A Hardware Platform for Network Intrusion Detection and Prevention

Chris Clark[1], Wenke Lee[2], David Schimmel[1], Didier Contis[1], Mohamed Koné[2], Ashley Thomas[2]
Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology, Atlanta, GA, USA

[1]*School of Electrical and Computer Engineering*
{cclark, schimmel, didier}@ece.gatech.edu

[2]*College of Computing*
{wenke, lechefs, athomas}@cc.gatech.edu

Abstract

The current generation of centralized network intrusion detection systems (NIDS) have various limitations on their performance and effectiveness. In this paper, we argue that intrusion detection analysis should be distributed to network node IDS (NNIDS) running in hardware on the end hosts. An NNIDS can unambiguously inspect traffic to and from the host, and when implemented on the network interface hardware, can function independently of the host operating system to provide better protection with less overhead than software implementations. We discuss the computation and communication characteristics of typical software intrusion detection analysis tasks. Then, we describe our efforts in mapping these tasks to a hardware platform using COTS components including Intel IXP network processors and Xilinx Virtex FPGAs. We report the performance of our prototype NNIDS implementation and provide analysis on how the network processor architecture affects the performance. Our results show that the NNIDS can achieve high performance with a pipeline of processing stages and careful allocation of tasks to the most appropriate hardware resources.

1. Introduction

The current generation of network intrusion detection systems (NIDS) have several limitations on their performance and effectiveness. Many of these limits arise from some inherent problems with the traditional placement of the NIDS sensors within the network infrastructure. Sensors are typically positioned at the aggregation points between the internal and external networks and monitor traffic for a large number of internal hosts. However, there may be other external entry points that go unmonitored, such as dial-up and wide-area wireless (cellular) data connections at the end-hosts. Also, a sensor at the gateway typically does not monitor traffic between internal hosts, so it cannot detect internal attacks. The performance problems with this type of centralized NIDS placement include limited throughput and poor scalability. Recent studies [1, 2, 3] have shown that modern NIDS have difficulty dealing with high-speed network traffic. Others [4, 5] have shown how attackers can use this fact to hide their exploits by overloading an NIDS with extraneous information while executing an attack. Furthermore, centralized NIDS do not scale well as network speed and the number of attacks increases. Since network traffic is increasing faster than computer performance [6] and new attacks appear almost daily, these problems will only get worse with time. Therefore, it is important to explore different architectures for deploying intrusion detection sensors.

We suggest that, in order for a network intrusion detection system to accurately detect attacks in a large, high-speed network environment, the bulk of analysis should be performed by distributed and collaborative network node IDS (NNIDS) running at the end hosts. Advantages of this approach over centralized analysis include: a large reduction in the quantity of data to be analyzed, the ability to analyze end-to-end encrypted traffic, the ability to adapt the analysis based on knowledge of the end system, and the capability to actively control the types and rates of traffic received and sent by a host. A NNIDS is in the unique position to prevent incoming attacks from reaching the host operating system or application. In addition, a NNIDS

can prevent outgoing attacks or quarantine an infected host to keep it from infecting other internal or external hosts. On the other hand, a distributed architecture increases the difficulty of managing the sensors and detecting distributed attacks. However, these issues have been addressed in related contexts [7, 8, 9, 10, 11, 12, 13].

Our research aims to develop NNIDS that can keep up (i.e. avoid packet drop) with the traffic rate that an end-host can accept. These NNIDS should be able to reliably generate timely and accurate alerts as intrusions occur and have the intrinsic ability to scale as network infrastructure and attack sophistication evolve. Research in algorithms for attack analysis and traffic profiling are important components of this goal. However, our current research focus is on another essential component: design and implementation of a hardware platform that enables high-speed, reliable, and scalable network intrusion detection.

We have built a prototype NNIDS, based on the open-source IDS, Snort [14], on a network interface utilizing an Intel IXP 1200 network processor [15] and a Xilinx Virtex-1000 FPGA [16] co-processor. Our tests show that the NNIDS can keep up with traffic up to 100 Mbps. We believe that the same design, when ported to the recently-released IXP 2x00 network processor series, will enable the NNIDS to keep up with traffic of at least 1 Gbps.

In the remaining sections of this paper, we will discuss the design of high-speed NNIDS, the implementation of a prototype system, and the results of experimental studies. In Section 2, we will examine the design rationales and principles. We will discuss the need for hardware-based NNIDS and some system architecture considerations. In Section 3, we will describe our hardware platform and its use in the development of a prototype NNIDS. We will explain how IDS components were mapped to and implemented on a pipeline of processing elements. In Section 4, we will present the results of our system evaluation. Finally, in Section 5, we will provide a summary and discuss future work.

2. Design rationales and principles

In this section, we discuss some considerations in the design and implementation of high-speed, reliable, and scalable network intrusion detection systems.

