Scalable Hardware Monitors to Protect Network Processors from Data Plane Attacks

Kekai Hu, Harikrishnan Chandrikakutty, Russell Tessier and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA
{khu,chandrikakut,tessier,wolf}@ecs.umass.edu

***Abstract*—Modern router hardware in computer networks is based on programmable network processors, which implement various packet forwarding operations in software. These processor systems are vulnerable to attacks that can be launched entirely through the data plane of the network without any access to the control interface of the router. Prior work has shown that a single malformed UDP packet can take over a network processor running vulnerable packet processing software and trigger a devastating denial-of-service attack from within the network. One possible defense mechanism for these resource-constrained network processors is the use of hardware monitoring systems that track the operations of each processor core. Any deviation from programmed behavior indicates an attack and triggers reset and recovery actions. Such hardware monitors have been studied extensively for single processor cores, but network processors consist of dozens to hundreds of processors with highly dynamic workloads. In this paper, we present the design of a Scalable Hardware Monitoring Grid, which allows the dynamic sharing of hardware monitoring resources among processor cores. We show the scalability of our monitoring system to network processors with large numbers of cores. We also present a multicore prototype implementation of the monitoring system on an FPGA platform.**

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

I. Introduction

Routers are an essential component of today's Internet. Routers connect network links and perform the protocol processing operations that are necessary to send traffic along the correct path, perform various correctness and security checks, and keep track of performance and traffic statistics. While the Internet Protocol (IP) only has a very small set of required operations [1], there are countless additional functions that have been added to improve network performance, to allow for provider-specific network management, and to perform accounting. To implement these functions, routers no longer use application-specific integrated circuits (ASICs), but programmable network processors (NPs) [2]. Network processors are high-performance embedded systems with many processor cores that are programmed with software. The use of software instead of hard-coded logic allows router vendors and network providers to customize and update router functionality as necessary. Practically all modern high-performance routers use network processors.

While network processors offer great benefits in terms of flexibility, since they can be reprogrammed, they also exhibit potential security risks. Just as general-purpose workstation and server processors have software vulnerabilities that can be attacked remotely, network processors have software with potential security vulnerabilities. Such security vulnerabilities can be exploited to change the behavior of the router. In particular, prior work has shown that an experimental network processor with a security vulnerability in packet processing code can be attacked by sending a single User Datagram Protocol (UDP) packet [3]. The result of the attack was the indefinite retransmission of the attack packet on the outgoing link at full data rate. This type of attack is particularly concerning since it can be launched through the data plane of the network (i.e., no access to the control interface of the router is necessary) Its effect can be devastating since routers in the network inherently have access to multiple high-bandwidth links. Thus, these types of attack can trigger Gigabits of attack traffic with a single transmission.

While similar vulnerabilities and attacks have not yet been disclosed for current commercial router systems, there are no fundamental reasons why they cannot be found. In particular, network processor software complexity continues to grow as more features are deployed and thus the attack surface continues to increase. It is important to note that even a single vulnerability can have devastating effects on the operation of the Internet due to the homogeneity of the network equipment ecosystem. Currently, the network equipment market is dominated by a small number of vendors. If a vulnerability in deployed network processor code can be exploited, then a large number of systems can be effected simultaneously. The ability to take down a significant fraction of all network devices in a short time would allow an attacker to drastically affect critical infrastructure. Such capabilities are particularly concerning in the context of cyber warfare (e.g., [4]).

Defenses against attacks on network processors need to match the system constraints of these devices. In particular, network processors use simple processor cores that typically do not run full operating systems. Thus, conventional software protection mechanisms (e.g., anti-malware software) are not suitable for this domain. In addition, network intrusion detection systems (e.g., snort [5] or Bro [6]) are often only active on the ingress side of campus networks and thus do not protect the Internet core. Instead, hardware monitoring techniques have been proposed as an effective protection mechanism for network processors [7]. These hardware monitors operate in parallel to the embedded network processor cores and monitor the processor behavior during runtime. If any deviation from programmed behavior is detected, the processor core can be reset and continue operating without executing attack code.

