High-Performance Hardware Monitors to Protect Network Processors from Data Plane Attacks

Harikrishnan Chandrikakutty, Deepak Unnikrishnan, Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA
{chandrikakut,unnikrishnan,tessier}@ecs.umass.edu

Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA
and IMDEA Networks, Madrid, Spain
wolf@ecs.umass.edu

ABSTRACT

The Internet represents an essential communication infrastructure that needs to be protected from malicious attacks. Modern network routers are typically implemented using embedded multi-core network processors that are inherently vulnerable to attack. Hardware monitor subsystems, which can verify the behavior of router's packet processing system at runtime, can be used to identify and respond to an ever-changing range of attacks. While hardware monitors have primarily been described in the context of general-purpose computing, our work focuses on two important aspects that are relevant to the embedded networking domain: We present the design and prototype implementation of a high-performance monitor that can track every single processor instruction using low memory overhead. Additionally, our monitor is capable of defending against attacks on processors with a Harvard architecture, the dominant contemporary network processor organization. We demonstrate that our monitor architecture provides no network slowdown in the absence of an attack and provides the capability to drop attack packets without otherwise affecting regular network traffic when an attack occurs.

1. INTRODUCTION

The Internet is a critical infrastructure component in today's society. Many aspects of personal communication, business transactions, entertainment, digital government, etc. rely on the availability and correct operation of the Internet. With the continuing growth of the Internet, there are ongoing technical challenges to meet emerging needs for networking functionality, throughput performance, reliability, and security. To address these challenges, it is necessary to improve the security of networks, including the router devices that constitute the core of the network, with limited compromise in other networking goals.

In this work, we focus on the security of packet processing functions that are necessary to handle packet forwarding on a network router. The packet processing component of a modern router is typically implemented using a network processor (NP) system. A network processor has multiple simple embedded processor cores that can be programmed to handle network traffic. When changes are necessary, the software on this network processor can be changed to adapt the operation of the router. Unlike traditional routers that have been based on application-specific integrated circuit (ASIC) technology, network-processor-based routers can be adapted in their functionality. However, this flexibility raises an interesting security problem: routers that use software-programmable packet processors are vulnerable to attacks [1]. Vulnerable packet processing code can be exploited using malformed data packets to launch an in-network denial-of-service attack. This type of attack on contemporary NPs is addressed by this paper.

To reduce the vulnerability of packet processors in routers (as well as embedded processors in general), digital circuits that monitor run-time processor operation have been proposed [1–3]. These hardware monitors use information about correct processor software execution to track the instructions executed by a processor core. If an attack on the processor occurs, a deviation from expected, programmed behavior is detected and a recovery process is initiated. Compared to software-based protection mechanisms (e.g., virus scanners), hardware-based monitors require less performance overhead and react faster. In the networking domain, low overhead and fast detection speed are particularly important. Therefore, there have been ongoing efforts to further improve NP protection mechanisms to better support the applications supported by NPs and the processor organization typically exhibited by NPs.

In this paper, we address two key problems that have not been previously addressed in protecting network processors from software attacks. An effective NP monitoring system must verify every instruction that is executed by the processor and thus needs to operate at very high speeds. This instruction-based monitoring operation can be viewed as a finite automaton with a fixed number of acceptable paths. Prior work in hardware monitoring [1, 2] has been based on non-deterministic finite automaton (NFA) implementations, which potentially require high memory bandwidth when tracking multiple parallel states, or coarse verification at the level of basic blocks [3], which may not detect attacks before they are executed. For our first contribution, we present an instruction-level monitoring solution for NPs based on deterministic finite automaton (DFA) implementation, which overcomes the shortcomings of both prior techniques.

Prior work in embedded NP security has assumed a von Neumann processor architecture with a combined instruction and data memory, where an attack can execute code from the processor stack [1]. Most network processors, however, are based on Harvard architectures that separate instruction and data memory making stack-based code execution impossible. Therefore, it is questionable that data plane attacks are even possible in networks. For our second contribution, we present an attack example that demonstrates the

existence of Harvard architecture attacks and we show that our monitoring system is effective in defending NPs against them.

The specific contributions of our paper are:

1. Design of a high-performance hardware monitoring system for NPs: Our pipelined design can perform instruction verification with a single memory read per instruction and thus can operate at speeds sufficient to maintain line rate networking data transfer.

