DESIGN AND IMPLEMENTATION OF A HIGH-PERFORMANCE NETWORK INTRUSION PREVENTION SYSTEM

Konstantinos Xinidis[1], Kostas G. Anagnostakis[2], Evangelos P. Markatos[1]
[1]Institute of Computer Science, Foundation for Research and Technology Hellas, P.O Box 1385 Heraklio, GR-711-10 Greece {xinidis, markatos}@ics.forth.gr ; [2]Distributed Systems Laboratory, CIS Department, Univ. of Pennsylvania, 200 S. 33rd Street, Philadelphia, PA 19104 anagnost@dsl.cis.upenn.edu

Abstract: Network intrusion prevention systems provide proactive defense against security threats by detecting and blocking attack-related traffic. This task can be highly complex, and therefore, software-based network intrusion prevention systems have difficulty in handling high speed links. This paper describes the design and implementation of a high-performance network intrusion prevention system that combines the use of software-based network intrusion prevention sensors and a network processor board. The network processor acts as a customized load balancing splitter that cooperates with a set of modified content-based network intrusion detection sensors in processing network traffic. We show that the components of such a system, if co-designed, can achieve high performance, while minimizing redundant processing and communication. We have implemented the system using low-cost, off-the-shelf technology: an *IXP1200* network processor evaluation board and commodity PCs. Our evaluation shows that our enhancements can reduce the processing load of the sensors by at least 45% resulting in a system that can handle a fully-loaded Gigabit Ethernet link using at most four commodity PCs.

Key words: Network Intrusion Detection Systems, Network Intrusion Prevention Systems, network processors, load balancing

1. INTRODUCTION

The increasing importance of network infrastructure and services along with the high cost and difficulty of designing and enforcing end-system security policies has resulted in growing interest in complementary,

network-level security mechanisms, as provided by firewalls and network intrusion detection and prevention systems.

High-performance firewalls are rather easy to scale up to current edge-network speeds because their operation involves relatively simple operations such as matching a set of Access Control List-type policy rules against fixed-size packet headers. Unlike firewalls, network intrusion prevention systems (NIPSes) are significantly more complex and, as a result, are lagging behind routers and firewalls in the technology curve. The complexity stems mainly from the need to analyze not just headers but also packet content and higher-level protocols. Moreover, the function of NIPSes needs to be updated with new detection components and heuristics, due to the continuously evolving nature of network attacks.

Both complexity and the need for flexibility make it hard to design high-performance NIPSes. Application-Specific Integrated Circuits (ASICs) lack the needed flexibility while software-based systems are inherently limited in terms of performance. One design that offers both flexibility and performance is the use of multiple software-based systems behind a hardware-based load balancer. Although such a design can scale up to edge-network speeds, it still requires significant resources, in terms of the number of sensors, required rack-space, etc. It is therefore important to consider ways of improving the performance of such systems.

This paper explores the role that high-speed network processors (NPs) can play in scaling up network intrusion prevention systems. We focus on ways for exploiting the performance and programmability of NPs for boosting in-line network intrusion detection. We describe the architecture of a NIPS using commodity Personal Computers (PCs) as network intrusion detection sensors, fed by an *IXP1200*[8] network processor. We present the allocation of operations to components and the trade-offs we faced during designing and prototyping the system. For further details please refer to [20].

The rest of this paper is organized as follows. In Section 2 we describe the architecture and implementation of our system, called *Digenis*[a]. In Section 3 we examine the performance benefits of using NP-based load balancing and acceleration. We discuss work that is related to high-performance intrusion prevention in Section 4. Finally, we summarize and comment on future research directions in Section 5.

[a] Digenis Akritas, the ideal medieval Greek hero, is a bold warrior of the Euphrates frontier. He was a proficient warrior by the age of three and spent the rest of his life defending the Byzantine Empire from frontier invaders.

2. DESIGN AND IMPLEMENTATION

We faced a number of design challenges in constructing *Digenis* with respect to performance, flexibility and scalability:

Performance: The primary metric of interest in the design of a NIPS is throughput. That is, to be able to operate at network speeds of at least 1 Gbit/s without packet losses, so as to detect any attempted attack. Therefore, the system must be capable of analyzing all the incoming traffic under the most stringent conditions. Network intrusion detection systems (NIDSes) based on commodity PCs are able to monitor at speeds much lower than 1Gbit/s[2,5]. This necessitates the use of a distributed design with several intrusion detection sensors operating in parallel and supported by a load balancing traffic splitter[4,11,19]. At the same time, we want to minimize cost and use as few resources as possible. The use of an NP implementing the splitter appears reasonable, since it is likely to be cheaper than a custom ASIC, while load balancing operations seem to be well within the processing capacity of modern NPs. We also want to minimize the number of sensors needed. A key focus of our work is therefore on how to exploit the processing capacity on the NP to reduce the load of the sensors. A second important performance goal is minimizing the latency induced by the NIPS. There is a direct relationship between latency introduced by a networking device and the maximum throughput of TCP flows[b].

