Ourmon versus IXIA: Monitoring Gigabit Speeds

James R. Binkley, Bart Massey
Computer Science Department
Portland State University
Portland, OR USA
Paper ID: E00-241341058
Pages: 8
{*jrb,bart*}*@cs.pdx.edu*

Abstract—**We measure the performance of a modern PC host running *Ourmon*, our novel network monitoring system. The goal of the analysis is to understand system performance under gigabit load with both maximum-sized and minimum-sized Ethernet frames. Our testbed consists of an IXIA 1600 high-speed packet generator that can send packets at near theoretical speeds, a gigabit switch, and a roughly 2 GHz AMD PC host. We measure the performance of the underlying BPF network tap on FreeBSD, as well as the performance of various application-layer filters used by the Ourmon system. We show that using a BSD BPF kernel buffer size of several megabytes permits a workstation using the Ourmon filters to capture back-to-back maximum-sized Ethernet packets at gigabit speeds. We also show that when minimum-sized packets are sent, the BPF/Ourmon system fails at a relative small throughput rate. This failure appears to be due to limitations of the hardware and/or network stack rather than the Ourmon system proper. This result has troubling implications for network management and intrusion detection systems, given the current frequency of large volume Internet attacks using small packets.**

Keywords—**Intrusion detection, Ourmon, IXIA, network monitoring, network security.**

I. Introduction

The Internet has recently faced an increasing number of bandwidth-intensive denial-of-service (DOS) attacks. For example, in January 2003 the Slammer worm [1], [2] caused serious disruption. Slammer not only wasted bandwidth and affected reachability, but also seriously impacted the core routing infrastructure. At Portland State, four lab servers with 100 Mb NIC cards were infected simultaneously. These servers then sent approximately 360 Mb of small packets to random destinations outside of PSU. This attack clogged PSU's external connection to the Internet and also caused important network monitoring failures. Due to the semi-random nature of the IP destination addresses generated by the worm, the CPU utilization of a router sitting between network engineers and network instrumentation rose to 100%. Engineers were thus cut off from central network instrumentation at the start of the attack.

We recently acquired an IXIA 1600 high-speed packet generator. The Slammer attack inspired us to test our Ourmon [3] network monitor system against a set of Gigabit Ethernet (GigE) flows. Our test flows included maximum-sized (1518 byte) and minimum-sized (64 byte) UDP packets as well as rolling IP destination addresses.

The Ourmon network measurement system architecture consists of two parts: a front-end probe and a back-end graphic display system. Optimally, these two parts should run on two separate computers in order to minimize the application compute load on the probe itself. Our goal in these experiments has been to test the performance of the front-end Ourmon probe and its BPF network tap, rather than the back-end system.

We constructed a test system consisting of the IXIA with two GigE ports, a line speed GigE switch capable of port-mirroring, and a UNIX workstation with a GigE NIC card. The IXIA was set up to send packets from one GigE send port to the other port. The switch was set up to mirror packets from one IXIA port to the UNIX host running our front-end probe.

Like other tools including tcpdump [4], Snort[5], or Ntop [6], [7], the Ourmon front-end uses the BPF as a *packet tap*. The application takes a stream of unfiltered packets directly from an Ethernet device, bypassing the host TCP/IP stack. The interface interrupts on packet input, and hands the trimmed packet (containing all headers through layer 4) to the kernel BPF filter buffer. The front-end Ourmon probe reads packets, subjecting each packet in turn to a set of configuration filters. It thus makes sense to separately test the performance of BPF and the performance of the Ourmon probe component filters.

Our experimental questions include the following:

1. Using GigE with maximum or minimum-sized packets, at what bit rate can the underlying BPF tap and buffer system successfully process all packets?
2. Using GigE with maximum or minimum-sized packets, what is the smallest BPF kernel buffer size (if any) for

which all packets are successfully processed?
3. Ourmon has three kinds of filters: hardwired C filters, BPF-based interpreted filters, and a "top-N" flow analysis system. Can we determine anything about the relative performance of these filters? If we are receiving a high number of packets per second, which of these filters can keep up?
4. With the Slammer worm, we know that semi-random IP destinations led to inefficient route caching in intermediary routers. What happens when we subject our top-N flow monitor to rolling or semi-random IP destinations?

In section II we provide a brief introduction to the Ourmon system. In section III we discuss our test setup. In section IV we present test results, and in section V we provide analysis and conclusions.

