An Active Splitter Architecture for Intrusion Detection and Prevention

Konstantinos Xinidis, Ioannis Charitakis, Spiros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos

Abstract—State-of-the-art high-speed network intrusion detection and prevention systems are often designed using multiple intrusion detection sensors operating in parallel coupled with a suitable front-end load-balancing traffic splitter. In this paper, we argue that, rather than just passively providing generic load distribution, traffic splitters should implement more active operations on the traffic stream, with the goal of reducing the load on the sensors. We present an active splitter architecture and three methods for improving performance. The first is *early filtering/forwarding*, where a fraction of the packets is processed on the splitter instead of the sensors. The second is the use of *locality buffering*, where the splitter reorders packets in a way that improves memory access locality on the sensors. The third is the use of *cumulative acknowledgments*, a method that optimizes the coordination between the traffic splitter and the sensors. Our experiments suggest that early filtering reduces the number of packets to be processed by 32 percent, giving an 8 percent increase in sensor performance, locality buffers improve sensor performance by 10-18 percent, while cumulative acknowledgments improve performance by 50-90 percent. We have also developed a prototype active splitter on an IXP1200 network processor and show that the cost of the proposed approach is reasonable.

Index Terms—Network-level security and protection, network processors, intrusion detection and prevention.

1 Introduction

The increasing importance of networked services along with the high cost of enforcing end-system security policies has resulted in a growing interest in complementary, network-level security mechanisms, as provided by firewalls and network intrusion detection and prevention systems. Firewalls are network elements that filter undesirable traffic between two networks based on policies typically expressed as a set of rules to be checked against packet headers. Network Intrusion Detection Systems (NIDS) passively monitor traffic on a network and perform more advanced checks, including protocol and content inspection, to determine indications of possible attacks. Network Intrusion Prevention Systems (NIPS) combine the functionality of NIDS and firewalls, performing in-depth inspection and using this information to block possible attacks.

Firewalls are relatively easy to scale up for high-speed network links because their operation involves relatively simple operations, e.g., matching a set of Access Control List-type policy rules against fixed-size packet headers. Unlike firewalls, detection and prevention systems 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 packet headers but also packet content and higher-level protocols. Moreover, the function of these systems needs to be updated with new detection components and heuristics, considering the progress in detection technology as well as the continuously evolving nature of network attacks.

Both complexity and the need for flexibility make it hard to design a high-performance NIDS or NIPS. 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 software-based systems, required rack-space, etc. It is therefore important to consider ways of improving the performance of such systems.

This paper details our experience with examining the role of network processors (NPs) in building a high-speed NIDS/NIPS. We focus on ways for exploiting the performance and programmability of NPs for making a NIDS/NIPS more efficient. We consider an overall system structure similar to many commercial products [35], [34], that consists of a traffic splitter (implemented using NPs) that distributes the incoming traffic to sensors (implemented on general purpose PCs) for analysis.

We argue that splitters should be actively involved in analyzing traffic rather than just passively providing load-balancing functionality, with the goal of reducing the workload of the sensors and increasing the overall capacity of the system. There are different types of operations that can be performed on the splitter. First, it is possible to move part of the detection functionality to the splitter. Second, it is possible to implement optimizations and preprocessing

- *K. Xinidis, I. Charitakis, S. Antonatos and E.P. Markatos are with the Institute of Computer Science, Foundation for Research and Technology, PO Box 1385 Heraklion, GR-711-10 Greece. E-mail: {xinidis, haritak, antonat, markatos}@ics.forth.gr.*
- *K.G. Anagnostakis is with the Institute for Infocomm Research, 21 Heng Mui Keng Terrace, Singapore 119613. E-mail: kostas@i2r.a-star.edu.sg.*

Manuscript received 31 Aug. 2004; revised 2 Nov. 2005; accepted 10 Jan. 2006; published online 3 Feb. 2006.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0127-0804.

functions on the splitter, with the goal of reducing sensor load. Finally, it is possible to optimize the structure of the mechanisms used for splitter-sensor coordination. Although there are differences in the types of operations that can be performed, the overall approach applies to both detection (NIDS) and prevention (NIPS).

To illustrate our argument, we describe an active splitter architecture and analyze three mechanisms that can be implemented as part of the system. The first is based on the observation that a significant fraction of packets only require header processing. Given that header processing is relatively cheap (and can be easily performed in hardware or a network processor) we can implement an *early filtering* function (in the case of a NIDS) or an *early forwarding* function (in the case of a NIPS) as part of the splitter. The main benefit of this method is that the amount of traffic that needs to be transmitted and processed by the sensors can be reduced.

The second mechanism is based on the observation that different types of packets trigger different subsets of the NIPS rule set, placing a significant burden on the sensor memory architecture (i.e., reducing memory access locality). We present an algorithm for *locality buffering*, so that packets of the same type are grouped together on the splitter before being forwarded to the sensors. The benefit of this method is that it increases performance without altering the semantics of the traffic stream and without requiring changes on the sensors. We argue that the algorithm requires a reasonable amount of additional buffer memory and a small number of operations on each packet and can thus be efficiently implemented as part of the splitter.

The third mechanism, which only applies to prevention (NIPS) and not detection (NIDS), is based on the observation that coordination between the traffic splitter on the NP and the software-based sensors in a NIPS is inefficient, since it requires the transmission of every legitimate packet from the sensors back through the splitter. We demonstrate a more efficient coordination mechanism, using *cumulative acknowledgments*, that offers substantial performance benefits.

We must note that for the purposes of this paper, the proposed architecture and the specific mechanisms have only been examined in the context of detection and prevention that relies heavily on the string-matching model. As such, the mechanisms and the performance results presented in this work may not always be meaningful in a more general NIDS/NIPS context. For instance, while the cumulative acknowledgment scheme is independent of the detection components implemented on the sensors, its performance benefit is relative to the sensor processing workload. Similarly, the early filtering mechanism depends on the detection components as well as the incoming traffic. As the sophistication and complexity of detection increases (e.g., through the introduction of additional detection heuristics), designers many need to reexamine the effectiveness and the specific form of each mechanism. In particular, more fine-grained protocol analysis (as performed in systems such as Bro [27]) is likely to lead to higher gains in early filtering. One could, for example, completely filter out Web server response traffic, assuming it is trusted not to contain anything relevant to detection. This, however, would involve some additional processing as well as state tracking on the splitter.

Considering these observations, the main contribution of this paper is not the specific set of techniques, but the architectural argument on the choice and placement of functions in a high-performance, multilevel processing system, such as the splitter-sensor setting discussed here. To the best of our knowledge, current systems use the splitter simply as a dumb load balancer. In contrast, we have suggested that the splitter should be enhanced to be more actively involved in the detection process. This is made easier, yet not trivial, by the use of programmable NPs. However, because the (NP-based) splitter is hard to program (e.g., it has to be programmed in a low-level microassembly-like language) and can easily become a bottleneck, we were fundamentally restricted in how complex and heavyweight the functions to be implemented could be. Indeed, the functions we implemented on the splitter all turned out to be lightweight and (perhaps disappointingly) simple. Nevertheless, our results suggest that they offer significant performance benefits (e.g., roughly between 2x and 10x in throughput) at reasonably low cost. As NIDS and NIPS technology continues to evolve, the existing functions can be adapted and additional mechanisms can be added to further improve performance. Finally, as NPs mature and their programming tools improve, it will become easier to develop more sophisticated functions on the splitter.

1.1 Paper Organization

The rest of this paper is organized as follows: In Section 2, we provide a brief overview of how a NIDS/NIPS works and how load balancing is used in intrusion detection. In Section 3, we present the active splitter architecture and the performance-enhancing mechanisms that it can support. In Section 4, we present the detailed implementation of the splitter on the IXP1200 Network Processor and the modifications needed on the sensor side and, in Section 5, we present experiments examining the performance of the proposed system. We discuss related work in Section 6 and we conclude in Section 7.

2 Background

We first describe a simplified model of how a Network Intrusion Detection System (NIDS) operates. A NIDS examines network traffic and uses a variety of heuristics that try to identify attacks in the observed traffic. While research on detection heuristics is ongoing, most of the work can be classified into two broad categories: signature-based detection (compare [29]) and anomaly detection (compare [38], [39], [37], [4], [18], [19]).

In this paper, we only consider the signature-based detection methods that are implemented in systems such as `snort` [29] because they appear simpler, more mature, are in wide operational use, and are therefore much better understood than anomaly detection. Reexamining our work in the context of a wider set of detection mechanisms is a subject for future work.

