybuf[60];
unsigned short sum;
Pack (in,out);
sum= len1 + len2;
if(sum > MAX_PKT_SIZE) {
return -1;}
else {
memset(mybuf, buf1, len1);
memset((mybuf+len1), buf1, len2);
return 0;
}
```

CM protocol

```
{
unsigned int d;
unsigned int port;
_u32 ip_dst;
ip_dst= ip_dst_hi + ip_dst_low;
port = (ip_dst & 0x000000ff);
port = (port << 16);
d = (d1 | port);
pkt_dbg(0x137,  d);
pkt_dbg(0x157,  port);
put_pkt(0,d);
```

IPV4 application

Figure 3.3: Integer overflow vulnerable code

The variable *sum* is defined as *unsigned short* type. In normal cases, this code works fine. However, if an attacker sends a carefully malformed UDP packet, an integer overflow would be triggered. For example, if the *len*1 is set to a 16 bit value 0xfffe (decimal value 65534) and *len*2 is 10, the packet size value *sum* will be 8 instead of 65544 due to the integer overflow. Thus, the malformed packet can pass the length

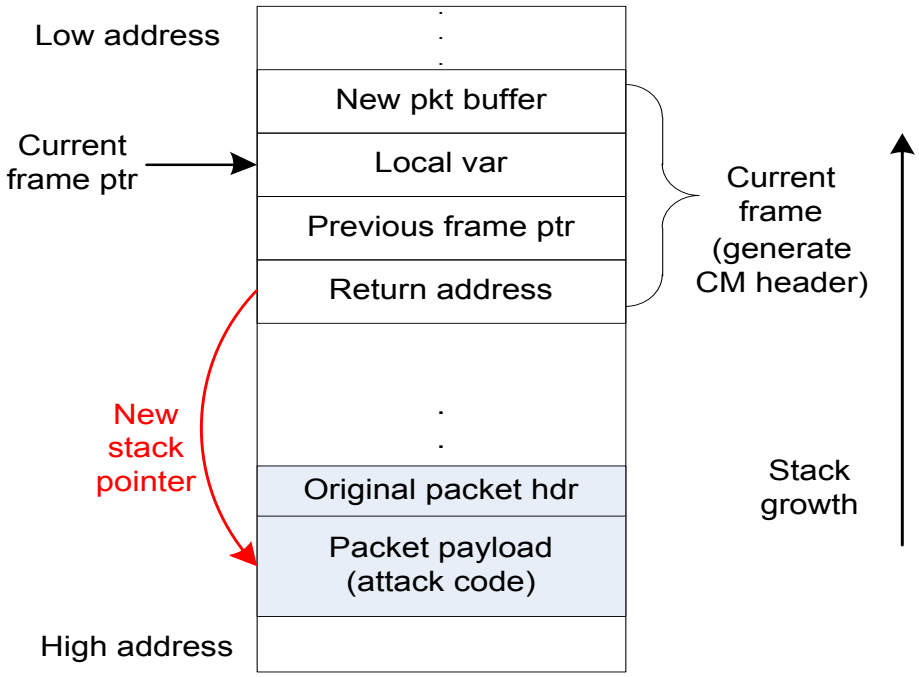

Figure 3.4: Stack overflow [22]

check. However, in the actual memory copy process, the application will copy 65544 bytes of packet data, rather than 8 bytes, to the processor memory.

As a result, the packet payload is copied over the stack. The packet payload of the attack packet is crafted in such a way that the return address is overwritten (Figure 3.4) to direct the control flow to the IPv4 packet forwarding application (which is library code on the processor core) and the value of the *ip_dst_low* field is 0xff. The port information gets updated with this value (the boxed instruction in the IPv4 code), forwarding the attack packet to *all* the outgoing ports at full data rate. Due to the very high data rates of modern routers, this type of attacks can lead to a DoS in the network in a very short time.

3.2 Defense Mechanisms Using Hardware Monitoring

Solutions to protect network processors from attacks on vulnerable processing code are constrained by the limited resources available on these systems. Network

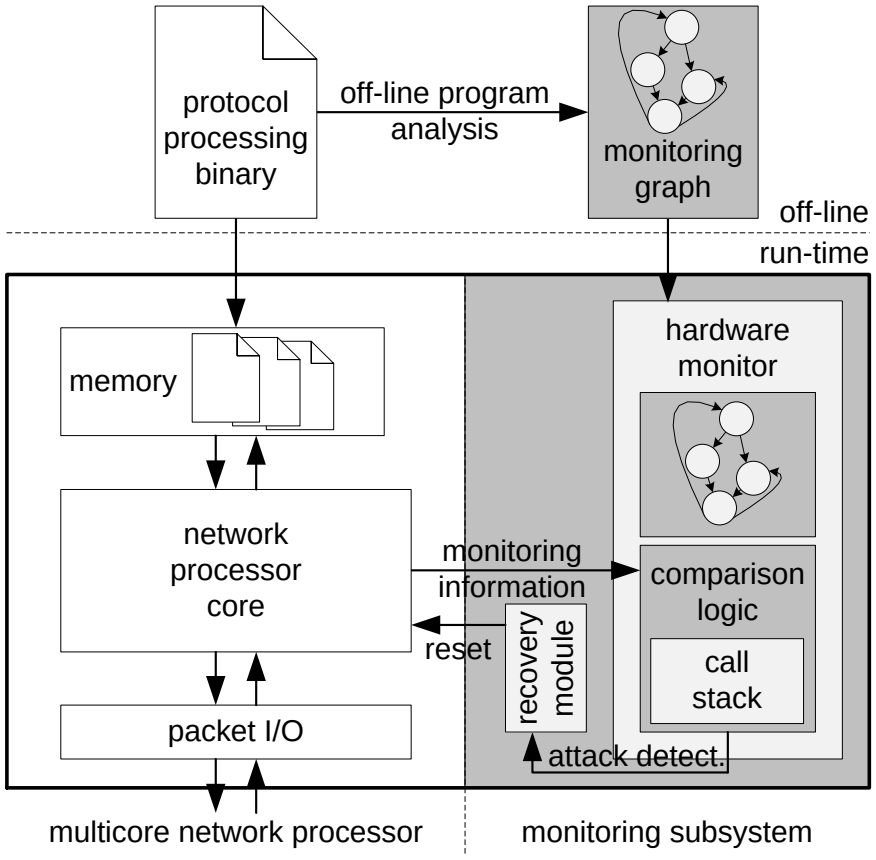

Figure 3.5: Hardware monitor for single processor core.

processors cannot run complex protection software and cannot dedicate lots of chip real estate to protection mechanisms. One promising approach is to use *hardware monitors*, which have been successfully used in resource-constrained embedded systems [9, 16, 43, 65, 74, 75, 90].

The operation of a hardware monitor is illustrated in Figure 3.5. The key idea is that the processing core reports what it is doing as a monitoring stream to the monitor. The monitor compares the operations of the processor core with what it thinks the core should be doing. If a discrepancy is detected, the recovery system is activated to reset the processor core. In order to inform the monitor of what

processing steps are valid, the processing binary is analyzed offline to extract the "monitoring graph" that contains all possible valid program execution sequences.

Based on the information which is used to generate the monitoring graph, hardware security monitors can be divided to two types: address-based monitors and monitors using instruction hashing. Both of them need to meet the following criteria:

1. Correct detection: Correctly identify malicious attacks.

2. Low resource overhead: Due to the limited resource availability in the network processors, the hardware monitors need to be designed with low resource overhead.

3. Fast detection: Since the network processor cores work in high speed, hardware monitors should be able to detect malicious behavior within one or a small number of clock cycles to avoid large performance overhead. The less performance overhead the better.

3.2.1 Address-based Hardware Security Monitor

An address-based hardware security monitor [21] uses instruction address information in the processor application binary to monitor processor behavior. Instructions are grouped into different basic blocks based on their positions to the branch instructions. All the instructions before the next branch instruction are classified to the same basic block.

As we can see in Figure 3.6, since the fifth instruction is a conditional jump instruction, all instructions from memory locations 1 to 5 are grouped as basic block zero. Similarly, instructions from memory locations 6 to 8 are grouped as basic block one because instruction eight is an unconditional jump instruction.

The high level architecture of the address-based hardware security monitor system is illustrated in Figure 3.7 [21]. It is a four stage pipeline architecture where each pipeline stage takes one clock cycle to complete.

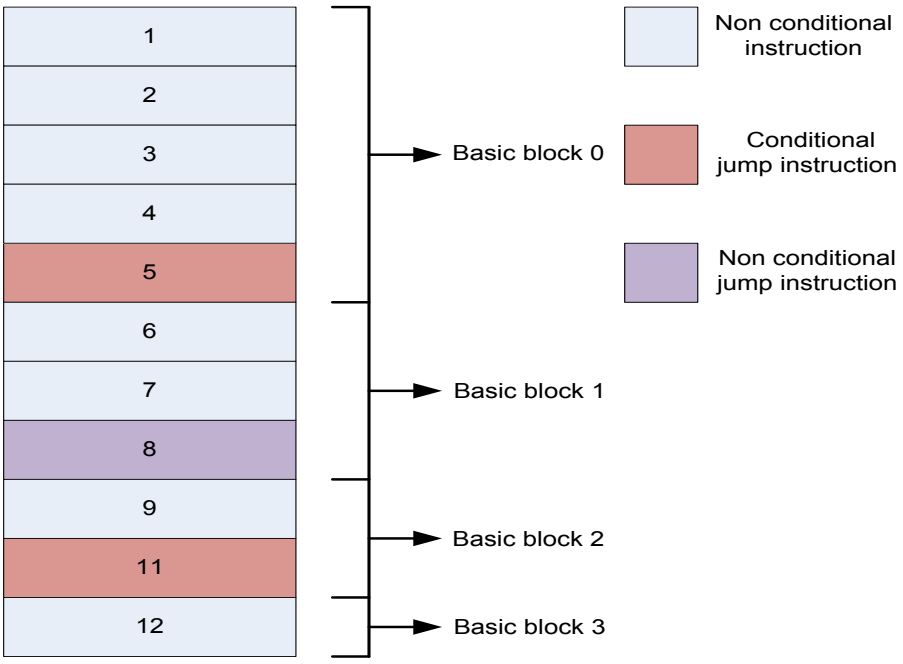

Figure 3.6: Address-based basic block representation [56]

- Stage 1: Index a block RAM (BRAM) with the instruction address of the currently executed instruction. The BRAM contains the basic block number of the instruction and the next-hop address.

- Stage 2: Store the basic block number in a FIFO block and forward it to stage 3.

- Stage 3: Compare the current basic block number input from the second stage with the block information of the just completed instruction from the FIFO block. If they are the same, this instruction belongs to the same basic block and the current instruction is valid. If not, check if the instruction belongs to the next basic block or the jump target basic block. If one of them matches with current instruction, it is a valid jump.

- Stage 4: The next-hop address for the just completed instruction is used to once again index the basic block memory. If the basic block for this target

is the same as the basic block of the currently-executed instruction, a valid instruction sequence is determined. Otherwise, an error signal is generated to stop processor operation.

A problem of the basic block monitoring strategy is that if the attack instruction belongs to the same basic block as the expected instruction, it can be undetectable. As we can observe from Figure 3.8, if the attacker injects malicious instructions (2 and 3) in the middle of a basic block (block 0), the monitor cannot detect these attack instructions. Moreover, keeping track of the basic block information and next-hop address information for all instructions in the basic block memory increases the on-chip memory overhead.

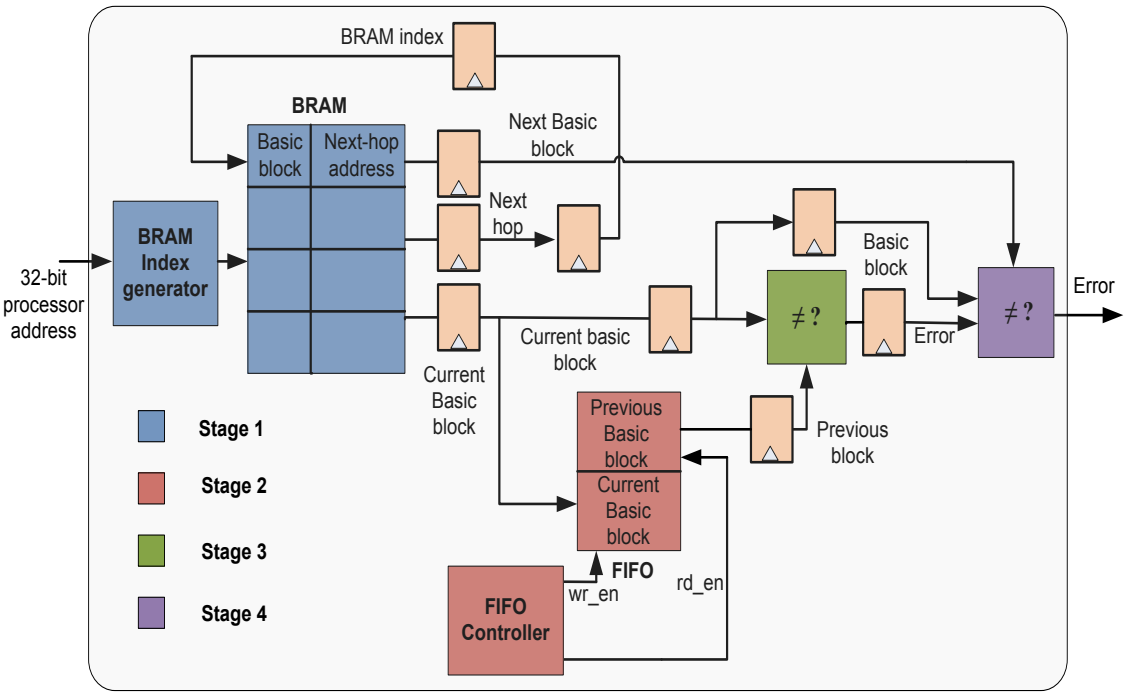

Figure 3.7: Address-based hardware security monitor architecture [21]

3.2.2 Hardware Security Monitor Using Instruction Hashing

To improve the limitations of address-based hardware monitors, instruction hashes are used in place of basic blocks to validate processor operation at runtime. A new

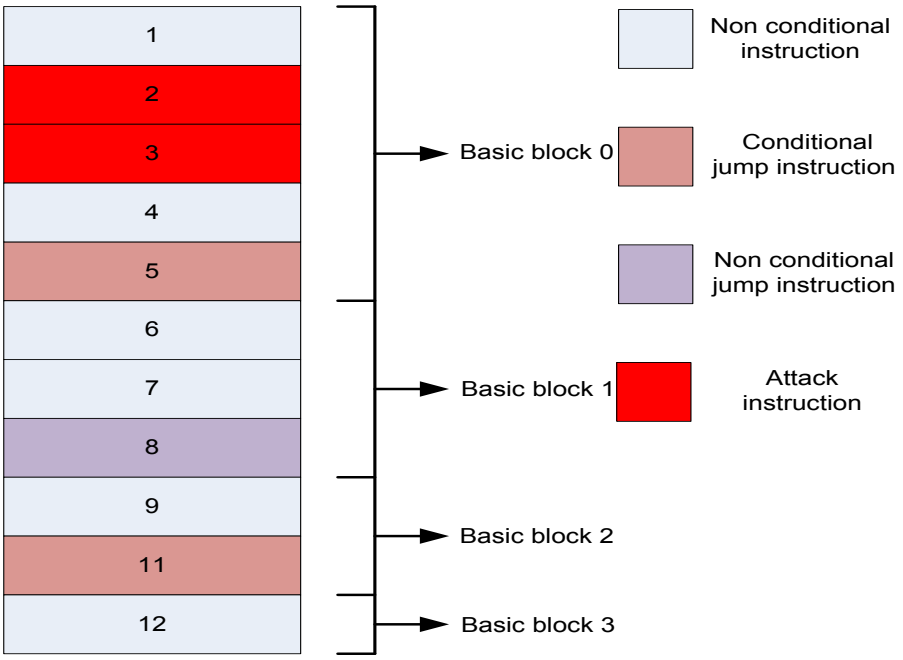

Figure 3.8: Undetectable attack instructions [56]

hardware monitoring strategy that can verify individual instructions is designed in [57].

The monitoring graph used by the hardware monitor is a state machine, where each state represents a specific processor instruction. The state machine is derived from the packet processing code as illustrated in Figure 3.9. Each processor instruction corresponds to a state. The edges between states are labeled with information relating to the next valid instruction that can be executed after the current instruction. In case of control flow operations, there may be multiple outgoing edges from each state (each being a valid transition). In this system, a 32-bit processor (i.e., open source embedded Plasma processor based on the MIPS instruction set) is used. The monitoring system uses a 4-bit hash of the next instruction to label edges in the monitoring graph (as has been recommended in [65]). A hash (instead of the full 32-bit instruction) is used to reduce the size of the monitoring graph and thus to

reduce the implementation overhead of the hardware monitor while still allowing instruction-by-instruction monitoring.

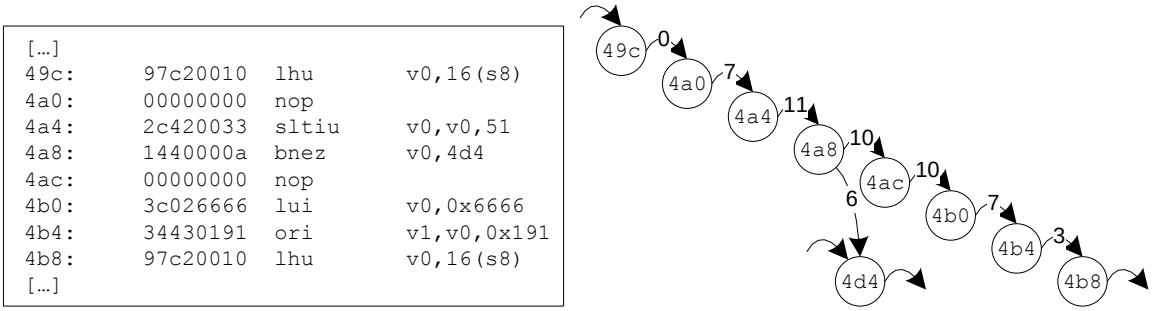

Figure 3.9: State machine generation from processing binary

Whenever the processor runs an instruction, it inputs this 32 bit instruction to a hash function that generates a hash value (can be any length, 3, 4, and 5 bits are evaluated in our work) for this instruction. Then, the possible hash values for this instruction are fetched from the monitoring graph and compared with the input hash value. It is possible that there are more than one hash value for this instruction due to the jump or branch instructions that have more than one target instructions. If the input hash value does not match any of the possible hash values, an error signal is generated indicating that an incorrect instruction has been executed.

The key improvements of the instruction hashing monitor over the address-based monitor are:

- An instruction hashing monitor checks processor behavior using instruction information, rather than address information, thus eliminating the undetectable attacks in address-based monitors.

- Next instruction information for conditional instructions are determined in the jump logic, reducing memory resource utilization.

The use of a hash (or any other method that uses a many-to-one mapping), however, leads to two fundamental problems:

- Attack detection ambiguity: The many-to-one mapping that occurs in a hash function of the monitor may make it possible for an attacker to remain undetected. This would require that the attack performs operations that lead to a sequence of hash values that matches the monitoring information of valid code. Mao et al. have shown that this probability decreases geometrically with the length of the attack code and thus is unlikely to lead to practical attacks [65] (in particular when the hash function is not known to the attacker). We do not consider this issue further in this work.

- Nondeterminism during monitoring: The many-to-one mapping also leads to nondeterminism in the monitoring graph. There may be a control flow instruction where each of the next instructions has the same hash value. As a result, the corresponding node in the monitoring graph has two outgoing edges with the same hash value (as illustrated in Figure 3.10). Since this nondeterminism can continue for multiple such control flow operations, it can lead to complex implementations [22], potentially slowing monitor performance.

In the following section, we show how we can address the latter problem by converting the nondeterministic monitoring graph into a deterministic monitoring graph, which is easier to use in high-performance implementations.

3.3 High Performance Hardware Security Monitor

To realize a deterministic instruction-level monitor, the NFA monitoring graph described in the previous section is converted to a DFA monitoring graph. This section describes how to implement a monitoring system that uses this DFA graph [57]. Note that significant portions of the work in this section were developed by Harikrishnan Chandrikakutty as part of [57]. This work was extended through additional

experimental results, as described starting in Section 3.4, for this dissertation. This section is provided for completeness.

3.3.1 Construction of Deterministic Monitoring Graph

Tracking nondeterministic finite automate is difficult to implement in practice since the automaton can have multiple active states. This leads to high bandwidth requirements between the monitoring logic and the memory that maintains the NFA since next-state information for all active states has to be fetched in each iteration. When using a DFA, in contrast, only one state is active and implementation becomes much easier.

To convert an NFA to a DFA, a standard powerset construction algorithm can be used [47]. This algorithm computes all possible state sets in which the automaton can be situated (i.e., the powerset). Based on the powerset, a DFA is then constructed. Figure 3.11 shows the DFA that corresponds to the NFA shown in Figure 3.10. Note that state `{3,5}` represents the sets of states to where state 2 can branch when hash value `c` is observed.

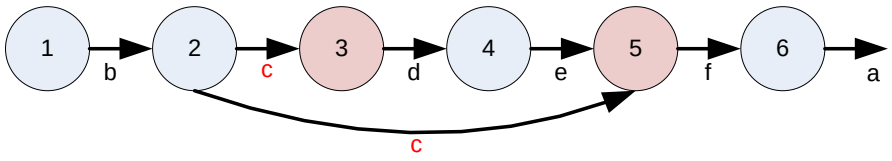

Figure 3.10: Nondeterministic monitoring graph.

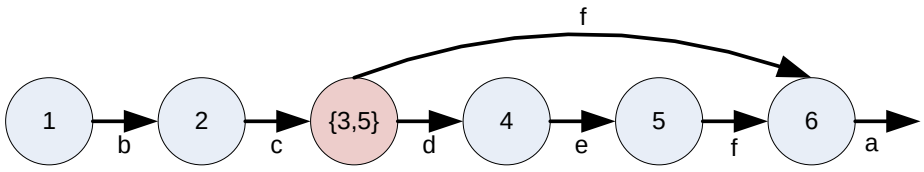

Figure 3.11: Deterministic monitoring graph after NFA-to-DFA conversion.

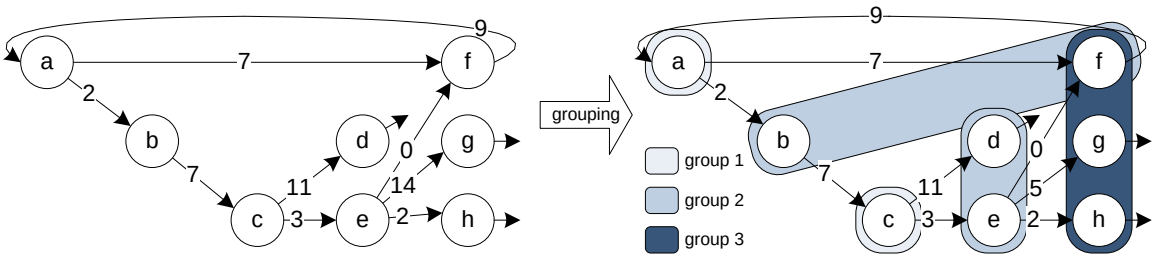

Figure 3.12: Grouping of DFA states.

One potential problem with NFA-to-DFA conversions is that the number of states in the DFA can grow exponentially over the number of states in the NFA. However, the monitoring NFAs constructed from binary code do not exhibit this pathological behavior. Experiments indicate that this increase is small and does not lead to drastically larger state machines (see Section 3.5). Thus, this approach is effective for creating deterministic hardware monitors.

3.3.2 Implementation of Monitoring System

A key challenge in the implementation of this hardware monitoring system is how to represent the monitoring DFA in memory. The comparison logic needs to be able to retrieve the information about next state transitions for every instruction that it tracks. Thus, state transitions need to be implemented with no more than one memory access per instruction (to keep up with the network processor core) and be as compact as possible (to minimize the implementation overhead of the monitor).

The information that needs to be stored in the monitoring memory is illustrated on the left side of Figure 3.12. Each state represents an instruction and an outgoing transition edge from this state represents the hash value of the next expected instruction in the execution sequence. For example, state `c` has two next states, `d` and `e`, with hash values 11 and 3, respectively.

A naïve way to store the state machine in RAM would be to store each state and all its possible edge transitions. This would require 2^h entries per state for an h-bit

hash. Since most states have only one or two outgoing edges, a large number of edge transitions would never be used, leading to inefficient memory use. Assuming that only two outgoing transitions exist for each state is also not feasible due to the cases where powerset construction creates states with up to 2^h outgoing edges. Finally, for performance reasons there should only be one memory access per state transition, which precludes a design where states with more than two outgoing edges are handled as special cases.

The main idea to compactly represent DFA states with varying numbers of outgoing edges is to encode all the necessary information in a single table entry and to group states by the number of outgoing edges. The main challenge in achieving compactness is to allocate exactly the amount of memory that is needed for each state to store next state information while still being able to index this memory without degrading to linear search. In the representation, states are grouped together if they have the same previous state. A state belongs to group g if the previous state has g outgoing edges. For a monitor with a 4-bit hash value, there are 16 possible groups. For example, in Figure 3.12 on the right side, groups are shown with different colors. Note that a state can belong to multiple groups (e.g., state `f` belongs to group 2 (because `a` has two outgoing edges, one to `b` and one to `f`) and to group 3 (because `e` has three outgoing edges)).

The memory layout and basic operation of the DFA monitor system is shown in Figure 3.13. The memory contains tuples of {number of next states, offset in state group, valid hash values on outgoing edges} and is logically divided into groups. The base addresses for each group are stored in a register file with 16 entries. Within a group, the sets of states that share the previous state are grouped together (e.g., `b` and `f` are together and `d` and `e` are together). Within a set, states are ordered by the hash value on their incoming edge (e.g., `e` before d because hash value 3 is smaller than hash value 11). The hash comparison block performs two functions: it

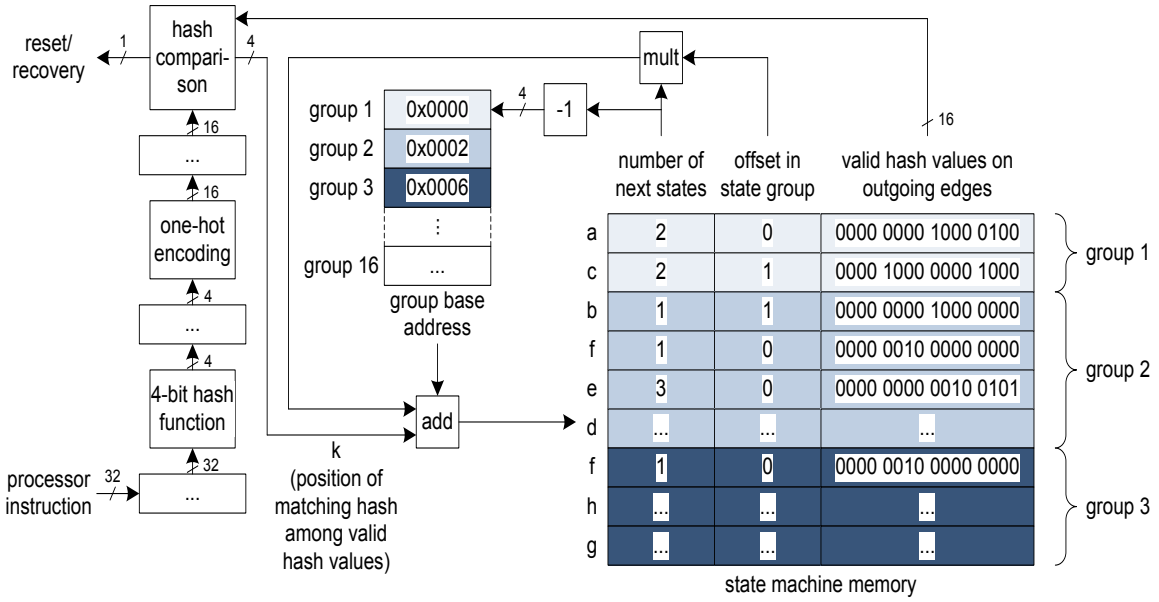

Figure 3.13: Memory based high performance security monitor architecture

determines if the one-hot coded hash bit is set in the 16-bit value read from memory and it determines k, which is the position of the matching hash value among the valid hash values read from memory.

To illustrate the operation of the monitor, an example transition is shown. Assume the monitor is in state `a` and the processor reports an instruction that leads to a hash value of 7. To perform the transition, the memory row labeled `a` is read. The tuple in this row indicates that there are two outgoing edges. The valid hash values of these two edges are stored in the 16-bit vector. To verify that the transition is valid, the hash comparison unit checks if bit 7 is set in the bit vector (which it is). If this bit is not set, then an invalid transition takes place, indicating an attack, and the processor is reset. After the check, the next state (i.e., state `f`) in the DFA needs to be found in memory. To determine the address of that state, the base address of the group of the next state is looked up in the register file (i.e., 0x0002 since the next state belongs to group 2). To this base address, the product of the set size (i.e., group number) and the offset in the state group is added (to index the correct set within the group).

Finally, k is added, which is the position of the matching hash in the bit vector (in this case 1 since 2 is the first matching hash (i.e., k=0) and 7 is the second matching hash (i.e., k=1)). Thus the memory location of state `f` is 0x0002 + 2×0 + 1 = 0x003.

Note that any state transition takes only one memory read from state machine memory and a lookup into a fixed-size register file. The DFA is represented compactly without wasting any memory slots (states shown with dots in Figure 3.13 point to other states not shown in this example). Thus, this representation lends itself to a high-performance implementation.

3.4 Prototype System Implementation

Although an end-system would likely be implemented in fixed logic, we have prototyped the described network processor and hardware monitoring system on a Stratix IV GX230 FPGA located on an Altera DE4 board. The router infrastructure surrounding the NP core is taken from the NetFPGA reference router [63], which has been migrated to the Stratix IV family. The DE4 board has four 1 Gbps Ethernet interfaces for packet input/output. In our prototype implementation, the single-core network processor is implemented as a soft core and the monitor is implemented in FPGA logic (using Quartus for synthesis, place and route). Only the memory initialization files need to be reconfigured on a per-application basis.

The automated offline analysis tool for security monitor generation is illustrated in Figure 3.14. To run networking code on the processor plus monitor system, the code is first passed through a standard MIPS-GCC compiler flow to generate assembly-level instructions. The output of the compiler allows for the identification of branch instructions and their target addresses. In our current implementation, all possible branch targets and return instructions are analyzed at compile time. The monitor can handle an arbitrary number of indirect branches to statically known targets (e.g., return addresses) since the NFA representation allows any number of outgoing

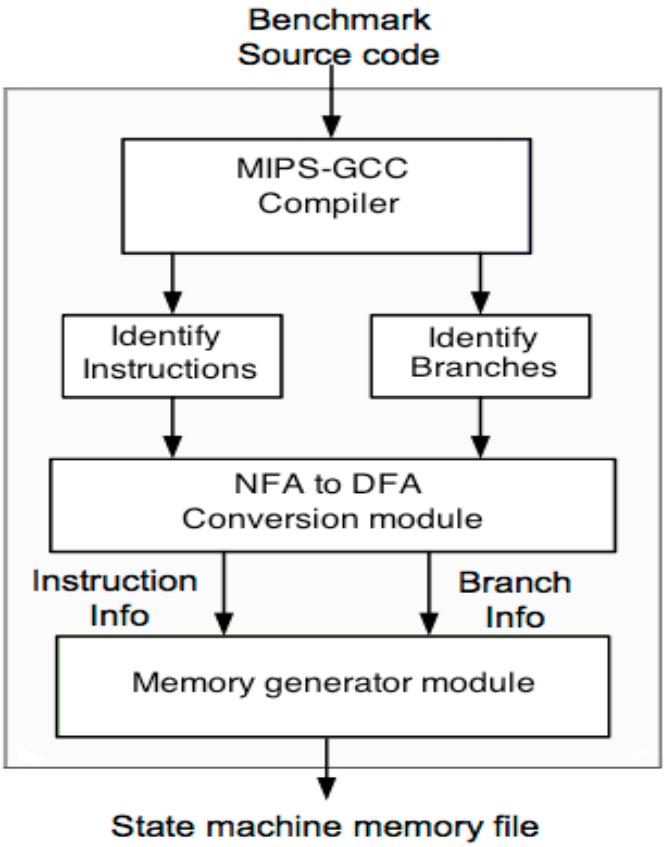

Figure 3.14: Offline analysis to create state machine memory file

branches. (Our monitor cannot handle indirect branches to statically unknown targets that are resolved at run time, but such programming constructs did not appear in any of 11 benchmark applications that we looked at). The DFA-to-NFA conversion starts with a non-deterministic NFA representation obtained from the compiler information. Through powerset construction, a DFA is constructed. This DFA is then converted into a monitoring state machine memory file using the process described in Section 3.3.1 and is loaded into the monitor when the processing binary is installed in the processor.

To evaluate our system, nine benchmarks from the NpBench suite [60] were processed with this flow. NpBench is a benchmark suite targeting modern network processor applications. The benchmark applications are categorized into three specific

Table 3.1: Statistics for NpBench benchmark applications

Netw. application	Category	No. of instr.	No. branch instr.	No. branches >2 targ.	Max branch targ.
crc	PPG	276	17	0	2
frag	PPG	573	70	3	3
red	TQG	802	88	1	2
md5	SMG	3147	211	24	8
ssld	TQG	828	91	1	5
wfq	TQG	905	112	2	2
mtc	SMG	2427	252	2	3
mpls-upstream	TQG	1603	322	9	10
mpls-downstream	TQG	1574	276	5	12

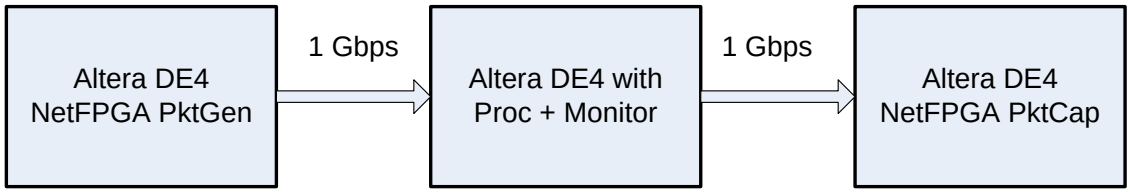

Figure 3.15: Network topology used for experimentation

functional groups - traffic management and quality of service group (TQG), security and media processing group (SMG) and packet processing group (PPG). A listing of the benchmarks and their application categories appears in Table 3.1. Since the presence of instruction branches has a direct impact on NFA-to-DFA conversion and monitoring state machine memory size, the number of branch instructions for each benchmark is included in the table. Return instructions at the end of subroutines often contain numerous targets since a subroutine can be called from numerous other functions. The number of these *jump register* instructions with more than two possible return addresses are listed in the table. Additionally, the maximum number of target addresses for any branch in each application is also included.

Table 3.2: Evaluation of monitoring approaches for the DFA approach and a previous NFA-only approach. The maximum number of memory accesses for the DFA approach is 1 for all benchmarks.

		Chasaki [22]		DFA approach		
Netw. application	No. of instr.	NFA states	Max. mem. access	DFA states	Mem. entries	Mem. overhead
crc	276	276	2	276	282	2.2%
frag	573	573	3	592	622	8.6%
red	802	802	2	805	847	5.6%
md5	3147	3147	8	3173	3228	2.6%
ssld	828	828	5	829	854	3.1%
wfq	905	905	2	914	953	5.3%
mtc	2427	2427	3	2460	2572	6.0%
mpls-up stream	1603	1603	10	1621	1753	9.4%
mpls-down stream	1574	1574	12	1582	1706	8.4%

The test topology that was used to verify the performance of our monitoring system is shown in Figure 3.15. For hardware experiments, packets were generated and transmitted to the DE4 with the network processor and the monitor at a 1 Gbps line rate by a separate DE4 card serving as a packet generator. This same card was used to receive the processed packets from the card with the NP. The packet generator tool allows for customizing the size and the throughput rate for the test packets.

3.5 Experimental Results

3.5.1 Monitoring Graphs

The results of generating instruction-level monitoring graphs for both our approach and a previous approach [22] are illustrated in Table 3.2. The number of entries in the state machine memory (Figure 3.13) for each benchmark are shown

in the *Mem. entries* column. A clear benefit of the new approach is speed. In all cases, only one access to the monitor memory is required for any benchmark (including the four shown here). The previous NFA-based approach requires up to twelve memory accesses for the benchmarks tested and potentially up to sixteen for other benchmarks. The conversion from an NFA to a DFA does incur a memory overhead of 5.7% on average for the benchmarks. The hash function used to convert a 32-bit instruction to a 4-bit hash value involved the summing of all eight 4-bit instruction nibbles. The result of the summation is the 4-bit hash value.

3.5.2 Evaluation of Hash Functions

As shown in Figure 3.13, each 32-bit instruction is converted to a hash value containing a small number of bits (e.g. $h = 4$). In Section 3.3.1 it is noted that an important aspect of the NFA-to-DFA conversion is limiting the number of cases in which the hash values for multiple edges leaving a state are the same (e.g. Figure 3.10). Limiting these cases avoids the creation of powerset state sets, and the corresponding increase in memory entries in state machine memory. To limit branch hash value collisions, it is desirable for the instruction hashes to be as evenly distributed across the range of possible instructions hashes as possible. Additionally, the 32-bit instruction to h-bit hash value conversion must be simple enough to be performed in one clock cycle.

Four hash functions were considered for this work.

1. **Sum of all ones in the 32-bit instruction** *(bit-sum)* - All binary digits are summed and the result is used to determine the h-bit hash value. For sums exceeding h-bits, only the bottom h bits are used as the hash value.

2. **Sum of all nibbles in the 32-bit instruction** *(nibble-sum)* - All 4-bit nibbles are summed and the result is used to determine the h-bit hash value. For sums exceeding h-bits, only the bottom h bits are used as the hash value.

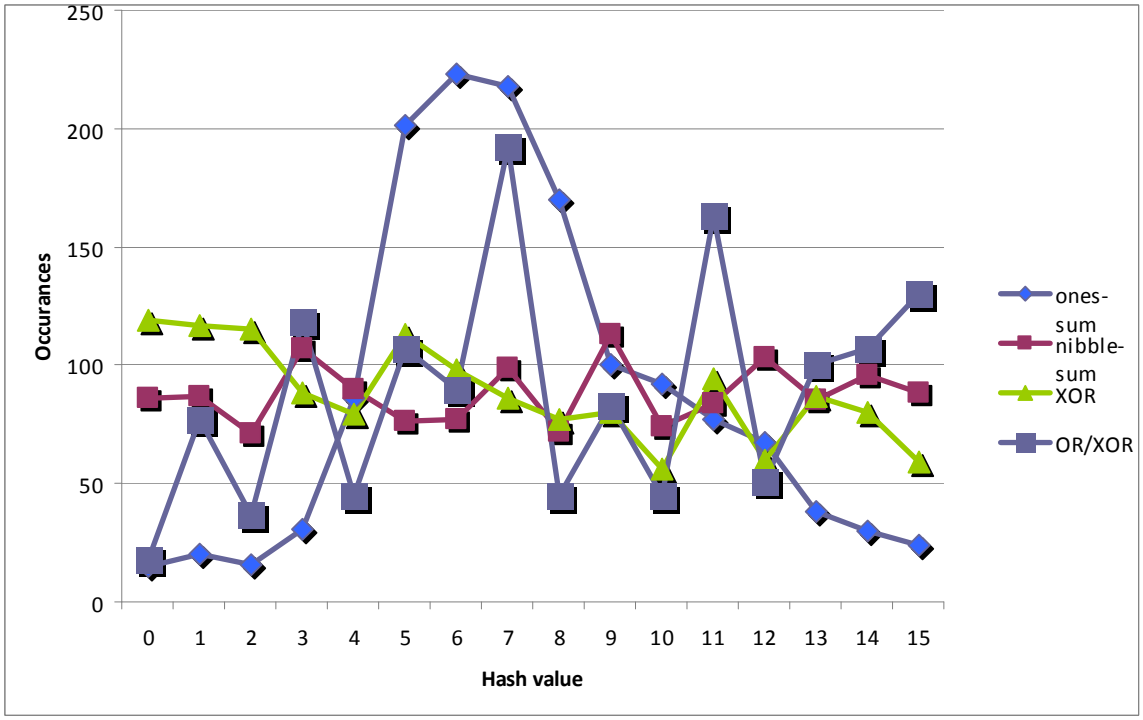

Figure 3.16: Distribution of occurances of each hash value for four hash functions. The results were generated for the *mpls-downstream* benchmark.

3. **XOR of h-bit chunks in the 32-bit instruction** (XOR) - The 32-bit instruction is broken into h-bit chunks which are then XORed together to generate an h-bit result.

4. **OR/XOR of h-bit chunks in the 32-bit instruction** *(OR/XOR)* - the 32-bit instruction is broken into h-bit chunks. Half the chunks are ORed together while the other half (including the final operation) are XORed.

An example of the distribution of 4-bit hash values for all instructions (except NOPs) for the *mpls-downstream* benchmark is shown in Figure 3.16. Plots for other benchmarks are similar. The *nibble-sum* approach to generating hash values is most effective in distributing hash values, although some variation from an even distribution is apparent. This result is likely due to the randomness caused by bit carries in generating the final hash values for *nibble-sum*. Note that the results for hash value

Table 3.3: Comparison of DFA states and state machine memory entries for different hash functions for a 4-bit hash

	nibble-sum		bit-sum			XOR			OR/XOR		
Network application	DFA states	Mem entries	DFA states	Mem entries	%Mem incr.	DFA states	Mem. entries	%Mem incr	DFA states	Mem entries	%Mem incr.
crc	276	282	276	285	1.06	276	282	0.00	279	287	1.77
frag	592	622	592	627	0.80	592	620	-0.32	592	622	0.00
red	805	847	808	857	1.18	806	850	0.35	807	851	0.47
