Securing Network Processors with High-Performance Hardware Monitors

Tilman Wolf, *Senior Member, IEEE*, Harikrishnan Chandrikakutty, Kekai Hu, Deepak Unnikrishnan and Russell Tessier, *Senior Member, IEEE*

Abstract—As the Internet becomes integrated into nearly all aspects of everyday life, its reliability grows in importance. This vital communication resource, which has become an inviting target for attackers, must be protected with the same vigor as the end-systems it interconnects. Recent trends in network router architecture towards programmability and flexibility have increased the susceptibility of communication hardware to software attacks which modify intended data processing and forwarding functions. Contemporary routers typically feature network processors, whose protocol processing functions are determined via software. Prior work has shown that these general-purpose software-based processing systems can be attacked with data packets sent through the Internet. As a defense mechanism, the correct functionality of a network processor can be verified by a hardware monitor that observes processor operation and compares it to expected behavior. In the event of an attack, the monitor can interrupt the network processor, suppress malicious behavior, and reset the processor to a usable state for processing of subsequent traffic. In this work, we present several significant advances in hardware monitoring for network processors. A low-overhead monitor architecture that evaluates correct network processor operation in real-time on an instruction-by-instruction basis is described and tested. The monitor is shown to effectively prevent stack smashing attacks on processors that use a Harvard architecture, a widely used network processor configuration. Through experimentation, we show that our approach to hardware monitoring does not affect data plane packet throughput. In the event of an attack, malicious packets are dropped while packets of regular network traffic proceed through the network unaffected. A full evaluation of monitor architectural parameters is provided to create an optimized monitor design.

Index Terms—computer network, security, hardware monitor, control flow, deterministic finite automaton, Harvard architecture.

1 Introduction

Over the past forty years, the Internet has grown from a modest research network to a critical communication resource used by billions of people across the world. Indeed, the reliable operation of this resource has become as critical to commerce, personal interaction, and government activities as traditional utilities, such as the power grid. 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.

To address this need, we have developed techniques to secure the processing of packets in the data plane of network routers. In contemporary routers, network processors (NPs) are frequently used to perform packet processing. These embedded processors (Figure 1) typically contain multiple simple RISC-based processing cores that can efficiently manipulate data packets but are potentially vulnerable to attacks initiated by data packets. The functionality of these programmable processors can easily be updated via software updates to provide a broad range of router functionality. However, this programmability leads to a significant drawback; the security of the router is only as strong as the software that programs it.

Recently, it has been shown that network processors can be successfully attacked to generate denial-of-service attacks [1]. Using strategically crafted data packets, these attacks exploit weaknesses in the packet processing software of the network processor. Specifically, it is demonstrated that a malicious packet can exploit errors in packet size boundary checking software to overwrite a network processor's stack. This stack smashing attack can then be used to modify the return address of the NP program, forcing control flow jumps to user-supplied code or to library functions already present in the system that can be used in unintended and malicious ways. An important point to note is that this type of attack vector is entirely in the *data plane* of the network. That is, the attacker does not hack into the control interface of the router, but merely transmits a malformed data packet. Thus, these "in-network" attacks cannot easily be defended against with conventional security mechanisms. Instead, our new high-performance hardware monitoring approach is able to quickly identify this type

T. Wolf, K. Hu, and R. Tessier are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, USA. H. Chandrikakutty is with Juniper Networks, Westford, MA, USA. D. Unnikrishnan is with Altera Corporation, San Jose, CA, USA. Contact e-mail: wolf@ecs.umass.edu.
Part of this work was done while T. Wolf was at IMDEA Networks, Madrid, Spain.
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.

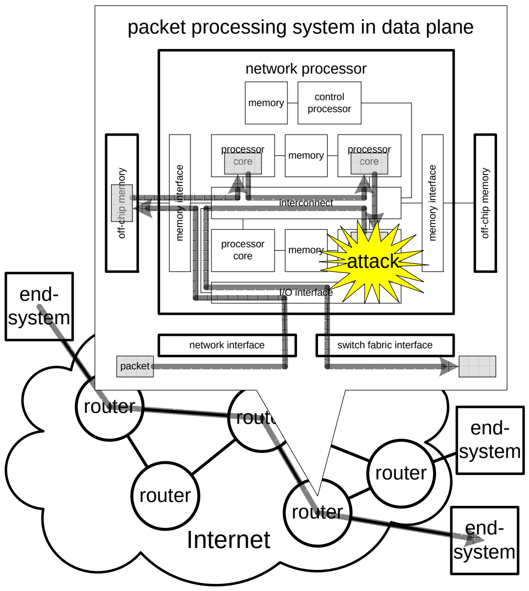


Fig. 1. Attack on packet processing system in network router data plane.

of attack, drop the offending packet, and reset the router to continue processing normal traffic.

Previous efforts have shown that specialized digital hardware can be used to monitor network processor operation and identify deviations from expected behavior [1]–[3]. This hardware typically examines the sequence of executed instructions by a processor core to determine if expected program sequencing is exhibited. Deviations from the expected instruction flow indicate that an attack is in progress and the monitor generates control signals to initiate processor core recovery. For most processor cores, hardware-based monitors, rather than software-based detection approaches, are needed since monitors operate at hardware speeds external to the core, thus avoiding packet processing slowdowns. As networking speeds increase, the need for rapid identification of attacks using a minimum amount of additional monitoring hardware is apparent. In particular, these mechanisms must be tuned to the processor configurations most commonly exhibited by contemporary NPs.

The research described in this manuscript addresses several important monitoring issues for network processors that must be considered to keep NPs safe from packet-based attacks. For comprehensive protection, every instruction executed by the NP should be validated in real-time, necessitating a high-performance monitoring solution. In general, the tracking of instructions is easily modeled as a finite state machine with a finite number of known paths. Although a non-deterministic finite automaton (NFA) can be used to model instruction sequencing for hardware monitoring [1], [2], this approach can require numerous memory lookups to differentiate multiple parallel states, limiting performance. Alternatively, monitoring can be more quickly performed by tracking coarse basic blocks instead of instructions [3], although this approach can exhibit a lag between when an attack starts execution and when it is identified. Our new approach, based on a deterministic finite automaton (DFA), provides an advance over both of these previous techniques.