2.1. Motivation for hardware-based NNIDS

In addition to the problems mentioned in Section 1, centralized NIDS have other weaknesses. A common and serious issue is that they typically do not have sufficient knowledge of the network topology and which operating systems are running on the network hosts. As a consequence, the NIDS and a host might interpret the same network traffic differently. This vulnerability allows attackers to evade detection by sending attack traffic to a host that looks harmless from the perspective of the NIDS [4, 5]. In addition, NIDS generally do not have the necessary keys (or enough resources) to examine end-to-end encrypted traffic for every host. This means that data sent over protocols such as SSL or SSH can not be analyzed by a centralized NIDS, giving attackers another means to evade detection.

One remedy to these problems is to use network node IDS (NNIDS) that each monitor the traffic to a single host. An NNIDS can unambiguously analyze the network data and have access to the key(s) to examine encrypted data. Some NNIDSs have been implemented as kernel- or application-level software. However, the overhead of intrusion detection analysis can severely degrade the performance of other applications running on the host. Furthermore, if an attacker manages to compromise the host, she can also disable the NNIDS so that all of her malicious activities will go undetected. We believe that these shortcomings can be adequately addressed by implementing the NNIDS on the network interface rather than on top of the host operating system.

Network processors will be widely available and affordable in the near future and can be integrated into a network interface card (NIC) with a cost similar to other high-end NICs. Having an NNIDS run on a NIC with a network processor has several advantages over a software NNIDS. These include minimal performance impact on the host system and much stronger protection for both the host and the IDS itself. A hardware NNIDS runs independently of the host operating system and can be made "subversion-resistant" so that it continues to function even if the attached host is compromised. An attacker cannot disable the NNIDS even if he penetrates the host because the control flows to the network interface can be very restrictive. These facts make it desirable to install hardware NNIDS in critical systems or even all of the nodes on the network. This deployment scheme can scale to large and complex networks because each NNIDS runs on an affordable NIC and unambiguously checks only the traffic for its attached node.

There are other research issues with NNIDSs in addition to the placement of the agent. The security policy that dictates network intrusion detection functions must be managed and enforced in a distributed fashion. This problem is similar to managing distributed firewalls [11]. We can learn from the research in distributed firewalls to develop a (perhaps similar) solution to this problem. The NNIDS also need to perform event-sharing

and collaborative analysis techniques to detect distributed attacks and share the workload when necessary. This problem is not necessarily unique to NNIDS because an NIDS using load-balancing techniques needs to deal with the same issue [17, 18]. In other words, we can borrow ideas from other research to address the issues with NNIDS.

2.2. Characterization of NIDS components

Before we can design and implement an NIDS on a network processor, we must first analyze the performance characteristics of NIDS analysis. A real-time NIDS monitors network traffic by sniffing (capturing) network packets and analyzing the traffic data according to intrusion detection rules. Typically, an NIDS runs as application-level software. Network traffic data is captured using an operating system utility, stored in OS kernel buffers, and then copied to NIDS application buffers for processing and analysis.

We use Snort [14] as an example to describe the main stages of packet processing and analysis in NIDS. In the Snort software, each captured packet goes through the following steps:

1. Packet decoding: Decodes the header information at the different layers and stores the information in data structures. All packets go through this step.
2. Preprocessing: Calls each preprocessor function in order, if applicable. The preprocessors used by default include IP fragment reassembly and partial TCP stream reassembly.
3. Detection: First, the values in a packet's header are used to select an appropriate subset of rules for further inspection. This subset consists of all the rules that are applicable to that packet. Second, the selected rules are evaluated sequentially.
4. Decision: When there is a match with one of the detection rules, its corresponding action, alert, or logging function is carried out.

An NIDS can be considered a queuing system where the packet buffers are the queues and the NIDS is the service engine. Obviously, if the NIDS processes the packets slower than their arrival, the buffers can be filled up and the newly-arriving packets will be dropped (i.e. not stored). As a result, the NIDS may not have sufficient information to accurately analyze the traffic and detect intrusions. Therefore, it is very important to design and implement NIDS to minimize dropped packets.

In our benchmarking experiments where Snort runs as application-level software, the service time ratios of the above steps are roughly: 3 for decoding, 10 for preprocessing, and 30 for detection. Logging can be very slow because of network or disk I/O. We also observe packet drops when the traffic rate goes above 50 Mbps.

In preprocessing, the bulk of compute-time is spent on bookkeeping and thus requires frequent memory accesses. For example, fragments of the same IP packets, or TCP payloads of the same stream need to be stored in data structures and looked up. In detection, the bulk of compute-time is spent on testing the conditions of the detection rules one by one. A typical NIDS can have 1,500 or more detection rules, and each rule can have several conditions that require pattern (or keyword) matching or statistics computation. Another system factor that slows down NIDS is the inefficiency of the network data path. Packet data is captured at the network interface, passed to the kernel via PCI bus, filtered to eliminate unwanted packets, and the remaining packets are stored in kernel buffers.