If the NIPS will be used at the boundary between an enterprise network and the Internet, latencies in the order of a few milliseconds may be tolerable. If the NIPS is deployed internally, and the network needs to support high-bandwidth local services (such as file sharing, etc.) the latency requirements are even more stringent. Particularly, there is a critical value for the round trip time (RTT) of a packet in each network. If the latency is below this critical value, TCP throughput is unaffected -- it is the line speed of the underlying network which becomes the bottleneck -- above this critical value, however, TCP throughput is negatively impacted. The critical value for RTT in a network supporting Gigabit speeds is 0.5 milliseconds. Thus, if we want the throughput of TCP to be unaffected, we must ensure that the latency imposed by our NIPS is less than 0.5 milliseconds.

However, Gigabit Ethernet links will rarely carry only a single TCP connection. Rather, a Gigabit Ethernet link supports hundreds, if not thousands of TCP connections, and this multiplexing mitigates the impact of latency on the overall throughput of the link[9]. In other words, it is possible

[b] Recall that TCP Throughput=Window/RTT where *Window* is the maximum TCP window size (default value is 64 Kbytes) and *RTT* is the round trip time in the network.

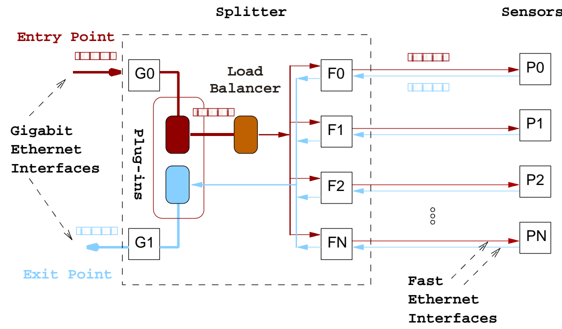

Figure 1. Architecture of *Digenis.*

to impose latency greater than 0.5 milliseconds without affecting the throughput of a link due to the high number of TCP connections.

Flexibility and Scalability: A NIPS needs to be flexible and scalable, both for scaling up to higher link speeds and more expensive detection functions, as well as for updating the detection heuristics. If the protection of a faster link or a more fine-grained detection is required, it would be desirable to reuse as much as possible of the existing hardware. Clearly, this property does not hold for ASIC-based NIPSes. However, it is remarkable that almost all NIPS providers ignore this dimension[8,13,19]. Furthermore, a prerequisite of flexibility is simplicity as extending a complex system may be hard and error-prone. It is therefore desirable for the hard-to-program elements of our system to be as generic as possible.

2.1 Architecture

Digenis is composed of a customized load balancing splitter and a number of content-based network intrusion detection sensors connected with the splitter (Figure 1). The splitter is the entry and exit point of the traffic that runs through the system. The basic task of the splitter is to evenly distribute the traffic across the sensors and to transmit the non-attack packets back to their destination. The sensors are responsible for the heavy task of inspecting the traffic for intrusion attempts. They maintain the required information for recognizing all the malicious traffic and deciding whether to forward or drop the packet. For every input packet, the splitter computes which sensor will be responsible to analyze this packet. Then, it forwards the packet to this sensor for inspection. The sensor searches for known attack patterns contained in the packet. If a pattern is found, then the packet is blocked, otherwise the packet is forwarded back to the splitter. The splitter receives the analyzed packet and transmits it to its destination.

Additionally, *Digenis* supports plug-ins that implement operations necessary to improve the performance of the system. A plug-in has two parts, one running on the splitter and one running on the sensors. These two parts cooperate in order to accomplish their task. In the context of this work we have designed a plug-in for *Digenis* that attempts to minimize the cost of sending a packet from a sensor to the splitter.

Splitter: The functionality of the splitter can be divided into the basic operations and the plug-ins that provide adequate operations to boost performance. The basic part of the splitter integrates the functionality of a load balancer -- it is responsible for distributing the incoming traffic across the output interfaces (ports). However, it differs from a common load balancer in that it must be *flow-preserving*, that is, all the packets belonging to the same flow[c] must be forwarded to the same output interface.

