Distributed Intrusion Detection with Intelligent Network Interfaces for Future Networks

Yan Luo Ke Xiang Jie Fan
Department of Electrical and Computer Engineering
University of Massachusetts Lowell
yan_luo@uml.edu, {ke_xiang,jie_fan}@student.uml.edu

Chunhui Zhang
Department of Computer Science
University of Massachusetts Lowell
czhang@cs.uml.edu

***Abstract*—Intrusion detection remains an important and challenging task in current and next generation networks (NGN). Emerging technologies such as multi-core processors and virtualization have changed the architecture of the building elements of NGN significantly, thus call for rethinking of how network processing is done. In this paper, we propose distributed intrusion detection using intelligent network interfaces where additional processing capabilities are available. We design and implement a prototype to perform pattern matching using network processors since pattern matching is one of the important workloads in intrusion detection. Through the experimental results, we show the feasibility and performance of distributed intrusion detection in next generation networks.**

I. Introduction

The Internet has been facing constant challenges from attacks, security breaches and abusive usages, thus calls for continuous effort to identify and block malicious traffic. Many network security applications such as intrusion detection systems [11] rely on pattern matching to identify attacks, and this approach has demonstrated high accuracy. Therefore, high speed pattern matching is regarded as an important technique leveraged by network security applications and remains a hot research problem in designing future networks.

The architecture of future Internet is expected to break the ossification of existing network architecture by embracing revolutionary changes. The packet processing capability is enhanced and distributed across the whole network to support virtualized networks, diversified services and ever-emerging new applications. This is made possible by the advancement of silicon technologies including multi-core processors and high speed interconnections. As a result, the processing power of an end host today is comparable to that of an edge router several years ago.

These updates in the building components of the Internet infrastructure shift the location where network processing is performed. One can now distribute complex packet processing workloads to processing elements along a packet's traveling path. End hosts with improved computing power can offload an increasingly large collection of processing tasks that are conventionally carried out by edge or core routers.

In this paper, we attempt to achieve *distributed intrusion detection* at end systems as opposed to centralized designs. We focus on high speed pattern matching within a hybrid multi-core based end host that consists of a host CPU and network interfaces with additional processing elements. We study regular expression pattern matching with network processors (NPs) as NPs have the merit of programmability which facilitates the porting of existing applications and development of new services. Our research in this paper shows the performance potential of network processors on pattern matching for the purpose of distributed intrusion detection at critical servers and end hosts.

The contribution of this paper includes:

- A proposal of distributed intrusion detection scheme on intelligent network interfaces. We propose to
- The design of a pattern matching engine with a network processor based network interface.
- Prototyping and performance evaluation of the distributed intrusion detection framework.

The rest of the paper is structured as follows. Section II presents the background of regular expression pattern matching and motivates our research. Section III describes the proposed distributed intrusion detection. Section IV gives the statistical analysis of intrusion detection signature patterns and optimization techniques. Section V describes our pattern matching engine built with Intel IXP NPs on network interfaces. Performance evaluation results are reported in Section VI. Section VII discusses related work, and finally the paper is concluded in Section VIII.

II. Background and Motivation

A network processor, such as Intel IXP [7] , usually contains multiple processing elements (PEs), co-processors, on-chip memory controllers, and high-speed network I/O interfaces. An NP connects to memory units of different size and speed. Fast SRAM memory stores control data structures while large DRAM memory is used to buffer packets after they arrive at the network interface cards (NICs). PEs execute instructions to process packet headers and payloads. After processing, the packets are transmitted to the next hop through the network interfaces.

Modern server architectures employ multi-core based processors for high throughput network processing. Moreover, network processors and FPGAs are incorporated into the NICs as new types of computing resources [10], [4], and we call them *intelligent NICs*. For instance, a commercially available multi-core server architecture, as shown in Fig. 1, has an NP based NIC, in contrast to a regular NIC. Network packets can

978-1-4244-3435-0/09/$25.00 ©2009 IEEE

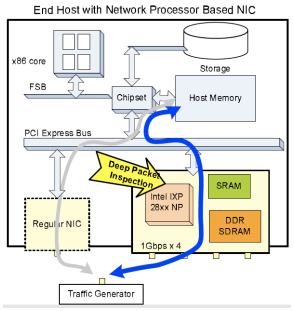

Fig. 1. End Host with Network Processor based NIC

now travel through the NP based NIC, instead of the regular one. This brings opportunities for additional deep packet inspection on the NP to offload the host CPUs. Such a system architecture has been leveraged in building programmable edge nodes for future networks with virtualization capabilities [9].