2. Algorithm for construction of a deterministic monitoring graph: We present a method to convert the monitoring graph of NP instructions, which initially is non-deterministic due to control-flow changes (e.g. branches), into a deterministic automaton. The representation of the DFA is compacted to allow a highly efficient implementation in the hardware monitor.

3. Demonstration of an attack on and defense of a Harvard architecture network processor: We demonstrate an in-network attack through the data plane of the network that exploits an integer overflow vulnerability to smash the processor stack and launch a return-to-library attack. This attack propagates the attack packet and crashes the processor system. We also show that our hardware monitor is effective in defending against this attack and allowing for continued NP-based router operation after attack identification and recovery.

The remainder of the paper is organized as follows. Section 2 discusses related work. The overall system architecture is introduced in Section 3. The construction of the monitoring data structure is presented in detail in Section 4. Section 5 describes an example attack that we use in our prototype system implementation described in Section 6. Section 7 presents evaluation results that show the effectiveness of our design. Section 8 summarizes the paper and offers directions for future work.

2. RELATED WORK

Programmability in the packet processing systems of routers has been used increasingly widely over the past decade. Most major router vendors employ network processors in their products (e.g., Cisco QuantumFlow [4], Cavium Octeon [5]). While the programmability of these devices is hidden from network users, it is used by vendors to extend system functionality. It can be expected that routers will continue to have programmable packet processing components, especially with network virtualization [6] emerging as promising technology for the future Internet.

While network security as a whole has received much recent attention (e.g., end-system vulnerabilities leading to botnets [7], worm propagation [8], etc.), there has been little focus on vulnerabilities in the networking infrastructure itself. Cui et al. [9] have surveyed vulnerabilities in the control plane of networks, where an attacker can potentially gain access to the router system. In the data plane, Chasaki et al. [1] have shown an example of how a simple integer overflow vulnerability can be exploited to launch a denial-of-service from within the network. In this case, a single malformed User Datagram Protocol (UDP) packet triggers the vulnerability, changes the network processor's operation, and causes a flood of attack packets to be sent by the system. We adapt this attack example to a processor system based on a Harvard architecture in our work to demonstrate the effectiveness of our monitoring system in detecting and stopping such data plane attacks in a practical networking environment.

Table 1: Comparison of Monitoring Approaches.

Monitor	granularity	implementation	underlying architecture
Chasaki et al. [1]	instruction	NFA	von Neumann
Arora et al. [3]	basic block	DFA	von Neumann
This paper	instruction	DFA	Harvard

Protection mechanisms for embedded processors have been proposed based on hardware monitors in general [1–3,10–13]. These monitors differ by the level of monitoring granularity (function calls, basic blocks, individual instructions) and if they require changes to source code or if they are based on program binaries. We only focus on approaches that do not require changes to the processor binaries. The main novelty of the monitoring system we present in this paper is highlighted in Table 1. In contrast to related work, we can monitor at the level of individual instruction and do so using a DFA, which can be implemented with higher performance.

3. HARDWARE MONITOR SYSTEM ARCHITECTURE

The system architecture of the network processor system with security monitor is shown in Figure 1. The network processor shown on the left of the figure is based on a conventional Harvard architecture with separate data memory for network packets and processing state and instruction memory for packet processing code. For simplicity, only a single processor core is shown; the system can easily be extended for multiple processor cores. The processing monitor on the right side of the figure verifies the operation of the processor instruction-by-instruction. For every instruction that is executed on the processor core, a hash value of the executed operation is reported to the monitor. The monitor uses the comparison logic to compare the reported hash value to the information that is stored in the monitoring graph. The monitoring graph is derived by offline analysis of the packet processing code binary.

Any attack on the system necessarily needs to change the operation of the processor core (otherwise the attack is not effective). This deviation leads to the processor reporting hash values that do not match with the monitoring graph. The comparison logic can detect this deviation and reset the processor in response. In the context of networking, such a reset and recovery operation is very simple: The current packet is dropped (i.e., the packet buffer is cleared), the processing state is reset (i.e., the stack is reset), and processing continues with the next packet. Since most packet processing operations are not stateful and there is no guarantee that packets are reliably delivered, no further recovery actions are necessary.

