An Active Traffic Splitter Architecture for Intrusion Detection*

I. Charitakis*, K. Anagnostakis†, E. Markatos*

*Institute of Computer Science
Foundation for Research and Technology - Hellas
P.O.Box 1385 Heraklio, GR-711-10 GREECE
{haritak,markatos}@ics.forth.gr

†Distributed Systems Laboratory
CIS Department, Univ. of Pennsylvania
200 S. 33rd Street, Phila, PA 19104, USA
anagnost@dsl.cis.upenn.edu

Abstract

Scaling network intrusion detection to high network speeds can be achieved using multiple sensors operating in parallel coupled with a suitable load balancing traffic splitter. This paper examines a splitter architecture that incorporates two methods for improving system performance: the first is the use of early filtering where a portion 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. Our experiments suggest that early filtering reduces the number of packets to be processed by 32%, giving a 8% increase in sensor performance, while locality buffers improve sensor performance by about 10%. Combined together, the two methods result in an overall improvement of 20% while the performance of the slowest sensor is improved by 14%.

1 Introduction

Network Intrusion Detection is receiving considerable attention as a mechanism for shielding against "attempts to compromise the confidentiality, integrity, availability, or to bypass the security mechanisms of a computer network" [3]. The typical function of a Network Intrusion Detection System (nIDS) is based on a set of *signatures*, each describing one known intrusion threat. A nIDS examines network traffic and determines whether any signatures indicating intrusion attempts are matched.

Effective intrusion detection requires significant computational resources: widely deployed systems such as *snort* [18] need to match packet headers and payloads against tens of header rules and often many hundreds of strings defining attack signatures. This task is much more expensive than the typical header processing performed by packet forwarders and firewalls. Therefore, performing intrusion detection at high network speeds (e.g. 1 Gbit/s and beyond) requires the use of multiple sensors operating in parallel, fed by a suitable traffic splitter element.

The characteristics of intrusion detection place certain constraints on the design of a traffic splitter. For instance, as observed by Kruegel *et al.* [11], packets that are part of a given attack context need to be processed by the same sensor. For widely-used systems such as *snort* that perform session-level content-matching this simply requires packets of a given session (flow) to be mapped to the same sensor; adequate solutions for flow-preserving load balancing already exist[4].

Given the high resource demands of intrusion detection, we consider ways of boosting sensor performance by rethinking the design of nIDS traffic splitters. We argue that traffic splitters should implement more active operations on the traffic stream with the goal of reducing the load on the sensors, rather than just passively providing generic, flow-preserving load distribution.

This paper presents two such active mechanisms. 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) we can implement this function as part of the splitter. The main benefit of this method of *early filtering* is that the amount of traffic that needs to be transmitted and processed by the sensors can be reduced significantly.

The second mechanism is based on the observation that different types of packets trigger different subsets of the nIDS ruleset, 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 work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union. The second author is also supported by ONR under Grant N00014-01-1-0795.

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 be efficiently implemented as part of the splitter.

We present experiments using the *snort* nIDS and traces from real network traffic analyzing the effect of the proposed methods on nIDS performance.

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 works and how load balancing is used for building scalable Internet services, including intrusion detection. In Section 3 we present a nIDS load balancing architecture implementing the proposed early filtering and locality buffering policies. In Section 4 we present experiments examining the performance of the proposed methods. In Section 5 we discuss implementation issues and conclude in Section 6.

2 Background

Network Intrusion Detection. We describe a (simplified) model of how a Network Intrusion Detection System (nIDS) operates. A nIDS examines network traffic and determines whether any signatures indicating intrusion attempts are matched. The simplest and most common form of nIDS inspection is to perform protocol header analysis and match string patterns against the payload of packets captured on a network link. Known systems following this model are *snort*[18] and *Bro*[17].

