Ourmon and Network Monitoring Performance
Extended Abstract

James R. Binkley
Computer Science Dept.
Portland State University
Portland, OR, USA
jrb@cs.pdx.edu

Bart Massey
Computer Science Dept.
Portland State University
Portland, OR, USA
bart@cs.pdx.edu

Draft of 2004/10/22 22:14

Abstract

Open source intrusion detection systems are increasingly being deployed as protection against network-initiated attacks. While such systems can be highly effective against known threats, they are more problematic against unknown attacks. Ourmon is an open-source network anomaly detector that has been developed over a period of several years at Portland State University. Ourmon monitors a target network to highlight abnormal network traffic.

One problem that intrusion detection systems face is network-intensive attacks that can overwhelm monitoring and analysis resources. Experiences with Ourmon in a real network environment data and experiments with an IXIA high-speed packet generator verify this problem. While anomaly detection systems are less sensitive to packet loss, this is still an issue for them. Several modifications to Ourmon have noticeably improved its performance, and further performance improvement is expected.

1 Introduction

The Ourmon [13] network monitoring system is an open-source tool for real-time monitoring and measurement of traffic characteristics of a computer network. Ourmon uses the Berkeley Packet Filter (BPF) to capture interesting features of the envelope of incoming IP packets. Network monitoring and data visualization are typically performed on separate hosts. The data visualization system uses standard network graphical tools to display the resulting measurements in a fashion that highlights anomalies.

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 Mb NIC cards were infected simultaneously. These servers then sent approximately 360 Mb 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, a router sitting between network engineers and network instrumentation experienced thrashing behavior that overutilized the CPU. Engineers were thus cut off from central network instrumentation at the start of the attack.

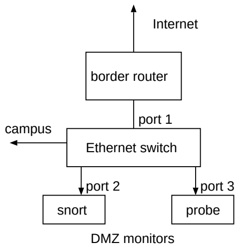

Figure 1: Ourmon network setup

As a result of the slammer attack, and also due to the timely and fortunate acquisition of an IXIA 1600 packet generator, experiments were conducted to evaluate the performance of the network monitoring portion of Ourmon with a series of realistic Gigabit Ethernet (gig-E) flows. The results of these experiments were quite interesting. For small packets on a gig-E line, the monitor failed to capture all of the presented packets even at relatively low input rates.

As a result of these experiments, attempts were made to understand the efficiency issues involved and to improve Ourmon's performance. Latencies associated with the BPF and with data capture were identified, and were dealt with relatively successfully.

The current Ourmon system appears to have performance sufficient to be a useful tool in the network environment in which it is fielded. However, further work could be done to improve its performance. Furthermore, the general problem of high-traffic network monitoring using open source and commodity hardware needs to be addressed in a more uniform fashion.

This extended abstract briefly introduces the Ourmon system (Section 2). It then discusses experiments with the Ourmon and the IXIA packet generator (Section 3) and the resulting mitigation activities (Section 4). Finally, some comments are made about the general problem of monitoring high-speed flows (Section 5).

2 Introduction to Ourmon

Ourmon is a real-time web-based network monitor somewhat similar to SNMP RMON [18] systems or ntop. 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 that of Snort (shown on port 2 of the switch). Ideally, the probe system captures all packets going to and from the Internet. The back-end Ourmon graphics engine normally runs on a second computer, which need not be in the center of a network.

A probe configuration file specifies various named filters (feature recognizers) for the probe to use. Probe output is written to a small ASCII file that represents a summarization of the last 30 seconds of filter activity in terms of byte or packet counts per named configuration filter. The probe output file acts as an input to the back-end, which in turn produces various graphics outputs and ASCII reports for web display. Two kinds of graphics are produced: RRDTOOL-based [15] strip charts used to visualize the filter outputs, and histograms used to display a "top-N" flow analysis.

The Ourmon probe uses the BPF in two ways. First the BPF acquires the packets using the kernel BPF buffer system. Second, the user-mode BPF interpreter is used to filter packet characteristics. The user-mode BPF filter takes multiple BPF filter expressions and groups them together in a BPF *filter set*. The back-end displays each filter output as a trace in an RRDTOOL strip chart. The current PSU DMZ probe software is running around 60 BPF expressions in sixteen filter sets.

Ourmon also supports a small set of "hardwired" C filters in the probe that are turned on via configuration names in the configuration file. As one example, a hardwired filter counts packets according to layer 2 unicast, multicast, or broadcast destination address types. Especially germane to the topic of high-volume flows is the *packet capture* filter. This filter collects statistics on dropped and counted packets provided directly from the BPF kernel code. The packet capture filter is used to determine when the kernel BPF mechanism is overloaded during testing. Figure 2 shows 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 re-infection. It can be seen that the Ourmon probe (at the time a Pentium-3) successfully characterized traffic during the attack, even though many packets were dropped.