Regarding load balancing, there are two possible approaches that we could use: stateful load balancing that requires from the system to hold state and hash-based load balancing[3,10,16] that experiences greater load imbalances. For the purposes of this paper, we assume that load imbalances are tolerable and use the simpler hash-based method. The input of the hash function is composed of the source and destination IP addresses of the packet.

Sensor: A sensor is a commodity PC that runs a modified popular NIDS and is connected with the splitter (through an Ethernet connection). A sensor receives traffic from the splitter and analyzes it for possible known attacks. In case that an attack is found, it notifies the splitter to block the offending packet(s), otherwise it informs the splitter that the packet(s) should be forwarded. A sensor maintains state about the traffic it analyzes in order to operate correctly. The maintained state includes the active TCP connections it has captured in the near past, TCP connections tagged as offending, fragmented packets and statistics about the connections per second to TCP/UDP destination ports.

2.1.1 Reducing Redundant Packet Transmission

We have designed a plug-in for *Digenis* that is responsible for reducing redundant packet transmission on the system. The idea behind this plug-in is the following: Suppose that the splitter stores temporarily (for a few milliseconds) the packets that it forwards to the sensors for analysis. Then there is no need for the sensors to send back to the splitter the analyzed

[c] In case of TCP/UDP traffic, we define a flow to consist of all the traffic of a TCP or UDP connection. Otherwise, a flow consists of all the traffic originating from a particular IP address and destined to a particular IP address.

packet, but only a unique identifier of that packet (PID). Because the splitter has previously stored the packet with this PID, it can infer the referenced packet and forward it to the appropriate destination. The only extra work for the splitter is to tag each packet with a PID, which is a trivial task. Although the additional processing cost to the splitter from this plug-in is minimal, the reduction to the load of the sensors is remarkable. However, this technique requires from the splitter to be equipped with additional memory for the buffering of the packets. As we will present in Section 3, the memory requirements are easily satisfied by modern NPs. Subsequently, we discuss how a sensor communicates the packet information back to the splitter.

Communication between Splitter-Sensor: The splitter communicates with the sensors in order to decide the action that should be performed, that is, forward or drop the packet. This is done with acknowledgments (ACKs) from the sensors to the splitter. An ACK is an ordinary Ethernet packet. It consists of an Ethernet header, followed by two bytes denoting the number of packets acknowledged (ACK factor) and followed by a set of four-byte integers representing the PIDs. There are other possible formats requiring less bytes and supporting higher ACK factors for this configuration. However, this approach is more scalable. There are several options regarding the information that these packets should contain. The sensors may send back to the splitter the following responses:

1. **Positive ACKs:** an ACK for every packet not related to any intrusion attempt.
2. **Positive cumulative ACKs:** an ACK for a set of packets not related to any intrusion attempt.
3. **Negative ACKs:** an ACK for every packet that belongs to an offending session.
4. **Negative cumulative ACKs:** an ACK for a set of packets that belong to an attack session.
5. **The packet received**.

Each of these solutions has its pros and cons. The packet received (PR) scheme, although it has the advantage that it does not require the splitter to temporary hold the packet in memory, it suffers from low performance. In Section 3, we evaluate some of these approaches, with regard to performance. Among positive and negative cumulative ACKs (CACKs) we have chosen the former ones. Negative CACKs have two major drawbacks: First, in order to be able to distinguish when a packet must be forwarded, we have to use a timeout value. Recall that, our NIPS must not drop any packet or an attack might be missed. As a result, we would be forced to choose a

timeout for the worst case scenario. The side-effect is that packets will experience a high latency. Second, it is impossible for the splitter to differentiate the case where the analyzed packet contained no intrusion from the case where the packet was dropped due to an error condition. We have chosen positive CACKs (P-CACKs) because they supersede positive ACKs.

2.2 Implementation

We have implemented *Digenis* using low-cost, off-the-shelf technology: an Intel *IXP1200* Ethernet evaluation board and commodity PCs.