A nIDS is built as passive monitoring system that reads packets from a network interface through packet capture facilities such as `libpcap`[14]. After a number of normalization passes, each packet is checked against the nIDS ruleset. A ruleset is typically organized as a two-dimensional chain data-structure, where each element - often called a *chain header* - tests the input packet against a packet header rule. When a packet header rule is matched, the chain header usually points to a set of signature tests, including payload signatures that trigger the execution of a string matching algorithm. String matching is the single most expensive part of nIDS operation and several nIDS-specific algorithms have been recently proposed[5, 7, 1].

Traffic load balancing. Traffic load balancing is a well studied method for building scalable Internet services, and has been applied to widely deployed systems such as Web servers [8]. In [8] load balancing is based on explicit feedback from each server. In this way load balancing is almost perfect. However, state is required so as to forward the packets of the same flow always to the same server. This makes the splitter more complicated and therefore the scaling to Gbit/s speeds becomes more difficult. The specifics of the load balancing policy are important in examining the effect of the mechanisms we propose to be implemented as part of the traffic splitter. However, for this purpose of this paper, we assume that load imbalances are tolerable and use a simpler hash-based method.

Using a hash function for flow-preserving load balancing is quite common [19, 6, 4]. In case of Web caches [19] it has been shown that a hashing method can be used to determine the target server. It is also shown that the hashing method can be adapted for providing compensation for server failures and supporting dynamic addition of new servers. In [6], the authors suggest a hybrid approach of hashing and preservation of state. The authors use per-flow hashing to classify a flow to flow groups. Per-group state is then used to choose a server for all flows within a group. A detailed study of hash functions for load balancing is presented in [4]. Other recent work on load balancing includes [20] and [10] where the authors investigate the implementation of load balancing using network processors.

nIDS load balancing. There are many IDS load balancing products available, such as [21], but with little information publicly available on details of the design and the load balancing policies. Recent research [11] examines a general approach for load balancing as applied to high speed intrusion detection. The authors propose an two-stage architecture for determining the set of sensors that will process a given packet. The first stage attempts to equally distribute packets while the second stage examines packets for determining a suitable set of sensors for final processing. The decision of where to send a packet is based on rules describing the attack contexts to which a packet may belong. The main focus of the work is therefore in preserving detection semantics in a generalized model of intrusion detection, assuming different types of detection such as statistical methods, anomaly detection and content-matching. In contrast, the work we present in this paper is performance-oriented and focuses on the specific case of content-matching intrusion detection as widely deployed today.

Early Filtering and load balancing. The idea of providing filtering functionality on a load balancer is also discussed in [8], where the splitting device is instructed to block traffic destined to unpublished ports. Although the functionality provided is similar, the goals are different: our goal is to enhance sensor performance, not provide firewall-like protection from irrelevant or malicious traffic.

Locality enhancing techniques. Locality enhancing techniques for improving server performance are well studied. For example, in [13] the authors try to improve request locality on a Web cache, demonstrating significant improvements in file system performance. To the best of our knowl-

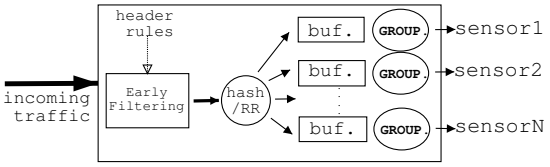

Figure 1. The active nIDS splitter architecture

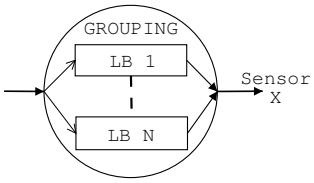

Figure 2. Packet grouping using Locality Buffers

edge, our work is the first attempt to providing locality enhancements as part of a load balancer, and the first to do so in the context of intrusion detection.

3 Design

There are four main goals in designing a nIDS 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, and we will discuss how early filtering and locality buffering can provide further benefits. Third, the load balancing algorithm needs to be efficient enough to operate at high network speeds. Fourth, the system should (ideally) not require modification to the sensor function.