While many different hardware monitor designs have been proposed in prior and related work, they all focus on single-core systems with static (or very slowly changing) workloads. Network processors, however, use dozens or hundreds of parallel processor cores and have processing workloads that can change dynamically based on the network traffic [8]. Thus, the problem of how to realize an entire *multicore hardware monitoring system* is critical for developing effective protection mechanisms for network processors. In this paper, we present the design and prototype implementation of a Scalable Hardware Monitoring Grid (SHMG) that provides a solution to this problem. Our design uses an interconnection network between processors and monitors to dynamically assign processors to monitors based on their processing workload. In addition, monitors can be shared by multiple processor cores to reduce the implementation overhead for the monitoring system. Our analysis shows that our design can scale to large numbers of processor 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 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.

Overall, our analysis and implementation results show that our design of the Scalable Hardware Monitoring Grid is an effective and scalable solution to providing protection for network processors and thus for the Internet infrastructure.

This remainder of the paper is organized as follows. Section II discusses related work. We describe data plane attacks and potential defense mechanisms in detail in Section III. Section IV introduces the design of our Scalable Hardware Monitoring Grid. We evaluate the runtime performance of our design in Section V. Results from a prototype implementation are presented in Section VI. Section VII summarizes and concludes this paper.

II. 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 [9], Cavium Octeon [10], and EZchip NP-5 [11] with data rates in the low hundreds of Gigabits per second.

Attacks on networking devices have been described in [12], but that work explored vulnerabilities in the *control plane*, where attacks aim to hack into the control interface of a router (e.g., IOS [13]). 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 [14], [15].

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 [16]–[18] and has also been proposed for use in network processors [7]. 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 [19].

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 may experience highly dynamic workload changes based on changing traffic patterns [8], effective solutions for such an environment need to be developed.

III. Data Plane Attacks and Defenses

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

A. 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

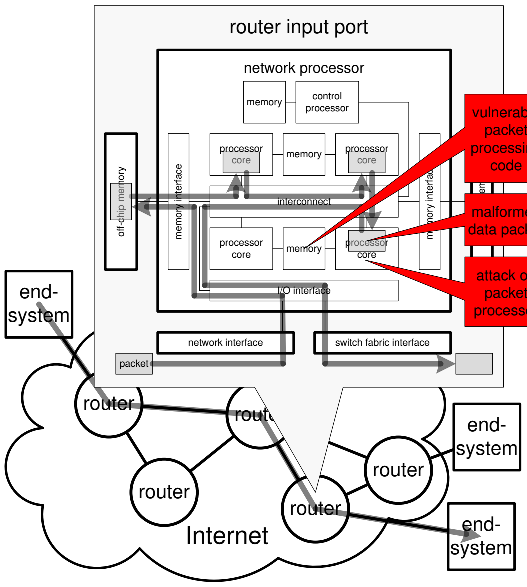

Fig. 1. Attack on network processor.

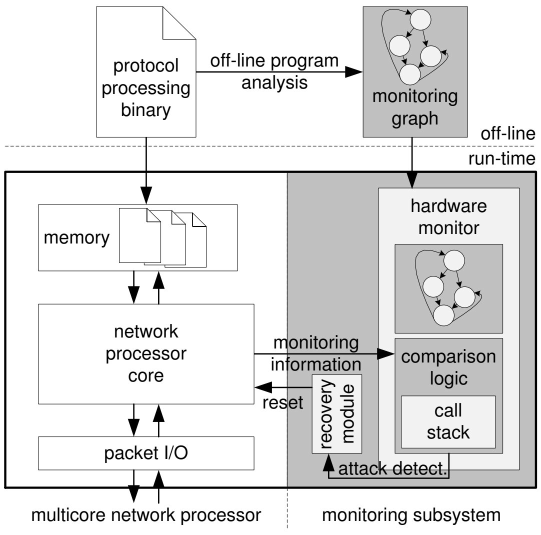

Fig. 2. Hardware monitor for single processor core.

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 [19], leading to return-to-libc attacks.

