2013

Protecting Network Processors with High Performance Logic Based Monitors

Harikrishnan Kumarapillai Chandrikakutty
University of Massachusetts - Amherst, chandrikakut@ecs.umass.edu

PROTECTING NETWORK PROCESSORS WITH HIGH PERFORMANCE LOGIC BASED MONITORS

A Thesis Presented

by

HARIKRISHNAN KUMARAPILLAI CHANDRIKAKUTTY

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

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

May 2013

ELECTRICAL AND COMPUTER ENGINEERING

PROTECTING NETWORK PROCESSORS WITH HIGH PERFORMANCE LOGIC BASED MONITORS

A Thesis Presented

by

HARIKRISHNAN KUMARAPILLAI CHANDRIKAKUTTY

Approved as to style and content by:

ACKNOWLEDGMENTS

I would like to thank the National Science Foundation for funding the project and giving me the opportunity to work on it. I am extremely thankful to Professor Russell Tessier for his constant motivation and guidance. I am also grateful to Professor Tilman Wolf and Professor Michael Zink for agreeing to be on my thesis committee. I would like to extend my gratitude to Deepak Unnikrishnan of Reconfigurable Computing Group for his valuable suggestions and support on the project. I would also like to thank my RCG lab mates-Murtaza, Justin, Kekai, Cory and Jia for making it a wonderful experience**.** Finally, I would like to thank my family for their encouragement and support during the course of my higher studies here in the United States.

ABSTRACT

PROTECTING NETWORK PROCESSORS WITH HIGH PERFORMANCE LOGIC BASED MONITORS

MAY 2013

HARIKRISHNAN KUMARAPILLAI CHANDRIKAKUTTY

B.Tech, COLLEGE OF ENGINEERING, TRIVANDRUM, INDIA

M.S. E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell G. Tessier

Technological advancements have transformed the way people interact with the world. The Internet now forms a critical infrastructure that links different aspects of our life like personal communication, business transactions, social networking, and advertising. In order to cater to this ever increasing communication overhead there has been a fundamental shift in the network infrastructure. Modern network routers often employ software programmable network processors instead of ASIC-based technology for higher throughput performance and adaptability to changing resource requirements. This programmability makes networking infrastructure vulnerable to new class of network attacks by compromising the software on network processors. This issue has resulted in the need for security systems which can monitor the behavior of network processors at run time.

This thesis describes an FPGA-based security monitoring system for multi-core network processors. The implemented security monitor improves upon previous hardware monitoring schemes. We demonstrate a state machine based hardware programmable monitor which can track program execution flow at run time. Applications are analyzed offline and a hash of the instructions is generated to form a state machine sequence. If the state machine deviates from expected behavior, an error flag is raised, forcing a network processor reset.

For testing purposes, the monitoring logic along with the multi-core network processor system is implemented in FPGA logic. In this research, we modify the network processor memory architecture to improve security monitor functionality. The efficiency of this approach is validated using a diverse set of network benchmarks. Experiments are performed on the prototype system using known network attacks to test the performance of the monitoring subsystem. Experimental results demonstrate that out security monitor approach provides an efficient monitoring system in detecting and recovering from network attacks with minimum overhead while maintaining line rate packet forwarding. Additionally, our monitor is capable of defending against attacks on processor with a Harvard architecture, the dominant contemporary network processor organization. We demonstrate that our monitor architecture provides no network slowdown in the absence of an attack and provides the capability to drop packets without otherwise affecting regular network traffic when an attack occurs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Communication is an essential aspect of modern human society. Rapid advancements in modern technology have brought about significant improvements in fields of personal communication, business transactions, entertainment, and digital government. The Internet forms a central aspect in many of these communication requirements. This ever growing dependence on the Internet has resulted in the need to improve different attributes of communication infrastructure like network functionality, throughput performance, reliability and security. The growth of Internet usage based on data from the International Telecommunications Union is illustrated in Figure 1.

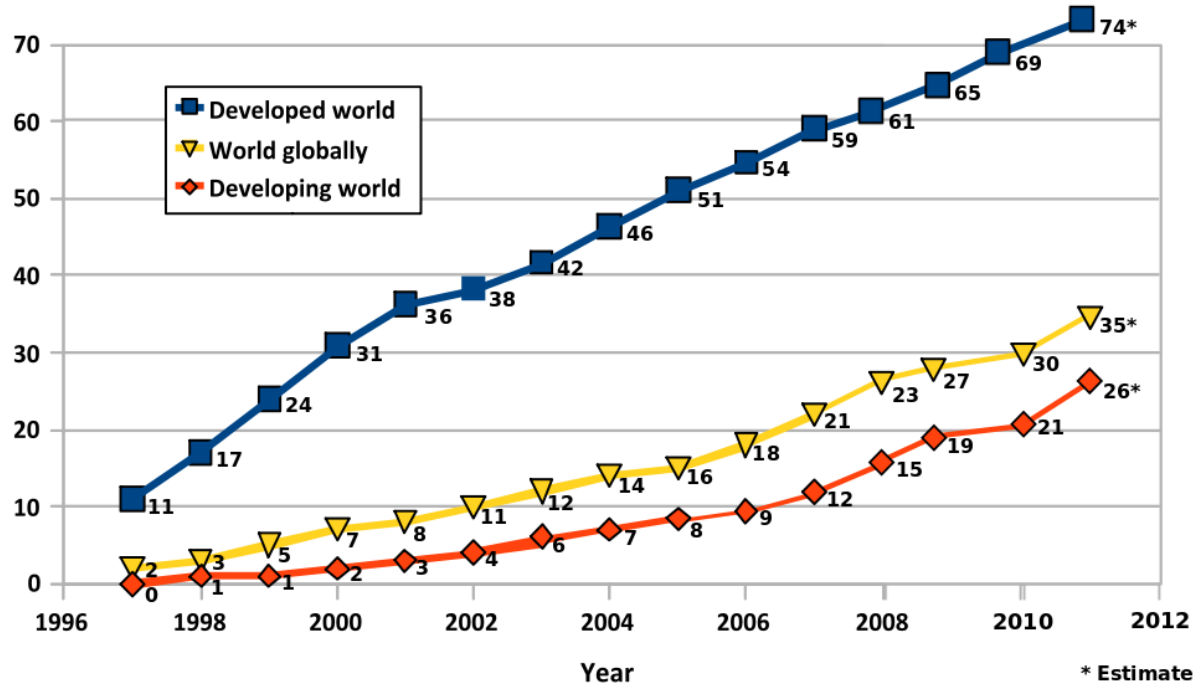


Figure 1: Internet Users per 100 Inhabitants [1]

There has been a fundamental shift in network infrastructure in order to support the need for high performance routing resources. Network routers constitute the core of network infrastructure and perform most of the packet processing applications. The lack of flexibility, programmability and manageability of existing network routers, implemented using application specific integrated circuit (ASIC) technology, highlights the need to consider other networking infrastructures. The need to experiment and adapt newer networking protocols and services has resulted in a shift to software programmable network processor based systems [29]. Network processors [10] have multiple processor cores that can be programmed to adapt to different networking requirements. The software programmability of network processor also makes it vulnerable to network attacks. This inherent vulnerability can be exploited to generate in-network denial of service attacks, as illustrated in Figure 2.

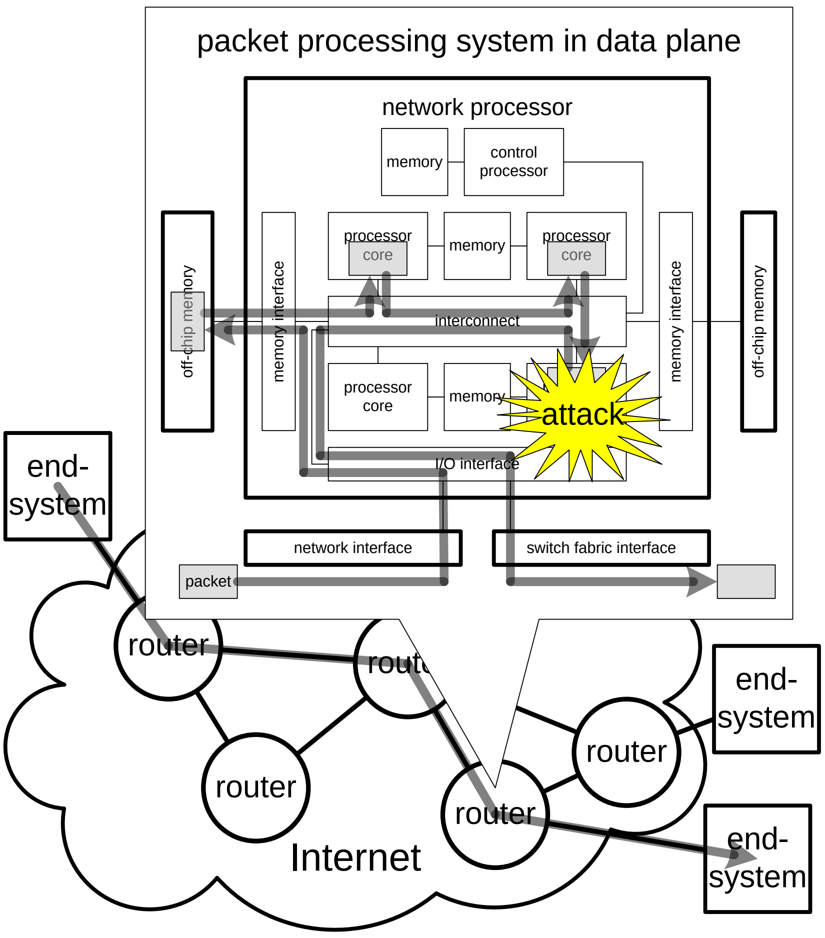


Figure 2: Attack on packet processing system in network router data plane [33]

The existing network security mechanisms for end systems like virus scanners and firewalls are not suitable for network processor based systems since these mechanisms need the support of operating systems. Network processors can benefit from dedicated monitors that can quickly and efficiently detect attacks with minimum resource overhead. A novel hardware based monitoring strategy has been proposed to reduce the vulnerability of network processor based systems [2] [3]. These hardware monitors keep track of program execution flow in the processor. Processor operation is compared to expected program behavior using information stored in the monitor memory. Any deviation from the expected behavior is detected and suitable recovery procedures are initiated. The experimental results highlight the benefits of using hardware monitors for monitoring network processor systems, like fast attack detection and low overhead.

Although the potential of hardware security monitors has been demonstrated in previous approaches, there is still room for improvement in terms of monitor detection accuracy, resource utilization and attack detection speed. Moreover, in multi-core network processor systems, there is an opportunity for sharing monitor resources when multiple processors execute the same application. In this document we present a programmable logic based monitoring system for monitoring multi-core network processor systems. The specific contributions of this work are:

1. The design of a high-performance programmable security monitor which uses hashes of network processor instructions to detect unintended processor behaviour. The application is analyzed offline and an efficient state machine is created which tracks program execution during runtime. If an expected sequence of instructions (represented as hash values) is not followed, an execution error is detected. The state machine can be implemented as either a non-deterministic finite automaton (NFA) or a deterministic finite automaton (DFA). In this research, we investigate security monitors based on DFA state machine implementations [6] [7].

2. A single-core network processor system with security monitor is implemented on an Altera DE4 system. The competence of our proposed system to detect known network attacks is evaluated.

3. We evaluated the resource requirements and throughput performance of our proposed architecture using a diverse set of networking benchmarks [43].

4. Network attacks on network processors based on Harvard architecture [47] are considered. We demonstrate an in-network attack through the data plane of the network that exploits an integer overflow vulnerability to smash the processor stack and launch a return-to-library attack. This attack propagates the attack packet and crashes the processor system. We also show that our hardware monitor is effective in defending against this attack and allowing for continued router operation after attack identification and recovery.

1.1. Organization of the document

The rest of the thesis document is organized as follow. Chapter 2 provides a detailed overview on the general background of network processors, hardware security monitors and the related work in this field. Chapter 3 describes two previous hardware security monitoring approaches and the subsequent improvements introduced in the work. Chapter 4 describes the four-core system architecture with security monitors. Chapter 5 explains the experimental setup for testing the prototype system. Chapter 6 discusses the benchmarks and the obtained results. Chapter 7 concludes the thesis with directions for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, an overview of the technologies needed to perform the proposed research is provided. These technologies include field-programmable gate array (FPGA) chips, embedded systems based on FPGAs, and network processors.

2.1. Field-programmable gate array

A field-programmable gate array is a semiconductor device that can be programmed by a user after it is manufactured. FPGAs contain programmable components called logic blocks and a hierarchy of interconnect elements (wires) which can be configured to connect these logic blocks, as illustrated in Figure 3. Hardware description languages (HDL), such as Verilog and VHDL, can be used to configure the device for implementing specific applications. Modern FPGAs also provide high speed transceivers, embedded memory blocks and high-speed I/Os that help perform complex computational operations [8]. Compared to application specific integrated technology, FPGAs allow for rapid prototyping, faster debugging, ability for easy reprogramming and shorter time to market.

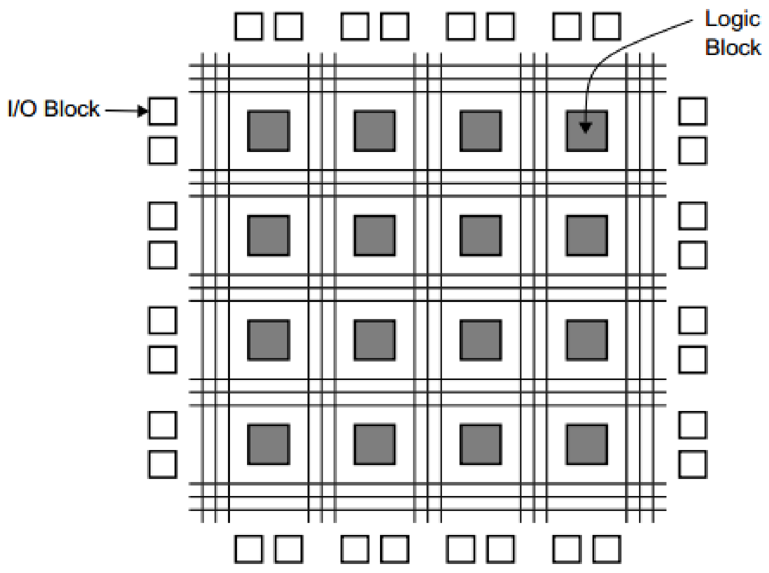


Figure 3: Structure of an FPGA [9]

2.2. Network Processors

The tremendous growth of modern communication infrastructure, such as the Internet, has resulted in the need for networking resources that can meet high throughput performance, flexibility and security. Even though general purpose processors (GPP) can support newly-introduced networking protocols and services, they often do not provide high throughput performance. ASIC processors can provide high throughput but generally do not allow straightforward functionality changes. Network processors (NP) represent the design space between these two approaches, as illustrated in Figure 4.

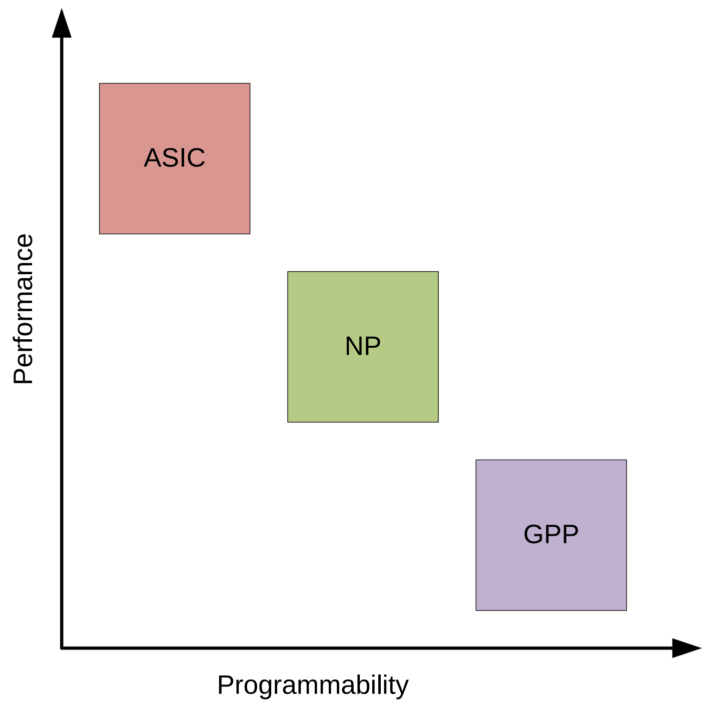


Figure 4: Processor design space

Network processors have multiple embedded processor cores which are software programmable to provide real time programmability and high throughput performance. The architecture of a simple network processor is illustrated in Figure 5.