The monitoring graph used by the hardware monitor is a state machine, where each state represents a specific processor instruction. The state machine is derived from the packet processing code as illustrated in Figure 2. Each processor instruction corresponds to a state. The edges between

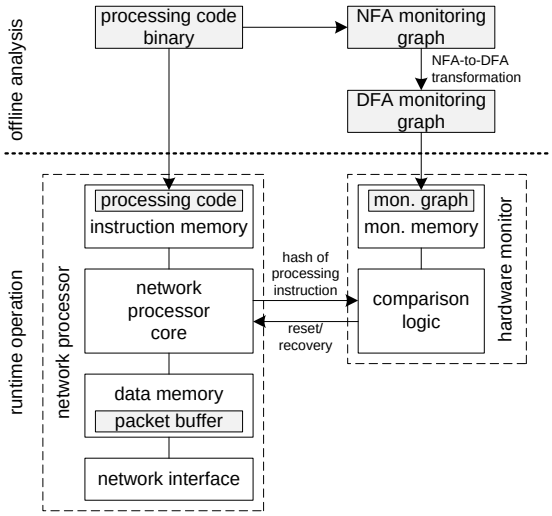

Figure 1: System architecture of network processor with security monitor.

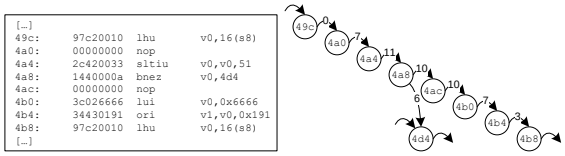

Figure 2: State machine generation from processing binary.

states are labeled with information relating to next valid instruction that can be executed after the current instruction. In case of control flow operations, there may be multiple outgoing edges from each state (each being a valid transition). In our system, we use a 32-bit processor (i.e., open source embedded Plasma processor based on the MIPS instruction set). The monitoring system uses a 4-bit hash of the next instruction to label edges in the monitoring graph (as has been recommended in [2]). A hash (instead of the full 32-bit instruction) is used to reduce the size of the monitoring graph and thus to reduce the implementation overhead of the hardware monitor while still allowing instruction-by-instruction monitoring. The use of a hash (or any other method that uses a many-to-one mapping), however, leads to two fundamental problems:

- Attack detection ambiguity: The many-to-one mapping that occurs in a hash function of the monitor may make it possible for an attacker to remain undetected. This would require that the attack performs operations that lead to a valid sequence of hash values that matches the monitoring information of valid code. Mao et al. have shown that this probability decreases geometrically with the length of the attack code and thus is unlikely to lead to practical attacks [2] (in particular when the hash function is not known to the attacker). We do not consider this issue further in this paper.

- Nondeterminism during monitoring: The many-to-one mapping also leads to nondeterminism in the monitoring graph. There may be a control flow instruction where each of the next instructions has the same hash value. As a result, the corresponding node in the monitoring graph has two outgoing edges with the same hash value (as illustrated in Figure 3). Since this nondeterminism can continue for multiple such control flow operation, it can lead to complex implementations [1], potentially slowing monitor performance.

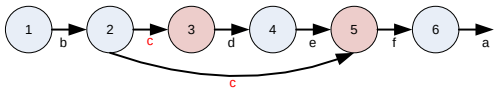

Figure 3: Nondeterministic monitoring graph.

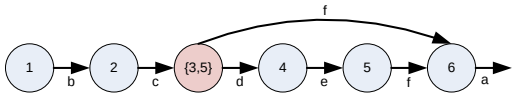

Figure 4: Deterministic monitoring graph after NFA-to-DFA conversion.

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

4. DETERMINISTIC PROCESSOR MONITORING

To realize a deterministic instruction-level monitor, we first convert the NFA monitoring graph described in the previous section to a DFA monitoring graph. We then describe how to implement a monitoring system that uses this DFA graph.

4.1 Construction of Deterministic Monitoring Graph

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

To convert an NFA to a DFA, a standard powerset construction algorithm can be used [14]. This algorithm computes all possible state sets in which the automaton can be (i.e., the powerset). Based on the powerset, a DFA is then constructed. Figure 4 shows the DFA that corresponds to the NFA shown in Figure 3. Note that state `{3,5}` represents the sets of states to where state 2 can branch when hash value `c` is observed.

