Locality, Network Control and Anomaly Detection

May 25, 2004

Abstract

Ourmon is a near real-time web-based lightweight network monitoring and anomaly detection system that captures packets using port-mirroring on Ethernet switches. It primarily displays data via graphics using either RRDTOOL graphs or via histograms for top talker style graphs. We have developed a theory that network scanning launched primarily by worm programs including TCP and UDP scanners may be caught by monitoring network control data including TCP control packets (SYNS, FINS, RESETS) and ICMP errors, or by monitoring certain carefully chosen metadata such as the flow count itself. We present new notions of "flow tuples" including a TCP SYN tuple and a UDP error tuple, along with some weighting schemes used in our system. We illustrate our ideas with examples of attacks as graphed by the Ourmon system, and relate those examples to our control theory ideas.

1 Introduction

Recently John McHugh and Carrie Gates at CERT [6] have presented a novel theory of anomaly detection based on locality. Multiscale locality has proven to be a key to understanding a wide variety of physical and other phenomena. Locality of program counter and data references turned out to be the key to the design of effective memory paging systems [2]. In this case, the key locality concept is the "working set," i.e a set of memory pages that, if maintained in the physical memory of the computer will allow the program (or programs) in execution to make progress without excessive page faulting. This work was in response to the observation that, on some time sharing computers, page faults occurred so frequently that the CPU was mostly idle, waiting for the page(s) containing the next data or instructions to be referenced to be loaded into memory. This phenomenon, termed thrashing, led to a variety of models of program behavior, the understanding of which allowed efficient implementation of paged memory systems. As a side benefit, this area also led to studies that resulted in efficient data structures and algorithms for dealing with data whose size demanded organization in virtual memory.

The thesis of the earlier paper was that locality principles are a key to distinguishing and understanding "normal" behavior in computer systems that may be subject to attack by outsiders. We feel that an understanding of normal is an important step towards understanding that portion of abnormal behavior that represents the actions of malicious users of the system. Our long term goal is to develop a sufficient understanding of the systems with which we work so that we can identify properties that are necessary parts of certain malicious activities, and, with luck, properties that are sufficient to indicate such activities.

In general, locality is manifest when the behavior of the system can be represented by relatively compact clusters in some dimensions of a multidimensional measurement space. Note that we have not used the terms "benign" and "malicious" as surrogates for normal and abnormal. In this context, abnormal means unusual. In some cases, as we attempt to understand why locality appears to characterize normal behavior, we may be able to make a case that certain classes of malicious behavior are necessarily abnormal in that it will necessarily fail to meet the "normal" clustering criteria. On the other hand, we may not be able to make the case that all normal activity necessarily satisfies the "normal" clustering criteria so that failure to cluster is evidence of malicious behavior, but does not identify such behavior with absolute certainty.

In network terms, one may baseline local network data, and consequently observe significant anomalies in a baseline, thus detecting that network attacks are occuring. Furthermore, this principle of locality is something that should be built into network-based anomaly detection systems. Locality is represented both in terms of network address clustering as well as in the temporal behavior of sources both internal and external to the network. Ourmon, the subject of this paper, is another lens through which we can perceive manifestations of locality.

Ourmon [7] is a network monitoring system that is somewhat akin to systems like SNMP RMON [17] in that it attempts to capture network data including top talker graphs of traditional flows, packet counts, and counts of general protocol data including TCP versus UDP traffic, etc. It runs in thirty second sample periods using various filters that in general capture integer counts or top talker tuple lists. Ourmon is divided up architecturally into two programs, a front-end and a back-end. The front-end uses some hardwired and some user-defined filters for capturing data and placing the summarized data in a small file, that is then passed at the sample period time to the back-end which graphs the data on the web or analyzes it in ASCII reports. Thus Ourmon is a near real-time system, as the worst case delay in front-end to back-end processing is never more than one minute. In this paper we will focus on back-end graphics as examples of Ourmon's capabilities. [1]

Ourmon has two fundamental techniques for displaying data on the web in a graphical fashion:

1. RRDTOOL [10] stripcharts based on integer counters. Ourmon uses RRDTOOL to construct graphs based on various individual integer counters. The filter mechanism here is often based on a technique where multiple Berkeley Packet Filter (BPF) [5] expressions can be grouped in a single RRDTOOL graph. This is a major graphical tool in the Ourmon system and is often useful for looking at network-wide information. For example, we use it to graph the total numbers of TCP SYN, FIN, and RESET packets in our network. See Figure 1 for an example of an BPF/RRDTOOL graph.

2. Top talker lists of information expressed as histograms. Top talker lists typically have some form of tuple associated with them that is either keyed on an individual IP source address or on the traditional IP flow tuple of (IP source, IP destination, IP protocol, L4 source port, L4 destination port). See Figure 2 for an example.

At PSU, during Fall 2003 we noticed that the total count of ICMP flows, which was unfortunately not graphed at that time, had increased from 10's or 100's of flows to over 100000 flows. This was because students had returned from summer break and had apparently brought a mass infection back with them of the Welchia/NACHI worm [16]. This particular worm includes a ping scanner to scan for additional IP addresses, thus many such worms raised the overall ICMP flow count. It should be noted that this was a case of no particular host standing out by itself in terms of network traffic. What stood out was the increase in network-wide traffic for ICMP.

Of course, this is an excellent example of the principle of locality. We knew that ICMP flows at PSU were typically on the order of 10..100's per sample period (30 seconds), and should not be 100000 per period. We were also alerted to a system-wide anomaly, not just an anomaly for a single host. This particular incident caused us to begin our recent work to steer Ourmon in the direction of becoming an anomaly detection system as well as a general network monitoring system. It also led us to propose two network anomaly system design principles that we believe are of general importance and have found to be profitable in our attempts to capture network intrusions both from a network-wide and per source IP host perspective. The two principles are as follows:

[1] For more architectural details, please see: *http://ourmon.cat.pdx.edu/ourmon/info.html.*

1. It is useful to focus on network control data. For TCP this means SYNS, FINS, and RESETS as well as ICMP errors. For UDP this principally means ICMP port unreachables although other ICMP errors are useful as well. As an example, we have a top talker mechanism that captures TCP SYN attacks and a BPF graph that shows network-wide counts of TCP SYNS, FINS, and RESETS. In general, we can use our BPF/RRDTOOL to graph network-wide behavior, and use various tuple capture filters to look at the behavior of individual IP sources.