Network processors consist of multiple processing elements and memory units connected by an on-chip network [10]. A control system determines the interaction between processors and memory elements and the processing required. Based on the workload, software on network processors can be changed to adapt to the processor operation. There are different performance metrics that need to be considered while designing network processors like cost, throughput performance, and power [11] [12]. With promising technologies, like network virtualization, for future internet architectures emerging, significant research is ongoing on network processor based systems [13] [14] [15].

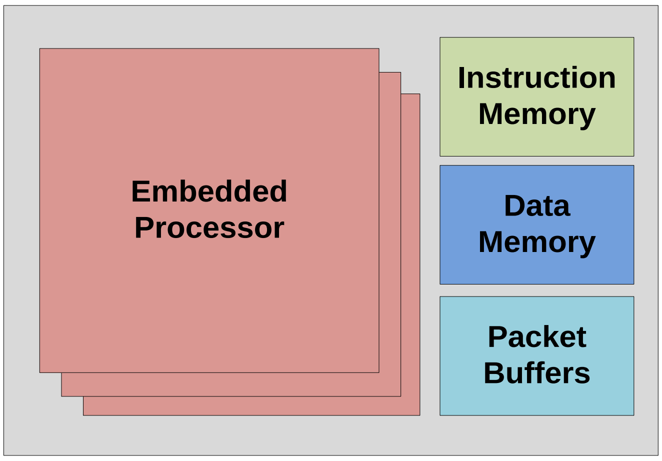


Figure 5: Simple network processor architecture

2.3. Secure monitoring

The software programmability of network processors raises a serious security concern. Network processor systems are vulnerable to network attacks that can remotely exploit their programmable nature. In this section, we look at some network attacks and the existing monitoring techniques for guarding against these attacks.

2.3.1. Network attacks

Network attacks [16] [17] [18] [19] exploit the vulnerabilities in network systems. Sniffing or snooping are network attacks which allow intruders to listen to or interpret traffic. If packets are not properly encrypted, sniffing attacks can give a full view of the ongoing communications. This attack can be used to read the data or cause the network to crash or become corrupted. Distributed denial-of-service attacks [20] occur when an attacker takes control of a large number of machines and installs attack software in them. These machines can then be used to bombard target attack sites with a large number of messages. This attack can result in preventing normal system usage and can also lead to the abnormal behavior of applications or services. Identity spoofing is another common network attack. Here, an attacker uses special programs to construct IP packets that appear to originate from valid addresses. This attack can also be used in combination with denial-of service attacks. The attacker can first take down an existing network connection between two end systems by denial of service attacks. A new connection can then be initiated with one of the end systems by sequence number guessing. The attackers can then modify, reroute or delete the data after taking control of network traffic. A routing attack is another kind of network attack which reroutes network traffic through attack systems.

2.3.2. Software based monitoring techniques

There are many existing software techniques for protecting network systems against attacks. Firewalls [21] and virus scanners are the most common solutions for Internet security [22]. Firewalls help in filtering out traffic that might be harmful. This filtering can happen at IP packet level, TCP session level or at the application level. Firewalls examine incoming packets and filters out malformed or attack packets. They are useful against spoofing attacks and denial-of-service attacks. Virus scanners are examples of intrusion detection systems [23]. These systems are devices or software applications that detect the presence of malicious traffic or services. Firewalls try to prevent intrusions which originate from outside the network. Intrusion detection systems detect suspected intrusions that have taken place and look for attacks being generating from inside the network. Another security measure is the use of encrypted packets for data communication. Here, cryptographic techniques are used to encrypt the data while in transmit. Communication is implemented using Internet Protocol Security (IPSec), a set of open Internet Engineering Task Force (IETF) standards. The encrypted packets have the same format as unencrypted packets and they are transmitted on the existing network framework.

2.3.3. Hardware based monitoring techniques

Existing security mechanisms, like virus scanners and firewalls, generally require powerful processors and operating systems. These resources are generally not associated with network routers on programmable network processors. Instead, hardware monitors can be used to protect network processors against vulnerabilities, as illustrated in Figure 6. Monitors use run time processing information to look for deviations from expected processor behavior. If an attack occurs, a deviation from the expected behavior is detected and a suitable recovery process is initiated. In this thesis project, we introduce a new programmable logic based security monitoring technique for detecting network attacks. In the next two sections we describe two FPGA-based systems which can be used to prototype such a system.

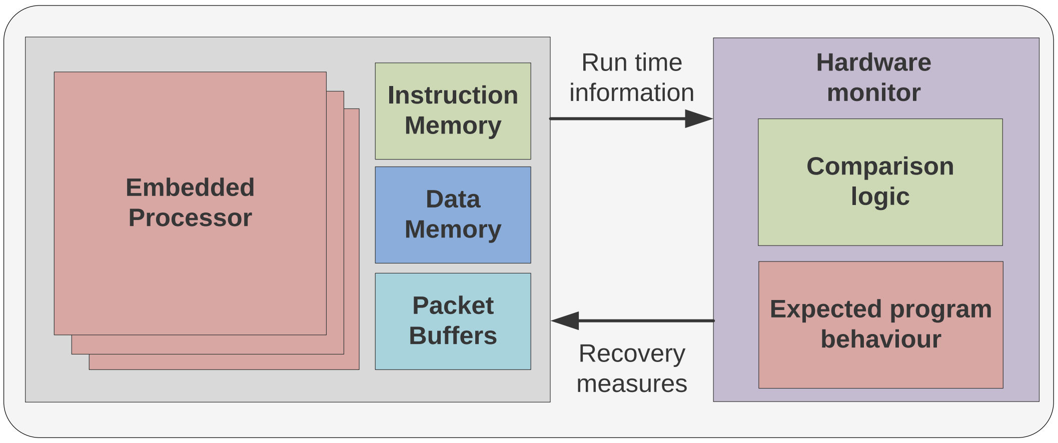


Figure 6: Network processor system with hardware monitor

2.4. NetFPGA 1G Infrastructure

The NetFPGA 1G [24] is a programmable platform for networking research that can operate at 1 Gbits per second line rate. The platform is actively used in networking research with more than 2000 boards deployed worldwide. The system includes a Xilinx Virtex II pro [25] based FPGA which can be configured to perform different networking applications. These applications include a reference router, a packet generator, and a network interface.

2.5. Altera DE4 NetFPGA infrastructure

Although the NetFPGA 1G is widely used, its logic capacity is limited, making it unsuitable for research for this project. The Altera DE4 NetFPGA [37] platform is a suitable alternative. This infrastructure will be heavily used for this thesis project. Important components of the design including the Altera DE4 board, the DE4 NetFPGA software infrastructure, and the reference router and packet generator modules, are described subsequently since they are referenced in our discussion of the proposed security architecture in coming chapters.

2.5.1. Altera DE4 FPGA board

The Altera DE4 FPGA board [26] is a research platform featuring an Altera Stratix IV FPGA [27]. The DE4 board which will be used to complete the work described in this thesis is shown in Figure 7. The main features of the DE4 board are:

- Stratix IV FPGA with 5x more logic and memory resources compared to a NetFPGA 1G platform.
- PCI Express interface which allow for faster host PC to FPGA data transfers.
- Up to 8GB external DDR2 memory .

2.5.2. Altera DE4 NetFPGA Infrastructure

Altera DE4 NetFPGA is an open source port of the NetFPGA 1G infrastructure to an Altera DE4 board. The DE4 NetFPGA provides network researchers with a powerful open platform to build complex network applications. A high-level view of the DE4 NetFPGA architecture is illustrated in Figure 8. We have successfully migrated the NetFPGA reference router [44] and packet generator [45] designs to the Altera DE4 platform.

Figure 7: Altera DE4 board

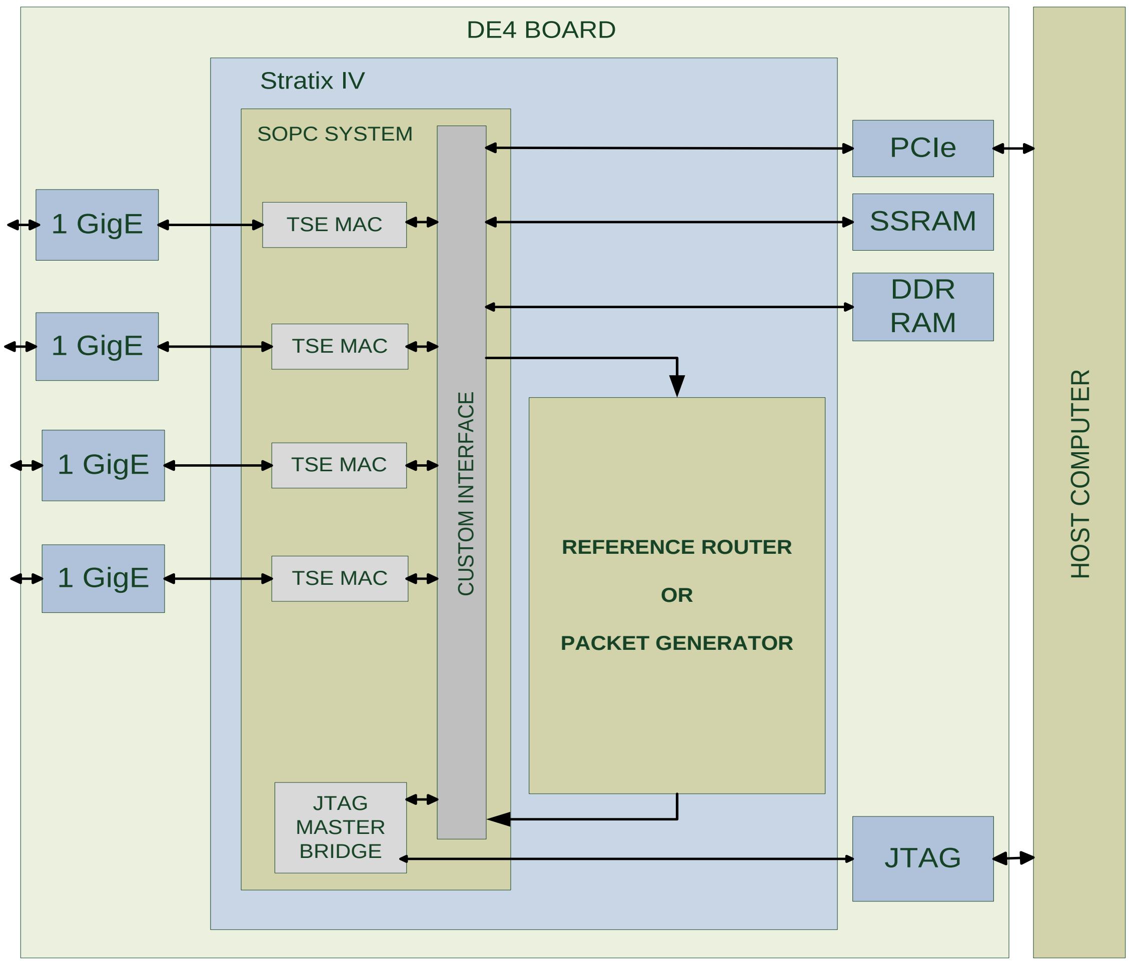


Figure 8: Altera DE4 NetFPGA Architecture

2.5.3. Altera DE4 NetFPGA reference router

The Altera DE4 reference router system integrates the NetFPGA reference router pipeline with Altera DE4 NetFPGA platform. The Altera DE4 NetFPGA reference router system is illustrated in the Figure 9.

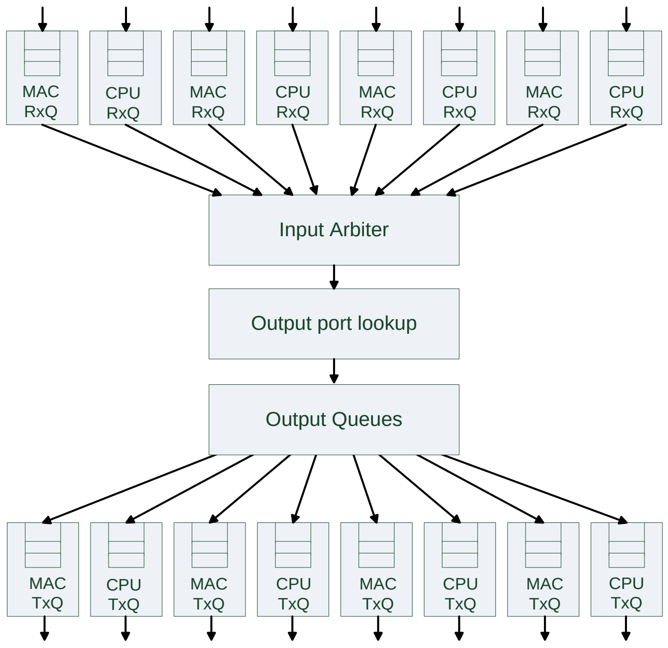


Figure 9: Reference router pipeline

The incoming packets from DE4 GigE MAC ports are stored in input queues (MAC RxQ and CPU RxQ). The input arbiter then forwards these packets to the output port lookup module. The output port look up module performs the network router operation. The output queues forward the packets to corresponding output MAC ports (MAC TxQ and CPU TxQ). The multi-core network processor system is implemented as part of the Altera DE4 NetFPGA reference router pipeline, as explained in section 4.5.

2.5.4. Altera DE4 NetFPGA packet generator

Altera DE4 NetFPGA packet generator application allows for packet generation and forwarded packet capture at line rate. The packet generator also provides packet transfer transmit and receive statistics. The Altera NetFPGA packet generator is used for testing the prototype system functionality, as explained in section 5.2.

2.6. Related work

Modern high-speed network infrastructures utilize network processors since they offer sophisticated packet processing capabilities and advanced protocol functionality. The programmable nature of network processors allows for experimentation with new architectures and protocols [10]. Research projects have examined improving network processor throughput, resource management, power analysis, packet classification, and deployment [28] [29]. Network processor based routers are commercially deployed by many major device vendors (e.g., Intel IXP2400 [30], Cisco QuantumFlow [31], Cavium Octeon [32]).

Network processor systems are vulnerable to remote attacks that target the software on the processors. Chasaki et al. [33] demonstrated how network processor systems can be exploited to launch denial-of-service attacks by using a single malformed packet. Protecting a network infrastructure against such malicious attacks is an important concern in network processor design. Hardware-assisted run-time monitoring techniques have been used in protecting embedded processors. Arora et al. [4] [34] showed how dedicated hardware monitors can be used to track and prevent unintended program behavior. The hardware monitor observes the run time execution of the processor and compares it with statically analyzed expected program behavior. Unexpected behavior is used to initiate appropriate response mechanisms. Mao et al. [35] used hardware monitors with offline analyzed control flow graph information to protect embedded processors.

The embedded nature of network processors allows for the use of hardware monitoring schemes against in-system attacks. Wolf et al. [36] proposed using a secure packet processing platform for network processors with hardware monitors. Using offline analyzed monitoring graphs, the program execution of packet processors can be monitored for attacks and suitable recovery measures taken. Chasaki et al. [2] have implemented a hardware security monitoring model that can detect known network attacks.

2.7. Summary

This chapter introduced several concepts that are essential for understanding the prototype system. We provide an overview of the Altera DE4 NetFPGA infrastructure used for development and testing of the prototype system. Although hardware security monitoring systems for network processors exist, there is still potential for improvement. The next chapter outlines two specific previous monitoring approaches that are limited in their capabilities. Subsequent chapters describe enhancements to overcome these limitations.

CHAPTER 3

SECURITY MONITOR SYSTEM REVIEW

This chapter introduces two previous hardware security monitoring techniques for network processors. We will first look at the implementation details of an existing hardware security monitor [2]. The limitations of this hardware security monitor motivate us to develop a programmable hardware security monitor. We will compare the advantages and drawbacks of these two security monitoring approaches. Based on these observations, we outline the important security monitoring features implemented in the new monitoring system presented in this thesis.

To provide a basis for comparing the two approaches, we state the main design challenges that need to be met when using hardware security monitors.

1. Correct detection: The monitoring system needs to correctly identify malicious attacks. There is a variety of information available for security monitors to keep track of processor operation at run time:

    - Instruction address: The security monitor can follow the instruction addresses of the application binary, since the feasible address sequences can often be predetermined.
    - Opcode: The security monitor can track the operations performed by the processor.
    - Instruction hash: The monitor can use a sequence of instruction hash values to verify processor behaviour.

    The main tradeoff for choosing a monitoring strategy depends on the availability of hardware resources and the difficulty in defeating the effectiveness of the monitoring behavior by an attacker. The security monitor can utilize one or more of the above mentioned monitoring options to correctly detect unintended behavior.

