Ourmon versus IXIA: Monitoring Gigabit Speeds

James R. Binkley
Computer Science
Portland State University
Technical Report TR-04-01, March 30 2004
jrb@cs.pdx.edu

March 11, 2004

Abstract

In this paper our goal is to analyze the performance of our network monitoring system called *Ourmon* and its underlying kernel delivery system the Berkeley Packet Filter (BPF) to see how well they perform using Gigabit Ethernet. We measure the performance of the underlying BPF network tap on FreeBSD, as well as the performance of application-layer filters used by our Ourmon system. Our test system consists of an IXIA 1600 high-speed packet generator which can send packets at near theoretical speeds. We show that if the BPF kernel buffer size is megabytes in size, a workstation using the Ourmon filters can avoid dropped packets at Gigabit speeds. We also show that when minimum-sized Ethernet packets are sent, the BPF/Ourmon system utterly fails at a relative small throughput rate.

1 Introduction

The Internet has recently faced an increasing number of bandwidth-intensive denial of service attacks. For example, in January 2003, the slammer worm [3, 11] caused serious disruption, not only wasting bandwidth and affecting reachability, but also demonstrating some serious side effects in core routing infrastructure. At Portland State, four lab servers with 100 megabit NIC cards were infected simultaneously. They then sent approximately 360 megabits of small packets to random destinations outside of PSU. This attack not only clogged PSU's external connection to the Internet, but it also caused serious network monitoring failures as well. 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%. Thus engineers were cut off from central network instrumentation at the start of the attack.

As a result of the slammer attack, and also due to our fortunate acquisition of an IXIA 1600 packet generator, we decided to test our Ourmon [13] network monitor system with a series of Gigabit Ethernet flows. Our tests included maximum-sized (1518 byte) and minimum-sized (64 byte) UDP packets, and flows with rolling IP destination addresses.

The Ourmon network measurement system architecturally has two parts: a front-end probe, and a back-end graphics 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 here was only to test the performance of the front-end Ourmon probe and its BPF "network tap", so we did not test the back-end graphics system.

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

Like other tools including tcpdump [17], snort[14], or ntop [4, 12], the Ourmon front-end uses the BPF as a "packet tap". This means that 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 places the trimmed packet (trimmed to give all headers through layer 4), in the kernel BPF filter. The front-end Ourmon probe then reads packets subjecting each packet in turn to a set of configuration filters. Hence it makes sense to test both the BPF performance by itself, and Ourmon probe component filters in turn.

Our experimental questions include the following:

- 1. Using Gigabit Ethernet, with maximum-sized packets, or minimum-sized packets at what bitrate can the underlying BPF tap and buffer system, not lose packets?

- 2. Given minimum and maximum packet sizes, if we encounter drops, can we increase the kernel BPF buffer size, and not drop packets?

- 3. Our Ourmon tool has three kinds of filters, simple hardwired C filters, multiple BPF-based interpreted expressions, and a top N flow analysis system. Can we determine anything about the relative performance of these filters? Put another way, if we are receiving a high number of packets per second, can we get useful work done with these filters?

- 4. With the slammer worm, we know that semi-random IP destinations made router route caching algorithms inefficient. Thus what happens when we subject our top N flow monitor to rolling IP destinations?

In section 2 we provide a brief introduction to the Ourmon system. In section 3 we discuss our test setup. In section 4 we present test results, and in section 5 we provide an analysis and conclusion.

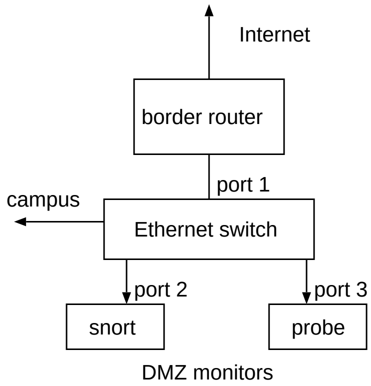

Figure 1: Ourmon network setup

2 Introduction to Ourmon

Although our goal in this paper is not to explain how the Ourmon system works in detail, it is still necessary that we explain enough of the Ourmon system functionality so that the reader will be able to appreciate the performance measurement work itself. As a result, we give a brief overview focused on the front-end packet capture system.