II. Introduction to Ourmon

In this section, we explain enough of the Ourmon system functionality to understand the performance measurement work reported in sections III and IV. In furtherance of this goal, we focus on the front-end packet capture system. The detailed workings of the Ourmon system are outside the scope of this paper. [1]

Ourmon is a real-time web-based network monitor. Ourmon is somewhat similar to SNMP RMON [8] systems or Ntop. Unlike SNMP RMON, Ourmon does not present a set of variables encapsulated in a probe accessed via the SNMP protocol from a management system. Instead, Ourmon assumes the port-mirroring functionality of Ethernet-based switches.

A typical setup may be seen in Figure 1. An Ethernet switch is configured to mirror (duplicate) packets sent to its Internet connection on port 1. All packets received via the Internet port are copied to port 3, which is running the front-end Ourmon probe on a FreeBSD system with the BPF packet tap. Thus Ourmon's probe setup is similar to that of Snort, which we show running on port 2 of the switch. The probe system ideally is attached to a border switch that sees all packets going to and from the Internet. The back-end Ourmon graphics engine is not performance critical: it may run on a second computer, which need not be in the center of a network.

The probe has an input configuration file and an output file. The input file, called *ourmon.conf*, specifies various named filters for the probe to use. Probe output is written to a small ASCII file called *mon.lite* that summarizes the last 30 seconds of filter activity in terms of byte or packet counts per named configuration filter. This file may be

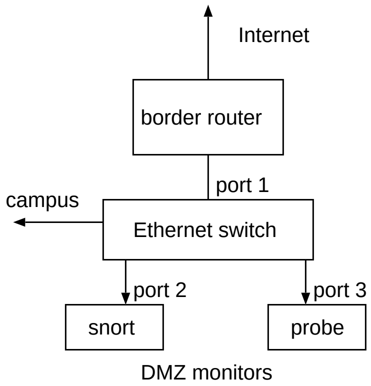

Fig. 1. Ourmon network setup

copied over the network to the graphics-engine computer, which in turn produces various graphic outputs and ASCII reports for web display. The back-end graphics engines produces several kinds of graphics. RRDTOOL-based [9] strip charts are used with BPF filter-sets and hardwired filters. Histograms are used to display the top-N flow analysis.

The Ourmon front-end process uses the BPF in two ways. It uses it to get packets from the kernel BPF buffer system. It also uses the BPF interpreter in user mode. The system can utilize multiple BPF filter expressions, group them together in a BPF *filter-set* and graph each expression in the set as a separate line in a single RRDTOOL strip chart graph.

As an example, here is a simplified configuration for one BPF filter-set. This filter-set groups the performance of five application services together, using one BPF expression each for secure-shell, combined P2P protocols, web, ftp, and email.

```
bpf "ports" "ssh" "tcp port 22"
bpf-next "p2p" "tcp port 1241 or
     tcp port 6881"
bpf-next "web" "tcp port 80 or
     tcp port 443"
bpf-next "ftp" "tcp port 20 or
     tcp port 21"
bpf-next "email" "tcp port 25"
```

Probe output for such a filter over a snapshot period might look like this:

```
bpf:ports:5:ssh:254153:p2p:19371519:
 web:41028782:ftp:32941:email:1157835
```

The filter configuration allows the user to name the composite graph "ports". Five separate user-mode BPF configuration expressions like "tcp port 22" are mapped to appropriate line labels ("ssh") in the same graph. Thus ssh/p2p/web/ftp/email byte counts will all appear in the same RRDTOOL graph as in Figure 2. The probe executes

[1] However for more information on the Ourmon architecture, please see: *http://ourmon.cat.pdx.edu/ourmon/info.html* .

the user-mode BPF runtime expressions on the incoming packet stream from the packet tap and counts matching bytes. At the sample period timeout, it outputs the *mon.lite* file which in this case includes the name of the filter-set and various (line label, byte count) tuples for each BPF expression. Multiple BPF filter-sets are possible: many separate BPF expressions can be executed in the probe application. The current PSU DMZ probe software is running around 60 BPF expressions in sixteen filter-sets and does not normally lose packets (barring attack periods).

Ourmon also supports a small set of "hardwired" filters programmed in C and turned on via special names in the configuration file. For example, a hardwired filter counts packets according to layer 2 unicast, multicast, or broadcast destination address types. One very important filter called the *packet capture* filter includes statistics on dropped and counted packets provided directly from the BPF kernel code. The packet capture filter is fundamental: it is used to determine when the kernel BPF mechanism is overloaded in our testing. Typical front-end output in the *mon.lite* file for that filter and the layer 2 packet address type filter might look like this:

```
pkts: caught 53420 drops: 0
fixed_cast: mcast: 2337215:
 unicast: 15691896: bcast: 0:
```

The *packet capture* filter ("pkts") output means that the BPF during the last sample period caught 53240 packets and dropped none. In Figure 3, we show an example back-end graph for this filter. Drops are in green and captured packets are in blue. This graph is from the day of a Slammer re-infection. It can be seen that the Ourmon probe (at the time a Pentium-3) has caught the attack, even though many packets have been dropped.

