View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarWorks@UMass Amherst

University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Open Access Dissertations

5-2012

Security Issues in Networked Embedded Devices

Danai Chasaki Sr
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/open_access_dissertations

Part of the Electrical and Computer Engineering Commons

Recommended Citation
Chasaki, Danai Sr, "Security Issues in Networked Embedded Devices" (2012). *Open Access Dissertations*. 541.
https://doi.org/10.7275/jfhv-ba50 https://scholarworks.umass.edu/open_access_dissertations/541

This Open Access Dissertation is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Open Access Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

SECURITY ISSUES IN NETWORKED EMBEDDED DEVICES

A Dissertation Presented

by

DANAI CHASAKI

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

DOCTOR OF PHILOSOPHY

May 2012

Electrical and Computer Engineering

SECURITY ISSUES IN NETWORKED EMBEDDED DEVICES

A Dissertation Presented

by

DANAI CHASAKI

Approved as to style and content by:

_____

Tilman Wolf, Chair

_____

Weibo Gong, Member

_____

Russell Tessier, Member

_____

Jenna Marquard, Member

_____

Christopher V. Hollot, Department Head
Electrical and Computer Engineering

To *Ilektra*, *Abhi*, *Vasoula* and *my parents*.

ACKNOWLEDGMENTS

The material presented in this dissertation is based upon work supported by the National Science Foundation under Grant Nos. CNS-0952524 and CNS-1115999.

I would especially like to thank Professor Tilman Wolf, who is a very knowledgeable researcher and has always given me the most valuable advice.

ABSTRACT

SECURITY ISSUES IN NETWORKED EMBEDDED DEVICES

MAY 2012

DANAI CHASAKI

Diploma, NATIONAL TECHNICAL UNIVERSITY OF ATHENS, GREECE

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Tilman Wolf

Embedded devices are ubiquitous; they are present in various sectors of everyday life: smart homes, automobiles, health care, telephony, industrial automation, networking etc. Embedded systems are well known for their dependability, and that is one of the reasons that they are preferred over general purpose machines in various applications. Traditional embedded computing is changing nowadays mainly due to the increasing number of heterogeneous embedded devices that are, more often than not, interconnected. Security in the field of networked embedded systems is becoming particularly important, because:

- Connected embedded devices can be attacked remotely.
- They are resource constrained.

This means, that due to their limited computational capabilities, a full-blown operating system that runs virus scanners and advanced intrusion detection techniques cannot be

supported. The two facts lead us to the conclusion that a new set of vulnerabilities emerges in the networked embedded system area, which cannot be tackled using traditional security solutions.

This work is focused on embedded systems that are used in the network domain. A very exciting instance of an embedded system that requires high performance, has limited processing resources and communicates with other embedded devises is a network processor (NP). Powerful network processors are central components of modern routers, which help them achieve flexibility and perform tasks with advanced processing requirements. In my work, I identified a new class of vulnerabilities specific to routers. The same set of vulnerabilities can apply to any other type of networked embedded device that is not traditionally programmable, but is gradually shifting towards programmability.

Security in the networking field is a crucial concern. Many attacks in existing networks are based on security vulnerabilities in end-systems or in the end-to-end protocols that they use. Inside the network, most practical attacks have focused on the control plane where routing information and other control data are exchanged. With the emergence of router systems that use programmable embedded processors, the data plane of the network also becomes a potential target for attacks. This trend towards attacks on the forwarding component in router systems is likely to speed up in next-generation networks, where virtualization requires even higher levels of programmability in the data path.

This dissertation demonstrates a real attack scenario on a programmable router and discusses how similar attacks can be realized. Specifically, we present an attack example that can launch a devastating denial-of-service attack by sending just a single packet. We show that vulnerable packet processing code can be exploited on a Click modular router as well as on a custom packet processor on the NetFPGA platform. Several defenses to target this specific type of attacks are presented, which are broadly applicable to a large scale of embedded devices. Security vulnerabilities can be addressed efficiently using hardware based extensions. For example, defense techniques based on processor monitoring can help

in detecting and avoiding such attacks. We believe that this work is an important step at providing comprehensive security solutions that can protect the data path of current and future networks.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . . **iv**

ABSTRACT . . . . . **v**

LIST OF TABLES . . . . . **xi**

LIST OF FIGURES . . . . . **xii**

CHAPTER

1. INTRODUCTION **1**

1.1 Future of Embedded Systems 1

1.1.1 Networked Embedded Systems Security 1

1.2 Security of Embedded Devices in Networks 2

1.2.1 Network Security 2
1.2.2 Programmability in the Data Path of Networks 3
1.2.3 New Vulnerabilities in Modern Routers 4

1.3 Contribution 5
1.4 Organization of Dissertation 6

2. RELATED WORK **8**

3. EMBEDDED PROCESSORS IN NETWORKS **12**

3.1 Programmability 12
3.2 Design Challenges 14
3.3 Software Development 16
3.4 Prototype Implementation 18
3.5 Performance 19

4. VULNERABILITIES AND ATTACKS IN NETWORK INFRASTRUCTURE . . . 22

4.1 Attack Classification . . . 22
4.2 Commercial Routers with Programmable Features . . . 23
4.3 Security Model for Network Infrastructure . . . 26

4.3.1 Security Requirements . . . 26
4.3.2 Attacker Capabilities . . . 26

5. NETWORK INFRASTRUCTURE ATTACK . . . 28

5.1 Vulnerability . . . 28

5.1.1 Vulnerable Protocol Processing Code . . . 30
5.1.2 Attack Packet . . . 33

5.2 Attack Demonstration . . . 35

6. DEFENSE MECHANISMS . . . 38

6.1 Secure Packet Processing . . . 38

6.1.1 Security Model . . . 38
6.1.2 Attack Detection through Monitoring . . . 39
6.1.3 System Overview . . . 41
6.1.4 Monitoring Functionality . . . 42

6.2 Prototype Implementation . . . 44

6.2.1 Instruction-level Monitor . . . 45
6.2.2 I/O Monitor . . . 48

6.3 Evaluation . . . 50

6.3.1 Triggering Invalid Behavior . . . 50
6.3.2 Security Monitor Operation . . . 50
6.3.3 Performance Results . . . 53

6.4 Defenses in Multi-core Packet Processing Systems . . . 55

6.4.1 Programmable-logic-based Monitoring . . . 56
6.4.2 System Details . . . 57
6.4.3 Comparison to Fixed Hardware Monitor . . . 58
6.4.4 Experimental Results . . . 59

6.5 Demonstration of Real Attack Detection . . . 61

6.6 Inferring Packet Processing Behavior using I/O monitors 62

6.6.1 Verification in the Data Path 63

6.6.1.1 Protection Mechanisms 63
6.6.1.2 Modular Verification 64
6.6.1.3 Port-to-Port Packet Flow 65

6.6.2 Illustration of Processing Time Distributions 67
6.6.3 Quantifying Changes in Processing 67
6.6.4 I/O Monitor Complexity 72

7. CONCLUSIONS AND FUTURE WORK 75

7.1 Conclusions 75
7.2 Future Work 76

7.2.1 Networked Embedded Systems Security 76

BIBLIOGRAPHY 78

LIST OF TABLES

Table **Page**

3.1 IP forwarding throughput. 19

6.1 Resource consumption and performance of single core system. 54

6.2 Resource consumption and performance of four-core system. 55

6.3 Resource utilization of network processor benchmarks 61

6.4 KL-Divergence of different processing scenarios 71

LIST OF FIGURES

Figure **Page**

3.1 Processing context in simplified network processor design. 16

3.2 Simple C program for accessing and decrementing the time-to-live field in the IP header. 17

3.3 Network processing platform design. 18

3.4 Resource estimation per processing unit on Virtex 5. 20

4.1 Examples of network attacks and defenses. 23

5.1 Example of in-network attack. Vulnerable packet processing systems on routers can be used to launch large-scale denial-of-service attacks with a single packet. 29

5.2 Protocol Header Insertion. 31

5.3 Example Application Code. 32

5.4 Stack Smashing. 34

5.5 Experimental Setup. 36

5.6 Traffic Rates at Input Port and Output Port of Vulnerable Router. Benign video traffic is shown in green, attack traffic is shown in red. 37

6.1 Security Monitoring on the Packet Processor. 40

6.2 Monitoring System Overview. 43

6.3 Instruction Level Monitor Design. 47

6.4 I/O Monitor Design. 49

6.5 Attack scenario. 51

6.6 Simulation Results. 51

6.7 Security Monitor Operation. 53

6.8 System overview for multi-core secure packet processing system with reconfigurable monitors 57

6.9 Four core network processor performance including monitoring for different packet sizes 60

6.10 Traffic Rates at Input Port and Output Port of Router with Processing Monitor. 61

6.11 Modular Processing Verification 64

6.12 Validating Packet Flow on Node m. 66

6.13 Processing Time Distributions for Elements 1-7 68

6.14 Total Processing Time Distribution vs. Convoluted Distributions of 7 elements 68

6.15 Processing Time Distributions for Normal/Attack Cases 70

6.16 CDFs of Different Distributions and KL-distance 71

6.17 Pseudo-code for I/O Monitor 73

7.1 Networked Embedded Devices 77

CHAPTER 1

INTRODUCTION

1.1 Future of Embedded Systems

Embedded systems are entering a new era of innovation, their potential use has grown so much that most microprocessor designs currently target this market. Embedded devices are becoming ubiquitous; they range from gateways to IP phones, smart buildings, automobile parts, portable medical devices, health care records, sensors, transportation devices, industrial automation, etc. There is a new term describing the way that all these embedded systems communicate and connect to the Internet, called "the Internet of things". Trends show that in the near future 15 billion embedded devices will be connected [20], and traditional micro-controllers have given way to heterogeneous networked devices. Critical characteristics of embedded devices are dependability (reliability), real time response, high performance and optimization for dedicated functions. However, the vision of how to use networked embedded devices in the future cannot be realized without security in the "Internet of things".

1.1.1 Networked Embedded Systems Security

Security in the field of networked embedded systems is of critical importance because these interconnected devices can be remotely hacked [30]. However, the limited resources of embedded devices do not allow a full operating system to run virus scanners and the network cannot afford firewalls and advanced intrusion detection systems. Therefore, a new set of vulnerabilities emerges in the networked embedded system area that cannot be tackled using traditional security solutions. Since hackers will always exist, and will likely

become smarter, it is essential to identify such vulnerabilities early and implement custom solutions against potential harmful interactions among the connected devices.

1.2 Security of Embedded Devices in Networks

This dissertation identifies a new class of vulnerabilities specific to embedded systems in the network domain, specifically on router systems that use programmable embedded processors. Several defenses are also developed to target this specific type of attacks, which are broadly applicable to a large class of embedded devices.

1.2.1 Network Security

The Internet is a critical component of modern communication infrastructure. While security concerns were not a priority during the design of the Internet [11], society's reliance on the Internet today requires that the network be protected from malicious attackers. Thus, it is essential that the network architecture is extended to integrate the core principles of information security specified by the CIA triad of confidentiality, integrity, and availability [48].

End-to-end security protocols have been developed to provide confidentiality and integrity (e.g., Transport Layer Security (TLS) protocols). However, providing assurance of availability is a more challenging problem as it involves the entire network infrastructure. In the current Internet, there are very few techniques available to protect the network from denial-of-service attacks (even for nation states [24]). Since most large-scale denial-of-service attacks are of a distributed nature and generated by botnets [18], defense mechanisms aim to prevent end-system intrusion and thus to limit access to platforms from which attacks can be launched. Widely deployed intrusion prevention systems include firewalls [32] and deep packet inspection [47].

Most network security efforts have focused on vulnerable end-systems that are exploited by remote attacks through the network, on denial-of-service attacks that use the

network to disable end-systems, and on general information security. Intrusion prevention mechanisms (and thus defenses against denial-of-service attacks) target end-systems. Until recently, the network infrastructure itself has not been a major concern for network security since it presented no practical attack target. However, there are several important trends in networking that indicate that such defenses are insufficient against novel attacks that target vulnerabilities in the *data plane* of the network infrastructure itself. New vulnerabilities of modern routers are emerging because the technology used to implement these core network components has changed in recent years.

1.2.2 Programmability in the Data Path of Networks

In the past, most high-performance network routers used application-specific integrated circuits (ASICs) to implement packet forwarding functions. While ASICs are costly to develop, they represented the only technology that could achieve the performance that was necessary for multi-Gigabit per second traffic forwarding. Over the last few years, however, the performance of general-purpose multi-core processors (e.g., network processors or high-end server processors) has reached a level where high traffic forwarding rates can be achieved. Since the functionality of an ASIC cannot be changed once it has been designed, the use of general-purpose processor provides a router vendor with much more flexibility to adjust a router's functionality after production [14]. Therefore, there is an ongoing shift in the industry toward developing routers based on programmable packet processing engines rather than based on ASICs.

Modern networks use numerous devices that are based on programmable components in the data path:

- High-performance router implementations use embedded multi-core processing systems (i.e., network processors) for packet forwarding to accommodate advanced data plane functions [14]. On these systems, packet forwarding is implemented as software operations on general-purpose processor cores.

- Recent clean-slate network architectures consider various new protocols and data communication paradigms. To accommodate networks with different protocol stacks on a single network infrastructure, network virtualization has been proposed [2, 50]. Since network slices are deployed dynamically on a virtualized network substrate, the routers in the substrate need to be able to support custom packet processing functions. This programmability can be achieved using network processors [52] or operating system virtualization on conventional workstation processors [25].

1.2.3 New Vulnerabilities in Modern Routers

A side-effect of this shift from ASIC-based routers to routers with programmable packet processors is that it gives rise to a new class of vulnerabilities and corresponding attacks. Routers based on ASICs represented no practical attack target since their functionality could not be changed except by replacing actual hardware. In contrast, routers based on general-purpose processors that run software to implement packet processing functions exhibit the same kind of vulnerabilities that have been observed and exploited in conventional end-systems and embedded systems: attackers can attempt to crash the system, change its operation, extract information, etc. A study of vulnerabilities in network devices in the current Internet indicates that there are large numbers of potential attack targets [12].

Vulnerabilities in the network infrastructure itself are particularly problematic. First, routers are shared infrastructure and outages can affect a large number of users. Second, some routers at the core of the network are connected to links with extremely high data rates. If an attacker can modify the behavior of a router to send out malicious traffic, devastating denial-of-service attacks can be launched using only one or a handful of vulnerable systems. When considering the potential impact of a denial-of-service attack that is initiated from a core router that is connected to dozens of links with 40 Gbps data rates, it becomes clear that there is a need to protect these systems.

1.3 Contribution

In our work, we demonstrate a practical example of a novel type of attack that exploits these vulnerabilities and thereby attacks the network infrastructure itself. Specifically, we demonstrate how benign protocol processing code (in our case, the insertion of a protocol header) can be exploited by a single data packet and trigger a denial-of-service that consumes the entire outgoing link bandwidth of a router [9]. We show this vulnerability for two specific systems, a Click modular router [23] and a custom packet processor [57] based on the NetFPGA platform [26], as representatives for the broad class of routers with programmable packet processors.

We also show that processor monitoring techniques can help in identifying and mitigating these attacks. We present a design and results from a prototype implementation of a secure packet processor [8]. This packet processor can perform custom packet forwarding functions to support a wide range of protocols. A special processing monitor is used to track the instruction-level operations of each processor core. These operations are compared to a representative model of the packet processing task that has been obtained through offline analysis of the packet processing binary. Under normal conditions, the operations reported by the processor match the model in the monitor. If an intrusion attack occurs that changes the operations of the processor (e.g., to execute malicious code), then these operations no longer match the model in the monitor. The secure packet processor can detect these conditions and initiate a recovery step that resets the processor core and allows continued operation.

We demonstrate the effectiveness of this secure processor design by presenting results from an implementation on a NetFPGA platform. The prototype can detect an example attack where the control flow of the packet processor deviates from that in the original processing binary. The system can correctly detect this attack, halt the processor, drop the packet, and restore the system within 6 instruction cycles. This very small time for recovery allows our secure packet processor to operate at full data rate even when under

attack. The overhead for adding a monitoring system to the packet processor is very small (0.8% increase on slice LUTs and 5.6% on memory elements).

The specific contributions of this dissertation are:

- To our knowledge, the first practical example of a novel type of attack on routers with programmable packet processors,
- A prototype implementation and evaluation of the attack on a Click modular router system and on a custom network processor based on the NetFPGA platform, and
- A discussion of mitigation techniques, including processor monitors.
- The design of a secure packet processor that uses existing monitoring techniques to detect the effects of an intrusion attack. The system can quickly recover from such attacks by resetting the processor system.
- A prototype implementation of this processing system on a NetFPGA platform. The processor core is based on a Plasma core that is extended with our monitoring and recovery system.
- Results from the operation of the prototype system that illustrate the correct detection of an attack and the fast recovery mechanism that allows the system to run at full speed even under attack.

We believe that this work presents an important first step towards designing router systems that provide the necessary flexibility in packet processing but also provide sufficient security mechanisms to protect modern network infrastructure.

1.4 Organization of Dissertation

The remainder of the dissertation is organized as follows:

Chapter 2 discusses related work on current and next-generation network security.

In Chapter 3, the basic characteristics of embedded processors in modern routers are introduced. This chapter provides key concepts related to the architecture and the programming model of a modern router that is based on a multi-core grid of embedded processors. Here, the design and implementation of a custom packet processor prototype are described, while special attention is given to the performance and scalability of this system.

Chapter 4 is an overview of the security problems that arise due to the programmability in the data plane of networks. These are vulnerabilities that target the network infrastructure itself, and can have a larger impact than common attacks on end systems.

A specific attack example is presented in Chapter 5. Results from an implementation of the attack in a real network setup are shown in Section 5.2.

Chapter 6 presents a discussion of defense mechanisms against this type of attacks. Section 6.1 introduces the design of a secure packet processor while Section 6.2 presents the prototype implementation of the secure system. Section 6.4 presents an alternative defense mechanism, which is ideal for multi-core packet processor in terms of accuracy and resource requirements.

Chapter 7 outlines my future research directions, summarizes and concludes the dissertation.