2. Carefully chosen metadata may also be of use for anomaly detection. For example, we originally chose to show the largest top talker IP flows as this is a very conventional way to display flow information. As it turns out, we should also have been graphing the number of flows. In the previous category we mentioned top talker SYN counting for individual IP sources. We also developed a metadata-based system that simply graphs the total number of suspicious systems sending many TCP SYNS, where we arbitrary declare that a certain number of SYNS - FINS per count period is "interesting". As a result, we have observed what appear to be coordinated SYN scanning "sweeps" that originate from many IP sources at a time.

We should point out that Ourmon at PSU is deployed in our DMZ. Our university has around 26000 students. We use Ourmon to look at network-wide traffic for a network with over 5000 hosts on it, 300+ Ethernet switches, and 10 routers plus external connections to Internet1 and Internet2. Thus all the graphs presented here come from our central DMZ monitoring station.

Our paper is organized around recently developed anomaly detection mechanisms based on the general locality theory. We illustrate these mechanisms with example graphs based on various attacks including large multiple IP source attacks that appear to be distributed SYN attacks, and small attacks based on single PSU hosts infected with recent viruses (since cleaned up). In section 2 we will look at TCP-based mechanisms that illustrate our control theory notion. In section 3 we look at mechanisms derived from ICMP errors, and UDP flows. In section 4 we look at some metadata examples including our worm counting mechanism. In the last sections we present related work, our conclusions, and possible future development work.

2 TCP Anomaly Detection

In this section, we are going to look at two filter mechanisms that are both focused on TCP anomaly detection.

We begin with Figure 1, where we show a daily (last 30 hours) picture using three BPF expressions in a RRDTOOL stripchart. This picture shows the total TCP SYN, FIN, and RESET packet counts for PSU traffic to and from the Internet for a period slightly over thirty hours. "Now" (10:00 AM) is on the right hand side and the stripchart moves to the left every thirty seconds. The top curve is the number of SYNs and the size of this curve has basically suppressed the FIN and RESET count lines, barring a small 6:00 am spike in the RESET line. Clearly the spikes in the SYN line indicate one set of major anomalies.

These attacks were caused by large-scale SYN attacks coming from the Internet into PSU during the time period.[2] The graph shows at the bottom that the variation in average SYNs to maximum SYNS was about 1 to 3. In other words a single attack could nearly triple the number of incoming SYNS. From historical experience we know that PSU's overall traffic is typically diurnal with peaks in the early afternoon. This graph makes a strong suggestion that the number of SYNS during the entire period at PSU is too high, and that the network is seeing a fair number of TCP SYN scans. One might expect that the number of SYNS and FINS would somehow march together even if there were less FINS. Of course, long-term baselining can help resolve this issue, but as this graph is new no such baseline currently exists. We should also point out that the distributed attack here

[2]We believe these attacks are distributed and coordinated SYN attacks that are simply looking for exploitable Microsoft systems, although IRC server attacks have also been observed.

is the same as shown in Figure 7 and Figure 8, which we will discuss later in our metadata section.

Our second TCP filter is based on a new top talker tuple, which is loosely modeled on our original flow list 5-tuple as found in Cisco's netflow tool [1]. We call this the *topn_syn* list and show an example histogram in Figure 2. The current generation tuple stored by the topn_syn list has the form:

```
(IP source address: SYNS: FINS: RESETS:
     TCPTOTAL)
```

The logical key in this tuple is an IP source address. SYNS, FINS, and RESETS are counts of TCP control packets. SYNS are counts of SYN packets sent from the IP source. FINS and RESETS are only counted when sent back to the IP source. In other words, they represent evidence of *two-way traffic*. The TCPTOTAL counter represents the total number of packets sent both ways (including control packets)[3]. Our current tuple sorting mechanism sorts on the number of SYN packets sent, although it also generates a number of interesting weights discussed below.

There are currently two weights used with this graph, which we call the *work weight*, and the *worm weight*. The *work weight* is computed as follows:

$$(S_s + F_r + R_r)/T_{sr}$$

and is expressed as a percent. The idea here is that we take the control packets counted and divide that count by the total number of TCP packets. This is roughly control divided by data. Obviousally 100% here is a very bad sign. On the other hand, if ordinary data packet exchange has occured, the weight will be much lower. (We will discuss the *worm weight* below in section 4 in more detail).

In the graph our histogram labels show the IP address, followed by the FIN (f), RESET (r), total count (t), and *work weight* plus an additional "worm" flag. The worm flag is based on the *work weight* metric and is set to "W" if that metric is 90% or more. 90% is probably a conservative number, however numerous manual checks with tcpdump [12] have shown instances greater than 90% to be worms (barring some email servers trying to return spam to non-existent addresses). The flag is set to "w" for a work metric in the range of 50..100%. Manual checks in this range have revealed both worms and IRC "bots" that seem to be anomalous as well (as well as one ill-behaved Gnutella P2P client). In the graph the line below the FIN and RESET counts shows the SYN count used for sorting.

A few packets of (simplified) tcpdump output for the top host in the graph are as follows:

```
131.252.205.73.3885 > 10.0.0.1.445: S
131.252.205.73.3886 > 10.0.0.2.445: S
131.252.205.73.3886 > 10.0.0.3.445: S
```

In other words, the PSU host in question is performing a port 445 (Microsoft file share) TCP SYN scan of external IP hosts. It has a virus and is searching for other hosts to infect.

Our first generation SYN tuple merely counted SYNS, FINS, and RESETS, and sorted on SYNS. We quickly learned that in general many SYNS and no FINS was a sign of an infected host. However we also found that we had false positives in that boxes running various forms of peer to peer applications also produced high rates of SYNS and it was not always clear if a "SYN-full" host had a virus or a Gnutella user. The most popular P2P clients at PSU are Bittorrent, Edonkey, Gnutella, and Kazaa. In order to eliminate false positives we made a study of TCP control packet counts with these P2P apps and determined that only Gnutella and Kazaa had work weights above 10%. Gnutella's average weight over millions of packets was on the order of 25%. Kazaa's weight was less at 20%. As a result of this process, and many instances of manual checks with tcpdump, we have mostly been able to ignore possible worms that in point of fact are P2P applications generating high numbers of FINS and RESETS.