One potential problem with NFA-to-DFA conversions is that the number of states in the DFA can grow exponentially over the number of states in the NFA. However, the

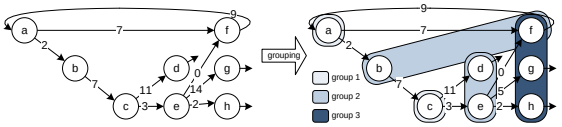

Figure 5: Grouping of DFA states.

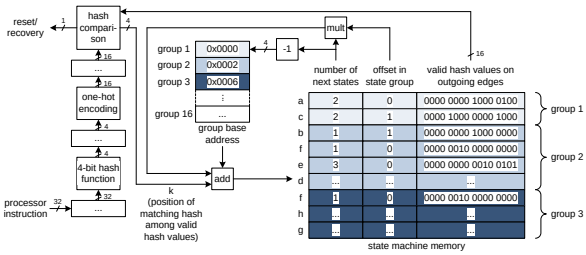

Figure 6: Memory representation of DFA monitoring graph.

monitoring NFAs constructed from binary code do not exhibit this pathological behavior. Our experiments indicate that this increase is small and does not lead to drastically larger state machines (see Section 7). Thus, this approach is effective for creating deterministic hardware monitors.

4.2 Implementation of Monitoring System

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

The information that needs to be stored in the monitoring memory is illustrated on the left side of Figure 5. Each state represents an instruction and an outgoing transition edge from this state represents the hash value of the next expected instruction in the execution sequence. For example, state `c` has two next states, `d` and `e`, with hash values 11 and 3, respectively.

A naïve way to store the state machine in RAM would be to store each state and all its possible edge transitions. This would require 2^h entries per state for an h-bit hash. Since most states have only one or two outgoing edges, a large number of edge transitions would never be used, leading to inefficient memory use. Assuming that only two outgoing transitions exist for each state is also not feasible due to the cases where powerset construction creates states with up to 2^h outgoing edges. Finally, for performance reasons we should only use one memory access per state transition, which precludes a design where states with more than two outgoing edges are handled as special cases.

Our main idea to compactly represent DFA states with varying numbers of outgoing edges is to encode all the necessary information in a single table entry and to group states by the number of outgoing edges. The main challenge in achieving compactness is to allocate exactly the amount of memory that is needed for each state to store next state information while still being able to index this memory without degrading to linear search. In our representation, we group states together if they have the same previous state. A state belongs to group g if the previous state has g outgoing edges. For a monitor with a 4-bit hash value, there are 16 possible groups. For example, in Figure 5 on the right side, groups are shown with different colors. Note that a state can belong to multiple groups (e.g., state `f` belongs to group 2 (because `a` has two outgoing edges, one to `b` and one to `f`) and to group 3 (because `e` has three outgoing edges)).

The memory layout and basic operation of our DFA monitor system is shown in Figure 6. The memory contains tuples of {number of next states, offset in state group, valid hash values on outgoing edges} and is logically divided into groups. The base addresses for each group are stored in register file with 16 entries. Within a group the sets of states that share the previous state are grouped together (e.g., `b` and `f` are together and `d` and `e` are together). Within a set, states are ordered by the hash value on their incoming edge (e.g., `e` before d because hash value 3 is smaller than hash value 11). The hash comparison block performs two functions: it determines if the one-hot coded hash bit is set in the 16-bit value read from memory and it determines k, which is the position of the matching hash value among the valid hash values read from memory.

To illustrate the operation of the monitor, we describe an example transition. Assume the monitor is in state `a` and the processor reports an instruction that leads to a hash value of 7. To perform the transition, the memory row labeled `a` is read. The tuple in this row indicates that there are two outgoing edges. The valid hash values of these two edges are stored in the 16-bit vector. To verify that the transition is valid, the hash comparison unit checks if bit 7 is set in the bit vector (which it is). If this bit is not set, then an invalid transition takes place, indicating an attack, and the processor is reset. After the check, the next state (i.e., state `f`) in the DFA needs to be found in memory. To determine the address of that state, the base address of the group of the next state is looked up in the register file (i.e., 0x0002 since the next state belongs to group 2). To this base address, the product of the set size (i.e., group number) and the offset in the state group is added (to index the correct set within the group). Finally, k is added, which is the position of the matching hash in the bit vector (in our case 1 since 2 is the first matching hash (i.e., k=0) and 7 is the second matching had (i.e., k=1)). Thus the memory location of state `f` is 0x0002 + 2×0 + 1 = 0x003.

