TOWSON UNIVERISITY
OFFICE OF GRADUATE STUDIES

THE EFFECT OF PACKET LOSS ON NETWORK INTRUSION DETECTION

by

Sidney Charles Smith

A thesis

Presented to the faculty of

Towson University

in partial fulfillment

of the requirements for the degree

Master of Science

Department of Computer Science

Towson University
Towson, Maryland 21252

May 2013

TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

THESIS APPROVAL PAGE

This is to certify that the thesis prepared by Sidney Charles Smith entitled "The Effect of Packet Loss on Network Intrusion Detection" has been approved by the thesis committee as satisfactorily completing the thesis requirements for the degree of Master of Science in Computer Science.

Robert J. Hammell II, Chair, Thesis Committee — Date 4/23/13

Alexander Wijesinha, Committee Member — Date 4/23/13

Siddharth Kaza, Committee Member — Date 4/23/13

Michael O'Leary, Committee Member — Date 4-23-13

Lisa Marvel, Committee Member — Date 4/23/13

Ananthram Swami, Committee Member — Date 7 May 2013

Janet V. DeHaven
Dean, Graduate Studies — 5/11/13

Acknowledgements

I would like to thank the following people for their contributions to this effort. Dr. Lisa Marvel was an endless source of advice and good ideas. She is responsible for suggesting the random and state algorithms. Mr. Carlos Mateo contributed by helping me first designing and then obtaining the equipment to construct the experimental environment. Mr. Travis Parker helped load the Packet Dropper runs into the X-Wray Stats and Performance Explorer (SPEX). Mr. Kin Wong constructed the experimental environment and conducted the experiments which used it. Although Mr. Justin Wray constructed the X-Wray SPEX for a different experiment a couple of years ago, I want to acknowledge his efforts from which I am continuing to benefit.

Abstract

In this thesis we review the problem of packet loss as it pertains to Network Intrusion Detection with the intent to build a model that can be used to predict the impact of packet loss. We examine the potential places where packet loss may occur dividing the problem into network, host, and sensor based packets loss. We review the literature not only for other similar work, but for any work which might provide insight into this issue. We posit theories about how that packet loss may present itself. We construct a test environment and conduct experiments to induce packet loss in this environment. We develop the Packet Dropper application that induces packet loss into a dataset based upon eight different dropping algorithms selected to cover the theories previously posited. We apply each of these eight algorithms with drop rates ranging from 0% to 100% in 5% increments to the DARPA 98 Training, DARPA 98 Test, DARPA 99, CDX 2009, and CCDC 2010 datasets analyzing the resulting abridged datasets with Snort to collect alert information. The Alert Loss Rate (ALR) is plotted against the packet loss rate (PLR) allowing us to make general inferences about the relationship between PLR and ALR. In this paper we discovered that deterministic, bounded random, and random algorithms closely match the dropping patterns found in the literature and that a capped algorithm models the packet loss that we observed in our experiments. We present formulas that provide reasonable upper and lower bounds for the impact of PLR on ALR allowing us to predict this impact with some level of confidence.

Table of Contents

Table of Tables

Table of Figures

Introduction

Network Intrusion Detection (NID) depends upon the sensor being able to see the network traffic between the adversary and the target. This is often done by placing the sensor on a mirrored port in the network boundary which sends a copy of all the packets flowing into and out of the network to the sensor. Packet loss occurs when some of these packets fail to make it all the way to the sensor software for analysis. Intuitively, we understand that a sensor cannot detect what it cannot see; therefore, this packet loss must have a negative impact on the sensor's ability to detect malicious activity. This fundamental truth is well known, and much work has been done to reduce or eliminate packet loss. Very little work has been done to understand, predict, and model packet loss or the impact on network intrusion detection. The novel focus of this research is to build a model that can be used to predict packet loss and the impact on intrusion detection.

Problem Definition

The Internet is a very dangerous place for packets. They can collide and be destroyed. They can encounter a host, switch or a router with a full buffer and be dropped. They could pass through faulty systems and be altered. Internet protocols such as TCP are designed with the ability to compensate for these problems and retransmit dropped or altered packets. The focus of this research is not packet loss in general, but specifically any packet loss where packets from the adversary reach the target, but do not reach the sensor from a NID point of view. We will divide this problem into three distinct areas: network based packet loss, host based packet loss, and sensor based packet loss which are illustrated in Figure 1.

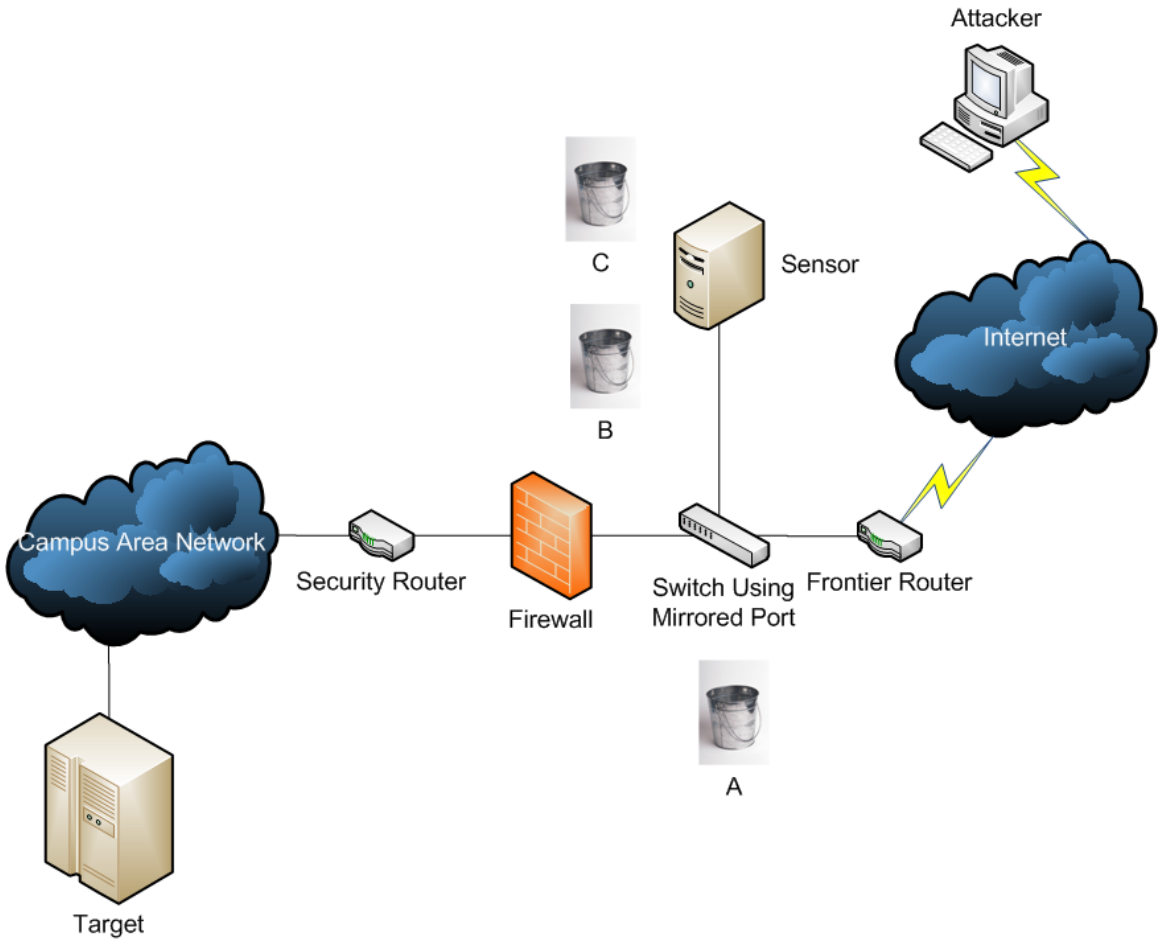

Figure 1 Packet Loss Diagram

Network Packet Loss. Network based packet loss is defined as any circumstance that would prevent packets which reach the target from reaching the network interface of the sensor and is represented by bit bucket A in Figure 1. Since the network from the switch to the sensor is significantly simpler than the network from switch to the target, we assume that almost all of this packet loss occurs in the switch itself as packets are delivered to the target network but fail to be mirrored.

Host Packet Loss. Host based packet loss is defined as any circumstance that would prevent packets which reach the network interface of the sensor from being presented to the analysis software and is represented by bit bucket B in Figure 1. A key

culprit here could be packets dropped because the buffers fill as new packets arrive faster than the CPU is capable of processing them.

Sensor Packet Loss. Sensor based packet loss is defined as circumstance that would prevent packets which are presented to the analysis software from being processed and is represented by bit bucket C in Figure 1. A prime candidate is resource exhaustion caused by the analysis software itself.

Objectives

Overall program objective: Methodology and Analysis of Capture Packet Loss and Impact on Network Intrusion Detection Systems (NIDS)				
Sub-topic	**Key Research Hypothesis**	**Research Question**	**Key Research Result Expected**	**Tangible Value**
Understanding Packet Loss	There exists patterns in packet loss that may be modeled with some level of fidelity	RQ1	A packet loss model which may be used to simulate packet loss	This model may be used to build packet loss resistant systems
Modeling Packet Loss	An application may be written that will simulate packet loss over an existing dataset	RQ3	A simulator will be developed that will allow analysis to be conducted without requiring experimentation using a network environment	This simulator may be used to inexpensively test packet loss resistant algorithms
Effect of Packet Loss	Packet Loss has a Predictable Impact upon NIDS that may be mathematically modeled.	RQ2, RQ4	A mathematical model for predicting the effect of packet loss on NIDS	This model may be used to set tolerances and allow network managers to make risk based decisions

Table 1 Overall program objective

Research Questions

During this research, we will answer the following research questions:

1. Is there sufficient regularity in packet loss to feasibility model it, or is this phenomenon sufficiently irregular as to be effectively considered random? The only work in this area seems to imply that packet loss is closely related to network traffic volume (Schaelicke & Freeland, 2005). However, this work did not consider network packet loss and appears not to have differentiated between host and sensor packet loss. This applies directly to sub-topic, "Understanding Packet Loss" in Table 1. If there is not sufficient regularity in packet loss, then we will be able to simulate packet loss as correctly as possible with a simple random packet loss algorithm.

2. Is the impact of packet loss on NIDS performance sufficiently regular to allow a formula to be developed which will accurately predict the effect? The only work in this area seemed to imply a fairly linear relationship where as packet loss increased, sensor alerts decreased; however, there seems to be very large standard deviation (Schaelicke & Freeland, 2005). This applies directly to the sub-topic, "Effect of Packet Loss" in Table 1. If the impact of packet loss on NIDS performance is completely random, then it will be impossible to set tolerances which may be used for risk based engineering decisions.

3. Would the same rate of packet loss induce the same loss of sensor alerts regardless of the strategy used to induce the packet loss? For example would a random packet loss induction of 5% cause the same loss in sensor alerts as 5% packet loss evenly distributed through the dataset, or would a sine wave based 5% packet loss algorithm cause the same loss in sensor alerts as a 5% packet loss based upon a Markov chain two

state channel model? This applies directly to the sub-topic, "Modeling Packet Loss" in Table 1. Using the Packet Dropper, which we will develop in this research, we will be able to compare the resulting loss in alerts at the same rate of packet loss over different loss algorithm. If the alert loss is highly dependent upon the packet loss algorithm, this implies that the correctness of the packet loss model is very important. Conversely, if the alert loss is independent of the packet loss strategy, then the correctness of the packet loss model is not very important.

4. Are the results independent of the composition of the network traffic? The only available results were obtained using only one dataset (Schaelicke & Freeland, 2005). Would the same result be obtained if different datasets were used? If packet loss were applied to capture the flag competition datasets that are exploit rich would be same relationship hold? If we applied packet loss to a much older dataset, would we see the same relationship implying that our results are likely to hold over time? This applies directly to the sub-topic, "Effect of Packet Loss" in Table 1 because the composition of network traffic varies significantly from site to site; if the results are highly dependent upon the composition of the network traffic, then we will unable to generalize our results.

Outline

The remainder of this paper is organized into the following sections. The "Literature Review" provides an overview of the existing literature as it pertains to network, host and sensor base packet loss. The "Methods and Materials" discusses our experimental work using the network test environment we created and the theoretical work using the Packet Dropper application that we developed. The "Results" discusses the findings from applying the Snort Network Intrusion Detection tool to datasets

abridged using the Packet Dropper. The “Discussion” section summarizes our findings and outlines directions for future research.

Literature Review

Previous research in this area has focused on eliminating packet loss. We will divide the literature review along the same lines that we have divided our research. One must keep in mind however, that this categorization is of our own making and previous research may not fit well into our categories. We observe that if we were able to model the Internet traffic that is received by the sensor, this analysis would be straight forward; however, modeling the Internet is a very difficult problem that has yet to be solved. Paxson and Floyd sited heterogeneity and rapid change as key factors complicating efforts to simulate the Internet (Paxson & Floyd, 1997). The heterogeneity and rapid change of the Internet has only increased since Paxson and Floyd did their research.