Splitter: We have implemented the splitter using an *IXP1200* network processor. The *IXP1200* chip contains six micro-engines with four hardware threads (contexts) each. Also, this chip has a general-purpose StrongARM processor core, a FIFO Bus Interface (FBI) unit and buses for off-chip memories (SRAM and SDRAM). The maximum addressable SRAM and SDRAM memory are 8 Mbytes and 256 Mbytes respectively. The FBI unit interfaces the *IXP1200* chip with the media access control (MAC) units through the IX bus. The FBI also contains a hash unit that can take 48-bit or 64-bit data and produce a 48- or 64-bit hash index. In our evaluation board, an IXF440 MAC unit (with eight Fast Ethernet interfaces) and an IXF1002 MAC unit (with two Gigabit Ethernet interfaces) are connected to the IX bus.

We have developed the application using micro-engine assembly language. The assignment of threads to tasks is done as follows: we assign eight threads for the receive part of the Gigabit Ethernet interface, one thread for the receive part of each of the eight Fast Ethernet interfaces, four threads for the transmit part of the eight Fast Ethernet interfaces, and four threads for the transmit part of the Gigabit Ethernet interface.

For the implementation of hash-based load balancing, we use the hash unit of the *IXP1200*. Also, for the temporary storage of the incoming packets until they are acknowledged we use a circular buffer which resides in SDRAM memory. This circular buffer must be large enough to prevent overwriting packets before their matching ACK is received.

Sensor: The functionality of the sensor has been implemented by modifying the popular NIDS *Snort* version 2.0.2[15]. The functionality of the sensor can be divided into three different phases: (1) the protocol decoding phase, (2) the detection phase, and (3) the prevention phase. In the first phase, the raw packet stream is separated into connections representing end-to-end activity of hosts. A connection, in case of IP traffic, can be identified by the source and destination IP addresses, transport protocol and UDP/TCP

ports. Then, a number of protocol-based operations are applied to these connections. The protocol handling ranges from network layer to application layer protocols. Some of the operations applied by the protocol-based handling are IP defragmentation, TCP stream reconstruction and identification of the URI in HTTP requests. The second phase consists of the actual detection. Here, the packet (or an equivalent higher-level protocol data unit) is checked against a database of detection heuristics representing attack patterns. Then follows the prevention phase. The action of this phase depends on the result of the previous one. If no attack is found, the sensor informs the splitter to forward the packets. If malicious activity is observed, then the prevention engine blocks the suspicious traffic by informing the splitter to not forward the packets belonging to the offending connection(s).

Extra Implementations: In addition to our splitter, for comparison purposes, we have implemented the following three configurations on the *IXP1200*:

- A forwarder (FWD) that transmits the traffic arriving at an input Gigabit Ethernet interface to an output Gigabit Ethernet interface.
- A load balancer (LB) that implements a flow-preserving load balancer with the same load-balancing characteristics as our splitter. The *IXP1200* receives traffic from a Gigabit Ethernet interface and transmits the traffic to eight Fast Ethernet interfaces.
- The last configuration (LB + FWD) implements the basic functionality of our splitter (without optimizations).

3. EVALUATION

In this Section we examine the performance of our architecture. We focus on the impact of our enhancements to sensor-splitter communication. In particular, we compare the performance of P-CACK vs. the PR scheme. We also show that such techniques can be efficiently supported by current network processors and that they do not significantly impair forwarding latency.

3.1 Experimental Environment

Splitter: The performance of the configurations running on the *IXP1200* is measured using the *IXP1200* Developer Workbench (version 2.01a)[7]. Specifically, we use the *transactor* provided by Intel. The *transactor* is a cycle-accurate architectural model of the *IXP1200* hardware. We simulate the configurations as they would run on a real *IXP1200* chip. We assume a

clock frequency of 232 MHz and a 64-bit IX bus with a clock frequency of 104 MHz.

Sensor: We use a 2.66 GHz Pentium IV Xeon processor with hyper-threading disabled. The PC has 512 Mbytes of DDR DRAM memory at 266 MHz. The PCI bus is 64-bit wide clocked at 66 MHz. The host operating system is Linux (kernel version 2.4.20, Red-Hat 9.0). The Gigabit Ethernet network interface is an Intel PRO/1000 MT Dual Port Server Adapter[6]. The sensor software is a modified *Snort* version 2.0.2, compiled with gcc version 3.2.2. We turn off all preprocessing in *Snort*. Unless noted otherwise, *Snort* is configured with the default rule-set.

Packet Traces: For the evaluation of *Digenis* we use three packet traces. The **FORTH.WEB** trace was captured at **ICS-FORTH** and only contains HTTP traffic. The **FORTH.LAN** trace was also captured at **ICS-FORTH** and contains traffic from an internal Local Area Network (LAN). Both traces contain the real payload of the packets. The **IDEVAL** traces are taken from MIT Lincoln Laboratory and were used in the 1999 DARPA Intrusion Detection Evaluation[12].