The third and last filter class in Ourmon is a "top-N" flow monitor. The front-end builds up a hash-sorted list of IP flows over the sample period and writes the top N (typically around 10) TCP, UDP, and IP (all IP protocols) flows to the output file. The back-end takes this information and produces graphical histograms and text reports. A flow is defined as a five-tuple consisting of IP source, IP destination, IP next protocol, L4 source port, and L4 destination port. See Figure 4 for an example top-N report: we show a DOS attack on a local IT administrator's host machine. The attack packets are launched over Internet2 using a spoofed IP source address. Multiple UDP flows, each around 1.5 Mb, are shown.

In summary, the front-end has three kinds of filters: hardwired C filters, user-mode BPF filter-sets, and top-N flow analysis. Intuitively we are interested in the execution cost of each of these three kinds of filters. While there is only one top-N flow filter, a user may program any number of BPF filter-sets: this complicated the analysis somewhat. The *packet capture* filter is especially important, as it serves to tell us when we are losing packets. We can view this as an important indicator that the combined kernel and probe application system is in failure mode. An important cause of failure is too much work done at the application layer, causing the application to fail to read buffered kernel packets in a timely manner.

III. Experimental Setup

The hardware used in our testing consists of three pieces of equipment:

1. An IXIA 1600 chassis-based packet generator with a two port GigE line card. One port sends packets and the other port receives packets.
2. A Packet Engines line speed GigE switch. Three ports on the switch are used: one for the IXIA send port, one for the IXIA receive port, and a third port connected to the UNIX host for mirroring the IXIA flow.
3. A 1.7 GHz AMD 2000 computer system. (The AMD processor is roughly comparable to a 2GHz Intel Pentium 4 processor.) The system motherboard is a Tyan Tiger MPX S2466N-4M. The motherboard has two 64-bit PCI slots: we use a SysKonnect SK-9843 SX GigE card in one of the slots.

Software used includes Ourmon 2.0 and the BSD libpcap [4] library 0.7.2. The host operating system is FreeBSD 4.7, running only the Ourmon front-end probe.

We set up the IXIA to send either minimum-sized packets or maximum-sized Ethernet packets. One port on the IXIA sent packets through the switch to the other IXIA port. All packets were UDP packets. The IXIA allows the user to select an arbitrary packet sending rate up to the maximum possible rate. It can also auto-increment IP destination addresses: we used this as an additional test against the top-N filter.

According to Peterson [10] the maximum and minimum theoretical packet rates for GigE are as shown in Table I. We used these values as a measurement baseline. We observed that the IXIA 1600 can indeed generate packets at nearly 100% of this rate for both maximum and minimum-sized packets. We used these numbers to make sure that our Ethernet switch did not drop packets: we hooked both IXIA GigE ports up directly to the switch and sent packets from one IXIA port to another IXIA. The IXIA's built-in counters at the receive port reported the same packet counts as at the send port.

The test methodology involves setting up a UNIX host with a driver script and some set of Ourmon filters. The front-end probe is started, and the IXIA is configured to send min or max packets at some fraction of the maximal rate. Ourmon is configured with some combination