The functionality of a signature-based NIDS can be divided into two different phases: the protocol decoding

phase and the detection phase. In the first phase, the raw packet stream is separated into connections representing end-to-end activity of hosts. In case of IP traffic, a connection 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 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 signatures representing attack patterns. The `snort` NIDS organizes the rule-set as a two-dimensional data-structure chain, where each element, called a *chain header*, tests the input packet against a packet header rule. When a packet header rule is matched, the chain header points to a set of signature tests, including payload signatures that trigger the execution of the pattern matching algorithm. Recent versions of `snort` organize the rule set in groups of rules that should be checked against packets that have the same destination port [32] and apply multipattern string matching algorithms [9], [11], [1] on the packet payload. Other systems, such as `Bro` [27] implement more elaborate protocol analysis rules in different ways. When an attack signature is detected, a NIDS typically issues an alert, while a NIPS would take further actions such as dropping the offending packet.

Recently, researchers have started to examine a general approach for load balancing tailored for high speed NIDS and NIPS [17]. In addition to research prototypes, commercial NIDS and NIPS load-balancing products have recently started to become available, such as [36], [28]. Although there is little publicly available information about the design of these systems, they are usually presented as dumb load balancers that simply distribute the incoming traffic to an array of off-the-shelf sensors (such as `snort`) for processing.

3 Design

There are four main goals in designing a NIPS traffic splitter. First, packets that belong to the same attack context need to be processed by the same sensor. Otherwise, certain attacks would not be detected. For content-based intrusion detection this can be achieved by mapping packets of the same flow to the same sensor. Second, traffic should be distributed so that overall system performance is maximized. Assuming a set of N identical sensors (in terms of resources, software, and configuration), a good way of achieving this is to distribute approximately $1/N$ of the total load to each sensor. Flow-level traffic distribution works well toward this goal. Third, the splitter needs to be efficient enough to operate at high network speeds. Therefore, any additional functionality should have low cost, so that the splitter does not become a bottleneck. Finally, the system should involve minimal, if any, modifications to the sensor function.

The overall architecture of our approach is shown in Fig. 1. The system is composed of an early filtering element, a load distribution element, a set of locality buffering units,

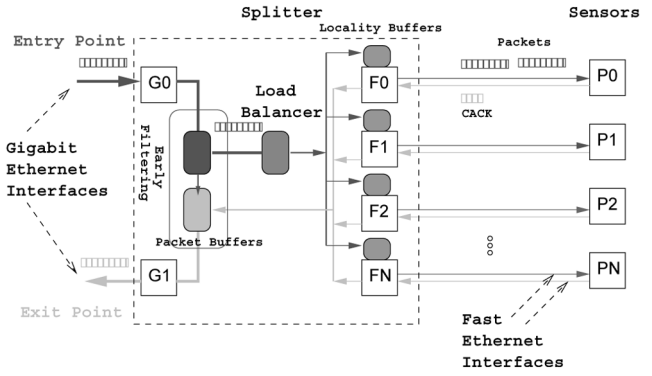

Fig. 1. The active NIPS splitter architecture.

one unit for each sensor, and a module that blocks intrusion packets based on the cumulative acknowledgments mechanism. All incoming network traffic arrives from the left side of Fig. 1 and enters the traffic splitter. The splitter, after some early-filtering preprocessing, divides the traffic through the load distribution element into separate streams and sends each of them to a different sensor which processes the incoming packets searching for possible intrusion attempts. Each sensor provides a response to our system indicating whether or not the packets it received contain an attack. All packets that do not contain an attack are forwarded to the exit point. Given that the end sensors are off-the-self intrusion detection systems, such as `snort` [29], our contribution is focused on the architecture and implementation of the traffic splitter.

In the remainder of this section, we will present each element in more detail.

3.1 Early Filtering and Forwarding

The goal of early filtering is to identify the incoming packets that do not contain any intrusions and filter them out immediately, preventing the system from unnecessarily sending them to the end sensors. The early filtering stage reduces the load on the end sensors and may also improve the performance of the overall system, as the process of sending the filtered-out packets from the splitter to the sensors is avoided.

To perform early filtering, we analyzed the default `snort` rule set (version 2.0.0) and found 165 rules that require only header (not payload) processing: We refer to this set of rules as the *EF rule set*.

Once the *EF rule set* has been identified, the splitter operates as follows: When a packet is received, it is first checked against the EF rule set. If no rule is matched and the packet contains no payload, then the packet is filtered out. Otherwise, it is forwarded to the end sensors for further processing. Note that packets that are forwarded to the sensors may belong to one of the following two classes: They matched one of the rules from the *EF rule set*, or they did not match any of the *EF rule set* rules but they contain payload. Packets belonging to the first class are forwarded to the sensors in order to be logged, while packets belonging to the second class are forwarded to the sensors in order to be examined against the rest of the `snort` rule set.

```
alert udp $EXTERNAL_NET any -> $HOME_NET 111
  (content:"|00 01 86 A0|"; depth:4; offset:12;
   content:"|00 00 00 05|"; within:4; distance:4;
   content:"|00 00 00 00|"; depth:4; offset:4;
  )
```

Fig. 2. RPC protocol rule, formulated as a string matching problem.

We must note that the specific instance of early filtering only applies to systems that are configured not to perform stateful inspection. For example, stateful inspection requires a complete TCP handshake before it checks a rule. If acknowledgments are dropped by early filtering, then stateful inspection cannot be performed. Similarly, TCP reassembly will not work if control packets (e.g., packets with SYN, ACK, or FIN flags) are dropped. Thus, this specific form of early filtering should be applied carefully, keeping in mind that some intrusion detection features may not work correctly [13]. To solve this problem, one could perform TCP reassembly on the splitter, which recent work has shown to be feasible [31].

Beyond the specific instance of early filtering presented above, one could more generally try to move *some* of the detection workload to the splitter. In particular, we have observed that many rules do not necessarily require full content inspection in terms of scanning the whole packet for a pattern at any offset. An example is shown in Fig. 2.[1] This would require rethinking the structure of a NIDS in general. Specifically, reducing detection to multipattern matching seems like a reasonable design for general purpose PCs, but more fine-grained analysis, as performed by other systems, such as Bro [27] might be a better model for multiprocessing architectures such as the one considered in this paper. This direction is outside the scope of this paper, and subject for future work.

3.2 Load Distribution

The goal of load distribution is to divide the network traffic among the end sensors so as to keep them as evenly loaded as possible. At the same time, the distribution of the network traffic should make sure that all packets of a network flow are examined by the same sensor, otherwise the system might miss an attack. As an example, think of an attack that is located at the boundaries of two packets. If the packets are sent to different sensors, the attack cannot be detected. Furthermore, preprocessing elements, such as TCP reassembly, need the entire flow to operate properly.

A simple and efficient approach for load distribution is to compute a hash function on some of the fields of the packet headers and to assign each packet to an end sensor based on the resulting value of this hash function. A hash function such as CRC16 [5], can evenly spread the flows among the sensors, so that each sensor will receive an approximately equal amount of work. Careful choice of the header fields that will be used as input to the hash function can result in a load-balancing policy that is flow preserving, i.e., packets of the same flow will be assigned to the same sensor. This can be easily accomplished by using the following header fields: protocol number, source IP address, destination IP address, source port, and destination port. Assuming well-behaved (e.g., TCP-friendly) traffic, this approach is also robust to variations in traffic load, as new flows will be assigned evenly among the available sensors. Of course, such an approach may not be robust against attackers attempting to overload the system to evade detection. Another problem with this specific instance of hash-based load balancing is that the analysis context may span across flows. For instance, analyzing FTP sessions requires processing both the control and the data connection on the same sensor. Because we include port numbers in the hash computations, the two connections are likely to be assigned to a different sensor. If hashing is performed on source and destination IP address only, then the context would be properly preserved. However, this might result in greater load imbalance. Another option would be for the FTP analyzer to explicitly set up state on the load balancer that overrides the hash-based assignment and redirects the FTP data connection to the right sensor. We have not implemented this functionality in our system, as these problems are beyond the main focus of this work.

For the purpose of our study, we have used a CRC16-like hashing function, which has been shown to perform well [5].

3.3 Locality Buffering

Locality buffering is a technique for adapting the packet stream in a way that accelerates sensor processing by improving the locality of its memory accesses and thus reducing its cache misses.