2. Resource overhead: Security monitors need to be designed while considering the limited resource availability of their implementation platform. Increasing the resource usage can adversely affect power consumption and design complexity.
3. Fast detection: Programmable network processor operation can be changed by altering the software on the processor. Hence, it is desirable to detect malicious behaviour quickly, preferably within one or a small number of clock cycles.

3.1 Address-based hardware security monitor system

An address-based hardware security monitor [2] uses instruction address information for monitoring processor behavior. The processor application binary is analyzed offline and instructions are classified into different basic blocks. Each basic block represents a set of instructions, before a branch instruction is encountered as illustrated in Figure 10.

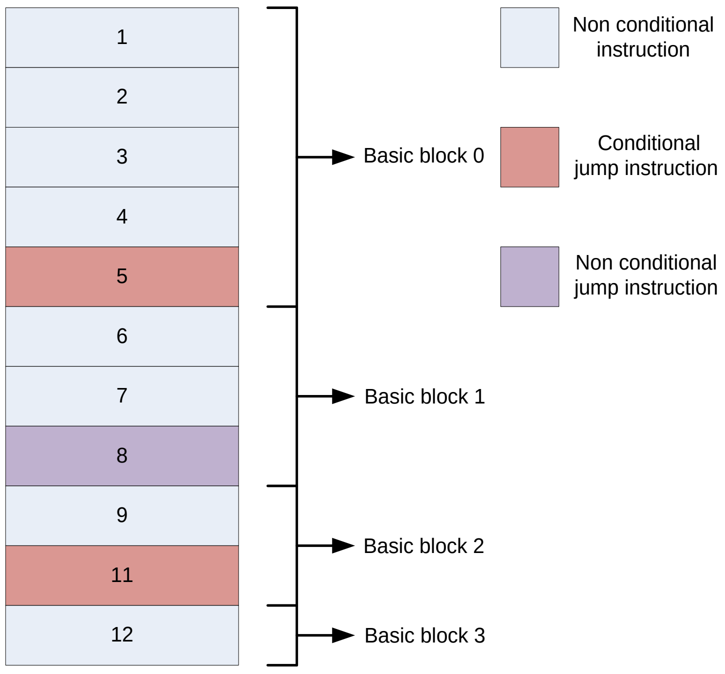

1
2
3
4
5
6
7
8
9
11
12
Basic block 0
Basic block 1
Basic block 2
Basic block 3
Non conditional instruction
Conditional jump instruction
Non conditional jump instruction

Figure 10: Basic block representation

The fifth instruction is a conditional jump instruction, and hence all instructions from memory locations 1 to 5 represent basic block zero. Similarly, instruction eight is an unconditional jump instruction. All instructions from memory locations 6 to 8 represent basic block one and so on. The basic block information for each instruction is used to validate the processor operation at runtime.

The high level architecture of the address-based hardware security monitor system is illustrated in Figure 11.

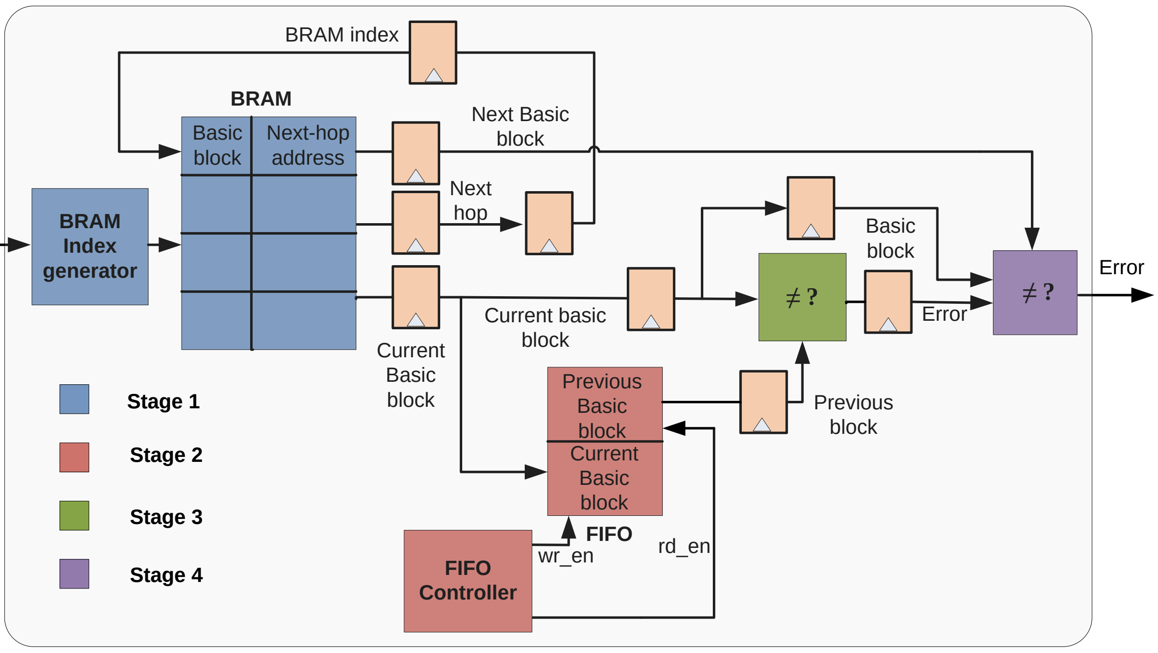


Figure 11: Address-based hardware security monitor architecture [2]

3.2. Address-based hardware monitor operation

The address-based hardware security monitor system is fashioned as a four stage pipeline. Each pipeline stage takes one clock cycle to complete. In the first stage the instruction address of the currently executed instruction in the processor is used to index a block RAM (BRAM). The BRAM outputs the basic block number of the instruction as well as the next-hop address, if there is one. In the second stage, we forward the basic block number to the third stage. At the same time, the basic block number output is stored in a FIFO block. During the third stage of the monitor operation, the current basic block number input from the second stage as well as the block information for the just completed instruction from the FIFO block are compared. If they are the same, the instructions belong to the same basic block and the currently executed instruction is valid. If not, check if the instruction is within the next basic block, which is a valid basic block jump (e.g. jump from basic block 0 to basic block 1). If not, a check is required to determine whether the currently executed instruction belongs to the basic block which contains the target instruction for a jump. During the fourth stage the next-hop address for the just completed instruction is used to once again index the basic block memory. If the basic block for this target is the same as the basic block of the currently-executed instruction, a valid instruction sequence is determined. Otherwise, an error signal is generated to stop processor operation.

3.2.1. System components

3.2.1.1. BRAM index generator module

The BRAM index generator module generates the index output to address the basic block memory (BRAM) from the 32-bit processor instruction address. The first instruction address from the processor represents address zero of the BRAM. This is used as the base address to generate remaining index outputs.

3.2.1.2. FIFO controller module

The FIFO controller module generates the write signal to store the current basic block information and the read signal to read the basic block information of the just completed instruction.

3.2.1.3. Basic block memory (BRAM)

Each element in the basic block memory (BRAM) corresponds to an instruction in the program sequence and is indexed by an instruction address of the application. The basic block memory contains two entries per index, the basic block number to which each instruction belongs and the next-hop address to where the instruction could jump. The next-hop address entry is empty for unconditional instructions.

3.2.1.4. Basic block FIFO

The basic block FIFO stores the basic block numbers of the just completed and currently executed instructions, as read from the basic block memory. These values are used to keep track of the processor execution path.

3.3. Limitations of address-based hardware security monitor.

The basic block monitoring strategy for tracking processor behavior does not directly validate individual instructions as they are executed by the processor. For example, malicious instructions can go undetected if the instructions belong to the same basic block as the expected instructions, as illustrated in Figure 12. In this example, malicious instructions (2 and 3) will not be detected as long as they follow the program memory address execution sequence. Moreover, the inclusion of a next-hop address field, which is required only for branch instructions, with the basic block information for all instructions in the basic block memory increases on-chip memory utilization. To reduce memory utilization, the basic block memory can be shared using multiple read ports. For example, the basic block memory can be read simultaneously during both stage 4 and stage 1 of the pipeline. Such sharing does not allow basic block memory sharing across multiple hardware monitors when multiple network processor cores execute the same program. So, in multi-core network processor systems, separate address-based hardware security monitors need to be generated when the processors execute the same application.

3.4. Programmable security monitor using instruction hashing

The limitations of the fixed hardware security monitor motivate us to develop a new monitoring strategy which can validate individual instructions and reduce the embedded memory usage for the monitor.

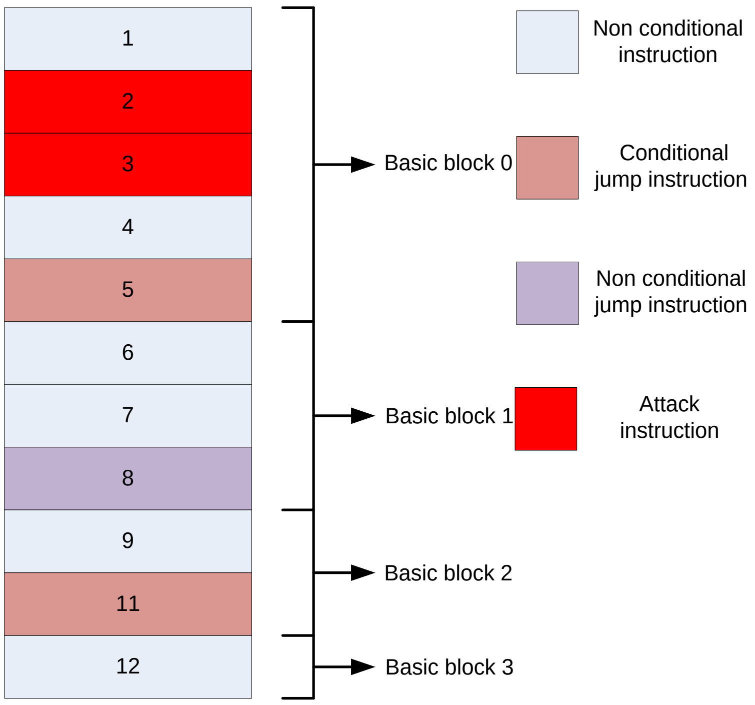

1
2
3
4
5
6
7
8
9
11
12
Basic block 0
Basic block 1
Basic block 2
Basic block 3
Non conditional instruction
Conditional jump instruction
Non conditional jump instruction
Attack instruction

Figure 12: Undetectable network attack

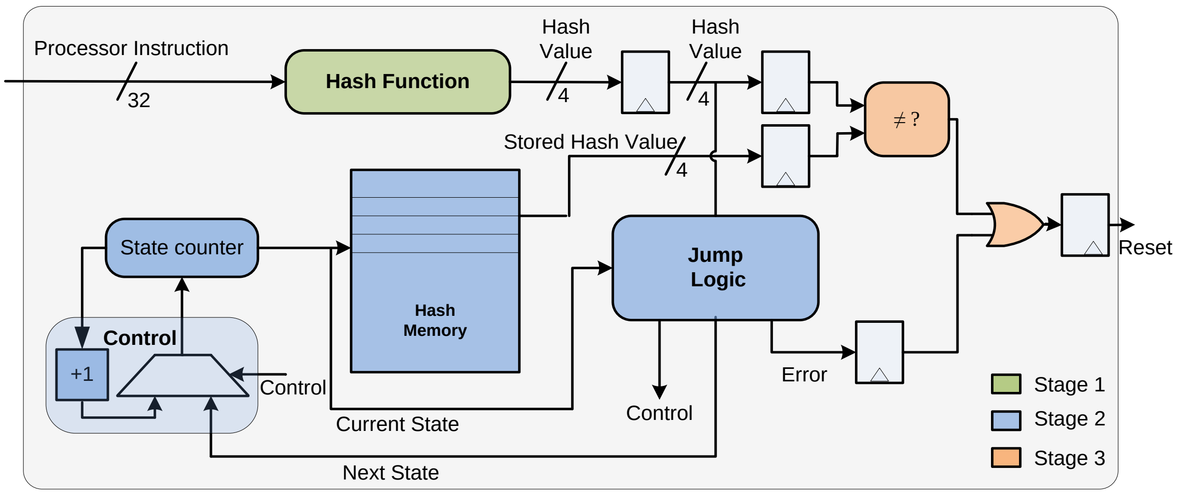


Figure 13: Programmable security monitor architecture

3.4.1. Programmable logic monitor operation

The operation of the programmable logic monitor relies on compile-time analysis of the program binary. An analysis tool determines the expected hash values for the binary and generates the Verilog files needed to synthesize the programmable security monitor, including the jump logic. As shown in Figure 13, the programmable logic monitor infrastructure is fashioned as a three stage pipeline flow. Each 32-bit binary instruction executed by the processor is input to the monitor. The first stage of security monitor generates the four bit hash value from this 32-bit instruction. In the second stage, two parallel operations are performed.

- The four bit hash value from the hash memory is fetched.
- The hash value is evaluated in the jump module. If the hash value does not match one of the target hash values for the jump, an error signal is generated indicating that an incorrect instruction has been executed.

In third stage, for unconditional instructions, a comparison is made between hash value computed from stage 1 and the value retrieved from hash memory. A reset signal is generated if there is a mismatch or if stage 2 generated an error signal.

3.4.2. System components

3.4.2.1. Hash function module

The hash function module computes a multi-bit hash value for the 32-bit processor instruction. In our initial experimentation we use a hash function which generates a modulo-16 value from the sum of all individual bits in the instruction to form a four-bit hash value.

3.4.2.2. Hash memory module

The hash memory module is a 2-port ROM block that stores the hash values generated during offline analysis. The hash values are used to keep track of processor behaviour during run time.

3.4.2.3. Jump logic module

The jump logic module provides the next-hop hash memory addresses for branch instructions when a branch is taken. The processor typically executes three different types of instructions.

1. An *unconditional instruction* is followed by execution of the next consecutive instruction in the processor memory.

1. An *unconditional branch instruction* will result in a jump to a non-consecutive memory location in instruction memory.
2. A *conditional branch instruction* can result in a jump to a new address location following the evaluation of the branch condition. If the branch is not taken, the next consecutive instruction is executed.

Since a branch condition for a conditional branch instruction can only be evaluated at run time, both possible hash values for target address locations need to be considered by the jump logic module. During runtime, the hash values of the instruction after the conditional branch are compared with the possible hash values of the branch targets to determine if a valid instruction is being executed. The jump logic module then determines the address in the hash memory for the appropriate target instruction.

3.4.2.4. Control module

The control module (the multiplexer and +1 adder) determines the address for the hash memory. For unconditional instructions, the hash value for the next instruction is stored in the next consecutive address location. In case of conditional instructions, the address value for the hash memory is output from the jump module.

3.4.2.5. Programmable security monitor advantages

The key improvements of the programmable security monitor using instruction hashing over the address-based monitor include:

1. The programmable security monitor uses a hash of each instruction to monitor processor behaviour instead of storing basic block information for each instruction. A hash value is computed for each program instruction and stored in a hash memory. As instructions are executed, the hash of the currently-executed instruction is compared against a stored hash value for the instruction. The hash value of the first instruction is located in hash memory at location 0x0, the second at 0x1, and so forth. A state counter is used to point to the hash of the currently executed instruction. The strength of the monitoring scheme improves with hash value bit width since the probability of hash collisions (collision probability is $1/2^x$, where x is the number of bits) is reduced. To keep hardware requirements reasonable, the width of the hash values is constrained to the minimum size which allows the required security. A four-bit hash value is chosen for monitoring purposes as this size provides a strong monitoring pattern (collision probability 0.0625) which limits memory overhead.
2. Next instruction addresses for conditional instructions are determined in the jump logic, reducing memory resource utilization. Combinational logic is used to determine the address of the next location in the hash memory for conditional instructions which are taken. For conditional instructions where a branch is not taken or for unconditional instructions, the next desired location in the hash memory is the next consecutive location in the memory.
3. The design of the programmable security monitor allows for sharing of the hash memory monitoring resource. The hash values are stored in a 2-port ROM memory block which is utilized only once during each instruction evaluation cycle. The two ports allow for the sharing of the hash memory, a critical resource, by two monitors that are evaluating the same executing program.

3.5. Limitations of programmable security monitor

The jump logic module can require significant logic resource utilization. The instruction at memory location 0 is a conditional jump instruction, with next-hop address location (1 or 4) evaluated based on the hash values (x and y). An offline analysis tool generates a state machine to perform this evaluation, initiated every time the memory address input is zero. Once a next state (1 or 4) is reached, the state machine will continue to evaluate. If there are *n* conditional branch instructions (with 2 next states), *2n* possible state transitions need to be considered. Additionally since the jump logic is implemented in logic, we need to re-synthesize the design for each different application.