2.3. Hardware architecture considerations

It is clear from the above discussion that there are potential performance gains if the NIDS components are implemented in a network processor where packet processing can take place close to the data source and can be carried out with a pipeline of processing engines. However, there are challenges to realize these performance gains.

Intrusion detection is an interesting application from a NP (network processor) hardware architecture perspective because of its substantial resource requirements. Intrusion detection analysis requires considerably more compute cycles and memory accesses per packet than required by traditional NP applications, such as IP routing and QoS scheduling. The analysis consists of several tasks with varying resource usage patterns; some tasks are compute-bound and some are memory-bound. Furthermore, the amount of work done for each packet is not constant.

When designing the NNIDS system architecture, we considered both the requirements of the various analysis tasks as well as the capabilities of each hardware component. Based on these properties and experimental testing, our goal was to determine the most efficient

		Complexity: Low	Complexity: High
Data Rate: High		Microengines	FPGA
Data Rate: Low		Micorengines or StrongARM	StrongARM

Figure 1. Task to hardware allocation

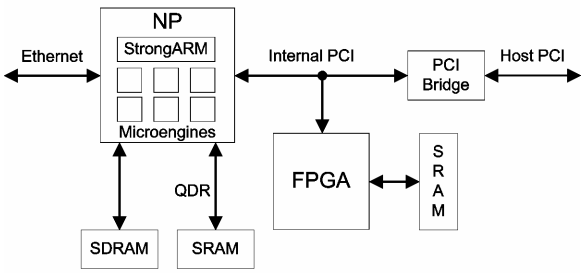

Figure 2. Prototype hardware platform

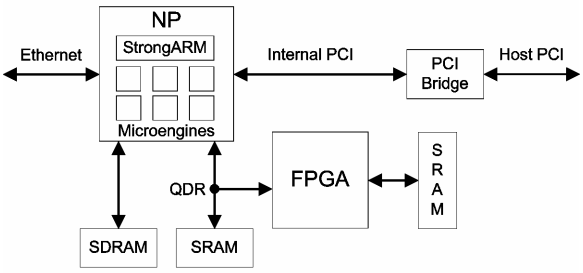

Figure 3. Proposed hardware platform

allocation of tasks to hardware resources. Some of these tasks fit well into existing NP architectures and some do not. Figure 1 summarizes our criteria for mapping tasks to hardware processing elements. On the IXP, processing requiring relatively few or simple operations to be applied to high-rate data can be implemented on the microengines. We put packet capturing and filtering, decoding, and preprocessing on the microengines. Each of these tasks naturally runs in a microengine thread. Computations that require complex calculations on lower-rate data are best carried out by the StrongARM processor. We run the IDS decision engine on the StrongARM. However, some IDS tasks require both complex computation and high throughput. This type of task is not feasible to implement on the network processor. For such cases, our approach is to map the computation onto dynamically reconfigurable hardware, which is able to achieve high performance by optimizing concurrency of the given computation. We use a field-programmable gate array (FPGA) co-processor to handle this type of task. In our system, the co-processor handles the keyword pattern-matching functions.

3. Prototype NNIDS on a network interface

In this section, we describe a programmable network interface and our implementation of a NNIDS on this platform.

3.1. Hardware platform

A block diagram of our hardware platform is shown in Figure 2. It uses the Radysis ENP-2505 development board [19] with four 100 Mbps Ethernet ports. The main components are an Intel IXP network processor and a Xilinx Virtex FPGA. The FPGA co-processor board is attached to a PCI mezzanine connector (PMC) and communicates with the NP via an internal 32-bit, 66 MHz PCI bus with a theoretical throughput of 2.1 Gbps. However, the overhead imposed by the PCI interface limits the type of tasks that can be off-loaded to the co-processor. The long latency of PCI transactions implies that large data transfers are more efficient than small transfers. Also, the FPGA must be able to obtain a large enough compute-time margin over the NP to overcome the cost of moving the computation across the PCI bus. One task that we have successfully off-loaded to the FPGA—packet payload searching—will be discussed in Section 3.4. We are also pursuing a more tightly-coupled NP-FPGA interface to improve performance and enable a broader class of tasks to be off-loaded to the co-processor. This would also allow the system to adapt to changing traffic conditions by dynamically reallocating tasks between the NP and FPGA. An ideal architecture would be to have the co-processor attached to the NP's SRAM memory bus and mapped into the NP's address space as shown in Figure 3. This makes the cost of accessing the FPGA comparable to the cost of memory reads and writes and enables a very fine-grained partitioning of tasks between the IXP and the FPGA. This is the same type of interface specified by the Network Processor Forum's Look Aside Interface LA-1.0 [20].