Ourmon is a real-time web-based network monitor somewhat similar to SNMP RMON [18] systems or ntop. Unlike SNMP RMON, it 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 in/out its Internet connection (port 1). All packets sent 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 Snort which we show running on port 2 of the switch. The probe system ideally sees all packets going to and from the Internet. The back-end Ourmon graphics engine 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 represents a summarization of the last 30 seconds of filter activity in terms of byte or packet counts per named configuration filter. This file may be copied over the network where it acts as an input to the graphics-engine computer which in turn produces various graphics outputs and ASCII reports for web display. The back-end graphics engines produces two kinds of graphics, including RRDTOOL-based [15] stripcharts, used with BPF *filter-sets* and hardwired filters, and histograms 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, where the configuration can take 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* that groups the performance of five application services together with one BPF expression per service, including secure-shell, various P2P protocols, web, ftp, and email.

```
bpf "ports" "ssh" "port 22"
bpf-next "p2p" "port 1241 or port 6881"
bpf-next "web" "port 80 or port 443"
bpf-next "ftp" "port 20 or port 21"
bpf-next "email" "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 first name the composite graph "ports", and then to map user-mode BPF configuration expressions like "port 22" 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 the five 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. It should be pointed out that multiple BPF *filter-sets* are possible, and thus 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.*

Ourmon also supports a small set of "hardwired" (programmed in C) filters in the probe that are turned on via configuration names in the configuration file. As one example, we have a hardwired filter that 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 as it was used to determine when the kernel BPF mechanism was overloaded in our testing. Typical front-end output in the *mon.lite* file for that filter and the layer 2 packet address type filter could 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 picture was taken on the day of a slammer reinfection. It can be seen that the Ourmon probe (at the time a Pentium-3) successfully caught the attack, even though many packets were dropped.

The third and last filter class in Ourmon is a "top N" flow monitor. The front-end builds up a hashed sorted list of IP flows over the sample time and writes the top N (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 the following 5-tuple: (IP source, IP destination, IP next protocol, L4 source port, L4 destination port). See Figure 4 for an example, where we show a denial of service attack with a spoofed IP source address launched over Internet2 at a local IT administrator's host machine. Multiple UDP flows, each around 1.5 megabits are shown.

In summary, the front-end has three kinds of filters, 1. hardwired C filters, 2. user-mode BPF *filter-sets*, and 3. top N flow analysis done with a dynamically allocated hashed list of flow IDs. Intuitively we are interested in the relative execution cost of these three kinds of filters. Although it must be pointed out that there is only one top N flow filter, and a user may program any number of BPF *filter sets*. Also bear in mind that the *packet capture* filter is 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 likely due to too much work done at the application layer causing the application to fail to read buffered kernel packets in a timely manner.

3 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 Gigabit Ethernet line card. We used the Gigabit line cards to both send and receive UDP packet flows. One port sent packets and the other port received packets.

- 2. a Packet Engines line speed Ethernet switch. Three ports on the switch are used, including 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 gigahertz AMD 2000 CPU computer system. (This system is roughly comparable to an Intel Pentium 4 processor). The motherboard for this system is a Tyan Tiger MPX S2466N-4M. This motherboard has two 64-bit PCI slots and we used a Syskonnect SK-9843 SX Gigabit Ethernet card in one of the slots.

Software used included Ourmon 2.0, and the libpcap [17] library version 0.7.2. The host operating system was FreeBSD 4.7, running only the Ourmon front-end probe.

We setup the IXIA to send either minimum-sized packets or maximum-sized Ethernet packets. One port on the IXIA would send packets through the switch to the other IXIA port. Thus we were able to use the IXIA's per port packet counters to verify that packets were not lost as input counts matched output counts on both IXIA ports. All packets were UDP packets. Also the IXIA allows the user to slow the packet sending rate from the maxiumum rate to 0%. The IXIA is also able to auto-increment IP destination addresses, and we used this as an additional test against the top N filter.

As a measurement baseline, according to [6], one can calculate the maximum and minimum theoretical packet rates for Gigabit Ethernet as found in Table 1 below. We observed that the IXIA 1600 can indeed generate packets at nearly 100% of this rate for both minimum and maximum-sized packets. We used these numbers to make sure that our Ethernet switch did not drop packets by hooking both IXIA Gigabit ports up directly to the switch, and then sending packets from one IXIA port to another IXIA port. The IXIA reported that the switch sent and received packets at line speed in both cases.

	size(bytes)	pps
min	64	1488000
max	1518	81300

Table 1: Gigabit Ethernet PPS

Our basic method of test implementation was to set up the UNIX host with a shellscript driver and some set of Ourmon filters, start the front-end probe, and then configure the IXIA to send min/max packets at some rate between 0..100%. With Ourmon we simply would configure in or out our three kinds of filters, 1. hardwired, 2 user-mode BPF, and 3. top N as desired, and then run tests observing the results in the *mon.lite* output file.

The shellscript driver for the experiment is as follows:

```
#!/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
```

The FreeBSD sysctl(8) command is used to set the kernel BPF buffersize to a given size. This is because recent versions of the PCAP library on FreeBSD will take this information and automatically size the kernel buffer to be used by the application (ourmon) to the same 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 (4k bytes in FreeBSD 4.9) and was intended for the tcpdump sniffer application. The parameters to the Ourmon probe program tell it to take input from a local configuration file, dump the output information to the screen every five seconds and use the SysKonnect card as the input interface.

Overall our test plan consisted of tests based on either maximum-sized packets and minimum-sized packets. If we dropped packets, we attempted in every case to stop dropping packets by increasing the kernel BPF buffer size (BSIZE above). If that failed, we would then reduce the IXIA's send rate to where we stopped dropping packets.

For testing, we decided that there were five possible types of Ourmon filters groupings including the packet capture filter by itself, and categorized filter tests into these types: 1. the packet capture filter (hereafter called the "null" filter, because it cannot be turned off, and is the only remaining statistic when all ourmon filters (hardwired, etc.) are removed from the configuration). 2. the hardwired C filters as a group, 3. BPF filters as 1 or more filter-sets, 4. the top N filter mechanism, and 5. a simple combination of all filters.

Of course, the null filter tells us whether or not the BPF in the kernel was losing packets, as it was displaying the count/drop information taken from the operating system. Except for the null filter, and except for the combination filter test, the filters were always put in by themselves in order to determine if the filter type itself had an impact on the overall performance. There were six hardwired C filters (at the time of testing). BPF *filter-sets* were based on a 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), as it seemed reasonable for BPF expressions to always fail to match.

For the top N test, as it was not interesting to test the top N mechanism with the same IP flow over and over again, 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 in IP destinations.

4 Test Results

4.1 Maximum Packets

In this set of tests, packets are always 1518 bytes, the normal maximum MTU for Ethernet packets. (This works out to a 986 megabit flow of UDP packets). Our tests were as follows: 1. null filter only, 2. hardwired filters only, 3. top N filter only, 4. various BPF *filter sets*, and 5. combined filters as summarized in Table 2.

The flow rate was set to maximum so the drop rate shows packets lost at Gigabit speeds. With only the the null filter, the configuration almost worked with the default BPF buffer size of 4 kBytes, except that some packets were lost at a 30 second interval. This may have something to do with an operating system timer. Increasing the kernel BPF buffer size to 128 kBytes removed the loss. Adding in the hardwired filters caused no additional loss.

