Dynamic Hardware Monitors for Network Processor Protection

Kekai Hu, Harikrishnan Kumarapillai Chandrikakutty, Zachary Goodman,
Russell Tessier, *Senior Member, IEEE*, and Tilman Wolf, *Senior Member, IEEE*

Abstract—The importance of the Internet for society is increasing. To ensure a functional Internet, its routers need to operate correctly. However, the need for router flexibility has led to the use of software-programmable network processors in routers, which exposes these systems to data plane attacks. Recently, hardware monitors have been introduced into network processors to verify the expected behavior of processor cores at run time. If instruction-level execution deviates from the expected sequence, an attack is identified, triggering processor core recovery efforts. In this manuscript, we describe a scalable network processor monitoring system that supports the reallocation of hardware monitors to processor cores in response to workload changes. The scalability of our monitoring architecture is demonstrated using theoretical models, simulation, and router system-level experiments implemented on an FPGA-based hardware platform. For a system with four processor cores and six monitors, the monitors result in a 6% logic and 38% memory bit overhead versus the processor's core logic and instruction storage. No slowdown of system throughput due to monitoring is reported.

Index Terms—network security, network infrastructure, data plane attack, hardware monitor, multicore processor, FPGA

1 Introduction

Network routers are core components of the Internet infrastructure. Routers implement the packet processing and forwarding operations that need to be performed on every packet that is transmitted through the network. Typical processing tasks on routers include security checks, data filtering, and traffic statistics collection. These functions augment basic forwarding as required by the Internet Protocol (IP) [1]. Since network functions may be introduced dynamically, routers require the programmability offered by network processors (NPs) [2] as opposed to fixed-function application-specific integrated circuits (ASICs). Network processors often feature multiple software-programmable processor cores, which operate in parallel to achieve high throughput rates. As a result, NPs are the computing device of choice in the large fraction of contemporary network routers.

- *K. Hu, Z. Goodman, R. Tessier and T. Wolf are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, 01003 USA e-mail: (wolf@ecs.umass.edu).*
- *H. Chandrikakutty is with Juniper Networks, Chelmsford, MA, 01824 USA*

The programmability offered by network processors, while providing system-level adaptability, also exposes potential security weaknesses. Similar to network end-systems, such as general-purpose desktop and server computers, software-based network processors are vulnerable to remote attacks. These attacks can cause routers to exhibit unpredictable and malicious behavior. Recently, it has been shown that the functionality of a network processor could be modified by processing a single User Datagram Protocol (UDP) packet [3] in the data plane. In this attack, the network processor is reprogrammed to indefinitely retransmit the malicious packet to downstream routers. This self-propagating attack can be particularly difficult to control since it only requires data plane access (i.e., no access to the control plane is needed by the attacker). Thus, rapid identification of these malicious packet processing operations must take place on the network processor to prevent the attack from propagating and disabling the network.

Since network routers are embedded systems, their defense mechanisms are necessarily limited in extent. Network processor cores typically do not execute operating systems, thus anti-malware software is not suitable in this context. Additionally, network intrusion detection systems (e.g., Snort [4] or Bro [5]) are often only positioned on the ingress side of campus networks and thus do not protect the Internet core. In response to this need, hardware monitors for network processor cores have been introduced to provide runtime execution protection [6]. A hardware monitor operates in concert with an embedded network processor core to assess runtime behavior. If anomalous or unexpected operation is observed, the core can be reinitialized to avoid processing malicious code while continuing to process subsequent benign packets.

Most previous hardware monitor designs have focused on processors with a single processor core executing a single program or a program that changes very infrequently. This model is not effective for network processors, which contain core counts in the hundreds and exhibit frequent changes in packet processing objectives with changing traffic patterns [7]. As a result, new monitoring techniques for embedded multicores are needed to ensure adequate system protection. This

manuscript presents the architecture and in-circuit hardware evaluation of a Scalable Hardware Monitoring Grid (SHMG) to address multicore monitoring. A lightweight interconnection network between processor cores and monitors is dynamically configured to form monitoring connections in response to packet processing needs. In some cases multiple processor cores can share a single monitor, reducing memory overhead. Results developed using both analysis and simulation indicate that our monitoring approach is scalable for network processors containing hundreds of processing cores.

The specific contributions of our work are:

- The design of a scalable architecture for hardware monitors that can be used in a practical network processor system with a large number of processor cores.
- An algorithm that can dynamically allocate monitors to processor cores as application packet workloads change.
- A simulation and analysis of performance of the proposed design at runtime that considers the effects of dynamically assigning processors to monitors and the resulting resource contention.
- A prototype system implementation of a hardware monitoring system on an field-programmable gate array (FPGA) platform that illustrates the feasibility of our design and provides detailed resource requirement numbers. A system which includes a four-core network processor with six monitors has been deployed and tested.

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

This remainder of the manuscript is organized as follows. Section 2 discusses related work. We describe data plane attacks and potential defense mechanisms in detail in Section 3. Section 4 introduces the design of our Scalable Hardware Monitoring Grid, which is evaluated in Section 5. The details of our monitor resource allocation algorithm are described in Section 6. Results from a prototype implementation are presented in Section 7. Section 8 concludes this paper and offers directions for future work.

2 Related Work

Network processors are used in routers to implement standard IP forwarding functions as well as advanced functions related to performance, network management, flow-based operations, etc. [2]. Network processors use on the order of tens to low hundreds of parallel cores in a single multi-processor system-on-chip (MPSoC) configuration. Example devices include Cisco QuantumFlow [8], Cavium Octeon [9], and EZchip NP-5 [10] with data rates in the low hundreds of Gigabits per second.

Attacks on networking devices have been described in [11], but that work explored vulnerabilities in the *control plane*, where attacks aim to hack into the control interface of a router (e.g., IOS [12]). In more recent work, Chasaki and Wolf have described attacks on network processors through the *data plane* [3], where attackers merely need to send malformed data packets. In our work, we focus on the latter type of attack.

Since the processor cores of routers are very simple, there are not sufficient resources to run complex intrusion detection or anti-malware software. These resource constraints are similar to what has been encountered in the embedded system domain. Embedded systems (of which network processors are one class) exhibit a range of vulnerabilities [13], [14].

One defense technique for systems, where software defenses are not practical, is hardware monitoring. A hardware monitor operates in parallel with a processor core and verifies that the core operates within certain constraints (e.g., not accessing certain memory locations, executing certain sequences of instructions, etc.). Hardware monitoring has been studied extensively for embedded systems [15]–[17] and has also been proposed for use in network processors [6]. In our recent work, we describe a high-performance implementation of such a hardware monitoring system that can meet the throughput demands of a network processor with a single processing core [18].

What has been missing in the space of hardware monitoring for network processors is a system-level design of a comprehensive monitoring solution that can *support a large number of processor cores* and can *adapt to quickly changing workloads*. Since network processors have many processor cores, it is not practical to equip every core with a monitor that can handle any type of processing since this would lead to prohibitively expensive monitors. Instead, monitors need to be limited to handle one or a few processing tasks and adapt as the workload changes. Since network processors may experience highly dynamic workload changes based on changing traffic patterns [7], effective solutions for such an environment need to be developed.