The SYN list graph can at times be paired with the BPF TCP control graph. It is fair to state that the BPF control graph presents a network-based point of

[3]The next generation of this tuple will include a sampling of L4 destination ports, and will also distinguish the number of TCP packets sent from the IP source, and the number of TCP packets returned to the IP source. This is important because a spoofing IP source might choose to send garbage data packets to attempt to convince Ourmon that it is doing real work.

view and the SYN list gives individual instances of IP sources generating high rates of SYNs. We have seen examples where a spike in the control graph can be matched up to a log entry in the SYN list at the same time. However it is also often the case that many IP sources are generating small numbers of SYNS at the same time, thus raising the SYN line on the BPF graph, but not showing any obvious evidence in the SYN list. As a result, we modified the SYN list to generate a "tcpworm" report based on a second metric. This reports allow us to see the entire set of IP sources generating suspicious numbers of SYNS. We will explore this second-order mechanism further in the metadata section below.

3 Network Errors - ICMP and UDP

The focus in the previous section was on TCP control packets as well as the notion of two-way work with the TCP protocol. In this section, we turn to look at filters that focus for the most part on ICMP errors. The notion of two-way exchange (work or errors returned) plays a role here as well. We look at two top talker mechanisms, one of which simply shows ICMP flows, and one that shows a weighted UDP error scheme. In addition we look at a BPF-based graph that shows a network view of ICMP unreachable errors. We present graphs from two attack periods: 1. a major (in terms of network disturbance) UDP slammer attack, and 2. a more minor UDP scanning attack.

First we look at the UDP slammer attack which is shown in Figure 5, Figure 3 and Figure 4. Figure 5 shows the amount of traffic generated by layer 4 protocols, and normally can be construed as a graph belonging to network management, not anomaly detection. Here we see a severe attack from a single infected PSU host generating UDP packets at a rate nearly reaching 70 megabits per second. Figure 4 shows a BPF-based graph that is a total network count of four kinds of ICMP unreachable packets, including network, host, port, and administrative prohibited unreachable messages. The characteristic attack pattern (roughly a square wave) can be seen here as well. Individual UDP attacks as it turns out often cause spikes in this graph (TCP attacks may cause them as well, although the spikes are often less sharp).

In addition to conventional top talker graphs for all IP, TCP, and UDP flows, we also have a top talker graph for ICMP flows. As it turns out this graph can also be viewed as an anomaly detector. In general conventional TCP, or UDP flows, simply do not show anomalies as they are designed to show "big bits". However as this particular UDP worm can be viewed as a large disturbance in the network, it also left a large trail of ICMP errors behind it. Thus Figure 3 is particulary interesting. Here our slammer worm can clearly be seen to be generating a trail of ICMP havoc elsewhere, as all the bars in the histogram involve one IP host (which is highly unusual). The host in question is causing various kinds of ICMP errors including ttl exceeded in transit, ICMP unreachables, and curiously enough, routing redirect messages from many hops away.[4] However this particular event (fortunately) is rare, and one is not likely to often witness such a large scale error footprint.

We have learned that in general over a longer period of time (say an hour or a day), one can build a cumulative log that will discover individual hosts that pile up great numbers of ICMP errors and that may not be otherwise visible. In general, hosts with many ICMP errors are often worms, although other factors cannot as of yet be ruled out. Large numbers of ICMP unreachables, ICMP ttl exceeded, or ICMP routing redirects may indicate a worm infested system. Of course ICMP host unreachables are usually caused by UDP worms, and not TCP, but TCP may cause other kinds of ICMP errors, including ICMP administrative prohibited messages. Our system keeps a one week log of all top talker tuples. It also produces an hourly summary for the current day of ICMP log entries (and top SYN generators as well). Here is a simplified example of the hourly ICMP summary report with two flow entries:

[4]In our flow tuple the major ICMP code is given on the LHS, and the minor code is given on the RHS, thus (3, 3) means ICMP unreachable, port.

```
00:00:01: 993: 10.0.0.1->
     131.252.244.66:[unreach/port]:1764146
04:58:36: 1220: 10.0.0.2->
     131.252.243.64:[unreach/host]:352163
```

The daily beginning time and number of instances (times the flow appeared in log entries for the day) for the ICMP flow in question are given. The first flow accumlated nearly two megabits of ICMP unreachable port errors. The second flow shows a different host that has acquired a fair number of instances and bytes of ICMP unreachable host errors. Of course, our first instance is our UDP slammer worm, but in general, the notion that scanning hosts generate ICMP errors behind them has proven useful in practice, even for smaller scale worms.

In our last example, we look at a new top talker tuple mechanism shown in Figure 6 that we call *top UDP errors*. This graph shows a PSU dormitory host with a UDP virus infection (the system was scanning port 1433). Here we have a tuple that consists of:

```
(IP Source Address: UDP packets sent:
UDP packets returned: ICMP errors returned)
```

Tuples in this graph are sorted by a two-way *udp error weight* computed as follows:

$$w = (U_s - U_r) * I_r$$

For each IP source, we count the number of UDP packets sent and UDP packets returned to a UDP host. Our computed weight is then the difference between UDP packets sent and received times the number of ICMP errors returned. For example, our worm here has sent around 5000 UDP packets sent, with few packets returned, and generated around 600 or so ICMP errors returned as well. Thus its weight is around 3 million and clearly exceeds the next highest UDP error generator in the graph by two orders of magnitude. This graph is extraordinary compelling in that the histogram bar for this particular host is clearly suppressing all other hosts on the graph. We have seen that UDP worms are much more rare compared to TCP worms, and this graph has taught us that statistical outliers produced by this weight mechanism are often useful indicators and likely to be worms.

4 Metadata Examples

In this section we look at two useful anomaly detection mechanisms that are "metadata" by which we mean second-order graphs derived computationally from existing filters. Both figures in this section show distributed TCP SYN attacks coming into PSU over the same time period. These attacks may have used the agobot (or phatbot) tool and seemed to involve remotely controlled IRC chat servers [15].

Figure 7 which we will call the *flow count* graph shows the count of flows for IP (all IP flows), TCP, UDP, and ICMP flows respectively. We assert that it is reasonable to view the count of flows in a network as part of the control plane of the local network. Our top talker flow mechanism only shows the top N flows in its histograms in terms of bit rate (as in Figure 3 showing the top ICMP flows). Here we instead show the total count of all the flows during the sample period and represent the count for the four kinds of flows in an RRDTOOL graph.

In this case, we can see a number of spikes in the count of TCP flows. In general, baselined RRDTOOL data has shown that PSU traffic in terms of the TCP flow count is diurnal, with perhaps 1k flows at night and 2k flows during the day. Here we see one spike at 9:30 PM on the previous day that has doubled the number of flows to around 4k. Of course in this case we have "flows" of one packet as PSU's IP destination address space (a class B) is being walked by multiple external IP sources. This graph shows a total of 5 TCP spikes, and also one UDP event as well.