CHAPTER 2

RELATED WORK

Programmability in the data plane of routers is widely used and many modern routers use programmable packet processors to implement protocol processing. Routers that use software for packet processing include workstation-based routers [19, 23], programmable routers [43], and virtualized router platforms [2]. High-performance router systems use multi-core packet processors (so-called "network processors") [6, 49]. Commercial examples of network processors are the Intel IXP2400 [21], the EZchip NP-3 [16], the LSI APP [27], the Cavium Octeon [7], and the Cisco QuantumFlow [10]. The number of processor cores in these chips ranges from as little as eight in the IXP2400 to over a hundred in the Cisco Silicon Packet Processor (SPP).

Addressing the problem of vulnerabilities in routers is also important in the context of research on the design of the future Internet [17, 35, 36]. The use of programmable packet processors is at the core of many future Internet designs (e.g., network virtualization [2, 50]). Thus, developing defense mechanisms to protect the packet processors in router systems is critical for the continued success of the Internet.

Network security research has focused on topics ranging from secure end-to-end protocols to anomaly detection heuristics [15]. To protect networks and end-systems from denial-of-service attacks, packet marking strategies have been proposed to identify spoofed sources [45]. Capabilities-based networks require positive authentication of traffic before forwarding is performed [5]. Extensions protect from denial-of-capabilities attacks [38] and provide one-hop containment [54]. These approaches defend against the effects of denial-of-service attacks.

The vast majority of security issues in networking are related to end-systems and protocols. One example is large-scale distributed denial-of-service attacks, which are generated by botnets [18]. Large scale DoS attacks have been previously studied in the context of worms [33]. Worms can spread quickly by infecting a large number of vulnerable end-systems and can absorb a large amount of network bandwidth. The key difference of the attack that we describe in this work is that it has an even more devastating effect: The attack is triggered with *a single packet*, absorbs *all bandwidth* of the outgoing link on the router, and can *propagate* to all vulnerable downstream routers.

The premise of a denial-of-service attack is that an attacker has a large number of systems from which traffic can be directed to a target. Therefore, an attacker needs to gain control over a large number of systems. This access is accomplished through intrusion techniques (e.g., trojan horse, etc.) that make the system (or parts of it) remotely controllable. From the network side, firewalls [32] and deep packet inspection [47] try to control end-system intrusion and thus limit the access to platforms from which attacks can be launched. On end-systems, virus scanner software can also identify some attacks. Secure protocols (e.g., IPsec [22]) are used to provide basic information security, including authentication and privacy.

Very little work has addressed security issues in the network infrastructure itself. A recent study [12] surveyed network devices that are considered vulnerable due to exposed administrative interfaces, which are part of the control plane of the network and can be protected by better management methods. In our work, we consider the data plane of the network, which inherently needs to be exposed and thus needs novel protection techniques. One such protection is based on processor monitoring, originally proposed for embedded systems in general [29] and recently adapted for network systems in our work [8]. Other defenses may be based on techniques from embedded system security [37].

While exploiting software vulnerabilities has been studied extensively on a range of different systems, it is important to note that packet processing systems on network routers

present unique challenges. Packet processors are highly parallel processor systems that typically do not run conventional operating systems. Also, the characteristics of a packet processing workload is fundamentally different (i.e., very simple and highly repetitive [41]) from what is encountered in other domains.

The use of virus scanner software as a defense against intrusion assumes that a sufficiently powerful processor and operating system are available. This assumption does not hold when considering embedded packet processors on routers. These systems frequently use network processors that operate without operating system support to maximize throughput performance. These embedded processing systems are vulnerable to intrusion just as conventional end-systems are [12]. Our work focuses on methods to protect these packet processors through hardware extensions that do not impact the overall processing performance.

We use processing monitors to track the operations on the packet processor. The monitor can determine if attacks occur because the processor's operations deviate from the operations that are valid (as determined by offline analysis of the processing binary). The techniques we use have been developed in prior work by us and others for the use in embedded systems in general [28]. Like network processors, embedded systems are characterized by the lack of operating system support for intrusion detection (and often the lack of performance to do so in software). Our previously designed hardware monitor for embedded systems can track each instruction of the processor and compare it to the processing model used by the monitor [28]. Other monitors [3, 39] use similar techniques, but operate at the granularity of basic blocks and thus are slower in detecting attacks. Similarly, the system in [60] determines correct operation across blocks of instructions. Other techniques extend the processor instruction set and micro-architecture to support special verification steps [34, 40]. Our approach differs from related work in that it does not require changes to the processing binary (which is beneficial for third-party code) and that attacks can be detected within a few instructions rather than at the end of a longer code block.

Our idea of using monitoring techniques to protect networking systems was first published in [55]. In this work, we make these general ideas more concrete by presenting a detailed design of the processing monitor and results from a prototype implementation on a NetFPGA system. The NetFPGA platform contains a Virtex2-Pro FPGA device and is used for experimental purposes in the networking domain. This work provides the first definite evidence that the proposed system can indeed detect and recover from attacks and can do so at speeds that do not degrade overall throughput performance. This last point is particularly important since the recovery mechanism should not present a target for a new type of denial-of-service attack where a single malicious packet can trigger time-consuming recovery operations.

Alternatively, programmable logic can be used in conjunction with the network processor in the router to detect attacks. The idea of augmenting fixed "hard" processors with on-chip programmable logic to achieve improved system efficiency has gained increasing acceptance in recent years. Recently, Intel announced the on-chip integration of an Altera FPGA core with an Atom processor [31] and several studies have examined techniques to embed a small amount of FPGA logic in an otherwise non-reconfigurable hardware platform [51]. The use of reconfigurable logic in network hardware is likely to increase given recent work [59] showing the benefits of using dynamic reconfiguration of programmable logic in adapting network router functionality.

CHAPTER 3

EMBEDDED PROCESSORS IN NETWORKS

Before we talk about vulnerabilities and potential attacks in the network infrastructure, we first describe key architectural concepts of network processors, which are essential components at the core of the network. We also briefly describe the design of a custom four-core packet processor, which was implemented in the Network Systems Lab at the University of Massachusetts Amherst, and is a part of my joint work with Dr. Qiang Wu. Based on this design we demonstrate an example of a DoS attack (in Chapter 5), and we evaluate our proposed hardware extensions for security provisioning (in Chapter 6).

3.1 Programmability

Modern routers use programmable packet processors on each port to implement packet forwarding and other advanced protocol functionality. The ability to change a router's operation by simply changing the software processed on router ports makes it possible to introduce new functions (e.g., monitoring, accounting, anomaly detection, blocking, etc.) without changing router hardware. An essential requirement for these systems is the availability of a high-performance packet processor that can deliver packet processing at data rates of multiple Gigabits per second. Such network processors (NPs) can be deployed as systems-on-a-chip based on multi-core architectures.

One of the key challenges in using a network processor to implement advanced packet processing functionality is software development. Many programming environments for NPs use very low levels of abstractions. While this approach helps with achieving high throughput performance, it also poses considerable challenges to the software developer.

Distributing processing workload between processor cores, coordinating shared resources, and manually allocating data structures to different memory types and banks is a difficult process. In environments where network functionality does not change frequently, it is conceivable to dedicate considerable resources to such software development. However, this software development approach becomes less practical in highly dynamic systems. The next-generation Internet is envisioned to be such a dynamic environment.

The next-generation Internet architecture is expected to rely on programmability in the infrastructure substrate to provide isolated network "slices" with functionally different protocol stacks [2,50]. In such a network, the processing workload on a router changes dynamically as slices are added or removed or as the amount of traffic within a slice changes. These dynamics require that the software on a network processor adapt at runtime without the involvement of a software developer. Thus, it is necessary to develop network processing systems where these dynamics can be handled by the system. The performance demands of packet processing do not allow the use of a completely general operating system. An operating system would use a considerable fraction of the network processor's resources. Instead, we focus on a solution where resource management is built into the network processor hardware and in turn allows a much simplified programming process.

We built a simplified network processor that attempts to hide the complexity of resource management in the network processor hardware. The software developer merely programs the functionality of packet processing. This approach contrasts other network processors, where functionality and resource management are tightly coupled (e.g., programmers need to explicitly choose allocation of data in SRAM or DRAM). By separating functionality from resource management, the system can more readily adapt to runtime conditions that could not have been predicted by the software developer.

3.2 Design Challenges

When designing a packet processing system, a design choice needs to be made on how complex a core processor is. More complex processors typically perform processing faster. However, they also require more hardware resources (memory, logic, interconnects) and consume more power. Assuming a fixed resource budget, a design can use either a larger number of simpler processors or a smaller number of larger processors. Using a larger number of simpler processor for packet processors provides several advantages specific to the networking domain:

- Higher throughput: Network processors are concerned with the throughput of the system (i.e., total amount of processing across all processors) rather than per-packet delay (i.e., executing speed of single processing task). A larger number of lower-performing processors achieves an overall higher throughput since the cost for speeding up single-thread performance is disproportionate to the performance gain. (Note that the processing delay on a router – even when performing complex tasks on a slow processor – is typically considerably less than the propagation delay encountered on transmission links. Thus, increasing delay for higher throughput is acceptable.)

- Lower power consumption: Following the argument for higher throughput, a system with more lower-speed processors consumes less power while achieving higher processing performance.

- Easier support for virtualization: It is convenient to provide isolation between slices at the level of a processor rather than attempting to achieve isolation of processing tasks within a processor. Having a larger number of processors available provides the basis for finer-grained resource allocation than is possible with a few large processors.

However, there are also several challenges when using a larger number of simpler processors. In particular, the management of these resources becomes more complex. Never-

theless, using a larger number of smaller processors is preferable when considering performance.

Another challenge in a packet processing system is to ensure that each packet processing function has access to data it needs. There are different types of data and information that are used for network processing:

- Packet processing program code: The instructions for processing a packet are typically stored in a dedicated instruction store.
- Packet data: The packet header and often the packet payload needs to be accessible to the packet processing function.
- Flow state: Many packet processing functions are not entirely stateless but maintain a small amount of per-flow state.
- Local processing state: This state encompasses temporary memory used by the packet processing function.
- Global processing state: This state information encompasses global per-function state as well as system-wide global state.

We use the term "context" to refer to the set of all these data and state information that are specific to one particular packet. In a typical implementation of packet processing systems, a processing function that tries to access this context needs to explicitly find the correct memory locations where it is stored. Such a manual management of packet processing context can lead to difficulties when programming packet processing functions and can also consume considerable processing resources. Instead, our packet processor design is based on providing hardware context management.

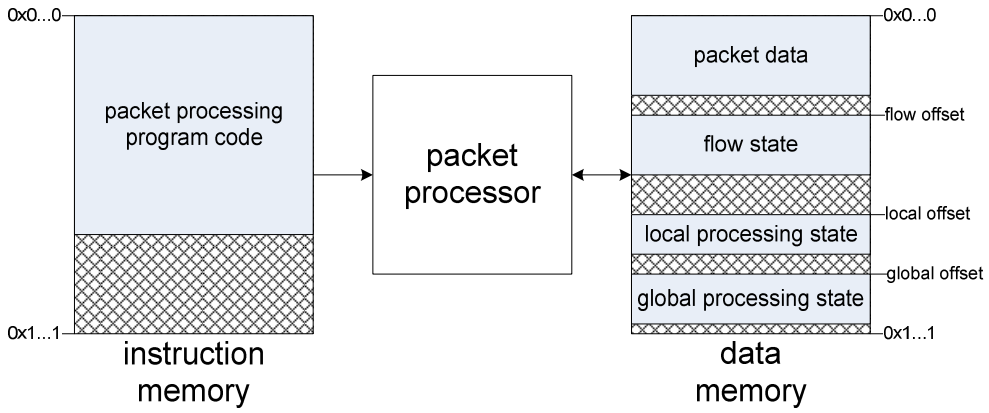


Figure 3.1. Processing context in simplified network processor design.

3.3 Software Development

In principle, a network processor consists of a grid of packet processors that are locally connected to each other. A control system determines how packets are moved between processors and what processing is performed. Using hardware support for moving packets between processor cores and switching processing contexts enables us to significantly simplify the software development process. Since the software developer does not need to explicitly manage packets or processing context, a much simpler processing environment can be presented. Through careful memory management, processing instructions, data structures, and the current packet can all be mapped to (virtually) fixed memory locations. Thus, the program can simply access them through static references.

Figure 3.1 illustrates this simplified environment. To achieve the desired simplicity, the packet processor is able to directly access on-chip memories, in which instructions (program code for multiple services), data and packets have been stored. The packet processor has an interface for reading program instructions and data memory and an interface for access to packet memory. In the instruction memory, the code for running a particular service is placed at a fixed, well-known offset. In the data memory we have placed the stack and global pointers at well-known offsets as well. With this design, packet processing and code development for packet processing is simplified. Packet data can be accessed via referenc-

```
#define IP_TTL 0x1000001E
#define pkt_get8(addr, data) \
  data = *((volatile unsigned char *) addr)
#define pkt_put8(addr, data) \
  *((volatile unsigned char *) addr) = data

typedef unsigned char _u8;
_u8 ip_ttl;

pkt_get8(IP_TTL, ip_ttl);

if (ip_ttl != 0){
  ip_ttl--; \\decrement TTL
  pkt_put8(IP_TTL, ip_ttl);
} else {
  ...handle TTL expiration...
}
```

Figure 3.2. Simple C program for accessing and decrementing the time-to-live field in the IP header.

ing the data memory on the (fixed) packet offset. Moreover, the program code is placed in a fixed location in the instruction memory and thus can be accessed easily by the processor.

An example of a piece of C code that accesses packet memory is shown in Figure 6.5. The code reads the time-to-live (TTL) field in the IP header and decrements it. Since the context is automatically mapped, the IP header can simply be accessed by a static reference. The hardware of the system ensures that this memory access is mapped to the correct physical address in the packet buffer that is currently in use. Similarly, data memory (and instruction memory) can be accessed. For example, to count the number of packets handled by an application, a simple counter can be declared:

```
static int packet_count
```

This counter can be incremented once per packet:

```
packet_count++
```

The automated context handling ensures that the memory state is maintained for the application across packets, and thus correct counting is possible.

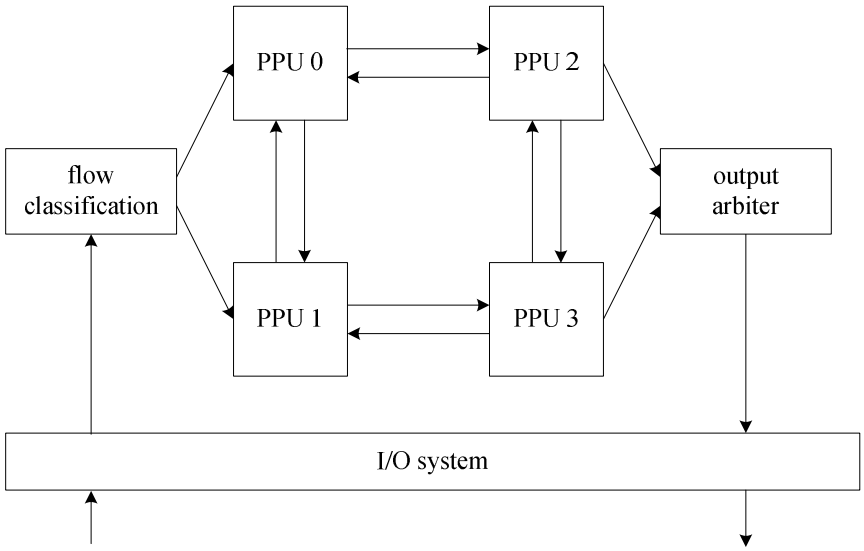


Figure 3.3. Network processing platform design.

3.4 Prototype Implementation

The high-level architecture of our four-core prototype system, which we have implemented on a NetFPGA [26], is shown in Figure 3.3. Packets enter through the I/O interface, get classified into flows and then distributed into the grid of packet processing units. Each processing unit has a set of packet processing applications preloaded (as determined by the runtime system), and is able to select the requested application based on control information determined during packet classification. Example applications could be IP-forwarding, IPsec, payload transcoding, privacy services, QoS, etc. After the processing steps have been completed, the packet is sent through the output arbiter to the outgoing interface. The processor core is a 32-bit Plasma processor [42], which uses the MIPS instruction set and operates at 62.5MHz.

One of the key design aspects of this system is that packet processors only use local memory. Avoiding the use of a global, shared memory interface helps in preserving the scalability of the design. As we show in the results in Section 3.5, the prototype implementation can scale to a 7×7 grid configuration with a linear increase in chip resources.

Table 3.1. IP forwarding throughput.

number of cores	small packets Mbps	small packets kpps	large packets Mbps	large packets kpps
1	154	302	2792	231
2	169	320	2769	229
3	167	327	2776	229
4	171	333	2785	230

3.5 Performance

In this section, we discuss performance results obtained from our prototype system. These results focus on throughput, performance, and scalability.

To evaluate the processing performance of the system, we consider the common IP forwarding application.

The throughput results for IP forwarding demonstrate the overall throughput performance of the prototype system. Table 3.1 shows the forwarding performance for small packets (64 bytes) and large packet (1512 bytes). For large packets, our network processor can achieve a peak forwarding rate of 2.79 Gigabits per second with only a single core. For small packets, the forwarding rate drops due to the per-packet overhead.

Note that our prototype is built on an FPGA-based system, which is convenient for prototyping, but not realistic for deployment in a real network. The clock rate of the processor units is only 62.5MHz. Achieving several Gigabits per second forwarding performance on such a system is a considerable achievement. In a realistic deployment, an ASIC-based implementation with considerably higher clock rates would be used. Thus, the forwarding performance would scale up accordingly.

To evaluate the scalability of our system architecture, we synthesized different configurations of our design on a larger FPGA (Virtex 5 XC5VLX330T). The increase in resource consumption scales with the number of processing units. The amount of registers, lookup tables, and flip-flops *per processing unit* is shown in Figure 3.4. The nearly constant values across all configurations indicate that our system can be scaled to large configuration without bottlenecks.