It has been previously shown that network processors with combined data and instruction memory (von Neumann architecture) are susceptible to attacks that write executable code to the processor stack [1]. However, contemporary network processors generally use separated instruction and data memories (Harvard architecture) for increased code security and performance. These architectures make it impossible to execute code from a stack located in data memory, drawing into question whether data plane attacks are feasible in these types of architectures. In this work, we show that data plane attacks on Harvard architecture NPs are feasible and a new instruction-level hardware monitoring system can be used to defeat them.

The specific contributions of our paper are:

1) Design of a high-performance hardware monitoring system for NPs: Our monitor 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 for 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.
4) A full evaluation of architectural parameters needed to build a hardware-based monitor for a broad collection of nine network processing benchmarks.

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]), 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. 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.

Protection mechanisms for embedded processors have been proposed based on hardware monitors in general [1]–[3], [10]–[14]. These monitors differ by the level of monitoring granularity (function calls, basic blocks, individual instructions, system calls) 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.

A preliminary discussion of the main ideas presented in this manuscript appeared in [15]. This previous work introduced the idea of per-instruction monitoring using a DFA-based monitor and protection for network processors based on Harvard architectures. In this work we explore a spectrum of possible implementation choices for our monitor, including the use and costs of a diverse set of hash functions in the monitor implementation and a system-level model for NP and monitor implementation. The number of benchmarks used for analysis is expanded from four to nine to better quantify monitoring overheads, including the effects of monitoring on network throughput. This manuscript also includes a discussion of the relationship between the storage requirements of monitoring information for a monitor and the control flow details of a monitored application. A description of the conversion of an application binary into a finite automata and a system-level model, including the monitor and NP, are also provided.

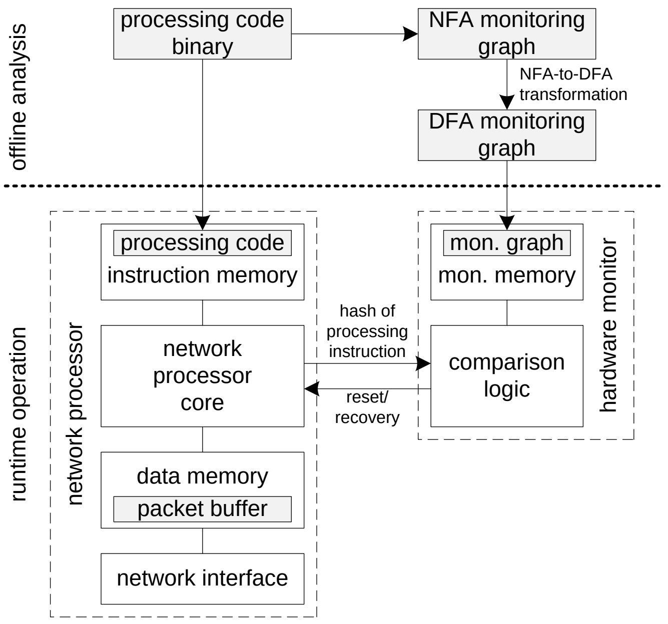


Fig. 2. System architecture of network processor with security monitor.

3 Hardware Monitor System Architecture

The system-level architecture of the network processor system with security monitor is shown in Figure 2. 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 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.

```
[…]
49c:     97c20010  lhu       v0,16(s8)
4a0:     00000000  nop
4a4:     2c420033  sltiu     v0,v0,51
4a8:     1440000a  bnez      v0,4d4
4ac:     00000000  nop
4b0:     3c026666  lui       v0,0x6666
4b4:     34430191  ori       v1,v0,0x191
4b8:     97c20010  lhu       v0,16(s8)
[…]
```

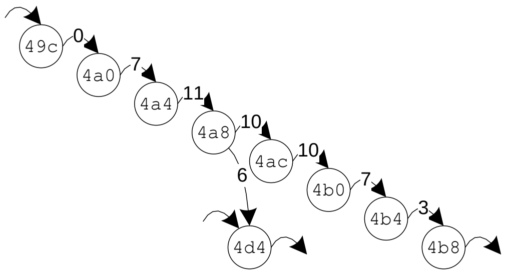


Fig. 3. State machine for hardware monitor generated from processing binary.

3.1 Monitoring Graphs

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 3. Each processor instruction corresponds to a state. The edges between 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 sequence of hash values that matches the monitoring information of valid instructions. Mao *et al.* have shown that this probability decreases geometrically with the number of instructions in the attack software code and thus is unlikely to lead to practical attacks [2]. We do not consider this issue further in developing the monitor for 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 5). Since this nondeterminism can continue for multiple such control flow operations, it can lead to complex implementations [1], potentially slowing monitor performance.

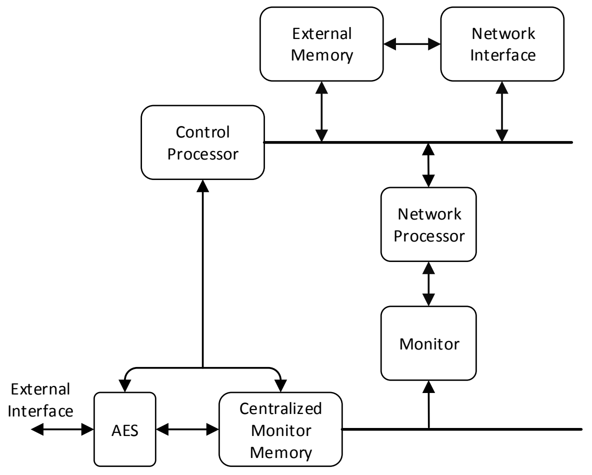


Fig. 4. Model of network processor with monitoring system.

In Section 4, 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.