The system architecture is shown in Figure 1. The system is composed of an early filtering element, a load distribution element and a set of locality buffering units, one for each sensor. In the remainder of this Section, we will present each of the elements in more detail.

3.1 Early filtering

The basic idea in early filtering is to implement part of the sensor functionality on the splitter. Since a fraction of packets is only subject to header analysis, which is significantly cheaper than content-matching, we can efficiently perform this function on the splitter. This is expected to reduce the load on the sensor but also on the overall system, as the process of sending packets from the splitter to the sensors can often be avoided.

To perform early filtering we analyze the nIDS ruleset and extract the rules that do not require content matching. We observe that this is a small portion of the default ruleset in *snort*: only 165 of 1700 rules. We refer to this set of rules as the *EF ruleset*. We expect that processing a small number of header rules in the EF ruleset on the splitter can be easily supported in hardware.

The splitter then operates as follows. When a packet is received it is first checked against the EF ruleset. If no rule is matched and the packet contains no payload then the packet is discarded. If no rule is matched but the packet does contain a payload that needs to be analyzed, it is forwarded to one of the sensors. If the packet matches a rule then again, it is forwarded to the sensors for generating an alert.

3.2 Load distribution

A simple and efficient approach for load distribution is to use a hash function on the packet headers. A hash function such as CRC16 can evenly spread the flows among the sensors, so that each sensor will receive an approximately equal amount of work. Careful choice of the input to the hash function can result in a load balancing policy that is flow preserving (e.g. packets of the same flow will be assigned to the same sensor); this can be accomplished by hashing on flow identifiers (*i.e.,* src[IP,port], dst[IP,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 sensors. Of course, this is not robust if traffic is not well-behaved (for instance, because of an attacker attempting to overload the system to evade detection). However, this problem is beyond the scope of this work and is also not specific to the proposed enhancements.

For the purpose of our study we have used a CRC16-like hashing function as it has been shown to perform well[4], requires no state and there are known hardware implementations.

3.3 Locality buffering

Locality buffering is a method for adapting the packet stream in a way that improves performance of each nIDS sensor. The idea is based on the fact that there are specific rulesets for specific types of traffic. Consequently, when checking a packet, each sensor will have to bring the working set that corresponds to the rules of that packet. Alternat-

ing between rulesets results in cache misses due to conflicts and therefore performance degrades.

To increase memory locality, the splitter analyzes incoming packets, and then places them on separate buffers trying to keep in each buffer packets of the same type. When a buffer becomes full, all packets are transmitted to the target sensor in sequence. This increases the average number of same-type packets that are received by the sensor back-to-back. Therefore increases memory locality which in turn results in better performance.

Exact classification of each packet according to the nIDS rule-groups can be complicated, we have opted for a simpler solution based on the following heuristics for determining the target locality buffer for a given packet:

SD Use a hash on source and destination ports of the packet, hereby evenly distributing packets among the target buffers. As the input stream is separated in different buffers, we would expect a higher probability for packets of the same type to arrive on a sensor back-to-back.

D Use a hash on destination port only. We expect this to further increase the probability of back to back packets of the same type compared to A, given that only the destination port is taken into account.

p_* Allocate a set of locality buffers for certain transport protocols. In this way, the LBs are divided into groups for traffic from a given protocol. For example, 12 out of 16 buffers are allocated for TCP packets and the remaining 4 buffers are used for other protocols. Within each LB-group, allocation is performed using methods SD or D - we therefore call these methods p_SD and p_D.

T_D Allocate a set of locality buffers for known traffic types and use method D for the remaining buffers/packets. For example, LB1 would receive only Web traffic, LB2 only NNTP traffic and LB3 only P2P traffic. Unclassified packets are then allocated to the rest of the LBs using method D hashing on the destination port only. The choice of traffic types can be made by profiling real network traffic and looking at how the nIDS ruleset is utilized.

p_T_D Allocate some locality buffers for specific protocols, some for specific types of traffic and the rest using method D. In this way, locality buffers are first split into groups accepting traffic of a specific protocol. (As in method p_*). Within each group some LBs will be dedicated to specific traffic types of the corresponding protocol and the rest will be allocated using D. (As in method T_D).