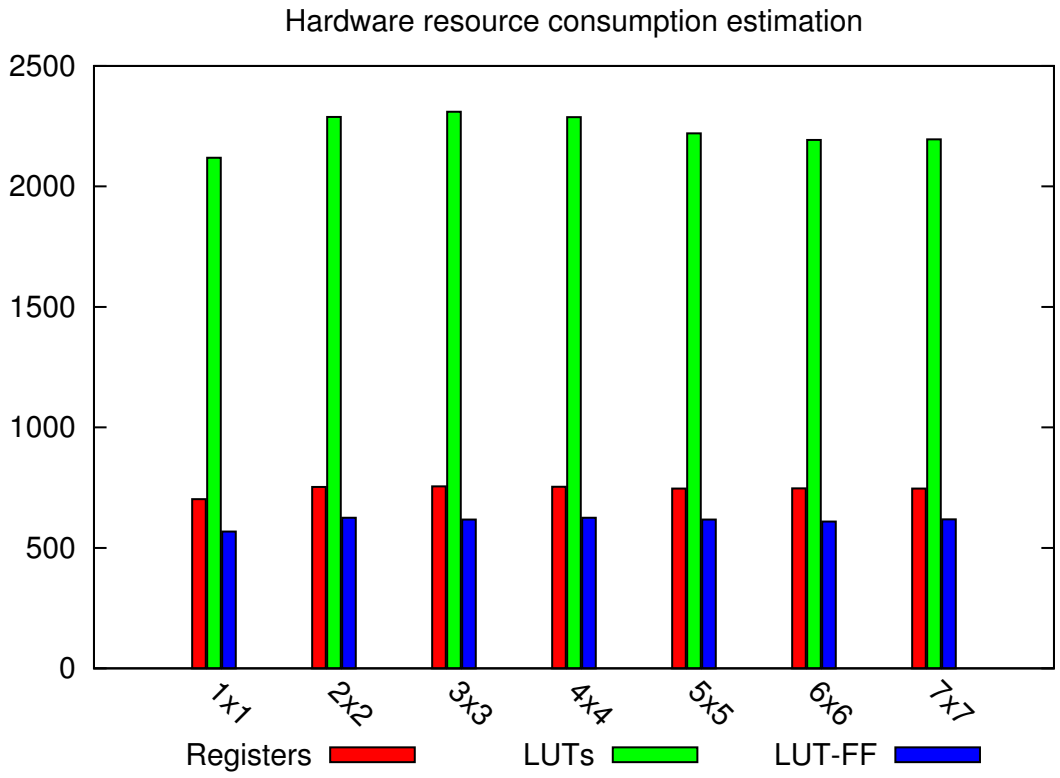


Figure 3.4. Resource estimation per processing unit on Virtex 5.

Thus, we believe that our simplified network processor system can provide ease of use as well as the necessary performance to support packet processing in the networks of today and the future.

CHAPTER 4

VULNERABILITIES AND ATTACKS IN NETWORK INFRASTRUCTURE

Before discussing the details of our specific attack in Chapter 5, we provide a brief overview of the vulnerabilities and potential attacks in the network infrastructure.

4.1 Attack Classification

The main functionality of the Internet (and any other data communication network) is to allow end-systems to communicate. As such, the Internet has served as a vehicle for many attacks where malicious users have gained unauthorized access to end-systems for the purpose of hacking, espionage, etc. In addition to such attacks that target access to data on end-systems, there are also denial-of-service attacks that aim to make end-systems temporarily inaccessible.

While attacks on end-systems are often highly visible due to news media attention, there are are also several other types of attacks on other components of the network. These attack types are shown in Figure 4.1 together with a few examples and common defense mechanisms. This figure by no means contains a comprehensive list of attacks and defenses, but merely a selection that helps in illustrating major differences in attack types. The control plane of the network, where routing information and other control information is exchanged, is a target of attacks that aim to disrupt the correct operation of the network (e.g., by stealthily redirecting traffic to malicious end-systems). In the data plane of the network, where the actual network traffic is transmitted between end-systems and routers, attackers may aim to eavesdrop on or intercept communications. It is here where a new type of attack that can lead to denial-of-service is emerging.

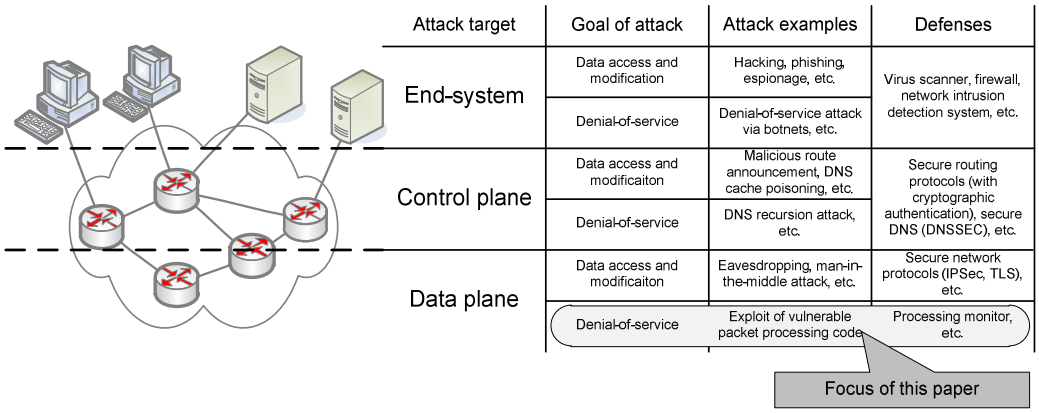


Figure 4.1. Examples of network attacks and defenses.

The attack on the data plane of the network that aims at denial of service is the main focus of this work. As explained in Chapter 1, this attack is rooted in the way modern routers are implemented. Until a few years ago, practically all high-performance routers used ASICs to implement packet forwarding operations. The function of an ASIC cannot be changed after it has been created and thus there was no way to change the forwarding operation of a router for the purpose of a network attack. However, the recent development of high-performance MPSoCs that are specialized for packet processing (i.e., network processors) has shifted router designs from ASIC-based packet forwarding to software-based forwarding on general-purpose processing systems [7, 10, 14, 53]. As with any software-based system, the flexibility provided by programmability also presents a security challenge as attackers can change the operation of the system for malicious purposes.

4.2 Commercial Routers with Programmable Features

Since attacks on the data plane of networks hinge on the presence of programmable packet processing systems in routers, we provide a brief overview of commercial router products. We show that programmable packet processors are indeed widely used in the Internet. Thus, these routers present the potential for the types of attacks we discuss in this dissertation. However, we do not want to imply that any specific products are vulnerable to

any specific attack. We merely want to show that the technology that is the basis for these attacks (i.e., programmable packet processors) is commonly used.

Modern Internet routers for the network core and the network edge typically employ programmable packet processors. This trend can be seen by examining the router products from two of the leading router equipment manufactures:

- Cisco: The Cisco CRS-1 core router uses a Cisco Silicon Packet Processor (SPP). The SPP is a 188-core processor that can be programmed to execute multiple advanced networking services. Cisco CRS-3is based on the 40-core Cisco Quantum Flow processor which also supports high level of parallelization and flexibility in terms of implementable services. Edge routers like the Cisco ASR 1000 and 9000 series are built on Quantum Flow as well. They are commonly used for applications that require parallel processing, security provisioning, QoS mechanisms and virtualization. The processors support any combination of layer 2 and layer 3 services and features. This is the most representative type of router for our work because services are programmed in pure C.

- Juniper: Juniper J and SRX series as well as Juniper MX series are used by both service providers and enterprise networks. They are designed on network processors e.g. Intel IXP models, which can efficiently implement services like load balancing, SSL offload, web acceleration, application security, access control et.c.

If we were to guess how an attacker would go about identifying vulnerabilities in modern routers and eventually exploiting them, it is natural to first consider the ways in which exploits have emerged in workstations.

Looking at recent attacks on Windows machines (e.g. Stuxnet worm 2010, Stateless Address Auto Configuration (SLAAC) attack 2011), we would have to think about the following questions:

- When was the attack first discovered?

- Was it first published or launched?
- How did the users become aware of it?
- When and how was it resolved?
- What was the vulnerability the attack was based on?
- What did the attacker know about the system/software in order to succeed with the exploit?
- Did the exploit require insider information?

Common causes of workstation attacks, which could very well apply to commercial routers are:

- Connectivity: Physical or virtual access to the system, increased privileges to simplify system operation
- Well-known Code: Software development based on common, well-known routines that an attacker might have encountered in other systems
- Dangerous input: All incoming packets/user input are not safe. There should be some sort of input validation in all systems.
- Absence of self-learning techniques: Systems should be able to remember past attacks and learn to drop dangerous input that has caused misbehavior in the past.

In order to hypothesize and predict how an actual attack can be realized on a commercial router, it would be helpful to work with major networking companies and discuss if they follow a specific process to identify potential attacks, how they resolve the issue once they discover it, how frequently they update their routers' firmware etc.

4.3 Security Model for Network Infrastructure

In our work, we use a straightforward security model that reflects the operation of the current Internet. Basically, we assume that the packet processing code on a router is benign (i.e., not intentionally malicious) and an attacker aims to exploit vulnerabilities in this code to change the operation of a router.

4.3.1 Security Requirements

The basic security requirement in our model is that the operation of the router does not change under attack. The infrastructure attacks we discuss here (see Figure 4.1) rely on the ability of an attacker to change the behavior of the packet processor (i.e., change in control flow or instruction memory) or its data (i.e., change in data memory). It is important to note that in most attack scenarios a modification of behavior is necessary even when modification of or access to data is the ultimate goal of the attack. This leads to two main security requirements that ensure that the router continues to perform correct protocol processing: (1) Benign packets should be processed according to protocol specifications without interference from possible attacks; (2) Malicious traffic should be identified and be discarded.

We show in Chapter 5 how an attacker can violate security requirement (1) and in Chapter 6 how processing monitoring can enforce requirement (2) and thus circumvent the problems caused by attack traffic.

4.3.2 Attacker Capabilities

The capabilities of an attacker that define the potential attack space include the following: (1) An attacker can send arbitrary data and control packets; (2) An attacker can modify instruction and data memory through exploits; (3) An attacker cannot modify the source code or binary of the protocol implementation before it is installed on the router; (4) An attacker cannot physically access the router. These capabilities reflect what most

practical attackers can do: they can try to hack a router remotely (i.e., (1) and (2)), but the basic functionality of the router is benign (i.e., (3) and (4)).

Based on this security model, we present a concrete attack [9].

CHAPTER 5

NETWORK INFRASTRUCTURE ATTACK

The main idea of the attack is illustrated in Figure 5.1. A cleverly crafted packet may be able to exploit software vulnerabilities (e.g., stack smashing attack) and change the operation of the packet processor. A simple change in the software could lead to an infinite loop where the same packet is transmitted repeatedly. Such an approach is particularly effective and damaging since the attack originates from within the network, where the compromised system may have access to links with tens of Gigabits per second bandwidth.

To describe the attack in detail, we briefly discuss the code vulnerability that we exploit, as specific example code that exploits this vulnerability in the context of protocol processing, and an example data packet that triggers an exploit of the vulnerability.

5.1 Vulnerability

Our attack exploits a vulnerability in the program that is executed on the packet processor of the routers. There are known C/C++ code exploits such as pointer subterfuge, use of `strcpy` and `memcpy` for buffer overflows, and integer vulnerabilities. A large number of them is present in commercial software designs and implementations. These vulnerabilities, under certain conditions, can be exploited by attackers, especially if programmers are not writing security-aware code.

The premise of our attack is that the packet processing code is benign and does not contain intentionally malicious code. The attacker sends a carefully crafted packet to one of the router's network interface cards. The processing of this packet turns the 'good' code/protocol routine that runs on the network processor into 'bad' code. There is nothing

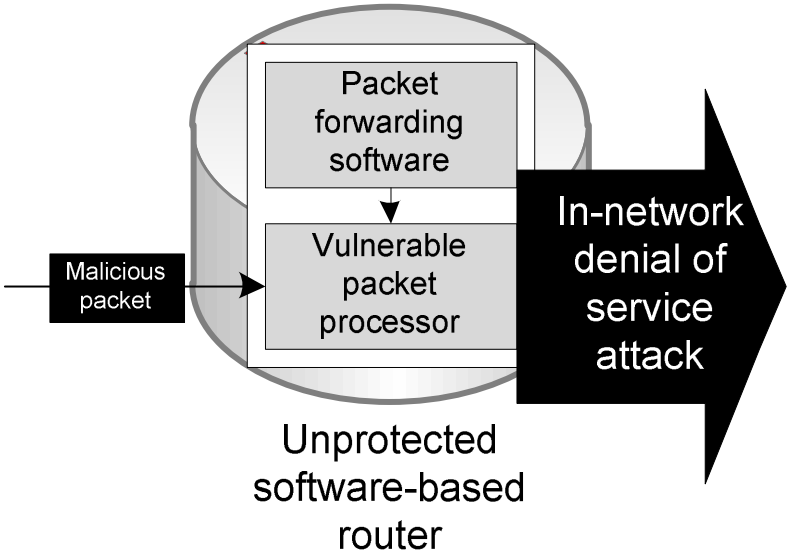


Figure 5.1. Example of in-network attack. Vulnerable packet processing systems on routers can be used to launch large-scale denial-of-service attacks with a single packet.

inherently wrong with the packet or the application code, but the combination of the two can lead to the processor's malfunctioning. In our case, the incoming packet changes the control flow of the routing and redirects it to malicious code that resides inside the payload of the attack packet. For all other packets, the correct processing is performed by the router.

The specific exploit we use in our attack is an integer vulnerability. Certain integer arithmetic operations, depending on the conditions, can result to unexpected outcome. Sign errors, truncation errors, integer overflows or underflows can occur, which, if not taken into account before the program execution, can lead to programs with unexpected behavior and security flaws [46].

Our attack is based on a vulnerability caused by an integer overflow. As we know, integers can represent values within a given range. For example, the integer type 'unsigned short' ranges from 0 to 65535. When a variable declared as short integer exceeds the upper limit, the assigned value wraps around zero in order to stay within the allowed limits. If the programmer does not anticipate this behavior, and the remaining of the program reuses that value at some point, potentially harmful things can happen. The following example code contains an integer overflow vulnerability:

```
unsigned short sum;
unsigned short one = 65532;
unsigned short two = 8;
sum = one + two;
```

The value assigned to the variable `sum` is not 65540, as one would expect, but 4 due to the limited amount of memory space that is assigned to it.

5.1.1 Vulnerable Protocol Processing Code

Routers perform a variety of protocol processing operations, ranging from simple IP forwarding to more advanced functions that include IPsec termination, intrusion detection, tunneling, etc. For our attack example, we assume that the protocol processing operation consists of adding a header to a packet. We are describing this operation in the context of the congestion management protocol described in [4] to be concrete, but it is important to note that the vulnerability can apply to a much broader range of protocol operations that add packet headers.

The congestion management (CM) protocol uses a custom protocol header that is inserted between the IP header and the UDP header. This process is illustrated in Figure 5.2. For the discussion of our attack, the detailed operation of the CM protocol and its header format is irrelevant. The important aspect is that CM adds a header in a packet.

The processing steps associated with the header insertion by the CM protocol are:

1. Parse headers to identify header boundary between IP and UDP.
2. Shift the UDP header (and higher layer headers and payload) to the right to make room for the CM header.
3. Insert CM header in packet.

Figure 5.3 shows pseudocode for the part of the program that inserts the new CM header in the original packet, which is the part of the code that contains a vulnerability.

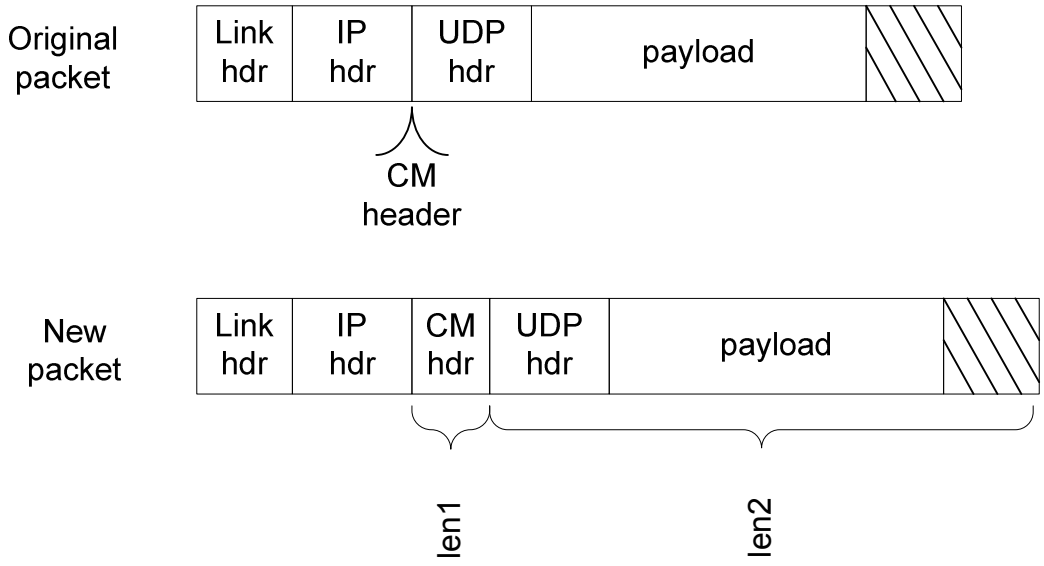


Figure 5.2. Protocol Header Insertion.

While writing the CM header generation part of the protocol, a security aware programmer would perform a check on the packet's total size before shifting the UDP datagram and inserting the new header into the original packet. This check is making sure that the outgoing packet – after the 12-byte CM header is appended to it – does not exceed the maximum datagram size. Only if the check `(CM_hdr_size + UDP_length) < MAX_PKT` passes, the original UDP datagram gets shifted by 12 bytes, and the CM header followed by the original UDP datagram are copied into the new packet buffer. The following line is the one that performs the shift and copy operation: `memcpy((new_pkt_buf+len1), original_pkt, len2);` where `len1` is the CM header size (12 bytes) and `len2` is the UDP datagram's total length. Since the total length field of the UDP header is a 16-bit field and the CM header is only 12 bytes long, the programmer could choose to assign `len1` and `len2` to 'unsigned short' integer types, so that the embedded processor's limited resources are not wasted.

```
#define MAX_PKT 1484

int generate_CM_header(int orig_pkt[], unsigned
short len1, unsigned short len2)
{

        int new_pkt_buf[MAX_PKT];

        unsigned short sum;
        sum= len1+ len2;
        if(sum > MAX_PKT) { return -1;}
        else {
        memcpy((new_pkt_buf+len1), orig_pkt, len2);


        ...

        return 0;
}

int main(int argc, char **argv)
{
        int orig_pkt[];

        ...

        generate_CM_header(orig_pkt, CM_hdr_size,
UDP_length);

        ...

}
```

Figure 5.3. Example Application Code.