Network Packet Loss. During their work on detecting malicious packet losses, Mizrak et al. encountered the problem of distinguishing malicious packet loss which is caused by a compromised router from benign packet loss which is simply part and parcel of the way traffic flows through the Internet. They observed that "modern routers routinely drop packets due to bursts in traffic that exceed their buffering capacities, and the widely used Transmission Control Protocol (TCP) is designed to cause such loses as part of its normal congestion control behavior" (Mizrak, Savage, & Marzullo, 2009). Although generalized packet loss is not the focus of this research because it is assumed that the target and sensor are seeing the same traffic, many sensors are connected to the network through a mirrored port on a switch. Since mirroring is the lowest priority task that switches perform, it is possible to create a situation where the malicious traffic would reach the target but fail to reach the sensor (O'Neill, 2007). As illustrated in Figure 1 at

bit bucket A, the network path diverges at the mirrored port on the switch. The focus of this research is those packets that reach the target but fail to reach the sensor.

Host Packet Loss. The movement of packets from the Network Interface Card (NIC) through the kernel to user space where most sensor software is executed with multiple points of failure as illustrated in Figure 2 which expands upon bit bucket B from Figure 1. There are many things that may cause this as illustrated by the solutions provided to correct the problem. Handling interrupts is significantly more costly than processing more regular code because it cannot benefit from advances in processor performance. Interrupt handling may constitute 15% of the total processing time. Interrupt cost may be reduced 60% and the packet loss rate (PLR) by 46% by aggregating 32 interrupts (Schaelicke & Freeland, 2005). Packet loss was greatly reduced by increasing the level-2 cache (Schaelicke & Freeland, 2005). The Multi-Parallel Intrusion Detection Architecture was able to achieve impressive capture rates on a 10GbE network by parallelizing both the kernel and user land detection processes by assigning each flow to a single core (Vasiliadis, Polychronakis, & Ioannidis, 2011). Yueai and Junjie employed multiple sensors with load balancing to address the problem of host based packet loss (Yueai & Junjie, 2009). Chung, et al. discusses host based packet loss as packets that cannot be copied from kernel to user space quickly enough to prevent loss. They explore moving the IDS engine into kernel space as one solution to this problem (Chung, Kim, Sohn, & Park, 2004).

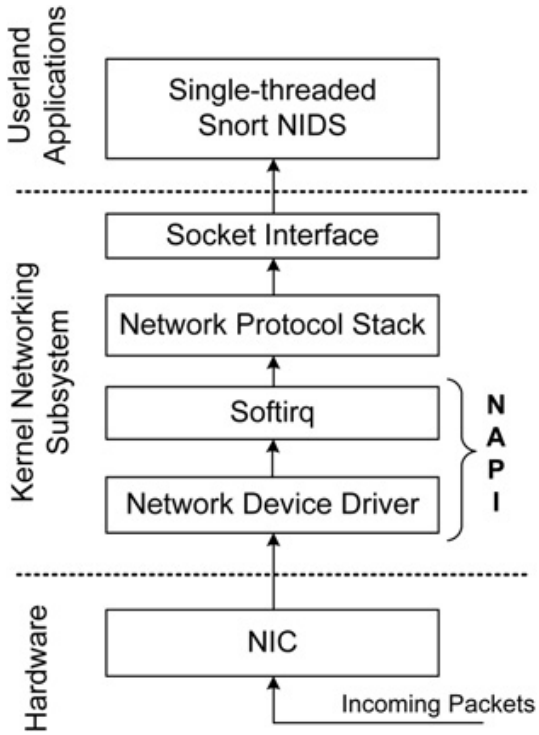

Figure 2 Snort NIDS underlying kernel support architecture (Salah & Kahtani, 2009)

Sensor Packet Loss. The number of different suggested solutions speaks to the various causes packets may not be processed by the analysis software. Some improvements in the PLR were obtained by increasing the level optimization employed in the compiler (Schaelicke & Freeland, 2005). Setting the netdev_budget, a Linux kernel configuration parameter, in the New Application Programming Interface (NAPI) to a low number like 2 has been shown to greatly decrease packet loss. This is primarily because both pulling packets from the network and analyzing the packets are bound by available CPU cycles and not buffer memory. By allocating the bulk of the CPU time to the sensor application, more packets may be processed and the packets that are dropped are dropped very early in the process where CPU cycles will not be wasted processing a packet which will only be dropped later (Salah & Kahtani, 2009). Significant improvements in the

PLR could also be achieved by pruning the sensor rule set (Schaelicke & Freeland, 2005). Multi-processing techniques have been employed to increase the throughput of the sensor and reduce packet loss (Kim, Park, Park, Jung, Eom, & Chung, 2011). In addition to this Song et al. point to the software crash as yet another cause of packet loss (Song, Yang, Chen, Zhao, & Fan, 2010). Wei et al. consider the problem of packet loss in an IPv6 environment. As a solution they propose breaking the detection task into three units: the Data Acquisition Unit, the Adapt Load Characteristic Analysis unit, and the Collaborative Analysis and Control Center (Wei, Fang, Li, Liu, & Yang, 2008).

Combined Effect. In their paper, "Characterizing Sources and Remedies for Packet Loss in Network Intrusion Detection Systems", Schaelicke and Freeland observed a near linear relationship between PLR and Alert Loss Rate (ALR). We very grossly simplified the results shown in Figure 3 and using a graphical method we could describe the relationship by the equation: $ALR = PLR - 15$. According to this model, if one is seeing a 20% PLR one should expect a 5% ALR. Looking at Figure 4 we can see that they observed packet loss which rose and fell almost identically to the traffic that they captured. Since the focus of their paper was reducing packet loss, they made little effort to validate or generalize their model. Their network traffic consisted of 13 seconds captured on the Internet connection of a major university containing over 530,000 packets that contained 521 known attacks at a ratio of about 1000 packets per alert (Schaelicke & Freeland, 2005). Although sufficient for their purposes, there is insufficient robustness to extrapolate a general packet loss to alert loss relationship from their results.

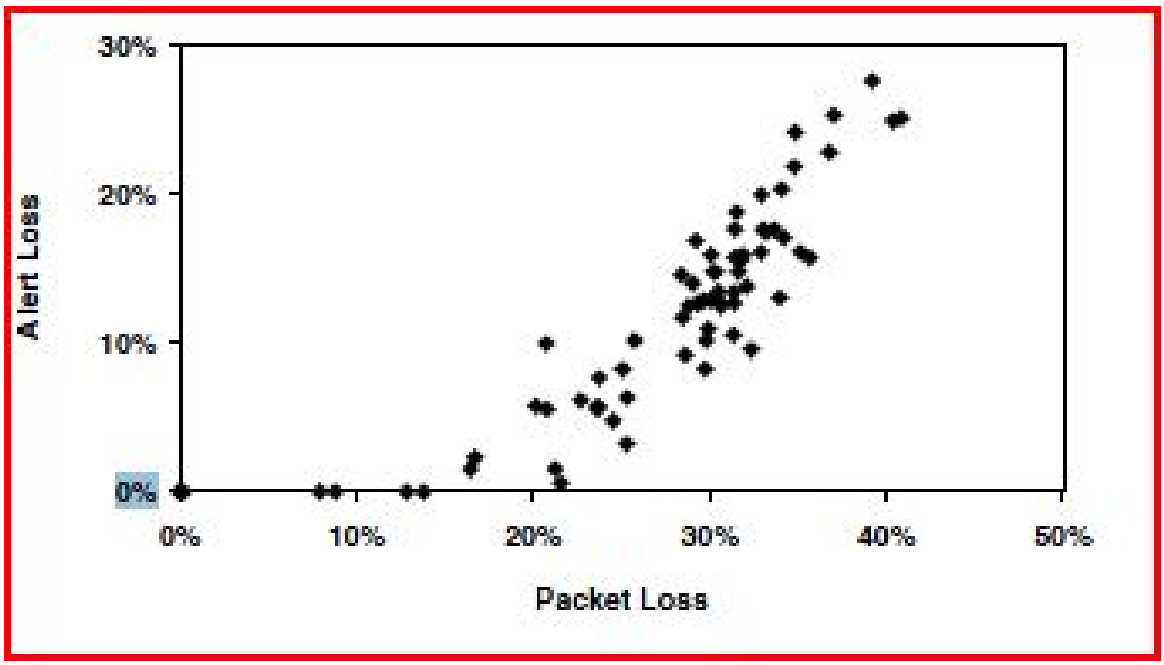

Figure 3 Alert Loss vs Packet Loss (Schaelicke & Freeland, 2005)

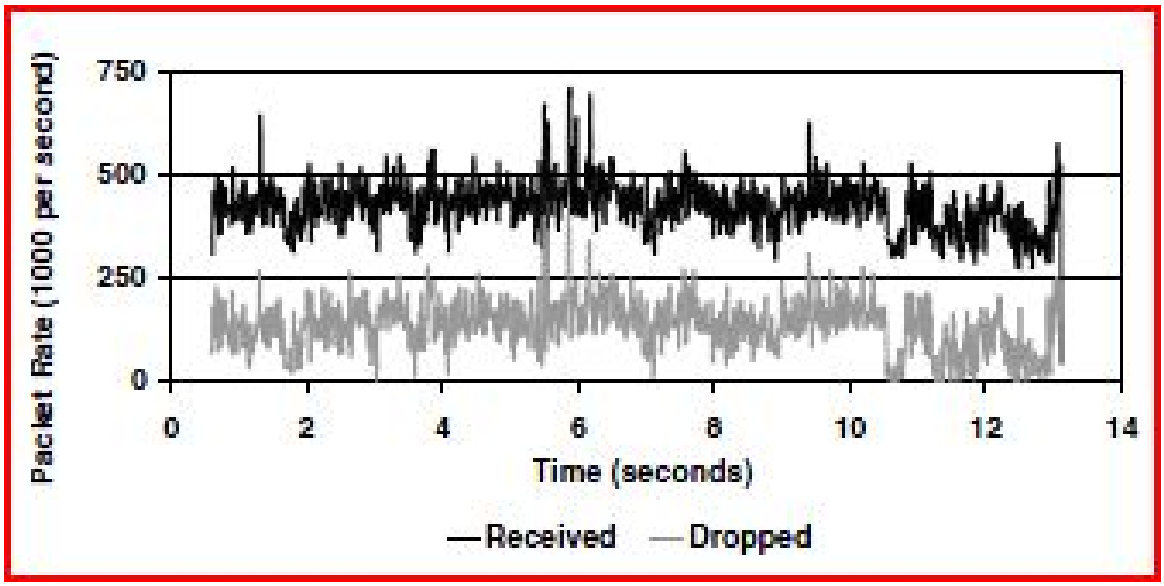

Figure 4 Packet Rate vs Time (Schaelicke & Freeland, 2005)

It is clear from all of the effort expended to reduce packet loss that it is generally considered to be a serious problem which promises to only get more serious as network bandwidth continues to outpace CPU clock speeds. Although much work has been done to minimize packet loss, very little work has been done to characterize packet loss and

even less work has been done to quantify the impact of packet loss on intrusion detection. The purpose of the effort is to understand, predict and model both packet loss itself and the impact upon network intrusion detection.

Methods and Materials

Understanding Packet Loss

In order to understand packet loss we will begin by reviewing the key points where packet loss might take place and based upon the information gained in the literature review we will posit theories on how this packet loss may present itself. Then we will design and conduct experiments to induce each type of packet loss in a laboratory environment capturing the traffic for analysis.

Theory

Network Packet Loss. Revisiting Figure 1 bit bucket A, we see a network switch with basically three network connections. The first connection is from the switch to the frontier router which is connected to the Internet point of presence. The second connection is from the switch to the firewall which is connected to the Intranet backbone. The third connection is from the mirrored port on the switch to the NID sensor. Given that the routing and firewall functions are significantly more complicated than the switching function, one would not expect the switch to have difficulty keeping pace with the two. However, there is a potential problem inherent in this design. Unlike network hubs which are half duplex, network switches are full duplex. Although traffic may flow from the firewall to the frontier router through the switch in both directions at the same time, there is still only one traffic path between the switch and the sensor. When packets pass each other at the switch, one must be buffered until the other has completed transmission to the sensor. In the event of heavy traffic in both directions, one could image this buffer filling and packets being lost. Since the nature of TCP's congestion control algorithm tends to make network traffic bursty, we theorize that we can model

this kind of packet loss using a two-state Markov chain similar to the one that has been used since 1960 to simulate channel fading and packet loss in wireless channels (Gilbert, 1960).

Host Packet Loss. Revisiting Figure 1 bit bucket B, we see buffers in the kernel used to hold packets that have been received by the NIC and have yet to be processed by the CPU. One could easily envision a general purpose computer system being unable to keep up with the firewall and the frontier router which are both dedicated network devices tuned specifically to move network traffic. We theorize that we can simulate this effect by implementing what we will call a capped algorithm. This capped algorithm will implement a queue of packets. As we look at each packet we will drain the queue of any packet that is some interval older than the current packet. Once this is complete, if the number of packets per interval or bytes per interval is below some cap, we will add the packet to the queue; otherwise we will drop the packet.

Sensor Packet Loss. Revisiting Figure 1 bit bucket C, we expect that the resource consumption of the sensor application itself will contribute the greatest component of packet loss at this level. We theorize that we can simulate this effect through some cyclic function such as a sine wave or a saw tooth wave which would model this pattern of resource utilization.

Experiments

We studied the phenomenon at the network, host, and sensor levels. We theorized that the number of packets dropped as network traffic increases would be regular enough that we will be able plot this relationship. Further, we theorized that we can use regression techniques to discover a formula that we can use to simulate packet loss with

some level of fidelity. Each of these experiments will produce an abridged dataset which we processed by an Open Source NIDS. These experiments are directly related to the sub-topic, "Understanding Packet Loss" from Table 1, and address research question 1.