Figure 8 which we will call the *worm count* graph is probably our most interesting artifact. It is a count of suspicious TCP SYN scanners that is produced as a side effect from the previously mentioned top talker SYN tuple list. In this case our front-end statistics producing engine takes the entire sorted SYN list, and sorts it again according to the following weight:

$$S_s - F_r > C$$

We simply subtract the FINS returned from the SYNS sent and only count the tuple if C, a constant, is greater than 40. There are three counters shown in the tworm graph, 1. total IP sources, 2. "us" (meaning IP sources that belong to the home network, PSU), and 3. "them" (IP sources that are external to the home network). Thus IP sources in the list of SYN tuples are counted in the worm count if they have C more SYNS than FINS. They are classified as "us" if they match a configured home network and mask and "them" otherwise. The resulting graph clearly shows a number of sustained attacking periods with IP source counts in the worst case around 1k[5]. We were incredulous at first as to whether or not this filter worked, but the curves produced by it have been verified by hand using tcpdump, and by other graphs including the flow count filter pictured below and the RRDTOOL/BPF graph of TCP control packets (Figure 1). Thus we believe it is producing credible results.

Our subtractive metric seems to work simply because when large attacks are not taking place we are ignoring large numbers of small producers of SYNS and FINS (most applications) and hence establish a stable and small count baseline that may consist of some P2P apps and some single host infections. When an attack occurs, typically most of the attacks produce SYNS with few or no FINS, and hence raise the curve. We choose 40 at the time as an initial conservative measure, and believe that number to be too high. We are in the process of collecting data during new attacks and hope to refine that constant with a long-term statistical study.

Ourmon produces a report file called "tcpworm.txt" that is logged, and contains SYN tuples for all of the IP sources that satisfied the worm weight metric. Thus we can review this file to determine which specific IP hosts were involved in an attack.

5 Related Work

Until quite recently, scan detection has received relatively little attention compared to other intrusive activities. Part of the reason is the ubiquitous nature of scan and scan-like background data. Another has been the relatively primitive measures used in many intrusion detection systems for dealing with scans. The typical IDS detects scans with a relatively simple threshold measure. SNORT[13] is typical of this approach.

Bro[9] is similar but somewhat more stateful. Recent work by Stolfo's group at Columbia[11] can detect much slower scans as well as some distributed scans by associating scan state with a subnet address rather than an individual IP address. In addition, he has established a network of detectors that exchange information about scans detected at widely separated locations. Work at Silicon Defense[14] collects statistically anomalous events over a long time period and attempts to cluster them into distinct surveillance attempts. We note that this is another example of locality applied to scan detection. The cited paper also contains extensive background information on the survelience detection problem.

More recently, Jung, *et. al.*[4] have looked at the ratio of connection attempts to connection successes as a function of network occupancy to choose between competing hypotheses that the source of the attempts is a scanner who does not know the network structure or a benign user who occasionally fails to make a connection due to broken URL links or faulty DNS information. Gates, McHugh, and Binkley[3] have applied the Jung, *et. al.* analysis to NetFlow data.

6 Conclusion

In conclusion, we have suggested that network control data may be viewed as a rich source of information about normal network locality. In particular, network control data as represented in terms of TCP control packets, counts of flows, ICMP errors, and in the case of individual IP sources, counts of packets sent to and returned from the IP sources themselves is useful for anomaly detection. We have shown graphs that exploit this phenonmenon which are based either on integer counts (using RRDTOOL) or a top talker graphs (histograms) based on information and weights associated with particular IP source addresses. In general, the RRDTOOL graphs are giving us a network point of view. The histograms give

[5] Agobot is capable of spoofing IP source addresses and thus there really isn't any known way at this time to tell how many true IP hosts were involved in these attacks.

us a per IP source view.

Our control theory notion has suggested a number of interesting new tuples and metrics including:

1. A per IP source TCP SYN tuple as follows: (IP source address, TCP SYN count, FIN count returned, RESET count returned, Total TCP packets sent). We have derived two metrics from this SYN tuple including a *work weight* metric that seems to help us rule out P2P-based false positives and a *worm weight* metric that has helped us display large multiple IP source SYN attacks and determine IP source addresses of those involved in those attacks.

2. A per IP source UDP count tuple: (IP source, UDP packets sent, UDP packets returned, total ICMP errors returned). We associated an error weight with this tuple that gives a quadratic weight function for IP sources generating many UDP packets with little or no UDP data returned and with many ICMP errors returned. In theory, this metric should make UDP scanners stand out from normal sources of UDP traffic.

One important thread in these tuples and weights is the notion of two-way data. Data returned may either lend credibility to the notion that real work is going on, or detract from that notion if the packets returned are errors. Another important aspect of these tuples are error counts. For TCP that primarily means RESETS and for UDP, ICMP errors (although ICMP errors for TCP sources cannot be ruled out).

Although we feel our work reported here is exciting, It is also recent and preliminary. Anomaly detection work takes time and must be based on long-term analysis and long-term baselining of normal (and abnormal) data. As an example of even a simple metric that needs study, consider our RRDTOOL/BPF graph of SYNS, FINS, and RESETS in the PSU network. We are not sure what ratio of SYNS to FINS is reasonable in the PSU network? One might also ask what the ratio should be for particular kinds of applications (email or web), particular hosts, subnets, autonomous systems, and the Internet as a whole? In other words, what kinds of "localities" might exist, and what might one expect a healthy locality to look like? Such information would be invaluable for determining the health of that locality and possibly as an efficiency metric for TCP-based applications.

In the near future, we intend to refine our TCP SYN tuple and add more information to it. For example, we need to break up the total TCP count variable into packets sent and packets returned simply because a spoofing IP source might inflate the total count to make it appear that it is doing real work. It would also be useful to add some sampled destination port information as this would help make it clear what kinds of attacks are occuring. We also intend to study our TCP SYN tuple and the variables and weights associated with it in a long term statistical study in the hopes of determining if some variables are more important than others.

References

[1] Cisco Systems. Cisco CNS NetFlow Collection Engine. `http://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/products_user_guide_chapter09186a00801ed569.html`, April 2004.

[2] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall Inc., 1973.

[3] C. Gates, J. McHugh, and J. Binkley. An analysis of threshold random walk. 2004, Submitted to RAID 2004.