III. Distributed Intrusion Detection at NICs

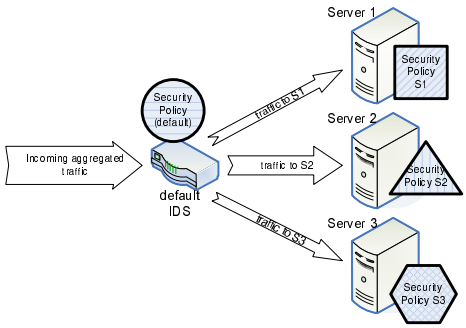

Fig. 2. The Architecture of a Distributed Intrusion Detection System

We propose to perform distributed intrusion detection at intelligent NICs that are equipped with packet processing engines such as network processors or FPGAs. Fig. 2 shows the architecture, which consists of a default IDS appliance and local intrusion detection devices at each servers to be secured. The incoming aggregated traffic is splitted and directed to the corresponding servers after passing through the default IDS. Different security policies, illustrated as different shapes, are applied at different locations. Compared to a centralized IDS, this distributed approach has several advantages:

- Capability of traffic checking at a finer granularity. Real-time intrusion detection at line speed is difficult, particularly when the checking is done on payloads in addition to the classic five-tuple. In this distributed method, the default IDS checks all incoming traffic against only the default policies, while an end system checks the traffic it receives against only its private security policies. With such a divide-conquer method, the intelligent NICs can offload the workload of a centralized IDS and inspect traffic at a finer granularity.
- Flexibility of applying customized security policies. The servers to be secured often have unique applications or traffic patterns, thus it makes sense to derive a set of dedicated security policies specific to that host. The owner of the servers can hand over these customized policies to the intelligent NICs and apply them on incoming traffic. In contrast, with a centralized IDS, all the incoming traffic are checked against the newly added policies, which is not necessary.
- Collection of attack forensics. This approach takes advantage of the processing capabilities in hybrid multi-core architecture with host CPUs and network processors. The intelligent NICs can retain valuable monitoring information, which can be retrieved and played back at a later time even when host OS is crashed by attacks.

There are many questions to be answered when designing and deploying such a distributed IDS. First, how to determine the default security policies and server-specific security policies is not clear. An IDS such as Snort [11] has thousands of security rules, which need to be divided to default rules and server specific ones. This is not a trivial task as rule sets keep expanding and server configurations vary from place to place. Second, investigation is needed on the performance impact on the servers, which are designed for providing particular services rather than IDS. The additional processing power of NPs and FPGAs can be leveraged, however, the workload partition is a difficult task and worth studying. NP and FPGAs provides high performance for network applications, while general purpose multicore processors has better programmability and flexibility, thus the system designers have to consider design trade-offs. Third, pattern matching, an important workload of IDS, is carried out on resource constraint intelligent NICs with limited memory space. It is necessary to optimize the design towards such environment to meet the performance requirements.

In this paper, we attempt to answer the third question by designing and optimizing a pattern matching engine on a network processor based NIC, which inspects incoming traffic against server specific security policies. We also investigate the performance impact on the servers when performing intrusion detection locally. We expect to prove the feasibility of the proposed distributed intrusion detection scheme through prototyping and experimental results before addressing the first and second questions in our future work.

IV. Pattern Matching Using Network Processor

Network security applications such as Snort [11] specify patterns (also interchangeably called rules) in either exact strings or regular expressions and search for them in packet payloads. For instance, in a Snort rule, “content:” begins the definition of an exact character string, while “pcre:” leads the Perl Compatible Regular Expressions (PCREs) that are more flexible in representing groups of similar patterns. Two kinds of automata, Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA), can be used to match against PCRE patterns. The space complexity of NFA is $O(n)$ and its searching complexity is $O(n^2)$ as there can be more than one active states during the matching process (back tracking is possible.) This leads to non-deterministic matching speeds that are not suitable for network applications.