md5	3173	3228	3208	3277	1.52	3181	3248	0.62	3190	3261	1.02
ssld	829	854	836	878	2.81	831	860	0.70	836	875	2.46
wfq	914	953	921	977	2.52	916	955	0.21	918	960	0.73
mtc	2460	2572	2460	2584	0.47	2459	2567	-0.19	2460	2571	-0.04
mpls-upstream	1621	1753	1625	1758	0.29	1622	1744	-0.51	1627	1757	0.23
mpls-downstream	1582	1706	1589	1732	1.52	1579	1694	-0.70	1584	1712	0.35
average					1.35			0.02			0.78

0 in Figure 3.16 do not include the large number of NOP instructions (instruction 0x00000000) in branch delay slots following branch instructions. These instructions are not used as targets for branches and can be omitted from the analysis.

The use of different hash functions directly impacts the required size of the monitoring state machine memory. Table 3.3 shows that the use of the *nibble-sum* hash approach reduces the number of required state memory entries for a range between 0.02% and 1.35% on average. The remainder of the results presented in this section were generated using the *nibble-sum* hash function.

The number of bits used in the hash values also affects the amount of memory required in state machine memory. Although each additional bit added to the hash value decreases the possibility that an attacker could craft a code sequence that has the same hash values as the expected code, the bit effectively doubles the size of the "valid hash values on outgoing edges" field in the memory shown in Figure 3.13. Table 3.4 illustrates the memory overheads for different hash value bit widths using the *nibble-sum* hash approach. The memory size for each hash bit-width is com-

Table 3.4: Comparison of DFA states and state machine memory entries and memory bits for different hash value sizes using the *nibble-sum* hash

	Chasaki [22]	3-bit			4-bit			5-bit		
Network application	Mem bits	Mem entries	Mem bits	Mem Overhead	Mem entries	Mem bits	Mem Overhead	Mem entries	Mem bits	Mem Overhead
crc	8280	288	6048	-27.0%	282	8460	2.2%	282	13254	60.1%
frag	17190	620	13020	-24.3%	622	18660	8.6%	623	29281	70.3%
red	24060	853	17913	-25.5%	847	25410	5.6%	845	39715	65.1%
md5	94410	3255	68355	-27.6%	3228	96840	2.6%	3227	151669	60.6%
ssld	24840	855	17955	-27.7%	854	25620	3.1%	855	40185	61.8%
wfq	27150	957	20097	-26.0%	953	28590	5.3%	951	44697	64.6%
mtc	72810	2590	54390	-25.3%	2572	77160	6.0%	2567	120649	65.7%
mpls-upstream	48090	1783	37443	-22.1%	1753	52590	9.4%	1738	81686	69.9%
mpls-downstream	47220	1727	36267	-23.2%	1706	51180	8.4%	1695	79665	68.7%

pared against the per-application required memory for the earlier Chasaki hardware monitoring approach.

3.5.3 Monitoring Speed and Effectiveness

Our network processor and monitoring system were successfully implemented on the DE4 platform. The lookup table (LUT), flip flop (FF), and memory resources required for the network processor core, monitor, and other interface circuitry for the router (e.g. buffers, input arbiter, queuing control, etc) are shown in Table 3.5. The NP memory includes space for up to 4096 monitor memory entries. All circuitry operated at 125 MHz, the same clock speed for the system without the monitor. Experiments in simulation and in the lab on FPGA hardware showed that the processor is able to forward packets ranging in size from 64 to 1500 bytes per packet at the same rate under monitoring as without monitoring (e.g. no slowdown for monitoring).

We tested the ability of the monitor-based system to detect and recover from an attack. The vulnerable application code shown in Figure 3.3 was implemented

Table 3.5: Memory based high performance hardware monitor resource utilization

Resources	Secure monitor	Network proc.	DE4 interface	Available in FPGA
LUTs	140	3,792	37,803	182,400
FFs	26	2,120	38,444	182,400
Mem. bits	131,072	201,216	2,550,800	14,625,792

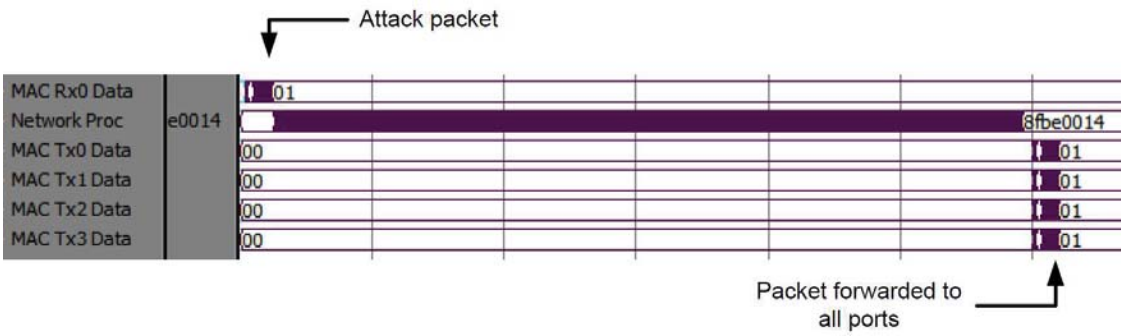

Figure 3.17: Simulation waveforms showing an attack and subsequent forwarding of the packet to all output ports. This behavior was confirmed using hardware.

and used with the NP to send copies of a packet to all ports of the router and then crash the router. We confirmed this behavior for a system without a monitor both in simulation and in hardware. A series of waveforms that demonstrate this behavior appear in Figure 3.17. As shown in Figure 3.18, after the monitor was added to the system, the attack packet was successfully identified, the NP was reset, and subsequent regular packets were routed successfully. This behavior was verified using our DE4 hardware setup.

In a final experiment, we evaluated the throughput and latency performance of our network processor system when attack packets are continuously sent intermingled with regular packets. Both regular packets and attack packets were generated at fixed rates from the packet generator system. The forwarded packets were received back at the packet generator and the throughput and latency were measured. Figure

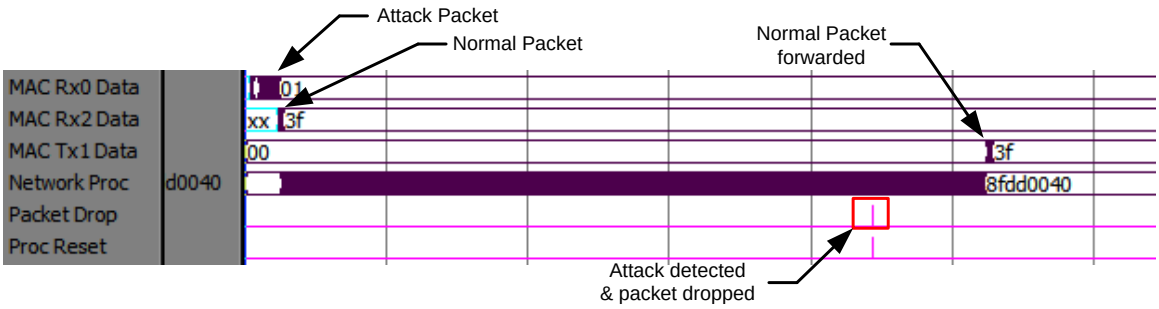

Figure 3.18: Simulation waveforms showing the identification of an attack packet and the successful forwarding of the subsequent packet. This behavior was confirmed using hardware.

3.19 shows the throughput performance of the network processor system for 256-byte packet sizes for varying ratios of regular packets to attack packets. When no attack packets are sent, the throughput of the network processor system increases and reaches a maximum. When attack packets are included the throughput reaches a maximum, and then decreases slightly before settling down. The average latency for 256-byte packets of regular traffic was measured at 104 us.

3.6 Summary and Conclusions

The effective use of the Internet depends on reliable network routers that are impervious to attack. In this chapter, we have explored a new class of data plane attacks in network processors and described a high-performance monitor for these attacks that requires only a single memory lookup per network processor instruction. This single memory lookup is maintained regardless of the complexity of the NP program using an NFA-to-DFA translation of the monitoring graph. This monitor, which tracks individual NP instructions, has been verified in hardware using an NP with a Harvard architecture. The presence of monitoring does not slow down NP operation since it is performed outside of the operational paths of the NP. Some of

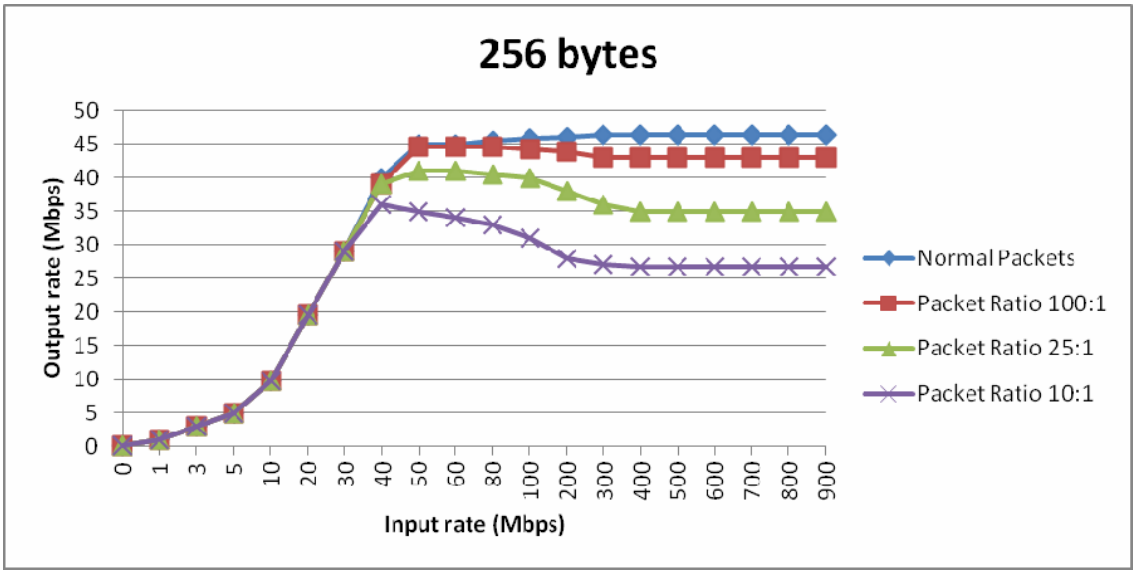

Figure 3.19: NP core throughput performance with security monitor under attack packets.

this material was developed by Harikrishnan Chandrikakutty and originally appeared in [57].

CHAPTER 4

SCALABLE HARDWARE MONITORING FOR MULTICORE NETWORK PROCESSORS

Until now, all the hardware monitor designs have focused on processors with a single processor core executing a single program or a program that changes very infrequently. Network processors, however, use dozens or hundreds of parallel processor cores and have processing workloads that can change dynamically based on the network traffic [86]. Thus, the problem of how to realize an entire *multicore hardware monitoring system* is critical for developing effective protection mechanisms for network processors. This chapter presents the architecture and in-circuit hardware evaluation of a Scalable Hardware Monitoring Grid (SHMG) that provides a solution to this problem. A lightweight interconnection network between processor cores and monitors is dynamically configured to form monitoring connections in response to packet processing needs. In some cases multiple processor cores can share a single monitor, reducing memory overhead. Results developed using both analytical analysis and simulation indicate that our monitoring approach is scalable for network processors containing hundreds of processing cores.

The specific contributions of this work are:

- The design of a scalable architecture for hardware monitors that can be used in a practical network processor system with a large number of processor cores.
- An algorithm which can dynamically allocate monitors to processor cores as application packet workloads change.

- A simulation and analytical analysis of performance of the proposed design at runtime that considers the effects of dynamically assigning processors to monitors and the resulting resource contention.

- A prototype system implementation of a hardware monitoring system on an field programmable gate array (FPGA) platform that illustrates the feasibility of our design and provides detailed resource requirement numbers.

The results indicate that our Scalable Hardware Monitoring Grid and associated allocation algorithm provide a low-overhead and scalable solution for network processor protection against data plane attacks, thus securing Internet infrastructure.

4.1 Scalable Hardware Monitoring Grid

4.1.1 Design Challenges

The development of a scalable monitoring system for multicore network processors has several challenges. The use of monitoring should not impact the throughput or latency of the network processor. For monitors that track individual instructions, each per-instruction monitoring operation must be completed in real time (i.e., during the execution of the instruction), so that deviations from expected program behavior are identified immediately. Additionally, the amount of hardware resources used for monitoring should be limited to the minimum necessary to reduce chip area and power consumption. Since network processor programs may change frequently, it must be possible to modify monitoring tasks for each NP core to accommodate changing workloads.

These challenges necessitate the design of a customized solution for multicore monitoring. Perhaps the most straightforward monitoring approach would be simply to attach a dedicated monitor to each individual NP core, following previous approaches to single-core monitoring, as shown in Figure 4.1. Although this approach minimizes

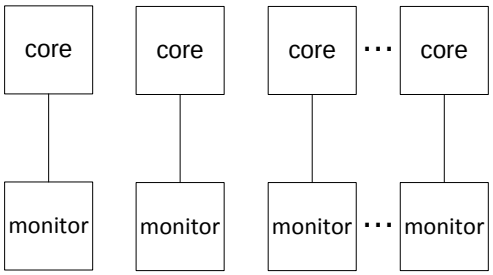

Figure 4.1: One-to-one configuration.

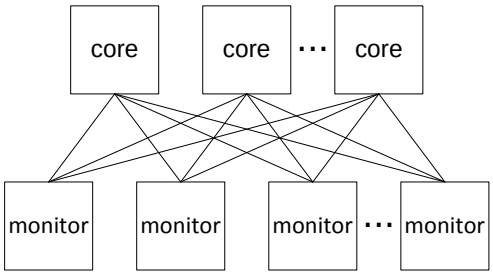

Figure 4.2: Full interconnect configuration.

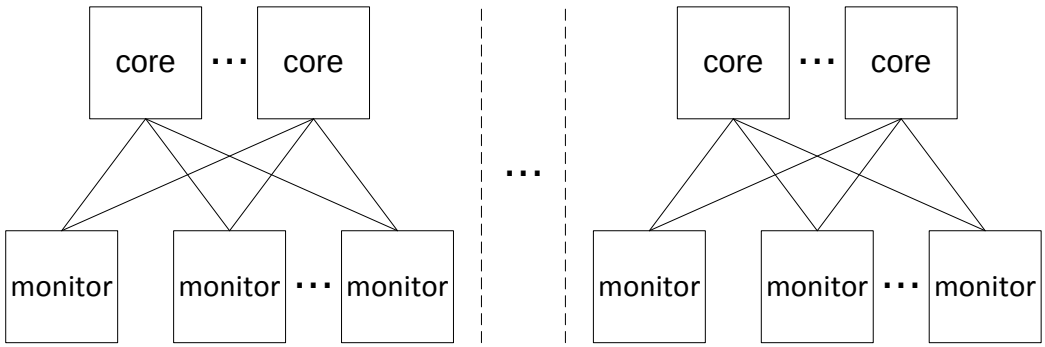

Figure 4.3: Cluster configuration.

the amount of interconnect hardware needed to connect an NP core to a monitor, it suffers from the need to reload monitoring information each time the attached NP core's program is changed. Alternatively, allowing an NP core to dynamically access any monitor among a pool of monitors as shown in Figure 4.2, while flexible, is expensive and incurs a high processor-to-monitor communication cost. In the next section, we describe a scalable monitoring grid system that balances these two concerns of area and performance overhead by using the clustered approach illustrated in Figure 4.3.

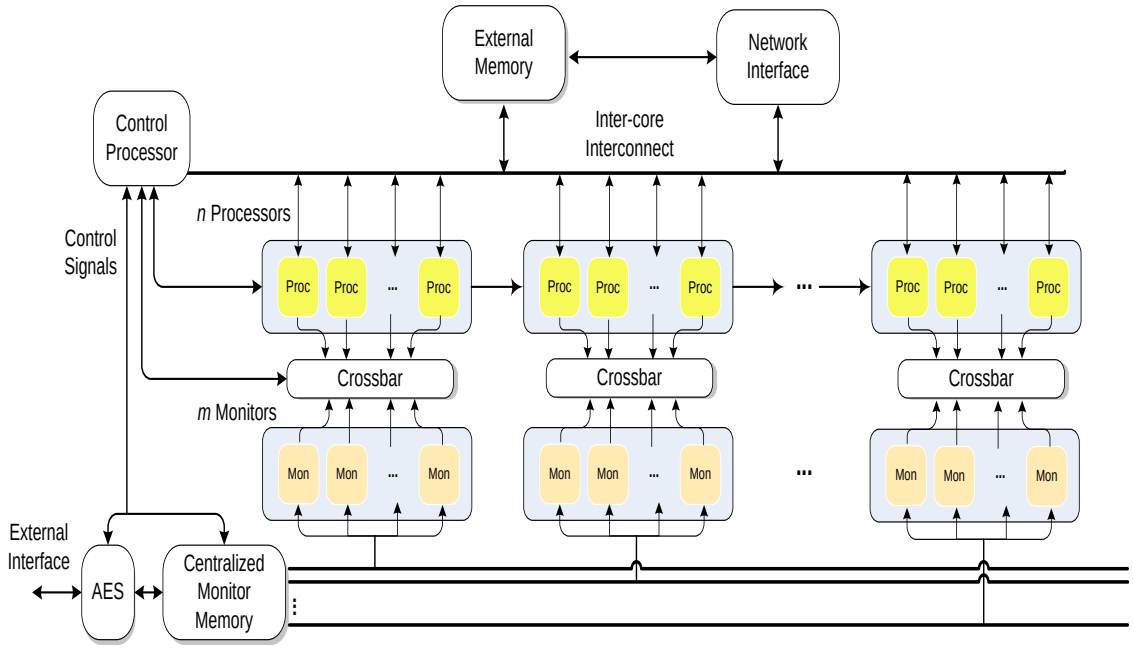

Figure 4.4: Overview of Scalable Hardware Monitoring Grid with network processor cores organized into clusters.

4.1.2 Architecture of Scalable Hardware Monitoring Grid

Our model of the multicore NP system including monitoring is shown in Figure 4.4. The architecture includes a control processor that coordinates overall NP operation by assigning arriving packets to individual NP cores. Each core executes a program using instructions from its local memory. External memory, which can be used to buffer packets and instructions for currently unused programs, is located off-chip. An on-chip interconnect is used to connect cores to external memory and outside interfaces. In this architecture, processors are grouped into *clusters* of n processors. Any of the processors in a cluster can be connected to any of m monitors.

The management of loading application-specific monitoring graphs into monitors and configuring specific processor-to-monitor connections is performed by the same control processor used to assign packets to NP cores. Copies of monitoring graphs for programs that are currently being executed or are likely to be executed in the

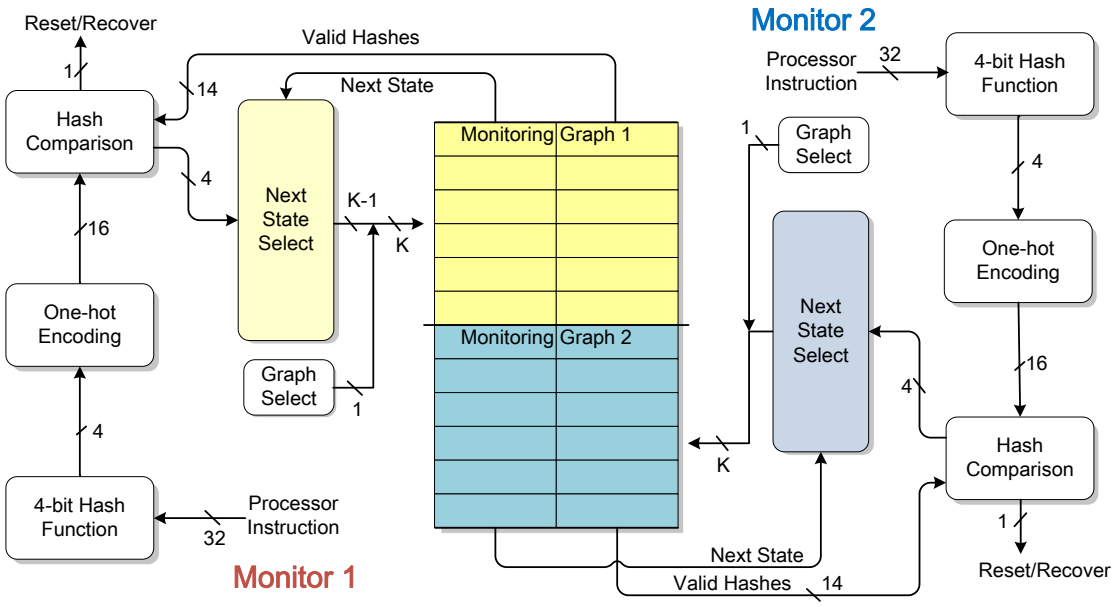

Figure 4.5: Two monitors sharing a single dual-ported memory block.

near future are stored on-chip in a *centralized monitor memory*. Monitoring graph information is encrypted when it is transferred onto the network processor via an external interface. An AES core is used by the control processor to decrypt the graphs and store them in the centralized memory. The amount of time needed to load a monitor with a graph from the centralized monitor memory is significant enough (e.g. tens of clock cycles) that reloading should be minimized. It is desirable to have a program monitor used by different cores at different times during packet processing, necessitating a flexible interconnection between NP cores and monitors. In cases where $m > n$, a total of $m - n$ monitors are unused at a given point in time, although they can be quickly activated in a few clock cycles by the control processor, if needed.

4.1.3 Multi-Ported Hardware Monitor Design

To support scalability, we have optimized the structure of single-processor monitors, which are capable of tracking NP core execution on an instruction-by-instruction basis. The monitoring graph for this class of monitor typically represents each pro-

gram instruction as a state in a state diagram [57]. Expected program execution can be modeled as transitions between known states. To evaluate correct processor operation for an instruction, the progression between states is tracked using instruction hash values. If the hash value of the instruction from the processor does not match the value stored in the monitoring graph for the instruction, a deviation from expected execution flow is detected and the processor is reset. For network processors, this action typically involves a stack reset and a packet drop. The monitoring graph for a program can be determined by analyzing the instruction flow of the program binary. For control flow instructions, multiple next states may be possible in the monitoring graph, requiring matching against several possible hash values.

The architecture of two monitors that perform this type of instruction-by-instruction monitoring is shown in Figure 4.5. The monitoring graph, which is stored in a memory block, includes one entry for each state in the execution state diagram. A k-bit pointer indicates the entry in the graph that corresponds to the currently executed instruction. As an instruction is executed, a four-bit hash value of the instruction is generated, which is then converted to a one-hot encoding. This encoding is then compared against the *expected* hash values that are stored in the graph entry. The next entry (memory row) in the monitoring graph is determined using next state information stored in the current entry and the matched hash value. The implemented monitor requires only one memory lookup per instruction, limiting the time overhead of monitoring.

Although separate hash comparison and next state select information is needed for each monitor, multiple monitoring graphs can be packed into the same memory block if the block is multi-ported (Figure 4.5). In the example, the monitoring graph for the monitor on the left is located in the top half of the memory block while the graph for the monitor on the right is located in the bottom half. For each monitor, the selection of which monitoring graph (top or bottom) is used by the monitor is set by a single

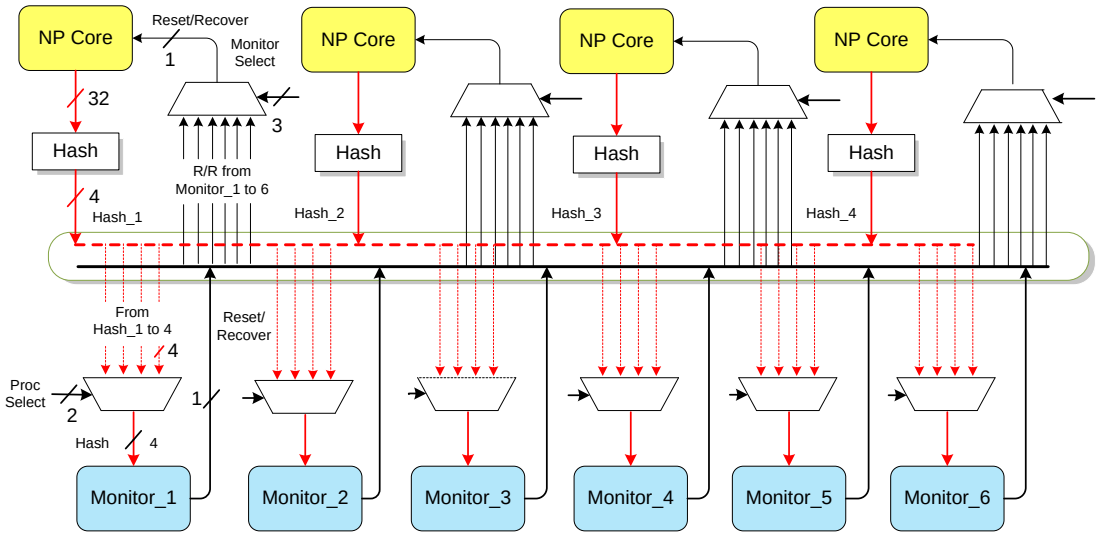

Figure 4.6: Flexible interconnection between processors and monitors in a cluster.

graph select bit which forms the top address bit into the block memory. A benefit of this shared memory block approach is the possibility of both monitors accessing the same monitoring graph at the same time without having to reload monitor memory (e.g. both associated NPs execute the same program and require the same monitor). In this case, the second graph in the memory block would be unused.

4.1.4 Scalable Processor-to-Monitor Interconnection

The detailed interconnection network between a cluster of n processors and m monitors is shown in Figure 4.6. In this architecture, any processor can be connected to any monitor via a series of n-to-1 (processor-to-monitor) and m-to-1 (monitor-to-processor) multiplexers. The four-bit hash values shown in Figure 4.5 are generated from instructions close to the processor, reducing processor-to-monitor interconnect. One of n four-bit values from the processors is selected for a specific monitor using multiplexer $\lceil log\ n \rceil$ select bits. During monitoring, a monitor generates a single reset/recover bit, which is returned to the monitored processor to indicate if an attack has occurred. In our implementation, this signal is sent to the target processor

via a multiplexer with m single-bit inputs. The *monitor* and *processor select* bits are generated by the control processor and sent to the appropriate multiplexers via decoders.

4.2 Runtime Analysis of SHMG

Although the SHMG runtime adaptation can adjust the processing resource distribution at runtime to maximize the system throughput, due to variations in workload, there may be a situation where more processors need to execute a particular program than monitors are available. In this case, some processors temporarily *block* (until a monitor becomes available, at which point they continue processing). We provide a brief analysis of the blocking probability of the system and the resulting throughput for different cluster configurations.

4.2.1 Monitor Configuration

In the n processors, m monitors SHMG system we defined in Section 4.1.2, for each program i ($1 \leq i \leq p$), we assume that t_i represents the average processing time and q_i represents the proportion of traffic that requires this program. We assume $\sum_{i=1}^{p} q_i = 1$, which implies that each packet is processed only by one program. (The analysis can be extended to consider more complex workload configurations.) The total amount of "work," w_i, that the network processor needs to do for each program i is the product of the traffic share and the processing time:

$$w_i = q_i \cdot t_i. \tag{4.1}$$

In order to make the assignment of monitors to programs match the operation of the network processors, we need to determine how many of the n processors are executing program i at any given time. We assume that processors randomly draw from available packets (and thus the associated programs) when they are available. Thus,

the probability of a processor being busy with processing program i, b_i, is proportional to the amount of work, w_i, that is incurred by the program (see Equation 4.1):

$$b_i = \frac{n \cdot w_i}{\sum_{j=1}^{p} w_j}. \tag{4.2}$$

That is, more processors are busy with program i if program i is either used by more traffic or has a longer average processing time.

Monitors should be configured to match the proportions of b_i for each program. The fraction of monitors, a_i, that should be assigned to monitor program i is

$$a_i = max\left(\frac{m}{n} \cdot b_i, 1\right). \tag{4.3}$$

Since each program needs to have at least one monitor assigned to it, the lower bound for a_i is 1.

In practice, the number of monitors per program needs to be an integer. We denote the integer allocation of monitors with A_i. One way to translate from a_i to A_i is to use a max-min fair allocation process.

4.2.2 Blocking Probability and Throughput

Given a monitoring system where A_i monitors are allocated to program i, we need to figure out what the probability is that the number of processors executing program i exceeds A_i (leading to blocking). The number of processors executing program i, B_i, is given by a binomial probability distribution

$$Pr(B_i = k) = \binom{n}{k} \left(\frac{b_i}{n}\right)^k \left(1 - \frac{b_i}{n}\right)^{n-k}. \tag{4.4}$$

The expected number of processors, R_i, that are blocked because of program i not having enough assigned monitors is

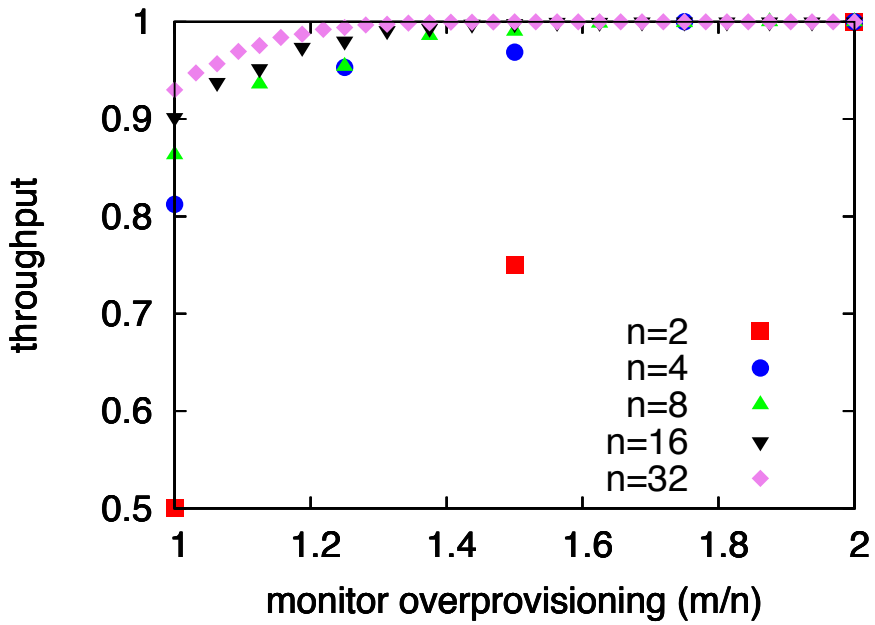

Figure 4.7: Throughput depending on overprovisioning of monitors for different numbers of processors (n).

$$R_i = \sum_{j=A_i+1}^{n} (j - A_i) Pr(B_i = j). \tag{4.5}$$

The total number of blocked processors, R, across all programs is

$$R = \sum_{i=1}^{p} R_i. \tag{4.6}$$

Note that in this case, the probabilities in R_i are not independent since $\sum_{i=1}^{p} B_i = n$.

The fraction of blocked processors is then $\frac{R}{n}$ and the throughput, t, of the system is

$$t = 1 - \frac{R}{n}. \tag{4.7}$$

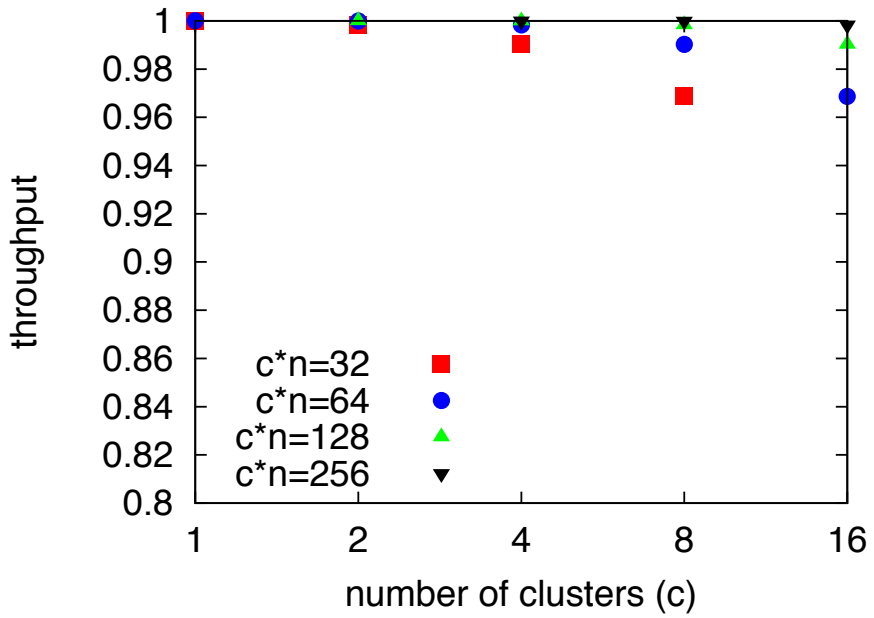

Figure 4.8: Throughput depending on number of clusters for different numbers of total processors ($c \cdot n$).

4.2.3 System Comparison

To illustrate the effect of blocking due to the unavailability of monitoring resources, we present several results based on the above analysis. For simplicity, we assume $p = 2$ programs with $w_1 = w_2$. Figure 4.7 shows the throughput as a function of how many more monitors than processors are in the system. We call this "monitor overprovisioning" (i.e., m/n). In the figure, the overprovisioning factor ranges from 1 (equal number of monitors and processors) to 2 (twice as many monitors as processors). The figure shows that only for very small configurations (e.g., $n = 2$ processors), there is a significant decrease in throughput. For larger configurations, there is only a slight decrease for low overprovisioning factors. For our prototype implementation, we choose a configuration of $n = 4$ processors and $m = 6$ monitors (i.e., $m/n = 1.5$), which achieves a throughput of over 96%.

The effect of clustering is shown in Figure 4.8. Since we need to cluster monitors to achieve scalability in the system implementation, a key question is how much

worse a clustered system performs compared to a system with no clustering (i.e., full interconnect between all processors and monitors). We denote the number of clusters with c. The figure shows the throughput for configurations with the same total number of processors and a monitor overprovisioning factor of 1.5. The full interconnect ($c = 1$) always achieves full throughput. As the number of clusters increases, small systems degrade in throughput slightly. However, if the number of processors per cluster does not drop below 8, throughput of over 99% can be achieved. These results indicate that using a clustered monitoring system instead of a full interconnect can achieve nearly full performance, while being much less costly to implement.

4.3 SHMG Runtime Resource Reallocation

While the previous section provides an analytical evaluation of dynamic resource allocation, system throughput based on varying workloads can also be evaluated through experimentation. In the Scalable Hardware Monitoring Grid design, the control processor (Figure 4.4) assigns programs to processors and monitors. As the traffic workload changes, the optimal assignment of cores and monitors should reflect the processing workload. In order to achieve this goal, a resource allocation algorithm is needed to dynamically reconfigure the SHMG at runtime based on network workload changes.

4.3.1 Reallocation Algorithm

The control processor periodically monitors network workload to assess the current allocation of processing resources. As network packets enter the network processor, they are buffered in the external memory in a series of packet queues. Each queue stores a different type of network packet. The control processor assigns packets to queues based on processing requirements and the number of packets in the queue

defines the queue length. Similar, stable queue lengths for each packet type reflects packet processing balance in terms of number of assigned processors and router processing speed. If the queue length increases significantly beyond the average length, the network traffic is too heavy compared to the current processing speed and more compute resources are needed for this program.

We assume the total input network traffic is fully utilized, i.e., when the workload increases for one packet type, the workloads of other packet types will be reduced. As mentioned in Section 4.1.2, a processor/monitor cluster consists of n processors and m monitors. The workload of the system consists of p different programs that each monitor may execute (one program per packet type). For practicality, we assume $m \geq n$ and $m \geq p$. A fully utilized system where no processor is idle is considered.

For each program i ($1 \leq i \leq p$), a_i is the number of processors assigned to the program and $ql_i(t)$ is the queue length at time t. If queue length ql_i increases and exceeds threshold θ, the required packet processing exceeds the current processing power and more processing resources need to be allocated to this program. Our algorithm performs this process in two steps: First, the algorithm examines all p queues to locate a program j which can release resources to program i; second, the algorithm determines which monitoring resources to allocate to the reassigned program (and thus which cluster is used). Each step is explained in detail in the following.

4.3.1.1 Identification of Program for Reallocation

In order to find the most suitable program j, the following criteria are applied during the search:

1. If a queue is empty, select this program to release one processor. If there is more than one empty queue, select the program that has had an empty queue for the longest time. For this purpose, an empty time marker te_k is used for each empty queue k to record the time the queue drained. The algorithm maintains

a priority queue for the empty queue that allows easy identification of the queue that has been empty longest (i.e., with minimum te_k).

2. If no queue is empty, select the program with the shortest queue length that has at least two processors allocated. (Each program is guaranteed at least one active processor in the system if it has a non-zero queue length, so deallocating resources from a program with a single processor is not allowed.)

This queue monitoring algorithm is shown in Algorithm 1. A program that needs an additional core is i and the program that releases a core is j. When a new packet is assigned to queue i, the algorithm assesses the queue length ql_i. If ql_i passes the threshold θ, an additional processor is assigned to program i. The algorithm examines the length of all queues to find program j based on the above criteria.

Algorithm 1 Queue Monitoring