[4] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In Proceedings of the IEEE Security and Privacy Conference, Oakland, California, May 2004.

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

[6] J. McHugh and C. Gates. Locality: A new paradigm for thinking about normal behavior

and outsider threat. In *New Security Paradigms Workshop*, Anscona, Switzerland, August 2003.

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

[8] Ourmon architecture description web page. `http://ourmon.cat.pdx.edu/ourmon/info.html`, May 2004.

[9] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In Proceedings of the 7th USENIX Security Symposium, San Antonio, Texas, January 1998.

[10] RRDTOOL web page. `http://people.ee.ethz.ch/~oetiker/webtools/rrdtool`. November 2003.

[11] S. Robertson, E. Siegel, M. Miller, and S.Stolfo. Surveillance Detection in High Bandwidth Environments. In Proceedings of the 2003 DARPA DISCEX III Conference. April, 2003.

[12] Tcpdump/libpcap home page. `http://www.tcpdump.org`, September 2003.

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

[14] S. Staniford, J. A. Hoagland, J. M. McAlerney, "Practical Automated Detection of Stealthy Portscans," Seventh ACM Conference on Computer and Communications Security, Athens, Greece, 2000.

[15] Symantec Virus Information, W32.HLLW.Gaobot.gen. `http://securityresponse.symantec.com/avcenter/venc/data/w32.hllw.gaobot.gen.html`, May 05, 2004.

[16] Symantec Virus Information, W32.Welchia.Worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.welchia.worm.html`, August 18, 2003.