The third and final filter class is the *top-N* flow monitor. The probe builds up a hashed sorted list of IP flows over the sample time. The top N (around 10) TCP, UDP, and overall IP flows are recorded in 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). Figure 3 shows the top-N graph for 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 Mb are shown.

The relative execution cost of these three classes of filters in a traffic-intensive environment is of interest. Note that while there is only one top-N flow filter, a user may program any number of BPF filter sets. The *packet capture* filter is an important indicator in this measurement: it serves to indicate loss of packets. This is an important indicator that the combined kernel and probe application system is in failure mode. This may be due to any number of causes: a likely one is that too much work is being done at the application layer, thus causing the application to fail to read buffered kernel packets in a timely manner.

3 Experiments with the IXIA

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 gig-E with packets of a given size, at what bit rate can the underlying BPF tap and buffer system not lose

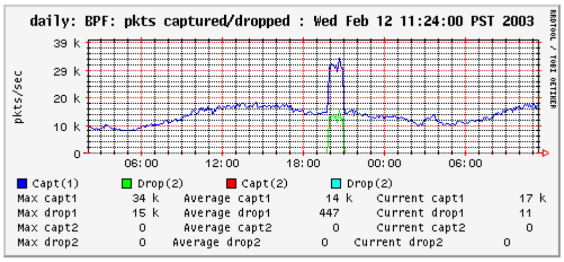

Figure 2: The packet capture filter graph showing counts and drops during a slammer attack

Figure 3: Top N UDP flow histogram showing a DOS attack

packets? 2) For a given packet size, if drops are encountered, can the kernel BPF buffer size be increased to mitigate the drops? 3) Ourmon has three classes of filters. Is there a reasonable characterization of the overall performance of filters of each class? Put another way: at high network packet rates, which of the Ourmon filter classes might be useful? 4) The slammer worm's semi-random IP destinations caused route-caching performance problems in routers. Does Ourmon's top-N flow monitor have similar performance problems given rolling IP destinations?

3.1 Experimental Setup

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

Space constraints preclude describing the hardware platform and test execution process in detail in this extended abstract: a thorough description will be given in the final report. Briefly, an IXIA 1600, a Packet Engines gig-E switch, and a 1.7 GHz AMD computer with a Syskonnect gig-E card were connected in a standard configuration. All tests were scripted, and data collection was automated. Software used included Ourmon 2.0 and libpcap [17] version 0.7.2. The host operating system was FreeBSD 4.7, running only the Ourmon front-end probe. Correct system operation was validated experimentally.

Tests were based on either maximum-sized packets and minimum-sized packets. For testing, five possible types of Ourmon filter tests were performed: 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 from one or more filter sets; 4) The top-N filter mechanism; 5) The combination of all filters.

The null filter detected any kernel BPF filter packet loss, by displaying the count/drop information taken from the operating system. For tests 3–5, the filters were always put in individually in order to determine if the filter type itself had an impact on the overall performance. There were 6 hardwired C filters at the time of testing. BPF filter sets were derived from 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.

The top-N test required a new flow for each iteration. A rolling IP destination setup was used, 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.

3.2 Test Results

For IXIA-based Ourmon tests, two different sets of flows were generated: one with maximum-sized packets (Section 3.2.1) and one with minimum-sized packets (Section 3.2.2). These results will be discussed in turn.

test	BPF sets	top-n flows	BPF minsize	drop rate
null filter			128 kB	0%
hardwired			128 kB	0%
top-n		1000	128 kB	0%
top-n		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 1: Maximum Packet Tests

3.2.1 Maximum-Sized Packets

Table 1 summarizes the performance of Ourmon given flows of maximum-sized packets. Entries shown as XXX in the table represent system failure. The flow rate was set to maximum: the drop rate thus shows packets lost at gig-E speeds. The null filter configuration almost worked with the default BPF buffer size of 4 KB. 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 KB 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. This is the most significant failure case with maximum MTU packets. Decreasing the IXIA flow-rate to 45 Mb completely mitigated these losses.

When the number of BPF filter sets was increased to 8 sets (32 BPF expressions), some loss was encountered. The kernel BPF buffer size was then increased. An unreasonably large BPF buffer of 7 MB completely mitigated the problem.

The complete test was lossless, since only 1000 top-N flows were used.

3.2.2 Minimum-Sized Packets

With minimum-sized packets of 64 bytes, the situation was much more problematic. Sub-experiments were conducted to explore the problem, focusing on different test domains.

Even with only the null filter counting drops, it was not possible to capture all packets at maximum rate. The effect of increasing kernel BPF buffer size was examined, as shown in Table 2. A buffer size of 256 KB appears optimal. At the speed of 76 Mb 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 not much more than 10% of the gig-E stream can be captured without loss.[1]

Once the maximum packet rate at which the kernel BPF could keep up was determined, the three previously described classes 2–4 of filters were tested in isolation. The BPF filter set tests were run with one and with two filter sets. The top-N filter test was run with varying numbers of different flows.