3.2 Model of Network Processor and Monitoring System

The model of an NP system, including a monitor, is shown in Figure 4. System operation is coordinated by a control processor that forwards incoming packets to the NP. Off-chip external memory provides bulk storage for the packets and programs used by the NP. The NP, control processor, monitor, and system interface ports are interconnected using an on-chip communications infrastructure. The same control processor used for the assignment of programs and packets to the NP is used for the loading of monitoring graphs into the monitor. Monitoring graphs for all applications executed by the NP are stored on-chip in a centralized monitor memory. For some network processor systems this storage could be implemented in non-volatile storage (e.g. EEPROM). Alternatively, the storage could be implemented in DRAM with monitoring graphs downloaded to the system each time power is applied. Encryption is used to cipher monitoring graph information when it is input into the system using the external interface port. A standard AES core is used to decipher the monitoring graphs and place them in centralized monitor memory.

4 Deterministic Processor Monitoring

To realize a deterministic instruction-level monitor, we first convert assembly instructions into a nondeterministic finite automata (NFA) monitoring graph. This graph is then converted into a DFA monitoring graph. The monitoring system uses this DFA graph to dynamically verify correct instruction execution.

4.1 Construction of Nondeterministic Monitoring Graph

As described in the previous section and shown in Figure 3, each instruction represents a state in an NFA. For monitoring to operate correctly, *all* possible executions paths through the program must be determined from the application binary. In our system, the conversion of instructions to NFA states is performed via a static analysis of the binary (e.g., at compile-time or after compilation) using a breadth first traversal of the program.

For instructions that do not alter control flow (e.g. *add*, *sub*), the next state in the NFA simply represents the next instruction in the code sequence, which can be quickly determined by scanning through the program.

For control flow operations, the determination of all possible next states requires the determination of all potential jump destinations during static analysis. We distinguish between the following cases:

- Direct jump instructions: For a jump instruction with a specified target address, the next NFA state for the instruction is the state associated with the target instruction.
- Branch instructions: For branches in which a target address is specified in the instruction, two next states in the NFA are possible (as shown for the *bnez* instruction in Figure 3). The states represent the next sequential instruction in the code and the branch target instruction.
- Indirect jump instructions: A more complicated situation occurs for indirect jumps. Our NFA generation approach supports indirect jumps, where the jump target address is stored in a register (e.g. MIPS instruction *jr $s3* indicates a jump to the address stored in register $s3) if all potential register values (jump targets) can be statically determined. For example, all return addresses for a subroutine (the next instruction after a call to the subroutine) can be determined by examining all calls to the subroutine. The collection of next states for the indirect jump to a subroutine return address is the collection of states representing these return address instructions. For other indirect jumps, it is possible to statically determine potential jump targets by observing where fixed instruction addresses are assigned to registers in the code. These addresses and subsequent address manipulations are tracked to determine all possible targets. For example:

```
addi $s0, $zero, target   // target -> $s0
addi $s1, $s0, 4          // target+4 -> $s1
jr $s1                    // jump to target+4
```

  Since *target* is an address label, it is flagged during static analysis and the register(s) to which its address (or modified versions) are assigned are tracked in case they are later used for indirect branches.

  Note that our NFA generation (and overall monitoring) approach does not work if the location of an indirect jump target address is reliant on an input data value determined at *run-time*. This issue could be addressed by performing code simulation to determine all possible dynamic jump target addresses, an approach which we leave for future work. None of the nine network processing applications we evaluate in Section 7 used input data to determine indirect jump targets, hence all NFA next states could be straightforwardly determined.

As mentioned in Section 3.1, a 4-bit hash of the next instruction is used to label the edge between states. This hash value is generated using the 32-bit instruction associated with the next state as a hash function input. Four hash functions were considered for use: an arithmetic sum of all 32 instruction bits, a modulo sum of all eight 4-bit nibbles, an XOR of the eight 4-bit nibbles, and an OR/XOR of the eight 4-bit nibbles. Further details on the implementation of these hash functions and their effectiveness is presented in Section 7.2.

4.2 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 [16]. This algorithm computes all possible state sets in which the automaton can be situated (i.e., the powerset). Based on the powerset, a DFA is then constructed. Figure 6 shows the DFA that corresponds to the NFA shown in Figure 5. 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 monitoring NFAs constructed from binary code for network processing applications (including the nine applications we examined) do not exhibit this pathological behavior since the number of control flow instructions

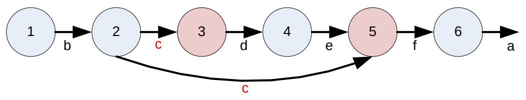


Fig. 5. Nondeterministic monitoring graph.

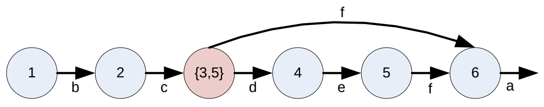


Fig. 6. Deterministic monitoring graph after NFA-to-DFA conversion.

(e.g. jumps and branches) in the applications is modest and the number of potential branch targets is small. These factors limit the need for additional DFA states which arise from hash value collisions on control flow instruction targets. Our experiments indicate that the increase in states from NFA-to-DFA is small and does not lead to drastically larger state machines (see Section 7). Thus, this approach is effective for creating deterministic hardware monitors.

4.3 Monitor Implementation

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 7. 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 implementation compactly represents DFA states with varying numbers of outgoing edges to encode all the necessary information in a single table entry and to group states by the number of outgoing edges. The approach achieves compactness by allocating 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 7 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 8. 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 a 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 hash (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 8 point to other states not shown in our example). Thus, this representation lends itself to a high-

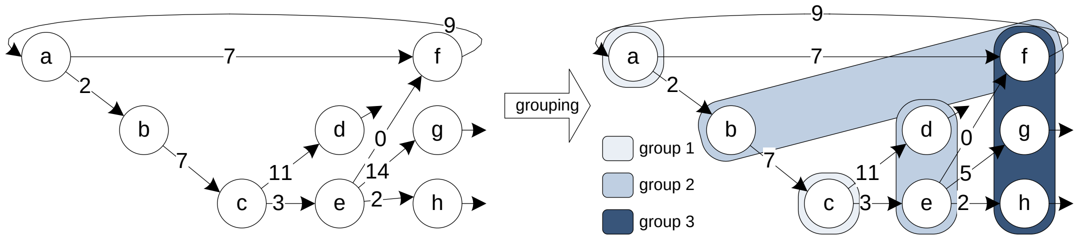


Fig. 7. Grouping of DFA states.