This paper substantially extends our previous conference publication on network processor multicore monitoring [19]. We examine the system-level operation of monitoring in greater detail using a cycle-accurate multicore simulator and additional benchmark applications. This infrastructure is used to evaluate a new workload allocation algorithm that has been developed to dynamically assign applications and monitors to processor cores. Results obtained via simulation match values determined using an analytical model and using hardware implemented in an FPGA-based board in the laboratory. Additionally, in this extended version we consider both the power and energy consumption of monitoring in addition to area. Direct comparisons are made for these parameters between processor cores, monitors, and interfaces. Finally, we carefully quantify the time needed to

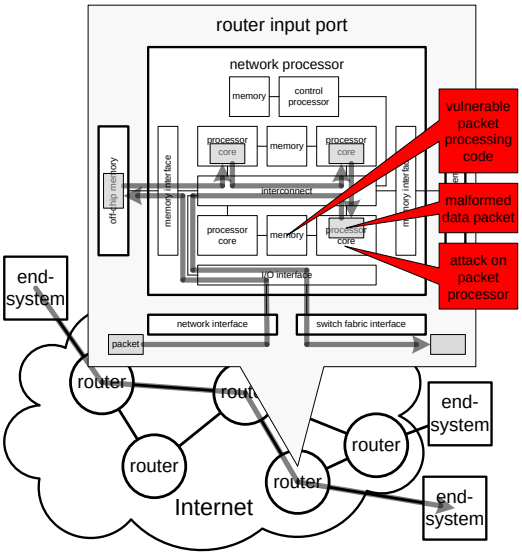

Fig. 1: Attack on a network processor.

load new monitoring information into a monitor versus the time needed to connect a processor to a previously-loaded monitor via programmable interconnect.

3 Data Plane Attacks and Defenses

To provide the necessary context for our Scalable Hardware Monitoring Grid, we briefly describe how network processors can be attacked in the data plane and how hardware monitors can be used to defend against these attacks.

3.1 Vulnerabilities in Networking Infrastructure

The typical system architecture and operation of a network processor is illustrated in Figure 1. Network processors are located at router ports, where they process traffic that is traversing the network.

Due to the very high data rates at the edge and the core of the network, network processors typically need to achieve throughput rates in the order of tens to hundreds of Gigabits per second. To provide the necessary processing performance, network processors are implemented as multi-processor systems-on-chip (MPSoC) with tens to hundreds of parallel processor cores. Each processor has access to local and shared memory and is connected through a global interconnect. Depending on the software configuration of the system, packets are dispatched to a single processor core for processing (run-to-completion processing) or passed between processor cores for different processing steps (pipelined processing). An on-chip control processor performs runtime management of processor core operation.

In order to fit such a large number of processor cores onto a single chip, each processor core can only use a small amount of chip real estate. Therefore, network processor cores are typically implemented as very simple reduced instruction set computer (RISC) cores with only a few kilobytes of instruction and data memory. These cores support a small number of hardware threads, but are not capable of running an operating system. Therefore, conventional software defenses used for workstation and server processors cannot be employed. Nevertheless, these cores are general-purpose processors and can be attacked just like more advanced processors on end-systems.

An attack scenario for network processors is illustrated in Figure 1. The premise for this attack is that the processing code on the network processor exhibits a vulnerability. It was shown in prior work that such a vulnerability can be introduced due to an uncaught integer overflow in an otherwise benign and fully functional packet processing function [3]. If a vulnerability in packet processing code is matched with a suitable attack packet (e.g., a malformed UDP packet), then an attack on a processor core can be launched. In the case of [3], the attack packet smashed the processor stack and led to the execution of code that was carried in the packet payload. The processor ended up re-transmitting the attack packet at full data rate on an outgoing link without recovering until the network processor was reset.

Launching a denial-of-service attack, such as in [3], can be done by using *a single packet* and can have more impact than conventional botnets, which are more complex to coordinate and are constrained by the access link bandwidth of the bots [20]. In addition, attacks on network processors have been shown both on systems that are based on von Neumann architecture [3], leading to arbitrary code execution, and on systems based on Harvard architecture [18], leading to return-to-libc attacks.

3.2 Defense Mechanisms With Hardware Monitoring

Solutions to protect network processors from attacks on vulnerable processing code are constrained by the limited resources available on these systems. One promising approach is to use *hardware monitors*, which have been successfully used in resource-constrained embedded systems [15]–[17].

The operation of a hardware monitor is illustrated in Figure 2. The key idea is that the processing core reports what it is doing as a monitoring stream to the monitor. The monitor compares the operations of the processor core with what it thinks the core should be doing. If a discrepancy is detected, the recovery system is activated to reset the processor core. In order to inform the monitor of what processing steps are valid, the processing binary is analyzed offline to extract the "monitoring graph" that contains all possible valid program execution sequences.

The granularity of monitoring can range from basic blocks [15] to individual processor instructions [17]. The detection times are as low as a single processor cycle, and the recovery times are in the order of tens of processor

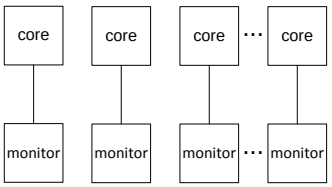

Fig. 3: One-to-one configuration.

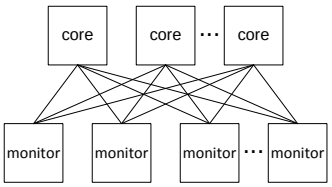

Fig. 4: Full interconnect configuration.

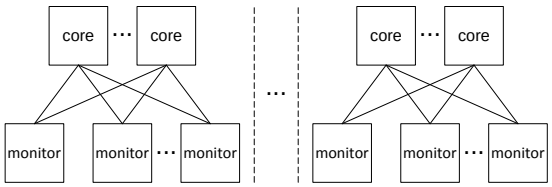

Fig. 5: Cluster configuration.

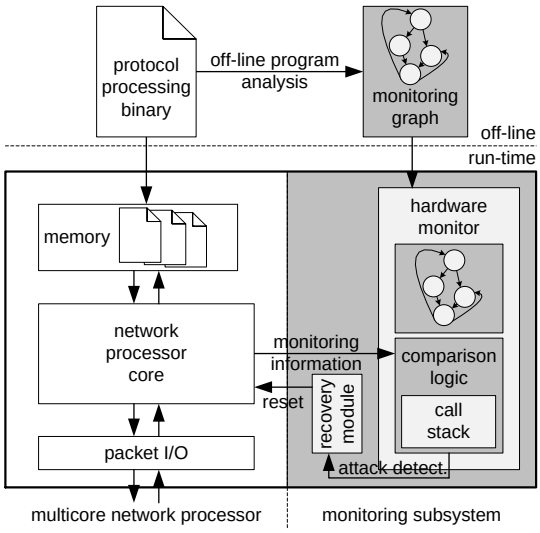

Fig. 2: Hardware monitor for a single processor core.

cycles. Since the monitor does not need to implement a full processor data path and the monitoring information can be compressed through hashing, the overall size of a typical monitor and its memory is about 10–20% of that of a processor core.

Hardware monitors for *single core* network processor systems have been demonstrated in prior work [3], [18]. These solutions, however, do not address three critical problems that appear in practical network processor systems:

- Multiple cores: Practical network processors use multiple processor cores in parallel, and all of these cores need to be protected by hardware monitors.
- Multiple processing binaries: Network processors need to perform different packet processing functions on different types of network traffic. These different operations are represented by different processing binaries on the network processing system. Thus, different cores may need to execute different binaries and need to be monitored by hardware monitors that match these binaries.
- Dynamically changing workload: Due to changes in network traffic during runtime, the workload of processor cores may change dynamically [7]. Thus, hardware monitors need to adapt to the changing processing binaries during runtime.