Method	Description
SD	hash(*S*rc+*D*st port)
D	hash(*D*st port)
p_SD	Divide LBs by *p*rotocol Apply method *SD*
p_D	Divide LBs by *p*rotocol Apply method *D*
T_D	Dedicate LBs to specific traffic *T*ype + method *D*
p_T_D	Divide LBs by *p*rotocol Apply method *T_D*

Table 1. Locality buffer allocation methods

The performance implications of these heuristics are analyzed in Section 4.

4 Experiments

We present experiments examining the effect of early filtering and locality buffering on nIDS performance.

For most experiments we use 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`).

Most experiments are performed by reading a packet trace from disk, except for the early filtering experiments where traffic is received from the network (to capture the effect of early filtering on the network subsystem). In the later case we use a simple network with two hosts A and B and a monitoring host S. Host A reads the trace from 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 was roughly 90%).

We use the `nlanr.MRA.1031627450` 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 [15]. As the trace only contains the header portion of each packet we had to add uniformly random payload data to create realistic traffic. (The use of random payloads for nIDS evaluation is shown in [2] to offer reasonably accurate performance estimates.)

4.1 Evaluation of Early Filtering

Analyzing the trace reveals that more than 40% of the packets do not contain any payload. Most of these packets

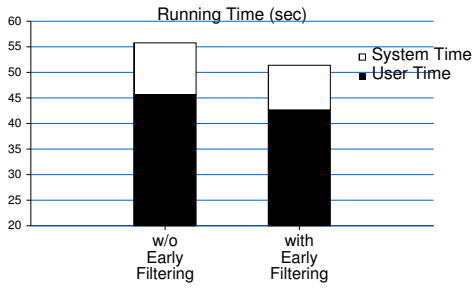

Figure 3. The effect of Early Filtering on sensor performance.

are TCP acknowledgments and none of them is matched by the EF ruleset. These packets can therefore be dropped by the splitter during early filtering instead of forwarding them to a sensor for analysis. Only a small fraction of packets are actually matched by the EF ruleset (less than 1% of total packets) and are therefore forwarded to the sensor.

On each sensor, early filtering is expected to reduce the actual detection workload as well as the burden on the network subsystem for processing interrupts and bringing packets from the network interface to user space for processing.

To measure the effect of early filtering on sensor performance we measure the user and system time of running *snort* with the original trace as well as a trace that does not contain the packets that are dropped by early filtering. The results are depicted in Figure 3. We observe that user time is reduced by 6.6% (45.67 sec vs. 42.66 sec) while system time is decreased by 16.8% (10.1 sec vs. 8.7 sec). Considering both user and system time the results suggest an overall improvement of 8%.

4.2 Performance of Load Sharing Hash Function

The performance of CRC16 as a hash function has been previously studied in [4]. Since our implementation is simplified, we need to confirm that our hashing function is fair enough in terms of evenly distributing load. For this purpose, we measure the the maximum number of packets assigned for the cases of 2, 4 and 8 sensors. Then we compare this number to the theoretical fair share (i.e. total number of packets / number of sensors). We use a large trace with approx. 80 seconds of activity over a 622 Mbit/s link to estimate the performance of the hashing function. As hashing reassembles a pseudo-random way of spreading the packets, a significant amount of traffic is required for the splitting to be fair. Over shorter periods of time the measured imbalance may be more pronounced. However, this is not important as it would be absorbed by the system if sufficient buffering is in place.

Table 2 depicts the maximum difference from the fair

Sensors	difference (in % of assigned packets) of most loaded from fair share
2	1.25%
4	5.70%
8	13.55%

Table 2. Performance of CRC16-based Load Sharing method.

share for different number of sensors. In the worst case of 8 sensors, the most loaded sensor received 13.55% more packets than its theoretical fair share which is acceptable for a simple implementation of the CRC16. In the long run (more than 80 sec of traffic for each sensor) we would expect that imbalance would decrease.