3.1.1. Intel IXP 1200. We use the Intel IXP 1200 network processor [15] in our implementation. It is a system-on-chip containing a StrongARM core and six programmable microengines and has a clock speed of 232 MHz. The StrongARM runs a version of Linux. Each microengine has hardware support for multi-threading, and can run a maximum of four threads. The StrongARM and all the microengines share 256 MB of 64-bit SDRAM and 8 MB of 32-bit SRAM in our configuration. The SDRAM has a peak bandwidth of 648 MBps and the SRAM has a peak bandwidth of 334 MBps.

3.1.2. FPGA co-processor. Field-Programmable Gate Arrays (FPGAs) have been used to accelerate many different algorithms, often achieving several orders-of-magnitude better performance than software implementations. This is made possible by their ability to be programmed with circuits customized to the given application and their capacity to perform massively-parallel computations.

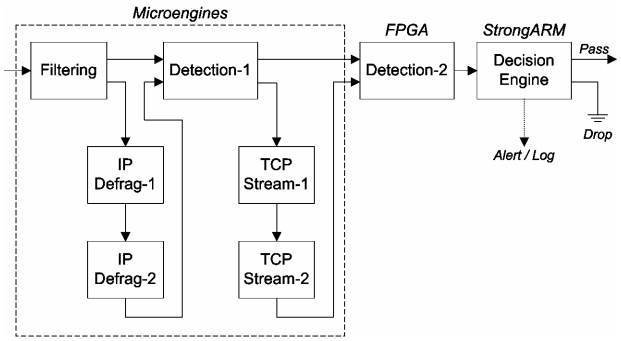

Figure 4. Detection pipeline

Our FPGA platform consists of a board containing a Xilinx Virtex-1000 FPGA [16], which is capable of implementing circuits with the equivalent of up to one million logic gates. The FPGA has a PCI interface for I/O as well as its own dedicated high-speed SRAM.

3.2. Snort implementation on IXP

We use Snort [14], a popular open-source NIDS software package, as the basis of our prototype NNIDS because it is loosely-coupled and easy to customize. Here, we briefly describe the main components of the Snort software. The packet capturing and filtering module is based on `libpcap` [21]. The packets are passed to the decoder to process the various packet headers. Each packet then passes through a series of preprocessors, including IP fragmentation reassembly and TCP stream reassembly. Then the packets are checked by the detection engine. Snort rules are organized to be matched in two phases. The first phase assigns each packet to a group based on the values of some header fields. The set of rules loaded at configuration-time determines the number of groups and the header values associated with each group. The second phase performs further analysis that depends on the assigned group, but usually includes a full search of the packet payload for a large number of patterns. Finally, the decision engine uses the results of the detection phase to take appropriate action.

Our task was to modify or restructure the Snort components to run on the Intel IXP 1200, following two important design principles. The first is to filter out as much unwanted (or uninteresting) data as early as possible. In our design, when it is appropriate according to the site-specific Snort configuration, the first phase of rule-matching is moved ahead of several preprocessors to filter out unmatched packets. The second principle is to split a Snort module if it has several processing stages with very different service times and assign the stages to different processing engines. In our design, this applies to IP fragmentation reassembly, TCP stream reassembly, and rule checking.

Figure 4 shows the architecture of our prototype NNIDS on the IXP 1200. The filtering module performs packet header based filtering. If the packet received is an IP fragment, it is enqueued for fragmentation reassembly. Otherwise, it is enqueued for phase one of rule checking. Fragmentation reassembly is carried out by two sub-components. Since fragments can arrive out of order, Defrag-1 re-orders arriving fragments and inserts them into a linked list. Defrag-2 reassembles the fragments only when the set is complete. It also detects fragmentation anomalies such as overlapping fragments. It is advantageous to split the original Snort module into two threads to exploit packet-level parallelism. Similarly, TCP stream reassembly is carried out by two sub-modules. Stream-1 validates the TCP packet and maintains session state information. Stream-2 reassembles the streams when they are complete or at intermediate points that are appropriate for the underlying application protocol.

The detection module is also split into two modules. Detection-1 runs on a microengine and performs the first phase of rule checking. The most significant task in Detection-2, payload pattern-matching, requires too much computation to be run on the microengines or the StrongARM. Therefore, it is completely offloaded to the FPGA. The StrongARM uses DMA transfers to send the packets over the PCI bus to the FPGA. The FPGA compares the packet to all of the stored patterns and generates a list of pattern matches. The detection engine on the StrongARM reads the match results and determines what actions, if any, should be taken.

3.3. Network interface to host

The NNIDS runs on the network interface card so that whenever the host communicates with the outside world, the traffic in both directions is analyzed by the NNIDS. We have implemented a bi-directional path between the network and the host that is based on [22]. Figure 5 shows the data flow for incoming and outgoing traffic. A host device driver makes our platform function as a conventional Ethernet interface in Linux. Since the network interface is performing some TCP/IP functions that would normally be done by the host anyways, it would be possible to offload these tasks from the host by developing an interface to a higher layer on the network stack.