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

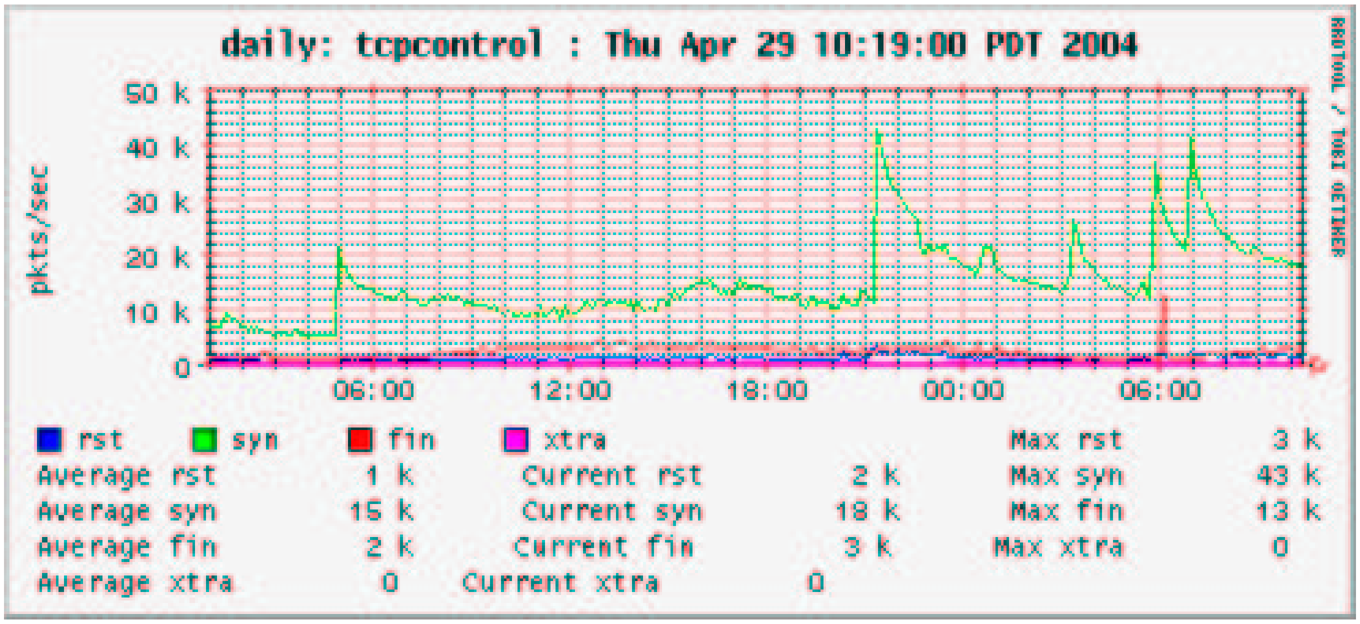


Figure 1: TCP Control Data - Large Syn Attack

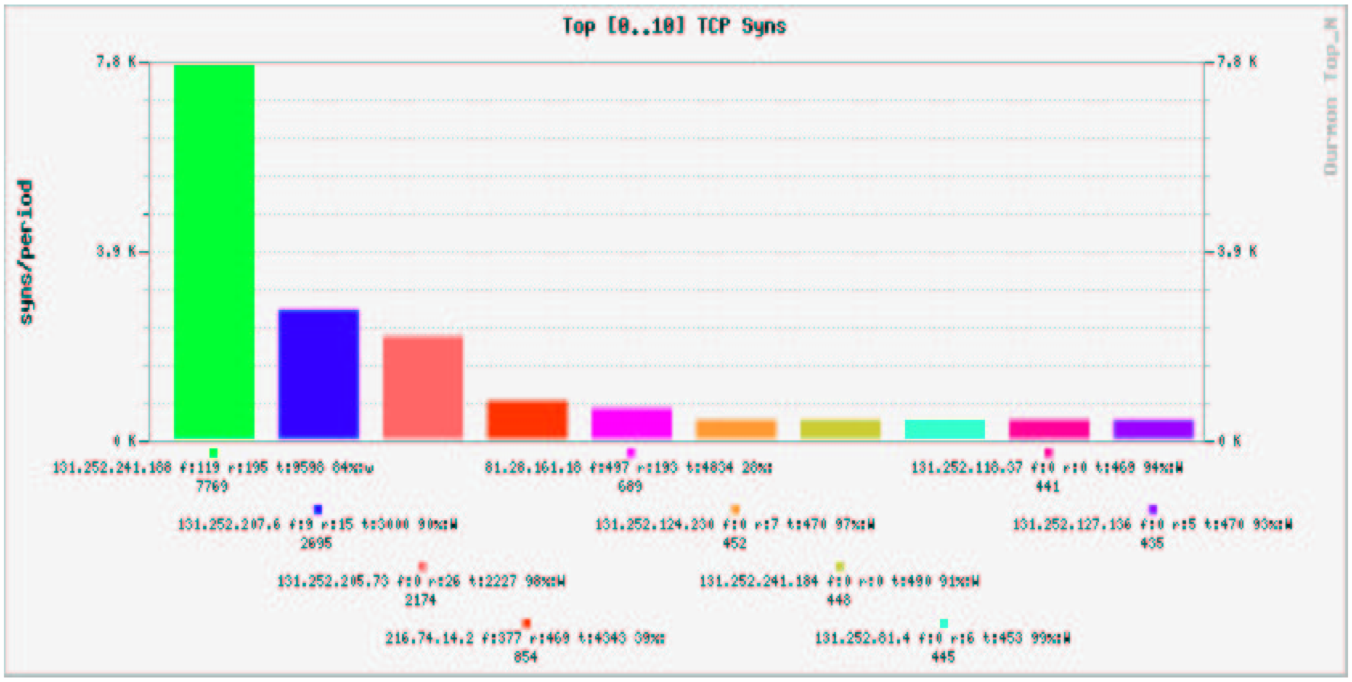
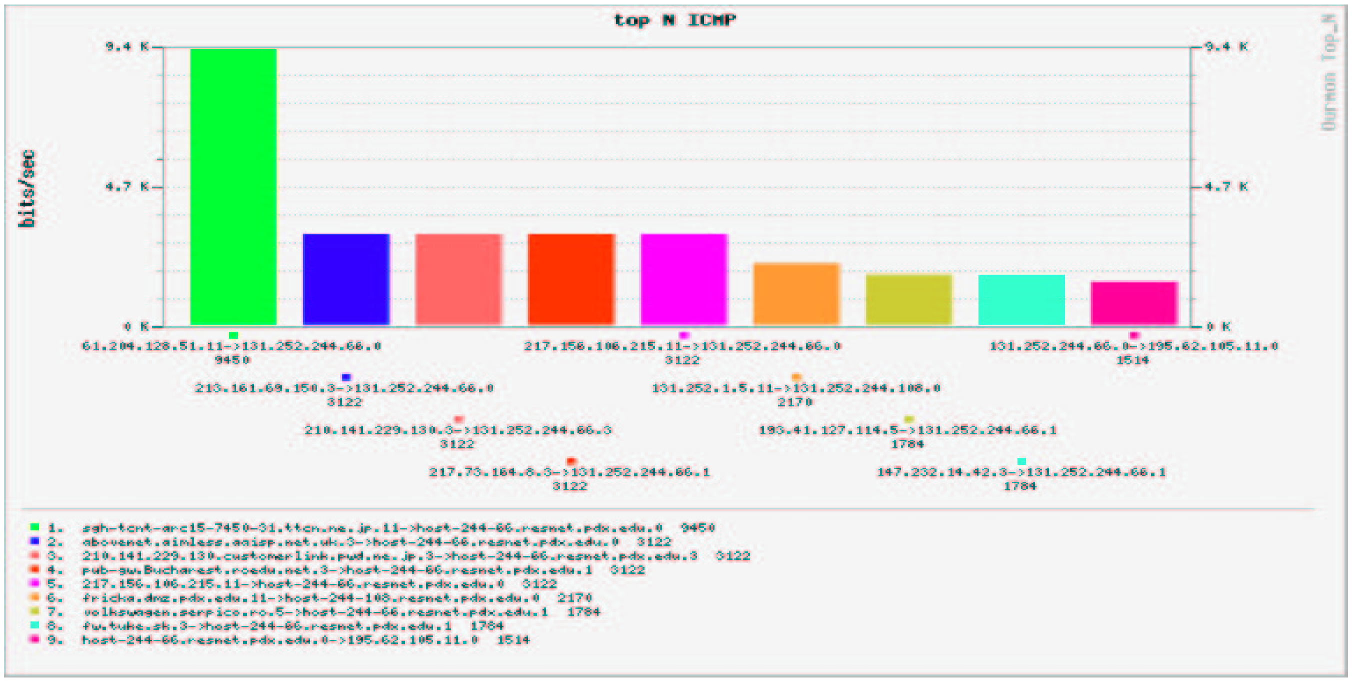
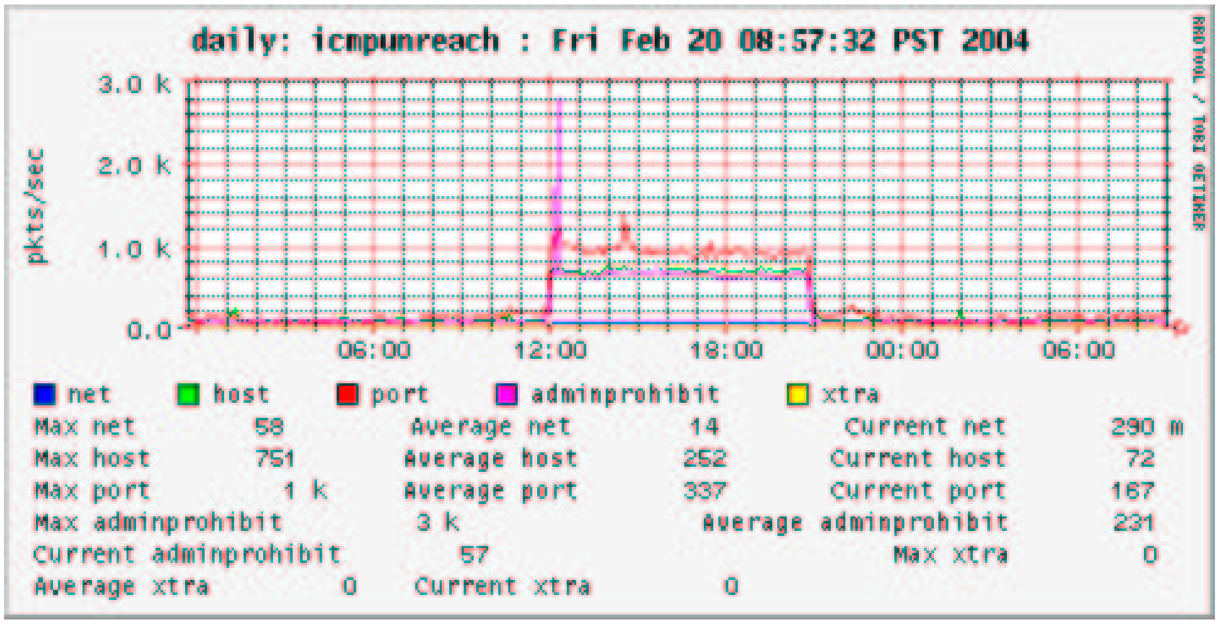