Locality buffering is based on the following observation. Each packet that arrives at the end sensor will be checked against rules that apply to the application protocol of the packet. For example, packets destined to a Web server will be checked against a set of rules which search for Web server attacks. This set of rules remains constant during the execution lifetime of the sensor. Similarly, packets destined to an FTP server will be checked against a set of rules which describe FTP server vulnerabilities. When checking a packet against a set of rules, each sensor will have to bring this rule set to the first and possibly the second-level cache of the processor. In the incoming traffic stream, packets from different network flows appear interleaved. As an example, consider a sensor that monitors a traffic stream consisting of packets belonging to a Web session and packets belonging to an FTP session. Web packets will arrive interleaved with FTP packets, which implies that the sensor may alternate the Web rule set and the FTP rule set in the cache, resulting in cache misses and reduced performance.

To increase memory locality and reduce cache misses, the proposed locality buffering mechanism attempts to rearrange the interleaving of packets in the packet stream so that packets that arrive back-to-back will trigger the same rule set as frequently as possible. To do so, our method uses a set of *locality buffers*. Instead of directly sending packets to the sensors, our approach initially places packets in locality buffers, and when a buffer becomes full, all packets are transmitted back-to-back to the target sensor. Therefore, packets arriving back-to-back at the sensor will have a

1. The *content* keyword specifies the pattern to search for in the packet payload. The *offset* keyword specifies the offset inside the packet payload to start the search and the *depth* keyword sets the maximum search depth. The *distance* keyword makes sure that there are at least N bytes between pattern matches and, finally, the *within* keyword makes sure that at most N bytes are between pattern matches.

higher probability of triggering the same rule set and improving locality. To avoid introducing latency when arrival rates are low, we periodically flush the locality buffers, while an even better solution (not implemented in our prototype) is to dynamically enable locality buffering only when sensors approach their maximum capacity. Frequently flushing the locality buffers is particularly important for a NIPS where the system latency does not only affect detection delay but also forwarding delay.

How many locality buffering units do we need and how do we assign packets to locality buffers? Ideally, we could replicate the header processing function implemented by the NIDS that decides which rule-group a packet belongs to, and allocate one locality buffering unit for each rule-group. This would be optimal in terms of performance, as it completely eliminates the possibility of packets in a single locality buffer triggering different rule-groups. However, to rely on this general approach, we need to assume that header processing is not overly expensive, or that it has to be performed anyway (e.g., to support early filtering). Furthermore, the number of rule-groups can be large,[2] and some rule-groups will rarely be triggered. Depending on the implementation the cost of maintaining an idle locality buffer could become a problem.

To make sure we address cases where it is undesirable to perform classification of packets to rule-groups that mirrors the NIDS rule set, we focus on a simpler approach based the following heuristics for determining the target locality buffer for a given packet (see Table 1):

- `src+dst`: We place a packet in a locality buffer based on the result of a hash function computed on the source and the destination ports of the packet. Using this approach we expect that packets belonging to different flows will end up in different buffers, thereby reducing packet interleaving.
- `dst`: We place a packet in a locality buffer based on the result of a hash function computed on the destination port only.
- `dst-static`: In this approach, we allocate a subset of locality buffers for known traffic types and use method `dst` for the remaining buffers/packets. For example, one buffer may receive only Web traffic, another buffer may receive only NNTP traffic, and a third buffer may receive only traffic of a popular P2P application. Unclassified packets are then allocated to the rest of the locality buffers using method `dst`, that is, hashing on the destination port only. The choice of traffic types can be made by profiling network traffic and looking at how the NIDS rule set is utilized.

Some of the positive effects of separating traffic based on port numbers may be diluted by the growing trend of applications using nonstandard (or even random) ports [16]. To counter this problem, the NIDS would have to adopt new approaches, such as [16] for identifying application protocols. Whether these new approaches can be integrated on the splitter side to be used for choosing locality buffers is unclear at this point.

2. The number of *chain-headers* is 265 for the default rule set in `snort` version 2.3.3, and 211 for `snort` version 2.0.0.

TABLE 1
Locality Buffer Allocation Methods

Method	Description
src+dst	hash(src+dst port)
dst	hash(dst port)
dst-static	Dedicated LBs for specific ports

3.4 Cumulative Acknowledgments

We have designed a simple mechanism for reducing redundant communication between the splitter and the sensors. The idea behind this mechanism 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 forward packets back through the splitter. Instead, sensors can send control messages to the splitter containing unique packet identifiers. Because the splitter has previously stored the packet with this unique identifier, it can determine the referenced packet and forward it to the appropriate destination. The only additional work for the splitter is to tag each packet with a unique identifier, which is a straightforward 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 the splitter to be equipped with additional memory for the buffering of the packets.

Our mechanism is designed as follows: The splitter needs to communicate with the sensors in order to decide the action that should be performed, e.g., to forward or drop a 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), followed by a set of four-byte integers representing the internal packet identifiers (*PID*s). There are other possible formats requiring less bytes and supporting higher ACK factors for this configuration. However, this approach seemed sufficient.

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 advantages and disadvantages. The packet received (PR) scheme does not require the splitter to temporary hold the packet in memory but it suffers in terms of performance. Negative acknowledgments 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, resulting in unnecessarily high latency. Second, it is impossible for the splitter to differentiate the case where the analyzed packet contained no attack from the case where the packet was dropped due to some error. Therefore, positive acknowledgments appear more suitable. The choice between simple ACKs and cumulative ACKs is based on the latency versus processing trade-off, which we discuss in more detail in Section 5.5.

In terms of memory requirements, there is a direct relationship between the processing latency of the sensors and the memory required on the splitter. The splitter needs memory to retain 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. A reasonable value, confirmed by measurements, is 200 milliseconds. Considering that the NIPS is supposed to analyze traffic at 1 Gbit/s, the required memory is approximately 25 MBytes.

4 Implementation

We have implemented the proposed architecture using the Intel IXP1200 network processor as the traffic splitter and general purpose PCs running a modified version of `snort` as the sensors. The IXP1200 network processor is equipped with one StrongArm processor core and six special-purpose processors called microengines. Each microengine is equipped with four hardware threads (contexts) which frequently context switch among themselves in order to mask memory latency. Also, this chip has an FBI unit and buses for off-chip memories (SRAM and SDRAM). The FBI unit connects the IXP1200 chip with the media access control (MAC) units through the Intel Exchange (IX) bus (a modified version of the PCI bus). The FBI also contains a hash unit that can take 48-bit or 64-bit data and produce a 48-bit or 64-bit hash index. In our experimental environment, the IXP1200 network processor is mounted on an ENP-2506 development board provided by Radisys. In addition to the processor, the board includes 256 MBytes of SDRAM and 8 MBytes of SRAM, two optical Gigabit Ethernet interfaces and a 64-bit external PCI interface. The IXP1200 network processor is internally clocked at 232 MHz.

The choice of `snort` on the sensor side is based on the observation that it is a widely used and mature system, that has been significantly optimized in the last few years [32], [9], [11], [1].

Concerning the development of the splitter architecture, we have used the microengine assembly language. The assignment of threads to tasks is done as follows: We assign 16 threads for the receive part of the two Gigabit Ethernet interfaces and eight threads for the transmit part of the two Gigabit Ethernet interfaces. Note that although the current implementation utilizes all the available microengines, there is headroom for further active operations to be implemented on the splitter. Regarding, the memory utilization of the IXP1200, only 32 MBytes of the total 256 MBytes are used for storing the actual contents of each packet. Also, only 2 MBytes of the total 8 MBytes of the SRAM memory are used for storing packet descriptors, per-packet metadata, locality buffer metadata, and synchronization variables. A more detailed description of the implementation of each part of our splitter architecture follows.

4.1 Early Filtering and Forwarding

To perform early filtering and forwarding on the splitter we first have to transform the set of `snort` rules into a form suitable for processing on the NP. For this purpose, we have designed S2I, a tool that transforms such a set of `snort` signatures into efficient microengine code for the microengines of IXP1200. The transformation is performed using a tree-structure in order to minimize the number of required checks. The resulting code together with a general runtime environment can be compiled, optimized, and loaded on the IXP1200 using the standard tool chain.

The benefits of this approach are based on the following observation. An interpretive approach where the signatures are kept in data structures in memory is expensive both in time (e.g., executed instructions and memory references) and space since for each signature, the interpreter input can be a large structure defining which fields to check, what operation to perform and against what value. A compiled approach is faster since it avoids the interpretation cost and allows for standard compiler optimizations. The compiled approach may also result in more compact code since many of the constants can be embedded in the instructions themselves, thus saving space.

An essential optimization pass performed by S2I is common-subexpression elimination using an expression evaluation tree. When several signatures share the same *prefix* conditions, these conditions are evaluated only once. Organizing the signature checks in a tree saves both space (each datum is stored once) and time (each condition is evaluated once). While this possibility is available to the programmer as well, implementing the code for a large number of signatures is error prone, reduces code readability, and is very hard to adapt to a new set of signatures. S2I provides performance close to that of hand-crafted code while offering the advantage of a standard and manageable high-level input specification.