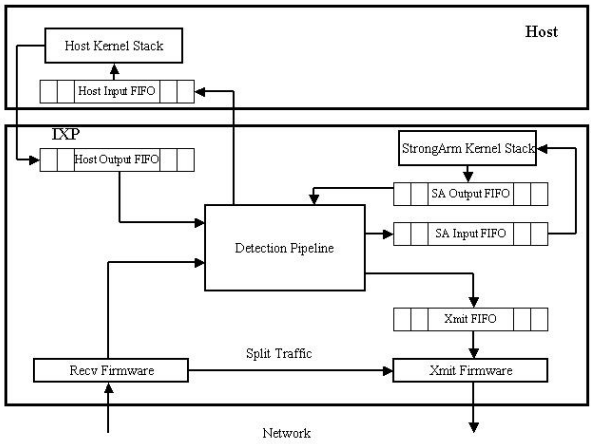

Figure 5. Network Interface with NNIDS

A region of the IXP SDRAM is mapped to the host address space and used as a packet FIFO by the device driver to transmit outbound traffic to the IXP. Similarly, a region of host RAM is mapped to the IXP address space and used as a FIFO for inbound traffic to the host. When active response is the local policy, firmware running in the IXP will determine whether to pass or drop each packet based on the detection outcome.

A second network device driver is implemented to allow the StrongARM to communicate with the outside world through the network. This enables remote administrators to send control and configuration messages to the StrongARM and receive status or alert information. In our design, all connections to the StrongARM are through this driver and treated the same. This means that a connection from the host to the StrongARM is treated the same as connections from an outside workstation, and is subject to intrusion detection processing. Thus, even when the host is compromised, the NNIDS will continue to function because attempts to compromise the system from the host can be detected and blocked by the detection engine.

3.4. Pattern-matching on FPGA

One of the most computationally-intensive tasks performed by Snort is pattern-matching on packet content [23]. Despite improved software pattern-matching algorithms [23, 24], pattern-matching is still the limiting factor in the analysis of high-speed traffic. Furthermore, the NP does not have the processing resources to handle this task. We eliminate this bottleneck by off-loading all the pattern-matching tasks to a reconfigurable FPGA co-processor.

The task of pattern-matching in NIDS consists of comparing a large number of known patterns against a stream of packets. An FPGA is well-suited for this task because it can implement thousands of pattern comparators operating in parallel. We have developed an FPGA design that compares a packet's content against every pattern in the Snort ruleset (over 1500 patterns) simultaneously [25]. This design provides high character density and high throughput, enabling the entire ruleset to fit into a low-end FPGA device while handling up to 1Gbps of data.

A block diagram of the FPGA pattern-matching co-processor is shown in Figure 6. The design is pipelined to process one character of packet data per clock cycle. An input buffer stores incoming 32-bit data words and serializes the bytes to output 8-bit characters. Next, the current character is decoded and character-match signals are distributed to the pattern-matching units. A pattern-