We present the design and prototype of a hardware monitoring *system* that can accommodate these requirements.

4 Scalable Hardware Monitoring Grid

4.1 Design Challenges

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

These challenges necessitate the design of a customized solution for multicore monitoring. Perhaps the most straightforward monitoring approach would be simply to attach a dedicated monitor to each individual NP core, following previous approaches to single-core monitoring, as shown in Figure 3. Although this approach minimizes the amount of interconnect hardware needed to connect an NP core to a monitor, it suffers from the need to reload monitoring information each time the attached NP core's program is changed. Alternatively, allowing an NP core to dynamically access any monitor among a pool of monitors as shown in Figure 4, while flexible, is expensive and incurs a high processor-to-monitor communication cost. In the next section, we describe a scalable monitoring grid system that balances these two concerns of area and performance overhead by using the clustered approach illustrated in Figure 5.

4.2 Architecture of Scalable Hardware Monitoring Grid

Our model of the multicore NP system including monitoring is shown in Figure 6. The architecture includes a control processor that coordinates overall NP operation

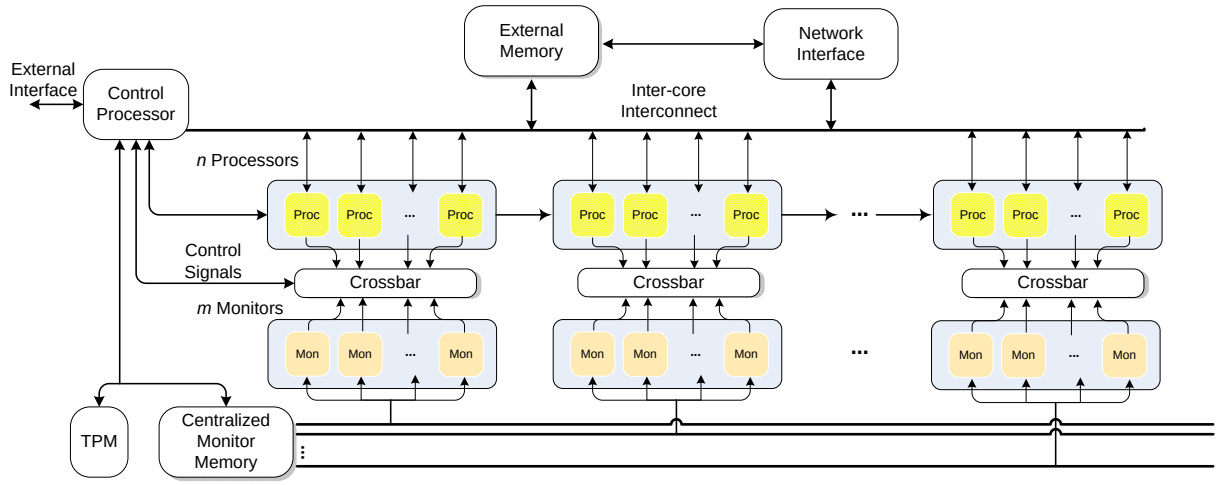

Fig. 6: Overview of Scalable Hardware Monitoring Grid with network processor cores organized into clusters. Note that the local processor and control processor memory shown in Figure 1 have been omitted for clarity.

by assigning arriving packets to individual NP cores. Each core executes a program using instructions from its local memory. External memory, which can be used to buffer packets and instructions for currently unused programs, is located off-chip. An on-chip interconnect is used to connect cores to external memory and outside interfaces. In this architecture, processors are grouped into *clusters* of n processors. Any of the processors in a cluster can be connected to any of m monitors.

The management of loading application-specific monitoring graphs into monitors and configuring specific processor-to-monitor connections is performed by the same control processor used to assign packets to NP cores. Graph loading is performed in conjunction with security key management hardware (e.g., a trusted platform module, *TPM*). Copies of monitoring graphs for programs that are currently being executed or are likely to be executed in the near future are stored on-chip in a *centralized monitor memory (CMM)*.

To ensure that monitoring graph information can be installed securely in the CMM, we use the following process to ensure authenticity, integrity and confidentiality:

- Authenticity and integrity: We use asymmetric cryptography to generate digital signatures that enable the TPM to verify authenticity and integrity of a monitoring graph. Public keys are installed using certificates that establish a chain of trust from the router manufacturer to the network operator to the TPM on the router system. Thus, an attacker cannot install a modified monitoring graph without having access to the private keys of the network operator.
- Confidentiality: We use symmetric cryptography to hide the monitoring graph information from an attacker when the monitoring graph is installed remotely through the network. The symmetric key is provided by the network operator securely to the TPM by encrypting it asymmetrically with the TPM's public key. Using this approach, an attacker cannot obtain any information about the monitoring graph.

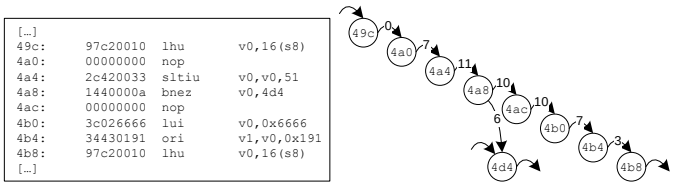

Fig. 7: State machine for hardware monitor generated from processing binary.

The details of the installation and verification process, including a detailed security analysis, are presented in [21].

The amount of time needed to load a monitor with a graph from the centralized monitor memory is significant enough that reloading should be minimized. It is desirable to have a program monitor used by different cores at different times during packet processing, necessitating a flexible interconnection between NP cores and monitors. In cases where $m > n$, a total of $m-n$ monitors are unused at a given point in time, although they can be activated by the control processor, if needed.

4.3 Multi-Ported Hardware Monitor Design

To support scalability, we have optimized the structure of single-processor monitors, which are capable of tracking NP core execution on an instruction-by-instruction basis. The monitoring graph for an application is generated from the application binary (Figure 2) at compile-time by performing an analysis of execution flow and feasible branch targets. Each instruction in the program can be represented as a state in a finite state machine (Figure 7). A transition between states is represented as the hash of the instruction binary. Control flow instructions (e.g., branch, jump) have multiple next states representing different flows of control.

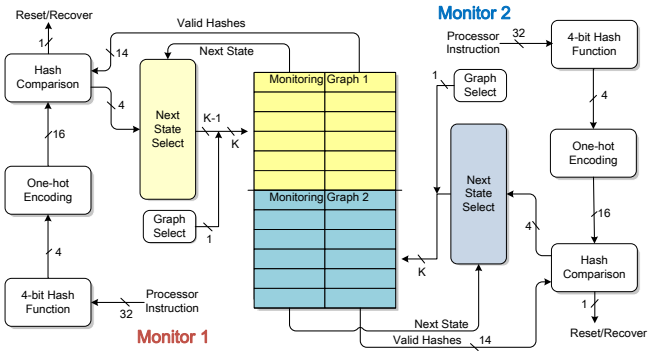

Fig. 8: Two monitors sharing a single dual-ported memory block.