```
1: when ql_i(t) + 1                                   ▷ When a new packet comes
2: if ql_i(t) ≥ ε then                                ▷ If queue passes threshold
3:     Prog_a ← i
4:     Prog_b ← 0
5:     te_max ← 0
6:     for j = 0 to p do                              ▷ Evaluate all queues
7:         if ql_j(t) = 0 then                        ▷ If queue empty
8:             if te_j ≥ te_max then                  ▷ And max empty time
9:                 Prog_b ← j
10:                te_max ← te_j
11:                ql_min ← 0
12:            end if
13:        else if ql_j(t) ≤ ql_min then
14:            if B_j ≥ 2 then ▷ find the shortest queue with more than 1 processors
15:                Prog_b ← j
16:                ql_min ← ql_j
17:            end if
18:        end if
19:    end for
20: end if
```

4.3.1.2 Identification of Monitor for Reallocation

After i and j have been determined, the next step is to select a specific system processor to switch from program j to program i. To minimize monitor reloading during the switching process, the selection is made as follows:

1. Identify all unused monitors in the system.

2. If there is an unused monitor that has a preloaded graph of program i, identify all the processors in the same cluster as this monitor. If there is a processor running program j in the same cluster, switch it to program i, disconnect the program j monitor and connect the processor to the program i monitor.

3. If there is no processor running program j in the same cluster, try to find another unused monitor with program i in a different cluster.

4. If there is no unused monitor that has preloaded program i, switch one processer j to program i and reload the least recently used monitor to program i in the same cluster of the switched processor.

After switching resources, several packets must be processed before the effect of the new configuration reflects on the queue lengths. To prevent additional programs from passing the threshold soon after resource switching and taking processing resources from the same program j, a mandatory delay δ is introduced. After one adaptation, new switching requests will be blocked until δ packets are processed.

4.3.1.3 Reallocation Algorithm Complexity

Overall, the runtime resource reallocation algorithm (RRRA) has two traversal operations:

- Evaluate all p queues to find j when the threshold θ is exceeded by a program i. This action which requires $O(p)$ time.

- Evaluate all monitors in the clusters that include program j to find a monitor and processor core to use or switch functionality, which requires $O(m+n)$ time.

In total, RRRA has an asymptotic complexity of $O(p+m+n)$, which is linear in the number of programs, monitors, and cores in the system.

4.3.2 System Simulation

A Java-based simulator was built to verify RRRA and evaluate runtime throughput results in comparison to the values determined with the analytical model in Section 4.2.3. The simulator can generate quantities of network packets in different ratios, and vary the ratios with time. With this time-changing input network traffic, the simulator assesses the behavior of RRRA and measures the runtime resource allocation and system throughput.

Figure 4.9 shows balanced network traffic (e.g. the total number of packets during a fixed time period is the same) for three different kinds of packets. The packet proportions change from 1: 1: 1 to 8: 1: 1, then to 1: 1: 8, and finally back to 1: 1: 1 and then decrease to 0. With this input network traffic, for simplicity, we assume the packet processing time for different packets are equal $t_1 = t_2 = t_3$. Figure 4.10 shows the number of processors running each program during the runtime in an experimental system with 16 processors and 24 monitors. The fact that the ratios of the processors assigned to each program follows the ratios of each packet types in the network traffic validates the effectiveness of the RRRA. In Figure 4.11, the system throughput is a maximum shortly after system processing starts, then it has three obvious transitions corresponding to the three network traffic allocation changes. The biggest drop happens when the network traffic changes dramatically from 8: 1: 1 to 1: 1: 8. With the RRRA, the system can adapt its resource allocation every time the traffic changes and the throughput quickly returns to a maximum value.

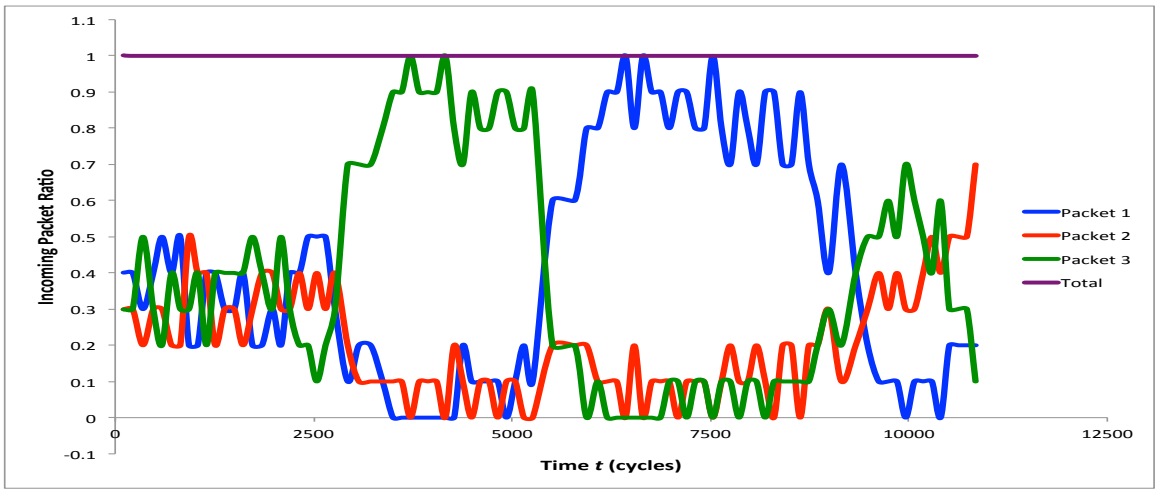

Figure 4.9: Input network traffic used during simulation.

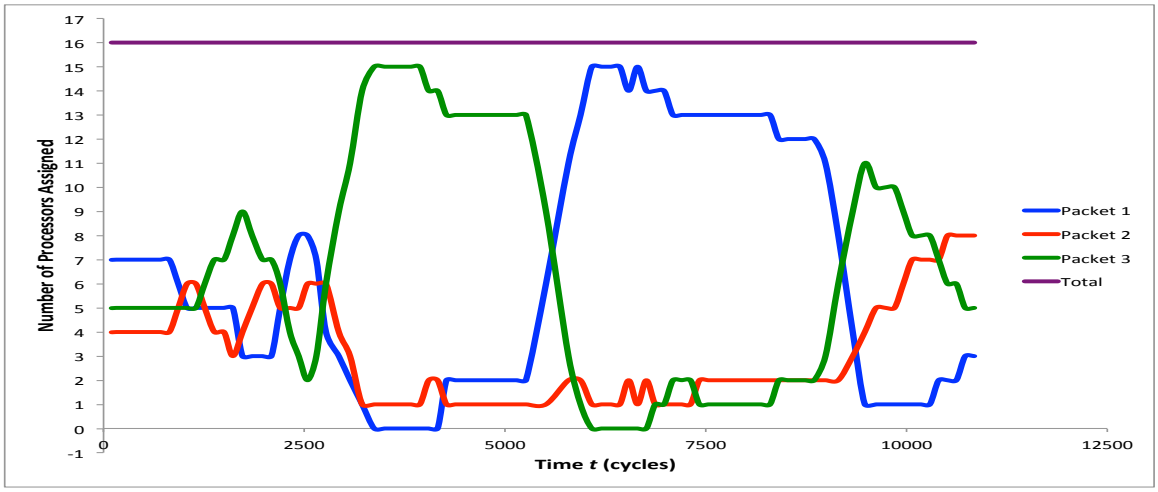

Figure 4.10: Processor distribution for different packet types as traffic allocation changes.

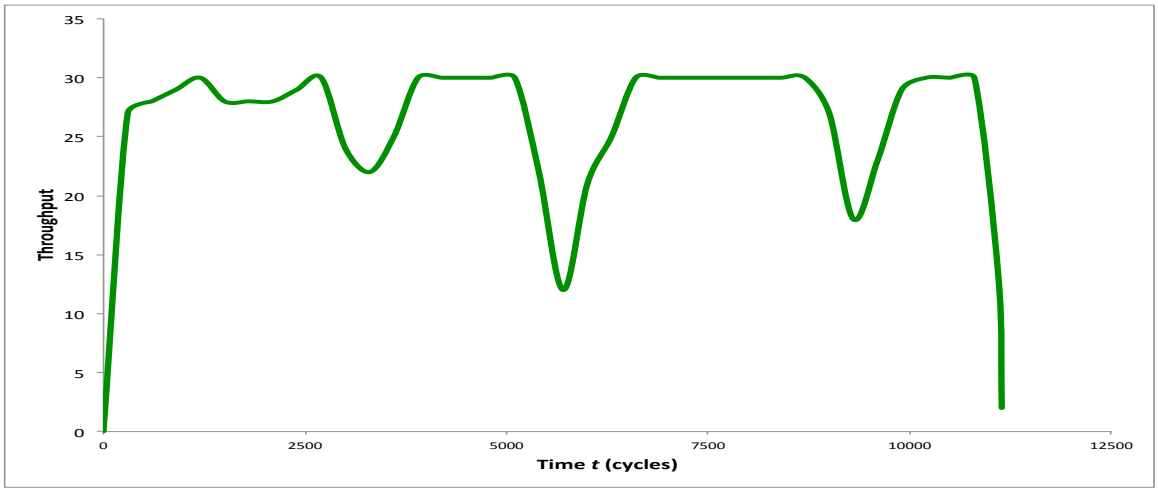

Figure 4.11: Throughput changes is relation to traffic changes.

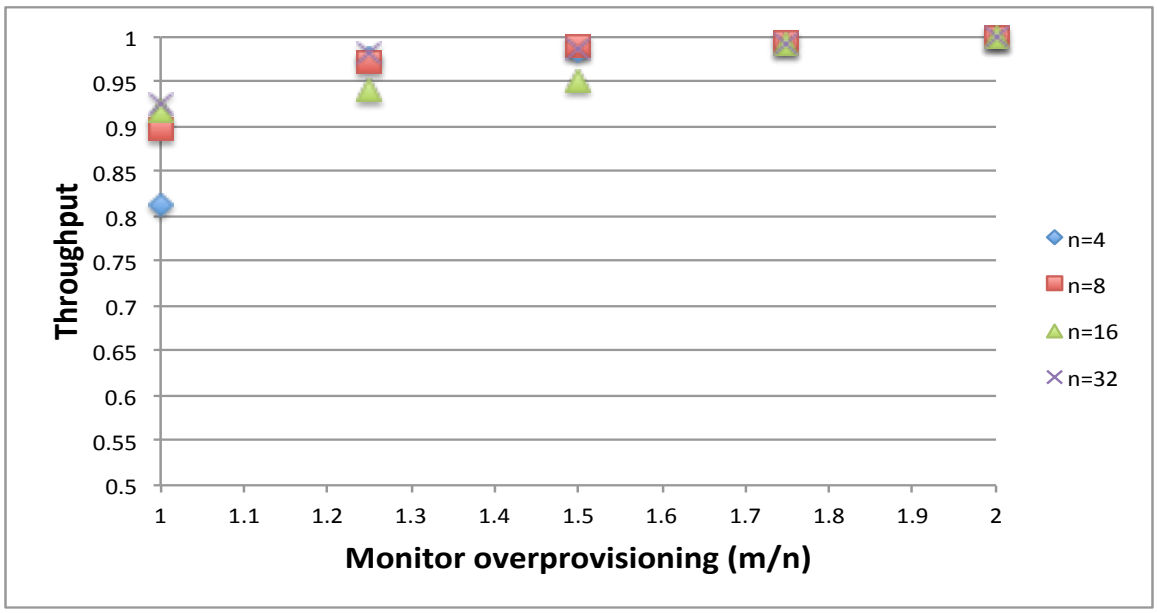

Figure 4.12: Simulation throughput depending on overprovisioning of monitors for different numbers of processors (n) with two different packets.

To verify the monitor overprovisioning analysis, two experiments were conducted in simulation. The first experiment considered the simplest case that $p = 2$ with $w_1 = w_2$, total processor number $n = 4, 8, 16, 32$, and the overprovisioning factor ranges from 1 to 2. The results shown in Figure 4.12 match with the previous analysis results in Figure 4.7. The second experiment kept the assumption of evenly distributed workload, but extended the program number to three and the overprovisioning factor range from 1 to 3. Results shown in Figure 4.13 indicate the same throughput trend as the two programs experiment. Although the upper bound of the overprovisioning range increases to 3, the throughput is still more than 95% after $m/n = 1.5$.

The effect of number of clusters on throughput was also measured in simulation. The experiment was setup with 1.5 monitor overprovisioning, the same as the previous analysis, and the measured total number of processors were 32, 64, 128 and 256. Compared to the analysis results in Figure 4.8, results from this experiment shown in Figure 4.14 demonstrate good consistency.

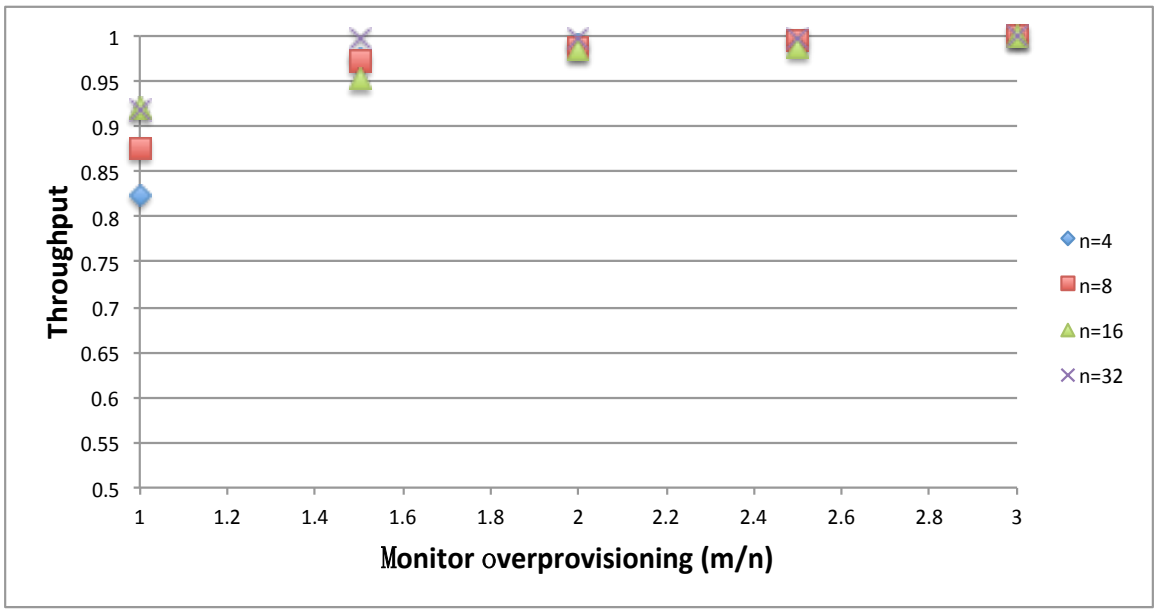

Figure 4.13: Simulation throughput depending on overprovisioning of monitors for different numbers of processors (n) with three different packets.

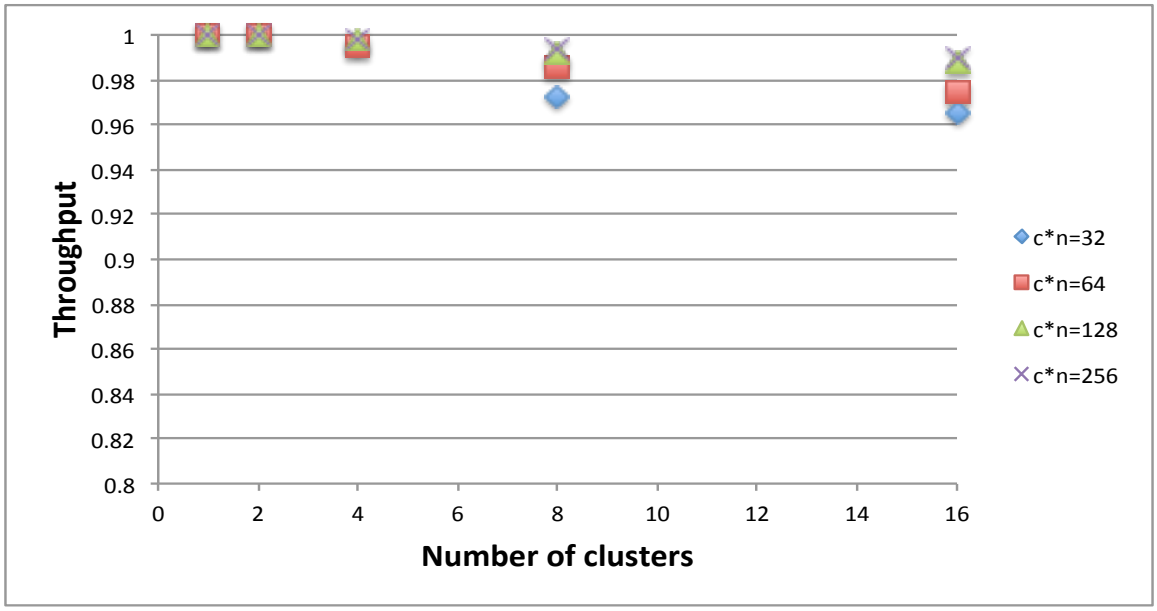

Figure 4.14: Simulation throughput depending on number of clusters for different numbers of processors (n) with two different packets.

4.4 Prototype Implementation and Evaluation

To demonstrate the effectiveness of our Scalable Hardware Monitoring Grid in a real system, we have implemented a prototype system.

4.4.1 Experimental Setup

We have implemented a prototype network processor in an Altera Stratix IV FPGA on an Altera DE4 board. This board contains four 1 Gbps Ethernet ports to receive and send network traffic. We implemented one SHMG cluster in the FPGA, consisting of four processor cores (soft processors created using a synthesizable PLASMA processor [76]) and six hardware monitors (i.e., $n = 4$ and $m = 6$). The flexible, multiplexer-based interconnect shown in Figure 4.6 is used to allow any processor to connect to any monitor within our cluster.

To evaluate the functionality and performance of the monitoring system, we transmit traffic through the prototype system. Packets are received on two of the Ethernet ports and transmitted on the other two. For each packet, a simple flow classifier determines the appropriate NP program for processing. After the packet is processed by a core, it is sent to the appropriate output queue for subsequent transmission.

We use two types of packets, which need different types of processing and thus different monitors: (1) IPv4 packets and (2) IPv4/UDP packets that require congestion management (CM) for processing. The processing code for IPv4 does not exhibit vulnerabilities, but the IPv4+CM processing code exhibits the integer overflow vulnerability described in Section 3.1. We introduce 1% of attack packets, which can trigger a stack smashing attack in the IPv4+CM processing code.

To generate the monitoring graph, the program is first passed through the standard MIPS-GCC compiler flow to generate assembly-level instructions. The compiler output allows the identification of branch instructions and their branch target addresses. The instructions and branch information are then processed to generate the

data structure used inside the hardware monitor. This data structure is then loaded into the SHMG system.

4.4.2 Experimental Results

Our system was verified through a series of experiments that were run on the FPGA in real time.

4.4.2.1 Correct Operation

To illustrate the operation of our SHMG, we have assigned two cores to process IPv4 and two cores to process IPv4+CM. Of the available six monitors, two are configured to monitor IPv4 and four are configured to monitor IPv4+CM (since the latter is more processing-intensive). All four NP cores execute program code from internal FPGA memory. The initial configuration of the monitors, program code, and the processor-to-monitor interconnect is set when the design is compiled to the FPGA and the bitstream is loaded into the design on system powerup.

Figure 4.15 shows the operation of a processor core and its corresponding monitor on the IPv4 program. (Waveform figures are generated through simulation in order to obtain signals; however, the same functionality has been verified in real-time operation of the system on network traffic.) Similarly, Figure 4.16 shows the operation of a core on the IPv4+CM program. In this case, the packet is benign and no attack occurs. Figure 4.17 shows the processing of an attack packet in IPv4+CM. The monitor identifies the attack since the stack gets smashed and the control flow is redirected to code that differs from what the program analysis has determined as valid. The processor core is then reset and continues processing the next packet. The reset operation completes in two cycles and thus does not affect the throughput performance of the system (and cannot be used as a target for denial of service attacks). Other processor cores continue processing without being affected.

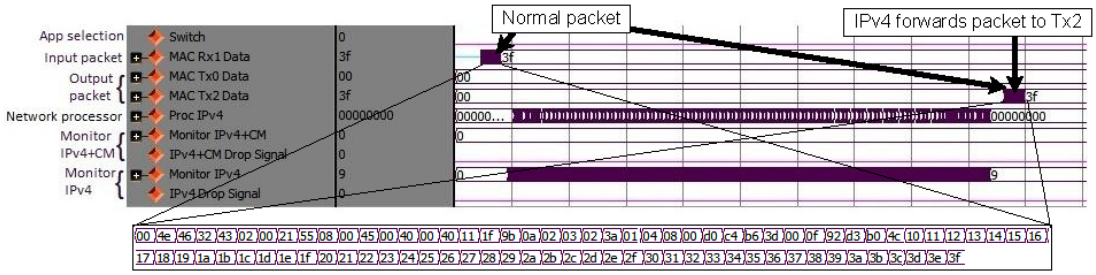

Figure 4.15: Simulation waveforms showing correct forwarding of an IPv4 packet.

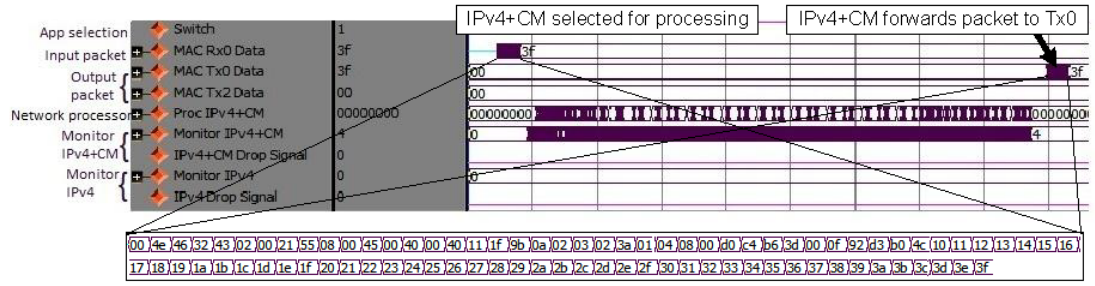

Figure 4.16: Simulation waveforms showing forwarding of an IPv4+CM packet.

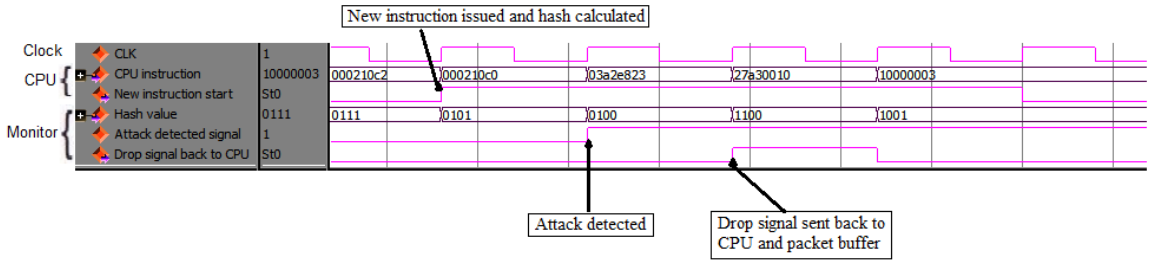

Figure 4.17: Simulation waveforms showing identification of and recovery from an IPv4+CM attack packet.

A key functionality of SHMG is the dynamic assignment of processors to hardware monitors. In our prototype system, we can trigger the reassignment of processors to monitors on-demand. In our experimental setup, we switch one of the processor cores from IPv4 (Figure 4.15) to IPv4+CM (Figure 4.16).

The processor-to-monitor interconnect for the core that was previously processing IPv4 packets is switched to connect the core to an unused IPv4+CM monitor. The affected NP core and newly connected monitor are then reset, and processing by the core commences. After this run-time reconfiguration, three NP cores process packets for IPv4+CM, while one core processes IPv4.

Thus, we are able to show dynamic reassignment of processors to monitors at runtime as well as the correct detection of and recovery from attacks.

4.4.2.2 Resource Requirements

The resource requirements for the FPGA in our prototype system are shown in Table 4.1. The lookup table (LUT), flip flop (FF), and memory resources (Bits) required for the network processor cores, monitors, switches and other circuitry are illustrated shown in Table 4.1. A LUT is a n-input, 1-output logic element that can perform any logic function of n inputs. Each monitoring graph can hold up to 4096 separate entries. The FPGA in the system is able to operate at 125 MHz. For this relatively small cluster size, the amount of logic needed for processor-to-monitor interconnection is less than 1% of the total logic needed for the monitors, cores, and processor-to-monitor interconnect since only hash value, reset, and control signals are communicated.

To assess the generality of our area results across different FPGA generations, we resynthesized the network processor cores, monitors, and interconnect to an Altera Stratix II device. The resulting LUT counts of 14,912, 774, and 92 for the processor cores, monitors, and interconnect, respectively are similar to the Stratix IV numbers.

Table 4.1: Resource utilization and dynamic power consumption in the prototype system

	Available in FPGA	DE4 interface	Network processors	SHMG	
				monitors	intrcon.
LUTs	182,400	33,427	15,025	816	96
	-	67.8%	30.4%	1.7%	0.1%
FFs	182,400	36,467	8,367	147	0
Bits	14,625,792	2,263,888	2,097,134	786,432	0
	-	44.0%	40.7%	15.3%	0%
Pwr (mW)	-	1490.83	388.6	41.76	5.30

For a Stratix II device, an LUT can range in size from 2-input to 7-input depending on the desired logic function. The distribution of input counts for LUTs across this input spectrum was similar for both architectures.

The dynamic power consumption of the components, shown in Table 4.1, was determined using the Altera PowerPlay power analyzer. The monitors and associated interconnect consumed 12% of the dynamic power of the processors. The network and PCI interfaces on the board consumed 3.4× more dynamic power than these components combined. Based on board level experimentation, average time to process one hundred 256 byte packets is 6 ms. As a result, the dynamic energy to process 100 256-byte packets at 125 MHz is 6 ms × 1926.49 mW = 11.56 mJ.

The throughput of our system including monitoring when processing normal 256-byte packets and an occasional attack packet is shown in Figure 4.18. The throughout of the system is limited by the processing capability of the processor cores, not monitoring. The throughput for normal packets is the same both with and without monitoring. A small throughput reduction is observed in the presence of attack packets due to the amount of time needed to flush the packet buffer.

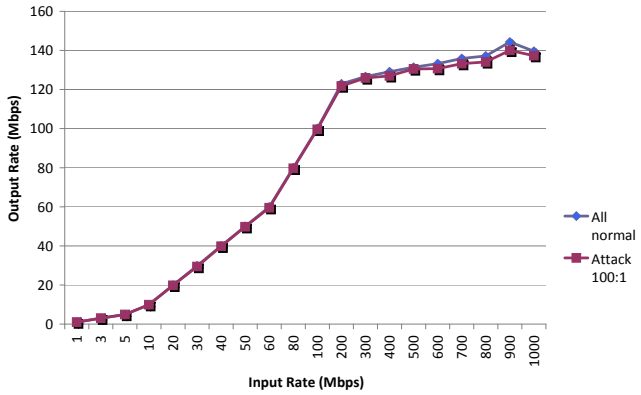

Figure 4.18: Throughput results when processing normal IPv4 with congestion management packets and when processing IPv4 with congestion management packets if one out of 100 packets is an attack packet.

4.4.2.3 Monitoring Graph Swap Time Overhead

To better illustrate the benefits of overprovisioning the monitors relative to processor count ($m > n$), we assess the average time required to swap monitors during a processor allocation for the case when $m/n = 1.5$ versus the case when a monitoring graph must be reloaded from centralized monitor memory for every processor reallocation. The steps required to perform each task of monitor swapping includes: identification of a new program i for allocation, identification of a program to swap out (j), identification of a target processor core and monitor for the new program, and monitor reload from centralized monitor memory (if needed). The reallocation operations needed to perform the first three steps in the list were discussed in Section 4.3. The following analysis is performed for a two cluster system with $n = 6$ processor cores and $m = 9$ monitors in each cluster. The control processor operates at 125 MHz, the clock speed for our prototype hardware implementation. Since processor throughput is one clock cycle, we equate an instruction execution to a clock cycle. The instructions for the programs are stored in each processor core's local memory.

The initial allocation and deallocation steps require examination of packet queues to identify a program i for allocation to a processor and the reduction of one processor

for a program j (Section 4.3.1.1). Our experimentation using system simulation shows that 28 control processor instructions are needed on average to identify a program i which requires an additional processor. An additional 28 control processor instructions are required to identify a program j that should have a processor deallocated. Combined, these actions require 0.45 μs.

The tasks needed to identify a monitor for reallocation are detailed in Section 4.3.1.2. This stage attempts to identify a spare monitor which has been previously loaded with the monitoring graph for program i and a processor core which is currently tasked with program j. This core is subsequently switched to program i and the processor core/monitor interconnect is configured for the new connection. The process of identifying a processor core, swapping its program, and locating a suitable preloaded monitor requires 197 instructions (clock cycles) on average, based on our simulation. The configuration of the interconnect between the monitor and processor core in the cluster requires 3 clock cycles. In total, these actions require 1.60 μs.

In some cases, if a spare monitor with the appropriate graph cannot be found, a graph must be loaded into monitor memory. To evaluate the average monitoring graph reloading cost from centralized monitor memory to the dual-ported memory in a monitor, nine benchmarks from the NpBench suite [60] were processed with an offline analysis flow. NpBench is a benchmark suite targeting modern network processor applications. The benchmark applications are categorized into three specific functional groups: traffic management and quality of service group (TQG), security and media processing group (SMG) and packet processing group (PPG). In our evaluation, monitor graph sizes generated with a 4-bit nibble-sum hash function were calculated and the graph read/write times to an on-chip memory were estimated for each of the benchmarks. The reloading time estimation was based on on-chip SRAM which is a 200 MHz SSRAM with a 16-bit data bus. Table 4.2 shows the evaluation results. The average reload time is found to be 13.34 μs.

Table 4.2: NpBench monitor graph reload cost

Network benchmark	Memory graph size (bits)	Graph reload time (cycles)	Graph reload time (ms)
crc	8460	529	2.64
frag	18660	1166	5.83
red	25410	1588	7.94
md5	96840	6052	30.26
ssld	25620	1601	8.01
wfq	28590	1787	8.93
mtc	77160	4822	24.11
mpls upstream	52590	3287	16.43
mpls downstream	51180	3199	15.99

Based on our simulation, we determined that it was necessary to reload a monitor from centralized monitor memory 16% of the time during a reallocation for $m/n = 1.5$. During the remaining cases, a spare monitor with program i was available in a cluster and could be connected to the newly-allocated processor core. As a result, the average amount of time needed to reallocate a processor core can be calculated as *program allocation time + monitor/processor identification time + %reload* $\times$ *graph reload time.* In total, this analysis results in an average reallocation time of 0.45 μs + 1.60 μs + 0.16 $\times$ 13.34 μs = 4.18 μs. In contrast, the amount of time needed if a processor is dedicated to a monitor is *program reallocation time + graph reload time.* In total, for the $m = n$ case, this analysis results in an average 0.45 μs + 13.34 μs = 13.79 μs delay.

4.5 Summary and Conclusions

To provide practical protection for network processors, which are multi-core systems with highly dynamic workloads, we have presented our design of a Scalable

Hardware Monitoring Grid in this chapter. This monitoring system groups multiple processors and monitors into clusters and provides an interconnect to dynamically assign processor cores to monitors based on their current workload. We present the hardware design of an efficient interconnect for these clusters and show through analysis that even small configurations can achieve throughput performance. We also present the results from an FPGA prototype implementation that shows the correct operation of our system and the ability to perform dynamic assignment of processor cores to monitors. We show that the system can correctly identify attacks and recover the attack core so that it can continue processing. The system overhead for our monitoring system is less than 6% compared to the processor system. Thus, our Scalable Hardware Monitoring Grid provides an effective and efficient mechanism for defending network infrastructure from a new class of attacks.

CHAPTER 5

SYSTEM-LEVEL SECURITY FOR NETWORK PROCESSORS WITH HARDWARE MONITORS

While the design and operation of network processor cores with hardware monitors running one or a small number of preconfigured applications is well-understood, there is also a need to look at the *system-level perspective* of the problem. There are two key challenges for a practical hardware monitoring system for network processors: (1) *Dynamics:* multiple processor cores and their monitors need to be managed and reprogrammed at runtime as network traffic and network functionality change. (2) *Homogeneity:* To simplify management, practical networks use large numbers of identical router devices, which can lead to Internet-scale failures in case an attack can be developed circumventing one specific monitoring system. This chapter addresses these system-level issues and presents the design of a monitoring system that can securely install binaries and monitors on network processors and parameterize these configurations such that potentially successful attacks cannot propagate.

The specific contributions of this work are:

- Design of a Secure Dynamic Multicore Hardware Monitoring System (SDM-Mon), which enables secure installation of binaries and monitors on network processor systems based on cryptographic principles and suitable key management.

- Design of a novel, high-performance, parameterizable hash function for use in hardware monitors enabling the deployment of diverse monitoring systems that are not susceptible to the same potential attack.

- Evaluation of a prototype system implementation showing SDMMon functionality and performance.

In this chapter, we first discuss our security model. Then we present the design of our system-level architecture followed by results from our prototype and related work.

5.1 Security Model

One key requirement for the security of the hardware monitoring system is that an attacker cannot modify the hardware monitoring graph. If the attacker could modify the monitoring graphs, an attack could be hidden by substituting the correct monitoring graph with one that would accept the attack as valid code. Prior and related work have not suitably addressed the question of how the monitoring graph is installed in a hardware monitoring system. For embedded systems that execute a single, unchanging binary (as was assumed in prior and related work), a one-time installation of the monitoring graph through a dedicated interface can be assumed. However, for network processors systems that need to dynamically download new processing code, there is no existing suitable solution.

In the following, we focus on how to achieve the secure installation of valid hardware monitoring graphs. We do not consider security issues relating to the hardware monitor itself since these issues have been addressed in prior chapters. Instead, the focus is on the security issues relating to dynamically installing monitoring graphs onto network processor systems while considering that attackers may tamper with this process to be able to launch attacks that accept malicious code as valid.

To make the security model realistic in the context of practical network operation, we consider three entities that are part of the system environment:

- Network processor manufacturer: The manufacturer produces network processors and router systems and sells them to the network operator. In some cases,

the network processor is manufactured by a different party and then integrated on the router system by the router manufacturer. For simplicity, we assume here that the same entity produces the router and the network processor on it.

- Network operator: The operator purchases the router system with network processors from the manufacturer and programs its operation for the network where it is used.

- Network processor device: The network processor device is programmed by the network operator. That is, the network processor needs to obtain processing binaries and monitoring graphs from the network operator.

5.1.1 Security Requirements

The specific system-level security requirements for a network processor system with hardware monitors are:

SR1 Only valid binaries and matching hardware monitor graphs should be installed on the network processor. Validity implies that the binary and monitor have been authenticated as being sourced from the network processor's network operator.

SR2 Hardware monitoring mechanisms should be sufficiently diverse – despite the operation of identical binaries – to avoid catastrophic failures in a highly homogeneous network environment in case of a successful attack.

SR3 Binaries, monitoring graphs, and hash parameters should be confidential to prevent an attacker from obtaining a hash parameter and from "stealing" the intellectual property of a binary.

SR4 Binaries and monitor graphs should only be identified as valid on one specific network processor system. This security requirement helps in preventing an attacker from injecting an binary from a different device.

5.1.2 Attacker Capabilities

We assume that attackers can do the following:

AC1 An attacker can observe any traffic and inject any type of traffic. To limit the scope of this work, we do not consider a case where an attacker can block all traffic on a link. This problem can be addressed through other techniques (e.g., multipath transmissions).

AC2 An attacker can generate a monitoring graph that matches a binary that is attacked with an attack chosen by the attacker.

To enable us to find a solution to this security problem, we also need to constrain the abilities of the attacker. Specifically, we assume the following limitations:

AC3 An attacker cannot obtain cryptographic keys stored by any of the three entities.

AC4 An attacker cannot break standard symmetric and asymmetric cryptographic algorithms.

These limitations also imply that we do not consider physical or side-channel attacks. While such attacks may exist, there is ongoing research to develop suitable protection mechanisms.

5.2 System-Level Architecture

The system architecture for SDMMon is shown in Figure 5.1. The main difference versus conventional hardware monitoring approaches, such as shown in Figure 3.5, is the use of signed binaries, monitoring graphs, and hash function parameters. This approach protects the system from attacks, as we discuss below.

A critical aspect for security and operational functionality is the set up of keys and cryptographic operations. Figure 5.2 shows the three entities that we consider for our work, the router manufacturer, the network operator, and the network processor

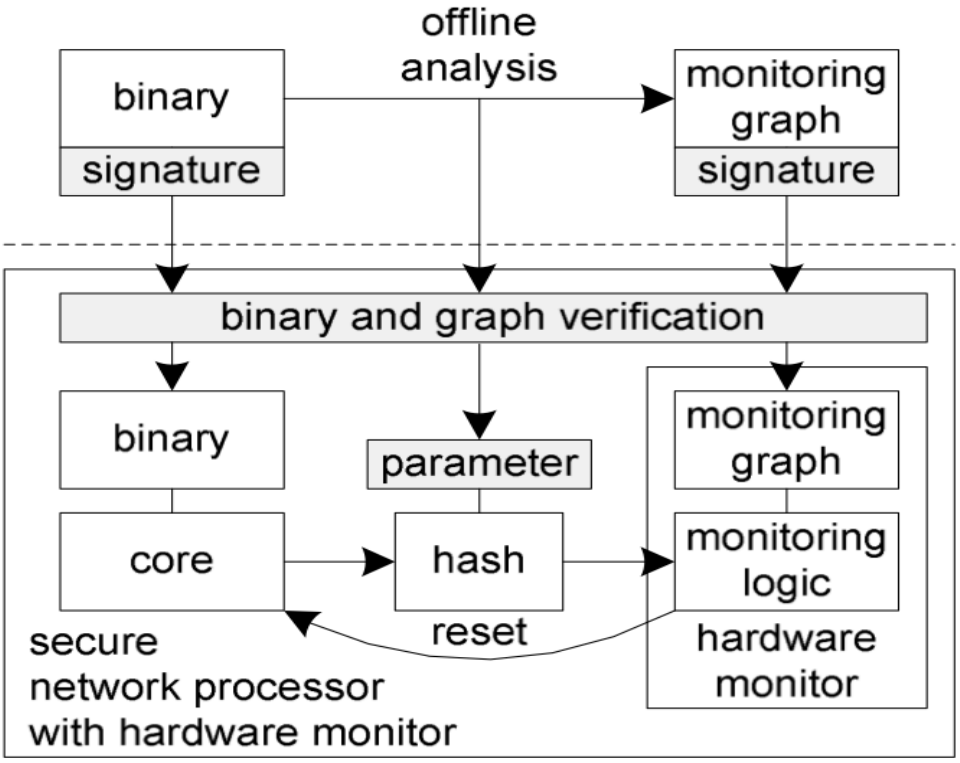

Figure 5.1: Hardware monitor with system-level security. Application binaries and monitoring graphs are signed to ensure authenticity and integrity. In addition, the hash function for each network processor core is parameterized differently to achieve heterogeneity.

device. In the following, we explain the interactions between these entities and how they achieve the required security properties.

5.2.1 Operation and Key Management

The following operations describe the interactions between the entities:

- At manufacturing time: During initial setup of the network processor, the manufacturer configures the device with a public key/private key pair (denoted as K_R^+ and K_R^-). The manufacturer also installs the manufacturer's public key (K_M^+) into the device so that a root of trust can be established. The keys can be stored in hardware logic or a trusted platform module.

- At installation time: When the network processor is installed in a network operator's network, the manufacturer provides a certificate that contains (at least) the network operator's public key signed with the manufacturer's private key. Using this certificate, the network processor can establish a chain of trust to the network operator. This certificate may be sent to the network processor once at boot time or with every reprogramming step.

- At programming time: To program the network processor, the network operator generates a monitoring graph obtained from the processing binary. The monitoring graph is then parameterized with a randomly chosen 32-bit hash parameter. The binary, the monitoring graph, and the hash parameter are then signed with the network operator's private key. In addition, the binary, monitoring graph, and hash parameter are encrypted with a random symmetric key (K_{sym}). The symmetric key is encrypted with the router's public key to ensure only the router can decrypt this information. The encrypted binary, monitoring graph, and hash parameter, the signature, the encrypted key, and the certificate are then transmitted over the network to the network processor. The network processor decrypts the data with the provided symmetric key (after applying the router's private key) and verifies the authenticity and integrity of the data with the public key of the network operator.

- At runtime: When the network processor performs packet processing operations, the processor reports its 32-bit operation to the parameterizable hash function. The 4-bit hashed operation is then reported to the hardware monitor that compares it to the monitoring graph.

5.2.2 Parameterizable Hashing

As we discussed in the previous chapters, the hash function used in our system takes the instruction word executed by the processor core and maps it to a smaller

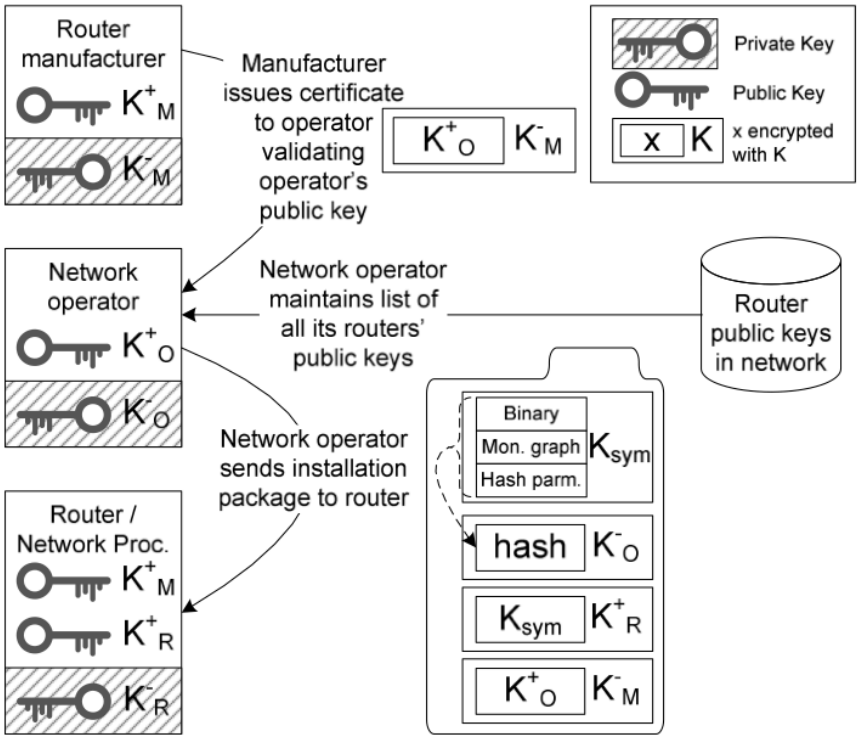

Figure 5.2: Security operations in SDMMon.

(e.g., 4-bit) hash value. Monitor graphs with small hash values can be represented very compactly and processed with a single memory access.

The drawback of using a hashed representation of the executed instruction word is that hash needs to be computed every processor clock cycle and that hashing is a many-to-one mapping. The latter can be exploited by a potential attacker by creating an attack that hijacks the processor with an instruction sequence that is identical to the hash values expected by the monitor. To provide security from such an attack, our SDMMon uses different hash function parameters on each router and these parameters are communicated securely between network operator and router. Thus, an attacker cannot know what hash function to expect and the only viable attack would be a brute force enumeration of different hash sequences. The use of different hash parameters on different routers also ensures that a potentially successful brute force attack on one system cannot be exploited on other systems.

Figure 5.3: Parameterizable hash function based on Merkle tree.

The design challenge for the hash function used in our hardware monitoring system is to provide good hash characteristics while allowing a high-performance implementation. Cryptographic hash functions would be a great choice since they can be parameterized with a cryptographic key and achieve strong collision resistance, but they require too much processing complexity. Instead, we aim for a hash function with weak collision resistance that can be implemented efficiently in hardware.

We base our hash function design on a Merkle tree [67], as shown in Figure 5.3. The tree structure can be efficiently implemented in hardware and requires only a logarithmic number of dependent operations based on the instruction and parameter length. Each tree node computes an 8-to-4 bit compression function as part of the overall hash calculation. Leaf nodes take 4 bits from the hash function parameter and 4 bits from the processor instruction.

Without knowledge of the parameter used in the calculation, an attacker cannot guess the mapping of a potential 32-bit attack instruction to its 4-bit hash. As

discussed above, brute force probing is possible, but difficult to implement for longer attacks.

5.2.3 Security Properties

Based on our system-level design of secure transmission of binaries and monitoring graphs, as well as the use of a hash function with different parameters for different router systems, we can now illustrate how our security requirements can be achieved.

1. Security requirement SR1 (only valid binaries installed) is achieved because the packages of binaries and monitoring graphs are signed by the network operator. Because the attacker cannot obtain the network operator's private key (AC3 and AC4) and because only valid network operators receive certificates from the manufacturer, the packages have to be authentic.

2. Security requirement SR2 (hardware monitor diversity) is achieved because each monitor instance uses a different, randomly chosen, hash parameter. The attacker can try to generate an attack that matches (AC2), but would need to do that using a very inefficient brute-force approach.

3. Security requirement SR3 (confidentiality) is achieved because the components of the package are encrypted with a symmetric key that is only available to the network operator and the router (because of AC3 and AC4). The attacker can observe the package (AC1), but cannot interpret the content.

4. Security requirement SR4 (binaries and monitor graphs specific to single system) is achieved because the symmetric encryption key is encrypted with the router's public key. Therefore, only the router for which a package is intended can correctly decrypt the information in it (again because of AC3 and AC4).