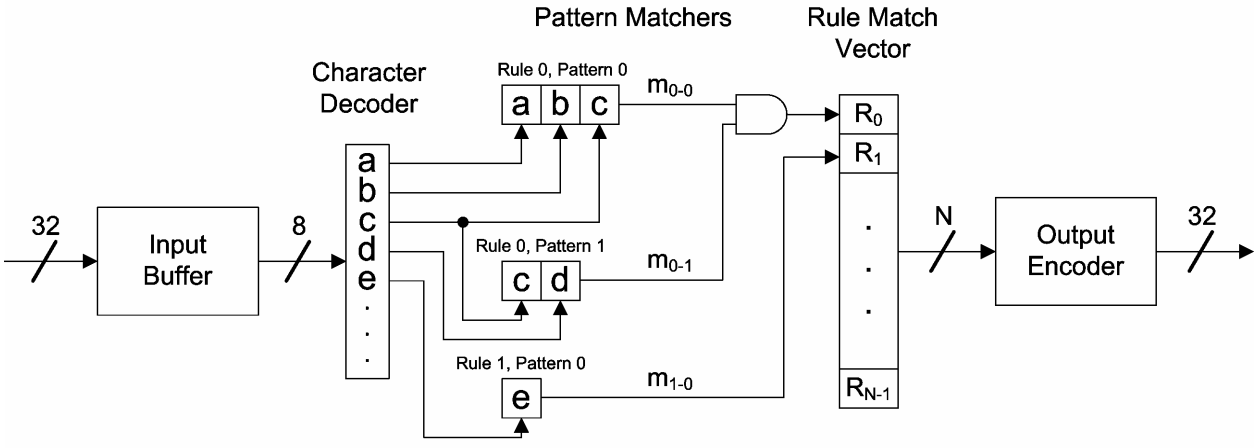

Figure 6. FPGA pattern-matching co-processor block diagram

matching unit is instantiated for each pattern in the ruleset. The pattern matchers use a non-deterministic finite automata (NFA) technique to track matches between the input data and the stored patterns. Each pattern-matching unit has an output indicating that a complete pattern match has occurred. For rules with multiple patterns, all of the corresponding pattern match outputs are passed through an AND gate to generate a rule match output. The rule match signals for all *N* rules are stored in a match vector. After the last character of a packet is processed, the output encoder packs the match results into 32-bit words and sends them to the IDS decision engine.

We have developed a software tool that translates a Snort rule file into an FPGA circuit description for matching pattern strings. The circuit generator supports all the standard Snort rule options for pattern matching. An additional feature not available in Snort is approximate pattern-matching [26]. Each pattern in a Snort rule can be specified to allow a certain number of character mismatches (substitutions, insertions, or deletions) between the pattern and a packet's content. This is useful for detecting an attack pattern that is expected to contain some variable content, but the exact variations are unknown or too numerous to list as separate patterns. It can also help detect new exploits that are similar to known exploits.

3.5. Reusable IXP libraries

Programming the microengines is difficult because there is no operating system or support library. In the course of this project, we have developed a set of libraries and development tools that are essential for building NIDS on the IXP. These include a memory management library, a queue management library, a multi-threaded packet capturing and filtering library, an IP fragment reassembly library, and a tool that converts standard `tcpdump` captures to the format used by the IXP simulator.

4. Evaluation and results

We evaluated the prototype system by performing functional verification, micro-benchmarks, and system-level benchmarks. The results are presented and analyzed in this section.

4.1. Functional verification

In order to verify that our system produces correct results, we compared it with the standard software distribution of Snort. We attached a computer with our NNIDS and a computer running standard Snort to a network hub. We also attached another computer with traffic generation software to the same hub. The traffic generator was used to send traffic containing a mixture of attack and non-attack traffic to the hub, allowing the traffic to be received simultaneously by both IDS computers. The output logs of each IDS sensor were compared, and we found that our system generated the same set of alerts as the standard Snort software.

4.2. Micro-benchmarks

For each of the NP components, we used the cycle-accurate IXP Developer's Workbench Simulator to thoroughly test the component and measure its performance. For the pattern-matching component, we ran the test in hardware and used timers in the StrongARM to measure performance.

4.2.1. Receive. Since there is a large overhead for processing each packet's header, the biggest influence on receive performance is the packet size, which determines the number of packet arrivals per second. We tested this module with a range of packet sizes and determined its achievable throughput based on the number of clock cycles required for each packet. The results are presented in Table 1.

Table 1. Receive performance

Packet size (Bytes)	Cycles / packet	Throughput (Mbps)
64	1863	64
512	3906	243
1024	6642	286

4.2.2. IP defragmentation. The critical factor in IP defragmentation processing is the number of fragments per packet. We find that the performance decreases as the number of fragments increases. The first phase of processing (Defrag-1) is a memory-bound process because the number of memory accesses required to insert fragments into the storage data structure is a function of the number of fragments, but the calculations performed on each accessed memory value are minimal. On the other hand, the second phase (Defrag-2) is a compute-bound process with execution time as a function of the number of packets because it must perform several consistency checks on each fragment before building the defragmented packet. Table 2 and Table 3 show the throughput of each phase for a 512-byte packet with varying numbers of fragments.

Table 2. Defrag-1 performance

Number of fragments	Cycles / frag	Throughput (Mbps)
4	842	282
8	931	128
16	1215	49
32	1381	22

Table 3. Defrag-2 performance

Number of fragments	Cycles / packet	Throughput (Mbps)
4	3203	297
8	4279	222
16	14519	65
32	25512	37

4.2.3. Rule-checking phase one. Detection-1 searches through a list of header values to determine if a given packet matches any of the rule header values. The list is structured so that there can be at most one match. Therefore, the worst case is when no match is found because the whole list must be traversed. This is a memory-bound process because only simple comparison tests are performed on each accessed memory value. With a single thread running this process, we find that the throughput is low in the worst case (34 Mbps) since the microengine is idle most of the time waiting for SRAM memory operations to complete. Performance could be improved by using multiple threads with each processing a different packet. Another way to help performance here would be to store the list of values in faster memory. Since the list is relatively small and changed infrequently, an ideal location would be in microengine-local memory, but this does not exist in the IXP 1200 (it does exist in the IXP 2x00).