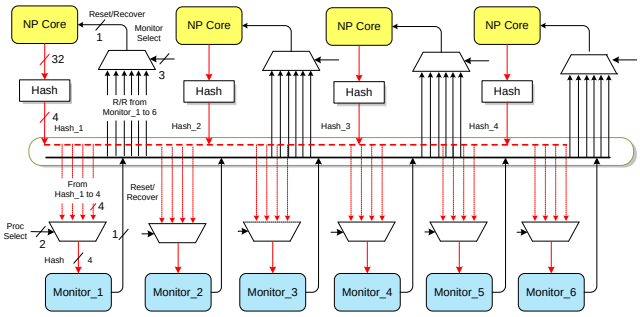

Fig. 9: Flexible interconnection between processors and monitors in a cluster.

The monitoring graph includes all feasible next states for the current state and the expected hash values for transitions from the current state. Hash values are used rather than the instruction values themselves to save memory space for the monitoring graph. Our approach handles hash collisions (e.g., multiple edges from a state with the same hash value) by converting the non-deterministic graph (NFA) generated directly from the binary to a deterministic finite automata (DFA). This process removes the effects of the hash collisions by adding states. Our approach is effective for all programs where the expected execution flow, including branches, can be determined either statically or via profiling. Our benchmark analysis shows that this is the typical case for NP applications. Full details of graph generation, including a detailed example, are shown in [18].

The architecture of two monitors that perform this type of instruction-by-instruction monitoring is shown in Figure 8. The monitoring graph, which is stored in a memory block, includes one entry for each state in the execution state diagram. A k-bit pointer indicates the entry in the graph that corresponds to the currently executed instruction. As an instruction is executed, a four-bit hash value of the instruction is generated, which is then converted to a one-hot encoding. This encoding is then compared against the *expected* hash values that are stored in the graph entry (*valid hashes*) for the instruction. The next entry (memory row) in the monitoring graph is determined using next state information stored in the current entry and the matched hash value. The implemented monitor requires only one memory lookup per instruction, limiting the time overhead of monitoring.

Although separate hash comparison and next state select information is needed for each monitor, multiple monitoring graphs can be packed into the same memory block if the block is multi-ported (Figure 8). In the example, the monitoring graph for the monitor on the left is located in the top half of the memory block while the graph for the monitor on the right is located in the bottom half. For each monitor, the selection of which monitoring graph (top or bottom) is used by the monitor is set by a single *graph select* bit which forms the top address bit into the block memory. A benefit of this shared memory block approach is the possibility of both monitors accessing the same monitoring graph at the same time without having to reload monitor memory (e.g. both associated NPs execute the same program and require the same monitor). In this case, the second graph in the memory block would be unused.

4.4 Scalable Processor-to-Monitor Interconnection

The detailed interconnection network between a cluster of n processors and m monitors is shown in Figure 9. In this architecture, any processor can be connected to any monitor via a series of n-to-1 (processor-to-monitor) and m-to-1 (monitor-to-processor) multiplexers. The four-bit hash values shown in Figure 8 are generated from instructions close to the processor, reducing processor-to-monitor interconnect. One of n four-bit values from the processors is selected for a specific monitor using multiplexer $\lceil log\ n \rceil$ select bits. During monitoring, a monitor generates a single reset/recover bit, which is returned to the monitored processor to indicate if an attack has occurred. In our implementation, this signal is sent to the target processor via a multiplexer with m single-bit inputs. The *monitor* and *processor select* bits are generated by the control processor and sent to the appropriate multiplexers via decoders.

5 Analysis of SHMG Adaptation

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

5.1 Monitor Configuration

In the n processors, m monitors SHMG system we defined in Section 4, for each program i ($1 \leq i \leq p$), we

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

$$w_i = q_i \cdot t_i. \tag{1}$$

In order to make the assignment of monitors to programs match the operation of the network processors, we need to determine how many of the n processors are executing program i at any given time. We assume that processors randomly draw from available packets (and thus the associated programs) when they are available. Thus, the probability of a processor being busy with processing program i, b_i, is proportional to the amount of work, w_i, that is incurred by the program (see Equation 1):

$$b_i = \frac{n \cdot w_i}{\sum_{j=1}^{p} w_j}. \tag{2}$$

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

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

$$a_i = max\left(\frac{m}{n} \cdot b_i, 1\right). \tag{3}$$

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

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

5.2 Blocking Probability and Throughput

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

$$Pr(B_i = k) = \binom{n}{k} \left(\frac{b_i}{n}\right)^k \left(1 - \frac{b_i}{n}\right)^{n-k}. \tag{4}$$

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

$$R_i = \sum_{j=A_i+1}^{n} (j - A_i) Pr(B_i = j). \tag{5}$$

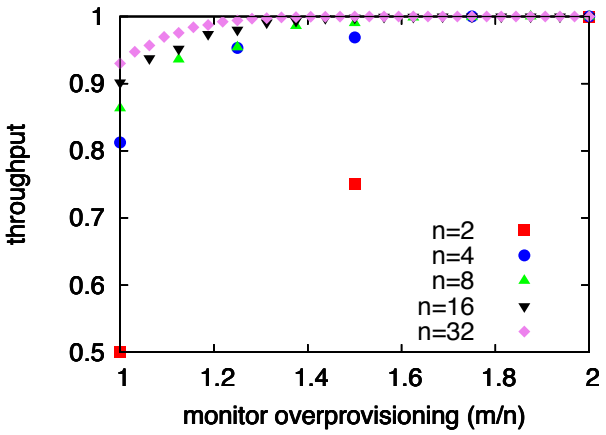

Fig. 10: Throughput depending on overprovisioning of monitors for different numbers of processors (n).

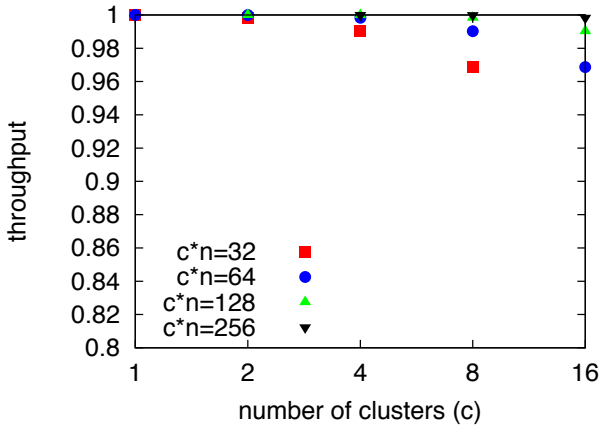

Fig. 11: Throughput depending on number of clusters for different numbers of total processors ($c \cdot n$).

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

$$R = \sum_{i=1}^{p} R_i. \tag{6}$$

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

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

$$t = 1 - \frac{R}{n}. \tag{7}$$

5.3 System Comparison

To illustrate the effect of blocking due to the unavailability of monitoring resources, we present several results based on the above analysis. For simplicity, we assume $p = 2$ programs with $w_1 = w_2$. Figure 10 shows the throughput as a function of how many more monitors than processors are in the system. We call this "monitor overprovisioning" (i.e., m/n). In the figure, the overprovisioning factor ranges from 1 (equal number of

monitors and processors) to 2 (twice as many monitors as processors). The figure shows that only for very small configurations (e.g., $n = 2$ processors), there is a significant decrease in throughput. For larger configurations, there is only a slight decrease for low overprovisioning factors. For our prototype implementation, we choose a configuration of $n = 4$ processors and $m = 6$ monitors (i.e., $m/n = 1.5$), which achieves a throughput of over 96%.