Experimental Environment

Figure 5 is a diagram of the network we constructed for conducting our experiments. Table 2 provides the hardware specifications, and Table 3 provides the software specification of this environment. The switch is configured as a layer 3 switch with 2 VLANs and traffic routed between them. The VLANs separates the hosts into an "external" network, VLAN 100, and an "internal" network, VLAN 200. This configuration allows mirroring of all traffic from both VLANs to a collection port which is similar to how the sensors are typically set up.

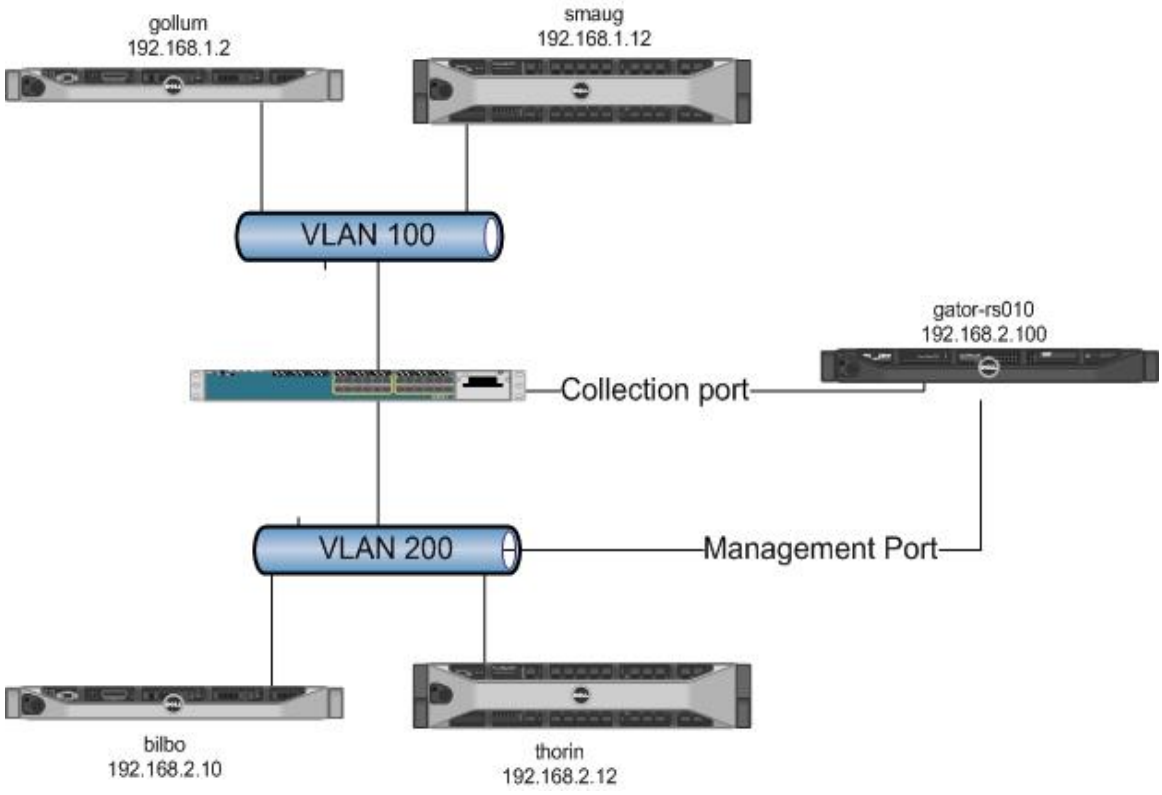

Figure 5 Experimental Environment

Name	Manufacture	Model	CPU	Memory	Hard drive	IP Address
Bilbo	Dell	PowerEdge R610	Intel Xeon 16 core X5450 @ 2.53 GHz	12 GB	4X 300 GB 10K SAS	192.168.2.10
Gator-rs010	Dell	PowerEdge R210 II	Intel Xeon 4 core, E31220 @ 3.10 GHz,	8 GB		192.168.2.100
Gollum	Dell	PowerEdge R610	Intel Xeon 16 core, E5540 @ 2.53 GHz	12 GB	4X 300GB 10K SAS	192.168.1.2
Smaug	Dell	PowerEdge 2950	Intel Xeon 8 core, X5450 @ 3.00 GHz,	8 GB	1X 300 GB 10K SAS	192.168.1.12
rsswitch	Cisco	Catalyst 3560-X				
Thorin	Dell	PowerEdge 2950	Intel Xeon 8 core, X5450 @ 3.00 GHz,	8 GB	6X 145 GB 15K SAS	192.168.2.12

Table 2 Hardware Specifications

Name	Source	Version
Snort	www.snort.org	2.9.4
Tcpdump	www.tcpdump.org	4.3.0
Libpcap	www.tcpdump.org	1.3.0
Tcpreplay	tcpreplay.synfin.net	3.4.4
MGEN	cs.itd.nrl.navy.mil	5.02

Table 3 Software Specifications

Experiment 1 Network Based Packet Loss.

First we configured Gollum to be the MGEN source and Bilbo to be the MGEN sink and sent packets from Gollum to Bilbo while collecting everything on Gator-rs010. We were not able, in this configuration, to send enough packets over the switch to cause the mirror to fail. Next using Aaron Turner's tcpreplay (Tcpreplay) we were able to replay the hour of network traffic from the Cyber Defense eXercise (CDX) 2009 that we will later use to show the impact of our packet loss algorithms. Tcpreplay provides the ability to rerun the traffic at arbitrary speeds. Table 4 lists the speed multiplier that we used and the packet loss we observed. We were able to replay that hour of the CDX 2009 data at speeds, over 1000 times the original speed, and were not able to produce packet

loss at the switch. Finally we configured Bilbo and Smaug as MGEN sources and Gollum and Thorin as MGEN sinks in an effort to introduce traffic in both directions. We ran this configuration at bursts of 30 MB/Sec, and we were unable to cause the switch to fail. We ran this configuration at bursts of 70 MB/Sec and saw 5% packet loss. This means that our Gigabit switch failed to mirror 5% of the traffic when we pushed 1.12 Gigabits over the network. A reasonable conclusion is that, at least for the equipment that we used in a configuration typical for network intrusion detection, mirror failure is not a significant problem.

Experiment 2 Host Based Packet Loss.

In order to test host based packet loss we configured our sensor with tcpdump to collect the network traffic. Using tcpreplay as we did above to exercise the switch we collected the information in Table 4. We noticed that if we asked tcpdump to do any analysis at all on the traffic, we started to see packet loss. From this we conclude that the hardware and operating system of the components that we used were capable of handling the traffic we were able to generate until userprocesses began to consume resources.

Run	Multiplier	Time (sec)	TimeRatio	PktsReceived	PktLoss
1	200	17.73	0.985	1,340,209	0
2	250	14.22	0.988	1,340,212	0
3	300	11.93	0.994	1,340,212	0
4	600	6.43	1.072	1,340,212	0
5	1,000	4.53	1.258	1,340,212	0
6	1,200	4.19	1.397	1,340,245	0
7	1,400	3.94	1.532	1,340,212	0

Table 4 Network Packet Loss Results

Experiment 3 Sensor Based Packet Loss.

Sensor based packet loss may occur when the NID software takes so much time to process the packets that they saturate the buffer and packets are dropped. In order to characterize this we will install the Snort sensor on the system and observe the packet loss. Replaying the hour from CDX 2009 at different rates we are able to show how Snort loses the ability to capture packets as the data rate increases as graphed in Figure 6. The left hand axis is the number of packets. The bottom axis is time adjusted to account for speeding up the process**;** for example the capture run at 250 times the speed of the original traffic is plotted as if it took the same amount of time as the original traffic**,** even though it actually took considerably less time. This was done to better show the relationship between the original and accelerated data streams. This graph shows that

Snort has a fixed limit to how many packets it can process. This graph looks very similar to the graphs of the capped algorithm that we will see later in Figures 20 and 21.

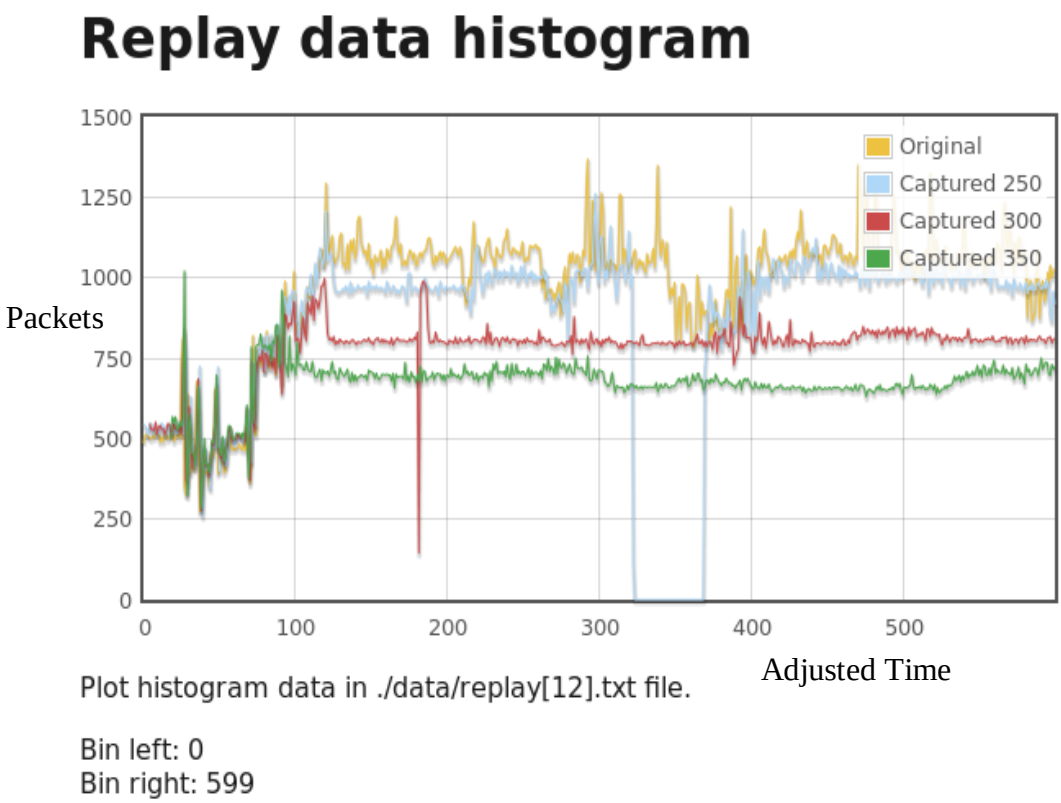

Figure 6 Packet Loss as Measured by Snort

Snort is capable of capturing packets to a file, or analyzing packets against a rule set looking for malicious network activity. These results were obtained running Snort in capture mode because we are unable to graph packet loss when running in analysis mode. Although, we expect packet loss to be greater when Snort is run in analysis mode; however, at the time we are unable to measure this. To measure it one would need to create a tool similar to tcpreplay that would replay a PCAP file at a multiple of its original speed. It would check the system clock before it writes each packet and if the

time to write that packet is in the past, it would send that packet into a bit bucket instead of sending it to snort. In this way we would have a record of which packets were lost for further analysis.

Modeling Packet Loss

Requirements

The Packet Dropper was built in order to model packet loss. This software has the following requirements:

1. The Packet Dropper must be able to read and write network traffic in a format that is compatible with available datasets and network intrusion detection software.

2. The Packet Dropper must be able to execute in a batch mode compatible with the X-Wray Statistics and Performance Explorer (X-Wray SPEX). The X-Wray SPEX was built for a previous study of the efficiency of NIDS.

3. The Packet Dropper must facilitate the addition of new algorithms for dropping packets.

4. The Packet Dropper must support a sufficient number of diverse dropping algorithms to ensure that research question 3 may be answered.

Pcapcat

In order to satisfy requirement number 1, we selected the PCAP format used by tcpdump**,** a popular Open Source command-line packet analyzer (Welcome). The initial version of the Packet Dropper simply reads packets from a file in PCAP format and writes the packets out to a file in PCAP format. It is useful for our purposes to be able to

concatenate several files in PCAP format. Each file in PCAP format begins with some global data; therefore, tools like cat will not work. We decided to expand this phase just to implement a tool that would allow us to concatenate these files. We understand that tools like tcpsplice and mergecap will do this job; however, we wanted the functionality incorporated into the packet dropper, and the exercise provided a good first step.

Upon initial consideration this would seem to be a simple algorithm implemented with the following pseudo code:

```
Open output file for writing;
For each input file {
     Open input file for reading;
     While read a packet from the input file {
          Write a packet to the output file;
     }
     Close input file;
}
Close output file;
```

The problem is that information from the input file, specifically the cap length, is necessary to properly open the output file; therefore the pseudo code looks more like this:

```
For each input file {
     Open the input file;
     If the output file is not  opened {
          Open the output file with information from the
first input file;
     }
     While reading a packet from the input file is
successful {
          Write a packet to the output file;
     }
     Close input file;
}
Close output file;
```

User Interface

Since the Packet Dropper will need to be executed in batch mode, a command line user interface was selected. It was also considered useful to be able to configure the

Packet Dropper from a configuration file or through the environment. The Configuration library was written to facilitate this. The Configuration object contains a data structure of Configuration Items which provide specific information about how to configure each item. See Figure 7 for the Configuration class diagram.