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

Table 2: Minimum Packets and Null Filter

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

Table 3: Hardwired and BPF Tests

Table 3 shows the results for the hardwired and BPF tests. Table 4 shows the results for the top-N tests. Hardwired filters have no impact on performance. BPF filters have some impact, and it can be seen that even at a modest 76 Mb as a starting point, real work has a cost. At this speed, 1000 unique flows is stressful for the top-N filter. Reducing the flow rate to 45 Mb allowed the computer to process the data. Unfortunately 10000 flows with any kernel buffer size or speed simply failed. This suggests that simple standard hashing techniques might be too slow to keep up with denial of service attacks.

In a final experiment, all three types of filters were measured at the same time. Only the flow-rate rate was varied: the buffer size was held constant at 256 KB. The IXIA was sending 1000 flows.

Probably because of the top-N filter fielding 1000 flows, the flow-rate had to be reduced by roughly one-half in order to prevent drops. The filters here are actually fairly minimal: there is only one BPF filter set. In reality one would need to use more BPF filter sets, as this feature is fundamental to the use of Ourmon. The bottom line is that flow rates as low as 38 Mb/sec were necessary for even a modest amount of work to be performed without packet loss.

4 Mitigation

The poor performance of the Ourmon probe on even modest flows of small packets was of obvious concern. While it has been demonstrated that Ourmon is useful as an anomaly detector even under conditions of severe packet loss (indeed, such a loss constitutes an anomaly indicator in its own right), the exhibited performance was both unexpectedly poor and a potential threat to some of the conclusions reached in real-world use. Several strategies were thus pursued in improving performance: two of these were somewhat successful.

The top-N flow monitor is both one of Ourmon's most useful tools and one of its least performant. It was observed that the hashing/caching strategies and data structures used in the initial implementation of this feature could be vastly improved. This improvement was done: initial results have been gratifying. While top-N is still a bottleneck, much larger flows can now be observed.

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

Table 4: Minimum Packets—top-N Tests

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

Table 5: Minimum Packets—All Filter Types

The user-level interpreted BPF filter performance also was a cause for some concern. Ourmon supports hard-coded C filters: as a stopgap measure, some of the filtering work is being moved into these filters. In the long-run, a more performant implementation of configurable filtering is highly desirable.

Evaluation of this mitigation work is not quite complete, but the final report will explain the improvements in algorithms in more detail and give quantitative measurements.

Other mitigations are highly desirable, but require much more effort. A number of schemes have been considered for improving the kernel-level raw packet capture performance. More measurement work needs to be done to understand the details of these losses.

5 Concluding Comments

Ourmon is a novel open-source network monitoring and anomaly detection tool.

The system most similar to Ourmon is probably ntop [4]. Ntop is a single program intended to run on desktops: it 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: it relies heavily on user programmable BPF and RRDTOOL-based graphics. From a security point of view, the differences between Ourmon and ntop are not so important. Ourmon and ntop are similar lightweight tools that show anomalous behavior via graphs. This contrasts sharply with an IDS tool like Snort that does signature-based analysis on every packet. Ourmon is lightweight compared to Snort: it looks only at the network headers, ignoring the data payload. It is thus reasonable to predict that Snort would fail to process some network loads that Ourmon can handle.

There are three points that may be gleaned from these test results. First, the default BPF buffer size in FreeBSD of 4 KB is inadequate for a network monitoring system. A larger buffer of at least 256 KB (modest for modern systems) is suggested. Network administrators should understand that a multi-MB buffer may be needed. As a point of comparison, the current Ourmon probe deployed in the PSU DMZ is running on an 2 GHz Intel Pentium 4, has a 7 MB kernel buffer, 60 BPF expressions, multiple kinds of top-N filters, and only drops packets during TCP SYN attacks.

Second, BPF filters seem to have a kernel buffer cost associated with them. There seems to be a relationship between kernel buffer space and the number of BPFs used by Ourmon. 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.

Finally, it is important not to miss the big picture. A 2 GHz Pentium-4 class computer drops most packets in a maximum-rate minimum-packet-size flow. These results are worsened if the computer is expected to do actual application level work with the data

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. This appears to be quite expensive. However, as Bruce Schneier comments [16], "Security is a chain. It's only as secure as the weakest link." 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, an attack on an IDS monitor might first blind it with small packets and then sneak a one-packet worm payload past it. For an IDS to effective, small-packet capture must be solved.

Some work has been done to solve this problem. For example, Mogul and Ramakrishnan [10] 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. Ioannidis et al [5] suggest 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. Begel et al [2] report improvements to the BPF using both machine-code compilation and various optimization techniques. Kreugel et al [7] report an interesting hardware parallel engine based on a flow slicing technique that is focused on improving Snort's performance under high-speed conditions.

There appears to be no easy solution to the packet-capture problem. As a result of the tests reported here, the authors have adopted the long-term objective of trying to produce a parallelized Ourmon system for the Intel IXP [1] processor.

Availability

Ourmon is freely available at `http://ourmon.cat.pdx.edu/ourmon` under the BSD License.

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.