Because we can ensure that our security requirements are maintained, we claim to achieve system-level security for network processors with hardware monitors.

Table 5.1: System-level security prototype resource use on DE4 FPGA

	Available on FPGA	Nios II contr. proc.	NP core with hw monitor
LUTs	182,400	13,477	41,735
FFs	182,400	16,899	40,590
Memory bits	14,625,792	497,976	2,883,088

5.3 Prototype System Implementation

To illustrate how a secure system for network processors with hardware monitors can be implemented in practice, we present results from a prototype implementation.

5.3.1 System Setup

We have implemented a prototype SDMMon (Figure 5.4) in an Altera Stratix IV FPGA on an Altera DE4 board. A reconfigurable network processor was implemented with a PLASMA processor and a reconfigurable hardware monitor was connected to this NP. For the security and dynamic control purpose, a Nios II soft processor was implemented as the control processor. The board contains four 1Gbps Ethernet ports, through which the control processor can reach the network operator's server. The system can download, decrypt, and verify the binaries and monitor graphs, load the binaries and graphs to the shared memory, and reconfigure the network processor and hardware monitor. We have installed a μClinux operating system in the Nios II core to provide essential service support such as TCP/IP, FTP, SSH, and OpenSSL.

The relative size of the control processor compared to a network processor core with hardware monitor is shown in Table 5.1. The control processor, which performs all the security operations, is only about one third the size of a network processor core with hardware monitor. In addition to the resources shown in the table, the control processor also requires 2895kB of memory for the operating system image.

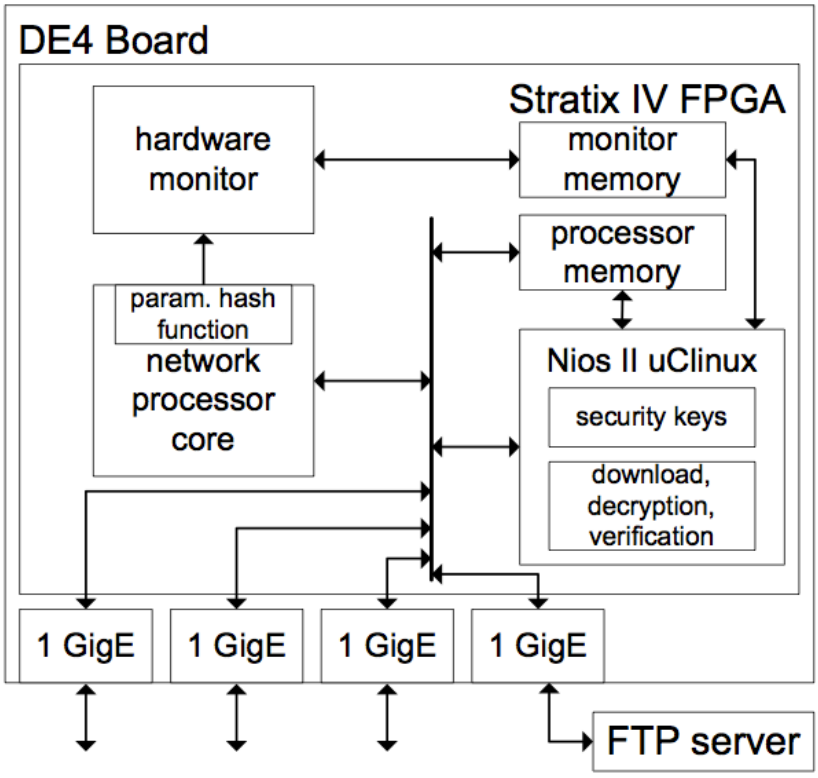

Figure 5.4: Prototype network processor system with system-level security management configuration

5.3.2 Binary and Monitoring Graph Installation

We have implemented the decryption and verification steps on the embedded control processor of the network processor system. All cryptographic operations use the commercial-grade OpenSSL toolkit (version 1.01e).

We generate public/private key pairs for all three entities – network processor manufacturer, network operator, and network processor device – using the RSA algorithm with key length of 2048 bits. We sign the operator's public key with the manufacturer's private key to create a certificate to establish a chain of trust.

The package of binary (for IPv4+CM), monitoring graph, and hash parameter is signed with the operator's private key. Since the package size exceeds the RSA algorithm's capacity to encrypt it, we encrypted the package using a randomly chosen

Table 5.2: Processing of security functions on Nios II

Step	Time (s)
Download data from FTP server	1.90
Check manufacturer certificate of network operator's public key K_O^+	3.33
Decrypt AES key K_{sym} using router's private key K_R^-	8.74
Decrypt package with AES key K_{sym}	7.73
Verify packet signature with network operator's public key K_O^+	3.92
Total	25.62
Total (no networking or certificate check)	20.39

AES symmetric key. That symmetric key is then encrypted with the router's public key to allow secure exchange.

At the network processor device, we verify the certificate to make sure the operator is indeed the operator and not an attacker. We also decrypt the AES key using the router's private key, making it able to access the package upon decryption of the AES algorithm. We can then verify the signature using the operator's public key. Finally, we unpack the package and obtain the binary, monitoring graph, and hash parameter. These files are then installed in the memory network processor device.

The running times of the various steps taken on the control processor are shown in Table 5.2. The total time is about 25 seconds, which is acceptable since new processing applications for network processors are created at slower time scales. (Note that switching between applications already installed on the network processor can be done quickly to accommodate dynamic changes in workload by keeping multiple binaries and graphs in memory.) When skipping the certificate check (which has to be done only once) and ignoring network delay (which can be decreased based on server location), then the verification time is around 20 seconds.

Table 5.3: Implementation cost of hash functions

	Bitcount hash	Merkle tree hash
LUTs	103	95
FFs	61	61
Memory bits	0	32

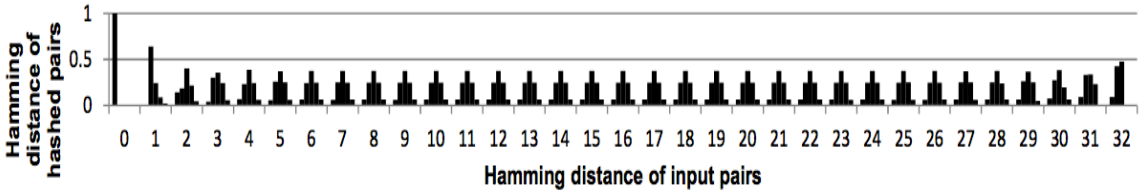

Figure 5.5: Distribution of hash values using our Merkle-tree-based hashing.

5.3.3 Hash Function Evaluation

We have also implemented the parameterizable hash function on the prototype system. As compression function f, we use the 4-bit arithmetic sum of both 4-bit inputs. The resource consumption of the implementation of the hash function is shown in Table 5.3. For comparison, the resources for a typical conventional hash function (counting set bits in the word to hash) are also shown. It can be seen that the resource requirements are comparable. Our Merkle tree hash requires less logic, but requires memory to store the parameter, whereas the bitcount hash does not require memory. Both hash functions are fast enough to compute the hash within the available cycle time on our system.

To show that the parameterizable hash function based on Merkle trees is effective for our goal, i.e., providing diversity across systems (security requirement SR2), we evaluate the distribution of generated hash values. We do this by randomly creating 32-bit value pairs (i.e., two different processor instructions) and comparing their 4-bit hashed value pair. As a comparison metric, we use the Hamming distance between each pair, which is an indication on the number of bits that is different between

the values in the pair. Figure 5.5 shows the distribution of Hamming distances for hashed values for each possible Hamming distance of the original pair. The Hamming distance between 4-bit hashed value pairs can take on values $0 \ldots 4$ and thus each distribution consists of five bars. Ideally, the hash function creates an approximation of a Gaussian distribution for all pairs with Hamming distance other than zero. For a Hamming distance of zero (i.e., elements are identical), the Hamming distance of the hashed values also needs to be zero to guarantee consistent hashing of the same value. In the figure, we can see that for all but the most extreme Hamming distance of input pairs, the ideal distribution is achieved. Since the hash function is symmetric for hashed value and parameter value, the same distribution properties hold true for different parameter values. Thus, an attacker cannot determine the hash values of a code sequence without knowing the hashing parameter. Also, diversity of monitoring across routers is achieved since different parameter values lead to different hash values in the monitoring graph.

5.4 Summary and Conclusions

We have presented a system-level security design that ensures that hardware monitors on network processors can be programmed dynamically, while ensuring that attackers cannot tamper with the monitoring system. Our design is based on the use of cryptographic principles to securely transfer all the necessary data to the network processor. We have also designed a parameterizable hash function that allows monitoring graphs to be customized to each individual router system in order to protect from a cascading attack. We have demonstrated the operation of our system in hardware on an FPGA-based platform. We believe that this work provides an important contribution toward moving from device-level security to system-level security in embedded hardware monitoring.

CHAPTER 6

COMPLETE SYSTEM IMPLEMENTATION OF MULTICORE NETWORK PROCESSOR SECURITY PLATFORM

In earlier chapters, we have described techniques to protect network processors from data plane attacks by introducing security system components including a hardware security monitor, scalable hardware monitor grid, parameterized hashing, and monitor graph download management. In this chapter, we introduce the design of a comprehensive multicore NP security platform that includes these components. The system implementation on an Altera DE5 development board [2] will be described in detail.

6.1 Design Challenges

To provide a complete integration of our ideas, additional challenges must be met. These challenges include addressing increased Internet speeds, FPGA device size limits, and user interfaces. In this chapter we integrate our work on an Altera DE5 board which includes a 10 Gbps Ethernet interface. Additional extensions include:

1. A Linux operating system [33] is used to support network and cryptographic services such as TCP/IP [72], FTP [73] and OpenSSL [6]. To promote security, the communication between the OS and the hardware is carefully designed. Physical addresses are remapped to virtual addresses by a memory management unit (MMU) and device drivers are used by the OS to recognize customized hardware modules.

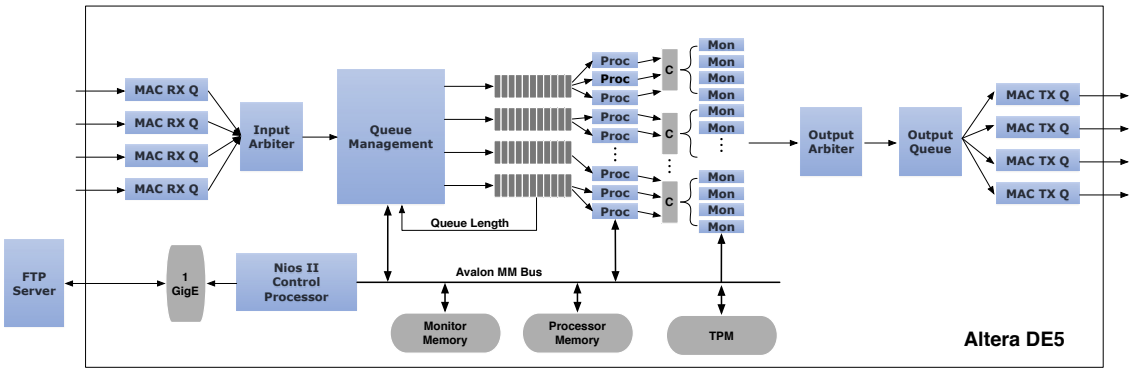

Figure 6.1: System architecture of the multicore network processor security platform

2. As described in Section 5.2.1, security keys must be installed in the NP system at both manufacturing and installation time. To prevent attackers from stealing these keys using software based attacks, a Trusted Platform Module (TPM) must be used to securely store these keys.

3. In order to verify the functionality of the multicore NP security platform system, a network packet generator, which can send and capture network packets through 10G SFP+ ports, is designed for the DE5 board so that multiple DE5 boards can be connected for experiments.

6.2 Multicore NP Security Platform System Architecture

The system architecture of our complete platform is illustrated in Figure 6.1. The platform consists of four 10G SFP+ high speed Ethernet ports, a DDR3 SDRAM main memory, a Nios II control processor, a customized TPM module, and the Scalable Hardware Monitoring Grid (SHMG) logic. As the fundamental hardware component in the system, the SHMG includes input and output arbiters that interface to the 10G Ethernet ports, queues and queue management logic that buffer input network packets, and a multi-core multi-monitor cluster. The system hardware gets network packets from the 10G network port, identifies the packet types and distributes the

packets to NP cores, monitors the packet processing in the NP cores, and forwards the packets to corresponding output network ports after processing them.

To achieve the system level security features described in Chapter 5, a μClinux OS is installed in the Nios II control processor to support the network and cryptographic services. Linux based device drivers and C software are designed to implement the following key functions:

1. Communicate with a remote server through an Ethernet interface and securely download encrypted processor binaries and monitor graphs from the server when necessary.

2. Communicate with the TPM module to get security keys, decrypt the processor binaries and monitor graphs and put them into processor memory and monitor memory.

3. Monitor the queue lengths in the hardware. When a queue length passes a pre-configured threshold, partially reconfigure the processors and monitors and the interconnection between the processors and monitors in SHMG without affecting the work of other processors and monitors.

We introduce the details of the hardware and software implementations and present system evaluation experimental results in the remainder of this chapter.

6.3 System Hardware Implementation

Compared with the Altera DE4 board used for experiments in previous chapters, the DE5 has significantly more FPGA and memory resources and supports 10G rather than 1G Ethernet interfaces.

Table 6.1: Key features of DE4 vs DE5.

	DE4	DE5
FPGA	EP4SGX230C2	5SGXEA7N2F45C2
Logic elements	228,000	622,000
On-chip memory	17,133Kb	50Mb
DRAM	DDR2 SO-DIMM	DDR3 SO-DIMM
Ethernet interface	Gigabit Ethernet (GigE)	10G SFP+

6.3.1 Altera DE5 FPGA Board

The Altera DE5 development and education board features a Stratix V GX FPGA and integrates transceivers that transfer data at a maximum of 12.5 Gbps, allowing low-latency, straight connections to four external 10G SFP+ modules. Table 6.1 compares the DE5 key features with those in the DE4.

6.3.2 10 Gbps Ethernet

The DE5 board has four independent 10G SFP+ connectors that use one transceiver channel each from the Stratix V GX FPGA device. These modules take in serial data from the Stratix V GX FPGA device and transform them to optical signals [2]. Altera provides 10 Gbps Ethernet (10GbE) Media Access Controller (MAC) [12] and 10GBASE-R PHY [13] IP components to handle the SFI interface with a optical SFP+ running at 10.3125 Gbps. A complete 10 Gbps Ethernet solution (shown in Figure 6.2) must include both the 10G Ethernet MAC and the 10GBASE-R PHY.

The 10Gbps Ethernet MAC IP core is a configurable component that implements the IEEE 802.3-2008 specification. It uses the Avalon streaming (Avalon-ST) interface on the client side to talk with the user defined hardware module and the single data rate (SDR) XGMII on the network side to talk with the 10GBASE-R PHY.

The Altera 10GBASE-R PHY IP core implements the functionality described in IEEE Standard 802.3 Clause 45 [4]. It delivers serialized data to an optical module

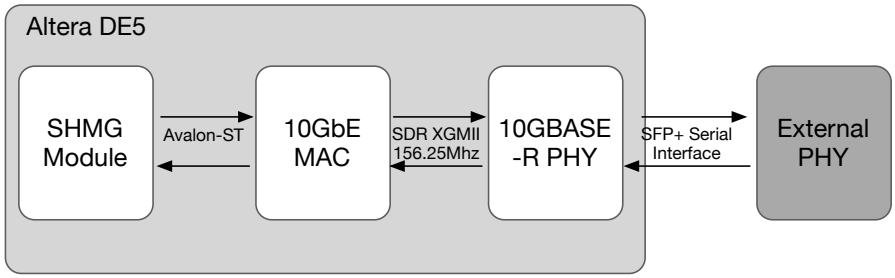

Figure 6.2: 10 Gbps Ethernet Solution.

that drives optical fiber at a line rate of 10.3125Gbps. In our system, since there are four connectors, we use a 4-channel implementation of 10GBASE-R, each channel of the 10GBASE-R PHY IP core operates independently.

6.3.3 Scalable Hardware Monitoring Grid Implementation

We have implemented a four NP core, six monitor SHMG cluster (Figure 4.6) in our system. The NP cores are 32-bit MIPS soft processors created using a synthesizable PLASMA processor [76]. Two different network applications (IPv4 and IPv4 with congestion management) are cross compiled to the processor cores and monitor graphs are generated as described in Section 3.3.1. Both the processor cores and the monitors are modified to support runtime application switches.

When a network packet arrives from the 10GbE MAC, it goes through an input arbiter that identifies its packet types and distributes it to an appropriate queue. Whenever a NP core is available, it fetches a packet from its corresponding queue. At runtime, if one queue passes the predefined threshold, the queue management unit generates a "reconfiguration required" message to software in the μClinux OS. The software reads all queue lengths from hardware to determine which processor and monitor will be reallocated.

6.3.4 Trusted Platform Module

A Trusted Platform Module (TPM) is a dedicated microprocessor or circuit that stores keys, passwords and digital certificates. The Trusted Computing Group (TCG) [7] was chartered to create the TPM specification. Typically, a TPM is affixed to the motherboard of a PC to ensure the security of the information stored there from external software attack and physical theft.

In our design, we have implemented a hardware TPM as a system module instead of using a separate TPM chip. It communicates with the Nios II control processor and the SHMG through an Avalon MM bus. In this TPM module, a unique 2048-bit RSA public key/private key pair is stored as the router's keys: K_R^+ and K_R^-. They can be used to verify the authentication of the hardware device, as described in Section 5.2.1. The 2048-bit RSA public key of the manufacturer K_M^+ and the manufacturer provided certificate that contains the network operator's public key K_O^+ signed with the manufacturer's private key K_M^- are also stored in the TPM. Using these keys and certificate, the network processor can establish a chain of trust from the network manufacturer to the network operator. In our implementation, we use Secure Hash Algorithm (SHA-1) [69] to generate the certificate. SHA-1 is the most widely used SHA function, and it is employed in several widely used applications and protocols.

We wrote a device driver for this TPM in μClinux OS so that user space software is able to read the keys from the hardware. During program update, the keys are read by the control processor and used to verify and decrypt the binary and monitoring graph package following the steps in Section 5.3.2. Any software modification to the keys in the TPM is not allowed. Device driver development is described in Appendix A.

6.4 DE5-Based Packet Generator

In order to test and evaluate our NP security platform, we need to send network packets through the 10G SFP+ interfaces and measure output performance. Thus, a circuit that performs packet generation and network packet capture through the 10G SFP+ connectors is an essential part of the system design. For this purpose, we have designed and implemented a network packet generator for the Altera DE5.

The DE5 packet generator implements two major functions: (1) it generates and sends network traffic to the 10Gbps Ethernet ports with customized packet size, packet type, number of packets, and throughput rate and (2) it captures network packets from 10Gbps Ethernet ports and measures the number of packets of different types and the throughput of the network.

The system architecture of the packet generator is shown in Figure 6.3. The DE5 FPGA board connects to a host PC through the PCI Express bus. Users of the packet generator use the DE5 packet generator configuration utility software to read pcap format network packets and select a customized packet size, number and throughput rate. The software configures the hardware packet composer with these user selected parameters through the PCIe interface and the hardware packet composer generates the network traffic. Incoming network packets from any of the four 10GbE ports is captured by the packet capturer and traffic statistics are logged and sent to the PC software through the PCIe interface. Software records these statistics.

We used Microsoft Visual Studio 2010 to develop the PC software and 32-bit Jungo Win Driver [5] to establish communication between the host PC and the Altera DE5 board. Win Driver exposes two APIs (OnRCSlaveWrite and OnRCSlaveRead) for applications to read and write using the PCI Express bus.

This DE5 network packet generator together with a DE5 reference router has been published online as an open source project at `http://www.ecs.umass.edu/ece/tessier/rcg/netfpga-de5/index.html`. An online tutorial is available which

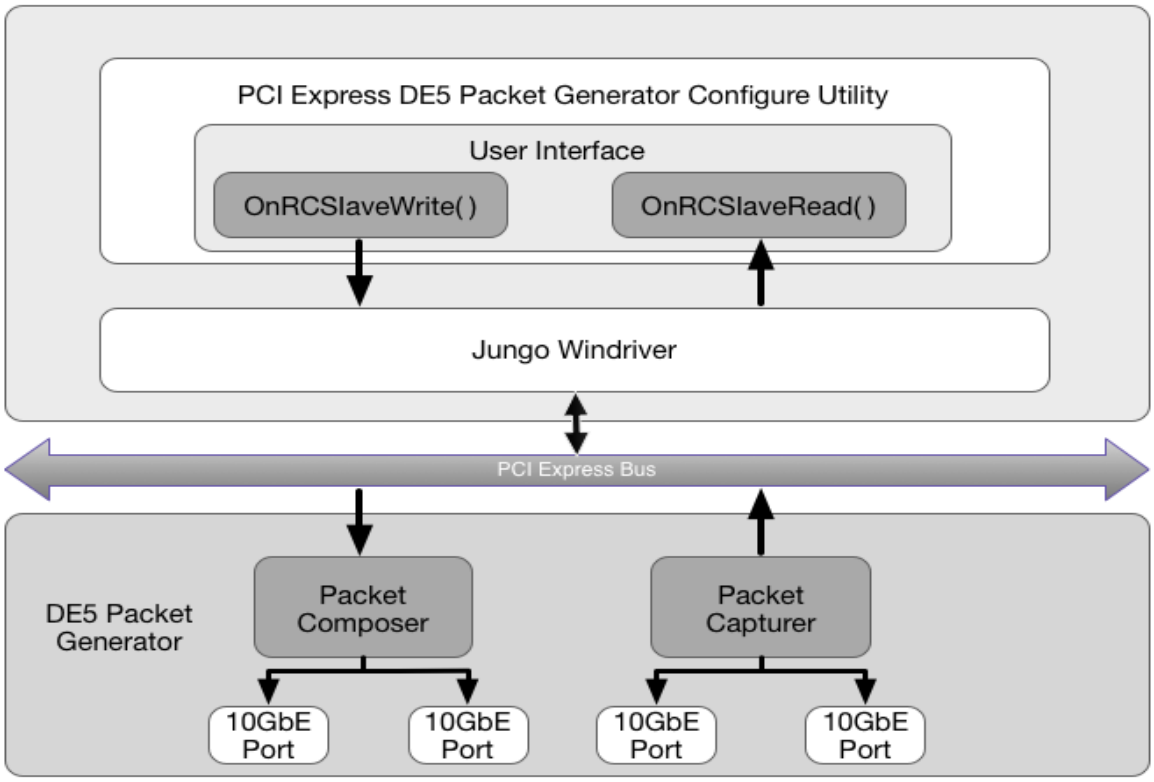

Figure 6.3: System Architecture of the DE5 Packet Generator

describes the steps needed to download the DE5 NetFPGA packet generator design, compile the design using Altera Quartus II software, and use the packet generator.

6.5 Experiments and Results

6.5.1 Evaluation Metrics

Three evaluation metrics are used to verify the DE5 Multicore NP Security Platform system:

1. Correctness of functionality: The system is able to process network packets, detect and recover from the attack packets, correctly read security keys from the TPM and decrypt binary and monitor graph package, and dynamically reconfigure the SHMG at run time.

2. Resource utilization: The lookup table (LUT), flip flop (FF), and memory resources (bits) utilization of the system are assessed.

3. Throughput: The system throughput performance is measured and compared with the throughput results in Chapter 4.

Details of these experiments are provided below.

6.5.2 Functionality Validation

The following experiments are performed to validate the functionality of our system:

1. Correct network packet processing: The system is connected to a packet generator as shown in Figure 6.4. IPv4 and IPv4+CM packets are sent through one 10GbpE port to the system and captured on another port. 500 IPv4 packets and 500 IPv4+CM packets were sent in 10 experiments (100 packets are sent each time). All packets were correctly received.

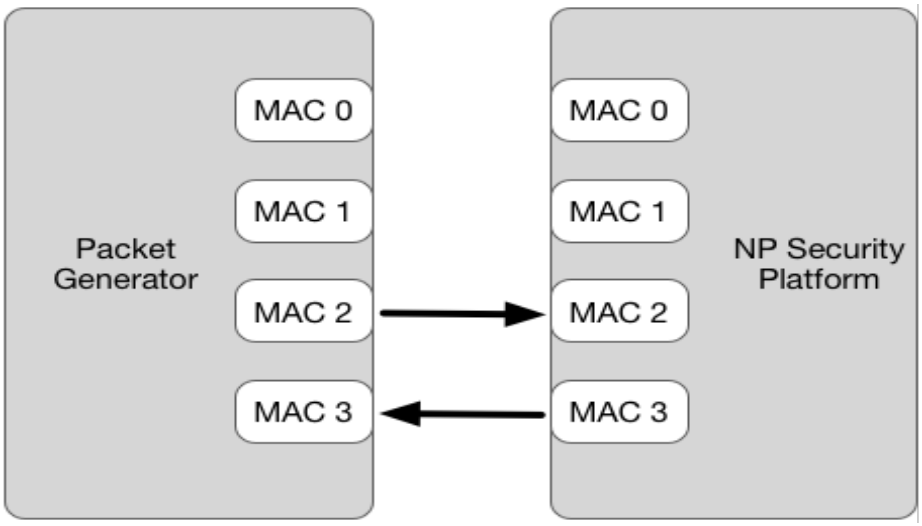

Figure 6.4: Multicore NP security platform test scenario

2. Attack detection: Normal packets combined with attack packets are sent. We verify that the normal packets are processed appropriately and the attack packets are detected and discarded. In our experiment, we used one port to send normal packets (MAC 2), one port to send attack packets (MAC 0) and one port to receive packets (MAC 3). 100 normal packets were sent with 1 attack packet and the results showed that the normal packets were forwarded and the attack packets were discarded.

3. Dynamic resource reallocation: Since it is not possible to measure the intermediate status of the processor cores nor the runtime throughput changes before the packet generator receives all the packets, it is not possible to directly keep track of the runtime resource reallocation in the hardware. However, this experiment is designed to show that the control processor has the ability to reconfigure the SHMG at runtime.

   The processor cores are configured to perform different actions: half of the cores (2 total) forward packets to MAC 3 and half of them (2 total) forward packets to MAC 1. For each 100 packets sent to the system through MAC 2, 50 packets are output via MAC 1 and 50 packets are output via MAC 3. Subsequently, one processor core is reconfigured to forward packets to MAC 1 instead of MAC 3. For each 100 packets sent to the system, 75 packets come from MAC 1 and 25 packets come from MAC 3. This experiment shows that the system is capable of reconfiguring the SHMG at runtime.

4. TPM driver test: We store a 32-bit number (0x12344321 in the experiment) in the TPM hardware and use sample user code to read and print it on the μClinux console. The result is shown in Figure 6.5. In the key management operations, two 2048-bit RSA keys K_M^+, K_R^- and the certificate of K_O^+ are provided to μClinux for package decryption.

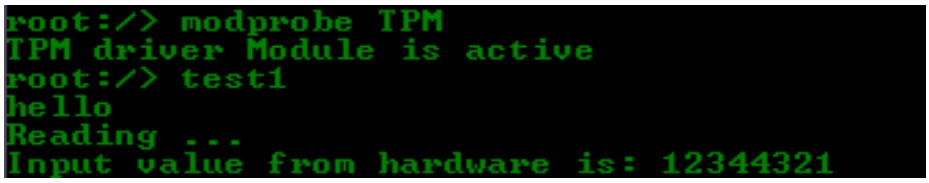

Figure 6.5: TPM test result

5. Secure binary and monitoring graph installation: We have performed experiments similar to those in Section 5.3.2 to evaluate the processing time of the security functions in our system. The experimental results are listed in Table 6.2. Note that there are two modifications in this experiment compared to the previous one:

   (a) Although FTP service is supported in our system, since there is no remote FTP host that supports the 10Gbps SFP+ interface, we did not perform downloading measurements in the experiment. The data package was preloaded into the system.

   (b) Since the security keys are stored in the hardware TPM, not in the OS file system, one extra step is needed to read the keys from the hardware before the decryption operations.

   Most of the security functions run faster in this system than in the DE4 system (Table 6.2) because the Nios II in this system runs at higher clock frequency (156.25 MHz vs 125 MHz). The time cost of decrypting the AES key using the router's private key takes a longer time because we use a different OpenSSL utility to do the decryption in this experiment.

6.5.3 Resource Utilization

The resource utilization of our system is shown in Table 6.3. The lookup table (LUT), flip flop (FF), and memory resources (bits) are listed for different components:

Table 6.2: Processing time of security functions on DE5

Step	Time (s)
Load keys from hardware TPM	1.15
Check manufacturer certificate of network operator's public key K_O^+	1.77
Decrypt AES key K_{sym} using router's private key K_R^-	15.98
Decrypt package with AES key K_{sym}	3.13
Verify packet signature with network operator's public key K_O^+	2.08
Total	24.11
Total (no certificate check)	22.34

Table 6.3: Multicore NP Security Platform Resource Utilization on DE5 FPGA

Resources	Secure monitors	Network proc.	Nios II contr. proc.	DE5 interface	Available in FPGA
LUTs	552 1.1%	16,051 32.9%	2,168 4.5%	30,036 61.5%	234,720 -
FFs	114 0.2%	8,452 16.4%	2,532 4.9%	40,483 78.5%	234,72 -
Mem. bits	786,432 7.8%	1,179,648 11.7%	70,336 0.7%	8,060,856 79.8%	52,428,800 -

the four network processor cores, six hardware monitors, Nios II control processor and other circuitry. In addition, the μClinux kernel image needs 5078 Kb in the SDRAM.

6.5.4 System Throughput

The throughput of our system is measured with two network packet sizes: 64-byte and 256-byte (Figure 6.6). The maximum system throughput is similar to the one shown in Figure 4.18. This shows that the throughout of the system is limited by the processing capability of the processor cores, not the network port speed. An increase

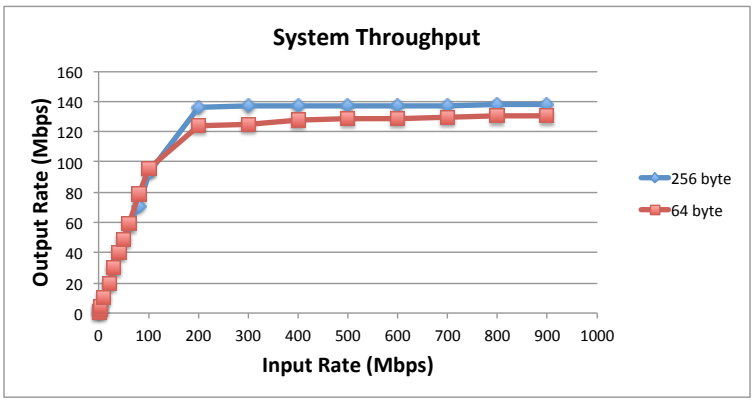

Figure 6.6: Throughput results when processing 64-byte and 256-byte IPv4+CM packets.

in the total number of cores increases the system throughput. The average latency for 256-byte packets of regular traffic was measured at 92 us.

6.6 Summary and Conclusions

This chapter presents an integration of various components of the dissertation research. A multicore network processor security platform has been implemented on a DE5 development board. The system hardware and software implementations are described in detail and system results demonstrate its effectiveness.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this dissertation, we focus on security issues in network processor systems with hardware monitors. A detailed, multicore monitoring approach has been developed. This system is scalable and allows for fast run-time changes to system function. We are able to download new monitoring graphs to the system from external sources using a secure protocol.

7.1 Summary of Contributions

For our first contribution, we have demonstrated a Scalable Hardware Monitoring Grid (SHMG) design that solves three critical problems in practical network processor systems: multiple cores, multiple processing binaries and dynamically changing workload. SHMG provides a scalable processor-to-monitor interconnection architecture to balance the area cost and the performance overhead. A resource reallocation algorithm is proposed for reconfiguring the processor cores and hardware monitors to adapt to the dynamically changing network traffic at runtime. Our evaluation validates the functionality of the algorithm and demonstrates that the system overhead for our monitoring system is less than 6% compared to the processor system.

Second, we developed a Secure Dynamic Multicore Hardware Monitoring System (SDMMon) model for secure installation of processor binaries and monitor graphs in a system level implementation. A Merkle tree based parameterizable high-performance hash function is presented to prevent attackers from applying a successful attack on one network device to other devices. We have demonstrated the operation of our

system in hardware on an FPGA-based platform. This work provides an important contribution toward moving from device-level security to system level security in embedded hardware monitoring.

To conclude this dissertation research, a Multicore Network Processor Security Platform system has been implemented on an Altera DE5 board. It integrates all the security features of SHMG and SDMMon. A hardware TPM is included in this system to assist the key operations during processor binary and monitor graph updates. Our evaluation shows the correctness of the system and provides for evaluation of resource cost and performance. The resource overhead of our sytem is around 5% compared to the multicore processor system and the maximum throughput of a four-core, six-monitor system is 140 Mbps. Network attacks are detected within one to two clock cycles.

7.2 Future Work

Hardware-assisted monitoring is a lightweight and efficient technique for detecting a wide variety of attacks in embedded systems. It provides many new dimensions that are worth exploring in future work.

Unknown indirect branches: Since the monitor graphs of the current hardware monitor design are generated at compile time, a monitor can only handle indirect branches to statically known targets (e.g., return addresses). It cannot handle indirect branches to statically unknown targets that are resolved at run time. Although network applications usually do not have such programming constructs (all 11 NpBench benchmarks have known compile-time known targets), unknown indirect branches exist in general purpose programs. A hardware monitor design that is able to handle these indirect branches is desirable to make hardware monitors more applicable to generalized systems.

Dynamically resizable monitor: Considering that the code sizes of applications may vary greatly, a static monitor memory size which is determined by the largest application size causes resource waste for other, smaller applications. Dynamic reconfiguration of the monitor memory structure could eliminate this memory overhead. The monitoring memory could be made dynamically resizable depending on the number of instructions or monitoring graph states in the target application.

Multiple FSMs in one monitor: A monitoring approach could be developed which uses multiple small finite state machines rather than a single FSM in a hardware monitor. This approach could potentially reduce the size of monitor graphs by removing redundant information. In such a design, monitoring information is distributed into several small FSMs and the monitor uses these small FSMs together to establish correct functional operation.

OS security: The hardware monitor architecture we presented in this dissertation protects a user application from its vulnerabilities. However, it does not account for the operating system and its vulnerability. Modern operating systems are deployed in a large variety of embedded systems. It is desirable to apply hardware monitoring to these OS-based systems. Some work has been done in this direction, for example, MTHM [83] introduces a fine-grained monitoring architecture that supports multiple contexts under the control of an operating system. However, this design does not address the problem of OS code vulnerability.

Portable monitor: Although dedicated hardware monitors are an efficient security solution for network processors, it may not be feasible to deploy monitors in all network routers since it requires a modification to current router architecture that may be costly. An alternative solution that involves minimum router hardware changes could be a portable monitor which is a separate device that can be plugged into existing processor interfaces. In this way, the network processors and the hardware monitors are independent, and the processors only need an interface to

communicate with the monitors. In general purpose embedded processors, a portable monitor could be a USB device or a special unit that connects to the motherboard through PCIe. Note that this type of approach is still vulnerable to simple physical attacks.

APPENDIX A

SYSTEM SOFTWARE IMPLEMENTATION

In this section, we introduce the installation and configuration of μClinux [33] and a memory management unit (MMU) on the DE5 board and the μClinux device driver development flow. A sample kernel driver code for the TPM module is provided in Appendix B.

A.1 Installation and Configuration of μClinux on DE5

The original μClinux was a fork of the Linux 2.0 kernel for microcontrollers in embedded systems without a memory management unit (MMU). Today's embedded μClinux operating system includes Linux kernel releases for 2.0 2.4 and 2.6, as well as a collection of user applications, libraries and tool chains. It has been widely used in embedded systems and forms the basis of many products, such as network routers, security cameras, DVD and MP3 players, VoIP phone and gateways, scanners, and card readers. In our system, we use μClinux that targets 2.6 series Linux kernels.

A.1.1 Setup the Development Environment

To be able to build μClinux, a Linux based PC is required. In our case, we used a VMware based virtual machine that installs Fedora 18 64-bit version as the compilation environment. In the virtual machine, we downloaded the latest μClinux distribution and precompiled NIOS II to the Linux cross complier toolchain[1] [2].

[1] `http://sopc.et.ntust.edu.tw/`

[2] `ftp://ftp.altera.com/outgoing/nios2-linux/20120802`

The virtual machine for this project was obtained from `http://sdrv.ms/11cMAUy` [46]. There are 43 rar files in the distribution and the total size is 22 GB. After downloading and unzipping these files, the virtual machine is opened with VMware, and the development environment including the μClinux distribution and the precompiled toolchain can then be used.

A.1.2 DeviceTree

DeviceTree is a way of describing hardware in an embedded system [44]. This information is needed during μClinux kernel compilation so that the kernel knows the hardware architecture of the target system. Altera Qsys doesn't directly provide a DeviceTree file (.dts file), it only provides a .sopcinfo file which has to be converted to a .dts file. We use a Java program (sopc2dts) to generate device-tree sources (.dts files) from a Qsys .sopcinfo file. JDK 1.5 or above is required to compile this tool.

To recreate our virtual machine environment, go to the directory `~/uClinux/tools/sopc2dts`, the sopc2dts tool has already been compiled and generated. Type the following command will open the GUI of sopc2dts tool shown in Figure A.1.