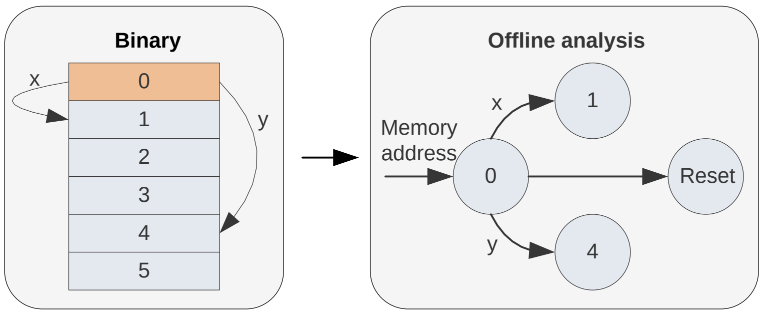


Figure 14: Jump logic implementation

3.6. Comparison of address-based hardware and programmable monitors

3.6.1. Resource overhead

We compared the resource utilization of the programmable security monitor to the address-based hardware security monitor technique. Table 1 provides the comparison results from MiBench [42][42] benchmarks and two other packet processing applications (IPV4 [39] and CM [40]). Since the next-hop addresses are not stored in block memory, the programmable security monitor approach results in a considerable reduction in memory utilization compared to the fixed hardware monitoring technique. The logic resources (ALUTs and registers) show a considerable increase in the programmable security monitor approach with program size, which results from the jump logic implementation illustrated in Figure 14.

	Instructions		Address-based Monitor			Programmable logic monitor		
Benchmark	Total	Control Flow	ALUs	Registers	Memory bits	ALUs	Registers	Memory bits
Des	739	15	40	35	12416	238	147	4096
factorial	141	10	40	33	3200	217	137	1024
fir	175	13	40	33	3200	252	146	1024
iquant	371	13	40	34	6272	314	170	2048
IPV4(1core)	327	17	40	34	6272	313	159	2048
CM (1core)	289	21	40	34	6272	329	167	2048
IPV4(4core)	327	17	160	136	25088	863	486	2048
CM(4core)	289	21	160	136	25088	974	529	2048

Table 1: Resource utilization comparison

3.7. Memory based programmable security monitor approach

This section presents an introduction to a memory based technique for representing hash based state machines. Figure 15 illustrates a hash based state machine. Here (0,1,2,3,4...) represent the states and (a,b,c,d...) represent the input (hash values) to the states. The state machine is traversed based on the current state and the hash input.

Consider such a state machine with a two bit hash input. For any state there are at most four outgoing edges possible based on the input values (00, 01, 10, 11). A naïve way to store the state machine in RAM would be to store each state and all possible edge transitions as illustrated in Figure 16. The current state and the input hash pattern can be used to index the memory to find the next state transition. Since for most states, the next state transition may not be present for all input patterns, a valid bit is provided to verify the state transition.

For example in Figure 15, state 0 has only 1 state transition, state 2 hash 2 transitions, state3 has 3 transitions and so on.

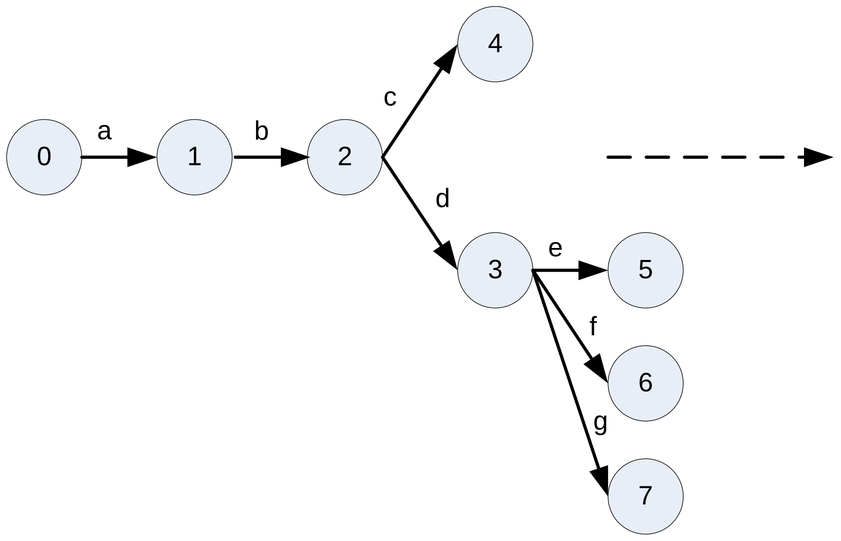


Figure 15: DFA state machine

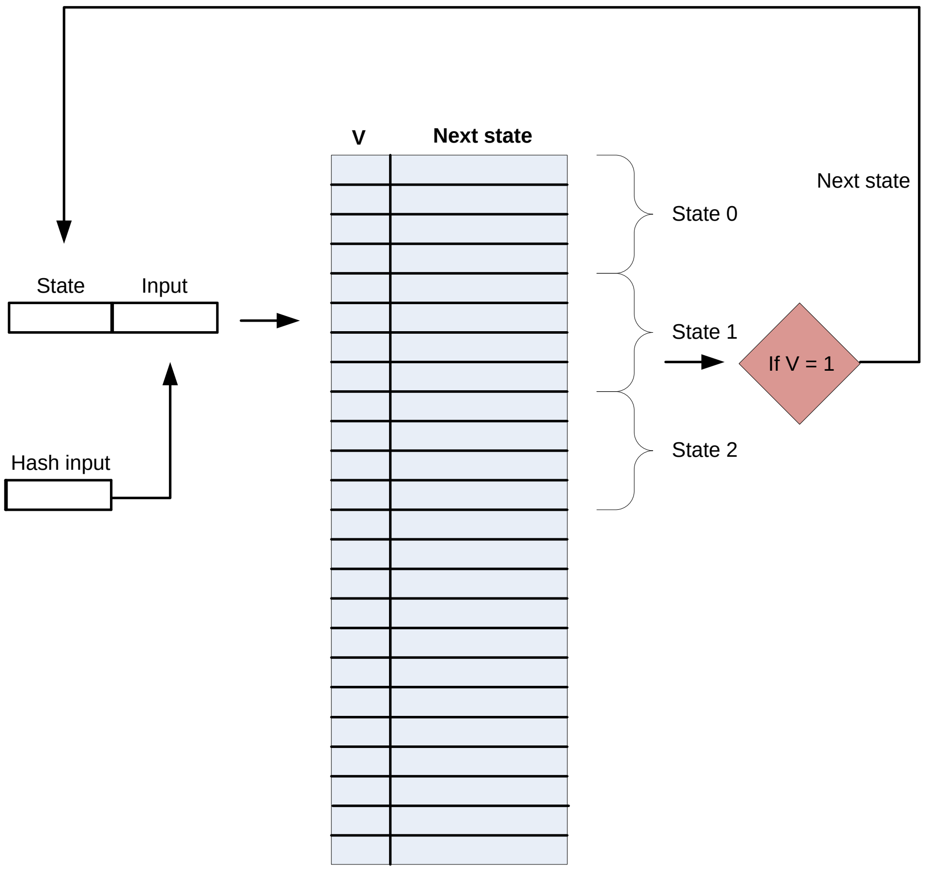


Figure 16: DFA single memory representation

The memory representation illustrated in Figure 16 is not a cost effective solution since it results in a large number of unused memory locations. This is since there is a memory location for each possible input combination. An alternative solution is to provide separate memory logic for multiple next states as illustrated in Figure 17. For example in Figure 15 , for states (0, 1 and 2) we index only a single memory (memory for one state). When state 2 is indexed, the jump bit (J) is set high to indicate more than 1 next state. In the next step we index all the parallel memory blocks (memory for multiple states) simultaneously. If the valid bit (V) is set and the hash input matches (either c or d), we move to the corresponding state (either 3 or 4). For a four bit hash input we need 16 parallel memory blocks.

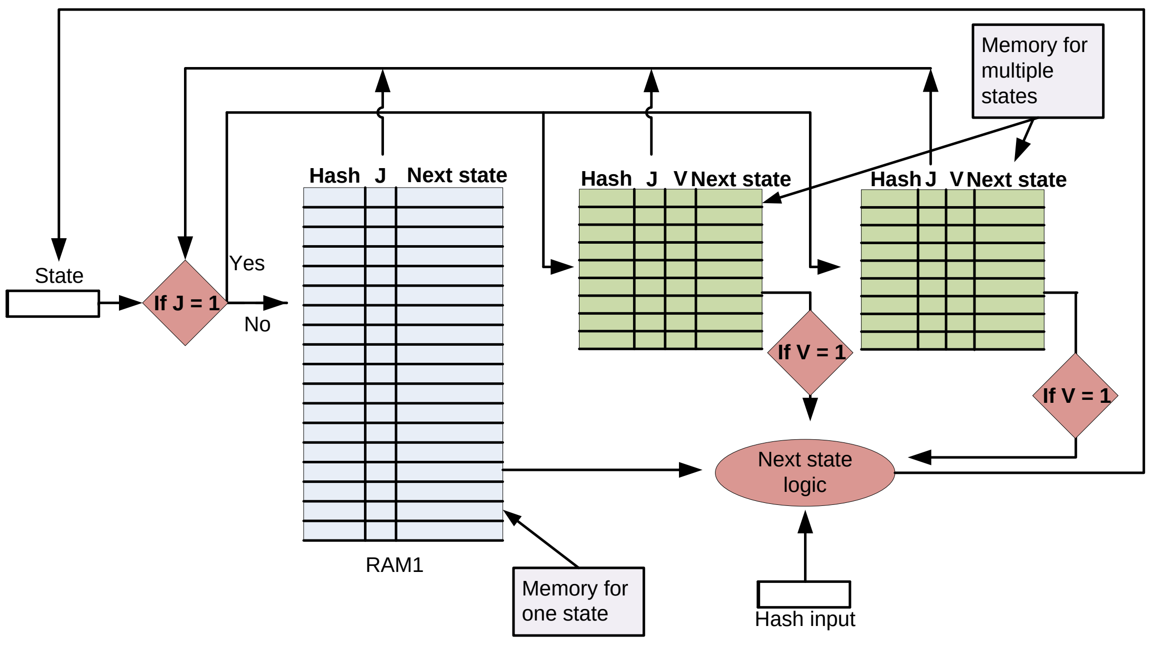


Figure 17: DFA parallel memory representation

The memory representation illustrated in Figure 17, requires simultaneous memory access (2^h, where h is the input bit width) and in many cases there are unused memory locations. We need a compact solution which requires only single memory lookup during every state transition and does not result in unused memory locations. Thus, state transitions need to be implemented with no more than one memory access per instruction (to keep up with the network processor core) and be as compact as possible (to minimize the implementation overhead of the monitor). In the next chapter we introduce the proposed security monitoring approach which presents a compact solution to represent a state machine in memory.

3.8. Conclusion

Based on the two monitoring techniques mentioned above, we outline the following important features of the two monitoring approaches:

1. Four-bit instruction hashes can be utilized to monitor processor behaviour since it provides a compact representation for each instruction.
2. It is desirable to limit or eliminate target instruction address locations for conditional branches.
3. The evaluation circuitry for incorrect hash values should be efficiently implemented. In the programmable monitor case, this circuitry is implemented using combinational logic. In the next chapter, a memory-based implementation is presented.

CHAPTER 4

SYSTEM ARCHITECTURE

In this chapter, we present a new high performance security monitor system for multi-core network processors. The generation of security monitors for a multi-core system begins with the offline analysis of packet processing application binaries. In this chapter we describe an automated process to generate these binaries and configure available monitors fashioned from hardware. We implemented our prototype system as part of a reference router on the Altera DE4 NetFPGA platform. This chapter describes the individual components in the development of the prototype system.

4.1. Memory based programmable security monitor architecture

The high level view of our proposed memory based programmable security monitor architecture is illustrated in Figure 18.

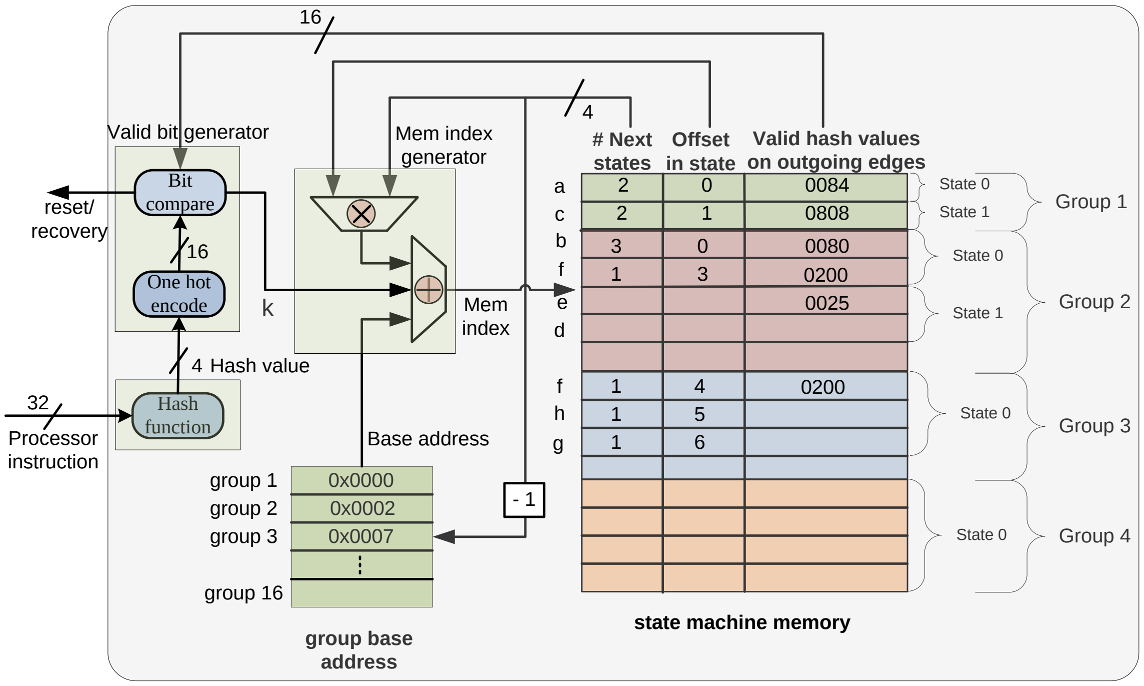


Figure 18: Memory based programmable security monitor architecture

We introduce the following enhancements over the previous two security monitor models.

1. The processor operation is monitored using four-bit instruction hashes. The limitations of address-based hardware monitoring scheme, as explained in section 3.3, motivated this choice. This change allows the validation of individual instructions, and reduces the embedded memory usage of the monitor.
2. The programmable security monitor scheme discussed in previous chapter suffers from logic increase in the jump logic unit due to a need to determine next hop addresses for the hash memory, as illustrated in section 3.5. Additionally it requires re-synthesizing the design for different application. In order to overcome these issues, we utilize a memory based approach to store hash values. This will provide for a more compact monitor solution.

4.1.2. Memory based programmable security monitor operation

The information that needs to be stored in the monitoring memory is illustrated on the left side of Figure 19. Each state represents an instruction and an outgoing transition edge from this state represents the hash value of the next expected instruction in the execution sequence. For example, the state c has two next states, d and e, with hash values 11 and 3, respectively.

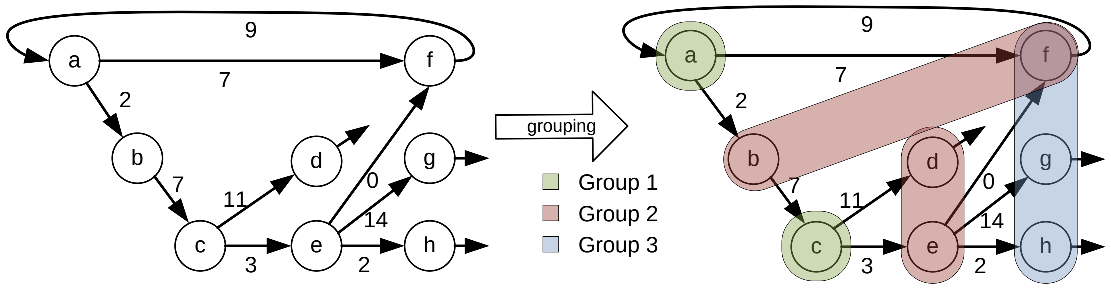


Figure 19: Grouping of states

Our main idea to compactly represent DFA states with varying numbers of outgoing edges is to encode all the necessary information in a single table entry and to group states by the number of outgoing edges. The main challenge in achieving compactness is to allocate exactly the amount of memory that is needed for each state to store next state information while still being able to index this memory without degrading to a linear search. In our representation, we group states if they have same previous state. A state belongs to group g if the previous state has g outgoing edges. For a monitor with 4-bit hash value, there are 16 possible groups. For example, in Figure 19 on the right side, groups are shown with different colors. Note that a state can belong to multiple groups (e.g., state f belongs to group 2 (because a has two outgoing edges, one to b and one to f) and to group 3 (because e has three outgoing edges)).