For the IXP1200, the S2I compiler will also insert context swap directives in certain points of the code. Context swaps are needed to voluntary let the current thread swap out of execution so that other threads on the same microengine will have a chance to execute. This is done to avoid monopolizing a microengine for too long. If all microengines are claimed by running threads, then the buffer of the monitored port is likely to overflow, causing packet loss. More information on the S2I compiler is provided in [6].

4.2 Load Balancing

Each incoming packet received by the splitter on the NP is assigned to a target sensor that will inspect the packet for possible attacks. Sensor assignment is performed in a flow preserving manner, e.g., all packets of the same flow will always be assigned to the same sensor. This is accomplished by assigning packets to sensors based on the result of a hash function applied on the source and destination IP addresses and TCP/UDP ports of the packet.

For the implementation of the hash-based load balancing, we used the hash unit of the IXP1200. Specifically, every input packet is checked to verify that it is not an

IP fragment. If it is not a fragment, the source and destination IP addresses and UDP/TCP ports are send to the hash unit. Then, the last N bits of the result specify the destination sensor. If it is an IP fragment, then the packet is enqueued to the StrongARM. The StrongARM drains this queue and assembles the IP fragments into a nonfragmented IP packet. After the StrongARM acquires the nonfragmented IP packet, we enqueue this packet to the microengines which are then responsible to perform the hashing. The hashing function we used is CRC16.

4.3 Locality Buffering

Following sensor assignment, each packet is assigned to one of 16 locality buffers (dedicated to each sensor) based on the result of a hash function computed on the packet's destination port. An exception to this rule is packets belonging to specific traffic categories that have dedicated buffers, such as packets destined to port 80 (Web client traffic), originating from port 80 (Web server traffic), etc. When a locality buffer becomes full, all packets are enqueued in the transmit queue and transferred to the sensor in a single burst (e.g., back-to-back).

We have chosen to implement locality buffering on the splitter for two reasons. First, locality buffering is a function that is straightforward and cheap enough to implement on the splitter, as we will demonstrate in Section 5. Second, implementing it on the sensor is both cumbersome and expensive. It would require copying packets from the buffers as delivered by `libpcap` to the locality buffers, as `libpcap` (and the underlying kernel packet capture facility) is not designed to give control over buffer allocation to the application. To address this problem, one would have to modify the kernel code. This, however, results in code that is OS-specific and therefore not easily portable. Without this enhancement, any benefit derived from improved locality is overshadowed by the cost of copying packets.

4.4 Cumulative Acknowledgments

The main modification needed to support cumulative acknowledgments is for the splitter to store packets before transmission to the sensors and accessing them upon receipt of a cumulative acknowledgment. For this purpose, we use a circular buffer which resides in SDRAM memory. The circular buffer needs to be large enough to prevent overwriting packets before their matching acknowledgment is received. Before requesting an unallocated buffer, we first need to know the size of the packet; otherwise, we would be forced to use a conservative estimate that would lead to memory waste. Because the IXP1200 transfers packets in 64-byte chunks (called *mpackets*), the actual packet size is not known until the microengines receive the last *mpacket*. To avoid this problem, we extract the packet size from the IP header, which is in the first *mpacket*. Every packet received from the interface *G0* (shown in Fig. 1) is stored in the circular buffer. Then, the pointer to the next free buffer is advanced by the size of the packet. As the SDRAM on the IXP1200 is only quad-word (8 bytes) addressable, the pointer is advanced by the packet size plus some bytes for quad-word alignment.

The sensor function is implemented by modifying the `snort` NIDS. In particular, we have modified the *action* phase of the sensors, e.g., the function performed after detection, so that the sensor sends P-CACKs back to the splitter if no attack is identified. For the transmission of control packets from the sensor to the splitter, we used *libnet* [30].

More precisely, we use the tagging option of `snort` to keep track of offending sessions and to decide whether or not to transmit P-CACKs back to the splitter. This option is embedded in the rules of `snort` and gives the sensor the ability to tag the packets that are part of a current attack context. When the sensor finds an attack in a packet, it marks the session corresponding to the packet as an attack session. If, afterward, the sensor receives packets that are determined to be part of an attack session, it (silently) drops these packets and does not send P-CACKs back to the splitter. This way, the attack is effectively blocked. One can choose to block specific packets, the whole session, or all the traffic generated by the source of the offending packet. The designer of the rules can also specify how long the offending source should be blocked by providing a timeout value or a packet count threshold.

5 Experiments

In this section, we first present the effect of the proposed techniques on NIDS/NIPS performance and then examine the cost of implementing the active splitter architecture. For the experimental evaluation of the sensors, we use two different platforms. The first platform is used for the evaluation of the early filtering and locality buffering techniques while the second platform is used for the evaluation of the cumulative acknowledgments technique.

The first platform is a Dell PowerEdge 500SC equipped with a 1.13 GHz Pentium III processor PC with 8 KB L1 cache, 512 KB L2 cache and 512 MB of main memory. The host operating system is Linux (kernel version 2.4.17, Redhat 7.2). The NIDS software is `snort` version 2.0-beta20 compiled with `gcc` version 2.96 (optimization flags `O2`).

The second platform is a Dell PowerEdge 1600SC equipped with 2.66 GHz Pentium IV Xeon processor (hyper-threading disabled) and 512 MBytes of DDR memory at 266 MHz. The PCI bus is 64-bit wide clocked at 66 MHz. The host operating system is Linux (kernel version 2.4.22, Red-Hat 9.0). The NIDS software is a modified version of `snort` 2.0.2, compiled with gcc version 3.2.2. We turn off all preprocessing in `snort`. In most experiments, `snort` is configured with the default rule set.

The locality buffering experiments are performed by reading packet traces from a hard disk, while the early filtering experiments use traffic received from the network (to capture the effect of early filtering on the network subsystem). In the latter case, we use a simple network with two hosts A and B and a monitoring host S. Host A reads the trace from a file and sends traffic to host B (using `tcpreplay`) over a 100 Mbit/s Ethernet switch configured to mirror the traffic to host S. As the exact timing of trace packets has negligible effect on NIDS behavior, we simply replay the trace at maximum rate (link utilization is roughly 90 percent).

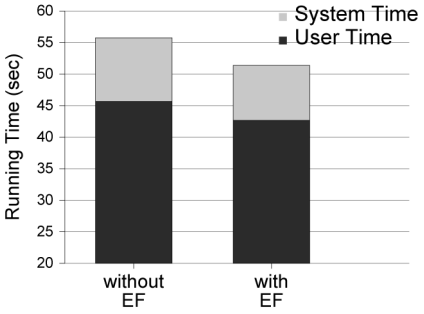

Fig. 3. The effect of early filtering on sensor performance.

We drive our experiments using a packet trace from the NLANR archive captured in September 2002 on the OC12c (622 Mbit/s) PoS link connecting the Merit premises in East Lansing to Internet2/Abilene [26]. The trace contains roughly 2.7 million packets with an average size of 762 bytes, 96 percent of which are TCP packets and 3.55 percent are UDP packets. Since the trace contains only packet headers, we retrofit packets with uniformly random data as their payload.[3]

5.1 Early Filtering/Forwarding

In our first set of experiments, we set out to explore the benefits of using early filtering. We observe that for the trace we used in our experiments, more than 40 percent of the packets do not contain any payload. A closer look reveals that most of these packets are TCP acknowledgments and more than 99 percent of these packets do not match any of the rules in the EF rule set, and can therefore be safely dropped by the splitter during early filtering.

To measure the effect of early filtering on sensor performance, we measure the user and system time of running `snort` on two traces: the original trace as well as a stripped-down trace that does not contain the packets that would have been dropped by early filtering. The results are presented in Fig. 3. The left bar of the figure shows the processing time on the original trace, while the right bar shows the processing time on the stripped-down trace. We observe that user time is reduced by 6.6 percent while system time is decreased by 16.8 percent, resulting in an overall improvement of roughly 8 percent.

5.2 Load Balancing

In this section, we explore the load-balancing properties of the CRC16 hash function which is used to distribute packets to the available sensors. For this purpose, we measure the *maximum* number of packets received by any sensor, as well as the *average* number of packets received by the sensors for the cases of two, four, and eight sensors. Fig. 4 shows the difference between the maximum and the average number of packets received by two, four, and eight sensors. We see that this difference is rather small for the case of two sensors (1.25 percent), and more noticeable for the case of eight sensors (13.55 percent).