3.2 Results

3.2.1 Performance of the Splitter

All the *IXP1200* configurations described in Section 2 (LB, FWD, our splitter, and LB+FWD) handle at most the IP and UDP/TCP header of the incoming packets. Thus, we argue that the most demanding traffic for these configurations is the traffic consisting of a high percentage of small packets, namely 64-byte packets[d]. We simulate the above configurations and the results show that all the configurations are capable of sustaining line speed even with traffic consisting of only 64-byte packets[e]. This is expected as the theoretical forwarding capacity of the *IXP1200* chip is greater than 1600 Mbit/s[7].

While all the configurations sustain line speeds, we use as a metric for comparison the utilization of the micro-engines and the utilization of SRAM and SDRAM memories[f]. These are some of the resources that may become the bottleneck, considering that the *IXP1200* specification states that the maximum IX bus throughput is 6 Gbit/s. In Figure 2 we present the average utilization of the micro-engines and the utilization of the SRAM and

[d] This is the smallest possible packet in an Ethernet link including the 4-byte Ethernet CRC.
[e] The splitter uses P-CACK scheme with a factor of eight.
[f] More accurately, we measure the utilization of the buses of SRAM and SDRAM memories.

SDRAM memories for the described configurations. We observe that our approach is efficient and does not consume all the resources of the *IXP1200*, leaving headroom for even more offloading of the sensors. Particularly, the results suggest that the extra cost of the splitter compared to the load balancer is affordable[g].

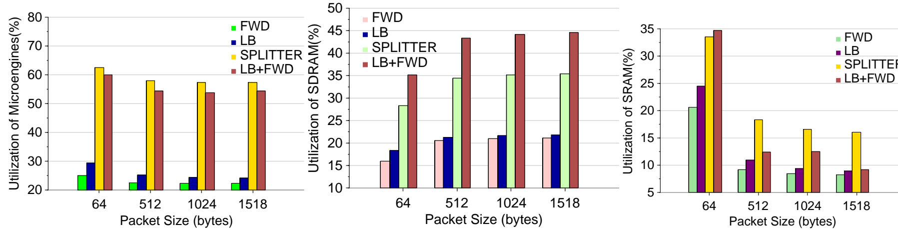

Figure 2. Utilization of the IXP1200 micro-engines, SDRAM and SRAM memories for different packet sizes. It is obvious that the splitter configuration does not consume all the resources of the *IXP1200*.

3.2.2 Performance of the Sensor

We first measure the processing cost of a sensor for different coordination schemes using the default rule-set. In this experiment *Snort* simply reads traffic from a packet trace[h], performs all the necessary NIPS functionality, and then transmits the coordination messages to a hypothetical splitter through a Gigabit Ethernet interface. Figure 3, shows the time that *Snort* spends to process all the packets for the **FORTH.WEB** trace including user and system time breakdown. The results show that the higher the P-CACK factor, the less the total running time for *Snort*. The running time is practically the same with the unmodified *Snort* for P-CACK with factor equal to 128. Also, *Snort* finished 45% faster for P-CACK with factor equal to 128 compared to the PR scheme. Moreover, we observe that the system time is lower than the user time. This confirms the fact that *Snort* spends most of its processing time in header and content matching which is counted in user time.

We also observe (Figures 3 and 4) that the improvement of the P-CACK scheme compared to the PR scheme depends very much on the trace used: the P-CACK scheme is from 45% to 3.8 times more efficient than the PR

[g] We have to mention that the increased utilization of the micro-engines in the case of the splitter configuration is caused by the instrumentation code we add to measure the performance of the splitter. While in the other configurations we do not add code for evaluation purposes, we are obliged to do so in the case of the splitter.

[h] We confirm that the hard disk is not the bottleneck by measuring the throughput of the hard disk and the transmit rate of *Snort*. As expected, the transmit rate of *Snort* is smaller than the throughput of the disk.

scheme. The reason is that the improvement depends on the detection load of the sensor. The smaller the detection load, the bigger the relative improvement. This becomes clearer if we determine where the improvement is coming from. The improvement stems from the fact that the P-CACK scheme reduces the overhead required for sending a packet to the network (*system time* in Figures 3 and 4). If the detection engine of a sensor is overloaded, then this overhead is a small fraction of the total workload of the sensor and reducing it does not lead to much improvement. In contrast, if the detection engine of a sensor is lightly loaded, this overhead consumes a significant fraction of the total workload of the sensor and reducing it results in a more notable improvement. For example, if the traffic is ruleset-intensive, then the detection load of the sensor increases and the relative improvement is small. On the other hand, for traffic that requires fewer rules to be checked for every packet, the detection load of the sensor will be minimal and the improvement will be greater.