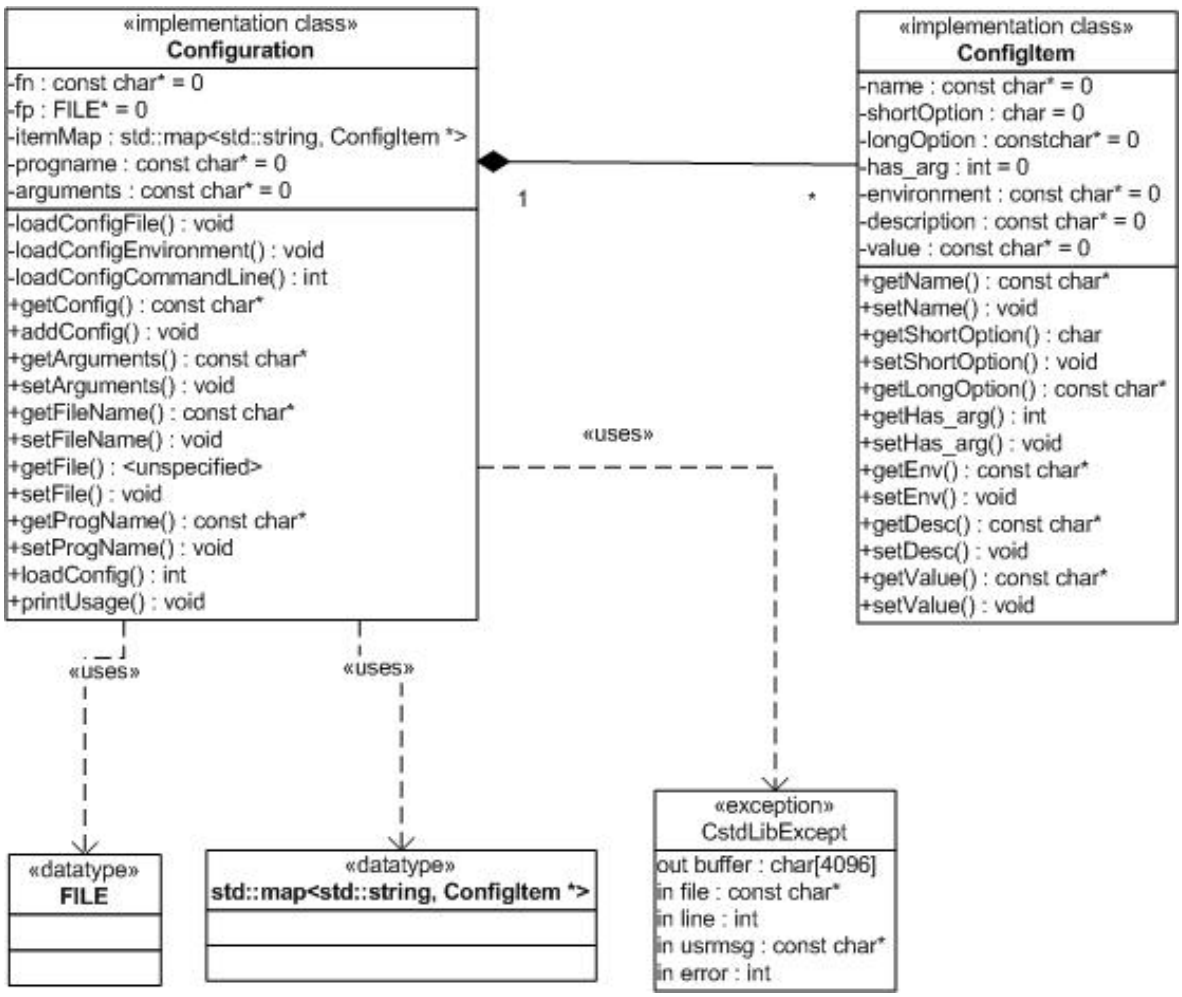

Figure 7 UML Class Diagram for Configuration Objects

Using the Configuration library first the Configuration object must be created and configured then ConfigItem object are created, configured, and added to the Configuration object. Once this is complete the user would invoke the loadConfig method passing argc, and argv from the command line. Then call getConfig("*ci name*") whenever the value of a configuration item is required. The biggest problem with this is

that we want the values specified on the command line to override values specified in the environment or in a configuration file, and we want the values specified on the environment to over ride values specified in the configuration file. So what happens if we specify a configuration file on the command line? The Packet Dropper processes the new file, and then processes the environment and the command line again.

The Deterministic Algorithm

First we implemented a very simple algorithm that evenly distributes packet loss across the dataset. We did that as a control to help discover how dependent ALR is upon the way the packets are dropped.

At this point we introduced the Dropper class. Dropper objects contain all of the parameters necessary to implement a dropping algorithm plus the methods to manipulate them. The Dropper class contains a virtual dropit() method which takes a pcap packet header as an argument and returns true if the packet is to be dropped or false if the packet is to be kept. Each dropping algorithm is implemented as subclass of the Dropper class and implements the dropit() method. Looking ahead we notice that the deterministic and the bounded random algorithms are both based upon dropping over an interval and that the sinusoidal and sawtooth algorithms are both based upon a function; therefore, we will implement the subclasses IntervalDropper and FunctionDropper to capture the similar parts of those algorithms. See Figure 8 for the Dropper class diagram.

Originally this algorithm was called simply “even.” Very late in this process the name was changed to deterministic because we felt that name more clearly describes the algorithm; however, artifacts of the original name still remain especially in the Packet Dropper application. For example the Class which implements the deterministic

algorithm is still called the EvenDropper, and in order to invoke the deterministic algorithm one would use the option "–`algorithm even`" not "–`algorithm deterministic`."

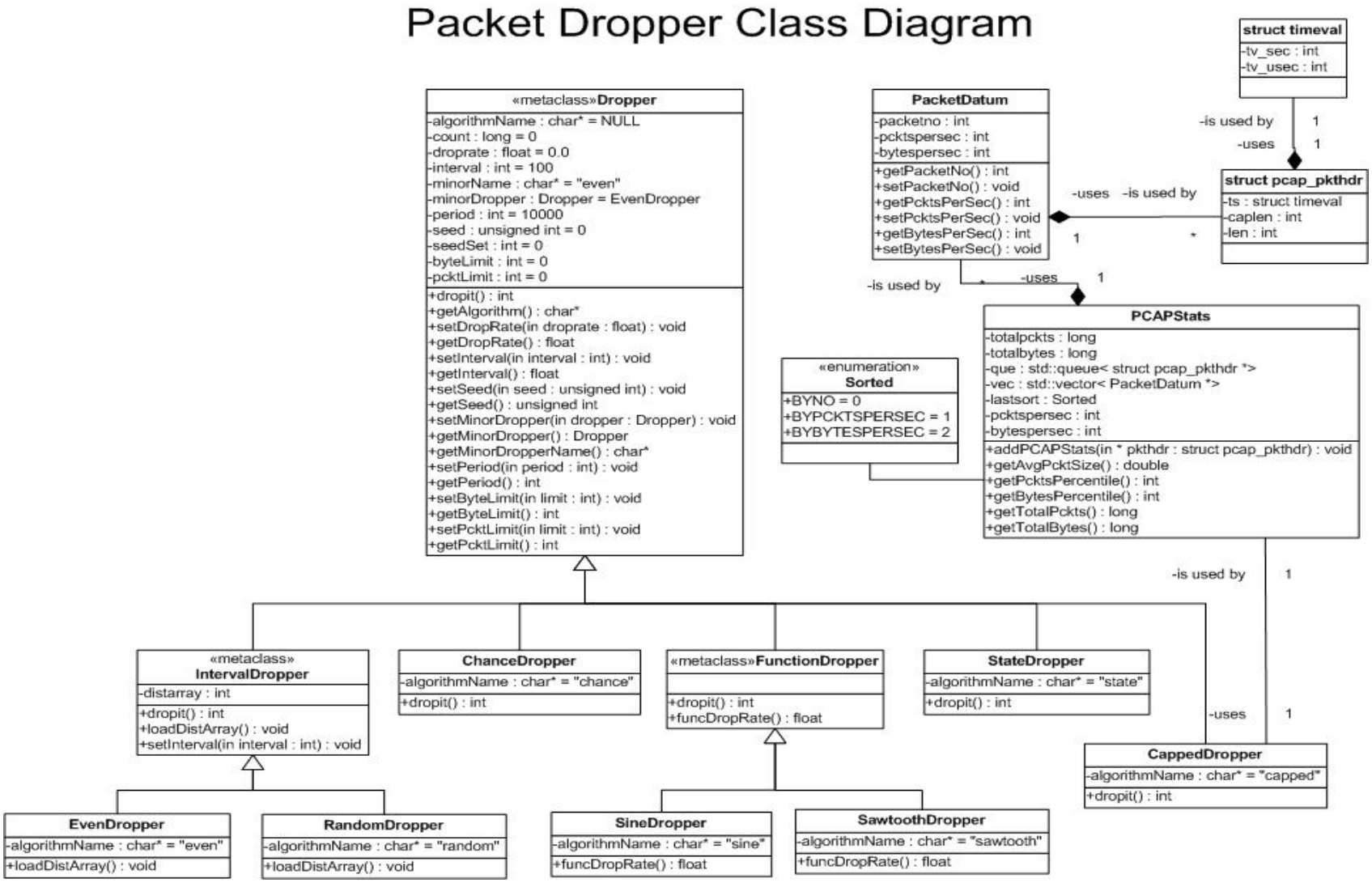

Figure 8 Dropper Class Diagram

The interval based dropping algorithms do their work with two methods. The first method, loadDistArray(), loads a distribution array of interval length with either zero if the packet is to be kept and a one if the packet is to be dropped. The second method, dropit(), keeps a running count of packets and returns the value in the distribution array of the packet count modulo the interval. All of the interval class of dropping algorithms share the same code for the dropit() method; therefore it is implemented in the IntervalDropper class. Also the setInterval() method is overridden to create the distArray. Each of the different algorithms then only have to implement the loadDistArray() method. Below is the pseudo code for the dropit() method shared by all of the IntervalDropper subclasses:

```
Dropit( header ) {
      If (count % interval == 0) {
            loadDistArray();
      }
      retval = distarry[count % interval];
      count++;
      return retval;
}
```

The deterministic algorithm drops packets evenly through the stream at a given rate expressed as a percentage. For example: If the drop rate is set at 1%, PacketDropper will drop every 100th packet. If the drop rate is set at 2%, PacketDropper will drop every 50th packet. If the drop rate is set at 3%, PacketDropper will drop every 33rd packet. Below is the pseudo code to implement the loadDistArray() method for the deterministic algorithm:

```
loadDistArray() {
      Set all elements of the DistArray to zero;
      Compute how many packet to drop, todrop, by
multiplying the droprate times the interval;
      If the todrop is greater than zero {
```

```
            Compute the dropstep by dividing the interval by
todrop;
            For (i = dropstep -1; I < interval; I += dropstep
{
                  Distarray[i] = 1;
            }
      }
}
```

To illustrate the effects of these different algorithms on network traffic, we have graphed the Mb/second and packets/second of about five minutes of network traffic from CDX 2009. These five minutes were selected because the traffic was consistent enough for the peaks and valleys not to obscure the results. Looking at Figure 9 (Mb/second) and Figure 10 (packets/second) we can see that the deterministic dropping algorithm produces a line almost identical to the original line a little lower on the graph which is exactly the behavior that we would expect. This is very similar to what Schaelicke & Freeland observed (see Figure 4).