The memory contains tuples of {number of next states, offset in state group, valid hash values on outgoing edges} and is logically divided into groups. The base addresses for each group are stored in register file with 16 entries. Within a group the sets of states that share the previous state are grouped together (e.g., b and f are together and d and e are together). Within a set, states are ordered by the hash value on the incoming edge (e.g., e before d because hash value 3 is smaller than hash value 11).

To illustrate the operation of the monitor, we describe an example transition. As shown in Figure 18, each 32-bit binary instruction executed by the processor is also input to our security monitor during the fetch stage of the processor. Assume the monitor is in state a and the processor reports an instruction that leads to a hash value of 7. To perform the transition, the memory row labeled a is read. The tuple in this row indicates that there are two outgoing edges. The valid hash values of these two edges are stored in the 16-bit vector. To verify that the transition is valid, the hash comparison unit checks if bit 7 is set in the bit vector (which it is). If this bit is not set, then an invalid transition takes place, indicating an attack, and the processor is reset. After the check, the next state (i.e., state f) in the DFA needs to be found in memory. To determine the address of that state, the base address of the group of the next state is looked up in the register file (i.e., 0x002 since the next state belongs to group 2). To this base address, the product of the set size (i.e., group number) and the offset in the state group is added (to index the correct set within this group). Finally, k is added, which is the position of the matching hash in the bit vector (in our case 1 since 2 is the first matching hash (i.e., k=0) and 7 is the second matching hash (i.e., k=1)). Thus the memory location of state f is 0x002 + 2*0 + 1 = 0x003.

Note that any state transition takes only one memory read from state machine memory and a lookup into a fixed-size register file. The DFA is represented compactly without wasting any memory slots. Thus, this representation lends itself to high-performance implementation.

4.1.3. System components

4.1.3.1. Hash function module

The hash function module generates the four bit hash value for the 32-bit processor instruction. For our experimentation, we used a hash function which generates a modulo-16 value from the sum of all individual bits in the instruction.

4.1.3.2. Group base address module

The group base address module stores the base address of the different groups in the state machine memory module. The ***number of states*** value read from memory is used to index the group base address module. The corresponding ***base address*** value output is forwarded to the memory index generation module.

4.1.3.3. Valid bit generation module

This module compares the 4-bit hash value generated by the hash function module with all the read hash values from the memory to determine the position of the matching hash value. If no hash match occurs, an error signal is generated.

4.1.3.4. Memory index generation module

This module generates the memory address for indexing the state machine memory module. The general equation for memory index calculation is ***base address + number of states*offset in state group + k***. The ***base address*** value is generated by the group base address module, ***number of states*** and ***offset in state group*** values are read from memory and ***k*** is generated by the valid bit generation module

4.1.3.5. State machine memory module

The state machine memory module stores the hash values and next state values in a compact manner as explained in section 4.1.2. The RAM block is divided into different groups, which store the states having same number of next states. Each memory entry contains tuples of {number of states, offset in state group, valid hash values on outgoing edges}. For our experimentation we selected a 4096 deep RAM block as the state machine memory module. So the total size of the state machine memory block is 4096*(4+12+16) = 131K memory bits.

4.2. Offline analysis

The automated offline analysis tool for security monitor generation is illustrated in Figure 20. The application source code is first passed through a MIPS-GCC compiler. The compiler generates the 32-bit binary information for each instruction and the branch information for conditional instructions. The branch information contains all possible target addresses for the conditional instruction. In our current implementation, all possible branch targets and return instructions are analyzed at compile time. Then the DFA-to-NFA conversion starts with a non-deterministic NFA representation obtained from the compiler information. Through powerset construction, a DFA is constructed. This DFA is then converted into a memory initialization file and is loaded into the monitor when the processing binary is installed in the processor. The *NFA to DFA conversion* module will be explained in detail in section 4.3.

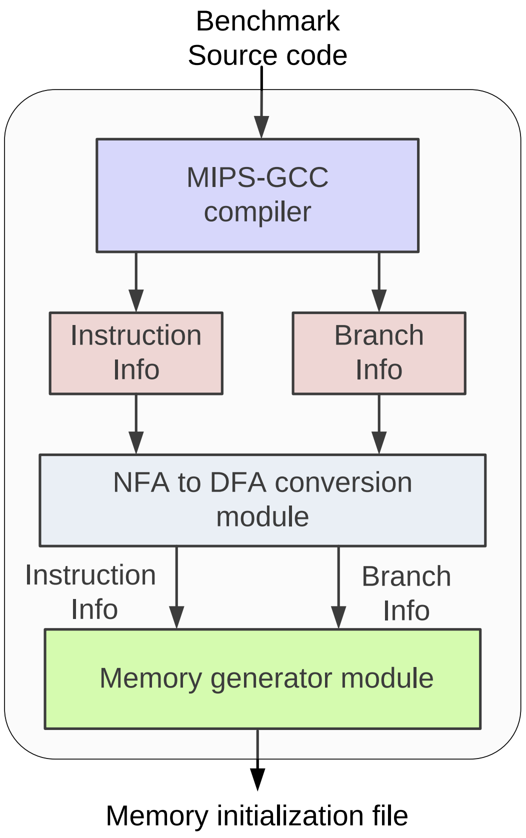


Figure 20: Offline analysis

4.3. Non deterministic finite automata to deterministic finite automata

Tracking nondeterministic finite automata is difficult to implement in practice since the automaton can have multiple active states. This leads to high bandwidth requirements between the monitoring logic and the memory that maintains the NFA since next-state information for all active states has to be fetched in each iteration. As illustrated in Figure 21, state 4 and 6 can be reached from state 3 for input condition (z). When using a DFA, in contrast, only one state is active and implementation becomes much easier. During offline analysis, state assignments must be made so that the control flow is distinct.

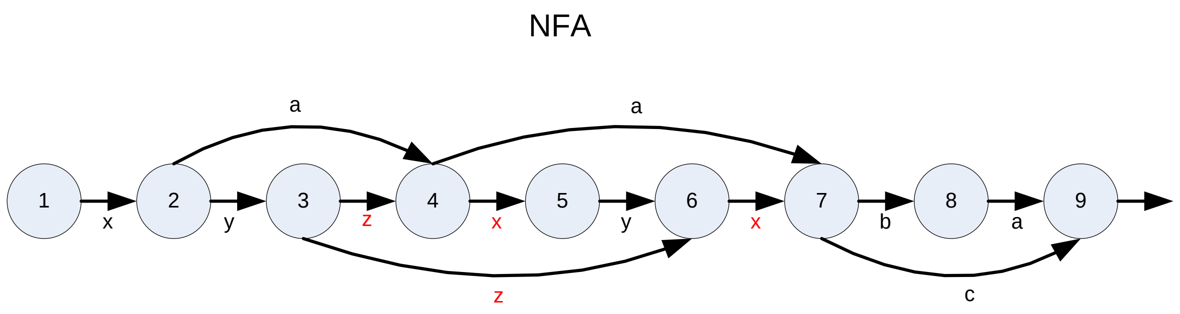


Figure 21: NFA state machine

Powerset construction [50] is a standard method used to convert an NFA to a DFA. The algorithm for NFA to DFA transformation using powerset construction is illustrated in Figure 22.

1. The algorithm begins by constructing an NFA state machine, with the each state node having the following elements: number of inputs, number of outputs, input list and output list (for example state 3 has number of inputs: 1, number of outputs: 2, input list: [state 2] and output list: [state 4, state 6]).
2. The algorithm then progresses to check for conditional branch instructions (*number of outputs* > *1*). In the NFA state machine in Figure 21, states 2, 3, 4 and 7 satisfy this condition.
3. The algorithm then proceeds to check for states exhibiting NFA property (*hash1* = *hash2*). In the above example state 3 has two output states (4 and 6) both reachable by the same input (z), which represents the hash value of the next state. The algorithm replaces states 4 and 6 in the output list of state 3 with a new single state ({4, 6}), distinct for the input value (z). The node elements of the states involved are updated (state 3 now has number of inputs: 1, number of outputs: 1, input list: [state 2] and output list: [state {4, 6}]. Similarly new state {4, 6} has number of inputs: 1, number of outputs: 2, input list: [state 3] and output list: [state 5, state 7]).
4. This procedure is continued until all outputs of the present state (if there are more than 2 outputs) and all states of the NFA state machine are traversed. Both state 3 and new state

{4, 6} exhibit NFA properties (for input values z and x respectively). This results in two additional states being included in the DFA state machine (state {4, 6} and state {5, 7}). The resulting DFA transformation using Powerset construction for the above NFA example is shown in Figure 23.

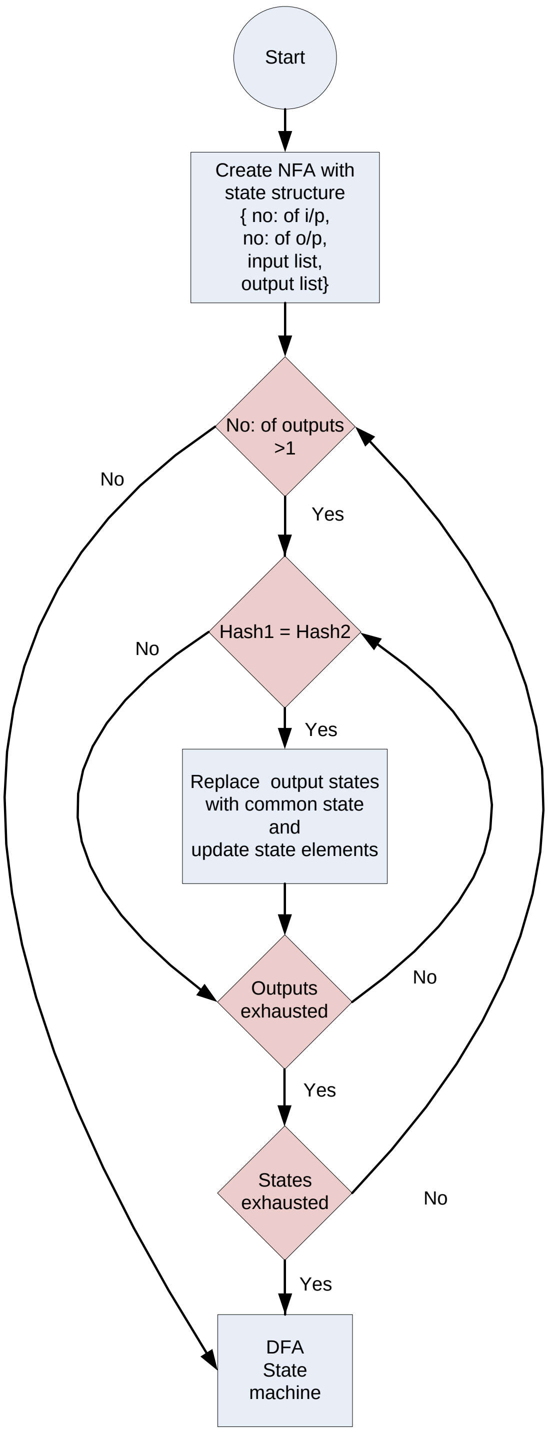

Start
Create NFA with state structure { no: of i/p, no: of o/p, input list, output list}
No: of outputs >1
No
Yes
Hash1 = Hash2
No
Yes
Replace output states with common state and update state elements
Outputs exhausted
No
Yes
States exhausted
No
Yes
DFA State machine

Figure 22: Powerset construction algorithm

In Figure 23, the states (1, 2, 3, 4, 5, 6, 7, 8, 9...) in the state machine represent the address locations of the hash memory. The hash values (0000 to 1111) are represented by the inputs to the state machine (x, y, z, a, b, c...).

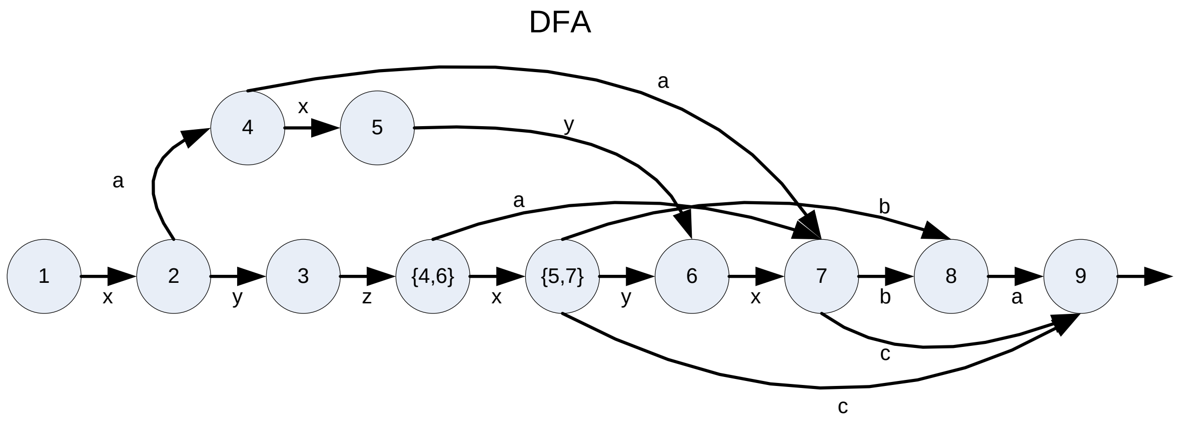


Figure 23: DFA state machine using powerset construction

The NFA to DFA transformation using powerset construction can result in states having more than two next-hop values (e.g. {5, 7}). The proposed security monitor architecture described in section 4.1 needs to be enhanced to support multiple next-hop address lookups. As part of the proposed work we will evaluate possible modifications to the CAM module to accommodate these multiple next-hop address lookups. This transformation is performed during offline analysis using software to convert the NFA state machine representation to a DFA representation.

4.4. Network processor core

The high level architecture of our network processor core is illustrated in Figure 24. The processor core consists of a 32-bit open source embedded Plasma processor [38] which is implemented in Verilog HDL and based on the MIPS architecture. The Plasma processor executes all MIPS user mode instructions except unaligned load and store instructions. The network processor core has a memory unit for storing program binaries and for storing data during program execution. It also has packet buffers to store the incoming network packet data.

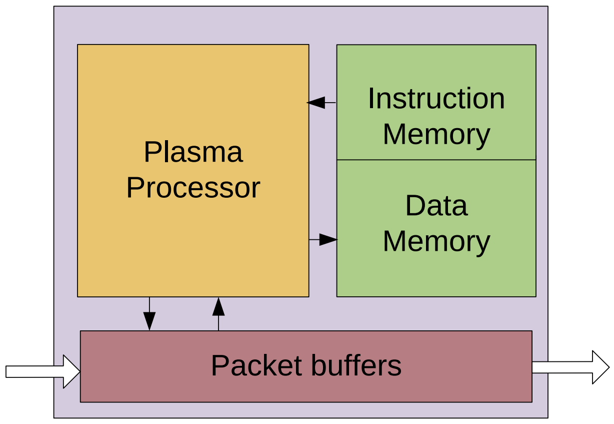


Figure 24: Network processor core

The network processor core integrated with a single security monitor system is shown in Figure 25. The security monitor keeps track of the instructions executed by the processor core. A reset signal is generated when a malicious behavior is detected.

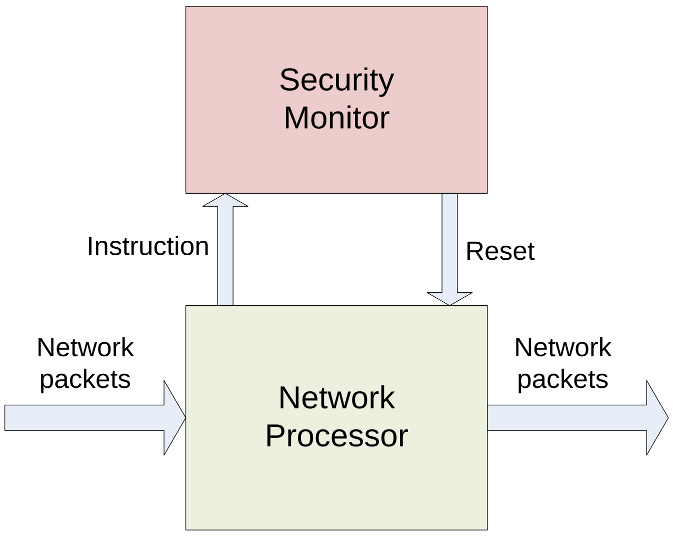


Figure 25: Network processor integrated with security monitor

4.5. Network processor architecture

A high level overview of the single-core network processor system with the security monitor incorporated in the Altera DE4 NetFPGA pipeline is illustrated in Figure 26. The packets arrive at the four Ethernet ports on DE4 board and are inserted into input queues (MAC RX Q). The input arbiter forwards these packets to the flow classification module. The flow classifier module assigns the incoming packet data to the network processor core. The instructions executed by the processor core are input to the monitor subsystem simultaneously. The security monitor can generate reset signal to the network processor core. The processed packets are forwarded by the output arbiter to the output queues. The packets output from the processor system are forwarded by the output queues to the corresponding MAC transmit ports (MAC TX Q).