In contrast, DFAs guarantee the matching process bounded by one active state per an input character, i.e., DFA's searching complexity is $O(1)$. However, the construction of DFA has a space complexity of $O(2^n)$. A pattern matching engine frequently accesses the finite automata stored in memory to find out a possible match in packet payloads.

DFA based pattern matching accesses memory to check match and look up the next state for an input byte. The faster the memory accesses, the better the matching performance. The size of the fast SRAM of a modern NP based system is limited to tens of megabytes and the on-chip memory is in hundreds of kilobytes. However, data have shown that the size of DFAs can easily reach over several hundred megabytes due to the exponential number of states and transitions generated from complex PCREs [15]. As a result, a large part of the DFA has to be kept in off-chip slow memory such as DRAM, thus leading to low matching speed. Therefore, our focus is to study the characteristics of DFAs and try to reduce DFA footprint so that they can be stored in faster but small SRAM to achieve regular expression matching at gigabits per second.

We create a regular expression compiler that takes Snort PCRE patterns as input and generates corresponding DFA for every PCRE pattern. As a result, each Snort rule file, containing k PCRE patterns, will have k DFAs with different sizes. We collect from the DFAs the statistical data including the number of states per rule file, the number of transitions per rule file, and the average number of transitions per state. We report only the number of transitions per state due to space limit.

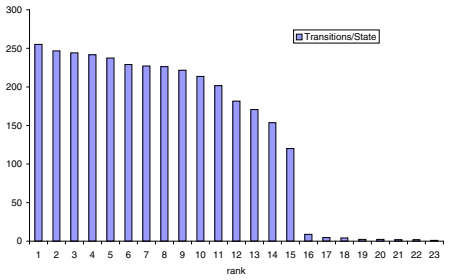

Fig. 3. Distribution of the average number of DFA transitions per state, ranked per rule file

We plot the average number of transitions per state (y axis) in Fig. 3, where x axis is the rule files ranked by the number of states. An interesting observation is that the DFA states tend to have either very dense (over 200) or very sparse (less than 8) transitions: over 82% of the DFAs have more than 200 transitions per state, 94% of the DFAs have more than 128 transitions per state, and 5% of the DFAs have up to 8 transitions per state. The dense transitions are due to meta characters such as "." and character sets such as "[^\n]". The sparse transitions come from simple PCRE patterns where the next transitions are limited. Such transition density distribution is observed from Snort PCRE rule sets although it is not clear whether this applies to other realistic rule sets. For Snort rule set, the disparity of transition density dictates how DFA states and transitions are stored in memory.

V. The Design

In this section, we describe the design of an intrusion detection engine built upon an Intel IXP NP. A DFA state contains memory pointers to possible next transitions, a flag indicating matched or not, and the identifier of the PCRE, if it is an "accept" state. The flag and identifier can be encoded with a fixed length bit vector, however, the number of memory pointers varies from state to state in a DFA. With the proposed encoding methods, the DFA generated from Snort rule set is in the order of tens of megabytes. Thus, we store DFA in SRAM since the on-chip memory (e.g. scratchpad of Intel IXP) is usually in the order of megabits.

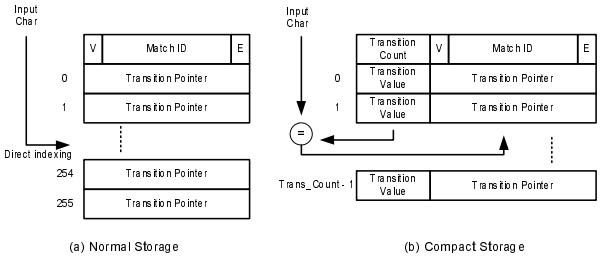

Fig. 4. Normal and Compact storage of DFA states and transitions.