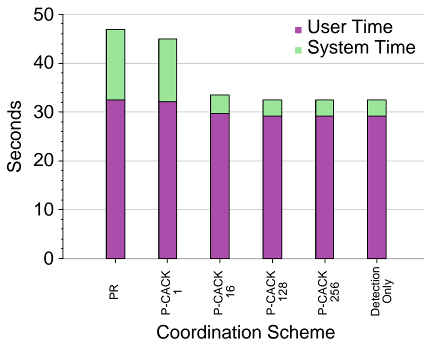

Figure 3. Processing cost of a sensor (time to process all packets in a trace), with user and system time breakdown (**FORTH.WEB** trace). We observe that the P-CACK scheme with factor 256 is 45% more efficient than the PR scheme.

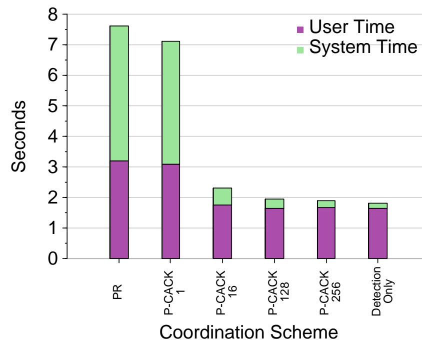

Figure 4. Processing cost of a sensor (time to process all packets in a trace), with user and system time breakdown (**IDEVAL** trace). We observe that the P-CACK scheme with factor 256 is 3.8 times more efficient than the PR scheme.

We also repeat the experiment on a PC with a slower Pentium III processor at 1.13 GHz and the same PCI bus characteristics and Ethernet network interfaces. The results (Figure 5) show that the improvement is smaller compared to the faster machine. When we examine more carefully the results, we observe that while *user time* doubles, the *system time* increases only by 30%. This happens because *user time* is mainly the time spent for content search and header matching, which are processor intensive

tasks. On the contrary, *system time* is dominated by the time spent for copying the packet from main memory, over the PCI bus, to the output network interface, handling interrupts and control registers of the Ethernet device. As the speed of processors increases faster than the speed of PCI buses and DRAM memories, we can argue that, as technology evolves, the effect of our enhancements will be even more pronounced – common processors are already running at 3.8 GHz, so the previously reported improvement is in fact a conservative result.

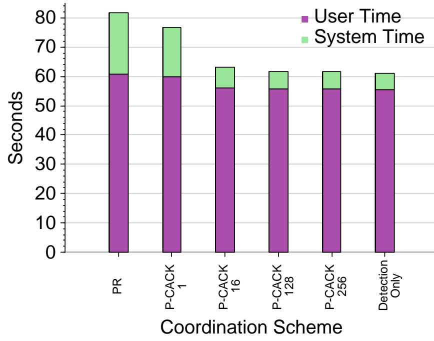

Figure 5. Processing cost of a slower sensor (FORTH.WEB trace). We can see that the improvement is smaller compared to the faster sensor.

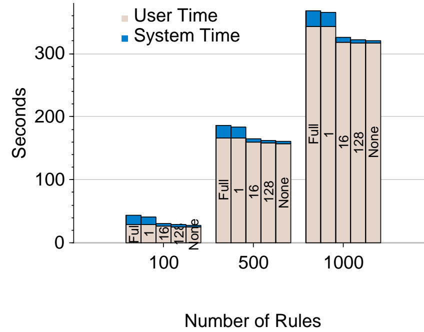

Figure 6. Performance of a sensor using incremental number of synthetic rules. We notice that as the number of rules increases the improvement of P-CACK scheme versus PR scheme decreases.

Table 1. Synthetic rule example.

alert tcp any any → any any (ack: 1; flags: S; content: ”RPC overflow”;)

All the above experiments are performed using the default rule-set of *Snort*. To further understand the correlation between the detection load of a sensor and the P-CACK scheme improvement we also experiment with variable, synthetic rule-sets. An example rule is shown in Table 1. Similarly to the previous experiment, *Snort* reads traffic from a packet trace and sends packets over a Gigabit Ethernet interface. The results are shown in Figure 6. We observe that as the number of rules increases the improvement of P-CACK scheme versus PR scheme decreases. In other words, as detection load increases, improvement decreases.