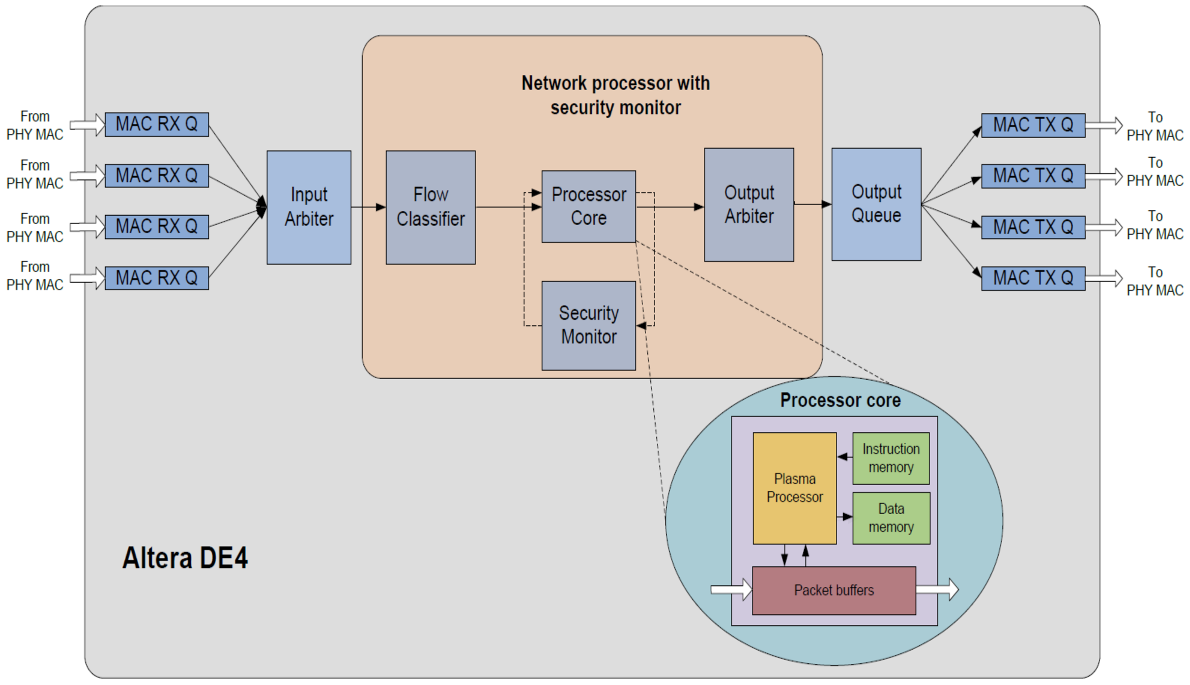


Figure 26: Single-core network processor system with security monitor in Altera DE4 NetFPGA pipeline.

4.6. Von Neumann versus Harvard architecture

The MIPS plasma processor used in our proposed network processor design utilizes a von Neumann [46] memory architecture, as illustrated in Figure 27. In a von Neumann architecture, a single physical memory is shared by both code and data. The processor does not make any distinction between whether the data or code is read or written. A memory interface arbitrates the memory access between the instruction read and data access.

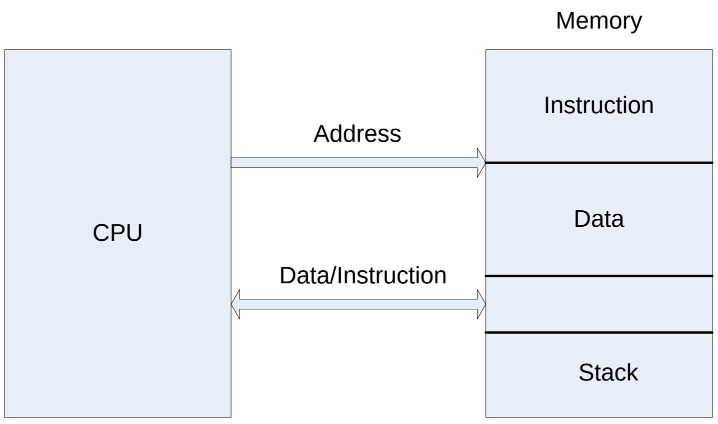


Figure 27: von Neumann architecture

This implementation style of von Neumann architecture makes it inherently vulnerable to code injection [48] attacks. A typical code injection attack is illustrated in Figure 28. The processor keeps track of the instructions it executes using the program counter. In a code injection attack, an attacker initially injects a malicious code into the processor's address space and directs the program counter to the address space where the malicious code resides. Attackers often employ different memory error techniques like stack overflow [52], format string vulnerability [53] and integer overflow [54] to trigger code injection attacks.

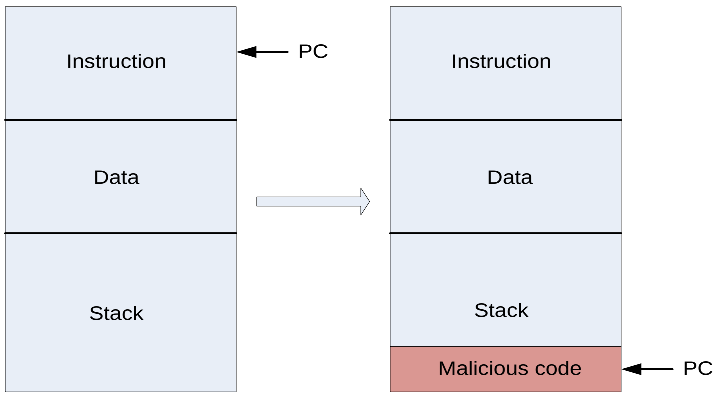


Figure 28: Code injection attack

In a Harvard architecture [47], the code and data are placed in separate physical address spaces. The Harvard memory architecture is illustrated in Figure 29. Separate buses provide instruction and data access, with each potentially having different word widths, timing and memory address structures. The processor can perform both the instruction read and the data memory access at the same time. The instructions are usually stored in read only memory while data is stored in read-write memory. Since a program counter cannot point to addresses in the data memory, code injection attacks are difficult to perform in a Harvard memory architecture. Even if an attacker successfully writes a malicious code in the stack, it will not be executed.

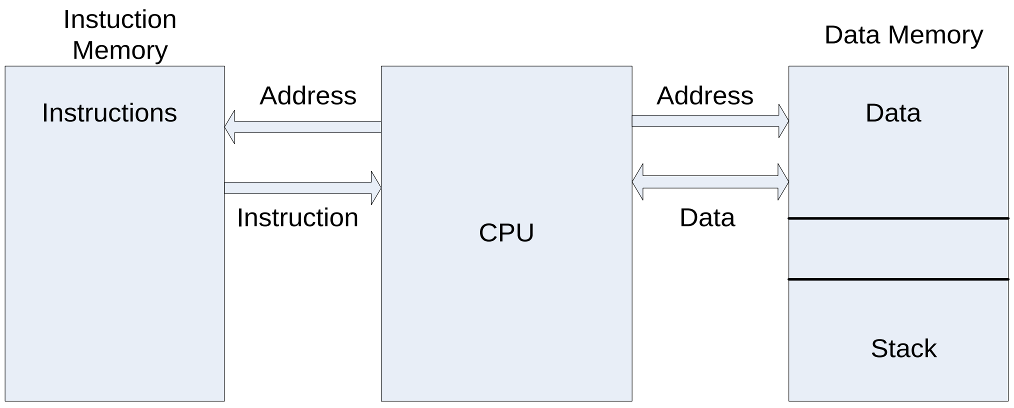


Figure 29: Harvard architecture

Even though general memory error techniques (integer overflow, heap overflow etc.) cannot be used to generate code injection attacks, Francillon et al. [49] demonstrated that code injection attacks are still feasible on Harvard architecture processors using return-oriented programming technique [55]. Here an attacker sends several specifically crafted packets to build a malicious stack one byte at a time. Once the stack is built, the attacker sends another specifically crafted packet that copies the malware to program memory using a return-oriented programming technique.

As part of the proposed work we evaluated a possible attack on a Harvard architecture and the ability of our proposed security monitor to detect it. The memory architecture of the plasma processor was modified to have separate instruction and data memory. The instruction memory was made read-only while data memory was made read-write.

CHAPTER 5

EXPERIMENTAL APPROACH

This chapter describes the experimental approach used for testing the prototype system. In the first section, we describe how a Harvard architecture attack can be constructed for the networking environment. The later section discusses the experimental setup, implemented utilizing Altera DE4 NetFPGA infrastructure. Finally, we outline the different evaluation metrics used for verifying the prototype system functionality.

5.1. Network attack generation

In this section, we describe how a Harvard memory architecture attack can be constructed for the networking environment and how our monitor can detect it. Figure 31 shows a portion of congestion management protocol (CM) and IPV4 packet forwarding application used to build an attack on the network processor system. The network attack used for testing the prototype system functionality exploits a simple integer overflow vulnerability of the congestion management [40] protocol application. A congestion management protocol inserts a custom protocol header in the packet header space between IP header and UDP header as illustrated in Figure 30.

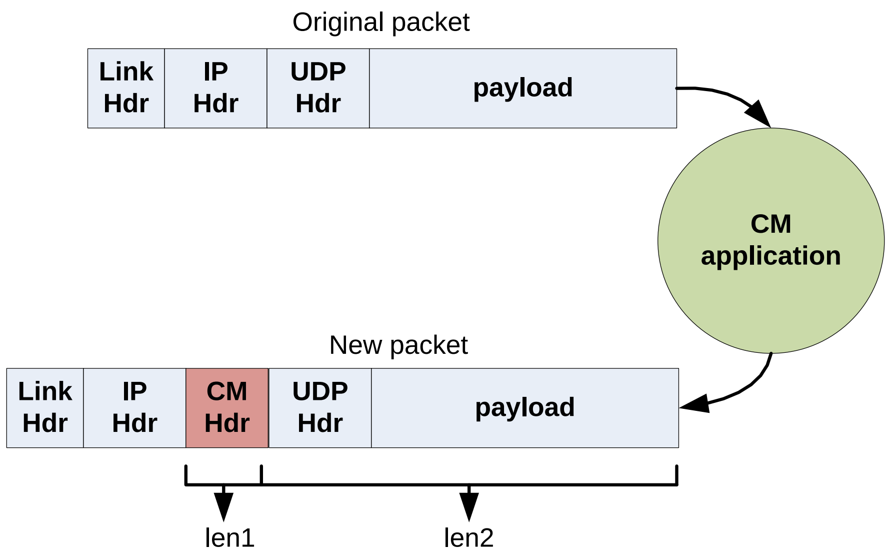
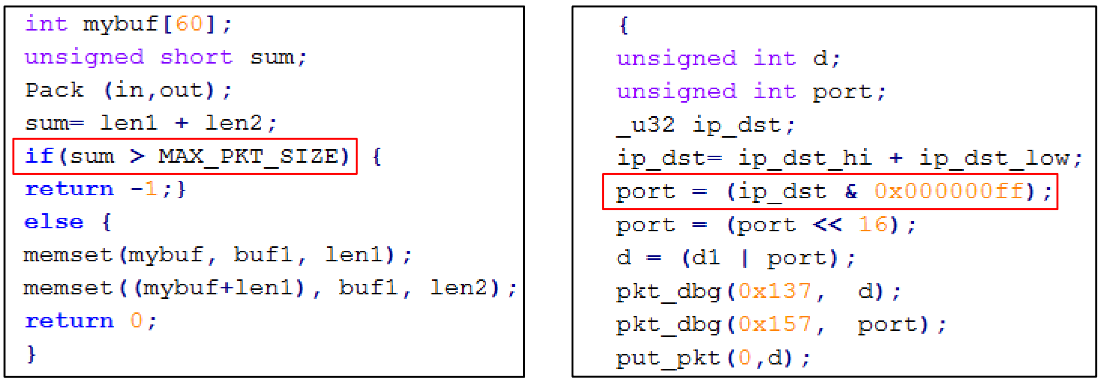


Figure 30: Congestion management header insertion

The application during this process needs to make sure the new packet size (*len1* + *len2*) does not exceed the maximum datagram length. In certain cases, this maximum packet size check can be exploited to create an integer overflow.



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

CM protocol

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

IPV4 application

Figure 31: Integer overflow vulnerable code

The variable *sum* is of type *unsigned short*. The CM application uses this variable to check whether the packet size (*sum* = *len1* + *len2*) after inserting the custom header has exceeded the maximum packet size limit (*sum* > *MAX_PKT_SIZE*). Any packet which satisfies the size check is then copied to processor data memory. However, an attacker can send a carefully crafted malformed UDP packet that can trigger an integer overflow. For example, an attack packet with malformed UDP length field (16 bit value 0xfffe (decimal value 65534)) will pass the maximum packet size check (since 65334 + 12 = 10, due to integer overflow). This will result in 65334 bytes of packet data to be copied to the processor memory space.

The packet payload of the attack packet is crafted in such a way that the return address is overwritten to direct the control flow to the IPV4 packet forwarding application (which is the library code on the processor) and the value of *ip_dst_low* field is 0xff. The port information gets updated with this value (the boxed instruction in the IPV4 code), forwarding the attack packet to all the outgoing ports and then crashing the processor system. As a result, the attack packet gets forwarded to all outgoing interfaces before the system crashes, thus propagating the attack through the network.

5.2. Experimental Setup

The test topology that will be used to verify the performance of our monitoring system is shown in Figure 32.

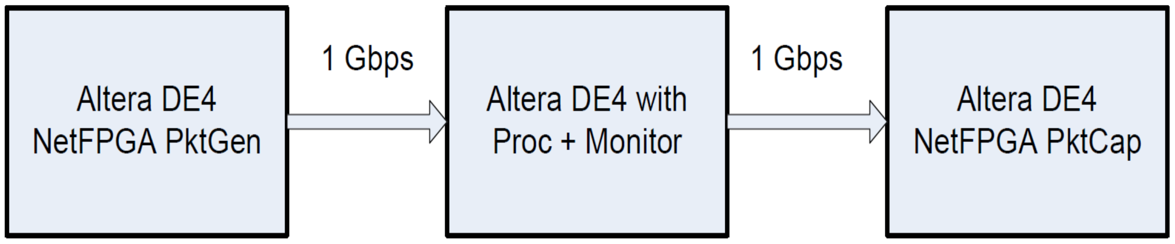

Altera DE4 NetFPGA PktGen
1 Gbps
Altera DE4 with Proc + Monitor
1 Gbps
Altera DE4 NetFPGA PktCap

Figure 32: Test topology

Altera DE4 packet generator is used to generate network packets, and to capture packets forwarded from the prototype system. The packet generator tool allows for customizing the size, the number of iterations, and the throughput rate for the test packet. The packet generator code is downloaded to one Altera DE4 board. The single core network processor system with security monitor, integrated along with the Altera DE4 NetFPGA packet generator pipeline is downloaded to another DE4 board. Ethernet MAC-PHY registers are configured through the JTAG cable. The experimental test setup is illustrated in Figure 33.

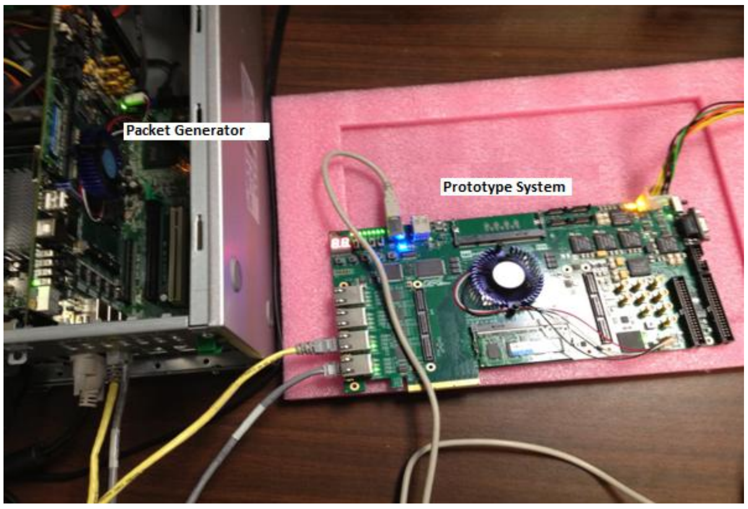


Figure 33: Experimental test setup

5.3. Evaluation metrics

The prototype system is tested in simulation using a ModelSim-Altera simulator [41], and in hardware using an Altera Signal-tap logic generator [56]. The different evaluation metrics for verifying the prototype system are listed below.