This code, while correct for CM protocol processing, contains a vulnerability that is based on an integer overflow in the length check. A carefully crafted attack packet can exploit this vulnerability.

5.1.2 Attack Packet

The vulnerability does not exhibit problematic behavior for most "normal" packets that are short enough to accommodate the 12-byte CM header within the maximum IP packet length. An attacker, on the other hand, can send a long UDP packet that triggers an overflow. If an attacker chooses to send a regular, oversized packet (larger than `MAX_PKT`), the size check will fail. However, if an attacker sends a packet with a malformed UDP length field (for example with the 16-bit value 0xfffc (65532 in decimal)), then the code performs incorrectly:

1. `CM_hdr_size + UDP_length = 12 + 65532 = 8` (incorrect due to integer overflow)

2. `CM_hdr_size + UDP_length < MAX_PKT` (even though it is not)

3. 65532 bytes are copied into the `new_pkt_buf`, which can only accommodate 1484 bytes

Due to the malformed UDP total length field of the incoming packet, the processing of the protocol code leads to unexpected program behavior. A large amount of data ends up being copied into a buffer that was not designed to handle more than the maximum datagram size. The result is the notorious buffer overflow attack, which will overwrite the processor's stack.

Figure 5.4 shows the stack of the processor when the function `generate_CM_header` is running. We can see the original incoming packet residing in the bottom of the stack, as part of the main function. The last few bytes of the original packet correspond to the payload and contain the attack code, which the attacker has devised. Once the function

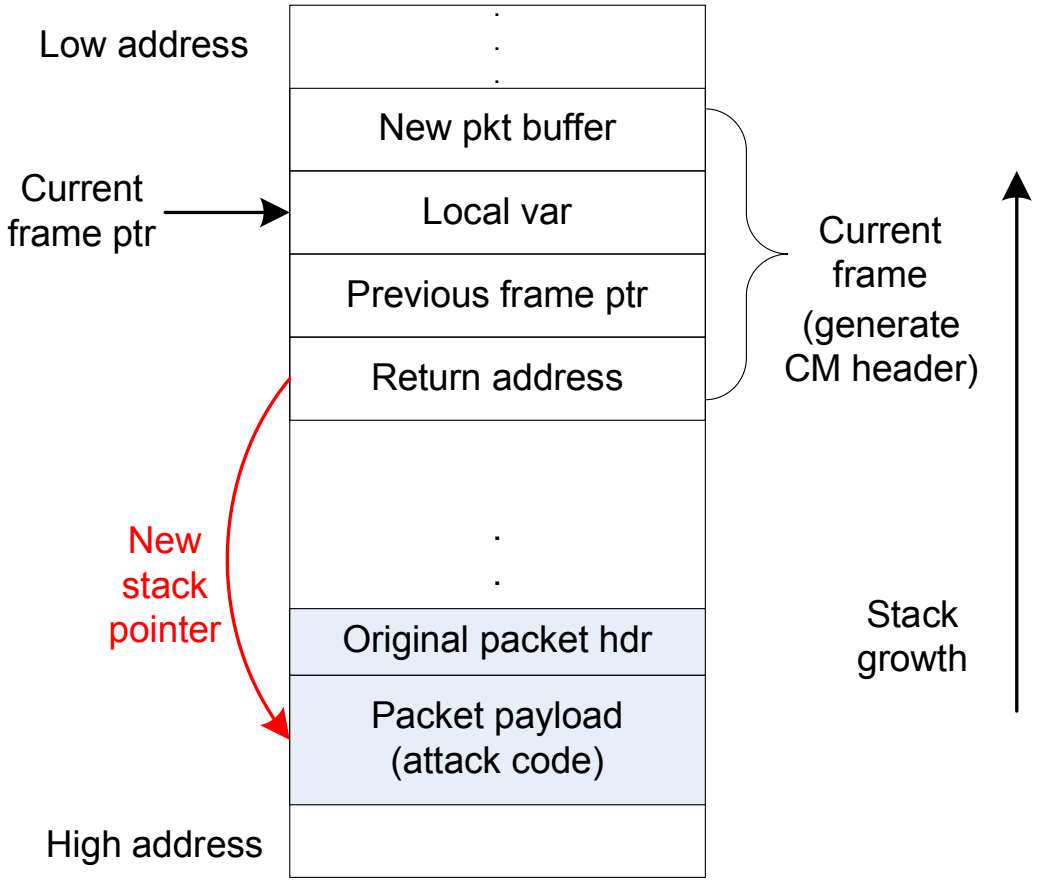


Figure 5.4. Stack Smashing.

`generate_CM_header` starts processing the incoming packet with the malformed UDP length field, the new packet buffer will overflow and start rewriting the local variables of the current frame, continue with the stack pointer and finally overwrite the return address of the current frame as well. Originally, the program should have jumped back to the calling function after finishing with the CM header generation, but when the return address is overwritten, the program will jump to whichever address the attacker has chosen! Of course, the attacker chooses to overwrite the return address with the stack memory address where the attack code begins. Thereby, the attacker can make the program jump to malicious code that is carried inside the packet payload.

In our attack, we insert a few instructions of assembly code into the payload, which repeatedly broadcast the same attack packet in an infinite loop. As we show in Section 5.2, a single attack packet of this type triggers a denial-of-service attack that jams the routers outgoing link at full data rate. While our attack is used for launching a denial-of-service service attack, it should be noted that an attacker could choose to run attack code with other purposes.

With this example, we demonstrate that vulnerabilities in software-based routers are not only hypothetical, but can occur in common protocol processing code. We also show that these vulnerabilities can be exploited to execute arbitrary attack code.

5.2 Attack Demonstration

To demonstrate the feasibility and effects of the attack described in this chapter, we show a prototype implementation in a real network. We have implemented the attack on the Click modular router which runs on a Linux system [23] and on a custom packet processor [57] based on the NetFPGA platform [26]. The custom packet processor uses a Plasma soft core [42], which is a 32-bit MIPS architecture processor.

Our experimental setup is shown in Figure 5.5. We send video traffic from one end-system to the other over a network consisting of two routers. The first router implements

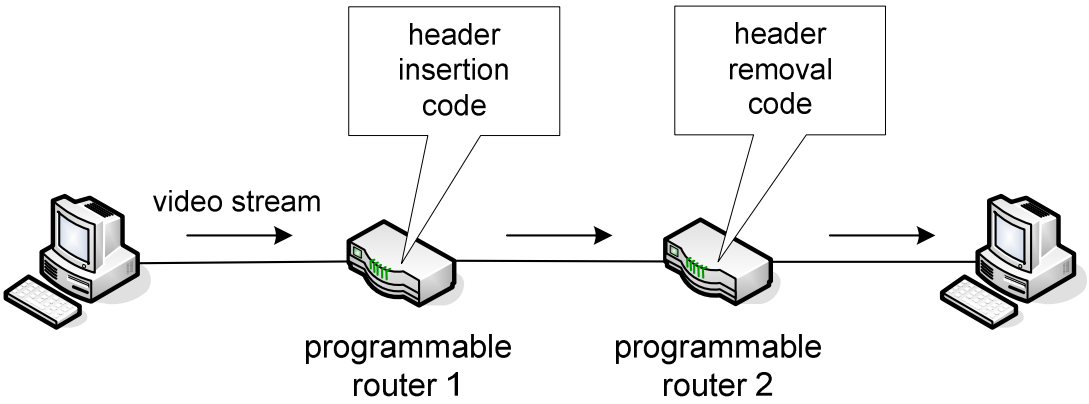
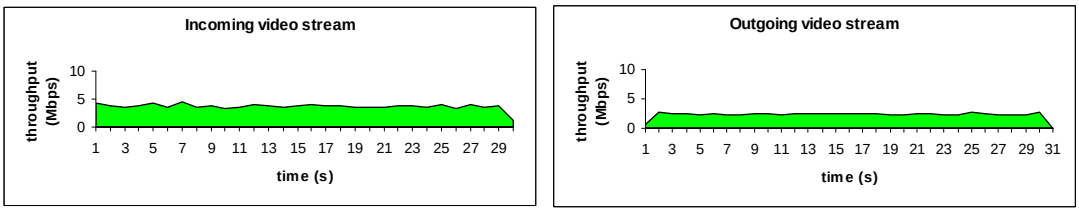


Figure 5.5. Experimental Setup.

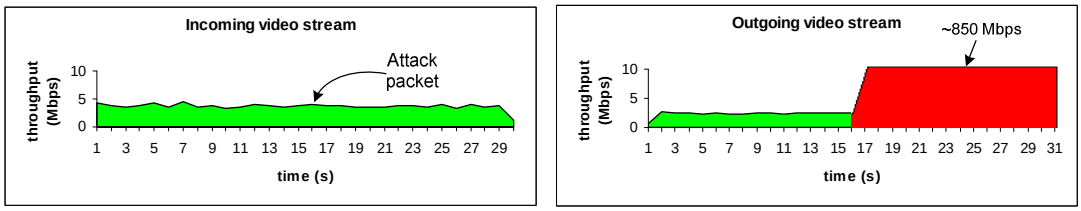
the CM header insertion processing described above. The second router removes the CM header. The header insertion routine on the first router is implemented as discussed in Section 5.1.1 and exhibits the integer overflow vulnerability.

We measured the incoming and outgoing traffic on the first router for different scenarios. Figure 5.6 shows the results for benign traffic and attack traffic. There is a 30-second video transmission as baseline traffic (shown in green). In the first scenario, only the benign traffic is sent and the router forwards it as expected. In the second scenario, a single attack packet is injected into the incoming traffic. Since the attack packet triggers an infinite loop of retransmitting itself, all output traffic consists of attack traffic (shown in red). There are two important observations: (1) No benign traffic is forwarded. Thus, the attack not only absorbs all unused bandwidth, but *all* bandwidth. (2) The amount of outgoing attack traffic is around 850 Mbps, which is close to the total link rate of the system. The performance of the system is limited to 850 Mbps due to the maximum clock frequency in our prototype. In a commercial high-performance router, attack traffic would be sent at the full link rate.

These results very clearly show that the attack we describe in this dissertation is indeed possible in practice and that it has devastating effects on the network by generating attack traffic at full link rates in the core of the network.



(a) Benign network traffic



(b) Benign traffic and single attack packet

Figure 5.6. Traffic Rates at Input Port and Output Port of Vulnerable Router. Benign video traffic is shown in green, attack traffic is shown in red.

CHAPTER 6

DEFENSE MECHANISMS

To defend against this type of attack on the packet processing systems of routers, a variety of security mechanisms could be used. These can range from using No eXecute (NX) bits to mark non-instruction memory to other security techniques used in embedded systems [37]. In this chapter, we propose a secure packet processor design [8]. We describe its operation and demonstrate that it can defend against the attack we describe.

6.1 Secure Packet Processing

The key concepts behind the design of a secure packet processing system are discussed in this section. More details on the implementation of the security features are covered in Section 6.2.

6.1.1 Security Model

To provide a basis for the discussion of security in our system, we state the security requirements and attacker capabilities explicitly. For security requirements, we assume that

- An attacker should not be able to make a router perform any action that deviates from normal forwarding behavior.
- Intrusion attempts through the data path of the router should lead to a drop of the offending packet.
- If an intrusion attempt has changed the internal state of the router, a recovery mechanism should reset the system to a secure state.

- Intrusion attempts should not lead to denial-of-service due to recovery overhead.

In the context of these security requirements, we assume that a malicious entity is able to perform the following actions:

- Send packets (control or data) to the packet processor, possibly triggering abnormal behavior.
- Gain remote access to the system and change the data memory, the instruction memory of the processor, log files, or extract and modify secret keys.
- Launch Denial-of-service attacks by sending massive traffic or by directly disabling links.
- Use reprogramming interfaces to control the entire router.

However, an attacker does not have physical access to the router and cannot access the binary file of the application currently executed on the packet processor, because it resides outside the platform. Once it is launched on the instruction memory of the hardware platform though, memory modification is considered a potential attack scenario.

The attacks mentioned in this section is not a comprehensive enumeration of all likely scenarios. We just outlined the general context of the possible ways in which a misbehaving user can attack a packet processing system. We tried to be as general as possible in our assumptions, and include most of current and next generation network vulnerabilities.

6.1.2 Attack Detection through Monitoring

The main idea behind our secure packet processor is to integrate monitoring functionality into the hardware of the packet processing system. When an attacker attempts to hack into the software-programmable processor cores of the system, they may succeed in changing the processing behavior of the system. However, the monitoring systems in the packet

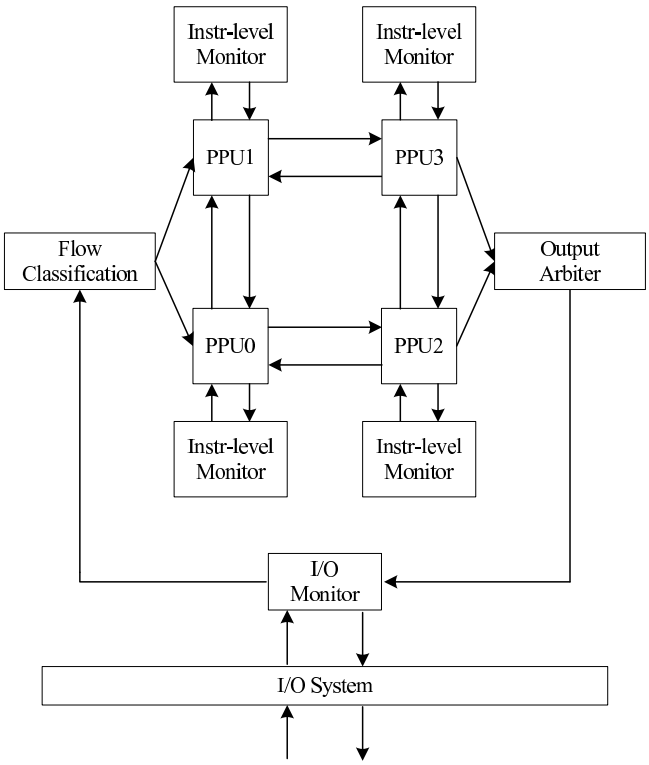


Figure 6.1. Security Monitoring on the Packet Processor.

processor can detect this change and trigger a response (i.e., packet drop and system recovery). Since our monitoring components are embedded in the system hardware, it is difficult for a hacker to attack both the processor and the (hard to access) monitors at the same time. Thus, this approach by design provides more security than a conventional general-purpose processing system.

There are several important challenges that need to be met when using hardware monitors in a processing system:

- Correct detection: The monitoring system needs to be able to correctly identify intrusion attacks. Our system achieves this by checking for any deviation from the known correct operation of the packet processor.

- Fast detection: When intrusion occurs, it is important to detect it quickly to reduce its potential impact. Our system can detect (and recover from) deviations in processing operations within only four processing cycles.

- Low overhead: The resource requirements for a monitor should be small to limit the impact of monitoring on system cost. Our monitoring system only requires single-digit percent additional hardware resources compared to a conventional packet processor implementation.

Before discussing the detailed operation of the monitoring system, we describe the high-level system architecture.

6.1.3 System Overview

The main design goal of our system is to provide security techniques to defend against potential attacks on a software-programmable packet processing system. Our system builds on the four-core prototype of a packet processor as described in Chapter 3.4 [57].

Figure 6.1 presents a system (high) level view of the software-programmable packet processor that uses security modules to ensure the correct functionality of a modern router. As discussed in [57] next generation routers are expected to have multiple packet processing units, in order achieve fast and balanced processing of packets that belong to different flows. A flow classification unit assigns packets to specific flows, and an output arbiter module sends the processed packets to the corresponding outputs.

Each packet processing unit will be possibly executing a different program in order to process the packets that are distributed to it. If we assume that an attacker is able to access and modify the instruction memory of each individual PPU (processing attack), the whole router will be compromised. To prevent that from happening, we can have an instruction-level security monitor attached to each PPU, which checks each and every instruction executed on the processor core in real time and determines if it is valid or not. We will explain the way this check is performed in the next section.

Moreover, due to vulnerabilities in the data path, we would expect the packet processor to be attacked at the protocol level as well. We can imagine a situation where valid processing instructions are executed on the PPUs, but still the overall router behavior is abnormal. For example, if the IP-forwarding routine is running on one of the cores, a Denial-of-Service attack could flood the system by requesting a large amount of duplicate packets to be forwarded to the output interfaces. To counter such kinds of attacks, we could use an I/O monitor attached to the I/O interface of the packet processor, which checks certain characteristics of incoming and outgoing packets: payload checksum, packet count, tags or time-stamps in the header of the packets etc.

6.1.4 Monitoring Functionality

Both monitoring systems function independently from the packet processor. They use separate hardware resources, making sure that an attack targeting the processor will not affect the security monitors' operation. Moreover, they use up as few resources as possible, while keeping the monitoring speed synchronized with the packet processor's speed.

The main idea behind the instruction-level monitor is illustrated in Figure 6.2.

- Prior to installing a specific protocol processing routine on the packet processor, we analyze the binary file of the application by breaking it down to basic blocks of instructions and determining all the possible execution paths.
- The derived information is stored in a 'basic block' data structure on the hardware platform.
- The processor, during runtime, keeps updating the security monitor with the monitoring stream.
- If the processor's current execution path deviates from the correct one - as instructed by the basic block data structure - the security monitor detects an error.

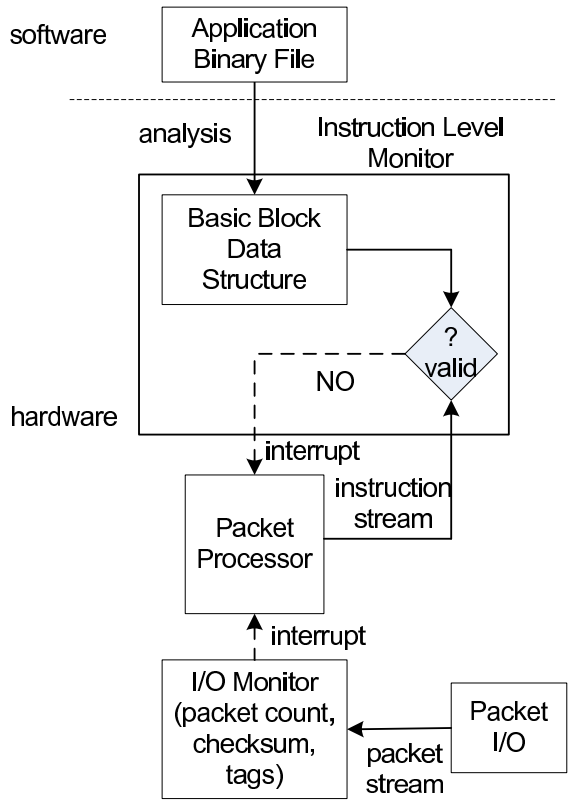


Figure 6.2. Monitoring System Overview.