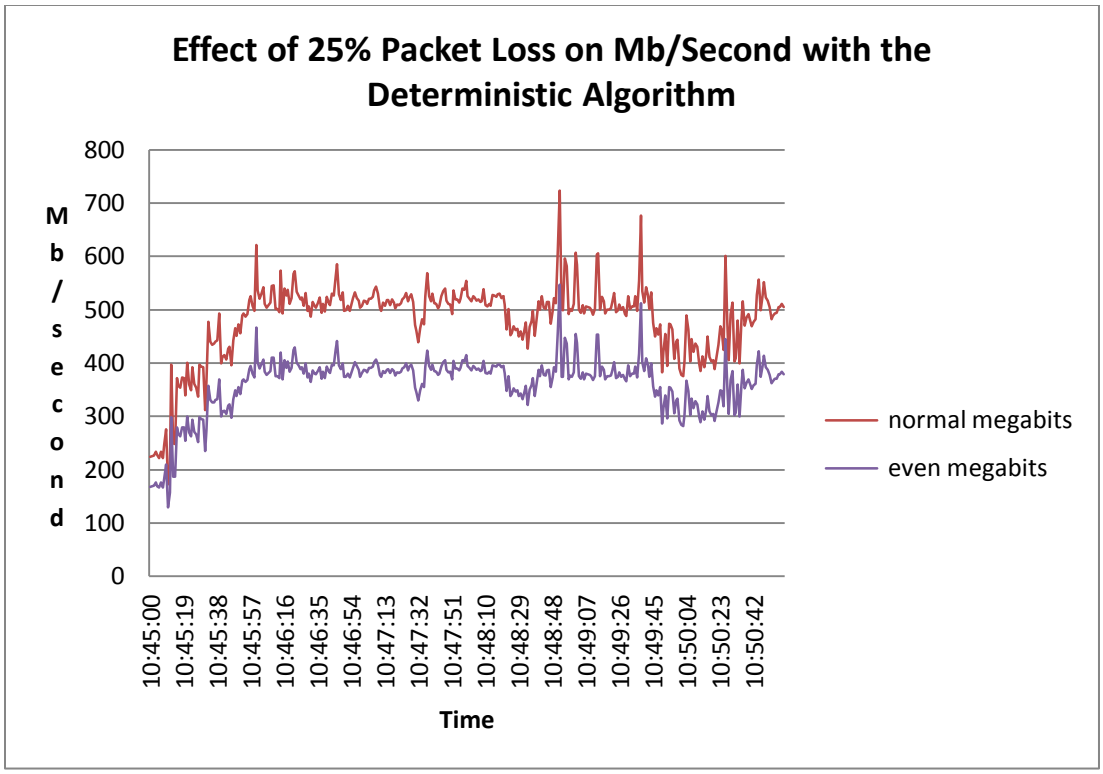

Figure 9 Effect of 25% Packet Loss on Mb/Second with the Deterministic Algorithm

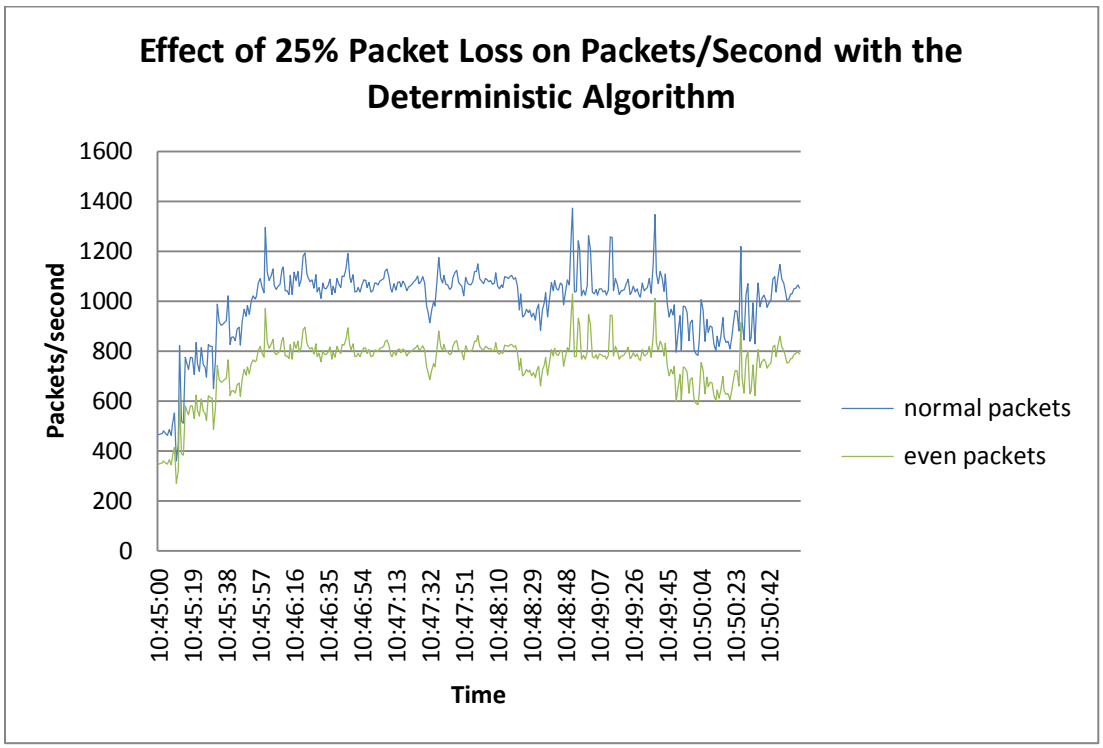

Figure 10 Effect of 25% Packet Loss on Packets/Second with the Deterministic Algorithm

The Bounded Random Algorithm

The next phase of the PacketDropper application adds the ability to drop packets randomly over an interval. This function is provided a user defined interval and it will drop random packets within this interval to meet the drop rate. Below is the pseudo code for the bounded random loadDistArrary() method:

```
loadDistArray() {
     Set all elements of the DistArray to zero;
     If the seed has not been set, set the seed
     Compute how many packet to drop, todrop, by
multiplying the droprate times the interval;
     For (i = 0; i < todrop; i++ {
          randnum = random();
          if (distarry[randnum % interval] == 0) {
               distarry[randnum % interval] = 1;
          } else {
               i--; // collision try again
          }
     }
}
```

As we see in Figures 11 and 12 over the same section of network traffic we get graphs that look very similar to the graphs we saw using the deterministic algorithm. This should not surprise us since we are dropping the same number of packets per interval; we are simply dropping different packets. Interestingly, increasing the interval does not seem to have a major effect. We used an interval of 10,000 for Figures 11 and 12.

Originally this algorithm was called simply "random." Very late in this process the name was changed to "bounded random" because we felt that name more clearly describes the algorithm; however, artifacts of the original name still remain especially in the Packet Dropper application. For example the Class which implements the bounded random algorithm is still called the RandomDropper, and in order to invoke the bounded

random algorithm one would use the option "`–algorithm random`" not "`--algorithm bounded random`."

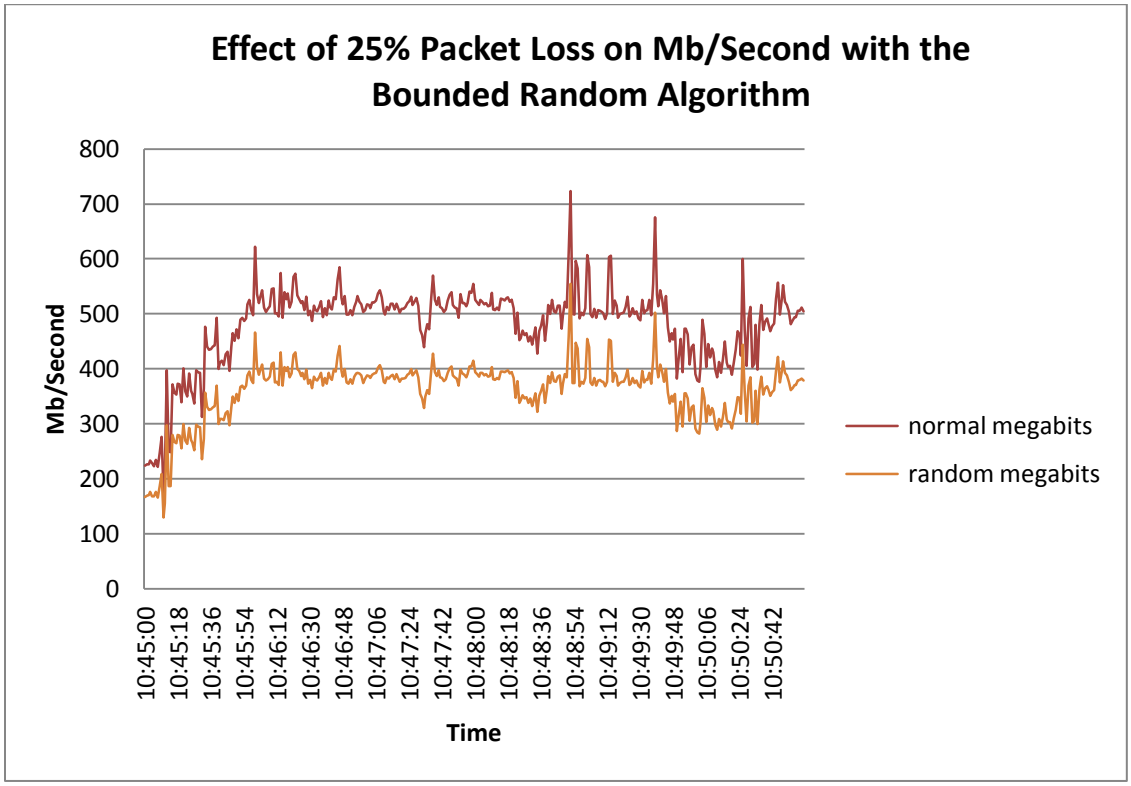

Figure 11 Effect of 25% Packet Loss on Mb/Second with the Bounded Random Algorithm

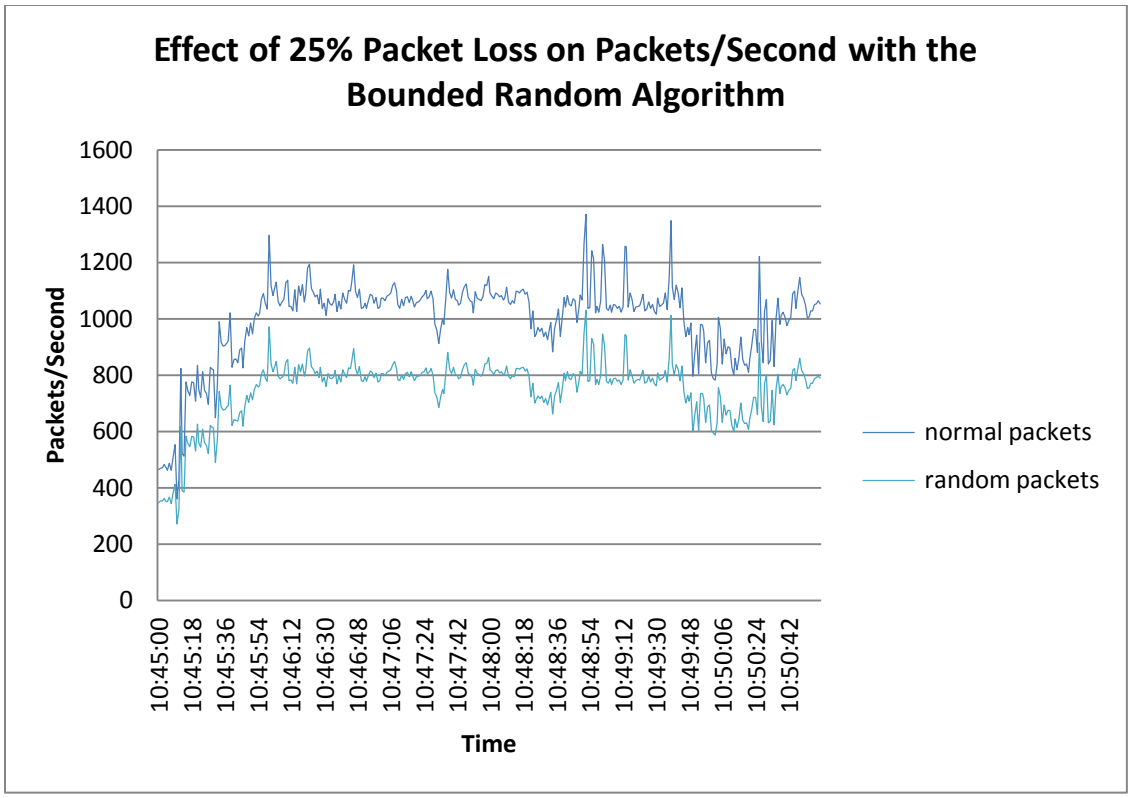

Figure 12 Effect of 25% Packet Loss on Packets/Second with a Bounded random Algorithm

The Random Algorithm

The random algorithm generates a random number for each packet read. If the random number is below the drop rate we drop the packet, otherwise we keep the packet. Below is the pseudo code for the dropit() method.

```
dropit() {
      If seed is not set
            Set the seed;
      randnum = random();
      Return (droprate * 100 > randnum % 100);
}
```

Looking at Figures 13 and 14 we can see that the random algorithm does not look all that different from the deterministic or bounded random algorithm.

Originally this algorithm was called simply “chance” to distinguish it from the algorithm originally called “random”. Very late in this process the name was changed to

"random" because we felt that name more clearly describes the algorithm; however, artifacts of the original name still rename especially in the Packet Dropper application. For example the Class which implements the random algorithm is still called the ChanceDropper, and in order to invoke the bounded random algorithm one would use the option "`–algorithm chance`" not "`--algorithm random`" as this would given one the bounded random algorithm instead.