These results are important for understanding the interaction between the load sharing policy and the locality buffering method as presented in the next subsection.

4.3 Effect of LBs on nIDS performance

To investigate the benefit of using locality buffers we measure the total nIDS workload in terms of measured user time on each sensor as well as the workload of the most loaded sensor, given that traffic is not perfectly distributed.

We determine how performance is affected when using different numbers of participating sensors, number and size of locality buffers and different heuristics for locality buffer allocation.

4.3.1 Effect of LB vs. number of sensors

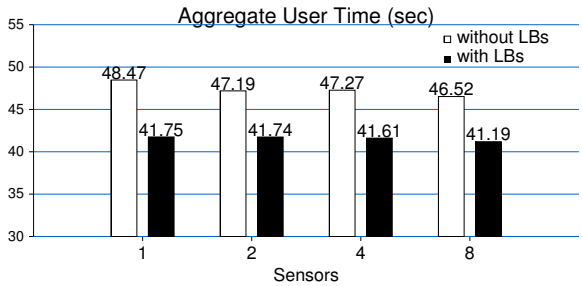

Figure 4. Aggregate user time over all sensors vs. number of sensors.

Figure 4 shows the aggregate user time for different numbers of sensors, and Figure 5 shows the the measured user time of the slowest (most loaded) sensor. For this set of experiments we use 16 LBs of 256 KB each and the D_B

allocation method. In all cases, using locality buffers improves the aggregate user time by at least 11.4% (8 sensors) and up to 13.8% (one sensor).

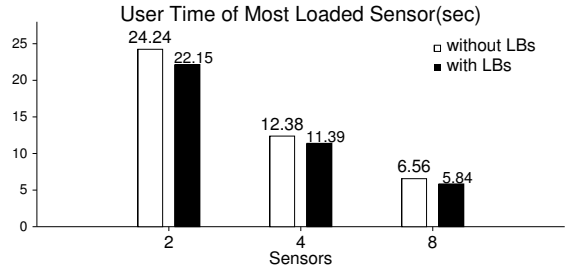

Figure 5. User time of slowest sensor vs. number of sensors for the experiments of Figure 4

One interesting observation from Figure 4 is that as the number of sensors increases, the effect of locality buffering is slightly decreasing. In fact, there is a slight improvement in aggregate user time even if we don't use locality buffering. This happens because distributing packets to different sensors demultiplexes the incoming traffic and increases the probability of same-type back-to-back packets. The positive effect of locality buffering is nevertheless evident.

To verify this observation we measure the average burst size (*e.g.,* the number of consecutive packets that have the same protocol and the same destination port) in the experiments of Figures 4 and 5. The results are presented in Figure 6.

We see that the average burst size is almost doubled when using LBs. We also observe that splitting traffic to more sensors slightly improves the mean burst size when not using LBs. A histogram of burst sizes when using four sensors is shown in Figure 7, showing that a almost half of the one-packet bursts are eliminated with roughly 5% of the packets grouped into large bursts of more than 17 packets.

4.3.2 LB dimensioning

We investigate how the size and the number of locality buffers affect performance. In this set of experiments we use four sensors and the locality buffers are allocated using method D_B . In each experiment we measure the difference in user time compared to a system without LBs.

Figure 8 shows the results of using different number of LBs per sensor when the the size of each LB is 256 KB. We observe that the improvement in aggregate user time varies between 6.8% (4 LBs) and 12.9% (32 and 64 LBs). Increasing the number of LBs beyond 32 does not appear to offer any benefit in terms of aggregate user time although it still improves the performance of the most loaded sensor. This suggests that using 32 or 64 LBs is a reasonable design choice.