3. It has been shown that the use of random payloads introduces an error of up to 30 percent in the measured IDS processing costs [2]. However, since we are interested in the relative (rather than the absolute) improvement in sensor performance, we believe that the benchmark is reasonable.

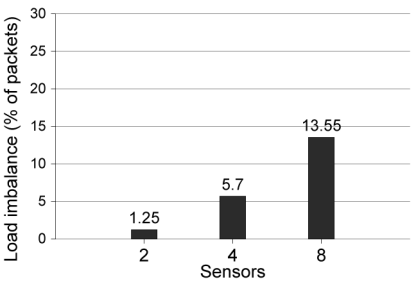

Fig. 4. Performance of CRC16-based load balancing method: difference in percent of assigned packets on most loaded sensor and fair share.

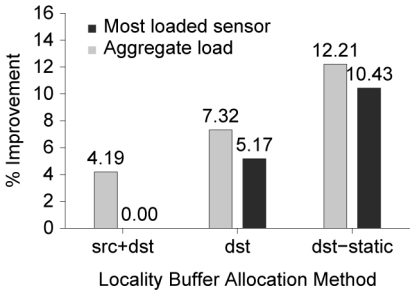

Fig. 5. Percentage of performance improvement when using different locality buffer allocation methods.

5.3 Locality Buffering

To quantify the benefit of locality buffers we measure NIDS performance using two metrics:

- aggregate user time:[4] the total user time spent by all `snort` sensors.
- maximum user time: the user time spent by the most loaded sensor.

The measurements are taken by applying the load-balancing and locality buffering algorithms on the original trace and then running `snort` on the generated trace. We determine how performance is affected by the locality buffering policy, the number of participating sensors, as well as the number and size of the locality buffers.

5.3.1 Effect of Different Locality Buffering Policies

We examine how the different policies for allocating locality buffers affect performance. For this set of experiments, we consider four sensors, 16 locality buffers per sensor, and 256 KB per buffer. Again, we measure the percentage of reduction in aggregate user time achieved by locality buffering.

Fig. 5 shows the performance improvement for different locality buffer allocation methods, in terms of the aggregate user time as well as the user time of the slowest sensor. We see that using hashing on the destination port only (*dst* policy) is better than simple hashing on both ports (*src+dst*) by more than 4 percent. The best result is obtained when assigning some of the locality buffers to specific types of traffic. This is observed in bars labeled *dst-static* which show an improvement of 12.19 percent. This is not surprising, as a

4. We ignore system time in our measurements as it is dominated by kernel overheads related to reading the network packets from the trace stored on the disk.

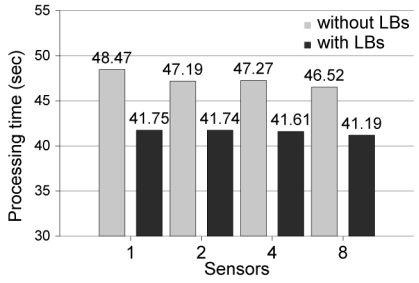

Fig. 6. Aggregate user time over all sensors versus number of sensors.

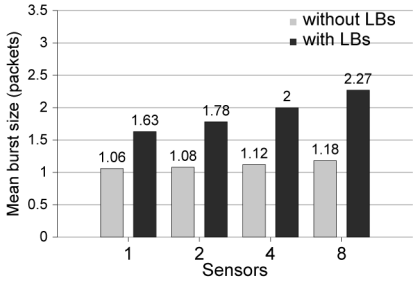

Fig. 8. Mean burst size versus number of sensors for the experiment of Figs. 6 and 7.

significant part of the trace includes Web traffic and, therefore, dedicating buffers to this kind of traffic results in longer bursts of similar packets.

5.3.2 Effect of Locality Buffers versus Number of Sensors

Fig. 6 shows the aggregate user time for different numbers of sensors, and Fig. 7 shows the user time of the slowest (e.g., the most loaded) sensor. For this set of experiments, we use 16 locality buffers of 256 KB each and the `dst-static` allocation method. Fig. 6 shows that using locality buffers reduces aggregate user time by at least 11.4 percent for eight sensors and up to 13.8 percent for a single sensor. Fig. 7 shows that using locality buffers reduces the processing load of the most loaded sensor by 9 percent-12 percent. An interesting observation from Fig. 6 is that as the number of sensors increases, the aggregate user time (in light gray bars) is decreasing. Although it is not entirely obvious why this happens, we conjecture that one possible reason is that distributing packets to a large number of different sensors, even in the absence of locality buffers, demultiplexes the incoming traffic and increases the probability of same-type back-to-back packets.

To verify this observation, we measure the average burst size, e.g., the number of consecutive packets that have the same protocol and destination port as received by the sensors. Fig. 8 presents the average burst size for one to eight sensors. By looking at Fig. 8, it is evident that the average burst size increases with the number of sensors. For example, in the absence of locality buffers, the average burst size increases from 1.06 packets to 1.18 packets, an 11 percent increase. Similarly, when locality buffers are being used, the average burst size increases from 1.63 to 2.27, a 39 percent increase. It is interesting, however, to note that the average burst size in almost all cases increases significantly with the use of locality buffers. For example, in the case of one sensor, locality buffers increase the burst size by 53 percent (from 1.06 to 1.63), and in the case of eight sensors by 92 percent (from 1.18 to 2.27).

5.3.3 Locality Buffer Dimensioning

In our next set of experiments, we investigate how performance is affected by the number of locality buffers (e.g., each for a different type of traffic) and the size of each buffer (e.g., the total amount of memory dedicated to buffer particular types of packets). We use four sensors and the locality buffers are allocated using method `dst-static`. In each experiment, we measure the difference in user time compared to a system without locality buffers.

Fig. 9 shows the results of using a different number of locality buffers per sensor when the size of each buffer is 256 KB. We observe that the improvement in aggregate user time varies between 6.8 percent (four buffers) and 12.9 percent (64 buffers). Increasing the number of locality buffers beyond 32 does not appear to offer further benefit in terms of aggregate user time, although the performance of the most loaded sensor continues to improve. This suggests that using 32 or 64 locality buffers per sensor is a reasonable design choice.

To measure how the size of each locality buffer affects performance, we measure the aggregate user time and the user time of the most loaded sensor for various buffer sizes. The results are presented in Fig. 10. The reduction in aggregate user time ranges from 9.3 percent to 13.31 percent for the cases of 64 KB and 512 KB, respectively. Using 256 KB per locality buffer seems like a reasonable choice, as the gain of increasing the buffer size from 256 KB to 512 KB is marginal.

5.4 Early Filtering Combined with Locality Buffering

To estimate the benefits of using both early filtering and locality buffering together, we apply the early filtering

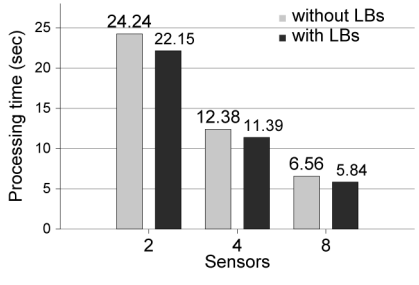

Fig. 7. User time of slowest sensor versus number of sensors for the experiments of Fig. 6.

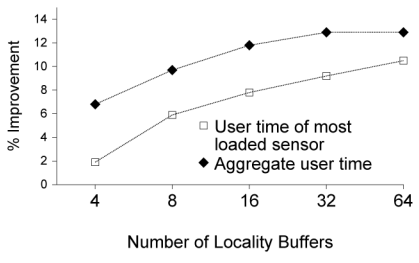

Fig. 9. Performance improvement (reduction in user time) using a different number of LBs.

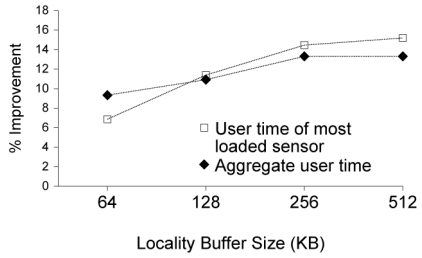

Fig. 10. Performance improvement (reduction in user time) as a function of locality buffer size.