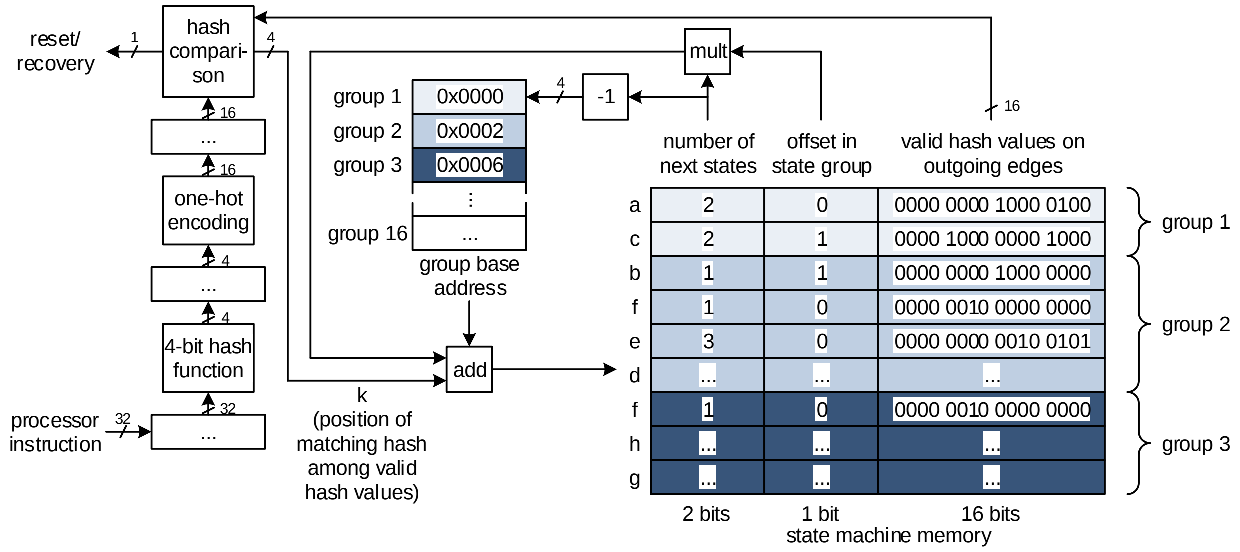


Fig. 8. Memory representation of DFA monitoring graph.

performance implementation.

The required size of the monitoring memory varies from application to application depending on the monitoring graph that is stored. The size of each field in the monitoring memory can be determined using parameters associated with the specific DFA. The required size of the monitoring memory and its associated fields for an application are determined from the DFA as follows:

- **Required rows in the monitoring memory** - At least one row is required for each DFA state in the monitoring graph. If a state is a member of multiple sets of states, one row is needed for each set in which the state is a member.
- **Number of groups** - The number of state groups implemented in the memory is equal to the number of group types of DFA states (e.g., Figure 7 shows group types of 1, 2 and 3). As noted earlier in this section, a state is part of one or more groups based on the fanout count of its fanin states. In the figure, three distinct state groups are identified leading to a partitioning of memory rows into three groups.
- **Offset in state group** - The number of bits needed for this field is determined by the state group with the maximum number of sets of states. In Figure 7, groups 1 and 2 contains two sets and group 3 contains one set. Since the maximum number of sets in a state group is 2, only 1 bit is needed for the encoding in the example.
- **Number of next states** - This field must accommodate a binary value which indicates the largest number of next states for any state in the DFA. For example, the largest fanout from a state in Figure 7 is three (from state e to states f, g, and h), a two-bit encoding is needed for the field. For an n-bit hash value, a maximum number of n bits is needed to encode this field.
- **Valid hash values on outgoing edges** - This field provides a one-hot encoding of hashes of all possible next states. For a hash value of n bits, a total of 2^n bits are needed to encode the field. In the example, since a 4-bit hash is used, the field is 16 bits in width.

It should be noted that both the number of rows in the monitoring memory and the bit width of the offset in the state group are application dependent. Since a monitor will likely accommodate a number of different applications, the size of these parameters should be defined as part of the architecture specification for the monitor.

4.4 Similarity to Previous NFA-based Monitoring Systems

Several previous research projects have explored approaches that have similarities to aspects of our work. The idea of using the results of static analysis of an application binary to generate monitoring information has been previously proposed [14]. Like our approach, an NFA of processor execution sequences is created, although system calls, rather than individual instructions, are tracked. If a system call is identified that does not occur at an appropriate point in the NFA, an attack is assumed. Our new system differs in a number of important ways. First, the hardware-level details of the monitoring system in [14] are not presented, only the NFA generation approach is described. Second, the authors explicitly state that although it is possible, they have not explored converting the NFA to a DFA. In our system, due to potential collisions of hash values in the hardware system implementation, a conversion to a DFA is a necessity. Finally, our approach can quickly predict attacks within one or a small number of instructions, rather than at the granularity of system calls, and it is optimized for packet dropping in network processors, rather than multipurpose embedded processing.

A monitoring system for network processors with some similarities to our approach was previously presented [1]. This system uses a static analysis of program control flow to build a non-deterministic finite automata (NFA) of *basic block* execution sequences. If basic blocks are executed out of sequence, a stack smashing attack is detected and the offending packet is dropped. One of the limitations of this approach is attack detection speed. Basic block information for each instruction is stored in a monitoring memory. If a control flow instruction (e.g. a jump) occurs, additional monitoring memory lookups are required to verify that the new basic block for the code after the jump is one that is expected from the static analysis. If t jump targets are possible for the control flow instruction, the monitoring memory must be accessed t times for the instruction to check for t distinct target basic blocks. This action can lead to a real-time monitoring slowdown.