Another interesting point is that the maximum relative improvement of P-CACK over PR is for small packets of 64 bytes. Small packets require less

time for content matching *user time* and communication *system time* is the dominant cost factor. In addition, in the case of 64-byte packets, the bottleneck is not the processor, as in the case of larger packets, but the PCI bus. This is clearly shown in the experiments involving the **IDEVAL** traces (Figure 4), which contain many small packets for emulating certain types of attacks such as SYN flooding. For this trace, the P-CACK scheme is 3 times more efficient compared to the PR scheme. This is also a nice side effect of the P-CACK scheme, in that it makes the NIPS more robust against TCP SYN flood attacks, given that such attacks consist of a big fraction of small packets.

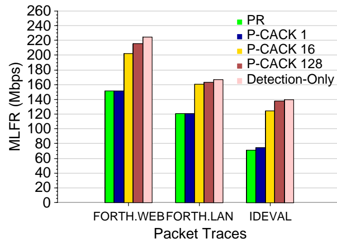

Figure 7. Maximum Loss Free Rate (MLFR) of a sensor using default rule-set.

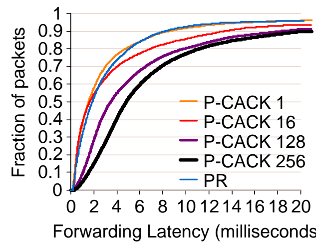

Figure 8. CDF for latency of a sensor. We notice that latency increases with the P-CACK factor.

3.2.3 Forwarding Latency of the Sensor

The highest portion of the latency imposed by our NIPS is due to content matching on the sensors. This happens due to the fact that content matching is the single most expensive operation in every NIPS. To measure forwarding latency, we use two hosts *A* and *B* with two Gigabit Ethernet network interfaces each, *eth0* and *eth1*. We connect the two interfaces of host *A* with the two interfaces of host *B* back-to-back. Everything that host *A* sends to network interface *eth0/eth1* is received by host *B* on network interface *eth0/eth1,* and vice versa. Host *A* reads a trace from a file and sends traffic to host *B* (using *tcpreplay*[1]). Host *B* runs *Snort*, which receives packets from interface *eth0* and sends replies to interface *eth1*. Host *A* matches the packet transmission time with the arrival time of the reply and computes the latency.

Initially, we estimate the maximum loss free rate (MLFR) of a sensor by replaying a packet trace and measuring the rate at which the sensor started dropping packets (Figure 7). In this experiment we set the input packet

buffer size to 16 Mbytes. We use MLFR to compute the latency that a sensor imposes to analyzed packets when reaching its processing capacity.

In this experiment, host *A* replays **FORTH.WEB** trace at the maximum loss free rate **of each communication scheme**. We observe that there are packets that experience very high latency. To understand this phenomenon, we measure the time that *Snort* spends in content and header matching using the *rdtsc*[17] instruction of the Pentium IV processor. The results show that the peaks in time spent for content and header matching coincide with the peaks in latency. This means that, when the required per packet operations increase, so does the latency. A consequence of this property is that packets that require a significant amount of processing slow down other packets that do not. This is a form of head of line (HOL) blocking.

Figure 8 shows the cumulative distribution function (CDF) for all ACK schemes when a sensor receives traffic at the MLFR of **FORTH.WEB** trace. We notice that latency increases with the P-CACK factor. An interesting observation is that the graph is heavy tailed, meaning that while most of the packets experience low latency, 5% of the packets exhibit very high latency (above 20 milliseconds). These are packets that are received from a sensor while the sensor has a temporary excess load. This may happen because, for example, some packets require too many rules to be checked. If too many such packets are received back-to-back, the system reaches (or exceeds) its capacity and the latency increases considerably.

3.2.4 Forwarding Latency of the Splitter

We argue that the overall latency that a packet experiences by our NIPS is due to the processing of the sensors and not the forwarding of the splitter. Also, the cycles spent by the splitter to forward a packet from the input interface to an output interface depend only on the packet length. This means that practically all packets of the same length experience almost the same latency.