4.2.4. Rule-checking phase two. The throughput of Detection-2 depends heavily on the time required to transfer a packet from the IXP to the FPGA over the PCI bus. As expected, the performance is better for large packets then for small packets. Once the data reaches the FPGA, the processing is completed very quickly. However, the PCI interface limits the overall performance of this module. As mentioned earlier, we hope to reduce this limitation by developing a higher-performance interface between the IXP and the FPGA. It is important to remember that our pipelined system is designed to filter uninteresting packets as soon as possible. Thus, for normal traffic, the rate of data reaching this final stage will be significantly less than the rate at the initial receive stage. Table 4 shows the worst case performance, which is when all incoming packets reach the Detection-2 phase.

Table 4. Detection-2 worst case performance

Packet size (Bytes)	Throughput (Mbps)
64	16
512	34
1024	51

The important metrics for the FPGA pattern-matcher are the number of pattern characters it can store and its throughput. We ran tests with different size rule sets loaded, including the full set default rules in the Snort software package that contains 17,537 characters. Generally with FPGAs, an increase in logic resource usage causes increased interconnect delay and reduced maximum operating frequency. Table 5 shows the throughput supported by the FPGA circuit for each rule set, but the actual throughput is limited by the PCI I/O connection.

Table 5. FPGA pattern-matching performance

Number of characters	Resource Usage	Freq (MHz)	Throughput (Mbps)
2,001	17%	119	951
4,012	25%	115	916
7,996	42%	101	809
17,537	80%	100	801

4.3. System Benchmarks

We ran some system-level benchmarks to determine how the components of the detection pipeline perform together. The testing environment was the same as that described in Section 4.1. We modeled our experiments after tests described in a report issued by the NSS Group [27], a testing lab for commercial IDS products. These tests are designed to measure the performance of the system under varying levels of load. We used a traffic generator to send different rates of fixed-size UDP packets to the NNIDS sensor. For each rate and packet size, we measured the percentage of packets that the sensor was able to process and determined the maximum rate at which the sensor could operate without dropping any packets. Because of limitations of the software and hardware in our packet-generating computer, we were not able to run tests at maximum rate with minimum-sized packets.

Since there is a fixed processing overhead for each packet, tests using small packets generally yield lower performance since there are more packets being sent per second. Due to our design goal of stopping the analysis of a packet as early as possible in the pipeline, the content of the packets has an effect on the performance. The

most significant factor is the outcome of the Detection-1 stage. If a packet's header matches the values of certain fields in one of the Snort rules, it must be further checked by the Detection-2 phase. Otherwise, no further processing is necessary. Due to the communication bottleneck in Detection-2, it can become the limiting component under high utilization. To determine the effects of packet size and Detection-1 matches, we ran two sets of tests: one with zero Detection-1 matches and one with 100 percent Detection-1 matches. The results of these tests are presented in Table 6 and Table 7, respectively.

Table 6. Best case (0% Detection-1 matches)

Packet Size	25 Mbps	50 Mbps	75 Mbps	100 Mbps	Max (Mbps)
64	100%	100%	100%	*	75
512	100%	100%	100%	100%	100
1024	100%	100%	100%	100%	100

* Our traffic generator could not send traffic at this rate for this size.

Table 7. Worst case (100% Detection-1 matches)

Packet Size	25 Mbps	50 Mbps	75 Mbps	100 Mbps	Max (Mbps)
64	69%	40%	25%	*	15
512	100%	100%	100%	100%	100
1024	100%	100%	100%	100%	100

* Our traffic generator could not send traffic at this rate for this size.

These tests show that our NNIDS network interface card, running on a 232 MHz IXP 1200 and a 100 MHz Xilinx Virtex-1000 FPGA, was able to achieve performance approximately equal to that reported by the NSS Group in their test of the Snort 2.0.2 software running on a high-end server with dual 1.8 GHz Pentium 4 processors and 2GB RAM [27].

5. Summary and Future Work

In this paper, we have discussed the need for building high-speed NIDS that can reliably generate alerts as intrusions occur and have the intrinsic ability to scale as network infrastructure and attack sophistication evolves. We have analyzed the key design principles and have argued that network intrusion detection functions should be carried out by distributed and collaborative NNIDS at the end-hosts. We have shown that an NNIDS running on the network interface instead of the host operating system can provide increased protection, reduced vulnerability to circumvention, and much lower overhead.

We have also described our experience in implementing a prototype NNIDS, based on Snort, an Intel IXP 1200, and a Xilinx Virtex-1000 FPGA. We also developed, and will make available, several libraries that are essential for building IDS on the IXP. We have conducted benchmarking experiments to study the performance characteristics of the NNIDS components. These experiments help us identify the performance bottlenecks and give insights on how to improve our design. System stress tests showed that our NNIDS can handle high-speed traffic without packet drops and achieve the same performance as the Snort software running on a dedicated high-end computer system.