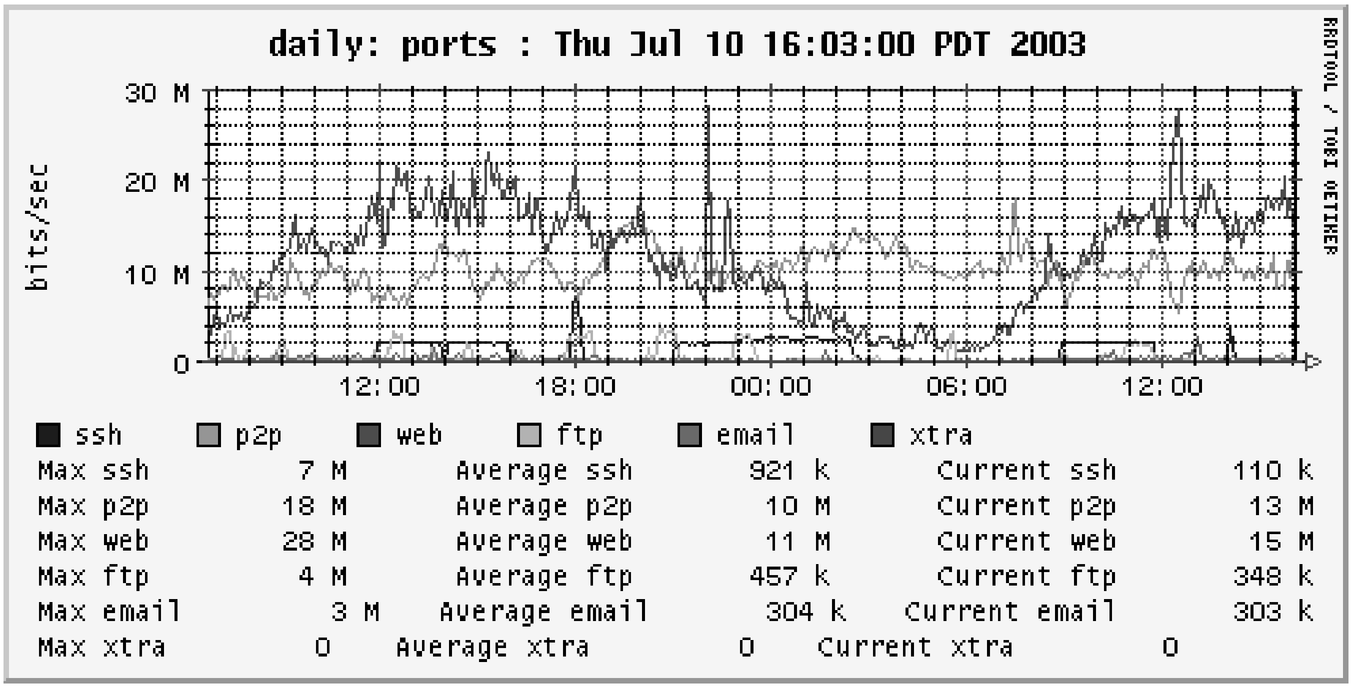

Fig. 2. A BPF filter-set graph showing application byte counts

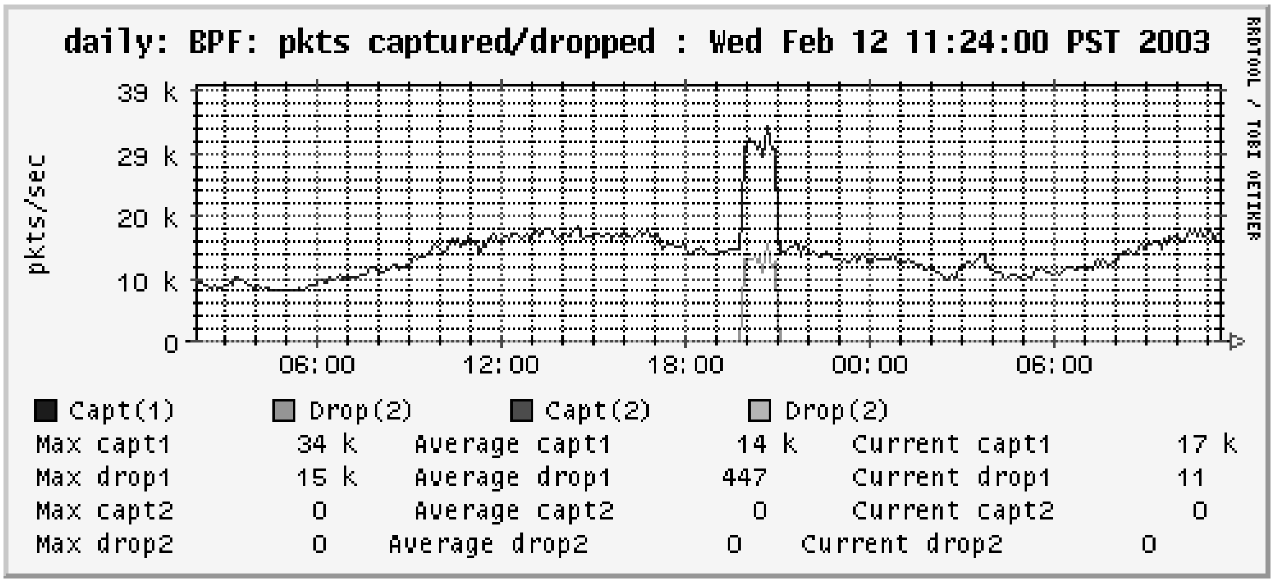

Fig. 3. The packet capture filter graph showing counts and drops during a Slammer attack

TABLE I
GIGE RATES

	bytes/pkt	pkts/s
min	64	1488000
max	1518	81300

of hardwired, user-mode BPF, and top-N filters as desired. The test flows are then started on the IXIA, and the results observed using the *mon.lite* output file.