5 Harvard Architecture Attacks

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

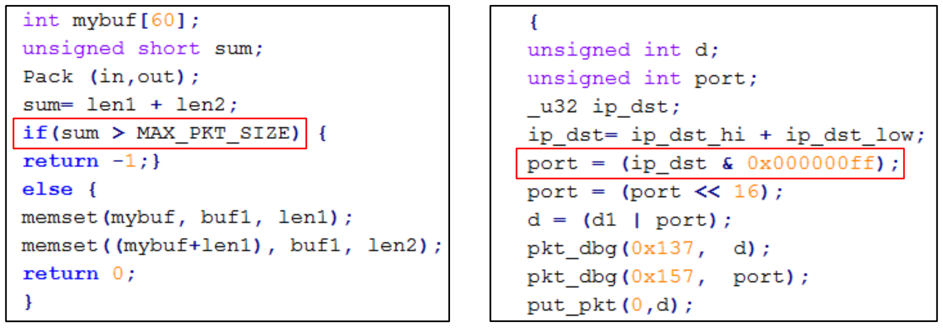
Even though general memory error techniques (integer overflow, heap overflow etc.) cannot be used to generate code injection attacks, Francillion et al. [17] demonstrated that code injection attacks are still feasible on a Harvard architecture processor using a return-oriented programming technique. 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 9 shows portions of congestion management protocol (CM) and an 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 the *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 there are no calls from the middle of the CM to the middle of the IPv4 application, no valid edges between states in the middle of the application are present. If such a transition is attempted, the 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. All packets in our system, including attack packets, are created using the NetFPGA packet generator tool [18]. All attack packets include 42 bytes for header with the remaining bytes used for data payload. All attack packets used for experimentation were the same.

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



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

CM protocol

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

IPV4 application

Fig. 9. Vulnerable application code.

TABLE 1
Resource utilization by our prototype implementation.

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

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.

6.1 FPGA-Based Prototype

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 circuitry for the router (e.g. buffers, input arbiter, queuing control, etc) are shown in Table 1. The monitor uses 140 LUTs and 26 FFs compared to 3,792 LUTs and 2,120 FFs for the network processor. In total, the prototype system including the network interface uses 23% of available LUTs, 22% of available FFs, and 20% of available memory bits in the FPGA.

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).

6.2 Monitoring Graph Generation

The automated offline analysis tool for security monitor generation is illustrated in Figure 10. To run networking

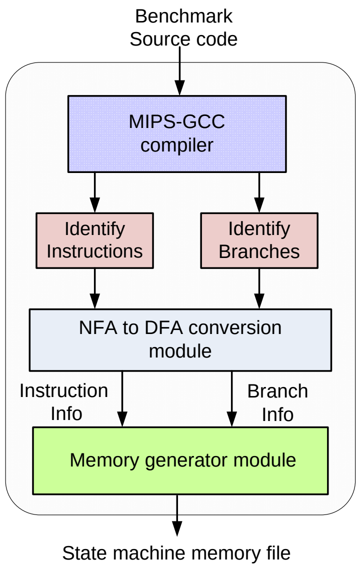


Fig. 10. Offline analysis to create state machine memory file.

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. The monitor can handle an arbitrary number of indirect branches to statically known targets (e.g., return addresses) since the NFA representation allows any number of outgoing branches. The NFA-to-DFA conversion starts with a non-deterministic NFA representation obtained from the compiler information using the approach outlined in Section 4.1. Through powerset construction, a DFA is constructed. This DFA is then converted into a monitoring state machine memory 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, nine benchmarks from the NpBench suite [19] were processed with this flow.

TABLE 2
Statistics for NpBench benchmark applications.

Network application	Cate-gory	No. of instr.	No. branch instr.	No. branches >2 targ.	Max branch targ.
crc	PPG	276	17	0	2
frag	PPG	573	70	3	3
red	TQG	802	88	0	2
md5	SMG	3,147	211	24	8
ssld	TQG	828	91	1	5
wfq	TQG	905	112	2	3
mtc	SMG	2,427	252	2	3
mpls-upstr.	TQG	1,603	322	9	10
mpls-dwnstr.	TQG	1,574	276	5	12

NpBench is a benchmark suite targeting modern network processor applications. The benchmark applications are categorized into three specific functional groups - the traffic management and quality of service group (TQG), the security and media processing group (SMG) and the packet processing group (PPG). A listing of the benchmarks and their application categories appears in Table 2. Since the presence of instruction branches has a direct impact on NFA-to-DFA conversion and monitoring state machine memory size, the number of control flow instructions for each benchmark is included in the table. Return instructions at the end of subroutines often contain numerous targets since a subroutine can be called from numerous other functions. The number of these *jump register* instructions with more than two possible return addresses is listed in the table. All applications, except *md5* with 24, have fewer than 10 *jump register* instructions. Additionally, the maximum number of target addresses for any branch in each application is also included. The *mpls-dwnstr.* application includes a control flow instruction with the largest number of branch targets, 12. Neither *crc* nor *red* contain a control flow instruction with more than 2 targets.

6.3 Network Setup

The simple test topology that was used to verify the performance of our monitoring system is shown in Figure 11. For hardware experiments, packets were generated and transmitted to the DE4 with the network processor and the 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. The packet generator tool allows for customizing the size and the throughput rate for the test packets.

7 Experimental Results

Our experimental results explore the size of the monitoring graphs generated by our system, the effect of different size hash functions, and the effectiveness of our system to detect actual attacks.

TABLE 3
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. appli-cation	No. of instr.	NFA states	Max. mem. access	DFA states	Mem. entries	Mem. over-head
crc	276	276	2	276	282	2.2%
frag	573	573	3	592	622	8.6%
red	802	802	2	805	847	5.6%
md5	3,147	3,147	8	3,173	3,228	2.6%
ssld	828	828	5	829	854	3.1%
wfq	905	905	2	914	953	5.3%
mtc	2,427	2,427	3	2,460	2572	6.0%
mpls-upstr.	1,603	1,603	10	1,621	1,753	9.4%
mpls-dwnstr.	1,574	1,574	12	1,582	1,706	8.4%

7.1 Monitoring Graphs

The results of generating instruction-level monitoring graphs for both our approach and a previous approach [1] described in Section 4.4 are illustrated in Table 3. The number of entries in the state machine memory (see Figure 8) for each benchmark is shown in the *Mem. entries* column. For these results, the *nibble-sum* hash function was used (see below).