Our on-going work includes optimizing the performance of our NNIDS, developing strategies for sustainable operation of the NNIDS under attacks through adaptation and active countermeasures, studying algorithms for distributed and collaborative intrusion detection, and further developing the analytical models for buffer and processor allocation. We also plan to port our design to the next generation of IXP processors and to utilize higher-performance and more tightly-integrated FPGA resources. We expect our system to reach at least 1Gbps on the IXP 2400 and even higher on the IXP 2800. We have tested FPGA pattern-matching designs that attain over 7 Gbps throughput with the entire Snort ruleset using 75% of a Xilinx Virtex2-6000 device. We are working on designs capable of pattern-matching at over 40 Gbps with a smaller ruleset or a larger FPGA.

In summary, we have provided a better understanding of the design principles and implementation techniques for building high-speed, reliable, and scalable network intrusion detection systems.

References

[1] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunninghan, and M. Zissman, "Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation," *DARPA Information Survivability Conference and Exposition*, Jan 2000.

[2] R. Lippmann, J. Haines, D. Fried, J. Korba, and K. Das., "Analysis and Results of the 1999 DARPA Off-line Intrusion Detection Evaluation," *Recent Advances in Intrusion Detection (RAID 2000)*, Oct 2000.

[3] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner, "State of the Practice of Intrusion Detection Technologies," Technical Report CMU/SEI-99-TR-028, CMU/SEI, 2000.

[4] T. H. Ptacek and T. N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Technical report, Secure Networks Inc., Jan 1998.

[5] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-time," *Computer Networks*, 31(23-24), Dec 1999.

[6] L.G. Roberts, "Beyond Moore's Law: Internet Growth Trends," *IEEE Computer*, pp. 117-119, Jan 2000.
[7] P. A. Porras and P. G. Neumann, "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances," *National Information Systems Security Conference*, Oct 1997.
[8] G. Vigna, R. A. Kemmerer, and P. Blix, "Designing a Web of Highly-configurable Intrusion Detection Sensors," *Recent Advances in Intrusion Detection (RAID 2001)*, Oct 2001.
[9] J. Balasubramaniyan, J. Garcia-Fernandez, D. Isacoff, E. Spafford, and D. Zamboni, "An Architecture for Intrusion Detection using Autonomous Agents," *14th IEEE Computer Security Applications Conference*, pp. 13-24, Dec 1998.
[10] R. Gopalakrishna and E. H. Spafford, "A Framework for Distributed Intrusion Detection Using Interest-driven Cooperating Agents," *Recent Advances in Intrusion Detection (RAID 2001)*, Oct 2001.
[11] S. M. Bellovin, "Distributed firewalls", *;login:*, Nov 1999.
[12] C. Payne and T. Markham, "Architecture and Applications for a Distributed Embedded Firewall," *17th Annual Computer Security Applications Conference*, 2001.
[13] "3Com Embedded Firewall Architecture for e-business," Technical brief, 3Com Corporation, Feb 2002.
[14] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," *USENIX LISA Conference*, Nov 1999. Software available at http://www.snort.org.
[15] "Intel Network Processors" http://www.intel.com/design/network/products/npfamily/
[16] "Virtex and Virtex-E Overview" http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=ss_vir
[17] "Gigabit Ethernet Intrusion Detection Solutions: Internet Security Systems RealSecure Network Sensors and Top Layer Networks AS3502 Gigabit AppSwitch Performance Test Results and Configuration Notes," White Paper, July 2000.
[18] C. Kruegel, F. Valeur, G. Vigna, and R.A. Kemmerer, "Stateful Intrusion Detection for High Speed Networks," *IEEE Symposium on Security and Privacy*, May 2002.
[19] "ENP-2505/2506 Data Sheet" http://www.radisys.com/oem_products/ds-page.cfm?productdatasheetsid=1055
[20] "Look Aside Interface LA-1.0," Network Processor Forum. http://www.npforum.org/techinfo/approved.shtml
[21] S. McCanne, C. Leres, and V. Jacobson, "libpcap", 1994. Available at ftp://ftp.ee.lbl.gov
[22] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev, **"**An Intel IXP1200-based Network Interface," *Workshop on Novel Uses of System Area Networks at HPCA (SAN-2 2003)*.
[23] M. Fisk and G. Varghese, "Fast Content-based Packet Handling for Intrusion Detection," Technical Report CS2001-0670, UCSD, 2001.
[24] S. Staniford, C.J. Coit, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection," *DARPA Information Survivability Conference*, 2001.
[25] C.R. Clark and D.E. Schimmel, "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns," *International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2003.
[26] C.R. Clark and D.E. Schimmel, "A Pattern-Matching Co-processor for Network Intrusion Detection Systems," *International Conference on Field-Programmable Technology (FPT)*, Dec 2003.
[27] "100Mbps IDS Group Test, Edition 4", The NSS Group, Aug 2003. http://www.nss.co.uk