The test script is the Bourne Shell script shown in Figure 5. The FreeBSD *sysctl(8)* command is used to set the kernel BPF buffer size. This is because recent versions of the PCAP library on FreeBSD will take this information and automatically size the buffer to be used by the client application to match the kernel buffer size. This is a relatively new feature and was very handy for our experiment. It should be noted that the traditional size of the kernel BPF buffer is typically small (4KB in FreeBSD 4.9) and is intended for the tcpdump sniffer application. The parameters to the Ourmon probe program tell it to take input from

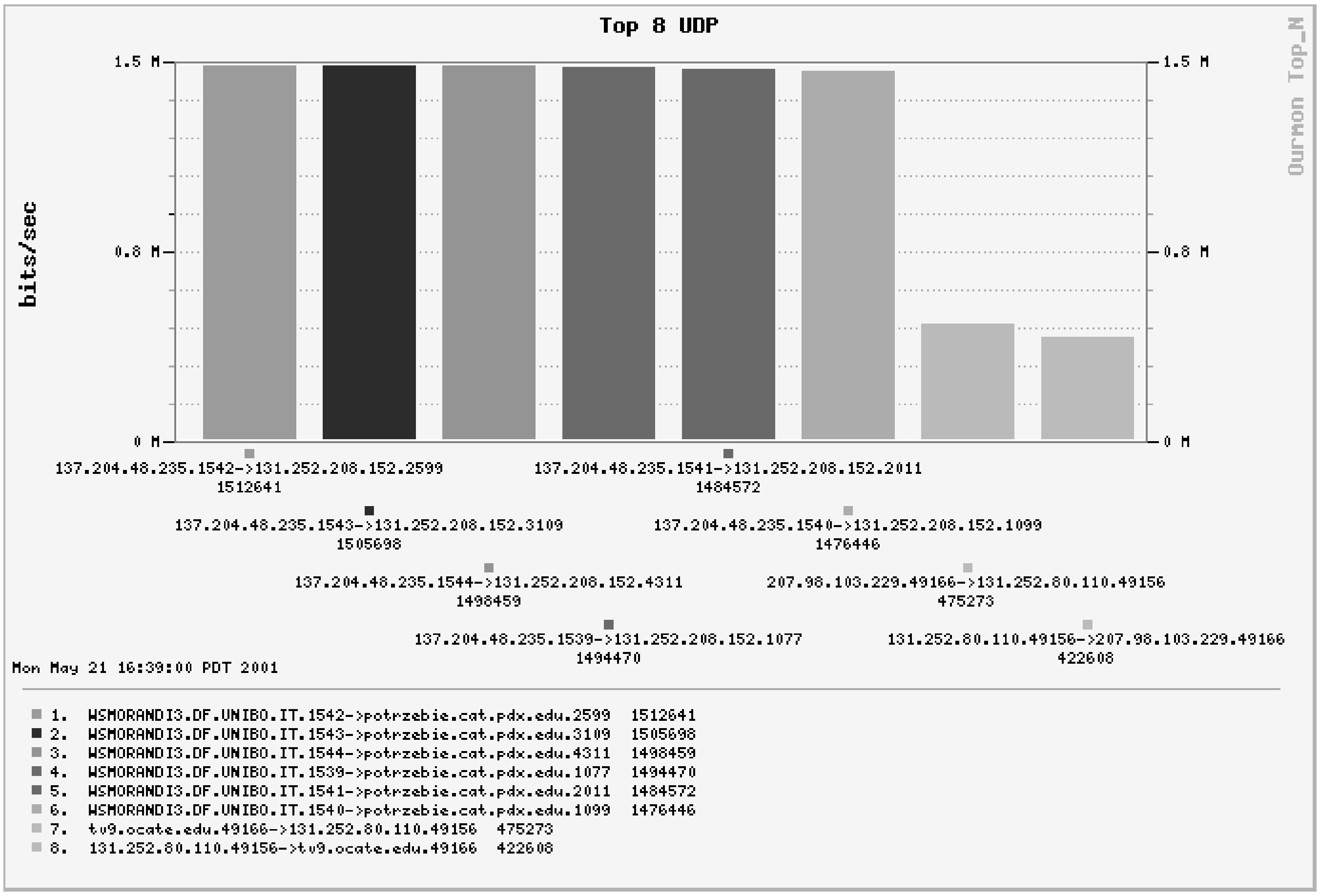

Fig. 4. Top-N UDP flow histogram showing a DOS attack

```
#!/bin/sh
BSIZE=1048576
sysctl -w debug.bpf_bufsize=$BSIZE
sysctl -w debug.bpf_maxbufsize=$BSIZE
./ourmon -a 5 -I sk0 -m /dev/tty \
  -f ./ourmon.conf
```

Fig. 5. Test script

a local configuration file, dump the output information to the screen every five seconds and use the SysKonnect card as the input interface.