The effect of clustering is shown in Figure 11. Since we need to cluster monitors to achieve scalability in the system implementation, a key question is how much worse a clustered system performs compared to a system with no clustering (i.e., full interconnect between all processors and monitors). We denote the number of clusters with c. The figure shows the throughput for configurations with the same total number of processors and a monitor overprovisioning factor of 1.5. The full interconnect ($c = 1$) always achieves full throughput. As the number of clusters increases, small systems degrade in throughput slightly. However, if the number of processors per cluster does not drop below 8, throughput of over 99% can be achieved. These results indicate that using a clustered monitoring system instead of a full interconnect can achieve nearly full performance, while being much less costly to implement.

6 Runtime Resource Reallocation Algorithm

While the previous section provides an analytical evaluation of dynamic resource allocation, system throughput based on varying workloads can also be evaluated through experimentation. In the Scalable Hardware Monitoring Grid design, the control processor (see Figure 6) assigns programs to processors and monitors. As the traffic workload changes, the optimal assignment of cores and monitors should reflect the processing workload. In order to achieve this goal, a Runtime Resource Reallocation Algorithm (RRRA) is needed to dynamically reconfigure the SHMG at runtime based on network workload changes.

6.1 Reallocation Algorithm

The control processor periodically monitors network workload to assess the current allocation of processing resources. As network packets enter the network processor, they are buffered in the external memory in a series of packet queues. Each queue stores a different type of network packet. The control processor assigns packets to queues based on processing requirements and the number of packets in the queue defines the queue length. Similar, stable queue lengths for each packet type reflects packet processing balance in terms of number of assigned processors and router processing speed. If the queue length increases significantly, the network traffic is too heavy compared to the current processing speed and more compute resources are needed for this program.

We assume the input network traffic fully utilizes the system, i.e., when the workload increases for one packet type, then the workloads of other packet types decrease. As mentioned in Section 4.2, a processor/monitor cluster consists of n processors and m monitors. The workload of the system consists of p different programs that each monitor may execute (one program per packet type). For practicality, we assume $m \geq n$ and $m \geq p$ and no processor is idle.

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

6.1.1 Identification of Program for Reallocation

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

1) If a queue is empty, select this program to release one processor. If there is more than one empty queue, select the program that has had an empty queue for the longest time. For this purpose, an empty time marker te_k is used for each empty queue k to record the time the queue drained. The algorithm maintains a priority queue for the empty queue that allows easy identification of the queue that has been empty longest (i.e., with minimum te_k).
2) If no queue is empty, select the program with the shortest queue length that has at least two processors allocated. (Each program is guaranteed at least one active processor in the system if it has a non-zero queue length, so deallocating resources from a program with a single processor is not allowed.)

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

6.1.2 Identification of Monitor for Reallocation

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

1) Identify all unused monitors in the system.

Algorithm 1 Runtime Resource Reallocation Algorithm

```
1: ql_i(t) ← ql_i(t) + 1                    ▷ packet arrival for program i
2: if ql_i(t) ≥ θ then                      ▷ queue length passes threshold
3:     ql_min ← θ                           ▷ initialize variables to find j
4:     te_min ← t
5:     for k = 0 to p do                    ▷ iterate over all queues
6:         if ql_k(t) = 0 then              ▷ queue empty
7:             if te_k ≤ te_min then        ▷ empty for longer time
8:                 j ← k
9:                 te_min ← te_k
10:                ql_min ← 0
11:            end if
12:        else if ql_j(t) ≤ ql_min then    ▷ queue is shorter
13:            if a_j ≥ 2 then              ▷ has 2 or more processors allocated
14:                j ← k
15:                ql_min ← ql_k(t)
16:            end if
17:        end if
18:    end for
19: end if
20: a_j ← a_j − 1
21: a_i ← a_i + 1
```

2) If there is an unused monitor that has a preloaded graph of program i, identify all the processors in the same cluster as this monitor. If there is a processor running program j in the same cluster, switch it to program i, disconnect the program j monitor and connect the processor to the program i monitor.
3) If there is no processor running program j in the same cluster, try to find another unused monitor with program i in a different cluster.
4) If there is no unused monitor that has preloaded program i, switch one processer j to program i and reload the least recently used monitor to program i in the same cluster of the switched processor.

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

6.1.3 Reallocation Algorithm Complexity

Overall, the Runtime Resource Reallocation Algorithm has two traversal operations:

- Evaluate all p queues to find j when the threshold θ is exceeded by a program i. This action which requires $O(p)$ time.
- Evaluate all monitors in the clusters that include program j to find a monitor and processor core to use or switch functionality, which requires $O(m+n)$ time.

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

6.2 System Simulation

A Java-based simulator was built to verify RRRA and evaluate runtime throughput results with the results obtained from the analytical model in Section 5.3. The simulator can generate network packets that require processing by programs in different ratios and can vary these ratios over time. With this time-changing network traffic input, the simulator assesses the behavior of RRRA and measures the runtime resource allocation and system throughput.

6.2.1 Runtime Adaptation

Figure 12 shows that network traffic is balanced (i.e., the total number of packets during a fixed time period is the same) for three different types of packets. The packet proportions change from $1:1:1$ to $8:1:1$, then to $1:1:8$, and finally back to $1:1:1$ and then decrease to 0. For simplicity, we assume that the packet processing time for all types is equal (i.e., $t_1 = t_2 = t_3$). Figure 13 shows the number of processors running each program during the runtime in an experimental system with 16 processors and 24 monitors. We can observe that the ratios of the processors assigned to each program follow the ratios of each packet type in the network traffic, thus validating the effectiveness of the RRRA. In Figure 14, the system throughput is at a maximum shortly after the system starts processing. Then, there are three obvious transitions corresponding to the three network traffic allocation changes. The biggest drop happens when the network traffic changes dramatically from $8:1:1$ to $1:1:8$. Using RRRA, the system can adapt its resource allocation to traffic changes and the throughput quickly returns to the maximum value.

6.2.2 Overprovisioning Simulation

To verify the monitor overprovisioning analysis, two experiments were conducted in simulation. The first experiment considered the simplest case of $p = 2$ with $w_1 = w_2$, total processor number $n = 4, 8, 16, 32$, and overprovisioning factor ranging from 1 to 2. The results shown in Figure 15 match with the previous analytical results in Figure 10. The second experiment kept the assumption of evenly distributed workload, but extended the program number to three and the overprovisioning factor range from 1 to 3. Although the upper bound of the overprovisioning range increases to 3, the throughput is still more than 95% after $m/n = 1.5$.

The effect of number of clusters on throughput was measured in a second simulation. The experiment was setup with monitor overprovisioning of 1.5, the same as the previous analytical analysis, and a total number of

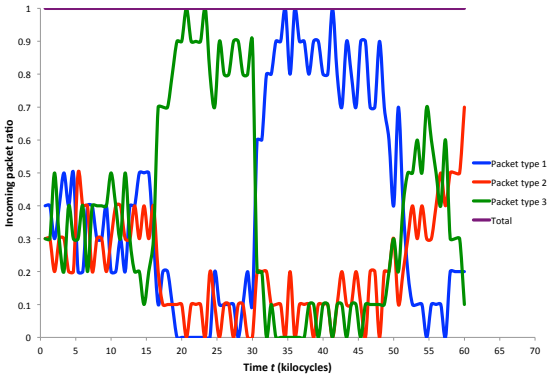

Fig. 12: Input network traffic used during simulation.