1. **Throughput performance**: Using IPV4 packet forwarding application [39], the single-core network processor system without that security monitor is tested for throughput performance for different packet sizes. The single core network processor system with security monitor illustrated in Figure 26 is tested for throughput performance using the attack model described in section 5.1.
2. **Attack Detection**: The prototype system described in Figure 26 is tested for attack detection capability, using the attack model mentioned in section 5.1. The security monitor system should detect any unintended processor behaviour and trigger appropriate recovery mechanisms to the processor.
3. **Resource overhead**: The resource utilization of the security monitor system is evaluated using a diverse set of network applications, as explained in next chapter. The resource savings facilitated by the proposed security approach over existing monitoring schemes is estimated.

CHAPTER 6

BENCHMARKS AND EXPERIMENTAL RESULTS

In this chapter we discuss the benchmarks used for testing the network processor system and the results of the experiments performed on the proposed security monitor architecture.

6.1. Evaluation benchmarks

Network workloads can be logically divided into data plane workloads and control plane workloads. The data plane is where data traffic is handled using actions such as packet forwarding, packet dropping, and encapsulation. The control plane handles complex packet management tasks like flow management, signaling, and routing updates. Control plane operations are usually less time critical, while data plane operations take place in real time on the network data path. Although network processors mostly target data plane applications, they are equally applicable to control plane operations. NpBench [43] is a benchmark suite targeting modern network processor applications. The benchmark applications are categorized into three specific functional groups - traffic management and quality of service group (TQG), security and media processing group (SMG) and packet processing group (PPG). The applications in these groups belong to either the data plane, control plane or both. The TQG benchmark falls in the category of both control plane and data plane processing, and includes applications related to routing, scheduling, switching, signaling and quality of service. The SMG benchmark is related to security applications like firewalls, admission control, encryption algorithms and media processing applications like media trans-coding. The PPG benchmark includes data plane processing applications like IP packet fragmentation, packet marking, editing and classification. The proposed network processor architecture will be evaluated using these diverse benchmark applications. Table 2 summarizes the different benchmarks provided by NpBench.

Group	**Applications**	**Data plane**	**Control plane**
TQG	Routing	X	X
	Scheduling	X	X
	Content-based Switching	X	X
	Weighted pair queuing	X	X
	Traffic shaping	X	X
	Load Balancing	X	X
	VLAN		X
	MPLS	X	X
SMG	Block cipher algorithm	X	
	Message cipher algorithm	X	
	Firewall application	X	X
	IPSec	X	X
	Virtual private network	X	X
	Public encryption	X	
	Usage-based accounting	X	X
	H.323	X	
	Media transcoding	X	X
	Duplicate data suppression	X	
PPG	IP-packet fragmentation	X	
	Packet encapsulation	X	
	Packet marking/editing	X	
	Packet classification	X	
	Checksum calculation	X	

Table 2: NpBench Benchmark applications [43].

6.2. Experimental results

6.2.1. Attack Detection

This section explains the experiments performed to test the ability of our proposed security monitoring system to detect and recover from an attack. We observed the security monitor operation in simulation using the ModelSim-Altera simulator [41], and in hardware using an Altera Signal-tap logic generator [56].

6.2.1.1. Network processor without security monitor

We initially tested the single-core network processor operation without the security monitor system when the attack described in section 5.1 is implemented. Figure 34 shows the simulation results for the behavior of the processor system. The attack packet was received through MAC port Rx0, and then forwarded to the network processor. The processor then forwards the attack packet to all the outgoing ports of the router and then crashes the router. This behavior was also verified in hardware.

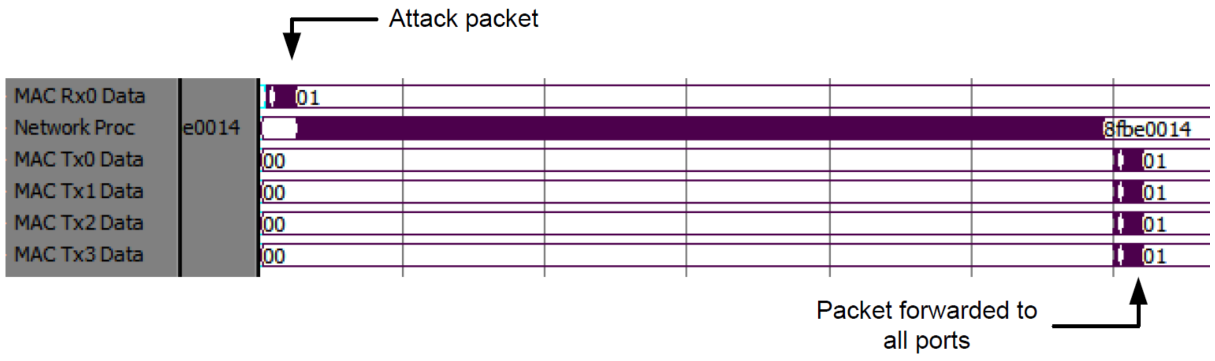


Figure 34: Simulation waveform showing attack packet propagation in the network processor system.

6.2.1.2. Network processor with security monitor

We then repeated the previous experiment after including the security monitor as illustrated in Figure 26. Figure 35 shows the simulation results for the behavior of the network processor system when an attack packet and normal packet are sent simultaneously.

After the monitor was included, the attack packet was successfully identified, the network processor was reset, and subsequent normal packets were routed successfully.

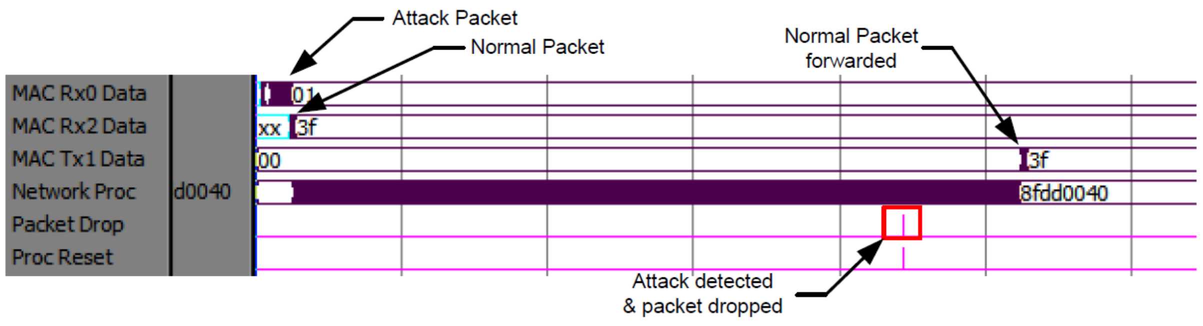


Figure 35: Simulation waveform showing the identification of the attack packet and successful forwarding of the subsequent packet.

6.2.2. Throughput performance

This section explains the experiments performed to measure the throughput of our proposed network processor system. The experimental setup mentioned in section 5.2 was implemented to perform these measurements.

6.2.2.1. Single-core network processor throughput performance

The single-core network processor system illustrated in Figure 26 was implemented, without the security monitor, on the Altera DE4 NetFPGA platform. Using a standard IPV4 packet forwarding application in the processor core, the throughput performance of the single-core system was tested. Network packets of different packet sizes were generated from the Altera DE4 packet generator, and send through the 1Gbps MAC ports of the Altera DE4 board. The forwarded packets were received back at the packet generator and the prototype system’s transmit-receive statistics were measured. The resulting throughput performance is illustrated in Figure 36.

The throughput of our network processor system improves as the packet size increases. The packet forwarding application works by comparing destination IP address in each packet header with IP address values stored in processor memory to select an output port. A reduction in packet size increases the per packet processing operation, and thus reduces the overall throughput performance.

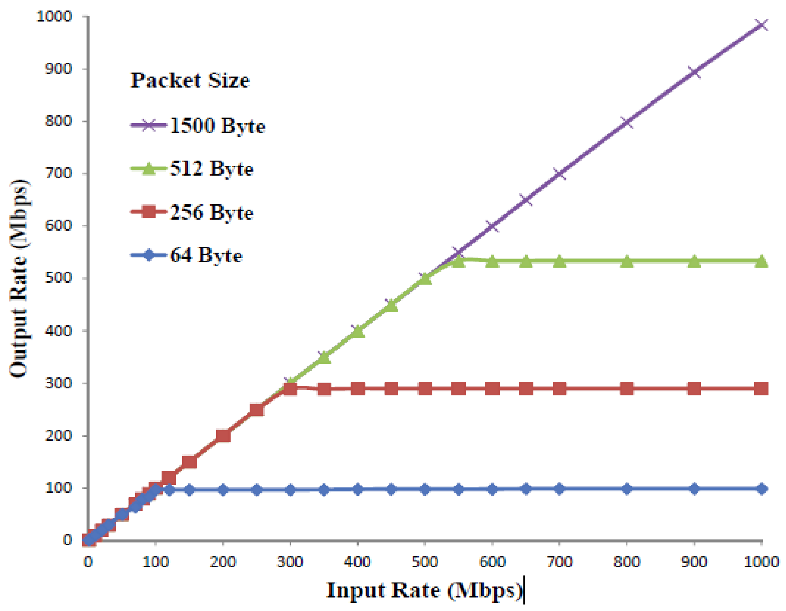
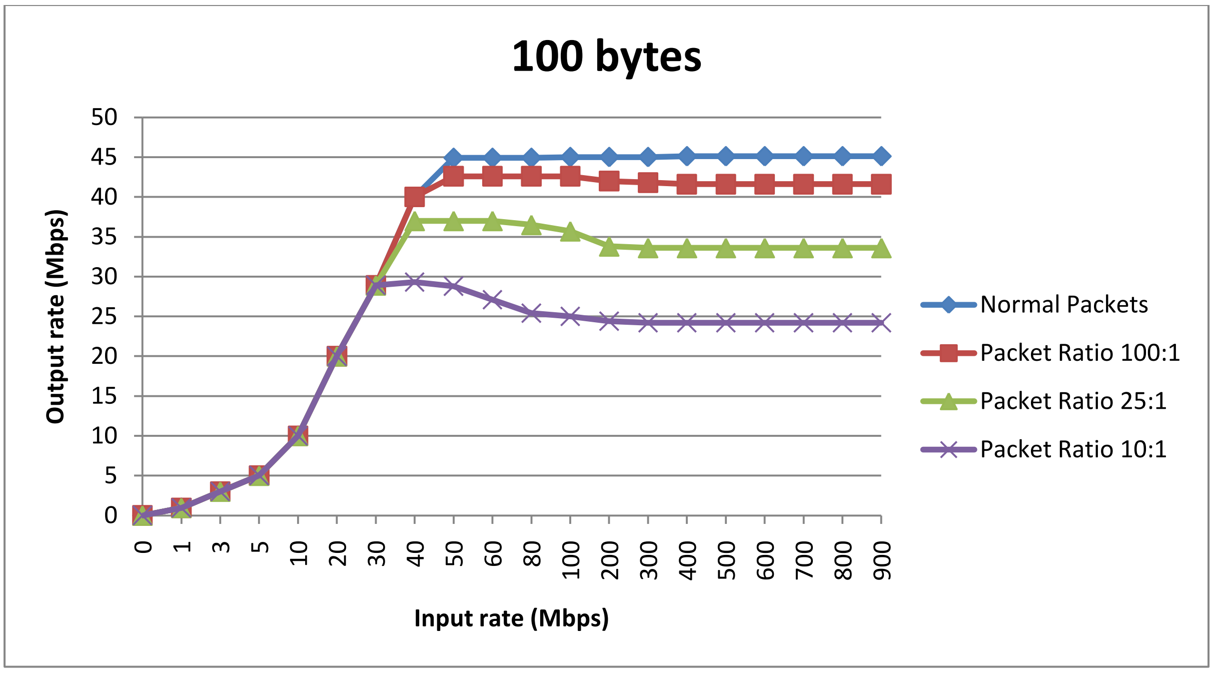


Figure 36: Single-core network processor throughput performance (IPV4)

6.2.2.2. Single-core network processor throughput performance under attack

In this experiment, we evaluate the throughput performance of our single-core network processor system illustrated in Figure 26 when attack packets are sent simultaneously along with normal packets. Normal packets are received at one Ethernet-MAC port (Rx0) of the network processor system; while attack packets are received simultaneously at another receive port (Rx1). Both normal packets and attack packets are generated at the same rate from the packet generator system. The forwarded packets are received back at the packet generator and the throughput is measured. Figure 38 shows the throughput performance of the network processor system for two different packet sizes for varying ratios of normal packets to attack packets. The vulnerable application shown in Figure 31 was used for testing purpose. When no attack packets are send the throughput of the network processor system increases and reaches a maximum. When attack packets are included the throughput reaches a maximum, and then decreases slightly before settling down.



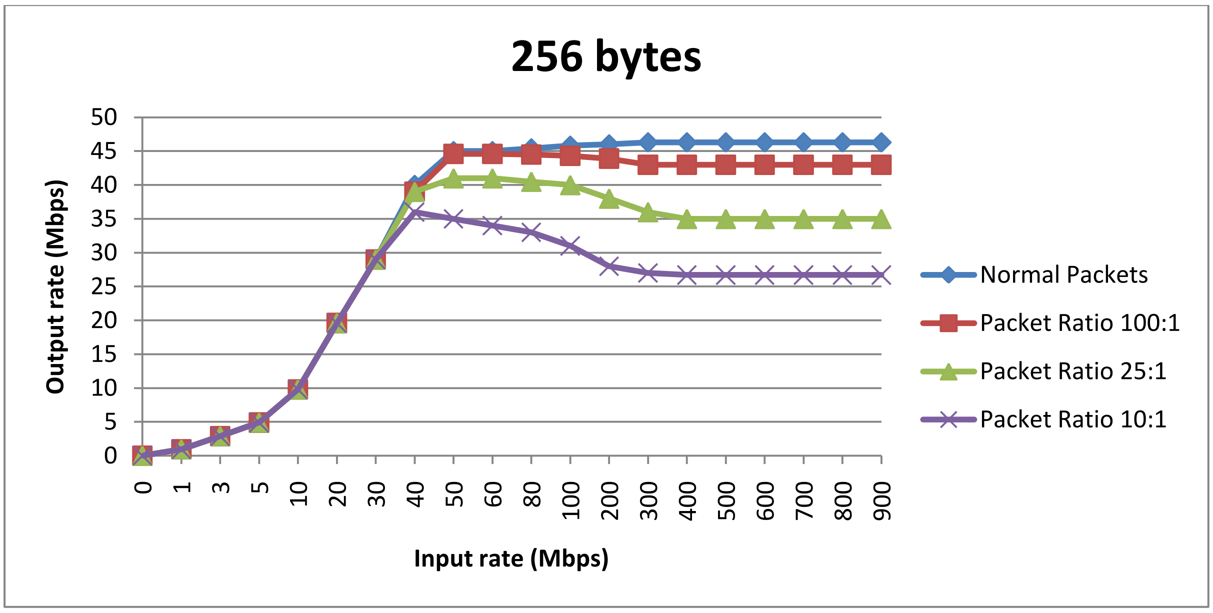


Figure 37: Single-core network processor throughput performance with security monitor under attack packets

As we increase the ratio of the attack packets sent to the processor system, the overall throughput of the system is reduced. This effect occurs because whenever an attack packet is detected, the security monitor generates a reset signal. The network processor and the packet buffer are reset before the processor can continue with the next packet.