We use two memory organizations, *default* and *compact*, to store a DFA state with its next transition pointers, as shown in Fig. 4. The encoding flag bit ("E") indicates the storage mode: 0 for default mode and 1 for compact mode. In (a) the default mode, a state contains 256 words for memory pointers, no matter how many of them are valid for a particular state. Some amount of memory space is wasted, however, the benefit is that the state machine can determine its next transition by using the input character as a direct index to the 256 pointers. This indexing can be achieved through simple address calculation. This storage mode is used for the DFAs whose transitions per state are very dense (over 128 transitions.)

```
if Encode == 1 then /* compact storage mode */
    extract Trans_count;
    read SRAM for Trans_count words;
    for i from 1 to Trans_count do
        extract trans_value ;
        compare input char with trans_value of i-th transition;
        if match the input char then
            extract Transition Pointer;
            break;
        end
    end
end
```

Algorithm 1: Look Up a Transition in a Compact DFA

In contrast to the default storage method, DFA states with sparse transitions (up to eight transitions per state) are stored in a compact mode as shown in Fig. 4(b). The reason we choose eight transitions is that the fast off-chip memory SRAM can be accessed in units of eight words per memory access using Intel IXP NPs. In this compact mode, the lookup of next transitions is done with Algorithm 1. Once eight DFA transition words are fetched, the NP executes a loop to compare the transition value encoded in a transition word with the input character (byte) from packet payload. This is the trade-off of compact storage

compared to the default storage mode. Due to the optimized instruction set of the NP, such program execution is more efficient than a DRAM access if the DFAs are too large to be kept in SRAM.

The multiple processing elements (a.k.a Microengines or MEs) of an Intel IXP NP can be programmed independently and organized in a hybrid pipeline. In the pipeline, there are mainly five stages from packet being received to finally transmitted out, namely receiving, processing, queue management, scheduling and transmitting. The packets are received from the network interface unit and stored in DRAM buffers. The meta data (e.g. size, DRAM address, receiving port number, etc.) of a packet are stored in SRAM and identified with a "buffer handle". The buffer handle is passed on to the next PE through a scratchpad ring, which is a hardware-assisted atomic message passing scheme in IXP NPs.

In the design, the Intel IXP2855 NPs have 16 MEs on a chip, among which up to 12 MEs are allocated for the pattern matching, and the rest four MEs are for stage 1, 3, 4, and 5. The SRAM module stores DFAs in either default or compact mode, while the DRAM module buffers the packets received. There are four independent channels in the SRAM controller. We use channel 2 of SRAM controller to access packet meta data and packet queue descriptors, and channel 3 for DFA states and transitions. This allocation ensures the accesses to SRAMs are balanced to avoid bottleneck.

VI. Performance Evaluation

A. Experiment Methodology

We implement the proposed intrusion detection design on an Intel IXP 2855 NP based NIC - Netronome's NFE-i8000 [10]. The NFE board is equipped with four 1Gbps Ethernet interfaces, 24MB SRAM and 768MB DRAM. The MEs of the IXP2855 NP are programmed to receive packets from two of the Ethernet ports, perform pattern matching, and forward the packets out through the other two egress ports. The packets are generated with two Linux boxes at their maximal speed, resulting in an aggregate packet flow up to 2Gbps with maximal packets of 1518 bytes.

The performance metric under study is the throughput of the system in units of megabits per second. To measure the throughput at high accuracy, we implement instrumentation code along with the pattern matching code, using on-chip counters and timestamps available in Intel IXP NP.

B. Experiment Data and Analysis

1) Pattern Matching vs. Plain Forwarding: We present the experiment results on the performance of regular expression matching in Fig. 5. The y axis is the measured throughput and the x axis is the number of microengines ranging from 1 to 6. The figure shows four configurations of the system. The baseline configuration is plain packet forwarding using the NP without searching for patterns in payloads. The top curve in the figure shows that the system supports 1.9Gbps consistently. The second top curve represents the throughput of the system which reads raw packet payload but without searching for patterns. In this configuration, the throughput scales as the number of MEs increases: six MEs can support up to 1.3Gbps.