B. Defense Mechanisms Using Hardware Monitoring

Solutions to protect network processors from attacks on vulnerable processing code are constrained by the limited resources available on these systems. One promising approach is to use *hardware monitors*, which have been successfully used in resource-constrained embedded systems [16]–[18].

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 [16] to individual processor instructions [18]. The detection times are as low as a single processor cycle, and the recovery times are in the order of tens of processor 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], [19]. These

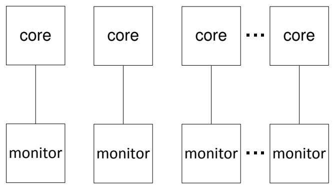

Fig. 3. One-to-one configuration.

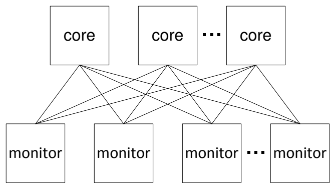

Fig. 4. Full interconnect configuration.

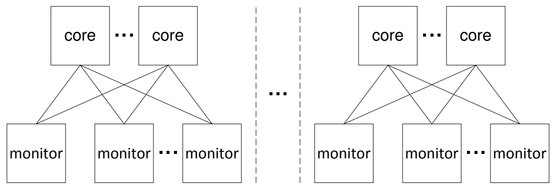

Fig. 5. Cluster configuration.

solutions, however, do not address two 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 [8]. 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.

IV. Scalable Hardware Monitoring Grid

A. 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.

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

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

C. 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

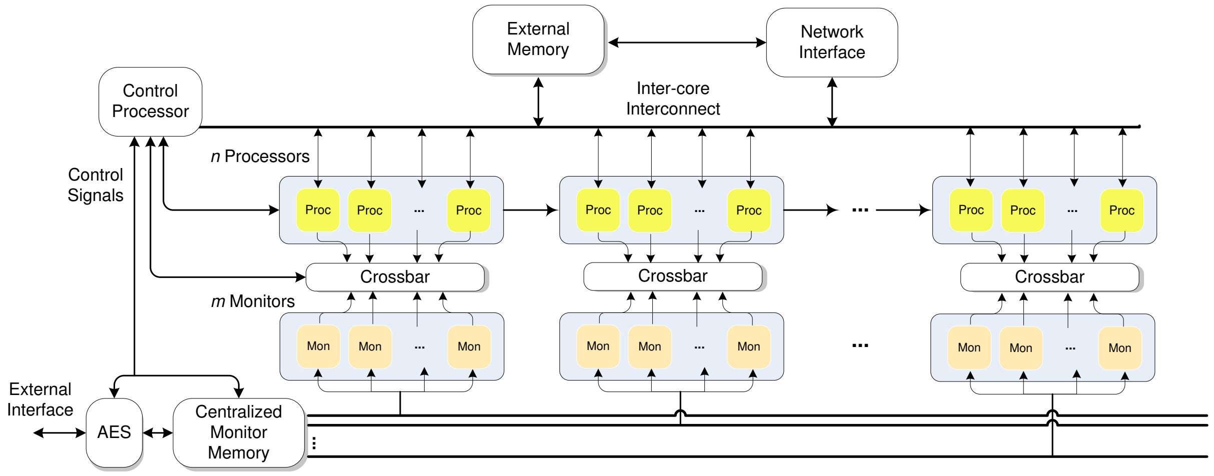

Fig. 6. Overview of Scalable Hardware Monitoring Grid with network processor cores organized into clusters. Note that the local processor and control processor memory shown in Figure 1 have been omitted for clarity.

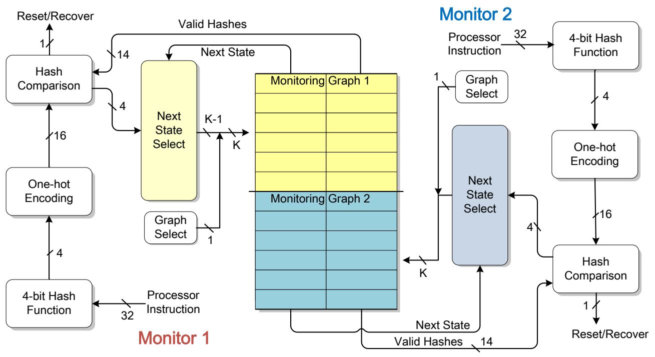

Fig. 7. Two monitors sharing a single dual-ported memory block.