```
$ java -jar sopc2dts.jar --gui
```

Choose the .sopcinfo file describing your system in the "Input" tab. Look over (and adjust) the various bits in the "Boardinfo" tab, select $cpu : nios2\_qsys\_0$ as the master. Press apply after you've made a change. Check the "Output" tab to see the effect on your generated device tree. After finishing all the changes, choose a filename in the "Boardinfo" tab and press save. The tool would generate a .dts file based on the input .sopcinfo file.

Since we have customized hardware components in our system (e.g. TPM module), the description of these modules has to be encoded in the device tree. In order for the sopc2dts software to recognize these modules, we need to modify the *hw.tcl* files of these modules. A *hw.tcl* is a file that describes the properties and behaviors of

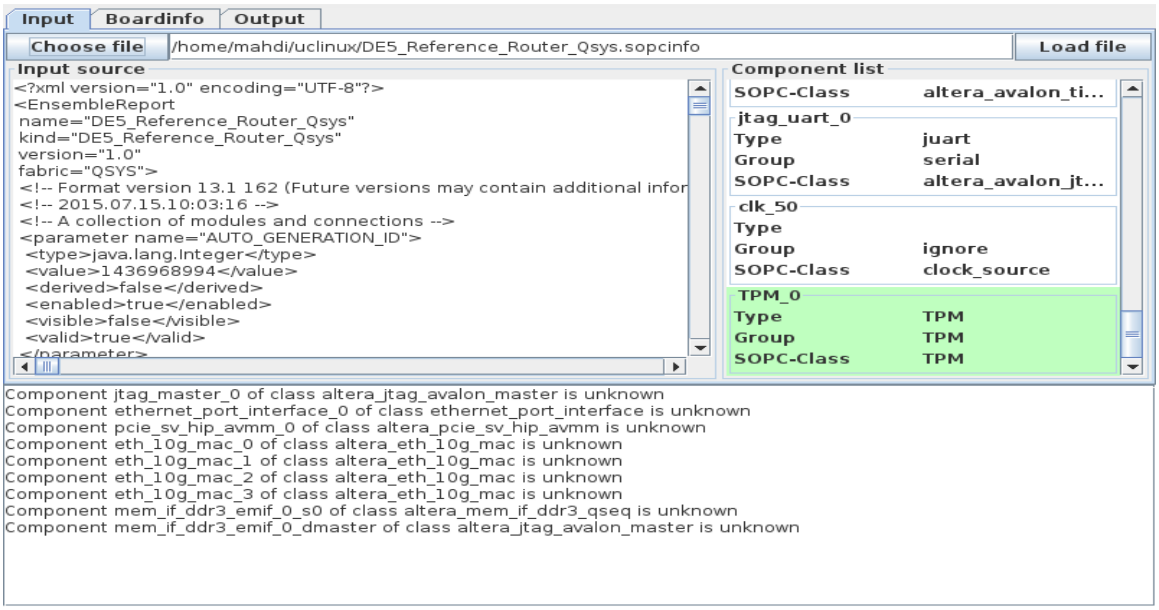

Figure A.1: GUI of sopc2dts tool

an Altera Qsys component, it allows for arbitrary name/value pairs to be assigned to modules and to interfaces with the *hw.tcl* statements. At design generation time, these assignments are put in the resulting sopcinfo file. Sopc2dts looks for these interface and module assignments to the namespace, embeddedsw.dts., when creating device trees. In our TPM example, we need to specify the vendor, name and group of the TPM module using the following lines in $TPM\_hw.tcl$.