In case of error detection, the monitoring system assumes that the instruction store must have been modified, and it takes the recovery route. It interrupts the packet processor's operation on the specific packet, reloads the protocol processing code on the processor from a storage place that is assumed to be secure and not accessible by the attacker.

To enhance the instruction-level monitor functionality, an I/O monitor [56] can be used to track the I/O behavior of the packet processing system. This is a higher level monitor that records protocol specific characteristics. It is not connected to the packet processor, but to the input and output interfaces. The I/O monitor correlates the stream of incoming packets to the stream of outgoing packets to detect cases where the router is not operating as expected. The abnormality here is detected from a network protocol perspective, whereas the router's function can look legitimate from a processing point of view (e.g. correctly executed instructions during a DoS attack).

There are several types of information that can be collected by just observing the incoming and outgoing packets. The most intuitive thing to document is the number of incoming and outgoing packets, and check for imbalance. Of course, if the flow of packets increases in the output interface, this does not always indicate an attack condition. The I/O monitor should be able to account for protocols that allow for multicast or broadcast services. Moreover, the checksum of the packet's payload can be computed to identify unauthorized alterations. By placing tags or time-stamps in the headers of the incoming packets, the I/O monitor will be able to detect whether a specific incoming packet is directed to the output interface with significant delay. Such an event signals abnormal degradation of processing performance.

6.2 Prototype Implementation

In this section, we provide in detail the design and proof-of-concept implementation of a packet processing system that uses the two security techniques we have described above.

Our prototype is implemented on a NetFPGA [26]. Our design is scalable and can be ported on other FPGA platforms or ASICs.

6.2.1 Instruction-level Monitor

For this prototype, to be consistent with the NetFPGA design and the packet processor speed, we used 64-bit data path and designed all the units to operate at 62.5MHz. The security monitor runs in parallel to the packet processing unit, and is designed to use four pipeline stages.

The first task is to decide what the monitoring stream, which the PPU continuously sends to the security monitor, should be. According to our assumptions, an attacker can abuse the packet processor's operation, either by modifying the current protocol routine to execute malicious code, or by adding some piece of code that performs malicious operations. We can monitor such malicious behavior by making the packet processor stream information regarding the current execution path in realtime. There is a variety of options to choose from:

- Opcode: By sending to the monitor opcode information, we monitor the operations performed on the processor, which indicate the functionality of the executed application. For an attack to become possible, the attacker will have to replace the instruction set, with another malicious set of instructions that use the same opcodes in the exact same sequence.

- Instruction address: Since the memory address used to store the instruction set is unique, the attacker would have to write malicious code that stores the new instructions in the same location in the instruction memory as the original application does. This would also require the malicious code to branch at the same exact points with the legitimate code.

- Instruction address+Instruction word: This kind of streaming pattern combines two pieces of information, and makes it harder for an attacker to come up with attack code

that goes undetected. We could also add the opcode, or control flow information to the monitoring stream, but this will cause a significance increase in the system's resource consumption.

- Hash of any of the above: The processor is streaming a compact hashed value of any of the above combinations. The more bits we use to compute the hash, the stronger the monitoring pattern is. However, the number of used bits will affect the memory utilization. After all, it is a trade off between available memory on the hardware platform and the strength of security features.

Depending on the information we choose to stream, the software analysis and the contents of the basic block data structure shown in Figure 6.2 have to be adapted accordingly. For our prototype, the instruction address information was used. Before we load a specific protocol processing routine on one of the processor cores, we analyze the application binary file off-line and break it down to a number of basic blocks. We place instructions that are executed the one after the other in the same basic block, which ends with a conditional or unconditional jump instruction. We use a block RAM memory on the FPGA to store information about the program's execution path. This memory (data structure) is indexed by the instruction address sequence of the application and contains two blocks for each entry. The first one is the basic block each instruction memory address belongs to, and the second is the potential next hop address the instruction could jump to. This BRAM is used as a guide to the correct processor core operation.

The implementation level details of the instruction level monitor are shown in Figure 6.3. Each pipeline stage takes once clock cycle to complete.

In the first pipeline stage, we extract the address of the currently executed instruction on the packet processing unit. We use this address to index the BRAM, which takes one cycle to output the basic block in which this instruction resided, and the next hop address (in case of a jump instruction), if there is one.

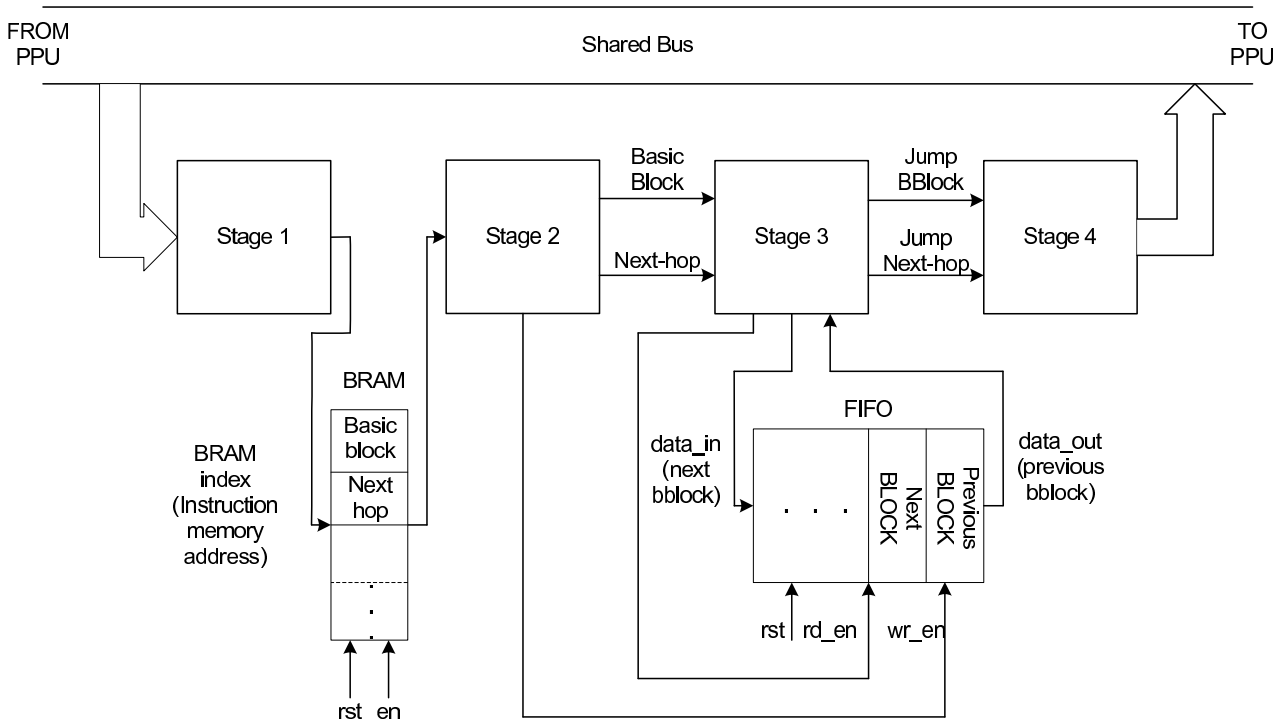


Figure 6.3. Instruction Level Monitor Design.

In the second stage, we propagate the current basic block and next hop information we get from the BRAM, and give those values as input to the third stage. At the same time, we record the current basic block information into a FIFO module. This FIFO is used to keep track of the execution path, by storing the previously and currently executed basic block numbers. This module has minimal memory requirements, because it only contains two values at a time. When we read from it, the head of the FIFO outputs the previous basic block, and when we write in it, we record the current basic block, which we read in the next clock cycle and so on.

The third and fourth stages are the most important, since they implement our monitoring algorithm:

- Check if the current basic block number matches the basic block of the previous instruction.
- If it does, it means it is a valid instruction – continue to the next one.

- If not, check if this instruction is within the next basic block.
- In case it is, this is a valid basic block jump – continue with the next instruction.
- If not, go to the fourth stage and use the next hop address information to index the BRAM. Verify that the currently executed instruction is a valid jump instruction.
- If it is, it denotes correct operation – continue.
- Otherwise, signal that the packet should be dropped.

In the final stage, the monitor sends the packet drop signal to the packet memory unit, which stops the processing of the current packet. The corresponding packet buffer drops this packet and is ready to receive a new one. At the same time the instruction memory is reset, so that the harmful code is overwritten and does not affect the next packet to be processed by the packet processor. While resetting the instruction memory, we first switch to a backup piece of memory (which resides safely inside the hardware platform) and then start reloading the memory initialization file back to the infected instruction memory of the processor. In this way, in case another attack happens during the processing of the next packets, we can immediately switch back to the corrected instruction memory.

6.2.2 I/O Monitor

The I/O monitor design in our prototype is very simple and is shown in Figure 6.4. The purpose of this implementation is to illustrate just one example of how a high-level (protocol related) monitor should function, and show that it is possible to implement this feature on our packet processing system with no additional overhead, and without having to compromise on the throughput of the processor. We use resources (logic) that are already present on the reference NetFPGA router, to design two counters, one attached to the input queue interface keeping track of the incoming packets, and the other one connected to the output queues, counting the outgoing packets (within a specific window).

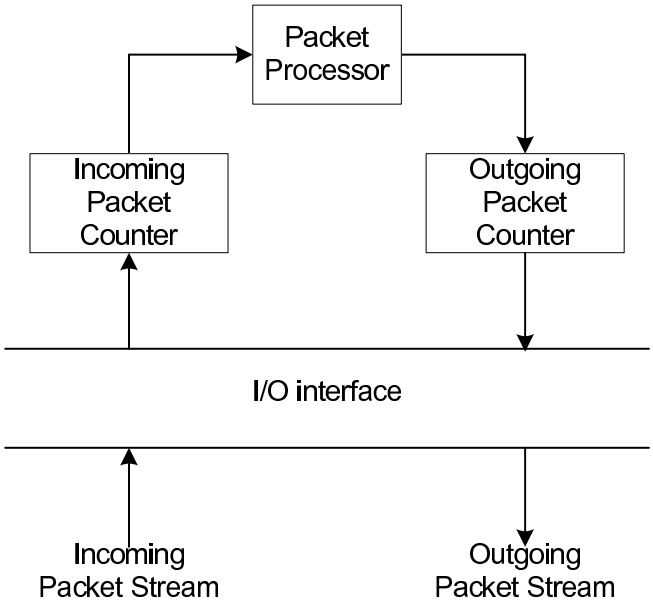


Figure 6.4. I/O Monitor Design.

Thanks to the flow classification module implemented on our router, we can easily extend the packet count functionality of the I/O monitor, and make the monitoring operation protocol-aware. It is possible to check the header of every packet to determine which protocol is currently running on each processor core. Based on this information, we can make the I/O monitor treat traffic in a protocol specific way. For example, if it is determined that a unicast protocol is executed (http, ftp etc.) the I/O monitor would expect the incoming packet count to be equal or greater than the outgoing packet count. In the case that the protocol allows for multicast capability, an increase in the outgoing packet count should be considered legitimate protocol behavior (one incoming packet corresponds to N outgoing). In either scenario, once unusual operation is observed, the packet memory should be flushed and the packet processor reset.

6.3 Evaluation

In this section, we demonstrate the correct functionality of the implemented security features, by crafting an attack example. We also discuss resource utilization and performance results. For the system's performance, we use the delay in detecting abnormal instructions and the system's throughput as metrics.

6.3.1 Triggering Invalid Behavior

In this scenario, we first load a unicast IP-forwarding routine on the packet processor. For our prototype, we have chosen to store instruction addresses for monitoring purposes. Alternatively, as mentioned in Section 6.2.1, we could have implemented a security monitor that contains the hash value of the instruction address and the corresponding instruction word. Such a monitoring pattern uses less memory resources and makes it even harder for the attacker to come up with attack code [28].

Let's assume that our application code at some point executes a branch instruction, with two possible basic blocks where the program can jump to. The corresponding disassembled application code is shown in Figure 6.5. As we can see, the potential jump targets are memory addresses 0x0218 or 0x01e4. The former contains valid code, whereas the latter corresponds to attack code. On the other hand, inside the security monitor we store instruction addresses only for the valid basic blocks.

Since we do not inform the security monitor that the program can possibly branch to memory address 0x01e4, the monitor will consider a jump to this particular address as an attack. So, in order to trigger an attack, we can send to the processor packets that cause our program to take this specific execution path (address 0x01e4). This path is unknown to the instruction level monitor, thus, the monitor treats the received packets as attack packets.

6.3.2 Security Monitor Operation

In Figure 6.7, we show how the whole system (packet processor and security monitor) operates under the attack scenario. The horizontal axis denotes the clock cycle at which

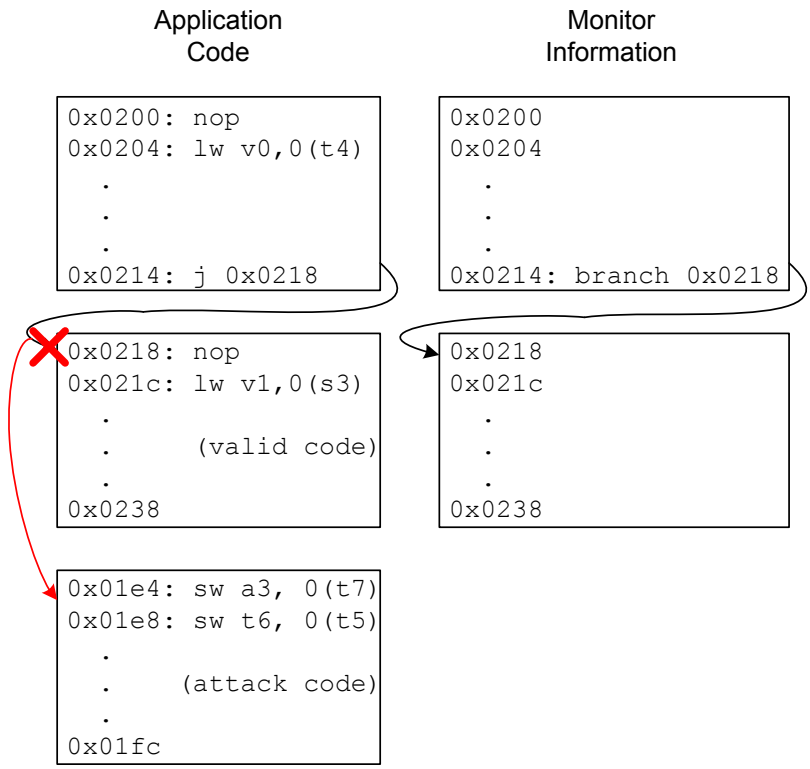


Figure 6.5. Attack scenario.

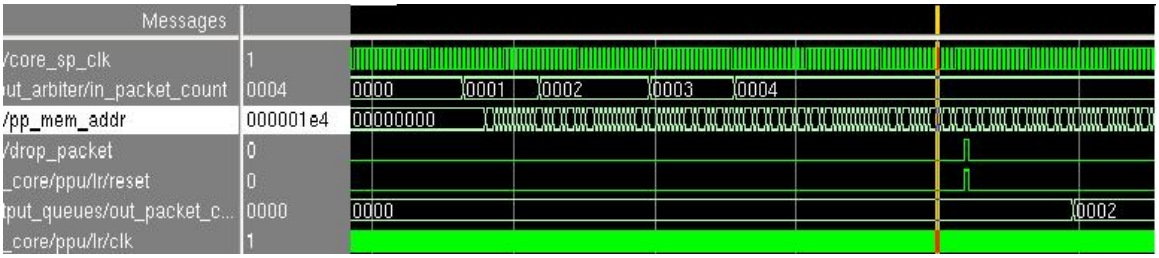


Figure 6.6. Simulation Results.

each operation takes place. In our experiment, we send 4 small packets (around 60 bytes long) into the router. The first is an 'attack' packet and the remaining 3 are valid packets. We should note here that under normal operation the total time the packet processor takes to forward a small packet is around 600 clock cycles [57].

The packet processing system functions as follows: The four packets come back to back into the packet processor's packet buffers. At some point, while the first packet is getting processed, the program jumps to the memory address 0x01e4, which is unknown to the monitor and triggers an attack response. The monitor starts performing the operations we described at Figure 6.3 in four pipelined stages, and 5 cycles later packet 1 is dropped. At this point our instruction-level monitor stalls the processing of the current packet, drops the packet, and in the same clock cycle the instruction memory is reset. The recovery phase of the system takes approximately 6 cycles. After those few cycles, the processing resumes and the remaining three packets are forwarded normally. Because of the 6 cycles, in which the system stays idle (recovery phase), the total processing time of packets 2,3 and 4 increases from 600 to 606 cycles.

In Figure 6.6 we can observe the same sequence of events in our simulator tool (Modelsim). Signal in_packet_count counts the incoming packets, out_packet_count the outgoing, and 'drop packet' is the signal that notifies the processor to drop the current packet. Packet 1 gets dropped when the signal $pp_mem_address$ is equal to 0x01e4, which is the initial instruction address of the 'attack' block of instructions. By setting the signal $ppu/lr/rst$, we rewrite the instruction memory of the IP-forwarding application. Due to space constraints, we only show packet 2 coming out of the output queue.

Parallel to the instruction level monitor, our I/O monitor counts incoming and outgoing packets. It reports 4 incoming packets and 3 outgoing. Since we experimented on the unicast IP-forwarding routine, our I/O monitor considers the protocol level behavior legitimate. As expected, it does not detect the instruction level attack.