Note that any state transition takes only one memory read from state machine memory and a lookup into a fixed-size register file. The DFA is represented compactly without wasting any memory slots (states shown with dots in Figure 6 point to other states not shown in our example). Thus, this representation lends itself to a high-performance implementation.

5. HARVARD ARCHITECTURE ATTACKS

Even though general memory error techniques (integer overflow, heap overflow etc.) cannot be used to generate code injection attacks, Francillion et al. [15] demonstrated

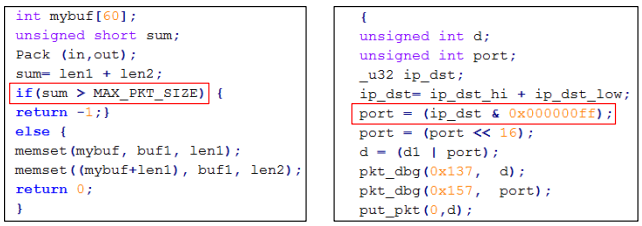

Figure 7: Vulnerable application code.

that code injection attacks are still feasible on a Harvard architecture processor using a return-oriented programming technique. Here, an attacker takes control of return instructions in the stack to chain attack code from an existing library function. Since the code is already present in executable memory, the attack will not be prevented from running. In this section, we describe how such an attack can be constructed for the networking environment and how our monitor can detect it.

Figure 7 shows a portion of congestion management protocol (CM) and IPV4 packet forwarding application used to build an attack on the network processor system. The congestion management protocol inserts a custom protocol header in the packet header space between the IP header and the UDP header. During this operation, the code needs to make sure the new packet size does not exceed the maximum datagram length (the boxed instruction in the CM code). Exploiting an integer overflow vulnerability, the boundary check in the CM code can be circumvented and the stack can be smashed. To do so, an attacker sends a malformed UDP packet with a size 0xfffe (decimal value 65534), which will pass the maximum packet size check (since 65334 + 12 = 10, due to integer overflow). As a result, the packet payload is copied over the stack. The packet payload of the attack packet is crafted in such a way that the return address is overwritten to direct the control flow to the IPv4 packet forwarding application (which is library code on the processor core) and the value of *ip_dst_low* field is 0xff. The port information gets updated with this value (the boxed instruction in the IPv4 code), forwarding the attack packet to *all* the outgoing ports and then crashing the processor system. As a result, the attack packet gets forwarded to all outgoing interfaces before the system crashes, thus propagating the attack through the network.

Since our hardware monitor has no valid edge between the states in the middle of the CM application and the IPv4 application, this attack is detected. As soon as the control flow changes, the hash values reported by the processor no longer match the monitoring information and the system is reset, dropping the malicious packet.

6. PROTOTYPE SYSTEM IMPLEMENTATION

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

To run networking code on the processor plus monitor system, the code is first passed through a standard MIPS-GCC compiler flow to generate assembly-level instructions. The output of the compiler allows for the identification of branch instructions and their target addresses. In our current implementation, all possible branch targets and return instructions are analyzed at compile time. Then, the NFA-to-DFA conversion starts with a non-deterministic NFA representation obtained from the compiler information. Through powerset construction, a DFA is constructed. This DFA is then converted into a memory initialization file using the process described in Section 4 and is loaded into the monitor when the processing binary is installed in the processor. To evaluate our system, four benchmarks from the NPbench suite [16] were processed with this flow.

Table 2: Evaluation of monitoring approaches for our new DFA approach and a previous NFA-only approach. The maximum number of memory accesses for our approach is 1 for all benchmarks.

		Chasaki [1]		Ours		
Netw. application	No. of instr.	NFA states	Max. mem. access	DFA states	Mem. entries	Mem. overhead
frag	573	573	3	592	627	9.4%
mtc	2427	2427	3	2460	2584	6.4%
red	802	802	2	808	857	6.8%
wfq	905	905	2	921	978	8.0%

7. EVALUATION RESULTS

7.1 Monitoring Graphs

The results of generating instruction-level monitoring graphs for both our approach and a previous approach [1] are illustrated in Table 2. The number of entries in the state machine memory (Figure 6) for each benchmark is shown in the *Mem. entries* column. A clear benefit of the new approach is speed. In all cases, only one access to the monitor memory is required for any benchmark (including the four shown here). The previous NFA-based approach requires up to three memory accesses for the benchmarks tested and potentially up to 16 for other benchmarks. The conversion from an NFA to a DFA does incur a memory overhead of 7.7% on average for the benchmarks.