```
set_module_assignment embeddeddsw.dts.vendor "ALTR"
set_module_assignment embeddeddsw.dts.name "TPM"
set_module_assignment embeddeddsw.dts.group "TPM"
```

A.1.3 Configure and Build μClinux

After generating a DeviceTree for the hardware system, we can use it to build a μClinux kernel. Go to the μClinux distribution directory and set the $PATH$ environment variable for the toolchain.

```
$ export PATH=$PATH:/path/to/nios2/toolchain/bin
```

Some configuration in the μClinux kernel is needed.

- Set the vendor and products.
- Set MMU support.
- Set the SDRAM memory base address.
- Set the path to the device tree source file.
- Configure the kernel features.
- Enable the Altera JTAG UART console support
- Enable the network services and cryptographic support.

These changes can be made in a GUI configuration interface with command "make menuconfig". Figure A.2 shows the main menu of this GUI. Go into "Vendor/Product Selection", select Altera as vendor and Nios II as product. Then go to "Kernel/Library/Defaults Selection", make sure kernel is Linux-2.6.x and Libc is none. Select "Customize Kernel Settings" and "Customize Application/Library Settings". Save the changes and exit.

If "Customize Kernel Settings" is selected, another GUI interface (Figure A.3) will be open after exiting the main menu. In this kernel configuration menu, we need to set all the kernel related configurations.

- In "Kernel features", set the timer frequency to 1000Hz, set the memory model to flat memory and set the low address space to 4096.
- In "Platform options", set the SDRAM base address to 0x0, note that this base address must match with the SDRAM base address in the Qsys project. Enable "Compile and link device tree into kernel image" and specify the path to the device tree source file.

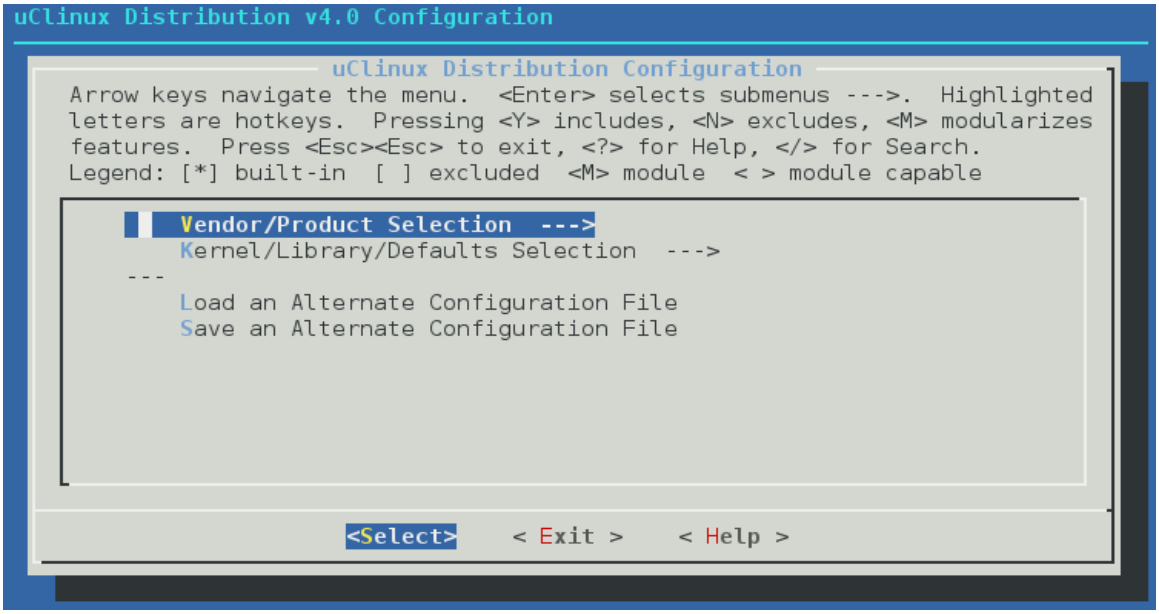

Figure A.2: μClinux GUI configuration main menu

- In "Device drivers", go to "Character devices" and enable Altera JTAG UART support in "Serial drivers", set the baudrate to 115200.

If "Customize Application/Library Settings" is selected, the Application/Library configuration GUI (Figure A.4) will be open after the kernel setting. In this GUI, we select applications such as ftp, openssl and telnet in "Network Applications". Note that we need to download and compile the source files for the selected applications, the absence of source code or errors in the source code would cause the failure of kernel compilation.

After configuring the μClinux kernel, type *make* in the command line to start building the kernel image. If the kernel image is built successfully without any compilation errors, you will see a message saying:

```
Kernel: arch/nios2/boot/zImage is ready
```

The kernel image is stored in `uClinux-dist\linux-2.6.x/arch/nios2/boot/`.

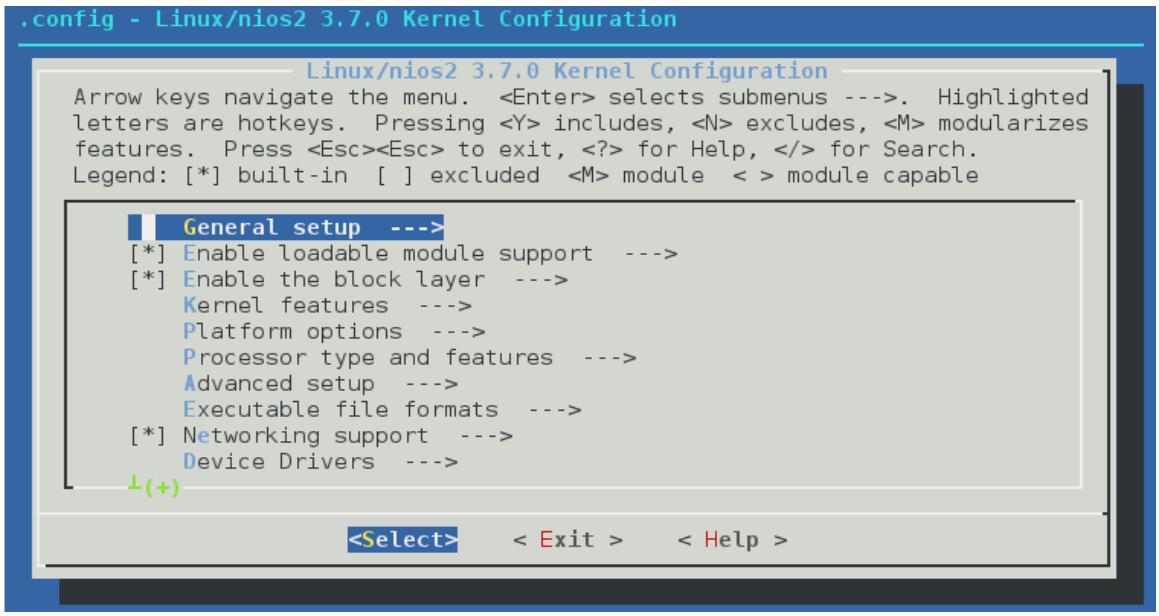

Figure A.3: μClinux GUI kernel configuration menu

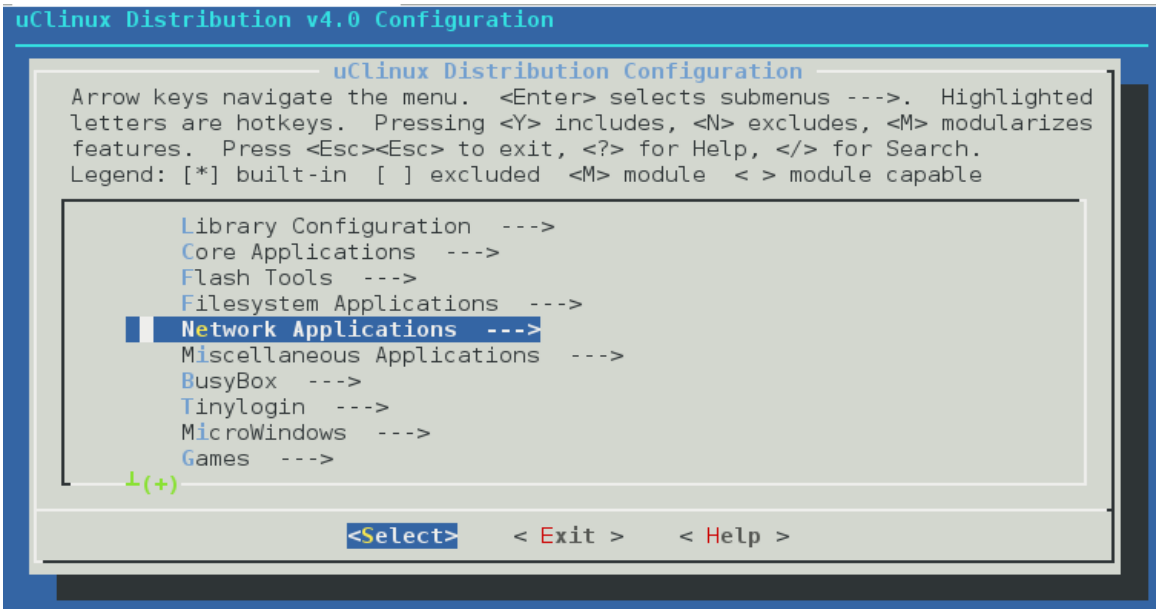

Figure A.4: μClinux GUI application configuration menu

A.1.4 Loading μClinux onto DE5

Turn on the DE5 board and connect it with the host PC through the USB-Blaster cable. In Quartus II, download the .sof file to the FPGA.

We use JTAG UART as a serial console to boot μClinux from memory. First, we copy the μClinux kernel image file *zImage* from the virtual machine to the Windows host machine and save it under the `path_to_DE5_project/sw` directory. Then we open the Altera Nios2 command shell from the *Tools* tab in Qsys, and change the command shell path to `path_to_DE5_project/sw`. Next, we download the kernel image into the memory on the board with this command:

```
$ nios2-download -c 1 -g zImage
```

After downloading the kernel, we use nios2-terminal to see the output from the μClinux boot up. Note that cygwin must be installed in your machine to support the Nios2 command shell.