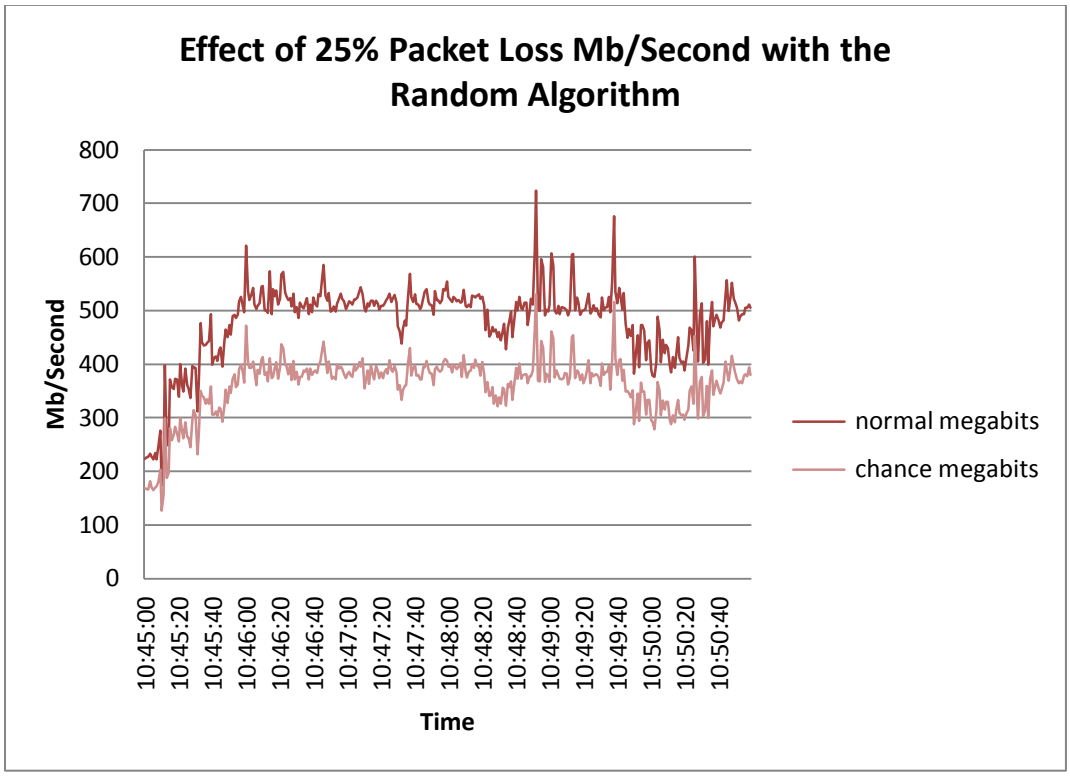

Figure 13 Effect of 25% Packet Loss on Mb/Second with the Random Algorithm

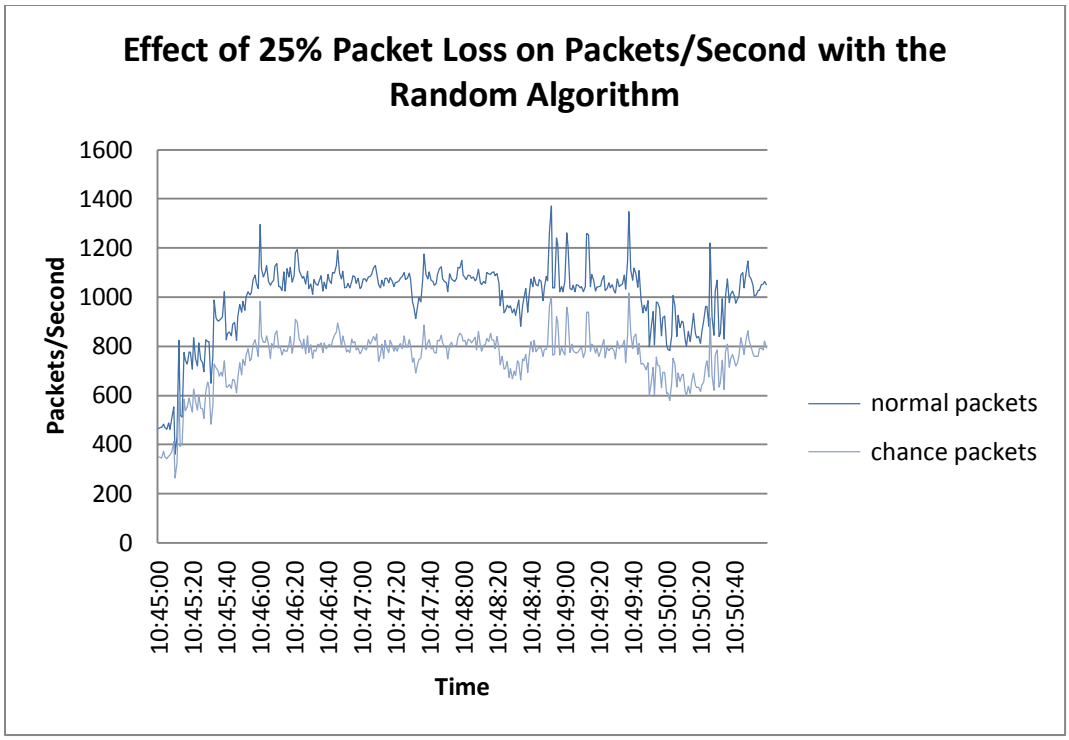

Figure 14 Effect of 25% Packet Loss upon Packets/Second with a Random Algorithm

The Sinusoidal Algorithm

At this point we introduce the concept of the function dropper. Function droppers divide a period of the function into intervals. For each interval the value of the function is computed for the center, and that value is used as the drop rate for that interval. Function droppers employ a minor dropping algorithm to drop packets along that interval. Since the input into the function is the packet count, the domain must include positive integers up to the maximum value for an integer. The range must be positive real numbers between 0 and 1 where the average value over a period is the drop rate.

The FunctionDropper class implements the dropit() method and establishes the virtual method funcDropRate() which will be implemented by the child classes. Below is the pseudo code for the dropit() class.

```
if we are at the end of an interval {
     compute the droprate using the funcDropRate() method;
     set the droprate of the minor dropper using the
setDropRate() method;
}

retval = minorDropper->dropit( header );
increment the count;
return retval;
```

The sine function has a natural period of 0 to 2π in radians which the default unit for the sin() function in the C standard library. In order to convert this natural period into the defined period we will divide our current position, x, by the period and multiply it by the end of the period for: $y = A \sin\left(\frac{X}{period} 2\pi\right)$. We want to compute the value of the center of next interval giving us: $x = count + \frac{interval}{2}$. Now the natural range of the sine function is from -1 to 1, but we need a range from 0 to 1 with a center at the drop rate. If we set the amplitude of the sine wave to the drop rate then add the adjusted sin value to the drop rate we get a function that moves from 0 to 2 times the drop rate centered around the drop rate. This is perfect for drop rates less than 50%. For drop rates greater than 50% the upper bound is outside of our range. To account for this if the drop rate is greater than 50% we use an amplitude of 1 minus the drop rate. This amplitude adjustment is depicted in Figure 15 which plots the sine wave used to adjust the drop rate for drop rates of 25%, 50%, and 75%. Notice that the amplitude of the wave is smaller the farther we get from 50% on both ends in order for the results of the function to remain within the range.

Originally this algorithm was called simply “sine.” Very late in this process the name was changed to “sinusoidal” because we felt that name more clearly describes the algorithm; however, artifacts of the original name still rename especially in the Packet

Dropper application. For example the Class which implements the deterministic algorithm is still called the SineDropper, and in order to invoke the bounded random algorithm one would use the option "`--algorithm sine`" not "`--algorithm sinusoidal`".

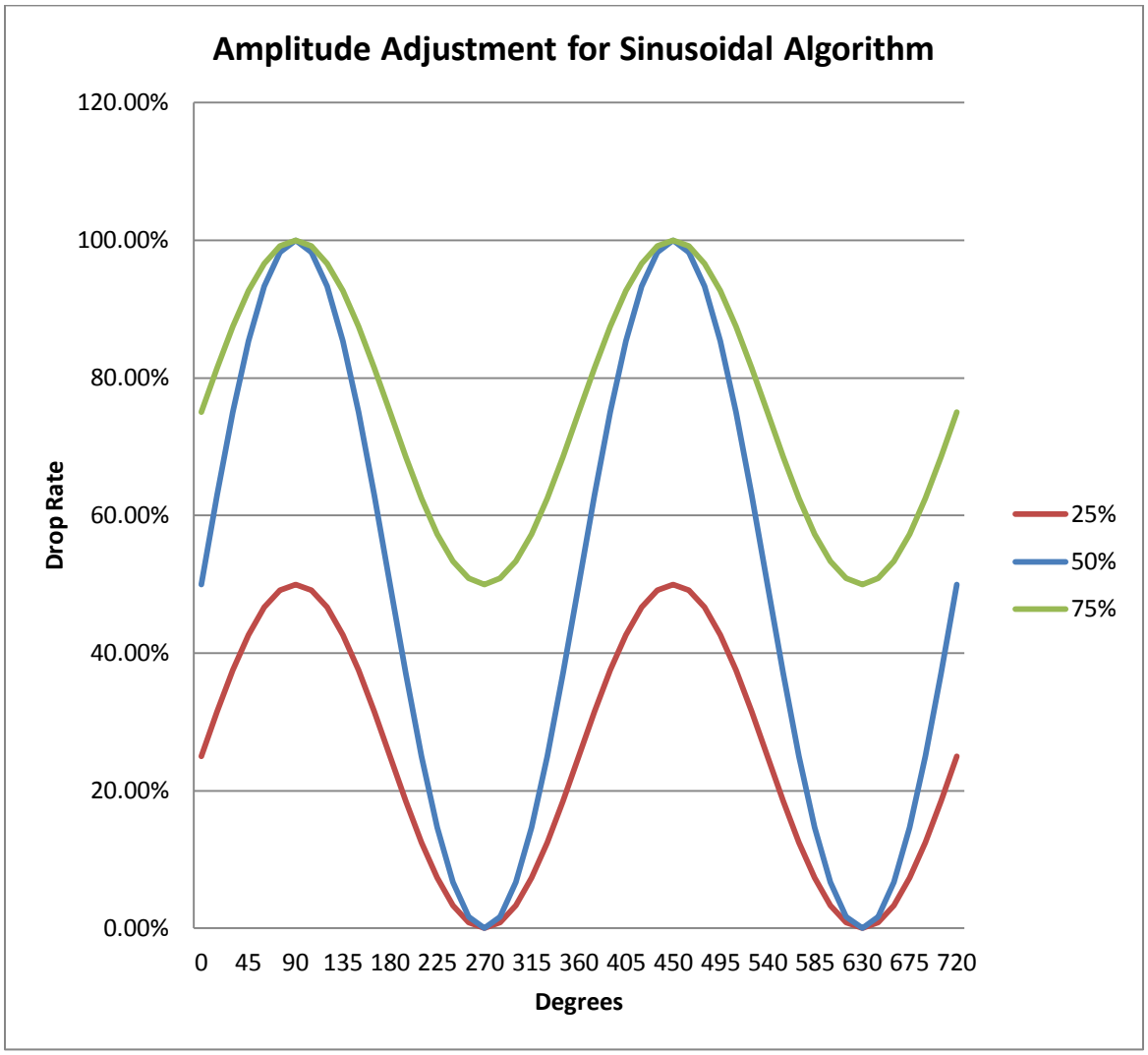

Figure 15 Amplitude Adjustment for the Sinusoidal Algorithm

The pseudo code for this method is given below:

```
Let x = count + interval/2;
Let y = sin(X/period 2π);
Let a = (droprate < 0.5)? droprate: 1 − droprate;
```

```
Let z = droprate + ay;
Return z;
```

Looking at Figures 16 and 17 which plots the effect of 25% packet loss using the sinusoidal algorithm on Mb/second and packets/second, we can clearly see the wave.