monitoring graph for this class of monitor typically represents each program instruction as a state in a state diagram [19]. Expected program execution can be modeled as transitions between known states. To evaluate correct processor operation for an instruction, the progression between states is tracked using instruction hash values. If the hash value of the instruction from the processor does not match the value stored in the monitoring graph for the instruction, a deviation from expected execution flow is detected and the processor is reset. For network processors, this action typically involves a stack reset and a packet drop. The monitoring graph for a program can be determined by analyzing the instruction flow of the program binary. For control flow instructions, multiple next states may be possible in the monitoring graph, requiring matching against several possible hash values.

The architecture of two monitors that perform this type of instruction-by-instruction monitoring is shown in Figure 7. 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. (See [18] for a justification of hash size.) This encoding is then compared against the *expected* hash values that are stored in the graph entry. The next entry (memory row) in the monitoring graph is determined using next state information stored in the current entry and the matched hash value. The implemented monitor requires only one memory lookup per instruction, limiting the time overhead of monitoring.

Although separate hash comparison and next state select information is needed for each monitor, multiple monitoring graphs can be packed into the same memory block if the block is multi-ported (Figure 7). 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.

D. Scalable Processor-to-Monitor Interconnection

The detailed interconnection network between a cluster of n processors and m monitors is shown in Figure 8. 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 7 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

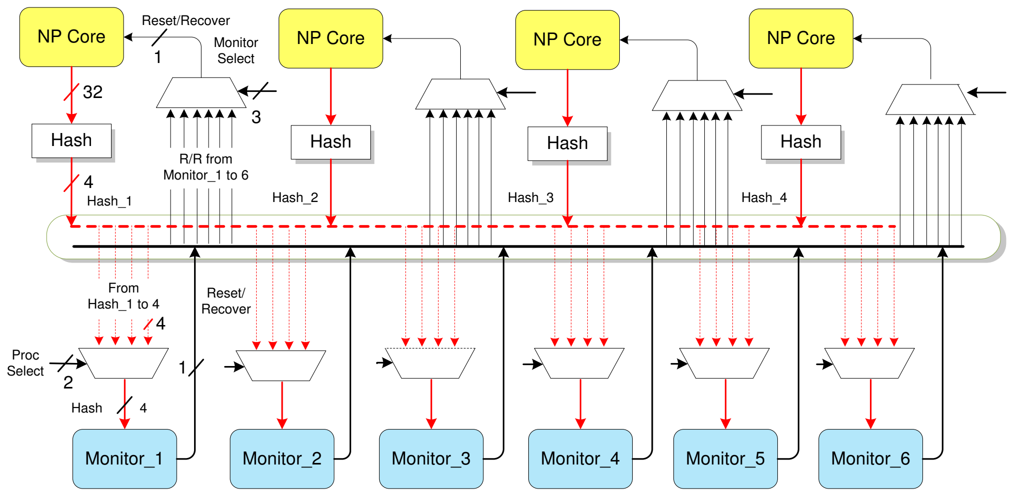

Fig. 8. Flexible interconnection between processors and monitors in a cluster.

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.

E. SHMG Monitor Operation

To verify correct NP core behavior during execution of a program, a monitor loaded with the proper monitoring graph must be connected to the processor prior to the start of execution. It is assumed that the monitoring graph has been previously generated using known techniques [19] and is available on-chip for loading. Monitoring and NP core execution for all cores take place in parallel across the network processor.