Tests were run using either maximum-sized or minimum-sized packets. If we dropped packets, we attempted in every case to eliminate packet drops by increasing the kernel BPF buffer size (BSIZE above). If that failed, we then reduced the IXIA's send rate until all packets were transmitted.

For testing, we identified five interesting categories of Ourmon filters and constructed filter tests for these categories.

null: The packet capture filter cannot be turned off, and is the only remaining statistic when all Ourmon filters (hardwired, etc.) are removed from the configuration.

hard: The hardwired C filters as a group.

bpf: BPF filters as one or more filter-sets.

top-n: The top-N filter mechanism.

combo: A simple combination of all filters.

The null filter tells us whether or not the BPF in the kernel was losing packets, as its count/drop information is taken from the operating system. The hard, bpf, and top-n filter categories were tested individually in order to determine if the filter type itself had an impact on the overall performance. The six hardwired C filters available at the time of testing were used in the tests. The bpf tests were based on a filter-set that had 4 simple filters in it. The individual BPF expressions were configured to capture TCP ports that could not match the output of the IXIA (UDP packets): it seemed reasonable for BPF expressions to always fail to match.

Repeatedly testing the top-N mechanism with the same IP flow would yield no new information. Therefore, for the top-N test we used a rolling IP destination setup where each subsequent UDP packet within a set of 1000 or 10000 had a different IP destination. This could be said to be a rough simulation of the Slammer worm, with its variation

TABLE II
MAXIMUM PACKET TESTS

test	sets	flows	bufsz (KB)	drops (%)
null			128	0
hard			128	0
top-n		1000	128	0
top-n		10000	—	80
bpf	1		128	0
bpf	4		128	0
bpf	8		128	20
bpf	8		7168	0
combo	1	1000	7168	0

in IP destinations.

IV. TEST RESULTS

Test results fall into two basic categories, which are reported separately: tests with maximum-sized packets, and tests with minimum-sized packets.

A. *Maximum Packets*

In this set of tests, packets were always 1518 bytes, the normal maximum MTU for Ethernet packets. (This works out to a 986 Mb flow of UDP packets). Our tests involved the null, hard, top-n, bpf, and combo categories as described in section III. The test results are summarized in Table II.

The flow rate was set to maximum: the drop rate therefore shows packets lost at gigabit speeds. In the null case, the configuration *almost* worked with the default BPF buffer size of 4 KB. However, some packets were lost at a 30 second interval. This may have had something to do with an operating system timer. Increasing the kernel BPF buffer size to 128 KB resulted in perfect transmission, even after adding in the hardwired filters.

The top-N filter worked with no loss at 1000 flows and completely failed at 10000 flows. Larger BPF buffers did not help (dashed table entry). This is the most significant failure case with maximum-sized packets. Decreasing the IXIA flow-rate to 45 Mb resulted in perfect transmission. For the bpf tests, we increased the number of filters to 8 sets (32 BPF expressions) before running into some loss. At that point, we increased the kernel BPF buffer size. We found that a very large buffer of 7 MB could indeed get us back to lossless transmission. With the combo configuration (hard + top-n + 1 bpf set) we did not experience any loss: note however that we used only 1000 flows with the top-N filter.

TABLE III
MINIMUM PACKETS AND NULL FILTER

bufsz (KB)	thresh (Mb/s)
32	53.33
128	68.52
256	76.19
512	76.19

TABLE IV
HARDWIRED AND BPF TESTS

test	sets	rate (Mb/s)	drops (%)
hard		76	0
bpf	1	68	0
bpf	2	53	0

B. *Minimum Packets*

Attempts to capture maximum-rate flows of minimum-sized packets (64 bytes) uncovered serious problems. We therefore report our results as a series of small experiments: each experiment focuses on a different test domain.

B.1 Null Filter Only

It was not always possible to capture all packets even in the null filter case. Instead we attempted to determine the effect of the kernel BPF buffer size on drop rates as shown in table III.

A buffer size of 256 KB appears optimal: at this size the system begins to drop packets at 76 Mb. Larger kernel buffers do not improve the result. Of course the most important aspect of this test is that we cannot capture more than around 10% of the GigE stream without loss. (Note that packet overhead for minimum packets results in a maximum flow of around 760 Mb.)