The top N filter worked with no loss at 1000 flows and completely failed at 10000 flows. Larger bpf-buffers did not help (shown as XXX in the table). This is the most significant failure case with maximum MTU packets. However we noted that if we decreased the IXIA flow-rate to 45mbits, it was possible to get back to 0% loss.

With the BPF *filter-sets*, we increased the number of filters to 8 sets (32 BPF expressions), and ran into some loss. At that point, we increased the kernel

test	bpf-sets	topn flows	min BPF size	drop rate
null filter			128 kB	0%
hardwired			128 kB	0%
topn		1000	128 kB	0%
topn		10000	XXX	80%
bpf	1		128 kB	0%
bpf	4		128 kB	0%
bpf	8		128 kB	20%
bpf	8		7 MB	0%
test config	1	1000	7 MB	0%

Table 2: Maximum Packet Tests

BPF buffer size. We found that a very large buffer of 7 MB could indeed get back to 0% packet loss. With our sample test configuration (hardwired filters + top N + 1 BPF set) we did not experience any loss, but this was because we were only using 1000 flows with the top N filter.

4.2 Minimum Packets

With minimum-sized packets of 64 bytes, due to the problems associated with capturing packets at that size, we will report our results as a series of small experiments. Each experiment focuses on a different test domain.

4.2.1 Null Filter Only

With only the null filter counting drops, it was not possible to capture all packets. Instead we attempted to determine if the kernel BPF buffer size had any impact as shown in the table below:

BPF buffer size	starting drop rate (mbits)
32 kB	53.33
128 kB	68.52
256 kB	76.19
512 kB	76.19

Table 3: Minimum Packets and Null Filter

A buffer size of 256 kBytes appears optimal and at the speed of 76 megabits the system begins to drop packets. 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 Gigabit stream without loss. [1]

4.2.2 Individual Filter Types

We have now determined at what rate the NULL filter can capture packets without further drops. We now measure what happens by adding in the three previously described sets of filters where each filter type is tested in isolation. In the BPF *filter-set* tests, we tried both one and two sets. In the top N filter test, we vary the number of different flows. Table 4 below shows the results for the hardwired and BPF tests. Table 5 shows the results for the top N tests.

test	bpf-sets	flow-rate (mbits)	drop rate
hardwired		76	0%
bpf	1	68	0%
bpf	2	53	0%

Table 4: Hardwired and BPF Tests

Hardwired filters have no impact on performance. BPF filters have some impact, and it can be seen that even at a modest 76 megabits as a starting point, real work has a cost.

At this speed, 1000 unique flows is stressful for the top N filter. However reducing the flow rate to 45 mbits allows the computer to process the data. Unfortunately 10000 flows with any kernel buffer size or