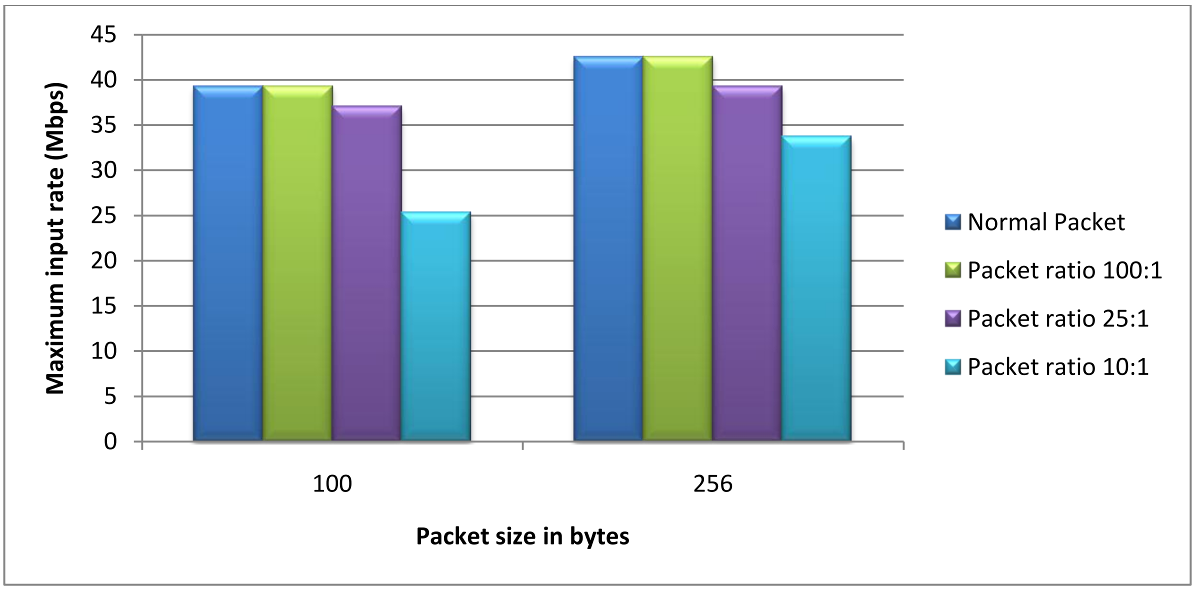


Figure 38: Maximum possible input rate for all normal packets to be forwarded successfully

Figure 38 shows the maximum rate at which packets can be received by the network processor system so that all normal packets are forwarded successfully. Only attack packets are dropped by the processor, while all the regular packets are forwarded successfully. The latency for the 100-byte packet is 24us while for 256-byte packet the latency is 104us.

When testing the throughput performance using larger packet sizes (512 bytes, 1500 bytes), the network processor does not forward the packets and the packets are lost. The reason for this packet loss could be either of the following two cases below.

1. For the application used for testing throughput performance under attack (CM protocol), the processor copies the packets to the data memory before forwarding the packets. The data memory may not be sufficient for processing large packets. So we may need to look at different applications to overcome this problem.

2. Packet generator sends packets to the network processor with a small inter packet delay. Since CM application operates on the entire packet, the inter packet delay becomes insufficient (as packet size increases) for the single core processor to effectively route the

packets, resulting in packet loss. A solution for this could be multi-core network processors.

6.2.3. Resource Utilization

This section explains the different resource utilization details of our proposed network processor system. The synthesis results were provided by Altera Quartus tool while the DFA memory resource utilization details were provided by the offline analysis tool explained in section 4.2.

6.2.3.1. Hash size and memory requirement

We initially explored the relation between hash size, DFA states, and state machine memory requirement for three different hash sizes. Table 3 shows the relation between hash size and DFA states. As hash size increases the probability of hash collision decreases ($1/2^h$, where h is the hash size), which reduces the number of DFA states. For example, for a three bit hash, if the sum of instruction bits for the multiple active states on the output of a control flow instruction in an NFA differ by a value of 8, we combine them to form a DFA state (e.g., if the sums are 2 and 10, then modulo 8 of both values is 2). When we move to higher hash sizes, this hash collision is avoided and the DFA states get reduced. For the benchmarks tested, most of the NFA states combined to form DFA have the same hash value, so they remain even when we increase the hash size. For the few states where hash collisions are avoided, we get a reduction in DFA states by increasing the hash size.

Table 4 shows the relationship between hash size and state machine memory. Increasing the hash size increases the size of memory entries exponentially since the ***valid hash values on outgoing edges*** field depends on hash size as explained in section 4.1. The memory overhead increases by 42% as we move from three bit hash to four bit hash and by 56% as we move from four bit hash to five bit hash. Having a larger hash size reduces the number of DFA states (probability of hash collision reduces) when the benchmark has a potentially large number of control flow instructions and memory accesses. We selected a four bit hash for our proposed security monitoring system since it provides sufficiently low collision probability (0.0625) without much memory overhead.

Benchmarks	**NFA states**	**DFA states**		
		Three bit hash	Four bit hash	Five bit hash
frag	573	594	592	591
red	802	808	808	807
ssld	828	836	836	833
wfq	905	921	921	918
mtc	2427	2460	2460	2459

Table 3: Hash size versus DFA states

Benchmarks	**NFA states**	**Three bit hash**		**Four bit hash**		**Five bit hash**	
		Mem. entries	Mem. bits	Mem. entries	Mem. bits	Mem. entries	Mem. bits
frag	573	629	13209	627	18810	626	29422
red	802	857	17997	857	25710	854	40138
ssld	828	879	18459	879	26340	871	40937
wfq	905	980	20580	978	29340	969	45543
mtc	2427	2584	59432	2584	82688	2581	126469

Table 4: Hash size versus state machine memory

6.2.3.2. DFA versus NFA monitoring graph comparison

The results of generating instruction-level monitoring graphs for both our approach and the previously mentioned approach in section 3.1 are illustrated in Table 5. The number of entries in the state machine memory is shown in the *Mem. entries* column. A clear benefit of our proposed approach is speed. In all cases, only one access to the monitor memory is required for any benchmark (including the five shown here). The previous NFA-based approach requires up to three memory accesses for the benchmarks tested and potentially up to 16 for other benchmarks. The conversion from NFA to a DFA does incur a memory overhead of 7.7% on average for the benchmarks.

		Chasaki[2]		**Proposed system**		
Net. application	No: of instructions	NFA states	Max mem. accesses	DFA states	Mem. entries	Mem. overhead
frag	573	573	3	592	627	9.4%
red	802	802	2	808	857	6.8%
ssld	828	828	3	836	879	6.2%
wfq	905	905	2	921	978	8.0%
mtc	2427	2427	3	2460	2584	6.4%

Table 5: Evaluation of monitoring approaches for our proposed DFA approach and a previous NFA approach.

6.2.3.3. Monitoring speed and resource utilization

The network processor system along with the security monitoring module was successfully implemented on the DE4 platform. The lookup table (LUT), flip flop (FF), and memory resources required for the single network processor core, monitor, and other interface circuitry for the router (e.g. buffers, input arbiter, queuing control) are shown in Table 6. The NP memory includes space for up to 4096 monitor memory entries. All circuitry operated at 125 MHz, the same clock speed for the system without the monitor.

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

Table 6: Resource utilization for single core network processor system

The lookup table (LUT), flip flop (FF), and memory resources required for both our approach and the previously mentioned approach in section 3.1 are illustrated in Table 7. The security monitor memory includes space for up to 4096 memory entries. The DFA based monitor has the advantage of evaluating 16 next states during every instruction cycle.

Resources	NFA based security monitor (basic blocks)	DFA based security monitor (4-bit hash)
LUTs	40	140
FFs	35	26
Memory bits	49,664	131,072

Table 7: Resource utilization comparison between NFA based and DFA based security monitors

This chapter summarized the evaluation benchmarks and the experimental results performed to test the functionality of our proposed network processor with the security monitoring system. Next chapter concludes the thesis and provides future directions.

CHAPTER 7

CONCLUSION AND FUTURE WORK

The thesis has outlined a new network processor architecture with a high-performance security monitor for detecting in-network attacks. The network processor requires only a single memory lookup per network processor instruction. This single memory lookup is maintained regardless of the complexity of the network processor program using NFA-to-DFA translation of the monitoring graph. Our monitor, which tracks individual processor instructions, has been verified in hardware using a network processor with a Harvard architecture. The presence of monitoring does not slow down the processor operation since it is performed outside the operational paths of the processor.

The network processor with security monitoring system was implemented as part of the Altera DE4 NetFPGA infrastructure. Results show that the throughput of the single-core network processor system increases as the packet size increases. The network processor was able to achieve line rate forwarding at packet size of 1500 bytes for IPV4 packet forwarding application. We demonstrated the ability of our security monitor system to detect and recover from network attacks without affecting the performance of the processor. Only the attack packets get dropped, while the regular packets are forwarded successfully. We illustrated the benefits of our security monitoring system over existing techniques in both memory access and resource utilization. Our evaluation of hash size to memory resource requirement showed that a four bit hash size provides sufficiently less collision probability without increasing the memory overhead.

In the future, we plan to evaluate our monitoring approach using a multi-core network processor. We also plan to look into the possibility of sharing monitoring logic between different processor cores when they execute the same application. We hope that the developed security monitor framework will facilitate rapid design space exploration of security monitor architectures for network processor systems.

BIBLIOGRAPHY

[1] http://en.wikipedia.org/wiki/History_of_the_Internet

[2] D. Chasaki and T. Wolf, “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.

[3] S. Mao and T. Wolf, “Hardware support for secure processing in embedded systems,” *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 847–854, Jun. 2010.

[4] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, “Secure embedded processing through hardware-assisted runtime monitoring,” in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE’05)*, Munich, Germany, Mar. 2005, pp. 178–183.

[5] Q. Wu and T. Wolf, “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.

[6] http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton

[7] Bremler-Barr, D. Hay and Y. Koral, "CompactDFA: Generic State Machine Compression for Scalable Pattern Matching," in Proc of IEEE (INFOCOM), vol., no., pp.1-9, 14-19 March 2010

[8] http://www.altera.com/products/fpga.html

[9] http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.pdf

[10] Q. Wu, D. Chasaki, and T. Wolf, “Implementation of a simplified network processor,” in *Proc. of IEEE International Conference on High Performance Switching and Routing (HPSR)*, Richardson, TX, June 2010

[11] T. Wolf and M.A. Franklin, “Performance models for network processor design," *IEEE Transactions on Parallel and Distributed Systems*, vol.17, no.6, pp.548-561, June 2006.

[12] Papaefstathiou, G. Kornaros and N. Zervos, "Software processing performance in network processors," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE’04),* vol.3, no., pp. 186- 191 Vol.3, 16-20 Feb. 2004.

[13] C. Tzi-Cker and P. Pradhan, "Cache memory design for network processors," in *Proc. of Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, vol., no., pp.409-418, 2000.

[14] X. Hong and W. Di, "A Component Model for Network Processor Based System," in *IEEE/ACS International Conference on Computer Systems and Applications (AICCSA '07)*, vol., no., pp.47-50, 13-16 May 2007.

[15] T.Wolf, W. Ning and T. Chia-Hui, "Design considerations for network processor operating systems," in *Symposium on Architecture for networking and communications systems (ANCS '05)*, vol., no., pp.71-80, 26-28 Oct. 2005.

[16] D. Geer, “Malicious bots threaten network security,” *Computer*, vol. 38, no. 1, pp. 18–20, 2005.

[17] D. Moore, C. Shannon, and J. Brown, “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.

[18] http://technet.microsoft.com/en-us/library/cc959354.aspx

[19] Z. Shi, "The automaton modeling of typical network attacks," in *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, vol.1, no., pp.243-246, 10-12 June 2011.

[20] P. Owezarski, "On the impact of DoS attacks on Internet traffic characteristics and QoS," in *Proc. of 14th International Conference on Computer Communications and Networks (ICCCN '05)*, vol., no., pp. 269- 274, 17-19 Oct. 2005.

[21] Y. Xin, C. Wei and W. Yantao, "The research of firewall technology in computer network security," in *Asia-Pacific Conference on Computational Intelligence and Industrial Applications (PACIIA '09)*, vol.2, no., pp.421-424, 28-29 Nov. 2009.

[22] G. Manimaran, "Internet infrastructure security," in *Proc. of 12th Annual IEEE Symposium on High Performance Interconnects (CONECT '04)*, vol., no., pp. 109, 25-27 Aug. 2004.

[23] M. Garuba, L. Chunmei and D. Fraites, "Intrusion Techniques: Comparative Study of Network Intrusion Detection Systems," in *Fifth International Conference on Information Technology: New Generations (ITNG '08)*, vol., no., pp.592-598, 7-9 April 2008.

[24] “NetFPGA,” http://netfpga.org/

[25] http://www.xilinx.com/univ/XUPV2P/Documentation/XUPV2P_User_Guide.pdf

[26] http://www.altera.com/education/univ/materials/boards/de4/unv-de4-board.html

[27] http://www.altera.com/devices/fpga/stratix-fpgas/stratix-iv/stxiv-index.jsp

[28] D. Srinivasan and F. Wu-chang, "Performance analysis of multi-dimensional packet classification on programmable network processors," in *29th Annual IEEE International Conference on Local Computer Networks*, vol., no., pp. 360- 367, 16-18 Nov. 2004.

[29] T. Wolf, "Challenges and Applications for Network-Processor-Based Programmable Routers," in *IEEE Sarnoff Symposium*, vol., no., pp.1-4, 27-28 March 2006.

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

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

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

[33] D. Chasaki, W. Qiang and T. Wolf, "Attacks on Network Infrastructure," in *Proc. of 20th International Conference on Computer Communications and Networks (ICCCN)*, vol., no., pp.1-8, July 31 2011-Aug. 4 2011.

[34] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol.14, no.12, pp.1295-1308, Dec. 2006.

[35] M. Shufu and T. Wolf, "Hardware Support for Secure Processing in Embedded Systems," in *44th ACM/IEEE Design Automation Conference (DAC '07)*, vol., no., pp.483-488, 4-8 June 2007.

[36] T. Wolf and R. Tessier, "Design of a Secure Router System for Next-Generation Networks," in *Third International Conference on Network and System Security (NSS '09)*, vol., no., pp.52-59, 19-21 Oct. 2009.

[37] "AlteraDE4 NetFPGA," http://keb302.ecs.umass.edu/de4web/DE4_NetFPGA/

[38] "Plasma processor," http://opencores.org/project,plasma

[39] K.Ravindran, N. Satish, J. Yujia and K. Keutzer, "An FPGA-based soft multiprocessor system for IPv4 packet forwarding," in *International Conference on Field Programmable Logic and Applications (FPL '05)*, vol., no., pp. 487-492, 24-26 Aug. 2005.

[40] H. Balakrishnan, H.S. Rahul and S. Seshan, "An integrated congestion management architecture for internet hosts," in *Proc. of the conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '99)*, Cambridge, MA, pp. 175–187, Sept. 1999.

[41] "Using ModeSim-Altera in Quartus II design flow" http://home.eng.iastate.edu/~zzhang/courses/cpre581-f05/resources/modelsim.pdf

[42] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization (WWC-4)*, vol., no., pp. 3- 14, 2 Dec. 2001.

[43] B.K. Lee and L.K. John, "NpBench: a benchmark suite for control plane and data plane applications for network processors," in *Proc of 21st International Conference on Computer Design*, vol., no., pp. 226- 233, 13-15 Oct. 2003.

[44] "NetFPGA reference router," http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/Guide#Walkthrough_the_Reference_Design

[45] "NetFPGA packet generator," http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/PacketGenerator

[46] L. Dadda, "The evolution of computer architectures," in *Proc. of 5th Annual European Computer Conference on Advanced Computer Technology, Reliable Systems and Applications (CompEuro '91)*, vol., no., pp. 9-16, 13-16 May 1991.

[47] "Architecture," http://ww1.microchip.com/downloads/en/devicedoc/31004a.pdf

[48] R. Riley, J. Xuxian and X. Dongyan, "An Architectural Approach to Preventing Code Injection Attacks," in *IEEE Transactions on Dependable and Secure Computing*, vol.7, no.4, pp.351-365, Oct.-Dec. 2010.

[49] A. Francillon and C. Castelluccia, "Code injection attacks on harvard architecture devices," in *Proc. of ACM CCS*, pp. 15-26, 2008.

[50] "Powerset construction," http://en.wikipedia.org/wiki/Powerset_construction

[51] "Altera TriMatrix Embedded Memory Blocks in Stratix IV Devices", http://www.altera.com/literature/hb/stratix-iv/stx4_siv51003.pdf

[52] "Stack overflow attacks", http://www.techrepublic.com/blog/security/basics-of-stack-smashing-attacks-and-defenses-against-them/2755

[53] "Format string attacks", http://www.defcon.org/images/defcon-18/dc-18-presentations/Haas/DEFCON-18-Haas-Adv-Format-String-Attacks.pdf

[54] G. Qijun, F. Christopher and N. Rizwan, "A study of self-propagating mal-packets in sensor networks: Attacks and defences," in *Computers and Security*, vol. 30, pp. 13-27, 2011.

[55] R. Roemer, E. Buchanan, H. Shacham and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," in *ACM Transactions on Information and Systems Security*, vol. 15, no. 1, Mar. 2012

[56] "Design debugging using Altera Signal Tap II Logic Analyzer," http://www.altera.com/literature/hb/qts/qts_qii53009.pdf