The connection of a monitor to an NP core for a specific program takes place in a series of steps prior to the start of program execution. Specific steps are coordinated as follows:

1) A packet is assigned to a specific NP core by the control processor, which also determines which monitor should be assigned to the NP core based on the program required to process the packet.
2) If the monitor associated with the program is already available in the m monitors assigned to the cluster and is unused, the control processor sets the appropriate $monitor$ and $processor\ select$ multiplexer inputs to connect the monitor to the NP core and resets the monitor state logic. In our scalable architecture, this selection can take place in one clock cycle.
3) If the needed monitor is not currently loaded and available among the m monitors, the monitoring graph needs to be loaded by the control processor into one of the unused monitors associated with the cluster. The controller determines which monitoring graph should be replaced using a least recently used or other replacement policy. The controller streams monitoring graph data from the centralized monitoring graph storage to the target monitor. Since this transfer is unidirectional, it can take place via one of several high-fanout busses shown in Figure 6. The controller then sets appropriate multiplexer select inputs to connect the monitor to the NP core.
4) Once the processor-to-monitor physical connection is made, a start signal is sent to both the monitor and NP core starting both program processing and monitoring.

Clearly, loading monitoring information from centralized memory to monitoring graphs is time consuming so it is desirable to minimize how often this action is needed.

V. Runtime Analysis of SHMG

Given the Scalable Hardware Monitoring Grid design, there is a key question of how to assign programs to monitors. Intuitively, the assignment of monitors should reflect the processing workload. However, due to variations in workload, there may be a situation where more processors need to execute a particular program than monitors are available. In this case, some processors temporarily *block* (until a monitor becomes available, at which point they continue processing). We provide a brief analysis of the blocking probability of the system and the resulting throughput for different cluster configurations.

A. Monitor Configuration

We consider a processor/monitor cluster consisting of n processors and m monitors. The workload of the system consists of p different programs that each monitor may execute. To make the system practical, we require $m \geq n$ and $m \geq p$.

For each program i ($1 \leq i \leq p$), we denote the average processing time with t_i and the proportion of traffic that requires this program with q_i. We require $\sum_{i=1}^{p} q_i = 1$, 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. We also assume a fully utilized system, where no processor is idle. 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.

B. 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

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

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

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

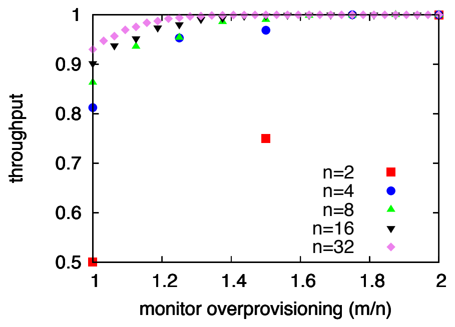

Fig. 9. Throughput depending on overprovisioning of monitors for different numbers of processors (n) per cluster.

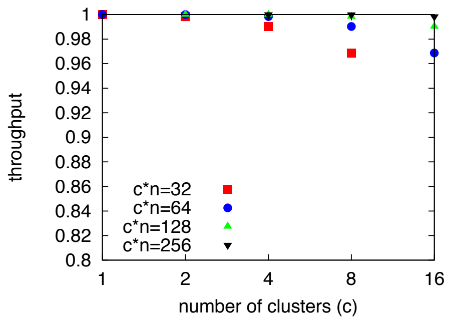

Fig. 10. Throughput depending on number of clusters for different numbers of total processors ($c \cdot n$).

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

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

C. 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 9 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 10. 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.

VI. Prototype Implementation and Evaluation

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

A. 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 [21]) and six hardware monitors (i.e., $n = 4$ and $m = 6$). The flexible, multiplexer-based interconnect shown in Figure 8 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 as described in [3].

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

B. Results

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

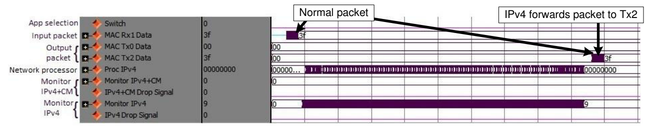

Fig. 11. Simulation waveforms showing correct forwarding of an IPv4 packet.

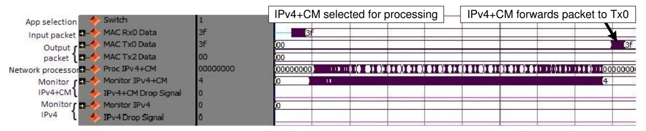

Fig. 12. Simulation waveforms showing forwarding of an IPv4+CM packet.