Figure 2: Top N TCP Syns - 445 Scanner



```
🟩 1.  sgh-tcnt-arc15-7450-31.ttcn.ne.jp.11->host-244-66.resnet.pdx.edu.0  9450
🟦 2.  abovenet.aimless.aaisp.net.uk.3->host-244-66.resnet.pdx.edu.0  3122
🟥 3.  210.141.229.130.customerlink.pwd.ne.jp.3->host-244-66.resnet.pdx.edu.3  3122
🟥 4.  pub-gw.Bucharest.roedu.net.3->host-244-66.resnet.pdx.edu.1  3122
🟪 5.  217.156.106.215.11->host-244-66.resnet.pdx.edu.0  3122
🟧 6.  fricka.dmz.pdx.edu.11->host-244-108.resnet.pdx.edu.0  2170
🟨 7.  volkswagen.serpico.ro.5->host-244-66.resnet.pdx.edu.1  1784
🟦 8.  fw.tuke.sk.3->host-244-66.resnet.pdx.edu.1  1784
🟥 9.  host-244-66.resnet.pdx.edu.0->195.62.105.11.0  1514
```

Figure 3: Top N ICMP flows - UDP slammer



```
🟦 net     🟩 host     🟥 port     🟪 adminprohibit     🟨 xtra
Max net            58        Average net       14        Current net       290 m
Max host           751       Average host      252       Current host      72
Max port           1 k       Average port      337       Current port      167
Max adminprohibit            3 k                 Average adminprohibit     231
Current adminprohibit         57                          Max xtra          0
Average xtra            0    Current xtra          0
```