[1] Packet overhead with minimum packets results in a maxiumum flow of around 760 megabits.

flows	drop-rate	buffer-size	flow-rate(mbits)
1	0%	256 kB	76
100	1%	256 kB	76
1000	25%	256 kB	76
1000	0%	256 kB	45
10000	50%	*	*

Table 5: Minimum Packets - Top N Tests

speed simply fails. This suggests that simple standard hashing techniques may be too slow to keep up with denial of service attacks.

4.2.3 All Filter Types

In this experiment we measure all three types of filters at the same time. Here we only vary the flow-rate, and do not vary the previous 256 kBytes buffer size. The IXIA was sending 1000 flows.

flow-rate (mbits)	drop-rate
76	44%
68	37%
53	18%
45	03%
38	0%

Table 6: Minimum Packets - All Filter Types

Probably because of the top N filter fielding 1000 flows, we see that we must reduce the flow-rate by roughly one-half in order to prevent drops. The filters here are truthfully fairly minimal as there is only one BPF *filter set*. In reality, one would want more BPF *filter sets* as this feature is fundamental to Ourmon. The bottom-line here is that we must reduce the flow-rate to 38 megabits for even a modest amount of work to be performed without packet loss.

5 Conclusion

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

In terms of related work, regarding Ourmon itself, probably the closest similar system is ntop [4]. Ntop is a single program intended to run on desktops and in some sense can be viewed as a network version of the UNIX top program. Ourmon is designed more on the a model of a traditional distributed SNMP probe and relies heavily on user programmable BPF and RRDTOOL-based graphics. However in this paper, our focus is security and the differences between ourmon and ntop are not 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 as compared to an IDS tool like Snort that does signature-based analysis on every packet. Ourmon is lightweight compared to Snort simply because it only looks at the network headers, and does not look at the data payload. Thus it is reasonable to expect that if Ourmon cannot process a certain packet load, Snort's processing is likely further impaired.

In summary, we suggest that there are three points that we may glean from our test results. We will briefly discuss them in turn, and mention related work as appropriate. The last item is the most important.

- 1. The default BPF buffer size in FreeBSD of 4 kBytes is inadequate for a network monitoring system. We suggest a larger buffer of at least 256 kBytes in keeping with modern systems. Network administrators should understand that a megabyte buffer may be needed. From the heuristic point of view, the current Ourmon probe deployed in the PSU DMZ is running on an 2 gigahertz Intel Pentium 4, has a

7 megabyte kernel buffer, 60 BPF expressions, multiple kinds of top N filters, and rarely drops packets.