```
$ nios2-terminal
```

If the kernel boots up successfully, you should see messages similar to Figure A.5.

A.2 μClinux Device Driver Development

Now that we have a functioning μClinux system up and running, the next important step is to write device drivers for the kernel to support the custom hardware that we added in Qsys. We use our TPM driver as a sample design to illustrate the driver development process.

A.2.1 Introduction of Linux Device Driver

In compute systems, a device driver is a computer program that provides a software interface for the operating systems and other computer programs to interact with a hardware device without needing to know details of the hardware. This in-

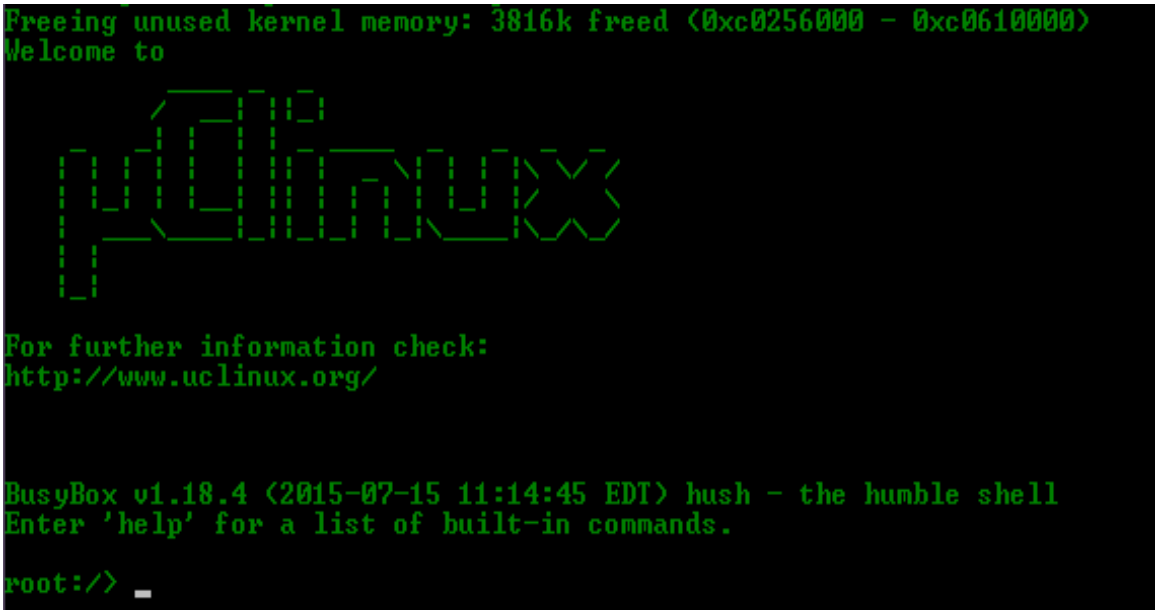

Figure A.5: μClinux boot up messages

terface is designed so that drivers can be built separately from the rest of the kernel and plugged in at runtime when needed [28].

The Linux way of looking at device drivers distinguishes between three fundamental device types: char device, block device, and network device.

- Character devices: A character (char) device is one that can be accessed as a stream of bytes. It is the most common type of device driver. There are several fundamental system calls that need to be implemented in a char type device driver including open, close, read and write. User applications treat the char driver like reading and writing to a file. The text console and the serial ports are common examples of char devices.

- Block devices: A block device is a device that can host a filesystem, it handles I/O operations that transfer one or more whole blocks (512 byte or a larger power of two). A typical example of block device is a disk.

- Network devices: A network driver handles the packet sends and receives from a network interface with other hosts. It is only used for hardware that is involved with data transfer using TCP/IP protocol.

The device driver we developed is a character driver.

A.2.2 User Space and Kernel Space

System memory in Linux can be divided into two distinct regions: kernel space and user space. Kernel space is where the core of the operating system stores and executes, while user space is a set of locations where normal user processes run. The kernel has full access to all memory and machine hardware, whereas processes running under user space have access only to a limited part of memory. User space processes can only access a small part of the kernel via system calls. If a user space process performs a system call, a software interrupt is sent to the kernel. The kernel dispatches an appropriate interrupt handler to take care of this system call.

A Linux device driver runs in kernel space. It implements system calls such as open, close, read and write. A user space program uses these system calls to communicate with the driver and the driver is in charge of accessing the hardware. Figure A.6 shows the relationship between user space, kernel space and hardware.

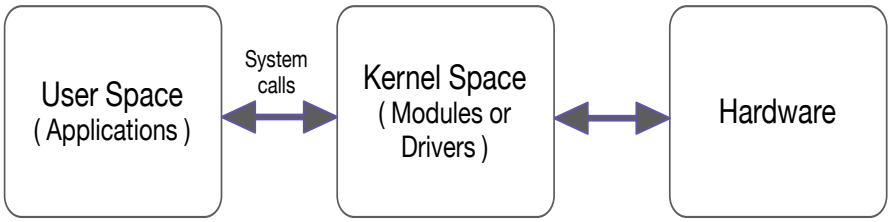

Figure A.6: User space, Kernel space and Hardware

A.2.3 Implementation of the driver

This section describes how to write a complete char device driver. The first step of driver writing is defining the capabilities the driver will offer to user programs. In our TPM sample design, our driver will open a hardware device and read keys from it, writing to the hardware is not needed. As a result, the driver needs to support open, close and read file operations.

Table A.1 lists all the device events that our driver needs to support and their associated user space and kernel space functions. User functions are implemented as statements that the user application can use in the C or C++ programs or Linux shell commands. The users interact with the kernel through these functions. Kernel functions are implemented in the device driver. They interact directly with the hardware and send feedback to the user functions.

Table A.1: Device events and their associated interfacing functions

Event	User Function (User Program)	Kernel Function (Driver)
Load Module	insmod	module_init()
Open Device	fopen	file_operation: open
Read Device	fread	file_operation: read
Close Device	fclose	file_operation: release
Remove Module	rmmod	module_exit()

The module initiate function *module_init*() and the module exit function *module_exit*() are the two most essential functions in a driver. Any char device driver has to at least implement these two functions. They register the driver in the kernel and release the driver from the kernel system.

In *module_init*(), the classic way to register a char device driver is to use API function *register_chrdev*.

```
int register_chrdev(unsigned int major, const char *name,
struct file_operations *fops);
```

Here, *major* is the unique major number that is either assigned by the driver developer or assigned by the kernel automatically for each driver, *name* is the name of the driver, and *fops* is the default file operations defined in the driver. The file operation is a struct code that contain pointers to all functions in the driver. The driver uses these pointers to forward user application requests to the correct handlers.

In *module_exit*(), the driver uses *unregister_chrdev* to remove the driver from the kernel system and release all memory that has been allocated to the driver.

The Linux command *insmod* and *rmmod* are used to load and remove loadable modules to the kernel system in the Linux console.

As shown in Table A.1, the driver has open and release functions to handle user space requests from *fopen* and *fclose*. The content of these function is based on the hardware specification. If the driver needs initialization before the read or write to the hardware, it can be initialized in the open function. If the driver needs clean up after the read or write, the release function can be used.

The read function in the driver must read a whole segment of data from the hardware and copy it to the user space. These capabilities are offered by the following kernel functions:

```
unsigned inb (unsigned port);
unsigned inw(unsigned port);
unsigned inl(unsigned port);
unsigned long copy_to_user (void __user * to, const void * from,
unsigned long n);
```

The first three functions are inline functions defined in the Linux kernel headers (`<asm/io.h>`) to access hardware I/O ports. They are used to read from hardware ports in 1 byte, 2 bytes or 4 bytes increments. After a value is read from the hardware, the *copy_to_user* function is used to copy this value to the user program. In this

function, *to* is the destination address in user space, $from$ is the source address in kernel space, and n is the number of bytes to copy.

After we have implemented all five kernel functions in the driver, the driver code is ready. The driver can be added into the μClinux kernel image.

- Copy the driver source code to `linux-2.6.x\source\drivers\misc`.
- Add an option to Kconfig in `linux-2.6.x\source\drivers\misc`. This action adds an entry in the GUI configuration tool for users to select this driver.

```
config TPM
    tristate "Example TPM driver module"
    help
    Enable example TPM module.
```

- Add object file to Makefile in `linux-2.6.x\source\drivers\misc`.

```
obj-$(CONFIG_TPM) += TPM.o
```

- Add this device to the file system by adding the following line to `vendors\Altera\nios2\device_table.txt`.

```
/dev/TPM    c    666    0    0    240    0    -    -    -
```

- Type command "make menuconfig". In the GUI interface, go to "Device Drivers → Misc devices" and select the TPM driver module as shown in Figure A.7.
- Type command "make". This action rebuilds the μClinux kernel, and the new kernel image includes our TPM driver module.

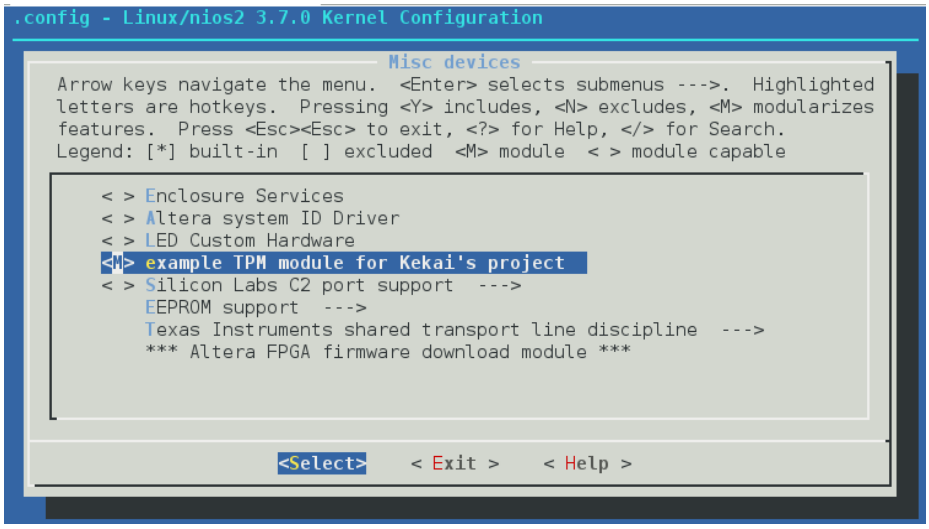

Figure A.7: Select the TPM driver in the configuration GUI

A.2.4 User space program

When the TPM driver is compiled in the kernel, a user space program can access it as a file using normal file operations such as *fopen*, *fclose*, *fread* and *fwrite*. For example, the following code will open the TPM driver for reading and return a pointer to a FILE object that is used to identify the stream on all further operations. If the open fails, a null pointer is returned.

```
FILE *opntest;
Opntest = fopen ( "/dev/TPM", "r" );
```

A simple user space program is provided in Appendix C as an example to show how a user space program access the driver. It reads a 32-bit value from the hardware TPM module and prints it out. It is necessary to cross-compile user-space programs for the NIOS II on a host machine. We use a *nios2-linux-gnu-gcc* cross-complier to compile a C source code.

```
nios2-linux-gnu-gcc source.c -o output_bin
```

Copy the *output_bin* to `/romfs/bin/` and recompile the kernel to create a new zImage that includes the user program.

APPENDIX B

TPM SAMPLE COMPONENT KERNEL DRIVER C CODE

```
#include <linux/init.h>
#include <linux/kernel.h> /*printk()*/
#include <linux/module.h>
#include <linux/mm.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <asm/asm-offsets.h>

/* major numbers for identifying the device files */
#define DATA_MAJOR 240

/* register addresses */
#define DATA_REGISTER 0x08002000

MODULE_AUTHOR("Kekai Hu");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Module which creates device handler for /dev/TPM");
MODULE_SUPPORTED_DEVICE("none");
```

```
/* device files open? */
int dat_is_open = 0;

/* called when data device file is opened */
static int dat_open(struct inode *inode, struct file *file)
{
        if (dat_is_open++)
                return -EBUSY;
        try_module_get(THIS_MODULE);
        return 0;
}

static int dat_release(struct inode *inode, struct file *file)
{
        --dat_is_open;
        module_put(THIS_MODULE);
        return 0;
}

/* called when a process reads from data file */
static ssize_t dat_read(struct file *filp, char *buffer,
size_t length, loff_t * offset)
{
        /* using inl to read from hardware,
        note that we have to use ioremap to map
        the physical address to virtual address first
```

```
        before using inl to read*/

        unsigned int data_buf;
        printk("Reading ....\n");
        int __iomem* membase;
        membase = ioremap(DATA_REGISTER, 4);
        data_buf = inl(membase + offset);
        copy_to_user(buffer, &data_buf,4);
        return 1;
}

/* definitions, which functions are called for /dev/TPM */
static struct file_operations fops_dat =
{
.read= dat_read,
.open= dat_open,
.release= dat_release
};

/* initialize the module */
static int __init mod_init(void)
{
        if(register_chrdev(DATA_MAJOR, "TPM", &fops_dat))
        {
        printk("register_chrdev of hello failed!\n");
        return -EIO;
        }
```

```
        printk("TPM driver Module is active\n");
        return 0;
}

/* exit the module */
static void __exit mod_exit(void)
{
        unregister_chrdev(DATA_MAJOR,"TPM");
        printk("Module is released\n");
}
/* what are the module init/exit functions */
module_init( mod_init );
module_exit( mod_exit );
```

APPENDIX C

SAMPLE USER SPACE C CODE

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        FILE *opntest;
        opntest = fopen("/dev/TPM","r");
        unsigned int temp = 0;
        fread(&temp,4,1,opntest);
        fclose(opntest);
        printf("Input value from hardware is: %x\n", temp);
        return 0;
}
```

BIBLIOGRAPHY

[1] Altera DE4. http://www.terasic.com.

[2] Altera DE5. http://www.altera.com/education/univ/materials/boards/de5/unv-de5-board.html.

[3] Common types of network attacks. http://technet.microsoft.com/en-us/library/cc959354.aspx.

[4] IEEE standard for Ethernet. http://www.trincoll.edu/Academics/MajorsAndMinors/Engineering/Documents/IEEE%20Standard%20for%20Ethernet.pdf.

[5] Jungo WinDriver PCIe Driver. http://www.jungo.com/st/products/windriver.

[6] OpenSSL manual. https://www.openssl.org/docs/.

[7] Trusted computing group. http://www.trustedcomputinggroup.org/.

[8] Software-Defined Networking: The New Norm for Networks. White paper, Open Networking Foundation, Apr. 2012.

[9] Abadi, Martín, Budiu, Mihai, Erlingsson, Úlfar, and Ligatti, Jay. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC) 13*, 1 (2009), 4.

[10] Akhshabi, Saamer, and Dovrolis, Constantine. The evolution of layered protocol stacks leads to an hourglass-shaped architecture. *SIGCOMM Comput. Commun. Rev. 41*, 4 (Aug. 2011), 206–217.

[11] Altera. Quartus II handbook version 9.1. Tech. rep., Altera, 2009.

[12] Altera. *10-Gbps Ethernet MAC MegaCore Function User Guide*, 2014. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/10gbps_mac.pdf.

[13] Altera. *Altera Transceiver PHY IP Core User Guide*, 2015. https://www.altera.com/en_US/pdfs/literature/ug/xcvr_user_guide.pdf.

[14] Anderson, Thomas, Peterson, Larry, Shenker, Scott, and Turner, Jonathan. Overcoming the internet impasse through virtualization. *Computer 38*, 4 (2005), 34–41.

[15] Andrews, Mike, and Whittaker, James A. Computer security. *IEEE Security & Privacy 2*, 5 (2004), 68–71.

[16] Arora, Divya, Ravi, Srivaths, Raghunathan, Anand, and Jha, Niraj K. Secure embedded processing through hardware-assisted run-time monitoring. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)* (Munich, Germany, Mar. 2005), pp. 178–183.

[17] Baker, Fred. Requirements for IP version 4 routers. RFC 1812, Network Working Group, June 1995.

[18] Balakrishnan, Hari, Rahul, Hariharan S, and Seshan, Srinivasan. An integrated congestion management architecture for internet hosts. In *ACM SIGCOMM Computer Communication Review* (1999), vol. 29, ACM, pp. 175–187.

[19] The Bro Project. *The Bro Network Security Monitor*, 2004. `http://www.bro-ids.org`.

[20] Cavium Networks. *OCTEON Plus CN58XX 4 to 16-Core MIPS64-Based SoCs*. Mountain View, CA, 2008.

[21] Chasaki, Danai, and Wolf, Tilman. Design of a secure packet processor. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)* (San Diego, CA, Oct. 2010).

[22] Chasaki, Danai, and Wolf, Tilman. Attacks and defenses in the data plane of networks. *IEEE Transactions on Dependable and Secure Computing 9*, 6 (Nov. 2012), 798–810.

[23] Cho, Young H, Navab, Shiva, and Mangione-Smith, William H. Specialized hardware for deep network packet filtering. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Springer, 2002, pp. 452–461.

[24] Chowdhury, NM Mosharaf Kabir, and Boutaba, Raouf. Network virtualization: state of the art and research challenges. *Communications Magazine, IEEE 47*, 7 (2009), 20–26.

[25] Cisco Systems, Inc. *The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor*. San Jose, CA, Feb. 2008.

[26] Comer, Douglas E. *Network Systems Design Using Network Processors: Intel 2XXX Version*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[27] Coppersmith, Don. The data encryption standard (DES) and its strength against attacks. *IBM journal of research and development 38*, 3 (1994), 243–250.

[28] Corbet, Jonathan, Rubini, Alessandro, and Kroah-Hartman, Greg. *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[29] Cui, Ang, Song, Yingbo, Prabhu, Pratap V., and Stolfo, Salvatore J. Brave new world: Pervasive insecurity of embedded network devices. In *Proc. of 12th International Symposium on Recent Advances in Intrusion Detection (RAID)* (Saint-Malo, France, Sept. 2009), vol. 5758 of *Lecture Notes in Computer Science*, pp. 378–380.

[30] Daemen, Joan, and Rijmen, Vincent. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[31] Day, John D, and Zimmermann, Hubert. The OSI reference model. *Proceedings of the IEEE 71*, 12 (1983), 1334–1340.

[32] Dierks, Tim. The transport layer security (tls) protocol version 1.2. RFC 5246, Network Working Group, 2008.

[33] Dionne, D. Jeff, and Durrant, Michael. μClinux description, Dec. 2007. `http://www.uclinux.org/description/`.

[34] Doraswamy, Naganand, and Harkins, Dan. *IPSec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional, 2003.

[35] Duan, H, Lockwood, JW, and Kang, SM. FPGA prototype queuing module for high performance ATM switching. In *ASIC Conference and Exhibit, 1994. Proceedings., Seventh Annual IEEE International* (1994), IEEE, pp. 429–432.

[36] Eatherton, Will. The push of network processing to the top of the pyramid. In *Keynote Presentation at ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)* (Princeton, NJ, Oct. 2005).

[37] Erlingsson, Úlfar. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004. AAI3114521.

[38] EZchip Technologies Ltd. *NP-5 – 240-Gigabit Network Processor for Carrier Ethernet Applications*. Yokneam, Israel, May 2012. `http://www.ezchip.com/`.

[39] Fielding, Roy, Gettys, Jim, Mogul, Jeffrey, Frystyk, Henrik, Masinter, Larry, Leach, Paul, and Berners-Lee, Tim. Hypertext transfer protocol–http/1.1. RFC 2616, Network Working Group, 1999.

[40] Garuba, Moses, Liu, Chunmei, and Fraites, Duane. Intrusion techniques: Comparative study of network intrusion detection systems. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on* (2008), IEEE, pp. 592–598.

[41] Geer, David. Malicious bots threaten network security. *Computer 38*, 1 (Jan. 2005), 18–20.

[42] Giladi, Ran. *Network Processors: Architecture, Programming, and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[43] Gogniat, Guy, Wolf, Tilman, Burleson, Wayne, Diguet, J-P, Bossuet, Lilian, and Vaslin, Romain. Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 16*, 2 (2008), 144–155.

[44] Granath, Johan. μClinux on NIOS2 with custom hardware and kernel module, Feb. 2013.

[45] Guttman, Barbara, and Roback, Edward A. *An introduction to computer security: the NIST handbook*. DIANE Publishing, 1995.

[46] Hafezi, Mahdi. How to boot μClinux on Altera NIOS II processor with MMU. `https://www.youtube.com/watch?v=RxlyezOf0Rs`.

[47] Hopcroft, John E., and Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[48] Hu, Kekai, Chandrikakutty, H., Goodman, Z., Tessier, R., and Wolf, T. Dynamic hardware monitors for network processor protection. *Computers, IEEE Transactions on PP*, 99 (May 2015).

[49] Hu, Kekai, Chandrikakutty, H., Tessier, R., and Wolf, T. Scalable hardware monitors to protect network processors from data plane attacks. In *Communications and Network Security (CNS), 2013 IEEE Conference on* (Washington, DC, Oct 2013), pp. 314–322.

[50] Hu, Kekai, Wolf, T., Teixeira, T., and Tessier, R. System-level security for network processors with hardware monitors. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (San Francisco, CA, June 2014), pp. 1–6.

[51] Hutchinson, Norman C., and Peterson, Larry L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan. 1991), 64–76.

[52] Jajodia, Sushil, Noel, Steven, and OBerry, Brian. Topological analysis of network attack vulnerability. In *Managing Cyber Threats*. Springer, 2005, pp. 247–266.

[53] Karlof, Chris, and Wagner, David. Secure routing in wireless sensor networks: Attacks and countermeasures. *Ad hoc networks 1*, 2 (2003), 293–315.

[54] Kohler, Eddie, Morris, Robert, Chen, Benjie, Jannotti, John, and Kaashoek, M. Frans. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (Aug. 2000), 263–297.

[55] Kolkman, Olaf M. DNSSEC operational practices. RFC 4641, Network Working Group, 2006.

[56] Kumarapillai Chandrikakutty, Harikrishnan. Protecting network processors with high performance logic based monitors. Master's thesis, University of Massachusetts Amherst, 2013.

[57] Kumarapillai Chandrikakutty, Harikrishnan, Unnikrishnan, Deepak, Tessier, Russell, and Wolf, Tilman. High-performance hardware monitors to protect network processors from data plane attacks. In *Proc. of 50th Design Automation Conference (DAC)* (Austin, TX, June 2013).

[58] Kuon, Ian, Tessier, Russell, and Rose, Jonathan. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom. 2*, 2 (Feb. 2008), 135–253.

[59] Langen, Dominik, Niemann, Jörg-Christian, Porrmann, Mario, Kalte, Heiko, and Rückert, Ulrich. Implementation of a RISC processor core for SoC designs–FPGA prototype vs. ASIC implementation. In *IEEE Workshop Heterogeneous Reconfigurable Systems on Chip (SoC)* (Hamburg, Germany, 2002).

[60] Lee, Byeong Kil, and John, Lizy Kurian. NpBench: A benchmark suite for control plane and data plane applications for network processors. In *Proc. of IEEE International Conference on Computer Design (ICCD)* (San Jose, CA, Oct. 2003), pp. 226–233.

[61] Lesk, Michael E. The new front line: Estonia under cyberassault. *IEEE Security & Privacy 5*, 4 (July 2007), 76–79.

[62] Lin, Kuan-Jiuh. Smtp-1: The first functionalized metalloporphyrin molecular sieves with large channels. *Angewandte Chemie International Edition 38*, 18 (1999), 2730–2732.

[63] Lockwood, John W, McKeown, Nick, Watson, Greg, Gibb, Glen, Hartke, Paul, Naous, Jad, Raghuraman, Ramanan, and Luo, Jianying. NetFPGA–an open platform for Gigabit-rate network switching and routing. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on* (2007), IEEE, pp. 160–161.

[64] Manimaran, G, and Al-Duwairi, Basheer. Internet infrastructure security. In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on* (2005), IEEE, pp. 6–7.

[65] Mao, Shufu, and Wolf, Tilman. Hardware support for secure processing in embedded systems. *IEEE Transactions on Computers 59*, 6 (June 2010), 847–854.

[66] McKeown, Nick, Anderson, Tom, Balakrishnan, Hari, Parulkar, Guru, Peterson, Larry, Rexford, Jennifer, Shenker, Scott, and Turner, Jonathan. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[67] Merkle, Ralph Charles. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, Stanford, CA, June 1979.

[68] Moore, D., Shannon, C., and Brown, J. Code-Red: a case study on the spread and victims of an Internet worm. In *Internet Measurement Workshop (IMW) 2002* (Marseille, France, Nov 2002), ACM SIGCOMM/USENIX Internet Measurement Workshop, pp. 273–284.

[69] NIST. *Secure hash standard*, 1995. Federal Information Processing Standards, FIPS-180-1.

[70] Oppliger, Rolf. Internet security: firewalls and beyond. *Communications of the ACM 40*, 5 (1997), 92–102.

[71] Pereira, Karl Savio Pimenta. *Characterization of FPGA-based high performance computers*. PhD thesis, Virginia Polytechnic Institute and State University, 2011.

[72] Postel, Jon. Internet protocol. RFC 791, Network Working Group, Sept. 1981.

[73] Postel, Jon, and Reynolds, Joyce. Rfc 959: File transfer protocol (FTP). *InterNet Network Working Group* (1985).

[74] Ragel, Roshan G, and Parameswaran, Sri. Impres: integrated monitoring for processor reliability and security. In *Proceedings of the 43rd annual Design Automation Conference* (2006), ACM, pp. 502–505.

[75] Ragel, Roshan G., Parameswaran, Sri, and Kia, Sayed Mohammad. Micro embedded monitoring for security in application specific instruction-set processors. In *Proc. of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (San Francisco, CA, Sept. 2005), pp. 304–314.

[76] Rhoads, Steve. *Plasma – most MIPS I(TM) Opcodes*, 2001. `http://www.opencores.org/project,plasma`.

[77] Roesch, Martin, et al. Snort: Lightweight intrusion detection for networks. In *LISA* (1999), vol. 99, pp. 229–238.

[78] Ruf, Lukas, Farkas, Karoly, Hug, Hanspeter, and Plattner, Bernhard. Network services on service extensible routers. In *Proc. of Seventh Annual International Working Conference on Active Networking (IWAN 2005)* (Sophia Antipolis, France, Nov. 2005).

[79] Sackett, George C. *Cisco router handbook*. McGraw-Hill, 2000.

[80] Sanzgiri, Kimaya, Dahill, Bridget, Levine, Brian Neil, Shields, Clay, and Belding-Royer, Elizabeth M. A secure routing protocol for ad hoc networks. In *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on* (2002), IEEE, pp. 78–87.

[81] Stallings, William. *Network Security Essentials: Applications and Standards (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[82] Tessier, Russell, Wolf, Tilman, Hu, Kekai, and Chandrikakutty, Harikrishnan. Reconfigurable network router security. In *Reconfigurable Logic: Architecture, Tools, and Applications*, Pierre-Emmanuel Gaillardon, Ed. CRC Press, 2015.

[83] Thomas, Tedy, Pouraghily, Arman, Hu, Kekai, Tessier, Russell, and Wolf, Tilman. Multi-task support for security-enabled embedded processors. In *Proc. of the IEEE Conference on Application-specific Systems, Architectures, and Processors (ASAP)* (Toronto, ON, Canada, July 2015).

[84] Wang, Haining, Zhang, Danlu, and Shin, Kang G. Syn-dog: Sniffing syn flooding sources. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (2002), IEEE, pp. 421–428.

[85] Wolf, Tilman, Chandrikakutty, Harikrishnan, Hu, Kekai, Unnikrishnan, Deepak, and Tessier, Russell. Securing network processors with high-performance hardware monitors. *IEEE Transactions on Dependable and Secure Computers PP*, 99 (Nov 2015).

[86] Wu, Qiang, and Wolf, Tilman. Runtime task allocation in multi-core packet processing systems. *IEEE Transactions on Parallel and Distributed Systems* (2012).

[87] Xilinx. ISE design suite. Tech. rep., Xilinx, 2012.

[88] Yau, David KY, and Chen, Xiangjing. Resource management in software-programmable router operating systems. *Selected Areas in Communications, IEEE Journal on 19*, 3 (2001), 488–500.

[89] Yue, Xin, Chen, Wei, and Wang, Yantao. The research of firewall technology in computer network security. In *Computational Intelligence and Industrial Applications, 2009. PACIIA 2009. Asia-Pacific Conference on* (2009), vol. 2, IEEE, pp. 421–424.

[90] Zambreno, Joseph, Choudhary, Alok, Simha, Rahul, Narahari, Bhagi, and Memon, Nasir. Safe-ops: An approach to embedded software security. *ACM Transactions on Embedded Computing Systems (TECS) 4*, 1 (2005), 189–210.