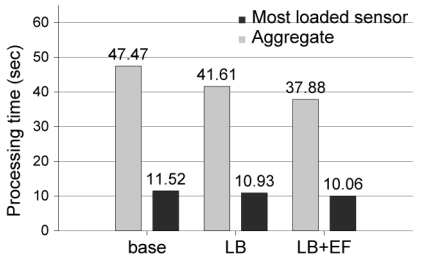

Fig. 11. Evaluation of EF + LB combined.

method on the packet trace and split the remaining packets to four sensors using 16 locality buffers of 256 KB per sensor and the *dst-static* locality buffering policy. Fig. 11 summarizes the results. The measured aggregate user time is 37.88 seconds compared to 41.61 seconds when using locality buffers only, reflecting an improvement of 8.9 percent. Compared to 47.27 seconds when not using locality buffers at all, the overall improvement of using both early filtering and locality buffering is 19.8 percent. For the slowest sensor, performance is increased by 5 percent when compared to using only locality buffers (from 11.52 to 10.93 seconds) and 14.4 percent when compared to not using early filtering or locality buffers.

5.5 Cumulative Acknowledgments

We 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,[5] performs all the necessary NIPS processing, and then transmits the coordination messages to a hypothetical splitter through a Gigabit Ethernet interface. We use three packet traces: FORTH.WEB is a trace of Web traffic obtained on a small LAN with around 50 workstations, FORTH.LAN is a trace of all traffic on a larger LAN with around 150 hosts (including both servers and workstations), and IDEVAL is a synthetic trace created specifically for IDS evaluation [21].

Fig. 12 shows the time that `snort` spends to process all the packets for the FORTH.WEB trace in terms of user and system time. The results show that the bigger the P-CACK factor, the less the total running time for `snort`. The running time is roughly the same with an unmodified detection-only sensor for a P-CACK factor equal to 128. Furthermore, `snort` is 45 percent faster for a P-CACK factor equal to 128 compared to the PR scheme. We also observe that most of the improvement is due to a reduction in system time.

5. 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.

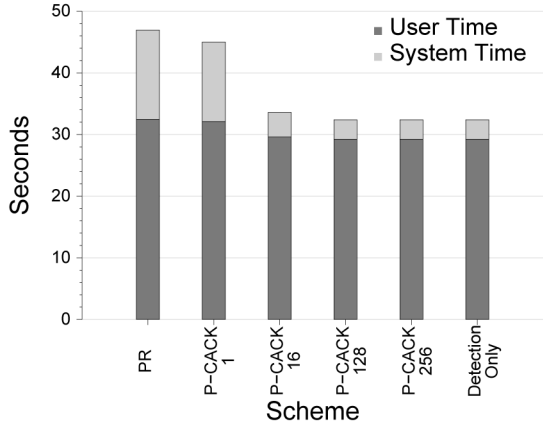

Fig. 12. Sensor processing cost (time to process all packets in a trace), with user and system time breakdown.

We also observe that the improvement of the P-CACK scheme compared to the PR scheme depends on the trace used: the P-CACK scheme was between 0.45 and 3 times more efficient than the PR 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 more clear if we determine the source of the improvement. We observe that the P-CACK scheme eliminates much of the overhead for sending packets back to the network (*system time* in Fig. 12). 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 the detection engine of a sensor is lightly loaded, this overhead consumes a big fraction of the total workload of the sensor, and reducing it results in a more notable improvement. For example, if the traffic is rule set-intensive, then the detection load of the sensor increases and the relative improvement is small. On the other hand, for traffic that requires less rules to be checked for every packet, the detection load of the sensor will be minimal and the improvement will be greater.

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 show that the improvement is smaller compared to the faster machine. When we examine the results more carefully, we observe that while *user time* doubles, the *system time* increases only by 30 percent. This happens because *user time* is mainly the time spent for content search and header matching, which are processor intensive tasks. In contrast, *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—processors are already running at 3.8 GHz and, therefore, the previously reported improvement is in fact a conservative result.

The above experiments are performed using the default rule-set of `snort`. To further understand the relationship between the detection load of a sensor and

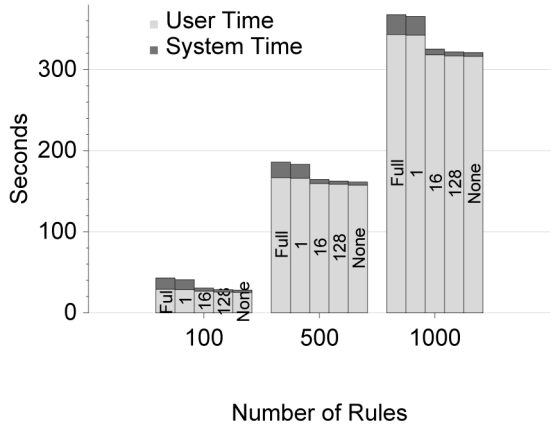

Fig. 13. Sensor performance using incremental number of synthetic rules.

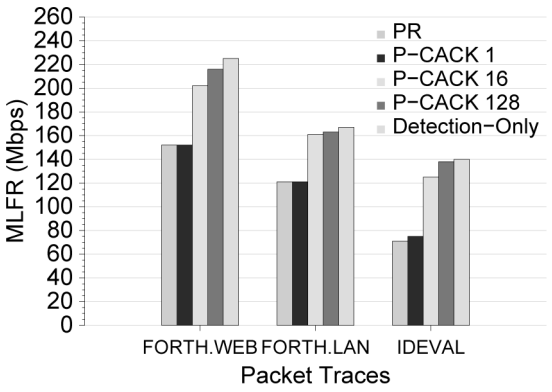

Fig. 14. Sensor Maximum Loss Free Rate (MLFR) using default rule-set.

the improvement of the P-CACK scheme, we also experiment with variable synthetic rule sets using the method of [3]. We generate synthetic rule sets that follow certain statistical properties of existing NIDS rule sets. For instance, the string-matching part of each rule is generated based on permutations of strings from a seed rule set, and the distribution of string lengths as well as the number of rules for each application protocol follow the corresponding measured distributions of the seed rule set. This approach is shown in [3] to offer a reasonable approximation as rule sets evolve over time, for the particular case of the snort NIDS. Similarly to the previous experiment, we use `snort` to read traffic from a trace and transmit packets to our system over a Gigabit Ethernet interface. The results are shown in Fig. 13. We observe that as the number of rules increases, the improvement of the P-CACK scheme versus the PR scheme decreases. In other words, as detection load increases, the improvement decreases.

Another interesting point is that we obtain the maximum relative improvement of P-CACK over PR 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, which contain many small packets for emulating certain types of attacks such as SYN flooding. For this trace, the P-CACK scheme is three 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 contain a large number of small packets.

The latency introduced by an IPS as a whole is mostly due to content matching on the sensors. This happens because content matching is the single most expensive operation in every NIPS. We first 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 (Fig. 14). In this experiment, we set the input packet buffer size to 16 MB. We see that the use of the P-CACK scheme improves MLFR considerably. The MLFR of P-CACK with a factor of 128 is very close to the MLFR of sensors that only perform detection. In other words, the additional cost of coordinating with the splitter becomes negligible.

We also measure the latency introduced by the P-CACK scheme. Fig. 15 shows the distribution of latency for all ACK schemes when a sensor receives traffic at the MLFR for the FORTH.WEB trace. We notice that latency increases with the P-CACK factor. An interesting observation is that although most packets experience very low latency, a small fraction of the packets (around 5 percent), exhibit very high latency. A closer look revealed that these are packets received while the sensor is temporarily overloaded. This happens when some packets require many rules to be checked: If too many such packets are received back-to-back, the offered load exceeds sensor capacity and latency increases considerably. To confirm this, we measured the time that `snort` spends in content and header matching using the `rdtsc` [33] instruction of the Pentium IV. The results show that the peaks in time spent for content and header matching overlap 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 may slow down other packets, which is essentially a form of head of line (HOL) blocking.

5.6 Evaluation of Network Processor Implementation

In this section, we report on the evaluation of the network-processor-based implementation. The performance of the splitter running on the IXP1200 is measured using the IXP1200 Developer Workbench (version 2.01a) [14]. Specifically, we use the *transactor* provided by Intel. The

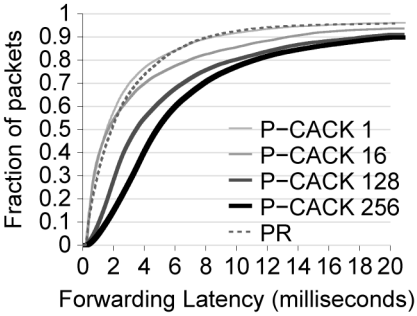

Fig. 15. Forwarding latency for NIPS with cumulative acknowledgments.

TABLE 2
Measured Capacity of the IXP1200-Based Implentations

packet size	measured throughput	theoretical maximum
64	500 Mbit/s	630 Mbit/s
1472	980 Mbit/s	980 Mbit/s

transactor is a cycle-accurate architectural model of the IXP1200 hardware. We consider four different configurations: a forwarder that includes early filtering/forwarding (EF + FWD), locality buffering (LB + FWD), all techniques (SPLITTER), and locality buffering, early filtering and forwarding (EF + LB + FWD) (without CACKs). 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.