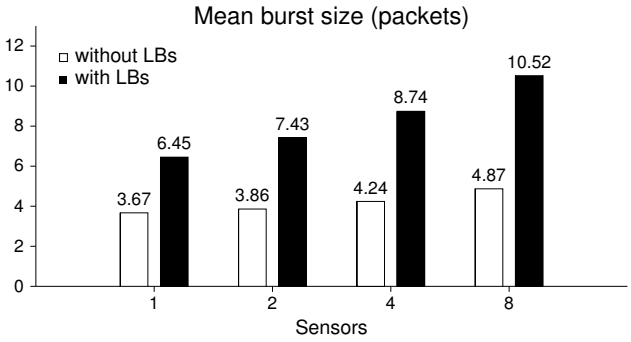

Figure 6. Mean burst size vs number of sensors for the experiment of Figures 4 and 5.

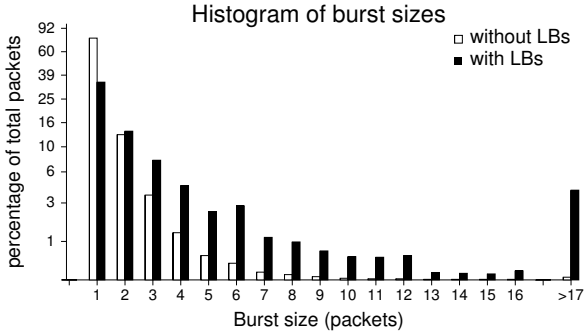

Figure 7. Histogram of burst sizes, with and without locality buffers for the case of 4 sensors of Figure 6.

To measure how the size of each LB affects performance we measure, again, the aggregate user time and the user time of the most loaded sensor for different LB sizes. The results are presented in Figure 9. The reduction in aggregate user time ranges from 9.3% to 13.31% for the cases of 64 KB and 512 KB respectively. Using 256 KB per LB seems like a reasonable choice, as the gain of increasing from 256 KB to 512 KB is marginal.

4.3.3 Effect of different locality buffering policies

We examine how the different heuristics for allocating locality buffers affects performance. For this set of experiments we use four sensors, 16 LBs per sensor and 256 KB per LB. Again, we measure the percentage of reduction in user time compared to not using LBs.

Figure 10 shows the performance improvement in terms of aggregate user time as well as user time of the slowest sensor, for different locality buffer allocation methods. We see that using hashing on the destination port only (D and p_D) is better than simple hashing on both ports (D and p_D). The performance improvement in aggregate user

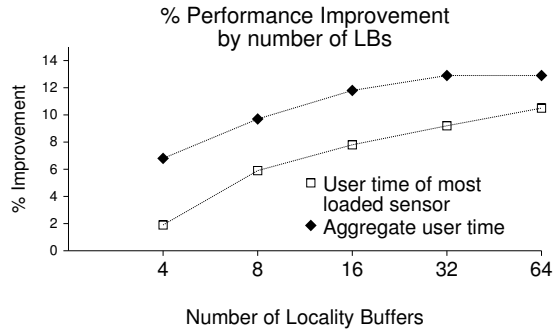

Figure 8. Performance improvement (reduction in user time) using different number of LBs.

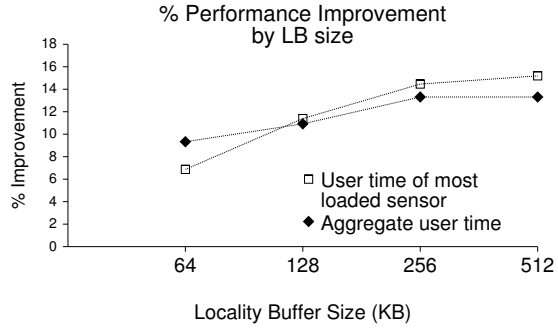

Figure 9. Performance improvement (reduction in user time) using different size for each LB.

time in the first two cases is 4.19% while in the second two cases becomes around 7%. We also see that the best performance is obtained when assigning some LBs to specific traffic. This is observed in bars *T_D* and *p_T_D* which show an improvement of 12.19% and 12.21% respectively. This is not surprising, as a significant part of the trace includes Web traffic, and therefore dedicating LBs to this kind of traffic results in longer bursts of similar packets.