3.2.5 Memory requirements

There is a direct relationship between latency imported by the sensors and required memory on the splitter. The splitter needs memory to save incoming packets until they are acknowledged by the sensors. The amount of memory the splitter needs depends on the highest possible latency that our NIPS will tolerate. If we set this value in a reasonable value, for example, 200 milliseconds then according to the fact that our NIPS analyzes traffic at 800 Mbit/s, the required memory is approximately 20 Mbytes. This means that the circular buffer of the *IXP1200* must be at least 20 Mbytes. This is a

reasonable requirement considering that the maximum addressable SDRAM memory of the *IXP1200* is 256 Mbytes.

4. SUMMARY AND CONCLUDING REMARKS

We have presented the design of *Digenis*, a high-performance Network Intrusion Prevention System (NIPS). The system consists of a customized load-balancing component built using the *IXP1200* Network Processor, and a number of sensors implemented on commodity PCs. In contrast to off-the-shelf load balancers used in NIPS products, our design exploits the programmability of NPs to move part of the intrusion prevention functionality from the sensors to the NP. We have focused on one method for boosting system performance by optimizing the coordination between the load balancer and the sensors. The result is a 45% improvement in performance, allowing the system to reach speeds of at least 1 Gbit/s.

There are several directions that we are currently pursuing. First, we are re-examining the structure of the sensor software. In particular, we consider the possibility of using a more fine-grained protocol processing model such as the one demonstrated by *Bro*[14], and we try to move part of the protocol processing functionality to the NP. Second, we are looking at ways for building a 10 Gbit/s NIPS using third-generation NPs.

ACKNOWLEDGEMENTS

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union, the GSRT project EAR (GSRT code: USA-022), and by ESTIA, a PAVET-NE project funded by the Greek General Secretariat of Research and Technology (PAVET-NE code: 04BEN8). Kostas Anagnostakis is also supported in part by ONR under Grant N00014-01-1-0795. Konstantinos Xinidis and E. P. Markatos are also with University of Crete. The work of Kostas Anagnostakis was done while at ICS-FORTH.

REFERENCES

1. Aaron Turner and Matt Bing. tcpreplay Tool. http://tcpreplay.sourceforge.net.
2. S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. Generating realistic workloads for intrusion detection systems. In Proceedings of the 4th ACM SIGSOFT/SIGMETRICS Workshop on Software and Performance (WOSP 2004), January 2004.
3. Z. Cao, Z.Wang, and E.W. Zegura. Performance of hashing based schemes for internet load balancing. In Proceedings of IEEE Infocom, pp. 323-341, 2000.

4. Y. Charitakis, K. G. Anagnostakis, and E. Markatos. An active splitter architecture for intrusion detection (short paper). In Proceedings of the Tenth IEEE/ACM Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2003), October 2003.
5. Y. Charitakis, D. Pnevmatikatos, E. P. Markatos, and K. G. Anagnostakis. Code generation for packet header intrusion analysis on the IXP1200 network processor. In Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2003), September 2003.
6. Intel Corporation. Intel PRO/1000 MT Dual Port Server Adapter. http://www.intel.com.
7. Intel Corporation. Intel IXP1200 Network Processor (white paper), 2000. http://developer.intel.com.
8. Internet Security Systems Inc. http://www.iss.net.
9. Intrusion Prevention Systems Group Test - Edition 1, NSS Group Ltd. http://www.nss.co.uk/acatalog/.
10. L. Kencl and J. Y. L. Boudec. Adaptive load sharing for network processors. In Proceedings of IEEE Infocom, June 2002.
11. C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In Proceedings of the IEEE Symposium on Security and Privacy, pp. 285-294, May 2002.
12. R. Lippmann, J.W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. Computer Networks, 34(4):579-595, October 2000.
13. Network Associates, Inc. http://www.networkassociates.com.
14. V. Paxson. Bro: A system for detecting network intruders in real-time. In Proceedings of the 7th USENIX Security Symposium, January 1998.
15. M. Roesch. Snort: Lightweight intrusion detection for networks. In Proc. of the second USENIX Symposium on Internet Technologies and Systems, November 1999. (Software available from http://www.snort.org).
16. R. Russo, L. Kencl, B. Metzler, and P. Droz. Scalable and adaptive load balancing on IBM Power NP. Technical report, Research Report - IBM Zurich, August 2002.
17. Time-Stamp Counter. http://www.intel.com/design/Xeon/applnots/24161825.pdf.
18. TippingPoint Technologies Inc. http://www.tippingpoint.com.
19. Top Layer Networks. http://www.toplayer.com.
20. K. Xinidis, K. G. Anagnostakis, and E. P. Markatos. Design and Implementation of a High-Performance Network Intrusion Prevention System. ICS-FORTH Technical Report 333, March 2004.