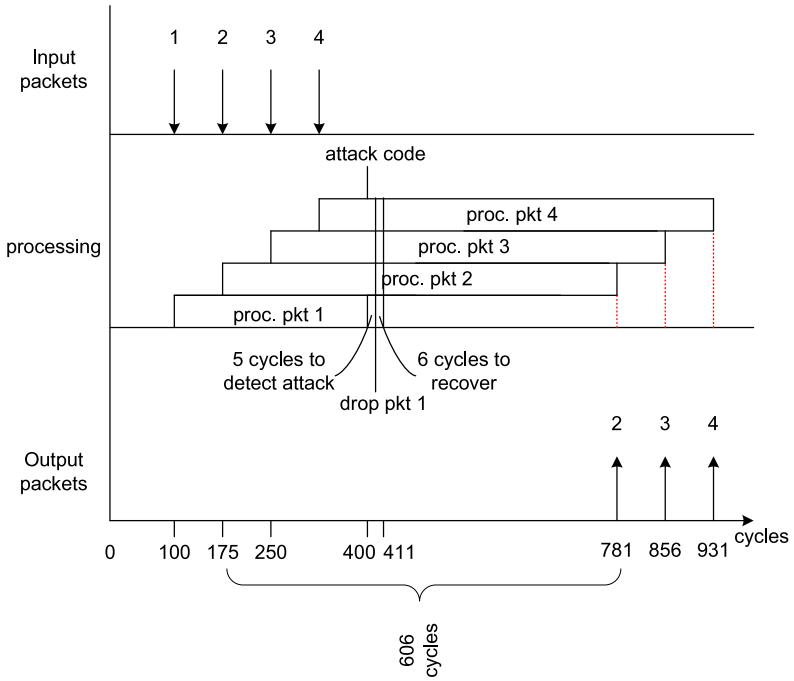


Figure 6.7. Security Monitor Operation.

A theoretical scenario where we would have the opposite outcome (the I/O monitor detecting an attack while the instruction level monitor does not detect the attack) would be if for the same unicast IP-forwarding protocol, we duplicate some packets and send them to all ports jamming them. Then the I/O monitor would count more outgoing packets than incoming, which is not usual behavior for a unicast protocol. On the other hand, there is no reason for the instruction level monitor to detect an attack, since the processing routine (forwarding) is executed correctly for all packets.

6.3.3 Performance Results

As mentioned in the previous section, once an attack is detected, the recovery phase of the instruction level monitor lasts only a few cycles. This time is necessary for the instruction set to be correctly reloaded on the NetFPGA. Compared to the number of cycles it takes for the processor to forward even a small packet (around 600 cycles), the time our system stays idle before resuming correct processing functionality is not significant. This

Table 6.1. Resource consumption and performance of single core system.

	Single core	Single core w/ main monitor	Single core w/ 2 monitors
Slice LUTs	15,025	15,112	15,134
BRAM (RAMB16s)	124	130	130
throughput(Mbps)	67.2	64.1	63.9

is an important feature because, otherwise, an attacker could just keep on sending packets that cause the system to misbehave, so that the processor is locked into a repeated effort of long recoveries, without doing any useful processing at the same time. That would become a vulnerability of our recovery mechanism that leads by itself to DoS attacks.

Here, we performed an experiment to measure the throughput of 1) our single core packet processing system and 2) our four-core packet processing system when the security monitors are on. Using three of the Ethernet ports on the NetFPGA system, we connect the network processor to three workstation computers for traffic generation and trace collection. The routing and processing steps for flows on the network processor are set up statically for each experiment. First, we experiment by sending valid packets only, and then by sending a combination of valid and invalid packets. We did not notice major changes in the processor's throughput in neither the single nor the four-core secure packet processing systems. We can still achieve high data rates (maximum of 100Mbps). In Table 6.1, we report the average throughput numbers for the single core processor and in Table 6.2 the corresponding results for the four-core processor with and without the security monitors. The important thing to note is that the throughput of the processors with embedded security monitors is almost the same with the throughput that the cores achieves by themselves. The data rates are in the order of 100Mbps because the experiment was performed by forwarding small packets. In the case of large packets the packet processor can achieve throughput greater than 2Gbps [57].

As for the resource consumption of our monitoring systems, the packet processing system with both security monitors uses 0.8% more slice LUTs compared to the single core processor alone. Memory-wise we observe an increase of 4.8% in the consumption of

Table 6.2. Resource consumption and performance of four-core system.

	4-core	4-core w/ main monitor	4-core w/ 2 monitors
Slice LUTs	22,469	22,847	22,872
BRAM (RAMB16s)	158	164	164
throughput(Mbps)	74.6	72.1	72

block RAMs. For the four-core system we observe that the logic utilization increases by 1.8% while memory consumption goes up by 3.8%

6.4 Defenses in Multi-core Packet Processing Systems

Packet processors have multiple simple embedded processor cores that can be programmed to handle network traffic. The hardware monitoring technique that has been proposed so far can protect each of the individual cores from potential attacks. In this section we present efforts in further improving the protection mechanisms for packet processors.

In previous sections, we have been using the NetFPGA platform for our designs, which is a reconfigurable hardware platform optimized for high-speed networking. It is based on the Xilinx Virtex-II Pro 50 FPGA, which has limited available logic, and can barely fit a 4-core packet processor. Since the Reconfigurable Computing Group of our department managed to port the NetFPGA reference router design to the Altera DE4 development board, we used the same board for our 4-core secure packet processing design. This board has approximately 10 times more logic than the Virtex-II Pro FPGA board and our preliminary calculations show that it can even be sufficient for a 16-core design. The objective of this implementation is to explore alternative monitoring implementations and evaluate the throughput performance and the scalability of a multi-core network processor with integrated security features. The following work was conducted in collaboration with Harikrishnan Chandrikakutty, Deepak Unnikrishnan, Russell Tessier and Tilman Wolf. Harikrishnan was in charge of the hardware implementation of the new monitoring idea and Deepak assisted in the experimentation part of the project.

6.4.1 Programmable-logic-based Monitoring

As we have shown in [9], vulnerable packet processing code can be exploited to launch a in-network denial-of-service attack, and, as a response to that, hardware monitoring techniques can be effectively used to reduce the vulnerability of packet processors in routers. In the networking domain, low overhead and fast detection speed are particularly important. Therefore, we need to think of ways to further improve the protection mechanisms for network processors.

One key aspect of processing monitors is that they should dynamically adapt to the processing operations that are implemented on the processor that is monitored. That means that the monitoring functionality may need to adapt during runtime as the workload of the system changes. In networks, there can be considerable variation in the processing workload due to changing network traffic [58] and thus the ability for adaptation in the monitoring subsystem is a necessary system requirement. In addition, network processors are based on multi-core architectures where traffic is processed by many parallel cores. Since many of these cores perform the same type of processing, it is possible to share some of the monitors among cores.

Our previous monitoring technique [8] has generally assumed a separate monitoring structure implemented in fixed hardware for each network processor core. Monitor functionality is programmed via an embedded memory within the monitor and in its preliminary form the system does not take advantage of monitor sharing between cores. However, for network processors which do not need monitoring, the dedicated monitoring hardware is left unused, wasting hardware resources. These observations suggest that programmable logic is an ideal platform for implementing processing monitors since it allows for high-performance implementation, reprogramming, and workload adaptation. Also, if the logic is not used for monitoring, it could potentially be used for different functions, such as network statistics gathering.

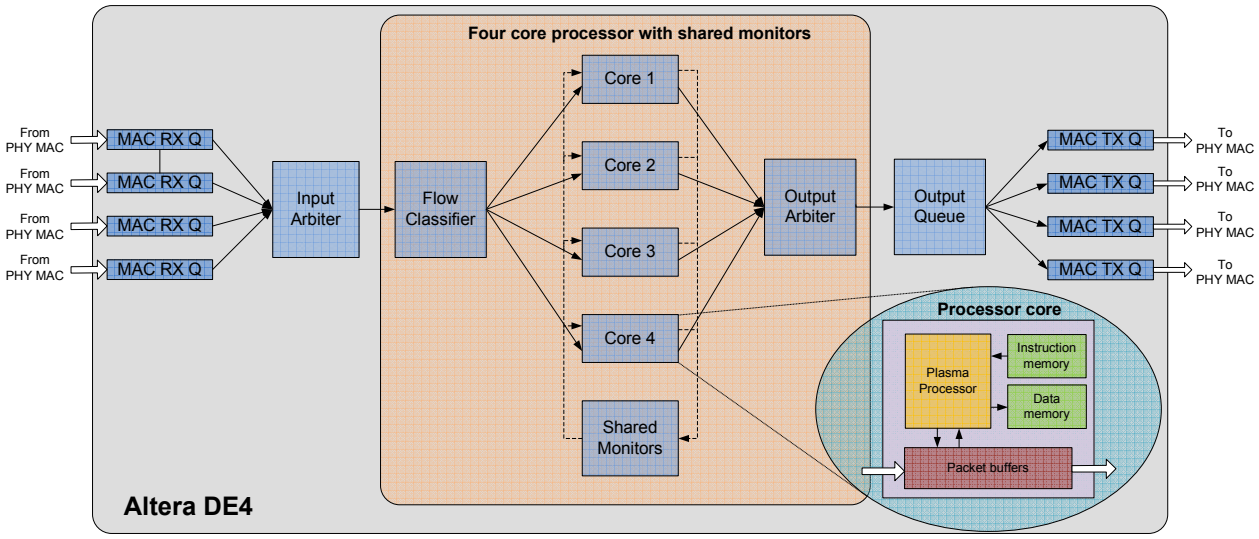


Figure 6.8. System overview for multi-core secure packet processing system with reconfigurable monitors

6.4.2 System Details

In this section, we briefly describe an FPGA-based monitoring system for multi-core network processors. The static instruction sequence used by a processor core is converted into a state machine that is implemented in FPGA logic. These monitors require minimal resource usage, can be configured on a per-application basis, and can be shared across multiple network processor cores performing the same packet processing operations.

A high-level overview of the implemented multi-core network processor system including monitors is shown in Figure 6.8. In this setup, four soft processor cores used for packet processing are implemented in a Stratix IV GX230 located on an Altera DE4 board. The router infrastructure surrounding the cores and the monitoring hardware are taken from the NetFPGA reference router [1] which has been migrated to the Stratix IV family. The DE4 board consists of four 1 Gbps Ethernet interfaces, a PCI Express interface, up to 8 GB of DDR RAM and 2 MB of SRAM. The hardware data path of the modified router is implemented as a pipeline of customizable modules.

In this implementation, each security monitor operates as an entity that is isolated from the processor cores. The FPGA logic and memory associated with the security monitors is not visible to the processor cores and an attack on a processor core will not impact the operation of its associated monitor. Our per-instruction monitoring approach requires an analysis of the program instruction flow immediately following compilation. As part of this analysis, both conditional and non-conditional instructions are examined. For non-conditional statements, instructions are assumed to execute from instruction memory in sequence. For branches, it is expected that either the following instruction in memory or the branch target is followed. A compact state machine is then constructed from this instruction analysis such that each instruction represents a state. During execution, the state machine progresses through states as instructions are executed. If the run-time execution instruction sequence deviates from the expected sequence for either branches or non-branches, a security error is detected and a reset signal is sent to the processor. This reset signal causes the current packet to be dropped, the processor registers to be reset, and processing is then restarted with the next assigned packet.

6.4.3 Comparison to Fixed Hardware Monitor

The previous implementation of a network processor security monitor [8] was designed for network processors which do not include reconfigurable hardware or monitor sharing. This monitor does not allow for any changes to the monitor logic functionality. The new programmable logic-based monitor idea has the following advantages:

1. The amount of information stored in block memory for the programmable logic is limited to small (e.g. four-bit) instruction hash values. This greatly reduces memory consumption versus the storage of branch target addresses and basic block numbers.

2. Jump logic for the programmable logic approach is fashioned from programmable logic rather than requiring extra block memory storage and a second block memory

lookup. This feature reduces the pipeline length of the new approach, allowing for a faster identification of an instruction flow error.

3. The programmable logic allows for customized connections between processor cores and needed monitors. Monitor sharing is easily accomplished by pipelining. For fixed architecture monitors, one monitor is dedicated per processor core, whether it is needed or not.

6.4.4 Experimental Results

The four-core network processor including shared monitors was successfully implemented on an Altera DE4 platform. The four-core system under monitoring was first evaluated for standard IPv4 packet forwarding for a series of packet sizes. A single 1 Gbps MAC port was used for packet receive and a separate port was used for transmit. For this experiment, no malicious packets were sent. A single monitor was shared by all four cores. As shown in Figure 6.9, the throughput of the four-core network processor improves as the packet size increases, an expected result. The experiment was also performed in hardware without the use of monitors. The results indicate that in the absence of malicious packets, the throughput graph is the same for both monitored and un-monitored packet flows. This result is not surprising since per-packet processing requires hundreds of cycles while monitoring, which occurs in parallel to processing, can achieve a throughput of one cycle per instruction.

A comparison of the previous fixed-hardware monitor architecture [8] and the new programmable-logic based approach in reconfigurable hardware appears in Table 6.3. Total instructions and control flow instructions (branches) are enumerated along with the resources required to build the monitors. Monitors for both standard packet forwarding (IPv4) and the CM approach are included. Programmable-logic monitors for four-core implementations were shared while individual fixed logic monitors were needed for four-core

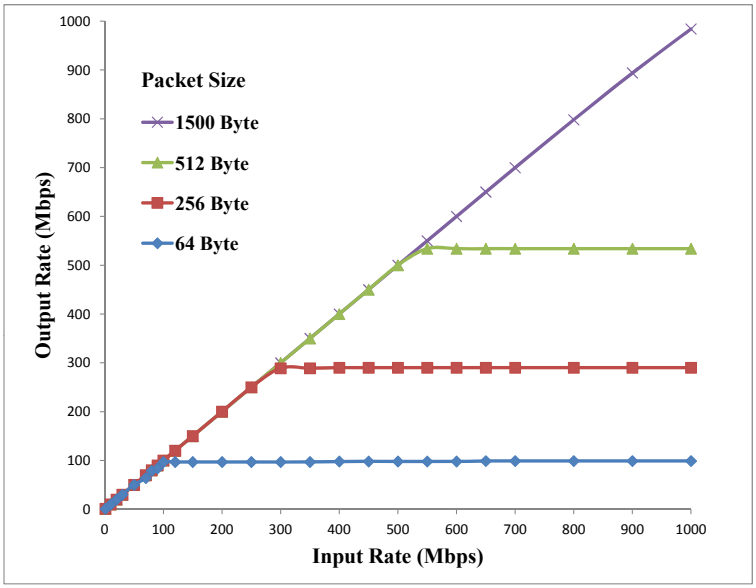
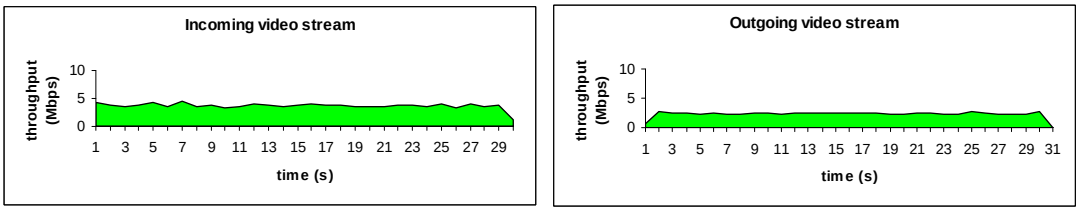


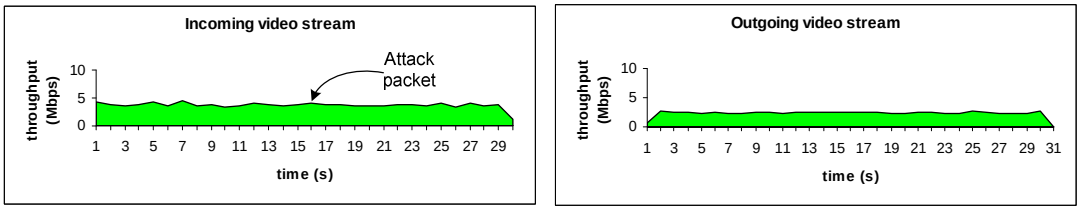
Figure 6.9. Four core network processor performance including monitoring for different packet sizes

Table 6.3. Resource utilization of network processor benchmarks

	Instructions		Fixed Hardware			Programmable Logic		
Benchmark	**Total**	**Control Flow**	**ALUTs**	**Regs**	**Mem.Bits**	**ALUTs**	**Regs**	**Mem.Bits**
IPv4 (1-core)	327	17	40	34	6272	313	159	2048
CM (1-core)	289	21	40	34	6272	329	167	2048
IPv4 (4-core)	327	17	160	136	25088	863	486	2048
CM (4-core)	289	21	160	136	25088	974	529	2048



(a) Benign network traffic



(b) Benign traffic and single attack packet

Figure 6.10. Traffic Rates at Input Port and Output Port of Router with Processing Monitor.

implementations. It is evident that the new approach saves considerable block memory resources in addition to allowing for resource sharing between the monitors.

6.5 Demonstration of Real Attack Detection

Lets see now how our system responds under a real attack scenario, like the one described in Section 5. We have experimented on the same custom processor used for the setup shown in Figure 5.5. The prototype successfully detects the example attack (and any other attack that changes the control flow), halts the processor, drops the packet, and restores the system within 6 instruction cycles. This very small time for recovery allows our secure packet processor to operate at full data rate even when under attack. The overhead

for adding a monitoring system to the packet processor is very small (0.8% increase on slice LUTs and 5.6% on memory elements).

Figure 6.10 shows the operation of the secure packet processor under attack. As can be seen, not only does the processor not fall victim to the attack, but it also continues to forward regular traffic without interruption.

While the results from our secure packet processor are encouraging by demonstrating that there are defenses against the types of attacks that we describe in this dissertation, it is important to note that such defenses are not currently deployed in the Internet. Existing software-based routers are still vulnerable and more research and development is necessary to design and deploy defenses against this novel type of attack.

6.6 Inferring Packet Processing Behavior using I/O monitors

We have so far assumed that packet processing is a monolithic processing operation. This assumption, however, does not always apply in practice. Due to the complexity of different packet processing tasks implemented on a router, modularity in processing is a necessity. Several approaches to modularity in packet processing have been proposed [23]. Some functions performed by different elements could be random early detection, flow classification, IP look-up, queuing or even drop-packet. In this section, we describe an idea that can support security in such a modular router.

The monitoring approach we have developed previously lends itself well for modular implementations as we show below. Nevertheless, the use of independent monitors for different processing modules poses a new type of security challenge: How can the correct overall operation of a router *across modules* be ensured? In particular, an attacker may change the operation of a router such that individual module monitors cannot detect an attack. Therefore, it is necessary to have a comprehensive defense mechanism that can protect modular router implementations from data path attacks.