We measure the capacity of the IXP1200-based splitter implementation. The results are shown in Table 2. We first measure only the transmission capacity of the splitter, by disabling all other functions and making the splitter transmit the same packet repeatedly over a Gigabit Ethernet link. For large packets (1,472 bytes), the system manages to achieve a transmission rate of around 980 Mbit/s which is equal to the theoretical maximum, while for small packets (64 bytes—the smallest possible packet on an Ethernet link) the achieved rate is around 500 Mbit/s. The theoretical maximum transmission rate on a Gigabit Ethernet link is around 627 Mbit/s because of Ethernet overheads and framing costs. Thus, we are limited by the IXP1200 chip to roughly 80 percent of the theoretical full line rate for 64-byte packets. Using the transmit code alone, the IXP can be used as a simple packet generator for stress-testing the performance of other network elements. To measure the processing capacity of the IXP1200-based splitter, we use one IXP1200 board as the traffic generator and another board as the splitter. The traffic generator was generating 1472-byte packets at 980 Mbit/s and 64-byte packets at 500 Mbit/s. In both experiments, the IXP1200-based splitter was able to sustain the offered load without any packet loss.

As the system sustains the full offered load, we look at the utilization of the microengines and the SRAM and SDRAM memory buses to measure the cost of the active splitter. These are the likely bottlenecks, considering, for instance, that the IXP1200 specification sets the maximum IX bus throughput to 6 Gbit/s. In Figs. 16, 17, and 18, we present the average utilization of the microengines and the SRAM and SDRAM memories for the described configurations. We note that the increased utilization of the microengines in the case of the splitter configuration is caused by the instrumentation code we had to 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. We observe that our approach is efficient and does not consume all the resources of the IXP1200. Thus, the extra cost of the active splitter compared to a passive load balancer seems affordable. Furthermore, the results indicate that there is some headroom for additional processing on the splitter, suggesting that additional active mechanisms can be supported. Finally, the difference in utilization and load between small and large packets shows that the splitter is likely to be able to support full line rates. In other words, the bottleneck is not the additional processing required for implementing the active splitter, but the maximum throughput of the IXP1200 transmission subsystem used in this experiment, which is currently limited to 500 Mbit/s.

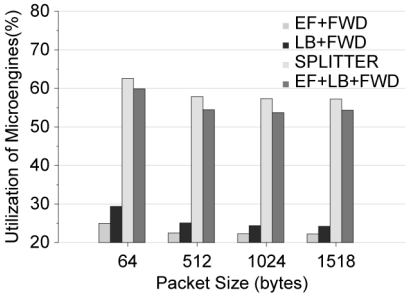

Fig. 16. Utilization of the IXP1200 microengines.

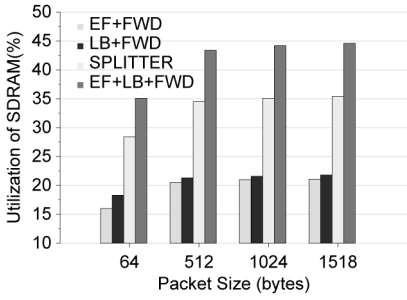

Fig. 17. Utilization of the SDRAM memory of the IXP1200.

6 Related Work

The use of load balancing for building a scalable NIDS has been examined in [17]. The authors propose a three-stage architecture for scaling stateful intrusion detection. They describe a partitioning approach that supports in-depth, stateful intrusion detection on high-speed links. The traffic is captured by a traffic *scatterer*, which equally distributes packets to a set of traffic *slicers*, in a round-robin fashion. Subsequently, the slicers are connected through a switch to a set of intrusion detection engines. The slicers examine packets for determining a suitable set of detection engines for final processing. The decision on which detection engine will analyze the packet is based on rules describing the attack contexts to which a packet may belong. The main focus of the work is to preserve detection semantics in a generalized model of intrusion detection, assuming different types of detection heuristics including statistical anomaly detection. In contrast, our work only considers

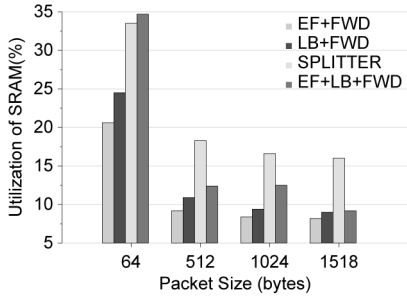

Fig. 18. Utilization of the SRAM memory of the IXP1200.

signature-based detection and, thus, relies on a simpler model of flow-preserving load balancing, focusing instead on investigating ways to offload the detection engines. This is achieved by rethinking the mapping of operations to the various components of the system.

Other research efforts recognize the issue of extensibility and have implement NIDS prototypes in reconfigurable hardware. Schuehler et al. [31] describe an architecture for a hardware-based content scanning system, capable of performing complete, stateful payload inspection on eight million TCP flows at 2.5 Gbit/s. They use a hardware circuit that combines a TCP processing engine, a per flow state store and a payload scanning engine. Similar architectures are also presented in [23], [20], [8]. One weakness of such designs is that programming hardware is likely to be more difficult than programming NPs.

A number of vendors use NPs to accelerate intrusion detection. Cisco uses IXPs on the *Cisco Catalyst 6500 Series IDS Module (IDSM-2)* [7] which is a platform capable of performing intrusion detection at 600 Mbit/s with 450-byte packets. This system supports up to 4,000 TCP connections per second (new arrivals) and up to 500,000 concurrent connections. Consystant [10] claims to have implemented `snort` on the IXP2400 network processor, but details on the structure and performance of this design are not available.

A number of vendors claim to have designed prevention systems that can operate at high speeds. For example, ISS offers the *Proventia G200* [15], a system designed for 200 Mbit/s networks. This device uses a software-based detection engine on an Intel platform. NetScreen provides the *IDP 500* [24] designed for 500 Mbit/s networks. This sensor is a hardware appliance that runs the Linux-based *IDP Sensor* software, based on the Dell PowerEdge 1750 hardware platform with dual-Pentium IV processors and 4GB RAM. McAffee has developed the *IntruShield 4000 Sensor (I-4000)* [25], claiming real-time prevention at speeds of up to 2 Gbit/s. In order to be able to reach that speed, the *I-4000* uses custom hardware for capturing packets and detecting and blocking attacks. TippingPoint uses custom high-speed security processors on the *UnityOne 2400* [34] and claims aggregate throughput of 2 Gbit/s. The *Attack Mitigator IPS 2400* [36] from Top Layer uses a combination of multiple *Attack Mitigator IPS 1000* sensors and load balancer units capable of analyzing traffic at 1 Gbit/s. Incoming traffic is evenly distributed by a load balancer to four *Attack Mitigator IPS 1000* devices and from there to a second load balancer which forwards packets to their destination.

Load balancing has been extensively used for building high-performance systems such as Web servers [12], [5]. The idea of combining filtering with load balancing is also discussed by Goldsmidt and Hunt [12], where the splitting device is instructed to block traffic destined to unpublished ports. Although the functionality proposed in [12] is similar to the functionality provided in our work, the goals are different: Our aim is to enhance sensor performance rather than to provide firewall-like protection against malicious traffic.

Locality enhancing techniques for improving server performance are also well studied. For example, [22] presents techniques for improving request locality on a Web cache, demonstrating significant benefits in file system performance. However, to the best of our knowledge, the locality buffering technique presented here is the first attempt to provide locality enhancements as part of a load balancer, and the first to do so in the context of intrusion detection.

7 Summary

We have proposed an active traffic splitter architecture for building network intrusion detection systems (NIDS) and network intrusion prevention systems (NIPS). Rather than acting as a passive load-balancing component, we have argued that the traffic splitter should actively manipulate the traffic stream ways that increase sensor performance.

We have presented and analyzed three specific examples of performance-enhancing techniques that have been implemented as part of our architecture: early filtering/forwarding, locality buffering, and cumulative acknowledgments. These mechanisms offer significant performance benefits in terms of reducing the processing load on the system as a whole. Our experiments have demonstrated improvements of 8 percent for early filtering, 10-17 percent for locality buffering, and 45-90 percent for cumulative acknowledgments. We have also confirmed that the implementation of the architecture using IXP1200 network processors is feasible.

Based on these results, we claim that active splitters are an effective way to scale the performance of NIDS and NIPS systems, enabling them to effectively monitor high-speed network links.