B.2 Individual Filter Types

Having determined baseline drop rates using the null filter, we could now proceed to measure the impact of other filter types. In the BPF filter-set tests, we tried both one and two filter-set configurations. In the top-N filter test, we varied the number of simultaneous flows. Table IV below shows the results for the hard and BPF tests. Table V shows the results for the top-N tests.

Hardwired filters appear to have no impact on performance. The bpf filters have some performance impact, visible even at a modest 76 Mb transfer rate. At this transfer rate, 1000 unique flows is stressful for the top-n filter. However reducing the flow rate to 45 Mb allows the filter to keep up with the data. Unfortunately, 10,000 flows can-

TABLE V
MINIMUM PACKETS—TOP-N TESTS

flows	thresh (Mb/s)	bufsz (KB)	rate (Mb/s)
1	0	256	76
100	1	256	76
1000	25	256	76
1000	0	256	45
10000	50	—	—

TABLE VI
MINIMUM PACKETS—ALL FILTER TYPES

rate (Mb/s)	drops (%)
76	44
68	37
53	18
45	3
38	0

not be handled with any kernel buffer size at any measured transfer rate. Subsequent work on hashing techniques has improved upon this result, but the top-n filter is still quite expensive.

B.3 Combination filtering

In this experiment we measure the combo filtering previously discussed. Here we vary only the flow-rate, holding the buffer size constant at 256 KB and the number of flows constant at 1000. Table VI shows the results.

We see that we must reduce the flow-rate to roughly one-half maximum in order to prevent drops. This is probably because of the impact of 1000 flows on the top-N filter. The filters here are in truth fairly minimal, as there is only one BPF filter-set. In reality one would want more filter-sets to get better traffic information. The bottom line is that we must reduce the flow-rate to 38 Mb for even a modest amount of work to be performed without packet loss.

V. CONCLUSION

Our paper introduces Ourmon, a new monitoring tool, and discusses experiments aimed at measuring the performance of both the underlying kernel BPF filter system and Ourmon front-end filter systems.

The measurement system closest to Ourmon is probably Ntop [6]. Ntop is a single program intended to run on desktops: it can be thought of as a network version of the UNIX top program. Ourmon is designed more along the lines of a traditional distributed SNMP probe, with a distinct separation between capture and display. It relies heavily on user-programmable BPF filters and RRDTOOL-based graphics. In this paper, our focus is chiefly on security impacts of measurement. Thus, the differences between Ourmon and Ntop are not as germane.

From the intrusion detection point of view, Ourmon and Ntop can be said to be similar lightweight tools that show anomalous behavior via graphs. In contrast, an IDS tool like Snort does signature-based analysis on every packet. Ourmon is lightweight compared to Snort: it looks only at the layer 1–4 network headers and entirely ignores the data payload. It is thus reasonable to expect that Snort's processing will be impacted even more than Ourmon's by high packet loads.

We can draw some interesting conclusions from our experimental work.

1. The default FreeBSD BPF buffer size of 4 KB is inadequate for a network monitoring system. We suggest a larger default buffer of at least 256 KB: this size should not unduly burden modern systems. Network administrators should understand that a multi-megabyte buffer may be needed. As a point of reference, the current Ourmon probe deployed in the PSU DMZ is running on an 2 GHz Intel Pentium 4 with an 8 MB kernel buffer. It runs 60 BPF expressions and multiple kinds of top-N filters, yet only drops packets during severe TCP SYN attacks.
2. Our BPF filters seem to have a kernel buffer cost associated with them. Our results suggest that there is a positive relationship between the amount of kernel buffer space needed to mask filter latency and the number of BPFs used in our application. Our tests seem to imply that the BPF mechanism is less costly than the top-N filter. However the BPF mechanism can have any number of expressions, and the expressions themselves can vary in complexity. It is thus hard to compare the BPF filter mechanism to the top-N filter mechanism in terms of compute power.
The real computation problem for the top-N system is that it is driven to extremis under attack attempting to cope with random IP addresses (source or destination). The hash-based top-N algorithm will first search for the given flow ID, and then perform an insert if it fails to find the flow. Consequently random flows always cause an insert. This leads to an interesting research question: How can we deal with boundary conditions caused by random IP addresses without unduly impacting efficiency mechanisms meant for normal bursty flows?
3. Our 2 GHz Pentium-4 class computer cannot capture more than 10% of the minimum-sized packet flow. Worse, if the computer is expected to perform actual application-level work using the data, the number of packets we capture without loss falls below 5%.