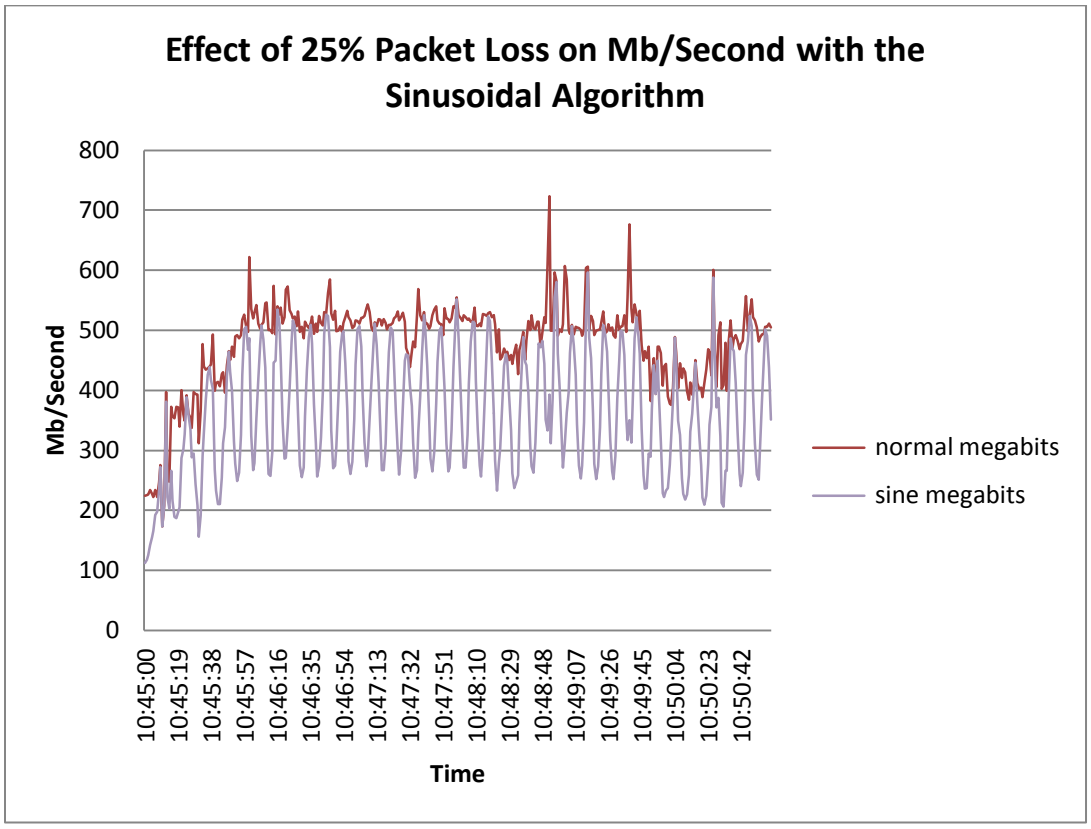

Figure 16 Effect of 25% Packet Loss on Mb/Second with a Sinusoidal Algorithm

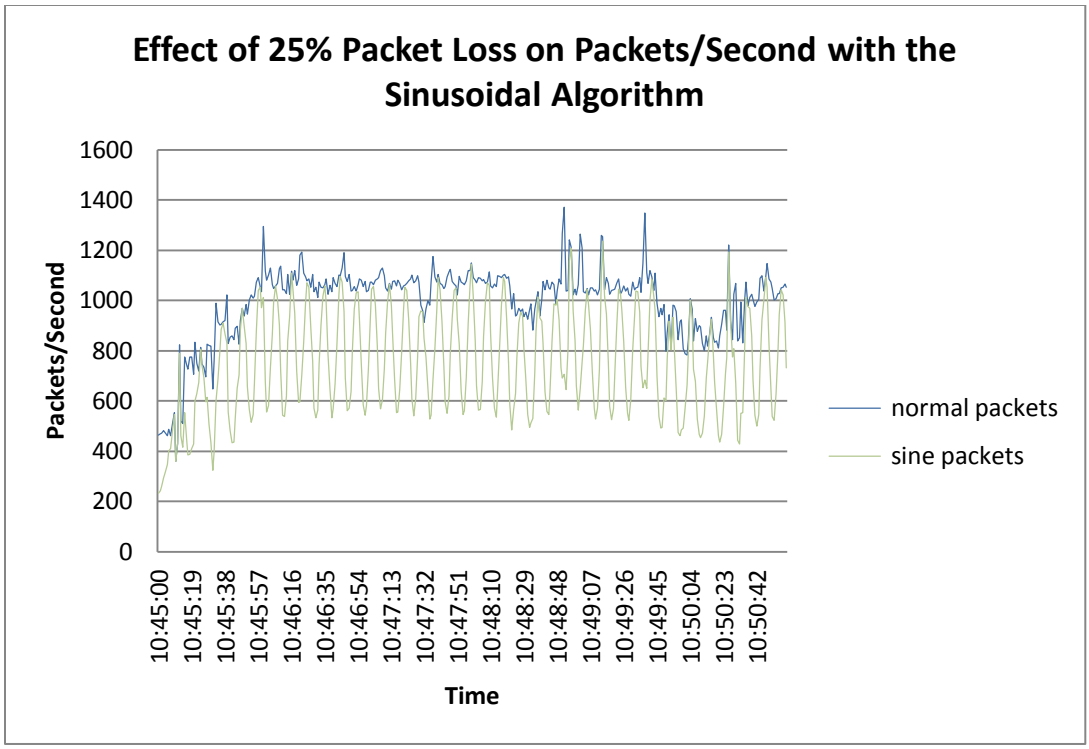

Figure 17 Effect of 25% Packet Loss on Packets/Second with a Sinusoidal Algorithm

A Sawtooth Algorithm

The sawtooth algorithm uses the following function which has domain of any positive integer and a range of negative one to positive one.

$$y = 2\left(\frac{count}{period} - \left\lfloor\frac{1}{2} + \frac{count}{period}\right\rfloor\right)$$

Like the sinusoidal algorithm we will adjust the amplitude to ensure that drop rate stays between zero and one hundred percent.

The pseudo code to implement this algorithm is very similar to the code for the sinusoidal algorithm.

```
Let x = count + interval/2;
Let y = 2(x/period − ⌊1/2 + x/period⌋);
Let a = (droprate < 0.5)? droprate: 1 − droprate;
Let z = droprate + ay;
Return z;
```

Looking at Figures 18 and 19 plotting the effect of this algorithm at a drop rate of 25% on Mb/second and packets/second, we can clearly see the sawtooth in the much sharper lines than we saw with the sinusoidal algorithm.

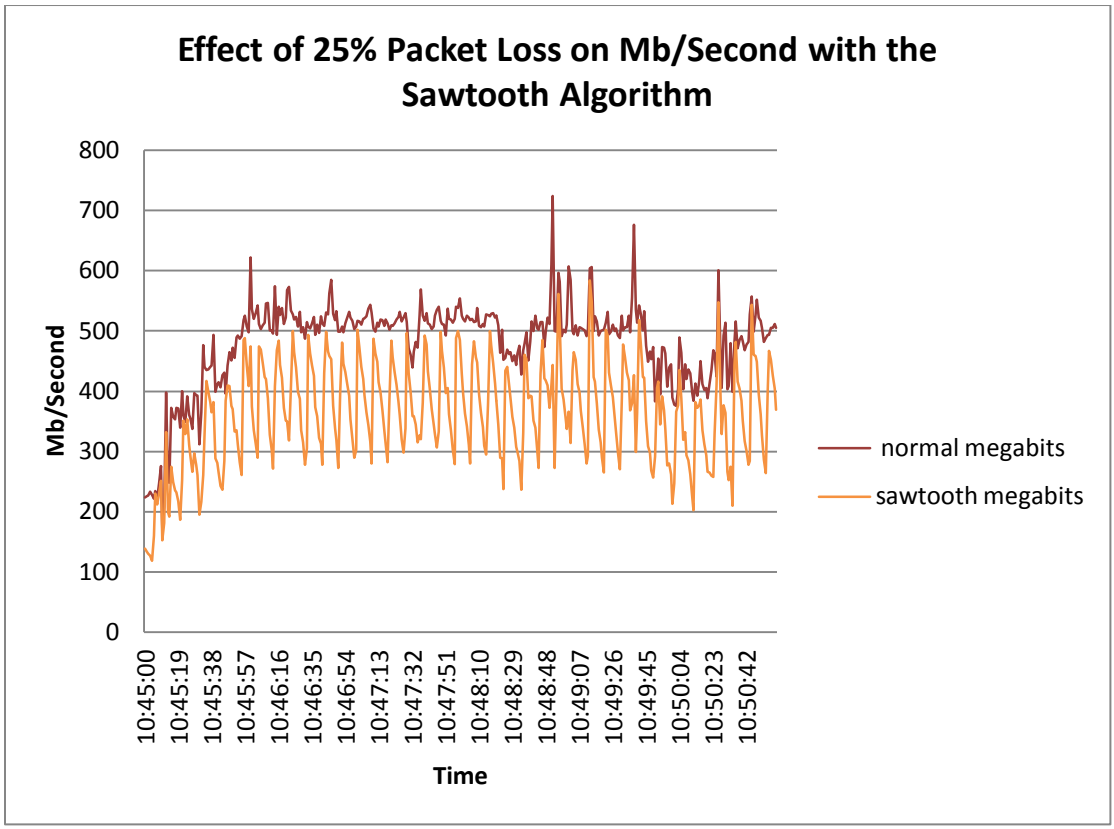

Figure 18 Effect of 25% Packet Loss on Mb/Second with the Sawtooth Algorithm

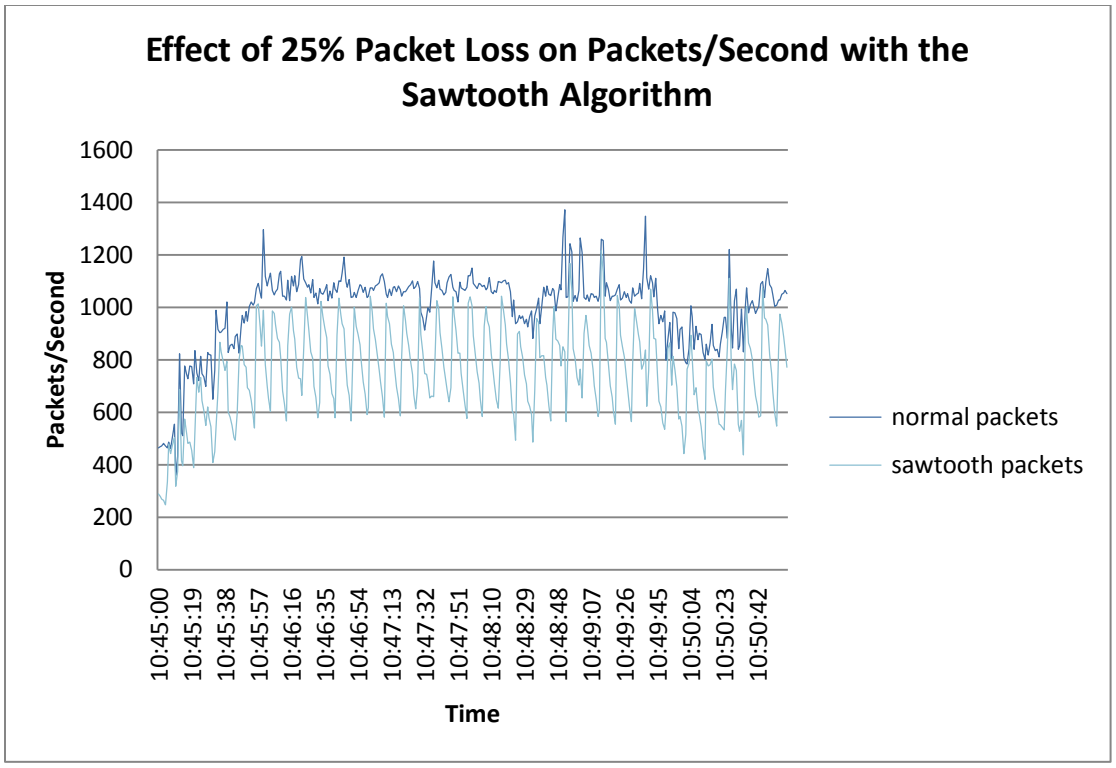

Figure 19 Effect of 25% Packet Loss on Packets/Second with the Sawtooth Algorithm

Pcapgraph

For the next phase of the Packet Dropper, we need to be able to keep track of the current Mb/second and packets/second so that we can drop packets when we have reached a cap. The Pcapgraph tool, which provides these measures for each second of the PCAP file, is a logical first step. We know that there are other tools that will graph the flow of a PCAP file; however, this was deemed a reasonable exercise because the lessons learned will be incorporated into the capped dropping algorithm.

The tricky part is the interior while loop. This ensures that if there are empty spots in the traffic, and we do not see a packet for an interval, we will write an entry with zero packets and zero Mb covering that interval. The pseudo code looks like this:

```
While we can get the next packet {
     If current is zero {
          Set current to packet time plus the interval;
```

```
            Set packets and megabits to zero;
        }
        While (current < time of the packet) {
            Write Date, Time, Packets, Megabits;
            Increment current by the interval;
            Set packets and megabits to zero;
        }
        Increment the packet count by one;
        Add the current packet size to megabits;
    }
```

Pcapstats

Pcapgraph is a good first step; however, there are a few more issues that need to be solved before we are ready to write the capped dropper. Pcapgraph computes the packets and Mb for a discreet interval, but what we really want are the packets and Mb for a rolling interval. The natural units to control the capped dropper would be maximum number of packets per second or maximum number of bits per second; however, the other droppers are driven by the drop rate. In order to compare the other droppers to the capped dropper we need to be able to set the packet or bit limit such that we meet the given drop rate. Pcapstats is the program written to explore these problems, and it also provided useful information about PCAP files.

We need to be able to compute the number of packets per second and Mb per second over an interval at the precise moment when we receive the packet. The idea is to simulate a buffer that fills if the number packets/second or Mb/second exceeds the limit. We create a queue of PCAP packet headers. When we read a new packet, we drain the queue of any packets older than the current packet minus the interval decrementing the number of packets and bits. Then we add the new packet incrementing the packet count and adding the size of the current packet to the Mb.

A Capped Algorithm

The capped algorithm will need to implement a first in first out (FIFO) data structure for PCAP headers. When the algorithm is passed a new packet header, the FIFO will be drained of all packets that were received an interval before the current packet appropriately decrementing the packet count and the Mb. When this operation is complete the packet count and Mb are compared against the caps and if they are above the caps the packet is dropped. Below is the pseudo code for the dropit() method of the capped algorithm:

```
Retval = false;
If the packet limit is greater than zero then
     retval = current packets/sec > packet limit;
If retval == false && byteLimit > 0 then
     Retval = current bytes/sec > byte limit;
If Retval == False
     Add current packet to the queue;
Return retval;
```

For the drop rate to match the percentile we need to add all of the packets to the queue; however, this produces a graph with huge canyons wherever the packets or bytes per second is over the cap. The effect that we are trying to model would produce plateaus where the caps are reached. In order to do this we must not enter packets that we drop into the queue. If we do this, we can no longer use the percentile to compute the cap limit from the drop rate. The way we compute the correct limit for a given drop rate is to run through the data using a binary selection algorithm repeatedly until the correct cap is discovered. As we look at the graph in Figures 20 and 21, we can see the plateau in the traffic.