Splitting the LBs by protocol does not appear to be useful: comparing *T_D* with *p_T_D* shows that the later has a smaller effect on the user time of the slowest sensor. A comparison of *D* and *p_D* shows a similar behavior. This can be explained by the fact that TCP traffic constitutes the vast majority of traffic, and therefore splitting the LBs by protocol is ineffective and essentially throws most of the traffic into the TCP LBs.

4.4 Evaluation of EF+LB

To estimate the benefits of using both early filtering and locality buffering together we apply the early filtering method on the packet trace and split the remaining packets to four sensors using 16 Locality Buffers of 256 KB per sensor and buffer allocation heuristic *T_D*. The measured ag-

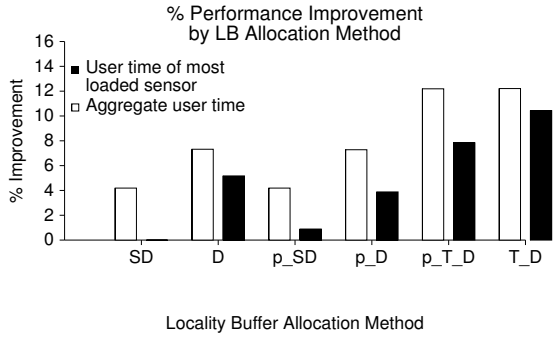

Figure 10. Percentage of performance improvement when using different Locality Buffer Allocation Methods.

gregate user time is 37.88 sec compared to 41.61 sec when using LBs only, reflecting an improvement of 8.9%. Compared to 47.27 sec when not using LBs at all, the overall improvement of using both EF and LB is 19.8%. For the slowest sensor, performance is increased by 5% when compared to using only LBs (from 11.52 sec to 10.93 sec) and 14.4% when compared to not using EF or LB.

5 Discussion

The experiments presented demonstrate the performance gains of early filtering and locality buffering. Although we have not prototyped the system we claim that this can be achieved at a reasonable cost based on the following three observations.

First, the cost of the computations involved is reasonable. Early filtering requires header classification for which efficient algorithms already exist[12, 9], and locality buffering only adds the simple operation of assigning a packet to a buffer based on the policies presented in Section 3 (for example, a simple hash on the destination port).

Second, the amount of buffer memory needed for locality buffers can be much less than the numbers presented in Section 4 if properly implemented. For simplicity, our study has assumed that the whole packet has to be copied from the input buffer to the locality buffer, therefore requiring separate memory for each LB and an extra copy of the whole packet from input to LB. A better implementation would involve a shared buffer for storing the packets and a small amount of memory per locality buffer for storing the packet descriptors instead of whole packets. For example, a configuration involving four sensors, with 32 LBs of 256 KB each, requires 32 MB of memory. Using descriptors (and assuming a minimum packet size of 64 B) each LB will require 4 KB of pointers (instead of 256 KB) and the total required space for all LBs will be 512 KB (instead of 32 MB). The shared buffer can then be dimensioned appropriately and is also likely to require much less memory due to

economies of scale.

Third, the design of hardware components for such a traffic splitter is well studied in the context of other network elements such as switches. For instance, many switches use shared buffers for storing packets and queues to store packet descriptors (*e.g.,* pointers), and packet descriptors are placed on queues after analyzing the header, etc. As there are fast hardware implementations of similar functionality capable of operating at more than 10 Gbit/s[16], we believe that the implementation of the proposed traffic splitter architecture is likely to be straightforward.

6 Summary and future work

We have proposed an active traffic splitter architecture for intrusion detection. Rather than acting as a passive load balancing component, we argue that the traffic splitter should actively manipulate the traffic stream in a way that increases sensor performance.