6.6.1 Verification in the Data Path

Here, we present a modular technique for validation of a router's correct operation by monitoring the "processing" of each module, examining input/output flow characteristics and correlating them with the processing time spent on individual modules [13].

6.6.1.1 Protection Mechanisms

In previous sections we have demonstrated the feasibility and effects of data plane attacks in a real network setting. One way to counter this type of attacks is to use processing monitors which track the operations on the network processor, as described in [8]. The monitor can determine if attacks occur because the processor's operations deviate from the operations that are valid - as determined by offline analysis of the processing binary. In that work, we effectively monitor the program execution as a whole.

However, due to vulnerabilities in the data path, we can expect attacks at the protocol level as well. We can have a situation where valid processing instructions are executed on the network processor, but still the overall router behavior is abnormal. For example, in the case of intentionally misleading routing information advertisement in RIP, a large amount of multi-cast packets could be directed to the wrong router interfaces. While observing the system's operation as a whole, we would detect incorrect packet flow behavior even though monitoring the "processing" of the system would determine that the program execution in an instruction-per-instruction basis is correct.

Overall, the processing monitor approach works well, but in terms of packet flow validation it is difficult to say anything detailed about the behavior of a complex piece of code. It can only be done at a level where rules are very lose. This problem is closely related to the lack of modularity in the processing monitor. Therefore, we propose a *modular* approach for verifying data-path processing.

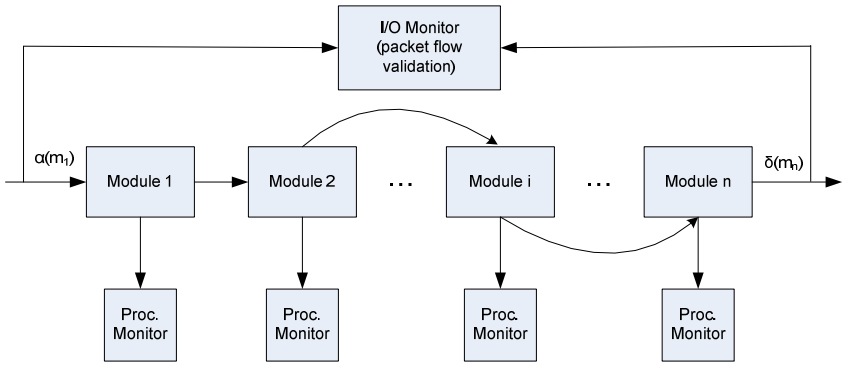


Figure 6.11. Modular Processing Verification

6.6.1.2 Modular Verification

We develop a scalable solution which uses both a processing monitor and an I/O monitor per module. We refer to Click as our example router architecture, because the system is already divided in individual packet processing modules (elements). Different elements implement simple router functions, which can be combined to one graph and build a complete and extendable router configuration. The power of this approach lies in the "user's" ability to combine modules into arbitrary graphs where packets proceed along one path through the graph.

In order to verify the correct operation of the system as a whole we need to : 1) verify the execution path of every element by using a processing monitor per element [8] which tracks all the individual instructions executed on the processor while the particular element is active and 2) use one I/O monitor that tracks the processing times of all packets on all elements and uses a mathematical model to judge if the overall processing in the system is correct. The two types of monitors are shown in Figure 6.11.

The high level idea regarding the operation of the I/O monitor is the following:

- We profile the processing time that each packet spends in each element
- We draw one histogram per element that depicts the distribution of the processing time values

- We normalize the histogram to plot the probability density function (pdf) of processing times per element
- We find a mathematical way to combine the pdfs of individual elements and create a model of the expected processing time distribution of the system as a whole (distribution Xconv)
- The next step is to profile the processing time that each time spends across all elements and create the probability density function of overall processing times (distribution Xall)
- Finally, we compare Xconv and Xall, and use a statistical metric to decide whether they are matching or not

The interesting part of the I/O monitor is that we base our decision of what normal processing should look like on the unique processing characteristics of every element. The results from profiling the element's processing times will solely depend on the functionality of the element and will thus help us determine if the system encounters any unusual delays while processing the packets.

6.6.1.3 Port-to-Port Packet Flow

In this section, we describe the mathematical basis of the I/O monitor in a more formal way.

Let's assume that we have n processing elements $m_1 \dots m_n$. The overall functionality of the router is represented by a graph connecting these elements. Graph $G = (M, E)$ consists of a set of all elements, M, and a set of directed edges, E, indicating that transition of traffic between elements is allowed. For simplicity of our discussion, we assume that all packets enter processing at node m_1 and leave processing at node m_n. Let $\alpha(m_i)$ denote the arrival rate of traffic a module m_i. The departure rate at module m_i is denoted by $\delta(m_i)$.

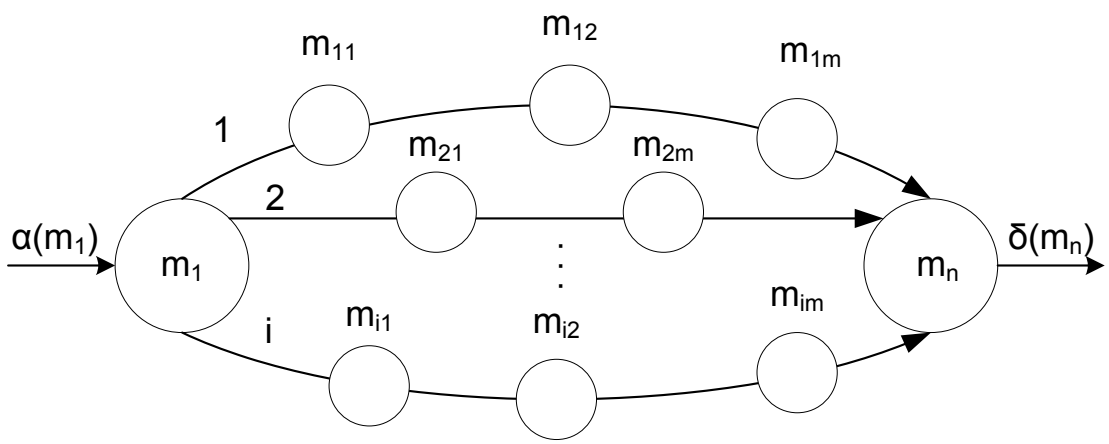


Figure 6.12. Validating Packet Flow on Node m.

To determine the characteristics of traffic at a node m_i, we determine all paths that can lead from the input to that node. Let $p(m_{i_1}, m_{i_2}, \ldots, m_{i_m})$ denote a path from m_{i_1} to m_{i_m} . Figure 6.12 shows an example network configuration and represents the introduced notation more clearly.

Let P_{im} be the probability density function (pdf) of the processing time spent on node m_{im}. The total processing time distribution on every path to node m_i can be computed by convolving the processing time distributions on all individual nodes in that particular path: $P_i = P_{i1} * P_{i2} * \cdots * P_{im}$. If path i receives an f_i percentage of the total traffic rate then the pdf on that path is $\sum_{i=1}^{m} f_i \cdot P_i$. Taking the weighted sum of pdfs over all paths, we can describe the expected processing time distribution of the system as a whole using the following equation:

$$P_{conv} = \sum_{i=1}^{m} f_i \cdot P_i \tag{6.1}$$

If P_{all} denotes the pdf of the overall processing we can compare P_{all} to P_{conv} and infer abnormal processing behavior at any element in the system. If an attacker targets one of the elements and changes its processing, the change will reflect in the distribution P_{all}, which will then become different from the modeled distribution P_{conv} and eventually the I/O monitor will signal an attack. As we explain in Section 6.6.3, we can measure

the difference between two probability distributions (and thus quantify a potential attack) adopting the Kullback-Leibler divergence method as used in probability theory.

6.6.2 Illustration of Processing Time Distributions

In this section, we show an example of how to combine individual modules into graphs and estimate the overall processing distribution function based on the convolution model (joint work with Xinming Chen and Tilman Wolf). This example serves as proof that our I/O monitor can validate processing correctly in case of normal processing conditions.

We use a test configuration on a Click modular router which consists of a single path of seven modules: DropBroadcasts - Strip - CheckIPHeader - FixIPSrc - DecIPTTL - IPFragmenter - EtherEncap; These are randomly chosen simple functions of Click modular router that either perform some kind of processing on the packets headers (e.g. modify the IPaddr, TTL fields), delete some bytes from each packet, fragment large packets or drop certain types of packets. We send a random packet trace to the router and profile the processing time each packet spends in each of the above elements. Figure 6.13 shows the frequency distributions of the seven elements. The horizontal axis denotes processing time (in CPU cycles), while the vertical axis is normalized frequency.

The overall processing time distribution on the seven elements is shown in Figure 6.14. The next step is to verify our mathematical model, so we compute the convolution of the seven separate distributions (Figure 6.14 as well) and compare it to the total distribution. As we can see, the two plots are very similar, which means we have a good mathematical model that estimates the overall processing time distribution of a system.

6.6.3 Quantifying Changes in Processing

In the previous section we verified the operation of our I/O monitor in case all seven elements do the "right" processing. Here, we intentionally change the behavior of the system to create attack conditions and see how our I/O monitor works under attack. The expected processing time distribution (computed using convolution) stays of course unchanged, but

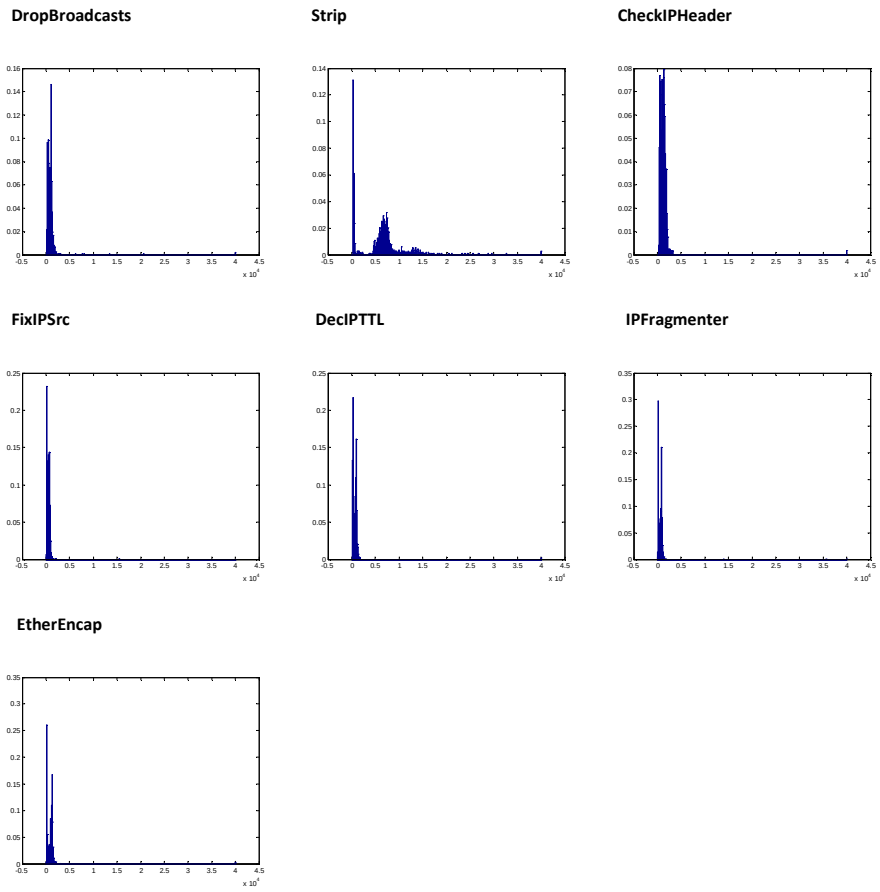


Figure 6.13. Processing Time Distributions for Elements 1-7

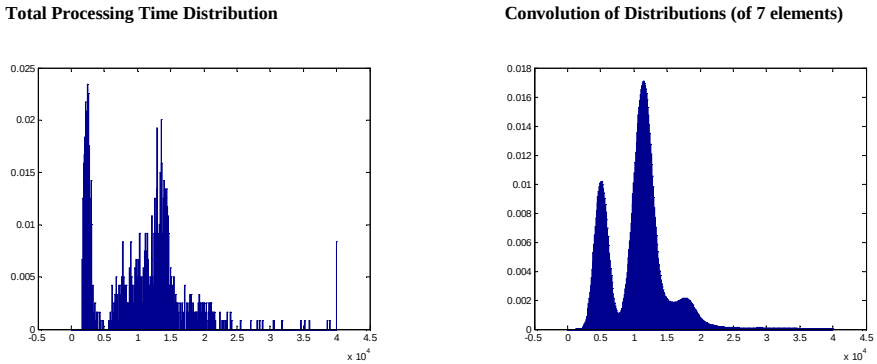


Figure 6.14. Total Processing Time Distribution vs. Convoluted Distributions of 7 elements

the overall processing time distribution (which is profiled) will change in case of an attack. Let's see how distributions change when we change one element's processing behavior (imitating possible attack behavior) and how we can quantify that change.

In this experiment, we use the same Click configuration: DropBroadcasts - Strip - CheckIPHeader - FixIPSrc - DecIPTTL - IPFragmenter - EtherEncap, and a different random packet trace. We modify the element DecIPTTL, by adding a variable number of loops in the source code. In this way, we create various attack conditions ranging from mild change in the processing to more aggressive changes; we experiment with zero to as high as 1,000,000 loops of instructions.

As we mentioned before, a change in the processing time distribution (pdf) of the 5th element will impact the overall processing time distribution plot of the system. The first plot of Figure 6.15 shows the overall processing time distribution of the Click configuration (which is profiled) in case no additional code is injected in the 5th element. We will take that as our reference of "normal" processing behavior. The rest of the plots in Figure 6.15 show the overall processing time distributions in case we inject 100, 1000, 10,000 and 1,000,000 loops.

A careful visual comparison can show us some changes in processing. We can actually identify differences slightly better if we plot the overall pdfs of the system as cdfs, as shown in Figure 6.16. It is obvious that 1,000,000 injected loops of code can be easily detected, but it becomes more difficult to detect 10,000-loop injection or less.

A formal way to quantify the changes in the overall processing distribution plots is to measure the Kullback-Leibler divergence [44]. KL divergence is an approach that quantifies in bits how close a probability distribution P is to a model distribution Q. Since our distributions are non-symmetric the KL distance of P based on Q will be different than the distance of Q based on P. Thus, we calculate the distance in both left-right and right-left directions: $DKL(P||Q) + DKL(Q||P)$. This metric can help us detect changes in processing between the normal case and the abnormal cases depicted in Figure 6.15. Table

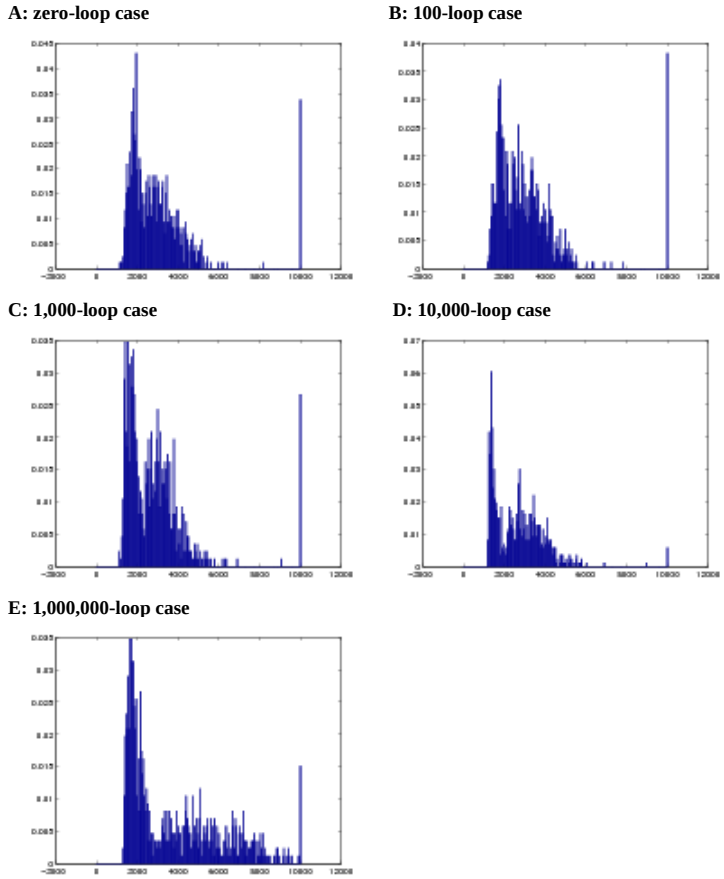


Figure 6.15. Processing Time Distributions for Normal/Attack Cases

6.4 shows these distances and their standard error. The same distances are also plotted in Figure 6.16. As expected, the more loops we insert in case A, the higher the KL-divergence metric between the initial and the new processing time distributions. The way to interpret these distances is by performing multiple experiments of normal and abnormal processing cases and determining the threshold above which the two distributions will be considered different and below which the difference will be considered small enough to fall into the normal processing category. For example, in Figure 6.16 the cut-off threshold could be 0.5. Anytime we compare a probability density function to the modeled one and they differ by more than 0.5, we will assume that the system is under attack.