- 2. Our BPF filters seem to have a kernel buffer cost associated with them. and our results suggest that there is a relationship between kernel buffer space, and the number of BPFs used in our application. The 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. Thus it is hard to compare the BPF filter mechanism to the top N filter mechanism in terms of compute power. However the real computation problem for the top N system is simply that under attack it is driven to extremis by random IP addresses (source or destination). This is because a hash algorithm will first search for the flow ID, and then perform an insert if it fails to find the flow. Consequently random flows always cause an insert. The research question here is this: How can we deal with boundary conditions caused by random IP addresses without unduly impacting efficiency mechanisms meant for normal bursty flows?
- 3. The final result is simply the observation that our 2 gigahertz Pentium 4 class computer cannot capture more than 10% of the minimum-sized packet flow. Worse, if the computer is expected to do actual application level work with the data, the number of packets we capture without loss falls below 5%.

This last item deserves extended discussion. Consider an IDS system like Snort that wants to run an arbitrary number of signatures over not only the packet headers, but the packet data as well, and may choose to inject the data into a database system. Clearly per packet processing time can tend to be unbounded based on what we want to do with an individual packet. Now consider what the security world tells us about the security principle called *weakest link*. For example, Bruce Schneier in a recent book [16] says: "Security is a chain. It's only as secure as the weakest link." Therefore an IDS system cannot afford to miss a single packet, as that packet may be the one with the slammer worm that will infect an internal host. Worse perhaps a set of systems coordinated in a distributed DOS attack can launch an attack on an IDS monitor and first blind it with small packets and then sneak a one packet worm payload past it. Packet capture for small packets is clearly an open security problem.

There is some related work in the area of capturing small packets. For example, Mogul and Ramakrishnan [10] name our phenomenon 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 is approximately half a micro-second per packet. One is left with two possibilities. One can either improve the individual compute performance of various filter mechanisms or one can attack the problem with parallelism.

Two papers on enhancements to the Berkeley packet filter suggest different ways that might help BPF performance itself. [5] suggests changing the BPF to a general purpose compute machine, by allowing backward branches, thus increasing the solution space for compute problems that might be solved in the kernel itself. [2] reports improvements to the BPF using both machine-code compilation and various optimization techniques, resulting in impressive performance improvements, that could certainly be used in the current Ourmon implementation. Lastly [7] reports an interesting hardware parallel engine based on a flow slicing technique that is focused on improving Snort's performance under high-speed conditions.