A clear benefit of our new approach is speed. In all cases, only one access to the monitor memory is required for any benchmark since a DFA is used. The previous NFA-based approach requires up to twelve memory accesses for the benchmarks tested (and potentially up to sixteen for other benchmarks when all possible 4-bit hash values exist as outgoing edges). The conversion from an NFA to a DFA does incur a memory overhead of 5.7% on average for the benchmarks. Thus, our system requires slightly more memory space in order to guarantee a single memory access per instruction.

7.2 Evaluation of Hash Functions

As shown in Figure 8, each 32-bit instruction is converted to a hash value containing a small number of bits (e.g. $h = 4$). The hash function used to convert a 32-bit instruction to a 4-bit hash value involves the summing of all eight 4-bit instruction nibbles. The result of the summation is the 4-bit hash value. However, there are other possible hash functions, which we have explored in our experimentation.

In Section 4.2, it is noted that an important aspect of the NFA-to-DFA conversion is limiting the number of cases where the hash values for multiple edges leaving a state are the same (e.g. Figure 5). Limiting these cases avoids the creation of powerset state sets, and the corresponding increase in memory entries in the state machine. To decrease the probability of these hash collisions, it is desirable for the instruction hashes to be as evenly distributed across the range of possible

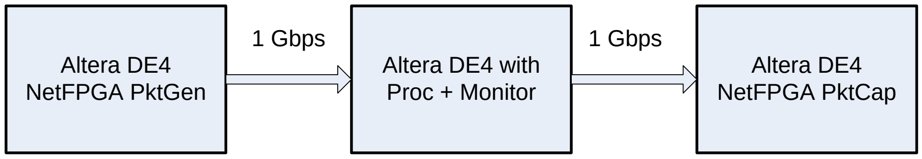

Altera DE4 NetFPGA PktGen
1 Gbps
Altera DE4 with Proc + Monitor
1 Gbps
Altera DE4 NetFPGA PktCap

Fig. 11. Network topology used for experimentation.

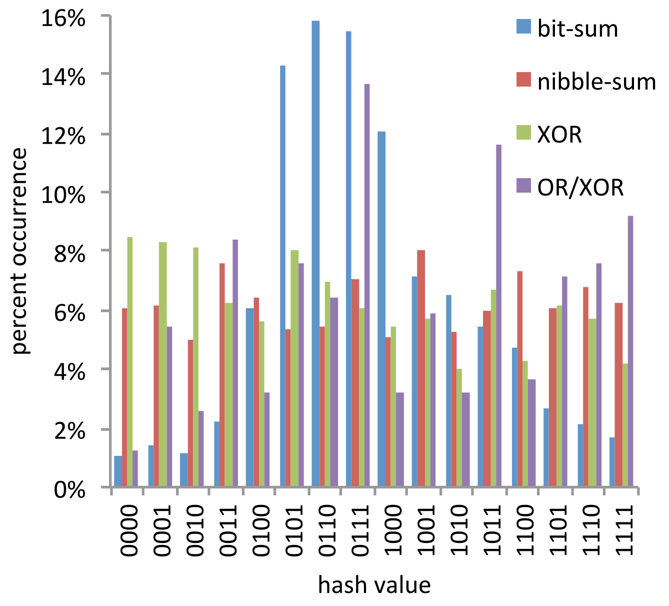


Fig. 12. Distribution of occurrences of different hash values generated the four explored hash functions for *mpls-downstream* benchmark.

instructions hashes as possible. Additionally, the 32-bit instruction to h-bit hash value conversion must be simple enough to be performed in one clock cycle.

Four hash functions were considered for this work:

- Sum of all ones in the 32-bit instruction (bit-sum): All binary digits are summed and the result is used to determine the h-bit hash value. For sums exceeding h-bits, only the bottom h bits are used as the hash value.
- Sum of all nibbles in the 32-bit instruction (nibble-sum): All 4-bit nibbles are summed and the result is used to determine the h-bit hash value. For sums exceeding h-bits, only the bottom h bits are used as the hash value.
- XOR of h-bit chunks in the 32-bit instruction (XOR): The 32-bit instruction is broken into h-bit chunks, which are then XORed together to generate an h-bit result.
- OR/XOR of h-bit chunks in the 32-bit instruction (OR/XOR): The 32-bit instruction is broken into h-bit chunks. Half the chunks are ORed together while the other half (including the final operation) are XORed.

The distributions of 4-bit hash values for all instructions for the *mpls-downstream* benchmark are shown in Figure 12. Plots for other benchmark applications are similar. Note that the results for hash value 0 in Figure 12 do not include the large number of NOP instructions (instruction 0x00000000) in branch delay slots following branch instructions. These instructions are not used as targets for branches and can be omitted from the analysis.

The *nibble-sum* approach to generating hash values is most effective in distributing hash values, approaching the ideal uniform distribution with $1/16 = 6.25\%$ probability of occurrence for any hash value. This result is likely due to the randomness caused by bit carries in generating the final hash values for *nibble-sum*.

The use of different hash functions directly impacts the required size of the monitoring state machine memory. Table 4 shows that the use of the *nibble-sum* hash approach reduces the number of required state memory entries versus other approaches by a range of between 0.02% and 1.35% on average. The remainder of the results presented in this section were generated using the *nibble-sum* hash function.

The number of bits used in the hash values also affects the amount of memory required in state machine memory. Although hash functions with more output bits decrease the possibility that an attacker can craft a useful code sequence, more bits also require more memory in the state machine. Each additional bit in the hash function effectively doubles the size of the "valid hash values on outgoing edges" field in the memory shown in Figure 8. Table 5 illustrates the memory overheads for different hash value bit widths using the *nibble-sum* hash approach. The results show a significant increase in memory size with every additional bit. A 4-bit hash value bit width requires a 40% memory increase over 3-bit, and a 5-bit hash value bit width requires a 100% increase over 3-bit.