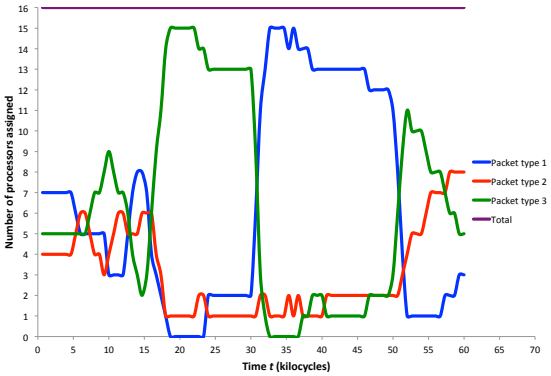

Fig. 13: Processor distribution for different packet types as traffic allocation changes.

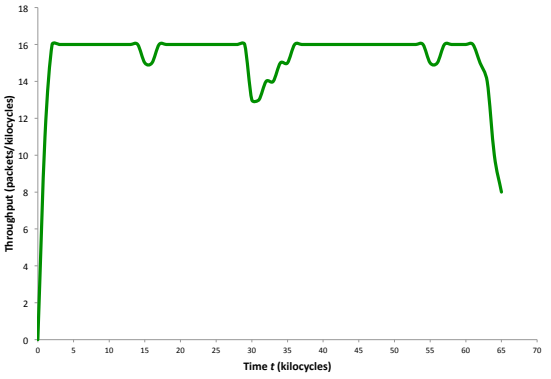

Fig. 14: Throughput changes is relation to traffic changes.

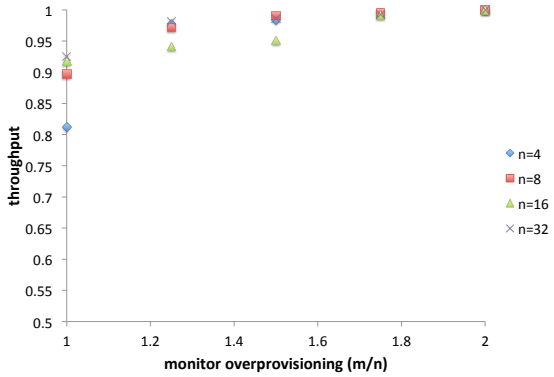

Fig. 15: Simulation throughput depending on overprovisioning of monitors for different numbers of processors (n) with two different packet types.

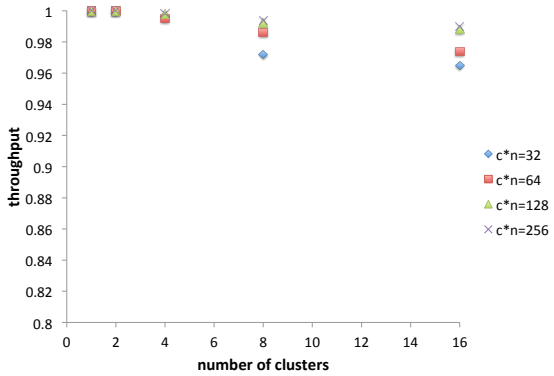

Fig. 16: Simulation throughput depending on number of clusters for different numbers of processors (n) with two different packet types.

processors of 32, 64, 128 and 256. Compared to the analytical results in Figure 11, results from this experiment, shown in Figure 16, demonstrate good consistency, thus supporting the appropriateness of the analytical model.

7 Prototype Implementation and Evaluation

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

7.1 Experimental Setup

We have implemented a prototype network processor in an Altera Stratix IV FPGA on an Altera DE4 board. This board contains four 1 Gbps Ethernet ports to receive

and send network traffic. We implemented one SHMG cluster in the FPGA, consisting of four processor cores (soft processors created using a synthesizable PLASMA processor [22]) and six hardware monitors (i.e., $n = 4$ and $m = 6$). The flexible, multiplexer-based interconnect shown in Figure 9 is used to allow any processor to connect to any monitor within our cluster.

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

We use two types of packets, which need different types of processing and thus different monitors: (1) IPv4 packets and (2) IPv4/UDP packets that require congestion management (CM) for processing. The processing code for IPv4 does not exhibit vulnerabilities, but the IPv4+CM processing code exhibits the integer overflow vulnerability described in [3]. We introduce 1% of attack packets, which can trigger a stack smashing attack in the IPv4+CM processing code [3].

To generate the monitoring graph, as described in Section 4.3, the program is first passed through the standard MIPS-GCC compiler flow to generate assembly-level instructions. The compiler output allows for the identification of branch instructions and their branch target addresses. The instructions and branch information are then processed to generate the data structure used inside the hardware monitor. This data structure is then loaded into the SHMG system.

7.2 Experimental Results

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

7.2.1 Correct Operation

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

Figure 17 shows the operation of a processor core and its corresponding monitor on the IPv4 program. (Waveform figures are generated through simulation in order to obtain signals; however, the same functionality has been verified in real-time operation of the system on network traffic.) Similarly, Figure 18 shows the operation of a core on the IPv4+CM program. In this case, the packet is benign and no attack occurs. Figure 19 shows the processing of an attack packet in IPv4+CM. The attack is identified within one cycle of the instruction fetch. Figure 19 shows that the monitor identifies the attack since the stack gets smashed and the control flow is redirected to code that differs from what the program analysis has determined as valid. The processor core is then reset and continues processing the next packet. The reset operation completes in two cycles and thus does not affect the throughput performance of the system (and cannot be used as a target for denial of service attacks). Other processor cores continue processing without being affected.

TABLE 1: Resource utilization and dynamic power consumption in the prototype system

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

A key functionality of SHMG is the dynamic assignment of processors to hardware monitors. In our prototype system, we can trigger the reassignment of processors to monitors on-demand. In our experimental setup, we switch one of the processor cores from IPv4 (Figure 17) to IPv4+CM (Figure 18). The processor-to-monitor interconnect for the core that was previously processing IPv4 packets is switched to connect the core to an unused IPv4+CM monitor. The affected NP core and newly connected monitor are then reset, and processing by the core commences. After this runtime reconfiguration, three NP cores process packets for IPv4+CM, while one core processes IPv4.

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

7.2.2 Resource Requirements and System Throughput

The resource requirements for the FPGA in our prototype system are shown in Table 1. The lookup table (LUT), flip flop (FF), and memory resources (Bits) required for the network processor cores, monitors, switches and other circuitry are shown in Table 1. An LUT is a n-input, 1-output logic element that can perform any logic function of n inputs. In Stratix IV devices, LUTs can be configured to have between two to seven inputs. Each monitoring graph can hold up to 4,096 separate entries. The FPGA in the system is able to operate at 125 MHz. For this relatively small cluster size, the amount of logic needed for processor-to-monitor interconnection is less than 1% of the total logic needed for the monitors, cores, and processor-to-

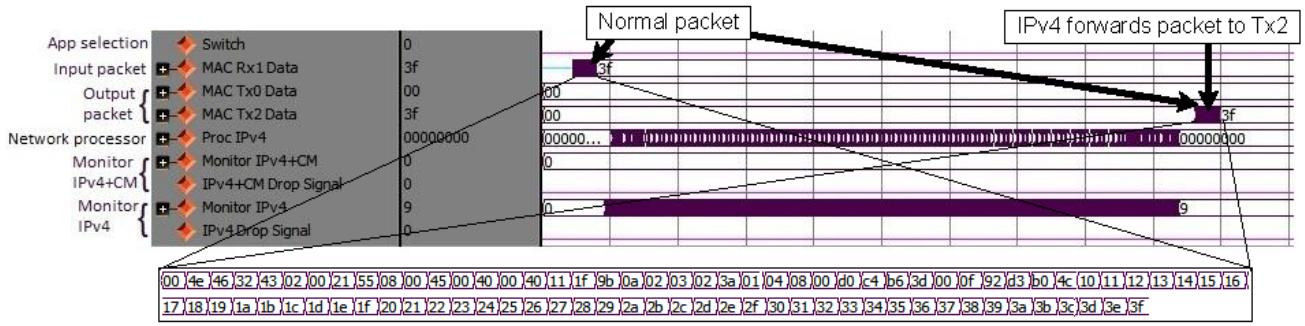

Fig. 17: Simulation waveforms showing correct forwarding of an IPv4 packet.