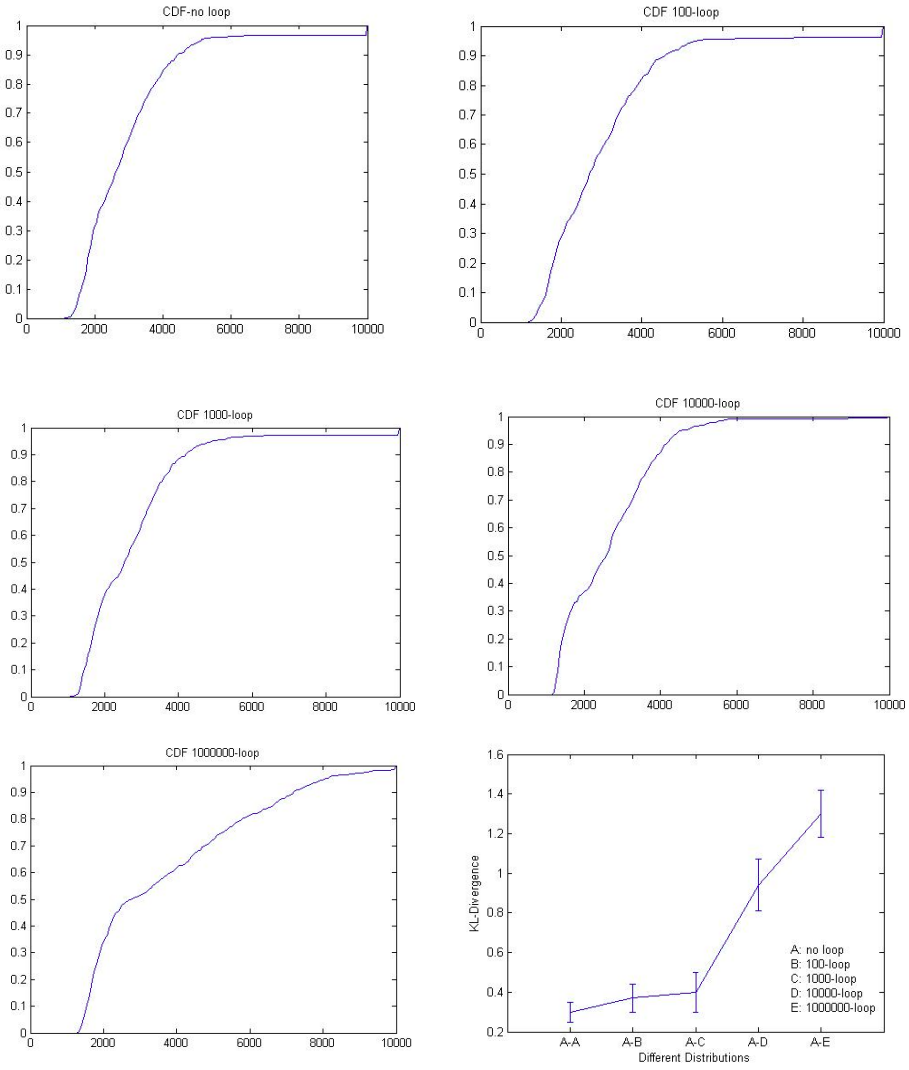


Figure 6.16. CDFs of Different Distributions and KL-distance

Table 6.4. KL-Divergence of different processing scenarios

Attack vs Normal Processing	Mean KLDiv	Std KLDiv
No-loop vs No-loop (reference)	0.30	0.05
100-loop vs No-loop	0.37	0.07
1000-loop vs No-loop	0.40	0.06
10000-loop vs No-loop	0.94	0.09
1M-loop vs No-loop	1.30	0.12

In the general case of a modular router that runs protocol code, we can have one extra module calculate the KL-divergence metric to detect changes in the processing distribution. Of course, every case is different, and we should carefully set the distance threshold that will be the cut-off for normal processing. The complexity of the KL-divergence metric is O(n), where n is the number of samples in the probability density function. This observation makes us think that the implementation of the I/O monitor will not be prohibitive in terms of performance overhead and that it is suitable for implementation in hardware, since it can be parallelized.

6.6.4 I/O Monitor Complexity

In this section, we analyze the complexity of such a system. The I/O monitor pseudo-code shown in Figure 6.17 performs the following steps:

- First of all, assumes a system with m modules and n samples of processing times
- Takes as input m arrays consisting of n profiled processing times, and the total processing time distribution the the system as a whole (Q)
- Calculates the histogram of processing times per module (O(n+k) complexity per module)
- Normalizes the histogram to produce the probability density function of every module (O(n)complexity per module)
- Computes the convolution of the pdfs of the m modules (O(n*m) complexity overall)
- Compares the convoluted distribution to the given distribution Q, to check of processing abnormalities
- The complexity of the KL-divergence metric used in the previous comparison is O(n)

Overall, we observe that we have a nested set of loops, one with m and one with n repetitions. This gives a total complexity of O(m*n) for the I/O monitor. We can roughly

Pseudocode for I/O monitor

```
Assume m modules, n samples
Input : mxn array of profiled processing times : times[m, n], overall system pdf Q
Output : Distance between convoluted processing times and distribution Q

    result = 1;
    for i = 1 to m do                          //repeat convolution m times
        count = histogram(times[i]);           //produce histogram (complexity is O(n + k), k = key for sorting alg.)
        sum = 0;
        for r = 1 to n do
            sum = sum + count[r];
        end for
        for k = 1 to n do
            frequency[k] = count[k]/sum;       //normalize histogram
        end for
        for f = 1 to 2n - 1 do
            result[f] = Σ_j P[j]frequency[f - j + 1];   //convolution computation
        end for
        for h = 1 to n do
            P[n] = result[n];
        end for
    end for
    distance = 0;
    for i = 1 to n do
    distance = distance + P[i]ln(P[i]/Q[i]) + Q[i]ln(Q[i]/P[i])   //KL - divergence computation
    end for

function histogram
input : collection of n items, each of which has a non - negative integer key whose maximum value is at most k
output : array of n items, in order by their keys
code :
allocate an array Count[0..k] ; initialize each array cell to zero ;
for each input item x :
   Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k :
   c = Count[i]
   Count[i] = total
   total = total + c

allocate an output array Output[0..n - 1] ;
for each input item x :
   store x in Output[Count[key(x)]]
   Count[key(x)] = Count[key(x)] + 1
return Output
```

Figure 6.17. Pseudo-code for I/O Monitor

estimate around 50 instructions being executed inside the large loop. Assuming a typical system of m=20 modules and taking n=10,000 samples of processing times per module, our system will execute around 10M instructions. With a 100MHz processor, we can get updates from the I/O monitor around every 100ms.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this chapter, we describe the conclusions of this dissertation and propose future directions for our work.

7.1 Conclusions

The use of software-based packet processing in routers with general-purpose processing engines is becoming more prevalent in the Internet. The use of software in the data plane of the network presents a target for novel intrusion and denial-of-service attacks that can have significant impact on the overall security of the network.

In our work, we have described and demonstrated a novel type of network attack, which exploits vulnerabilities in the packet processing systems of modern routers. We show how integer vulnerabilities in the implementation of a common protocol processing operation can be used to execute arbitrary attack code. Our attack can be used to launch devastating denial-of-service attacks in the core of the network.

We have also shown that defense mechanisms do exist. Our work presents the design and prototype implementation of a secure packet processor that is equipped with a monitoring system that can detect such attacks. The monitoring system compares the operation of the processor cores to the expected behavior that is obtained from analyzing the packet processing binary. A processing monitor continuously checks the validity of processor operations and triggers a recovery mechanism when deviations from expected behavior are detected. The prototype implementation of our system can detect intrusion attacks within

six processor cycles and recover the system in that time. The result from the prototype system indicate that our design is an effective approach to protecting networking infrastructure in the future Internet.

However, such defense methods are not currently deployed in the network. To our knowledge, this work represents the first time a practical attack on the data plane of the actual network infrastructure has been shown and thus provides an important step toward understanding and correcting vulnerabilities in the network infrastructure.

7.2 Future Work

The new attack vector that was identified in this thesis, and the defense methods that were proposed, are the beginning steps of our contribution to the field. This dissertation opens up interesting questions in both network and embedded system security fields.

7.2.1 Networked Embedded Systems Security

Packet processors inside modern routers are just one example of networked embedded devices. The vulnerabilities that routers are subject to can be applied to other areas where programmable networked devices are used. For example, the design of systems in wireless communications, health care or the power grid is shifting towards programmability because of flexibility needs.

Figure 6.12 shows different domains where the use of networked embedded devices is prevalent. Since these devices already communicate as large-scale cyber-physical systems, we believe that there is a lot of scope in trying to identify potential vulnerabilities that can affect their operation.

Critical characteristics of such embedded devices are the following:

- They form networks
- They can be attacked remotely

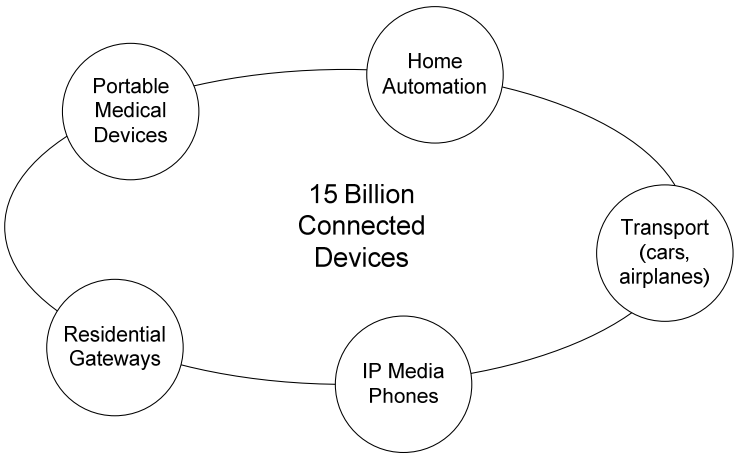


Figure 7.1. Networked Embedded Devices

- They are resource constrained

Judging from the characteristics of routing systems and our evidence of vulnerabilities in the networking domain, we can expect devices in neighboring domains to become attack targets. Exploring hardware/software co-design security solutions is essential for interconnected embedded devices. The problem domain needs defenses that are light-weight, fast, effective and can take advantage of parallelization (to be implemented in hardware), while other type of attacks can be addressed using software based security mechanisms. In this way, we can take advantage of the best of the two worlds - computing efficiency and high throughput.

Some of the questions that are open in the greater area of cyber-physical systems are: How can we design cyber-physical systems in a way that they are inherently secure? Or, in case inherent security is not possible, what kind of security extensions will make them secure?

BIBLIOGRAPHY

[1] NetFPGA. `http://www.netfpga.org/`.

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

[3] 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.

[4] Balakrishnan, Hari, Rahul, Hariharan S., and Seshan, Srinivasan. An integrated congestion management architecture for internet hosts. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM)* (Cambridge, MA, Sept. 1999), pp. 175–187.

[5] Ballani, Hitesh, Chawathe, Yatin, Ratnasamy, Sylvia, Roscoe, Timothy, and Shenker, Scott. Off by default! In *Proc. of Fourth Workshop on Hot Topics in Networks (HotNets-IV)* (College Park, MD, Nov. 2005).

[6] Bavier, Andy, Feamster, Nick, Huang, Mark, Peterson, Larry, and Rexford, Jennifer. In VINI veritas: realistic and controlled network experimentation. In *SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pisa, Italy, Aug. 2006), pp. 3–14.

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

[8] 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).

[9] Chasaki, Danai, Wu, Qiang, and Wolf, Tilman. Attacks on network infrastructure. In *Proc. of Twentieth IEEE International Conference on Computer Communications and Networks (ICCCN)* (Maui, HI, Aug. 2011).

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

[11] Clark, David D. The design philosophy of the DARPA Internet protocols. In *Proc. of ACM SIGCOMM 88* (Stanford, CA, Aug. 1988), pp. 106–114.

[12] 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.

[13] Danai Chasaki, Qiang Wu, and Wolf, Tilman. Inferring packet processing behavior using input/output monitors. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)* (Brooklyn, NY, Oct. 2011).

[14] 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).

[15] Estevez-Tapiador, Juan M., Garcia-Teodoro, Pedro, and Diaz-Verdejo, Jesus E. Anomaly detection methods in wired networks: a survey and taxonomy. *Computer Communications 27*, 16 (Oct. 2004), 1569–1584.

[16] EZchip Technologies Ltd. *NP-3 – 30-Gigabit Network Processor with Integrated Traffic Management*. Yokneam, Israel, May 2007. `http://www.ezchip.com/`.

[17] Feldmann, Anja. Internet clean-slate design: what and why? *SIGCOMM Computer Communication Review 37*, 3 (July 2007), 59–64.

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

[19] 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.

[20] IDC, Inc. *The Internet Reaches Late Adolescence*, 2009.

[21] Intel Corporation. *Intel Second Generation Network Processor*, 2005. `http://www.intel.com/design/network/products/npfamily/`.

[22] Kent, S., and Atkinson, R. Security architecture for the Internet protocol. RFC 2401, Network Working Group, Nov. 1998.

[23] 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.

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

[25] Liao, Yong, Yin, Dong, and Gao, Lixin. PdP: parallelizing data plane in virtual network substrate. In *Proc. of the First ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures (VISA)* (Barcelona, Spain, Aug. 2009), pp. 9–18.

[26] 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 *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education* (San Diego, CA, June 2007), pp. 160–161.

[27] LSI Corporation. *APP3300 Family of Advanced Communication Processors*, Aug. 2007. `http://www.lsi.com/`.

[28] Mao, Shufu, and Wolf, Tilman. Hardware support for secure processing in embedded systems. In *Proc. of 44th Design Automation Conference (DAC)* (San Diego, CA, June 2007), pp. 483–488.

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

[30] Matrosov, Aleksandr, and Rodionov, Eugene. *Stuxnet Under the Microscope*, 2011.

[31] Merritt, Rick. Intel packs Atom into tablets, set-tops, FPGAs. *EE Times* (2010).

[32] Mogul, Jeffrey C. Simple and flexible datagram access controls for UNIX-based gateways. In *USENIX Conference Proceedings* (Baltimore, MD, June 1989), pp. 203–221.

[33] Moore, David, Shannon, Colleen, and Brown, Jeffery. Code-Red: a case study on the spread and victims of an Internet worm. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement* (Marseille, France, Nov. 2002), pp. 273–284.

[34] Nakka, Nithin, Kalbarczyk, Zbigniew, Iyer, Ravishankar K., and Xu, Jun. An architectural framework for providing reliability and security support. In *Proc. of the 2004 International Conference on Dependable Systems and Networks (DSN)* (Florence, Italy, June 2004), pp. 585–594.

[35] National Science Foundation. *Future INternet Design*. `http://www.nets-find.net/`.

[36] National Science Foundation. *Global Environment for Network Innovation*. `http://www.geni.net/`.

[37] Parameswaran, Sri, and Wolf, Tilman. Embedded systems security – an overview. *Design Automation for Embedded Systems 12*, 3 (Sept. 2008), 173–183.

[38] Parno, Bryan, Wendlandt, Dan, Shi, Elaine, Perrig, Adrian, Maggs, Bruce, and Hu, Yih-Chun. Portcullis: protecting connection setup from denial-of-capability attacks. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications* (Kyoto, Japan, Aug. 2007), pp. 289–300.

[39] Ragel, Roshan G., and Parameswaran, Sri. IMPRES: integrated monitoring for processor reliability and security. In *Proc. of the 43rd Annual Conference on Design Automation (DAC)* (San Francisco, CA, USA, July 2006), pp. 502–505.

[40] 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.

[41] Ramaswamy, Ramaswamy, Weng, Ning, and Wolf, Tilman. Analysis of network processing workloads. *Journal of Systems Architecture 55*, 10 (Oct. 2009), 421–433.

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

[43] 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).

[44] S., Kullback, and R.A., Leibler. *On information and sufficiency*, 1951.

[45] Savage, Stefan, Wetherall, David, Karlin, Anna, and Anderson, Tom. Network support for IP traceback. *IEEE/ACM Transactions on Networking 9*, 3 (June 2001), 226–237.

[46] Seacord, Robert C. *Secure Coding in C and C++*, 1st ed. Addison-Wesley Professional, 2005.

[47] Snort. *The Open Source Network Intrusion Detection System*, 2004. `http://www.snort.org`.

[48] Stallings, William. *Cryptography and Network Security*, 4th ed. Prentice Hall, 2006.

[49] Turner, Jonathan S., Crowley, Patrick, DeHart, John, Freestone, Amy, Heller, Brandon, Kuhns, Fred, Kumar, Sailesh, Lockwood, John, Lu, Jing, Wilson, Michael, Wiseman, Charles, and Zar, David. Supercharging PlanetLab: a high performance, multi-application, overlay network platform. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications* (Kyoto, Japan, Aug. 2007), pp. 85–96.

[50] Turner, Jonathan S., and Taylor, David E. Diversifying the Internet. In *Proc. of IEEE Global Communications Conference (GLOBECOM)* (Saint Louis, MO, Nov. 2005), vol. 2.

[51] Wilton, Steve, Ho, C, Leong, P., Luk, W, and Quinton, B. A synthesizable datapath-oriented embedded fpga fabric. In *Proc. of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Feb. 2007).

[52] Wiseman, Charlie, Turner, Jonathan, Becchi, Michela, Crowley, Patrick, DeHart, John, Haitjema, Mart, James, Shakir, Kuhns, Fred, Lu, Jing, Parwatikar, Jyoti, Patney, Ritun, Wilson, Michael, Wong, Ken, and Zar, David. A remotely accessible network processor-based router for network experimentation. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)* (San Jose, CA, Nov. 2008), pp. 20–29.

[53] Wolf, Tilman. Challenges and applications for network-processor-based programmable routers. In *Proc. of IEEE Sarnoff Symposium* (Princeton, NJ, Mar. 2006).

[54] Wolf, Tilman. Data path credentials for high-performance capabilities-based networks. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)* (San Jose, CA, Nov. 2008), pp. 129–130.

[55] Wolf, Tilman, and Tessier, Russell. Design of a secure router system for next-generation networks. In *Proc. of Third International Conference on Network and System Security (NSS)* (Gold Coast, Australia, Oct. 2009).

[56] Wolf, Tilman, Tessier, Russell, and Prabhu, Gayatri. Securing the data path of next-generation router systems. *Computer Communications 34*, 4 (Apr. 2011), 598–606.

[57] Wu, Qiang, Chasaki, Danai, and Wolf, Tilman. Implementation of a simplified network processor. In *Proc. of IEEE International Conference on High Performance Switching and Routing (HPSR)* (Richardson, TX, June 2010).

[58] Wu, Qiang, and Wolf, Tilman. On runtime management in multi-core packet processing systems. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)* (San Jose, CA, Nov. 2008), pp. 69–78.

[59] Yin, Dong, Unnikrishnan, Deepak, Liao, Yong, Gao, Lixin, and Tessier, Russell. Customizing virtual networks with partial fpga reconfiguration. *SIGCOMM Computer Communication Review 41* (Jan. 2011), 125–132.

[60] Zambreno, Joseph, Choudhary, Alok, Simha, Rahul, Narahari, Bhagi, and Memon, Nasir. SAFE-OPS: An approach to embedded software security. *Transactions on Embedded Computing Sys. 4*, 1 (Feb. 2005), 189–210.