7.3 Monitoring Effectiveness and Speed

We tested the ability of the monitor-based system to detect and recover from an attack. The vulnerable application code shown in Figure 9 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. A series of waveforms that demonstrate this behavior appear in Figure 13. As shown in Figure 14, after the monitor was added to the system, the attack packet was successfully identified, the NP was reset, and subsequent regular packets were routed

TABLE 4
Comparison of DFA states and state machine memory entries for different hash functions for a 4-bit hash

	nibble-sum		bit-sum			XOR			OR/XOR		
Network application	DFA states	Mem entries	DFA states	Mem entries	% Mem incr.	DFA states	Mem. entries	% Mem incr	DFA states	Mem. entries	% Mem incr.
crc	276	282	276	285	1.06	276	282	0.00	279	287	1.77
frag	592	622	592	627	0.80	592	620	-0.32	592	622	0.00
red	805	847	808	857	1.18	806	850	0.35	807	851	0.47
md5	3,173	3,228	3,208	3,277	1.52	3,181	3,248	0.62	3,190	3,261	1.02
ssld	829	854	836	878	2.81	831	860	0.70	836	875	2.46
wfq	914	953	921	977	2.52	916	955	0.21	918	960	0.73
mtc	2,460	2,572	2,460	2,584	0.47	2,459	2,567	-0.19	2,460	2,571	-0.04
mpls-upstream	1,621	1,753	1,625	1,758	0.29	1,622	1,744	-0.51	1,627	1,757	0.23
mpls-downstream	1,582	1,706	1,589	1,732	1.52	1,579	1,694	-0.70	1,584	1,712	0.35
average					1.35			0.02			0.78

TABLE 5
Comparison of DFA states, state machine memory entries, and memory bits for different hash sizes.

	3-bit		4-bit			5-bit		
Network application	Mem entries	Mem bits	Mem entries	Mem bits	Increase over 3-bit	Mem entries	Mem bits	Increase over 4-bit
crc	288	6,048	282	8,460	39.9%	282	13,254	56.7%
frag	620	13,020	622	18,660	43.3%	623	29,281	56.9%
red	853	17,913	847	25,410	41.9%	845	39,715	56.3%
md5	3,255	68,355	3,228	96,840	41.7%	3,227	151,669	56.6%
ssld	855	17,955	854	25,620	42.7%	855	40,185	56.9%
wfq	957	20,097	953	28,590	42.3%	951	44,697	56.3%
mtc	2,590	54,390	2,572	77,160	41.9%	2,567	120,649	56.4%
mpls-upstream	1,783	37,443	1,753	52,590	40.5%	1,738	81,686	55.3%
mpls-downstream	1,727	36,267	1,706	51,180	41.1%	1,695	79,665	55.7%

successfully. This behavior was verified using our DE4 hardware setup.

In a final experiment, we evaluated the performance of our network processor system. In particular, we consider scenarios with varying amounts of attack packets that are mixed in with regular packets. Both regular packets and attack packets were generated at fixed rates by a packet generator system. Packet sizes for both types of packets were 256 bytes. We consider two cases, data processing efficiency, in which the processing rate of all packets (both regular and attack) is considered and the throughput of regular packets.

Data processing rates of just below 50 Mbps for normal network traffic are achieved, as shown in Figure 15. The processor performs around 5,355 instructions for each 256-byte packet (which is typical for a payload processing application, such as ours where the packet payload is moved [20]). Based on a processor clock rate of 125 MHz (which corresponds to a cycle time of 8 ns) and one cycle per instruction, processing for one packet requires around $5,355 \cdot 8ns = 42.68\ ms$. Since with each packet $256\ bytes \cdot 8 = 2048\ bits$ of data are processed, the expected rate is $2048\ bits\ /\ 42.68\ ms\ =\ 47.9\ Mbps$. Thus, the peak rate corresponds with the expectation. Note that the data processing rate increases as the percentage of attack packets increases since attack packets can be identified and dropped within a few cycles (since the attack code is early in the packet processing code) versus the thousands required for regular packets.

The throughput of regular packets is reduced by the percentage of attack packets that is used. Figure 16 shows the throughput of regular packets for varying ratios of regular packets to attack packets. Not surprisingly, throughput is reduced by roughly the percentage of attack packets. For example, for a 900 Mbps input rate, the output throughput for no attack packets versus 50% attack packets is $47.5\ Mbps$ versus $25.8\ Mbps$. As seen in Figure 16, the throughput of a system without monitoring is the same as the throughput with monitoring if no attack packets are sent.

While this throughput performance may seem very low for a modern network system, it is important to note that our results are for a single core only. With tens to hundreds of cores, such as can be found in modern network processors, and with applications that only process packet headers, aggregate throughput of tens of Gigabits per second can be achieved.

8 Summary and Future Work

The security of the Internet infrastructure is of great importance to our society. To protect the Internet, it is critical to develop routers that are impervious to attacks. Since a new class of data plane network attacks has emerged targeting software-programmable network processors, we have developed a hardware monitoring system that can detect these attacks and protect network processors.

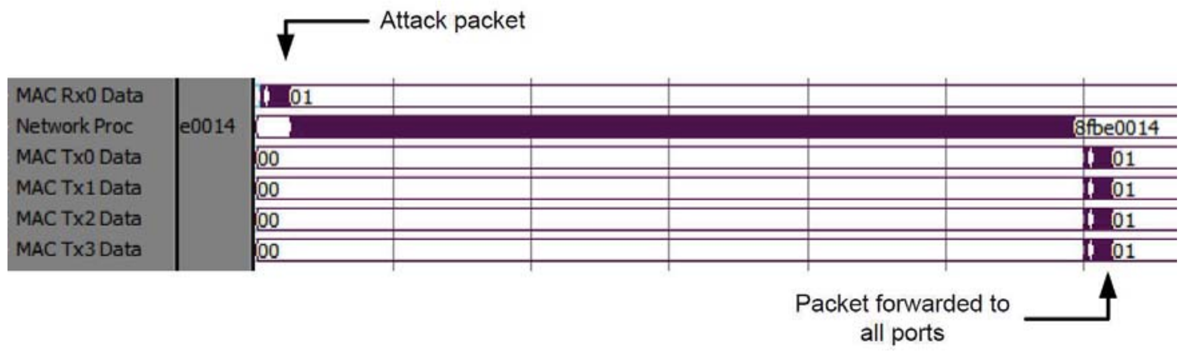


Fig. 13. Simulation waveforms showing an attack and subsequent forwarding of the packet to all output ports. This behavior was confirmed using hardware.