This last item deserves extended discussion. Consider an IDS system such as Snort. An IDS wants to run an arbitrary number of signatures over both the packet headers and the packet data, and may choose to inject its measurement results into a database system. Clearly per-packet processing times may become quite large in this scenario.

Now consider the security principle known as *weakest link*. For example, Bruce Schneier writes [11]: "Security is a chain. It's only as secure as the weakest link." An IDS system incurs a significant risk when it drops a single packet, as that packet may be the one with the Slammer worm that will infect an internal host. Worse, a set of coordinated systems might launch a distributed DOS attack against an IDS monitor, first blinding it with small packets and then sneaking a one-packet worm payload past it. Packet capture for small packets at high rates is an important open security problem.

There is some related work in the area of capturing small packets. For example, Mogul and Ramakrishnan [12] describe the phenomenon seen here as *receive livelock*. They present improved operating system scheduling algorithms that can lead to fair event scheduling, with the result that receive interrupts cannot freeze out all other operating system events.

One must consider that there is not a lot of time left with 1,488,000 small packets per second: this works out to approximately 0.7 μs per packet. Some sophisticated approach, such as improving the individual compute performance of various filter mechanisms or applying parallelism, is needed to attain adequate performance,

Some recent work on enhancements to the BPF suggests alternatives for improving BPF performance. The xPF system [13] expands the BPF to a general purpose computing machine by allowing backward branches. This provides the opportunity to enhance BPF performance by running filters entirely in-kernel. The BPF+ system [14] optimizes BPF performance using both machine-code compilation and various optimization techniques. This results in impressive performance improvements that we would like to exploit in Ourmon. A recent IDS [15] contains an interesting parallel hardware engine based on a flow slicing technique: this hardware reportedly improves Snort's performance under high packet loads. However making such a system that effectively uses parallelism and remains cost effective is a challenge.

As a result of our testing, we have adopted the long-term objective of trying to produce a parallelized Ourmon system. We intend to explore the porting to and parallelization of Ourmon on the high speed parallel Intel IXP [16] system.

VI. ACKNOWLEDGEMENTS

This work would not have been possible without the generous donation from the IXIA Corporation of an IXIA 1600 tester. We also wish to thank our graduate students Subha Singaram and Gayathri Nagabhushan for their help with this project.

REFERENCES

[1] "CERT Advisory CA-2003-04 MS-SQL Server Worm," Nov. 2003, URL `http://www.cert.org/advisories/CA-2003-04.html` accessed 03 May 2004.

[2] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "The spread of the Sapphire/Slammer worm," 2003, URL `http://www.cs.berkeley.edu/~nweaver/sapphire` accessed 03 May 2004.

[3] J. Binkley, "Ourmon web page," URL `http://ourmon.cat.pdx.edu/ourmon` accessed 05 May 2004.

[4] "Tcpdump/libpcap home page," URL `http://www.tcpdump.org` accessed 03 May 2004.

[5] M. Roesch, "Snort—Lightweight intrusion detection for networks," in *Proceedings of the USENIX LISA '99 Conference*, Nov. 1999.

[6] L. Deri and S. Suin, "Practical network security: Experiences with ntop," *IEEE Communications Magazine*, May 2000.

[7] "Ntop distribution page," URL `http://sourceforge.net/projects/ntop` accessed 03 May 2004.

[8] S. Waldbusser, "Rfc 2021: Remote network monitoring management information base version 2," Jan. 1997.

[9] "Rrdtool web page," URL `http://people.ee.ethz.ch/~oetiker/webtools/rrdtool` accessed 03 May 2004.

[10] S. Karlin and L. Peterson, "Maximum packet rates for full-duplex Ethernet," Tech. Rep. TR-645-02, Department of Computer Science, Princeton University, Feb. 2002.

[11] B. Schneier, *Secrets and Lies*, Wiley Computer Publishing, 2000.

[12] J.C. Mogul and K.K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, Aug. 1997.

[13] S. Ioannidis, K. Anagnostakis, J. Ioannidis, and A. D. Keromytis, "xPF: Packet filtering for low-cost network monitoring," in *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (MPSR)*, May 2002.

[14] A. Begel, S. McCanne, and S. Graham, "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture," in *Proceedings of ACM SIGCOMM*, Sept. 1999.

[15] C. Kruegel, F. Valeur, G Vignka, and R. Kemmerer, "Stateful intrusion detection in high-speed networks," in *Proceedings IEEE Symposium Security and Privacy*, 2002.

[16] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Worich, and H. Wilkinson, "The next generation of Intel IXP network processors," *Intel Technology Journal*, Aug. 2002.