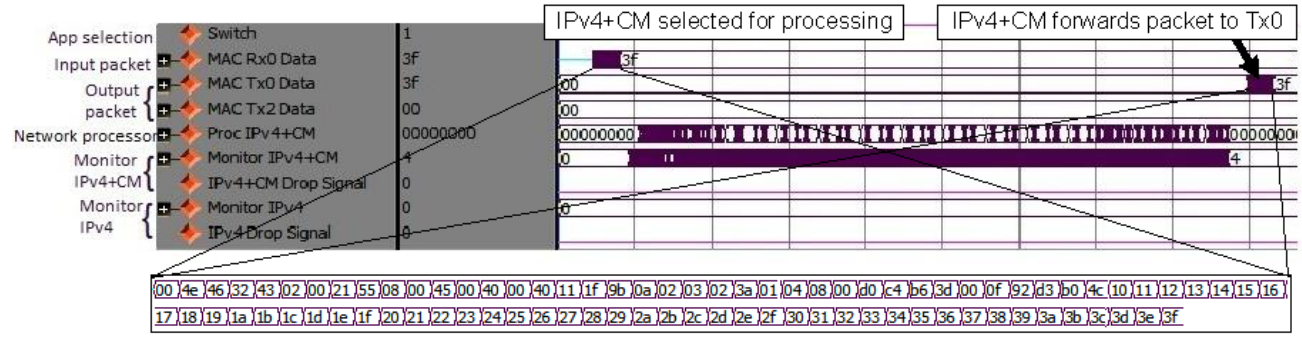

Fig. 18: Simulation waveforms showing forwarding of an IPv4+CM packet.

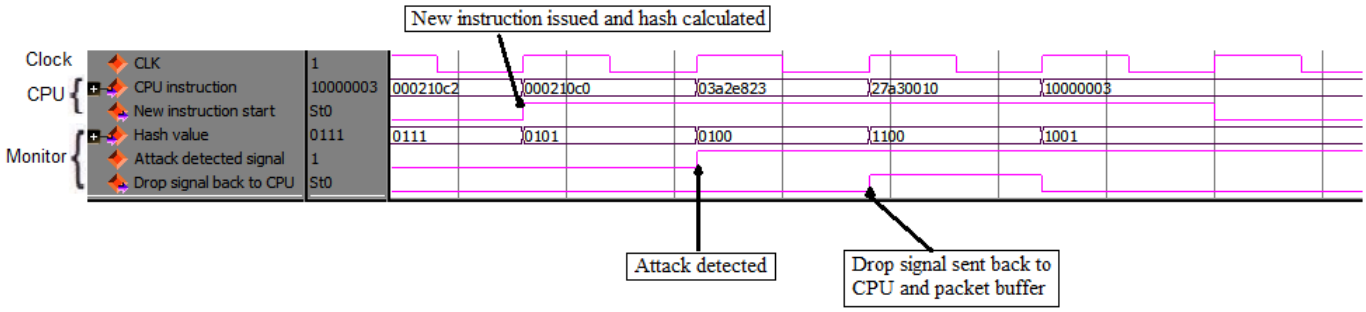

Fig. 19: Simulation waveforms showing identification of and recovery from an IPv4+CM attack packet.

monitor interconnect since only hash value, reset, and control signals are communicated.

To assess the generality of our area results across different FPGA generations, we resynthesized the network processor cores, monitors, and interconnect to an Altera Stratix II device. The resulting LUT counts of 14,912, 774, and 92 for the processor cores, monitors, and interconnect, respectively are similar to the Stratix IV numbers. For a Stratix II device, an LUT can range in size from 2-input to 7-input depending on the desired logic function. The distribution of input counts for LUTs across this input spectrum was similar for both architectures.

The dynamic power consumption of the components, shown in Table 1, was determined using the Altera PowerPlay power analyzer. The monitors and associated interconnect consumed 12% of the dynamic power of the processors. The network and PCI interfaces on the board consumed 3.4× more dynamic power than these components combined. Based on board level experimentation, average time to process one hundred 256 byte packets is 6 ms. As a result, the dynamic energy to process 100 256-byte packets at 125 MHz is 6 ms × 1926.49 mW = 11.56 mJ.

The throughput of our system including monitoring when processing normal 256-byte packets and an occasional attack packet is shown in Figure 20. The throughout of the system is limited by the processing capability of the processor cores, not monitoring. The throughput for normal packets is the same both with and without monitoring. A small throughput reduction is observed in the presence of attack packets due to the amount of time needed to flush the packet buffer.

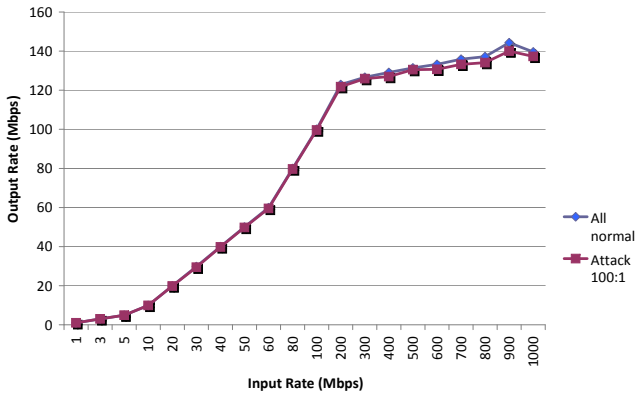

Fig. 20: Throughput results when processing normal IPv4 with congestion management packets and when processing IPv4 with congestion management packets if one out of 100 packets is an attack packet.

7.2.3 Monitoring Graph Swap Time Overhead

To better illustrate the benefits of overprovisioning the monitors relative to processor count ($m > n$), we assess the average time required to swap monitors during a processor allocation for the case when m/n = 1.5 versus the case when a monitoring graph must be reloaded from centralized monitor memory for every processor reallocation. The steps required to perform each task of monitor swapping includes: identification of a new program i for allocation, identification of a program to swap out (j), identification of a target processor core and monitor for the new program, and monitor reload from centralized monitor memory (if needed). The reallocation operations needed to perform the first three steps in the list were discussed in Section 6.1. The following analysis is performed for a two cluster system with $n = 6$ processor cores and $m = 9$ monitors in each cluster. The control processor operates at 125 MHz, the clock speed for our prototype hardware implementation. Since processor throughput is one clock cycle, we equate an instruction execution to a clock cycle. The instructions for the programs are stored in each processor core's local memory.