7.2 Monitoring Speed and Effectiveness

Our network processor and monitoring system were successfully implemented on the DE4 platform. The lookup table (LUT), flip flop (FF), and memory resources required for the network processor core, monitor, and other interface

Table 3: Resource Utilization

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

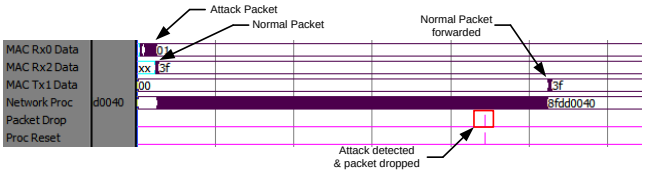

Figure 8: Simulation waveforms showing the identification of an attack packet and the successful forwarding of the subsequent packet. This behavior was confirmed using hardware.

circuitry for the router (e.g. buffers, input arbiter, queuing control, etc) are shown in Table 3. The NP memory includes space for up to 4096 monitor memory entries. All circuitry operated at 125 MHz, the same clock speed for the system without the monitor. Experiments in simulation and in the lab on FPGA hardware showed that the processor is able to forward packets ranging in size from 64 to 1500 bytes per packet at the same rate under monitoring as without monitoring (e.g. no slowdown for monitoring). For hardware experiments, packets were generated and transmitted to the DE4 with the NP and monitor at a 1 Gbps line rate by a separate DE4 card serving as a packet generator. This same card was used to receive the processed packets from the card with the NP.

In a final experiment, we tested the ability of the monitor-based system to detect and recover from an attack. The vulnerable application code shown in Figure 7 was implemented and used with the NP to send copies of a packet to all ports of the router and then crash the router. We confirmed this behavior for a system without a monitor both in simulation and in hardware. As shown in Figure 8, after the monitor was added to the system, the attack packet was successfully identified, the NP was reset, and subsequent regular packets were routed successfully. This behavior was verified using our DE4 hardware setup.

8. SUMMARY AND FUTURE WORK

The effective use of the Internet depends on reliable network routers that are impervious to attack. In this paper, we have described a high-performance monitor for a network processor that requires only a single memory lookup per network processor instruction. This single memory lookup is maintained regardless of the complexity of the NP program using an NFA-to-DFA translation of the monitoring graph. Our monitor, which tracks individual NP instructions, has been verified in hardware using an NP with a Harvard architecture. The presence of monitoring does not slow down NP operation since it is performed outside of the operational paths of the NP. In the future, we plan to evaluate our monitoring approach using a multi-core network processor.

9. REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

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

[6] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet impasse through virtualization," *Computer*, vol. 38, no. 4, pp. 34–41, Apr. 2005.

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

[8] D. Moore, C. Shannon, and J. Brown, "Code-Red: a case study on the spread and victims of an Internet worm," in *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement*, Marseille, France, Nov. 2002, pp. 273–284.

[9] 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.

[10] R. G. Ragel and S. Parameswaran, "IMPRES: integrated monitoring for processor reliability and security," in *Proc. of the 43rd Annual Conference on Design Automation (DAC)*, San Francisco, CA, USA, Jul. 2006, pp. 502–505.

[11] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, and N. Memon, "SAFE-OPS: An approach to embedded software security," *Transactions on Embedded Computing Sys.*, vol. 4, no. 1, pp. 189–210, Feb. 2005.

[12] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," in *ACM Conference on Computer and Communication Security (CCS)*, Alexandria, VA, Nov. 2005, pp. 340–353.

[13] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/high-performance embedded systems: the SAFES perspective," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 144–155, Feb. 2008.

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

[15] A. Francillon and C. Castelluccia, "Code injection attacks on Harvard-architecture devices," in *Proc. of*

the 15th ACM Conference on Computer and Communications Security (CSS), Alexandria, VA, Oct. 2008, pp. 15–26.

[16] 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.