The other two configurations perform pattern matching: one with reading one SRAM word (DFA state) per input character, and the other with reading 8 SRAM words. These two configurations correspond to the default DFA storage and compact storage, respectively. When NP reads default DFA transitions, the throughput is about 5% lower than consuming only raw payload bytes. But note that the size of the DFA in default storage mode can soon exceeds the available size of SRAM modules and cause part of the DFAs to be stored in cheaper and larger DRAM modules. The bottom curve represents pattern matching on compact DFAs. Reading eight SRAM words adds noticeable overhead, leading to the lower throughput than the first three configurations. However, storing DFA in SRAM still significantly outperforms the case of all DFA states being stored in DRAM, which has very poor performance (around 64Mbps, not shown in the figure) due to longer access latency and memory contention on DRAM. Therefore, our proposed compact storage method effectively alleviate the memory bottleneck problem and improve the pattern matching performance significantly.

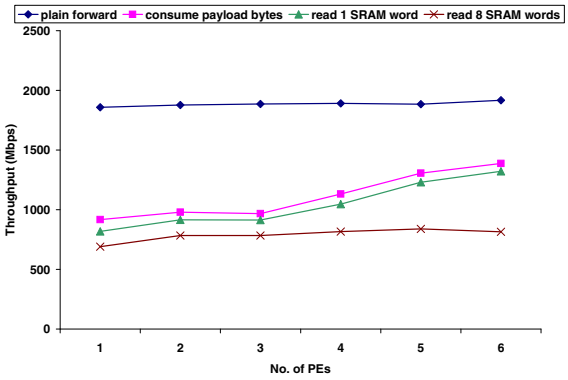

Fig. 5. Performance Comparison of Pattern Matching with Different Design Options

2) Scalability of Multi-cores on a NP: We conduct another experiment to study the scalability of multi-cores on an Intel IXP2855 NP. Fig. 6 shows that the performance increases as more PEs (microengines) are used to process packets and peaks at seven PEs. After that, more PEs do not bring improvement on the performance. We use the cycle accurate simulator within Intel IXA Software Development Kit to evaluate the impact of memory bandwidth through simulations. The SDK allows us to configure three different DRAM clocks that give different memory bandwidth. The results are shown in Fig. 7. The system throughput increases consistently with DRAM memory bandwidth. This suggests that the DRAM unit is the bottleneck and the system performance can improve with faster DRAM units.

C. Discussion

we learned several lessons from designing the prototype of a distributed IDS: (1) Distributed IDS is feasible with the availability of additional processing capabilities on NICs. (2)

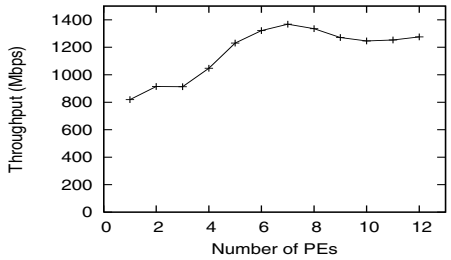

Fig. 6. Scalability of Multi-cores.

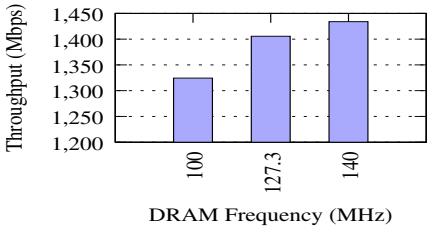

Fig. 7. Impact of DRAM bandwidth.

The size of the DFA and memory organization can largely affect the system performance. the DFA size so that it can be fit into on-chip memory for higher pattern matching performance; (3) The memory throughput is a determining factor of high performance network applications especially future payload processing applications; (4) it is important to exploit the architectural features of processing elements.

VII. Related Work

Both in [12] and [9], network processors are employed as major packet processing elements in providing virtualized network and accelerating complex packet processing. NetFPGA [5] is another programmable network processing engine that is designed for future network processing.