Figure 4: ICMP Unreachables - UDP slammer

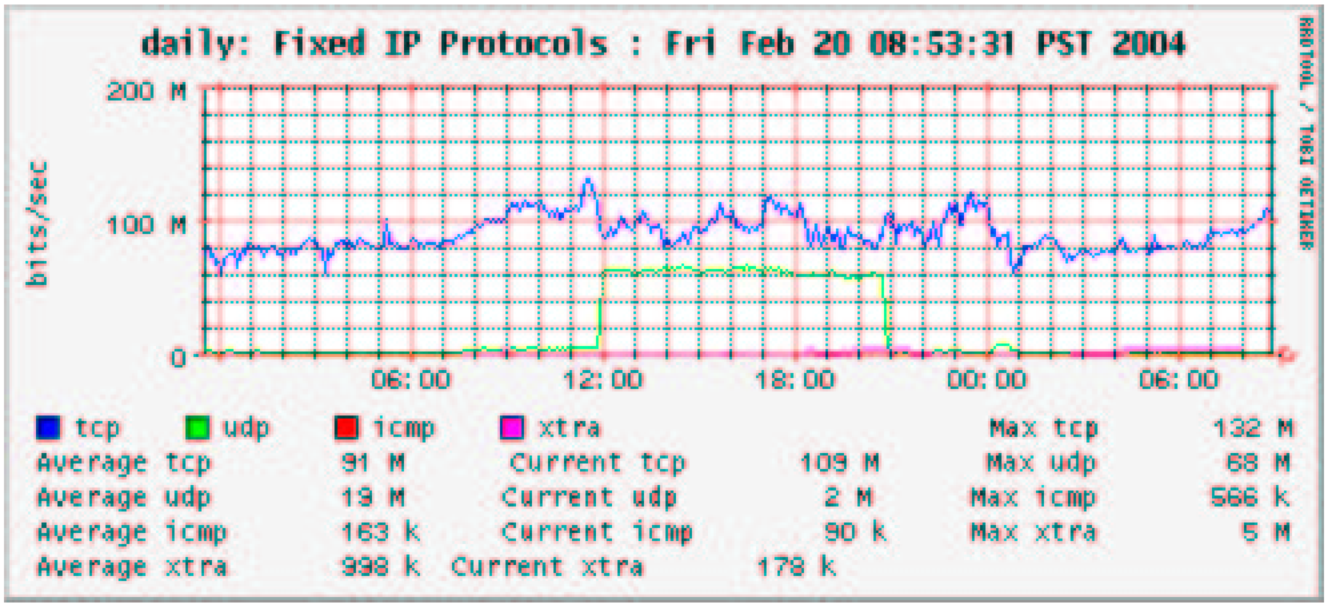


Figure 5: IP Protocol Bytes - UDP slammer

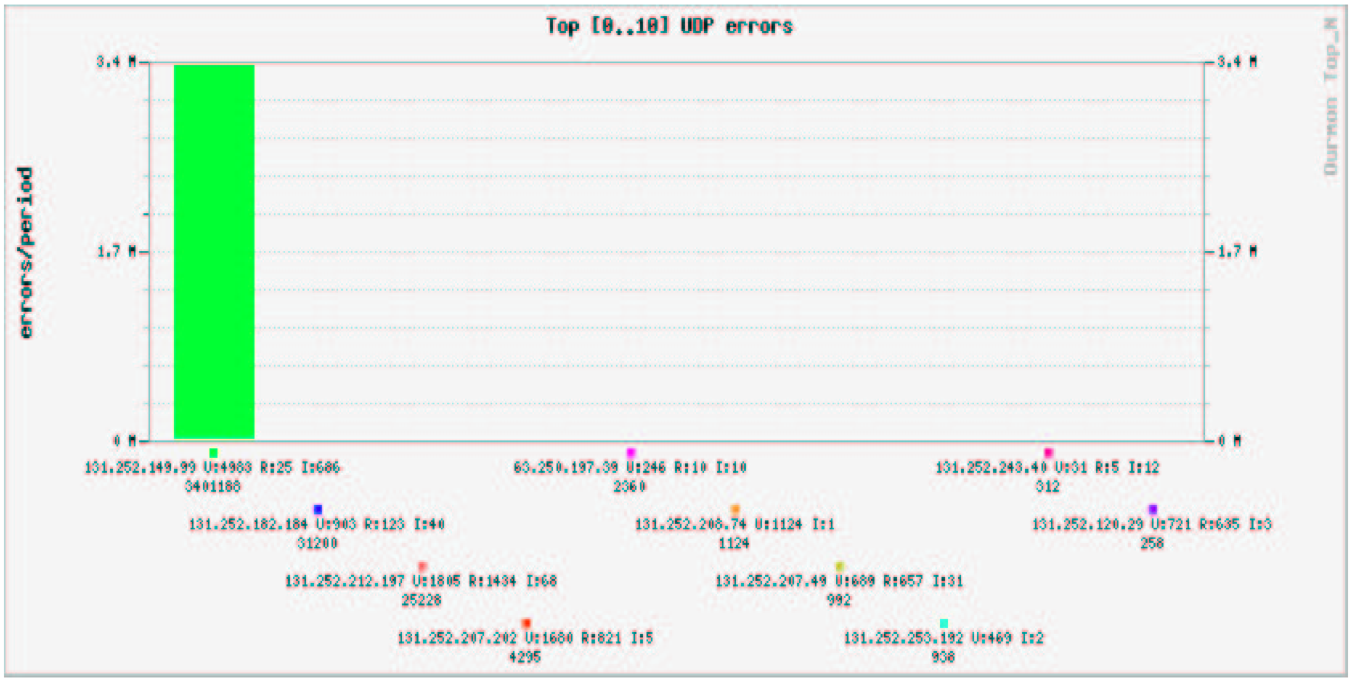


Figure 6: Top N UDP Errors - UDP port scanner

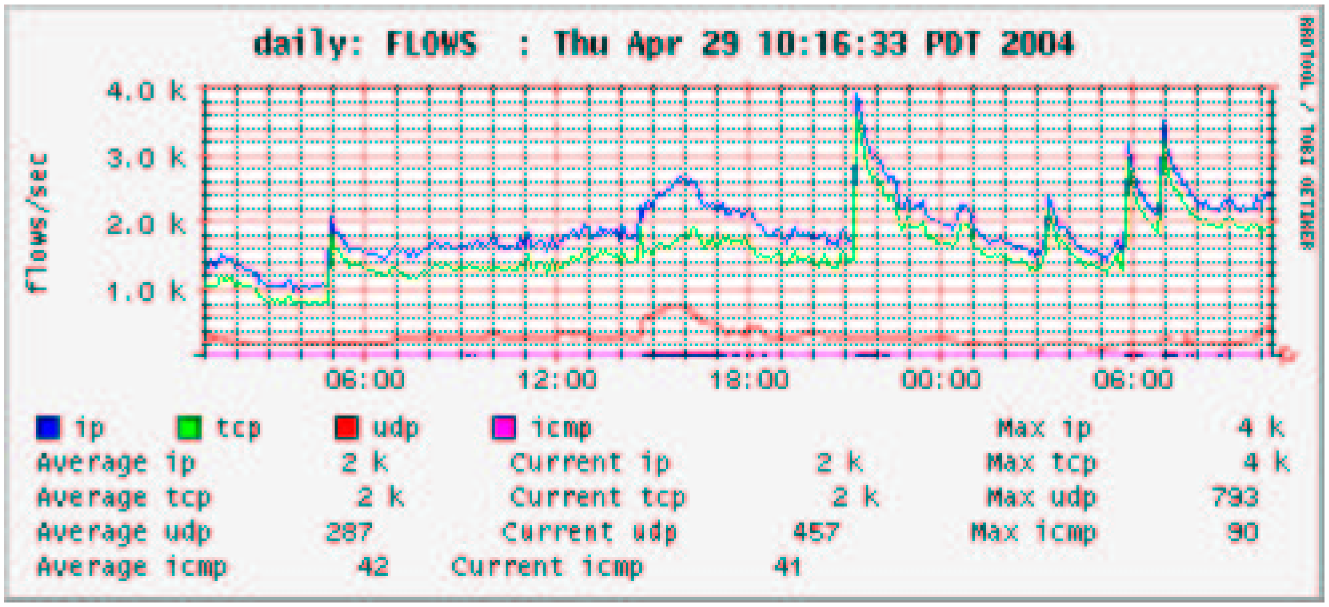


Figure 7: Top N flow count - Large Syn Attack

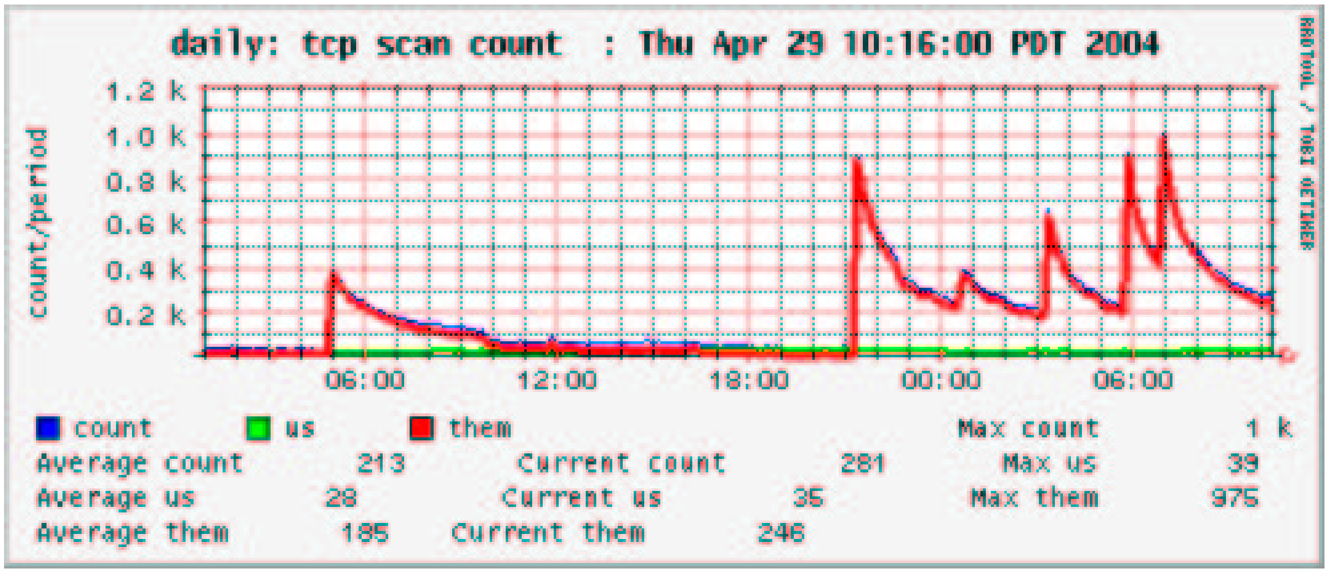


Figure 8: Worm Count - Large Syn Attack