Acknowledgments

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union and the GSRT project EAR (GSRT code: USA-022). K. Xinidis, I. Charitakis, S. Antonatos, and E.P. Markatos are also with University of Crete. The authors would like to thank the members of the DCS group at FORTH-ICS, Lam Vinh The (Terry) and the anonymous reviewers for useful suggestions and feedback on earlier versions of this paper.

References

[1] K.G. Anagnostakis, S. Antonatos, M. Polychronakis, and E.P. Markatos, "E^2xB: A Domain-Specific String Matching Algorithm for Intrusion Detection," *Proc. IFIP Int'l Information Security Conf. (SEC '03)*, May 2003.

[2] S. Antonatos, K.G. Anagnostakis, and E.P. Markatos, "Generating Realistic Workloads for Intrusion Detection Systems," *Proc. Fourth ACM SIGSOFT/SIGMETRICS Workshop Software and Performance (WOSP '04)*, Jan. 2004.

[3] S. Antonatos, K.G. Anagnostakis, M. Polychronakis, and E.P. Markatos, "Performance Analysis of Content Matching Intrusion Detection Systems," *Proc. Fourth IEEE/IPSJ Symp. Applications and the Internet (SAINT 2004)*, Jan. 2004.

[4] M. Bhattacharyya, M.G. Schultz, E. Eskin, S. Hershkop, and S.J. Stolfo, "MET: An Experimental System for Malicious Email Tracking," *Proc. New Security Paradigms Workshop (NSPW)*, pp. 1-12, Sept. 2002.

[5] Z. Cao, Z. Wang, and E.W. Zegura, "Performance of Hashing-Based Schemes for Internet Load Balancing," *Proc. IEEE Infocom*, pp. 323-341, 2000.

[6] I. Charitakis, D. Pnevmatikatos, E.P. Markatos, and K.G. Anagnostakis, "Code Generation for Packet Header Intrusion Analysis on the IXP1200 Network Processor," *Proc. Seventh Int'l Workshop Software and Compilers for Embedded Systems (SCOPES '03)*, Sept. 2003.

[7] Cisco Catalyst 6500 Series IDS Module (IDSM-2)}, http://www.cisco.com, 2006.

[8] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas, "A Hardware Platform for Network Intrusion Detection and Prevention," *Proc. Third Workshop Network Processors and Applications (NP3),* Feb. 2004.
[9] C.J. Coit, S. Staniford, and J. McAlerney, "Towards Faster Pattern Matching for Intrusion Detection, or Exceeding the Speed of Snort," *Proc. Second DARPA Information Survivability Conf. and Exposition (DISCEX II),* June 2002.
[10] Consystant Design Technologies, http://www.consystant.com, 2005.
[11] M. Fisk and G. Vargheseau, "An Analysis of Fast String Matching Applied to Content-Based Forwarding and Intrusion Detection," Technical Report CS2001-0670 (updated version), Univ. of California at San Diego, 2002.
[12] G. Goldszmidt and G. Hunt, "Scaling Internet Services by Dynamic Allocation of Connections," *Proc. Sixth IFIP/IEEE Int'l Symp. Intergrated Network Management,* pp. 171-184, May 1999.
[13] M. Handley, V. Paxson, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," *Proc. 10th USENIX Security Symp.,* 2001.
[14] Intel Corporation, "Intel IXP1200 Network Processor,"white paper, 2000, http://developer.intel.com.
[15] Internet Security Systems Inc., http://www.iss.net, 2006.
[16] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy, "Transport Layer Identification of P2P Traffic," *Proc. Internet Measurement Conf. (IMC),* Oct. 2004.
[17] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer, "Stateful Intrusion Detection for High-Speed Networks," *Proc. IEEE Symp. Security and Privacy,* pp. 285-294, May 2002.
[18] C. Kruegel and G. Vigna, "Anomaly Detection of Web-Based Attacks," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS),* pp. 251-261, Oct. 2003.
[19] W. Lee, S.J. Stolfo, P.K. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, and J. Zhang, "Real-Time Data Mining Based Intrusion Detection," *Proc. DISCEX II,* June 2001.
[20] S. Li, J. Torresen, and O. Soraasen, "Exploiting Reconfigurable Hardware for Network Security," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '03),* Apr. 2003.
[21] R. Lippmann, J.W. Haines, D.J. Fried, J. Korba, and K. Das, "The 1999 DARPA Off-Line Intrusion Detection Evaluation," *Computer Networks,* vol. 34, no. 4, pp. 579-595, Oct. 2000.
[22] E.P. Markatos, D.N. Pnevmatikatos, M.D. Flouris, and M.G.H. Katevenis, "Web-Conscious Storage Management for Web Proxies," *IEEE/ACM Trans. Networks,* vol. 10, no. 6, pp. 735-748, 2002.
[23] M. Necker, D. Contis, and D. Schimmel, "TCP-Stream Reassembly and State Tracking in Hardware," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '02),* Apr. 2002.
[24] NetScreen Technologies, http://www.netscreen.com, 2005.
[25] Network Associates, Inc., http://www.networkassociates.com, 2005.
[26] NLANR, "MRA Traffic Archive," Sept. 2002, http://pma.nlanr.net/PMA/Sites/MRA.html.
[27] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Proc. Seventh USENIX Security Symp.,* Jan. 1998.
[28] Peapod, "Radware Linkproof,"http://www.peapod.co.uk/radware_linkproof.htm, 2006.
[29] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," *Proc. Second USENIX Symp. Internet Technologies and Systems,* Nov. 1999, http://www.snort.org.
[30] M. Schiffman, "The Million Packet March," http://www.packetfactory.net/Projects/Libnet/, 2006.
[31] D.V. Schuehler, J. Moscola, and J.W. Lockwood, "Architecture for a Hardware-Based, TCP/IP Content-Processing System," *IEEE Micro,* vol. 24, no. 1, pp. 62-69, 2004.
[32] Sourcefire, Snort 2.0 - Detection Revisited, Oct. 2002, http://www.snort.org/docs/Snort_20_v4.pdf.
[33] Intel Xeon Processor MP Specification Update, Oct. 2005, http://download.intel.com/design/Xeon/specupdt/29074135.pdf.
[34] TippingPoint Technolgies Inc., http://www.tippingpoint.com, 2005.
[35] Top Layer Networks, http://www.toplayer.com, 2006.
[36] TopLayer, "IDS Load Balancer,"http://www.toplayer.com/, 2006.
[37] T. Toth and C. Kruegel, "Accurate Buffer Overflow Detection via Abstract Payload Execution," *Proc. Fifth Symp. Recent Advances in Intrusion Detection (RAID),* Oct. 2002.
[38] T. Toth and C. Kruegel, "Connection-History Based Anomaly Detection," *Proc. IEEE Workshop Information Assurance and Security,* June 2002.
[39] K. Wang and S.J. Stolfo, "Anomalous Payload-Based Network Intrusion Detection," *Proc. Seventh Int'l Symp. Recent Advanced in Intrusion Detection (RAID),* pp. 201-222, Sept. 2004.

Konstantinos Xinidis received the MSc degree and diploma in computer science from the University of Crete. His main research interests are in network monitoring, intrusion detection, and network processors.

Ioannis Charitakis received the MSc degree and diploma in computer science from the University of Crete. His main research interests are in network monitoring, intrusion detection, and network processors.

Spiros Antonatos received the MSc degree and diploma in Computer Science from the University of Crete. He is a PhD candidate in the Computer Science Department at the University of Crete. His main research interests are in network monitoring, intrusion detection, and performance evaluation.

Kostas G. Anagnostakis received the BSc degree in computer science from the University of Crete and the master's and PhD degrees in computer and information science from the University of Pennsylvania. He is currently a principal investigator on software systems security at the Institute for Infocomm Research (I2R) in Singapore. His main areas of interest are in distributed systems security, networking, performance evaluation, and in problems that lie at the intersection between computer science and economics.

Evangelos P. Markatos received the diploma in computer engineering from the University of Patras in 1988, and the MS and PhD degrees in computer science from the University of Rochester, New York, in 1990 and 1993, respectively. Since 1992, he has been an associated researcher at the Institute of Computer Science of the Foundation for Research and Technology-Hellas (ICS-FORTH) where he is currently the head of the Distributed Computing Systems Laboratory, and the head of the W3C Office in Greece. Since 1994, he has also been with the Computer Science Department at the University of Crete, where he is currently a full professor. He conducts research in several areas including distributed and parallel systems, the World Wide Web, Internet systems and technologies, as well as computer and communication systems security. He has been a reviewer for several prestigious journals, conferences, and IT projects. He is the author of more than 70 papers and book chapters. He is currently the coordinator of research projects funded by the European Union, by the Greek government, and by private organizations.