The initial allocation and deallocation steps require examination of packet queues to identify a program i for allocation to a processor and the reduction of one processor for a program j (Section 6.1.1). Our experimentation using system simulation shows that 28 control processor instructions are needed on average to identify a program i which requires an additional processor. An additional 28 control processor instructions are required to identify a program j that should have a processor deallocated. Combined, these actions require 0.45 μs.

The tasks needed to identify a monitor for reallocation are detailed in Section 6.1.2. This stage attempts to identify a spare monitor which has been previously loaded with the monitoring graph for program i and a processor core which is currently tasked with program j. This core is subsequently switched to program i and the processor core/monitor interconnect is configured for the new connection. The process of identifying a processor core, swapping its program, and locating a suitable preloaded monitor requires 197 instructions (clock cycles) on average, based on our simulation. The configuration of the interconnect between the monitor and processor core in the cluster requires 3 clock cycles. In total, these actions require 1.60 μs.

TABLE 2: NpBench monitor graph reload cost

Network benchmark	Memory graph size (bits)	Graph reload time (cycles)	Graph reload time (μs)
crc	8,460	529	2.64
frag	18,660	1,166	5.83
red	25,410	1,588	7.94
md5	96,840	6,052	30.26
ssld	25,620	1,601	8.01
wfq	28,590	1,787	8.93
mtc	77,160	4,822	24.11
mpls (up)	52,590	3,287	16.43
mpls (down)	51,180	3,199	15.99

In some cases, if a spare monitor with the appropriate graph cannot be found, a graph must be loaded into monitor memory. To evaluate the average monitoring graph reloading cost from centralized monitor memory to the dual-ported memory in a monitor, nine benchmarks from the NpBench suite [23] were processed with an offline analysis flow. NpBench is a benchmark suite targeting modern network processor applications. The benchmark applications are categorized into three specific functional groups: traffic management and quality of service group (TQG), security and media processing group (SMG) and packet processing group (PPG). In our evaluation, monitor graph sizes generated with a 4-bit nibble-sum hash function were calculated and the graph read/write times to an on-chip memory were estimated for each of the benchmarks. The reloading time estimation was based on on-chip SRAM which is a 200 MHz SSRAM with a 16-bit data bus. Table 2 shows the evaluation results. The average reload time is found to be 13.34 μs.

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

8 Conclusions and Future Work

The use of general-purpose processors to implement packet forwarding functions in routers has opened the door for a new class of network data plane attacks. Prior work has shown examples for such attacks and their considerable effects. To provide practical protection for network processors, which are multicore systems with highly dynamic workloads, we have presented our design of a Scalable Hardware Monitoring Grid. This monitoring system groups multiple processors and monitors into clusters and provides an interconnect to dynamically assign processor cores to monitors based on their current workload using a Runtime Resource Reallocation Algorithm. We show that the system can correctly identify attacks and recover the attacked core so that it can continue processing. In the future we plan to assess techniques to speed up the runtime swapping of monitoring information.

References

[1] F. Baker, "Requirements for IP version 4 routers," Network Working Group, RFC 1812, Jun. 1995.

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

[3] D. Chasaki and T. Wolf, "Attacks and defenses in the data plane of networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 6, pp. 798–810, Nov. 2012.

[4] *The Open Source Network Intrusion Detection System*, Snort, 2014, http://www.snort.org.

[5] *The Bro Network Security Monitor*, The Bro Project, 2014, http://www.bro-ids.org.

[6] D. Chasaki and T. Wolf, "Design of a secure packet processor," in *Proc. of ACM/IEEE Symp. on Arch. for Networking and Communication Systems (ANCS)*, San Diego, CA, Oct. 2010, pp. 1–10.

[7] Q. Wu and T. Wolf, "Runtime task allocation in multi-core packet processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 10, pp. 1934–1943, Oct. 2012.

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

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

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

[11] A. Cui, Y. Song, P. V. Prabhu, and S. J. Stolfo, "Brave new world: Pervasive insecurity of embedded network devices," in *Proc. of 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, ser. Lecture Notes in Computer Science, vol. 5758, Saint-Malo, France, Sep. 2009, pp. 378–380.

[12] Cisco, Inc., "Cisco IOS," http://www.cisco.com.

[13] P. Koopman, "Embedded system security," *Computer*, vol. 37, no. 7, pp. 95–97, Jul. 2004.

[14] S. Parameswaran and T. Wolf, "Embedded systems security – an overview," *Design Automation for Embedded Systems*, vol. 12, no. 3, pp. 173–183, Sep. 2008.

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

[16] R. G. Ragel, S. Parameswaran, and S. M. Kia, "Micro embedded monitoring for security in application specific instruction-set processors," in *Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, San Francisco, CA, Sep. 2005, pp. 304–314.

[17] 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.

[18] H. Kumarapillai Chandrikakutty, D. Unnikrishnan, R. Tessier, and T. Wolf, "High-performance hardware monitors to protect network processors from data plane attacks," in *Proc. of Design Automation Conference (DAC)*, Austin, TX, Jun. 2013, pp. 1–6.

[19] K. Hu, H. Chandrikakutty, R. Tessier, and T. Wolf, "Scalable hardware monitors to protect network processors from data plane attacks," in *Proc. of the IEEE Conference on Network and Computer Security*, Washington, DC, Oct. 2013, pp. 314–322.

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

[21] K. Hu, T. Wolf, T. Teixeira, and R. Tessier, "System-level security for network processors with hardware monitors," in *Proc. Design Automation Conf. (DAC)*, San Francisco, CA, Jun. 2014, pp. 1–6.

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

[23] B. K. Lee and L. K. John, "NpBench: A benchmark suite for control plane and data plane applications for network processors," in *Proc. of IEEE International Conference on Computer Design (ICCD)*, San Jose, CA, Oct. 2003, pp. 226–233.

Kekai Hu is a Ph.D. candidate in the Electrical and Computer Engineering department at the University of Massachusetts, Amherst. He received the B.S. and M.S. degrees in electrical and computer engineering from Wuhan University, China, in 2007 and 2009, respectively. His research interests include network routers, embedded system security, and computer architecture.

Harikrishnan Kumarapillai Chandrikakutty received the B.Tech. degree in applied electronics and instrumentation engineering from College of Engineering, Trivandrum, India in 2008 and the M.S. degree in electrical and computer engineering from the University of Massachusetts, Amherst in 2013. He is currently with Juniper Networks, Westford, MA, where he is involved in the design and verification of next generation routers and switch systems.

Zachary Goodman received the B.S. degree in electrical engineering from the University of Massachusetts, Amherst in 2014.

Russell Tessier (M'00-SM'07) received the B.S. degree in computer and systems engineering from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1989, and the S.M. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1992 and 1999, respectively. He is currently Professor of Electrical and Computer Engineering with the University of Massachusetts, Amherst, MA. His current research interests include computer architecture and FPGAs.

Tilman Wolf (M'02-SM'07) is Professor of Electrical and Computer Engineering at the University of Massachusetts Amherst. He received a Diplom in informatics from the University of Stuttgart, Germany, in 1998. He also received a M.S. in computer science in 1998, a M.S. in computer engineering in 2000, and a D.Sc. in computer science in 2002, all from Washington University in St. Louis. His research interests include Internet architecture, network routers, and embedded system security.