We have presented and analyzed two specific examples of performance-enhancing mechanisms. The first is *early filtering*, where a subset of the traffic is processed on the traffic splitter and filtered out in order to reduce the load on the sensors. In its most simple form, early filtering increases sensor performance by 8% by filtering out roughly 32% of the packets that are not subject to content matching. The header rules that constitute early filtering are only a small fraction of the nIDS ruleset making it easy to implement on the traffic splitter. The second method is *locality buffering*, where packets classified to the same subset of nIDS rules are buffered together before being forwarded to the sensors. By grouping same-type packets and sending them to the sensor back-to-back, this method increases memory access locality on the nIDS sensors resulting in improved performance. We have examined the effect of different buffering policies and buffer parameters and our results indicate that using 32 locality buffers of 256 KB each and a policy of using dedicated buffers for the major traffic groups results in a 10% reduction in nIDS load. When using both methods together, overall system performance is improved by 19.8%, while the running time of the most loaded sensor is improved by 14.4%.

We are currently investigating the performance of the proposed methods using a more diverse set of traffic traces. Additionally, we are building a prototype nIDS traffic splitter using a high-performance network processor.

Acknowledgments

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union. The second author is also supported in part by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795.

References

[1] K. G. Anagnostakis, S. Antonatos, M. Polychronakis, and E. P. Markatos. E^2xB: A domain-specific string matching algorithm for intrusion detection. In *Proceedings of IFIP International Information Security Conference (SEC'03)*, May 2003.

[2] S. Antonatos, K. G. Anagnostakis, M. Polychronakis, and E. P. Markatos. Benchmarking and design of string matching intrusion detection systems. Technical Report 315, ICS-FORTH, December 2002.

[3] R. Bace and P. Mell. *Intrusion Detection Systems*. National Institute of Standards and Technology (NIST), Special Publication 800-31, 2001.

[4] Z. Cao, Z. Wang, and E. W. Zegura. Performance of hashing-based schemes for internet load balancing. In *Proceedings of IEEE Infocom*, pages 323–341, 2000.

[5] C. J. Coit, S. Staniford, and J. McAlerney. Towards faster pattern matching for intrusion detection, or exceeding the speed of snort. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2002.

[6] G. Dittmann and A. Herkersdorf. Network processor load balancing for high-speed links. In *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 727–735, July 2002.

[7] M. Fisk and G. Varghese. An analysis of fast string matching applied to content-based forwarding and intrusion detection. Technical Report CS2001-0670 (updated version), University of California - San Diego, 2002.

[8] G. H. G. Goldszmidt. Scaling internet services by dynamic allocation of connections. In *Proceedings of the Sixth IFIP/IEEE International Symposium on Intergrated Network Management*, pages 171–184, May 1999.

[9] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of ACM SIGCOMM*, pages 147–160. ACM Press, 1999.

[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 Research on Security and Privacy*, Oakland, CA, May 2002. IEEE Press.

[12] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 203–214. ACM Press, 1998.

[13] E. P. Markatos, M. D. Flouris, D. N. Pnevmatikatos, and M. G. H. Katevenis. Web-conscious storage management for web proxies. Technical Report 275, Institute of Computer Science, Foundation of Research and Technology Hellas, 2000.

[14] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Laboratory, Berkeley, CA, available via anonymous ftp to *ftp.ee.lbl.gov*.

[15] NLANR. MRA traffic archive, September 2002. *http://pma.nlanr.net/PMA/Sites/MRA.html*.

[16] C. Partridge. et al. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking (TON)*, 6(3):237–248, 1998.

[17] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[18] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999. (available from *http://www.snort.org/*).

[19] K. Ross. Hash routing for collections of shared web caches. *IEEE Network*, 11(6), November-December 1997.

[20] 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.

[21] TopLayer. IDS load balancer. product description available through *http://www.toplayer.com/*.