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 and parallelization of Ourmon on the highspeed parallel Intel IXP system [1].

6 Acknowledgements

Much thanks to Bart Massey for his criticisms of this paper. We also thank the IXIA Corporation for their donation of an IXIA 1600.

References

[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Worich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, August 2002.

[2] A. Begel, S. McCanne, S. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. *Proceedings of ACM SIGCOMM*. September 1999.

[3] CERT Advisory CA-2003-04 MS-SQL Server Worm. `http://www.cert.org/advisories/CA-2003-04.html`, November 2003.

[4] L. Deri and S. Suin. Practical Network Security: Experiences with ntop, *IEEE Communications* Magazine, May 2000.

[5] 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.

[6] Karlin, Scott, Peterson, Larry, Maximum Packet Rates for Full-Duplex Ethernet, Technical Report TR-645-02, Department of Computer Science, Princeton University, Feb. 2002.

[7] C. Kruegel, F. Valeur, G Vignka, R. Kemmerer. Stateful Intrusion Detection in High-Speed Networks. In *Proceedings IEEE Symposium Security and Privacy*, IEEE Computer Society Press, Calif. 2002.

[8] Leffler, et. al., *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison-Wesley, 1989

[9] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter 1993 USENIX Conference*, San Diego, January 1993.

[10] J.C. Mogul and K.K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. In *ACM Transactions on Computer Systems*, 15(3):217-252, August 1997.

[11] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, N. Weaver. The Spread of the Sapphire/Slammer Worm. `http://www.cs.berkeley.edu/~nweaver/sapphire`. 2003.

[12] Ntop distribution page. `http://sourceforge.net/projects/ntop`. March 2004.

[13] Ourmon web page. `http://ourmon.cat.pdx.edu/ourmon`, March 2004.

[14] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. *In Proceedings of the USENIX LISA '99 Conference*, Novemember 1999.

[15] RRDTOOL web page. `http://people.ee.ethz.ch/~oetiker/webtools/rrdtool`. March 2004.

[16] B. Schneier. *Secrets and Lies*. p. xii. Wiley Computer Publishing. 2000.

[17] Tcpdump/libpcap home page. `http://www.tcpdump.org`, March 2004.

[18] Waldbusser, S. Remote Network Monitoring Management Information Base Version 2. IETF. RFC 2021, January 1997.

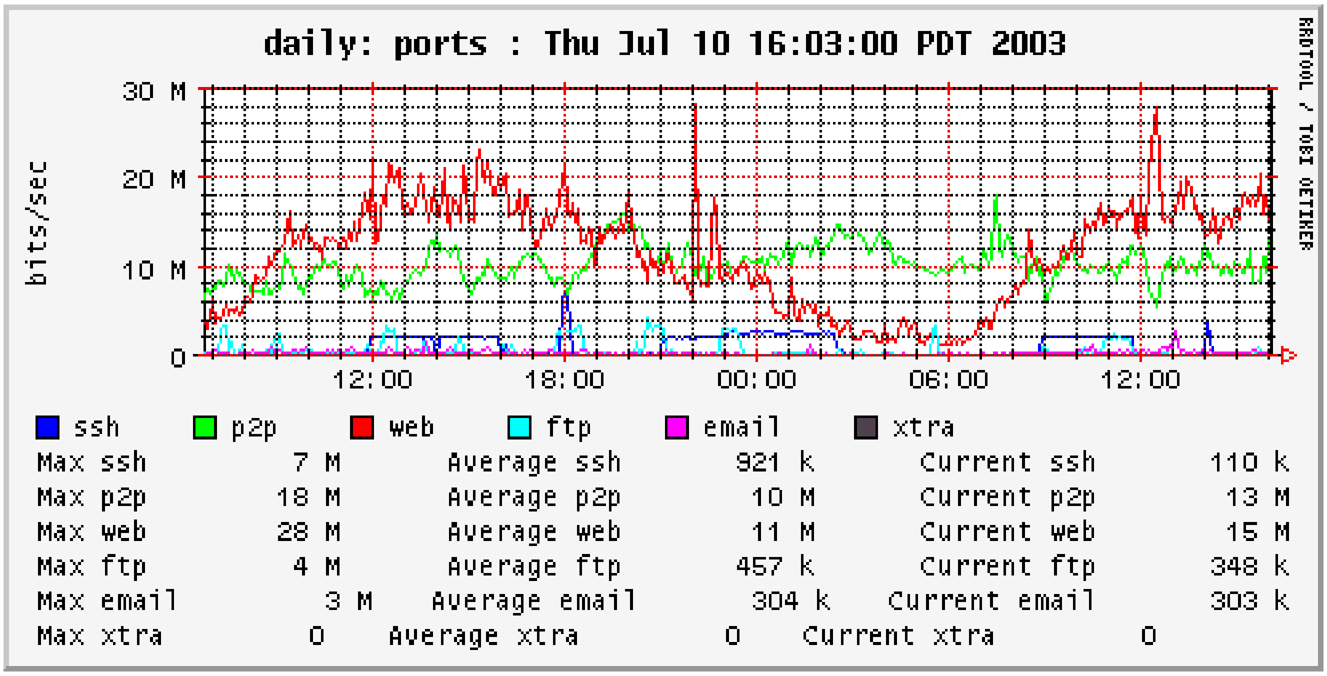

Figure 2: A BPF filter-set graph showing application byte counts

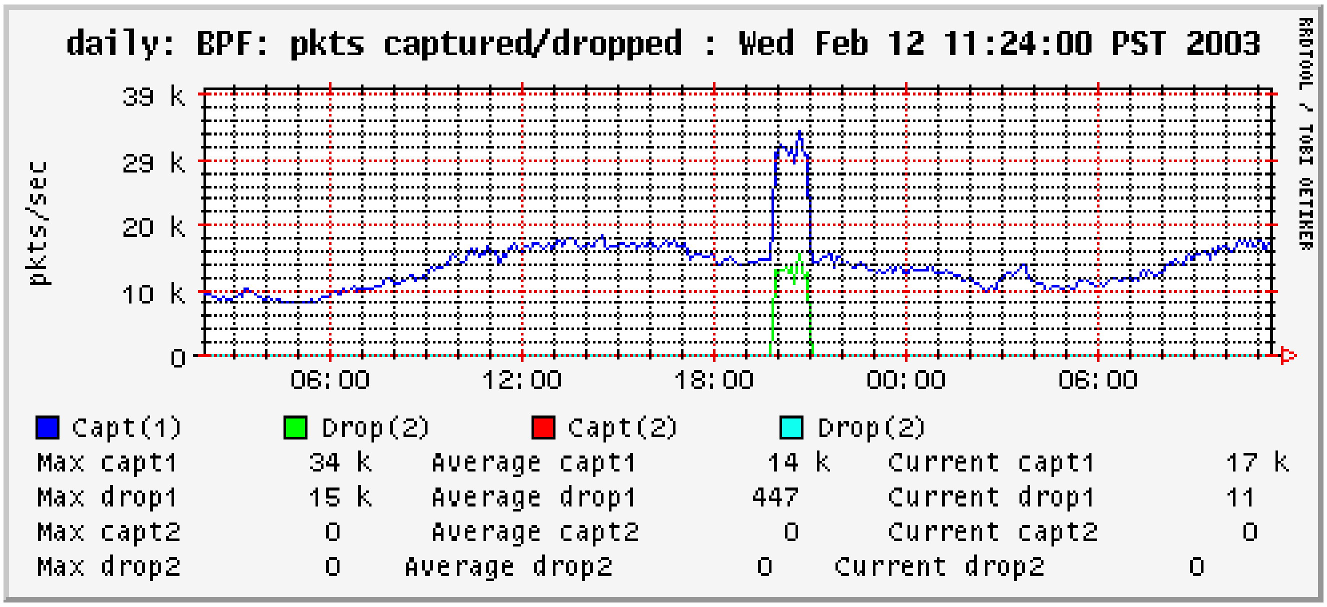

Figure 3: The packet capture filter graph showing counts and drops during a slammer attack

Figure 4: Top N UDP flow histogram showing a DOS attack