The most relevant existing work to our distributed IDS is the NIDS cluster presented by Vallentin et al. in [13]. In that work, intrusion detection is carried out with a cluster of PCs where frontend PC distributed network traffic to backend PCs and backend PCs run NIDS instances to perform the traffic analysis. Our work differs with [13] in location of IDS and hardware architecture. Our local IDS devices (the intelligent NICs) are tightly coupled with the severs as opposed to a centralized NIDS cluster. We employ NPs that are optimized for network processing as opposed to the general purpose processor based PCs.

Many signature-based systems have been architected for the FPGA [14], [6], [8] and ASIC [3], [1], taking advantage of the parallel structures available in these devices. Our work differs from these works in several aspects: (a) we utilize the programmability of NP and memory based DFA to enable flexible updates of security policies. FPGA and NFA based approaches such as [14], [8], [2] cannot accommodate new policies without reprogramming the FPGA; (b) our distributed IDS handles traffic inspection locally, where the line rate is generally lower than that of centralized IDS appliances. Thus, with moderate processing power at NICs, the distributed IDS can achieve satisfactory results. It is an open question that when line rate at end systems reaches 10Gbps and beyond, how this distributed IDS scales. We believe that, the continuous advance of processor technologies can scale with increasing the line rate.

VIII. Conclusion

In this paper, we propose a distributed IDS method that utilizes the additional processing power available at NICs. We design a regular expression pattern matching engine design with an Intel IXP network processors for detecting intrusions at end systems. We implement the prototype, conduct experiments to evaluate its performance, and discuss the lessons learnt from the design.

Acknowledgment

The authors thank Justin Latham for his earlier work on this topic. The work is supported in part by National Science Foundation under grant CNS0709001, a subcontract from BBN Technologies, and a grant from Intel Research Council.

References

[1] N. Sertac Artan, Rajdip Ghosh, Yanchuan Guo, and H. Jonathan Chao. A 10-Gbps High-Speed Single-Chip Network Intrusion Detection and Prevention System. In *IEEE Globecom 2007*, November 2007.

[2] Joao Bispo, Ioannis Sourdis, Joao M.P. Cardoso, and Stamatis Vassiliadis. Synthesis of regular expressions targeting fpgas: Current status and open issues. In *Int. Workshop on Applied Reconfigurable Computing (ARC 2007)*, pages 179–190, Mangaratiba, Brazil, March 2007.

[3] B.C. Brodie, R.K. Cytron, and D.E. Taylor. A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching. In *ISCA*, Boston, MA, June 2006.

[4] Glen Gibb, John W. Lockwood, Jad Naous, Paul Hartke, and Nick McKeown. NetFPGA: An Open Platform for Teaching How to Build Gigabit-rate Network Switches and Routers. *IEEE Transactions on Education*, 2008.

[5] Glen Gibb, John W. Lockwood, Jad Naous, Paul Hartke, and Nick McKeown. Netfpga: An open platform for teaching how to build gigabit-rate network switches and routers. *IEEE Transactions on Education*, 2008.

[6] C. Hayes and Y. Luo. Dpico: A high speed deep packet inspection engine using compact finite automata. In *ACM Symposium on Architecture for Network and Communication Systems*, Orlando, FL, December 2007.

[7] Intel Corp. Intel IXP2855 Network Processor Product Brief, 2005.

[8] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of regular expression pattern matching circuits on fpga. In *DATE 2006*, Munich, Germany, 2006.

[9] Y. Luo and C. Zhang. The design of a programmable network edge node with hybrid multi-core processors for virtual networks. In *IEEE International Conference on Computer Communications and Networks*, St Thomas, USVI, August 2008.

[10] Netronome. Product Brief - NFE-i8000 Network Acceleration Card, 2006. http://www.netronome.com/.

[11] Snort. http://www.snort.org/, 2003.

[12] J. Turner, P. Crowley, and et al. Supercharging planetlab – a high performance, multi-application, overlay network platform. In *SIGCOMM*, 2007.

[13] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Proc. RAID*, 2007.

[14] N. Weaver, V. Paxson, and J. M. Gonzalez. The shunt: An fpga-based accelerator for network intrusion prevention. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, Monterey, CA, 2007.

[15] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ACM Symposium on Architecture for Network and Communication Systems*, San Jose, CA, December 2006.