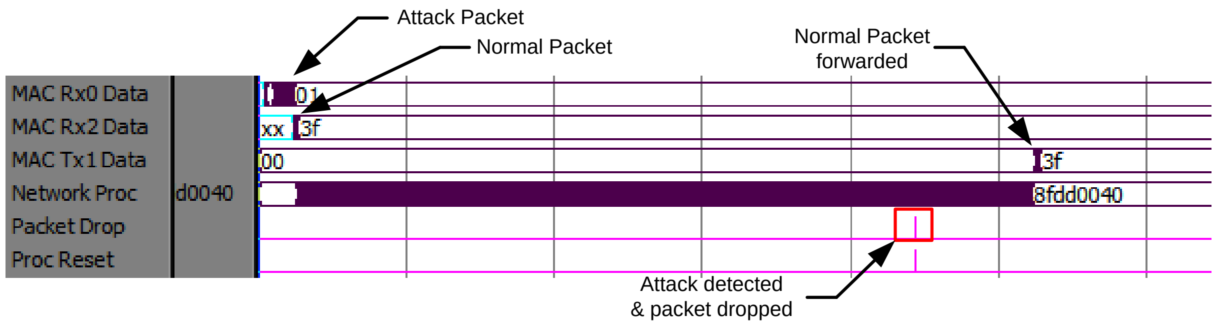


Fig. 14. Simulation waveforms showing the identification of an attack packet and the successful forwarding of the subsequent packet. This behavior was confirmed using hardware.

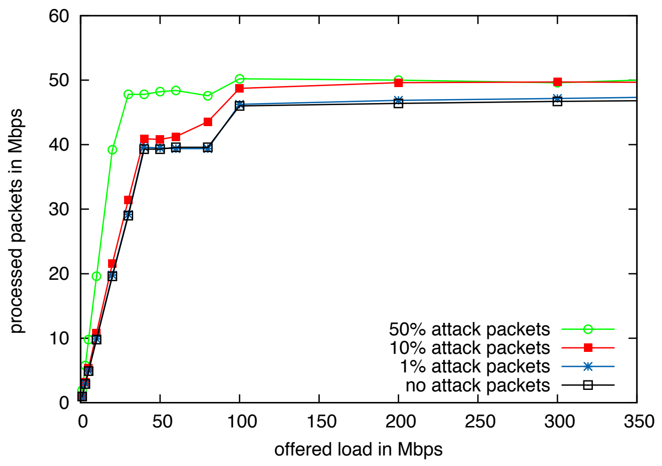


Fig. 15. NP core data processing of both regular and attack packets in terms of processed packets under varying loads of attack packets.

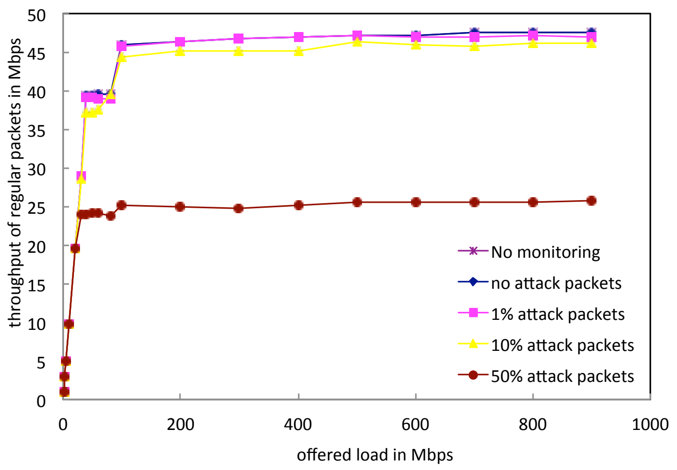


Fig. 16. NP core throughput in terms of processed regular packets under varying loads of attack packets.

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. Our results show that the use of DFA only increases memory size by 5.7%, compared to previous NFA approaches. Our prototype implementation of the monitoring systems shows our design is so efficient that even extremely large amounts of attack traffic do not lead to a degradation of throughput performance of the system.

We believe that this work presents an important step towards deploying effective and efficient hardware protection mechanisms for network processors in the Internet. Future work includes optimizing the monitoring memory architecture to consider the caching of frequently used monitoring graphs. Also, the possibility of crafting attacks which include instructions with the same sequence of hash values as legitimate code could be evaluated. Finally, the use of pre-deployment simulation to determine dynamic branch targets for monitoring could be considered.

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] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001, pp. 156–168.
[15] H. Chandrikakutty, D. Unnikrishnan, R. Tessier, and T. Wolf, "High-performance hardware monitors to protect network processors from data plane attacks," in *Proc. of the 2013 IEEE/ACM Design Automation Conference*, Austin, TX, Jun. 2013.
[16] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
[17] 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.
[18] G. Covington, G. Gibb, J. W. Lockwood, and N. McKeown, "A packet geneator on the NetFPGA platform," in *Proc. of the IEEE Symposium on Field-Programmable Gate Arrays*, Napa, CA, Apr. 2009, pp. 235–238.
[19] 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.
[20] R. Ramaswamy, N. Weng, and T. Wolf, "Analysis of network processing workloads," *Journal of Systems Architecture*, vol. 55, no. 10, pp. 421–433, Oct. 2009.

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.

He is engaged in research and teaching in the areas of computer networks, computer architecture, and embedded systems. His research interests include Internet architecture, network routers, and embedded system security.

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.

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. degree 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.

Deepak Unnikrishnan received his Ph.D. degree in Electrical and Computer Engineering from the University of Massachusetts, Amherst in 2013. He is a senior design engineer at Altera Corporation, San Jose, CA, where he is involved with the design and modeling of embedded high-speed transceiver blocks in next generation FPGA devices. His research interests include FPGA systems, parallel computing and virtual networking.

Russell Tessier (M00-SM07) 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 a professor of electrical and computer engineering with the University of Massachusetts, Amherst, MA. His current research interests include computer architecture, FPGAs, and system verification