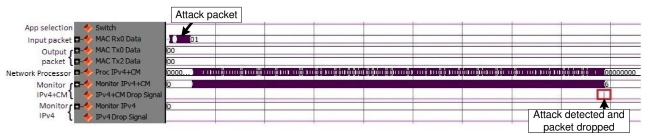

Fig. 13. Simulation waveforms showing identification of and recovery from an IPv4+CM attack packet.

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 11 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 12 shows the operation of a core on the IPv4+CM program. In this case, the packet is benign and no attack occurs. Figure 13 shows the processing of an attack packet in IPv4+CM. The monitor identifies the attack since the stack gets smashed and the control flow is redirected to code that differs from what the program analysis has determined as valid. The processor core is then reset and continues processing the next packet. The reset operation completes in two cycles and thus does not affect the throughput performance of the system (and cannot be used as a target for denial of service attacks). Other processor cores continue processing without being affected.

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 11) to IPv4+CM (Figure 12). The processor-to-monitor interconnect for the core that

TABLE I
RESOURCE UTILIZATION IN PROTOTYPE SYSTEM.

	Available in FPGA	DE4 interface	Network processors	SHMG monitors	SHMG interconn.
LUTs	182,400	33,427	15,025	816	96
FFs	182,400	36467	8,367	147	0
Bits	14,625,792	2,263,888	1,048,567	786,432	0

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

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

2) Resource Requirements: The resource requirements for the FPGA in our prototype system are shown in Table I. The lookup table (LUT), flip flop (FF), and memory resources (Bits) required for the network processor cores, monitors, switches and other circuitry are illustrated shown in Table I. A LUT is a 6-input, 1-output logic element that can perform any logic function of six inputs. Each monitoring graph can hold up to 4096 separate entries. The FPGA in the system is able to operate at 125 MHz. For this relatively small cluster size, the amount of logic needed for processor-to-monitor interconnection is less than 1% of the total logic needed for the monitors, cores, and processor-to-monitor interconnect.

VII. SUMMARY AND CONCLUSIONS

The use of general-purpose processors to implement packet forwarding functions in routers has opened the door for a new class of attacks in the data plane of the network. Prior work has shown examples for such attacks and their devastating effects. Hardware monitors that are co-located with network processors can provide defenses against these attacks. However, prior and related work has only focused on single-core monitors with static processing code. To provide practical protection for network processors, which are multi-core systems with highly dynamic workloads, we have presented our design of a Scalable Hardware Monitoring Grid. This monitoring system groups multiple processors and monitors into clusters and provides an interconnect to dynamically assign processor cores to monitors based on their current workload. We present the hardware design of an efficient interconnect for these clusters and show through analysis that even small configurations can achieve throughput performance. We also present the results from an FPGA prototype implementation that shows the correct operation of our system and the ability to perform dynamic assignment of processor cores to monitors. We show that the system can correctly identify attacks and recover the attack core so that it can continue processing. The system overhead for our monitoring system is less than 6% compared to the processor system. Thus, our Scalable Hardware Monitoring Grid provides an effective and efficient mechanism for defending network infrastructure from a new class of attacks.

VIII. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1115999. We gratefully acknowledge Altera Corporation's donation of the DE4 boards and Quartus software.

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] M. E. Lesk, "The new front line: Estonia under cyberassault," *IEEE Security & Privacy*, vol. 5, no. 4, pp. 76–79, Jul. 2007.

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

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

[7] 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.

[8] Q. Wu and T. Wolf, "Runtime task allocation in multi-core packet processing systems," *IEEE Transactions on Parallel and Distributed Systems*, 2012.

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

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

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

[12] 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.

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

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

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

[16] 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.

[17] R. G. Ragel, S. Parameswaran, and S. M. Kia, "Micro embedded monitoring for security in application specific instruction-set processors," in *Proc. of the 2005 international conference on Compilers, architectures and synthesis for embedded systems (CASES)*, San Francisco, CA, Sep. 2005, pp. 304–314.

[18] 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.

[19] 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 50th Design Automation Conference (DAC)*, Austin, TX, Jun. 2013.

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

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