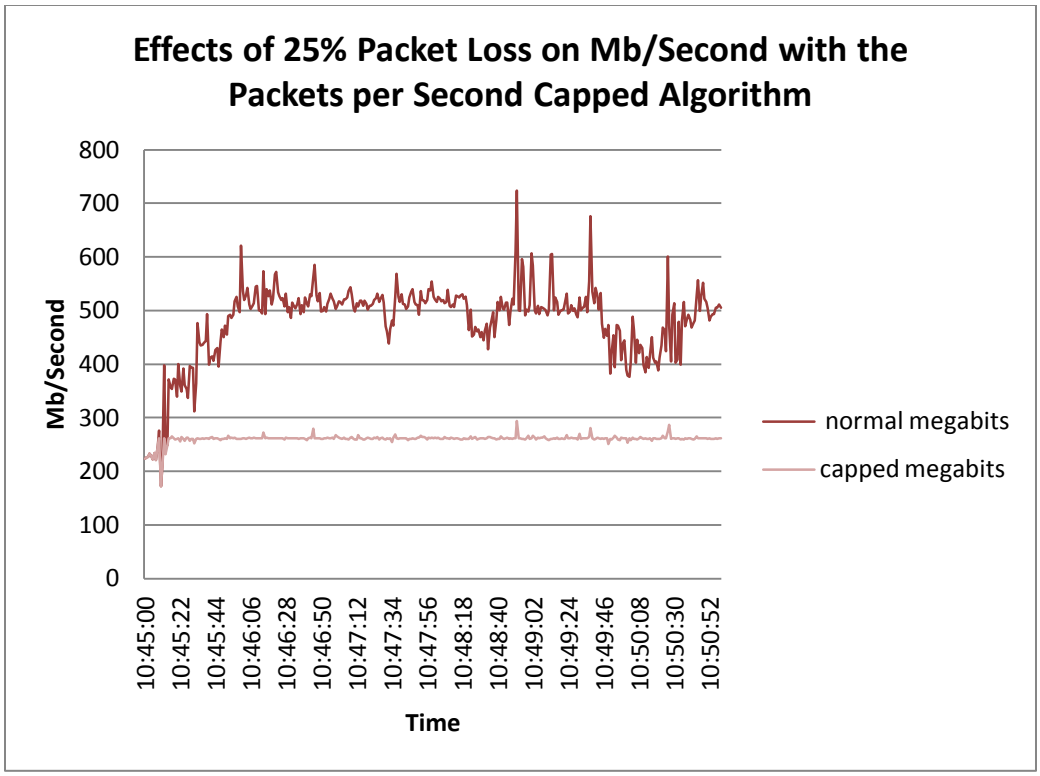

Figure 20 Effects of 25 % Packet Loss on Mb/Second with the Capped Algorithm

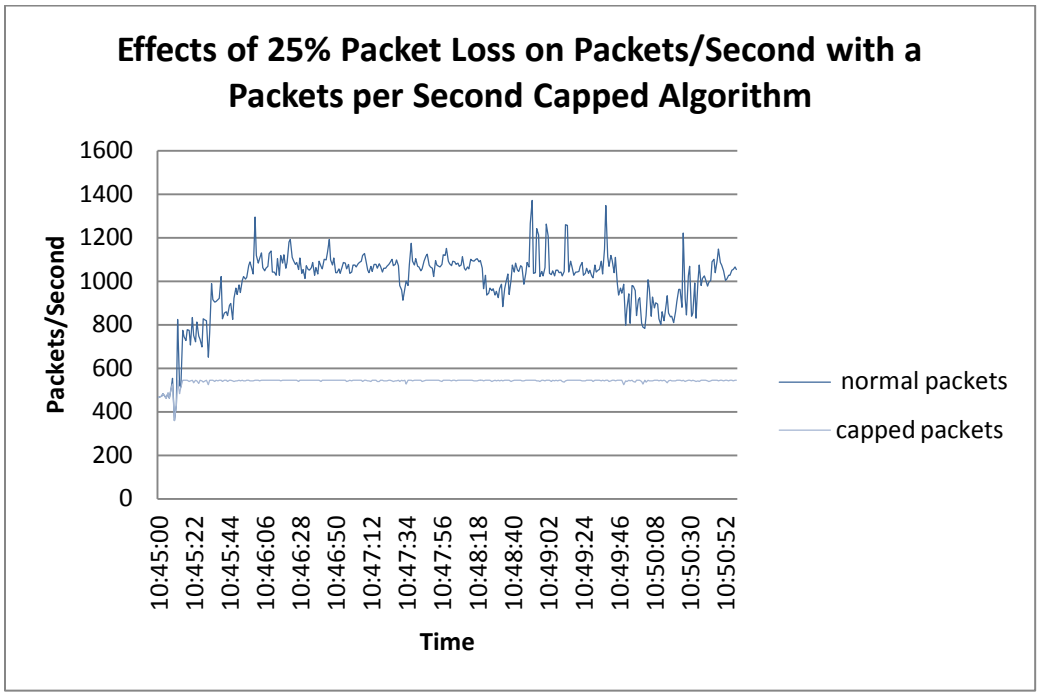

Figure 21 Effects of 25% Packet Loss with a Capped Algorithm on Packets/Second

A Two State Channel Model

The final theoretical algorithm is a based upon a Markov chain with two states used to generate bursts as described by Gilbert (Gilbert, 1960) and is illustrated in Figure 22 where state G is a good state where no dropping take place and state B is a bad state where packet dropping takes place. Let h represent the probability that a packet will be transmitted in state B. The value P represents the probability that we will transition from the good state to the bad state. The value p represents the probability that we will transition from the bad state to the good state.

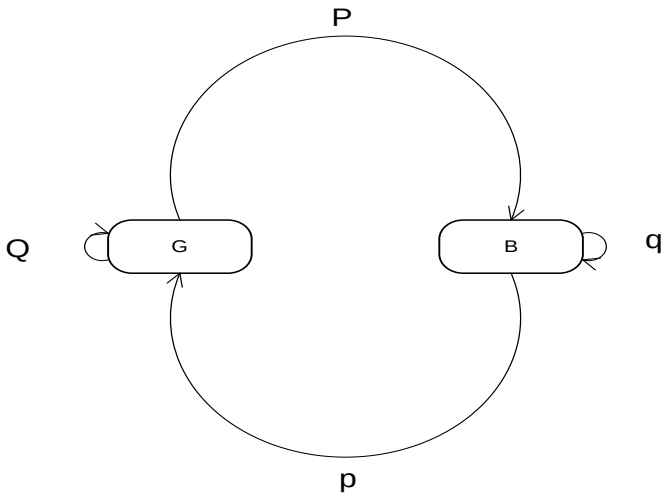

Figure 22 Two State Channel Model

Gilbert provides us with the following formula to calculate the error probability of the previous model letting d be the drop rate and P=P(Bad state|Good State) and Q = P(Good state|Bad state) ; *h* represents the probability that the a packet will transmitted in state *B* we have:

:

$$d = \frac{(1-h)P}{p+P}$$

Solving for P we get:

$$P = \frac{dp}{1 - h - d}$$

Gilbert used the following values P = 0.03, p = 0.25, h = 0.5. Since we want to vary the drop rate from 0% to 100% , those values will not work. To allow for the full domain yet ensure that the value of P stays within the range for 0 to 1 we need to set h = 0, and p = 0.05; The value of P is undefined when d = 1; therefore, we will add code to test for this condition and return true. Below is the pseudo code for this algorithm:

```
if (droprate == 1) {
     Return true;
}

if P equals zero {
     P = (droprate*p) / (1−h−droprate)
}

switch (state) {
case GOOD:
     retval = false;
     randnum = random number between 0 and 1;
     if (P > randnum) {
          state = BAD;
     }
     Break;
case BAD:
     randnum = random number between 0 and 1;
     retval = (h > randnum);
     randnum = random number between 0 and 1;
     if (p > randnum) {
          state = GOOD;
     }
     Break;
}
return retval;
```

In Figures 23 and 24 we can see the Mb/second and packets/second graphed for the state algorithm at 25% packet loss. Although the difference between the state and

deterministic algorithms is not as dramatic as the sinusoidal, sawtooth, or capped algorithms, it is quite different.

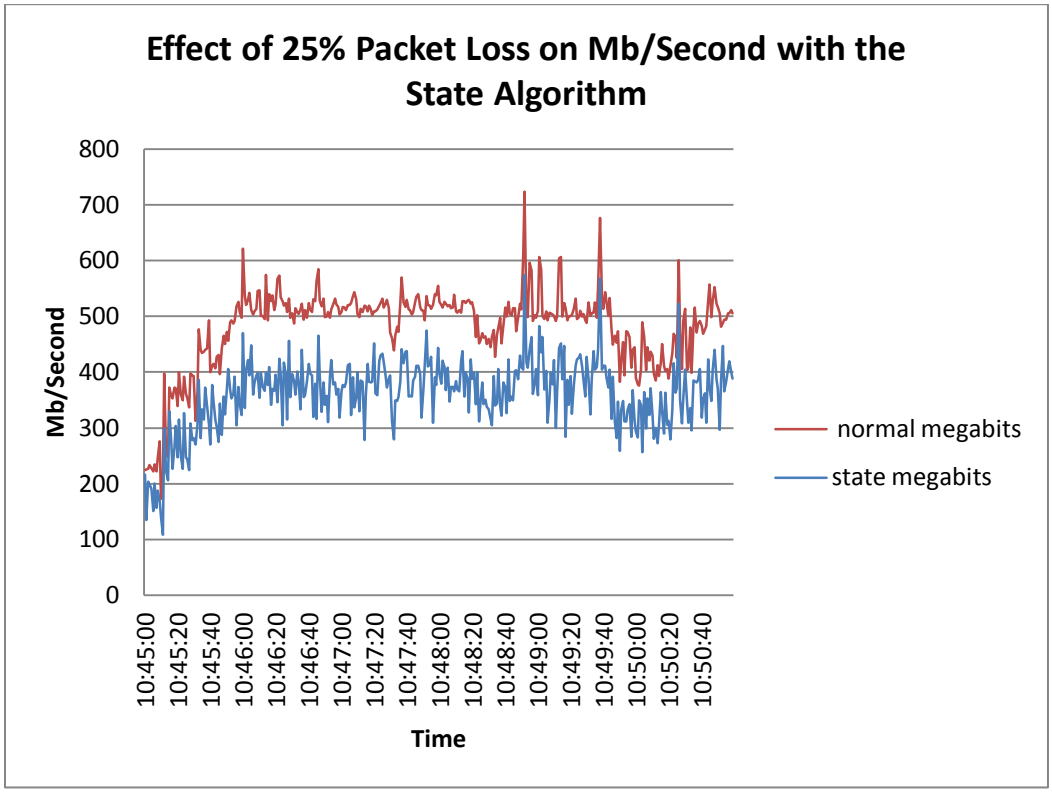

Figure 23 Effect of 25% Packet Loss on Mb/Second with the State Algorithm

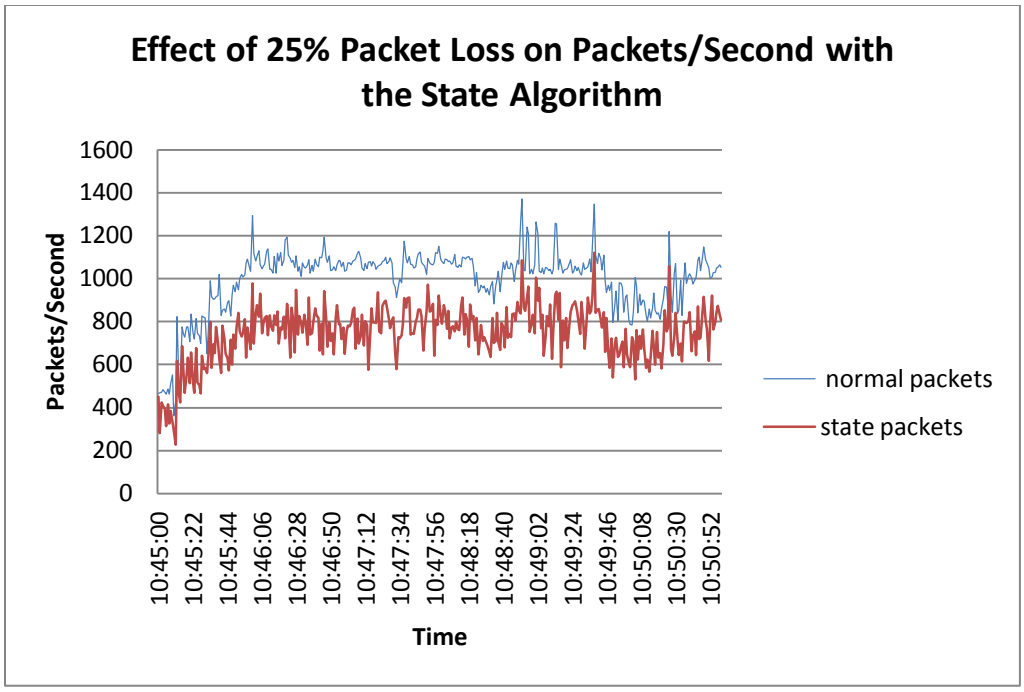

Figure 24 Effect of 25% Packet Loss on Packets/Second with the State Algorithm

Summary

In summary, we will review how well the packet dropper meets the requirements laid out above. Using the Open Source libpcap library allows the packet dropper to be compatible with most packet capture and network intrusion detection systems. The packet dropper has a robust interface that allows the user to configure it through a configuration file, the environment or the command line making it ideal for utilization in batch processing. The object oriented design of the packet dropper allows new algorithms to be added to the system requiring very little code outside the algorithm itself. The packet dropper currently supports seven algorithms which as we can see in Figures 25 and 26 provided a range of dropping options sufficient for us to determine if the manner in which packets are dropped has a direct bearing on the loss of intrusion alerts.

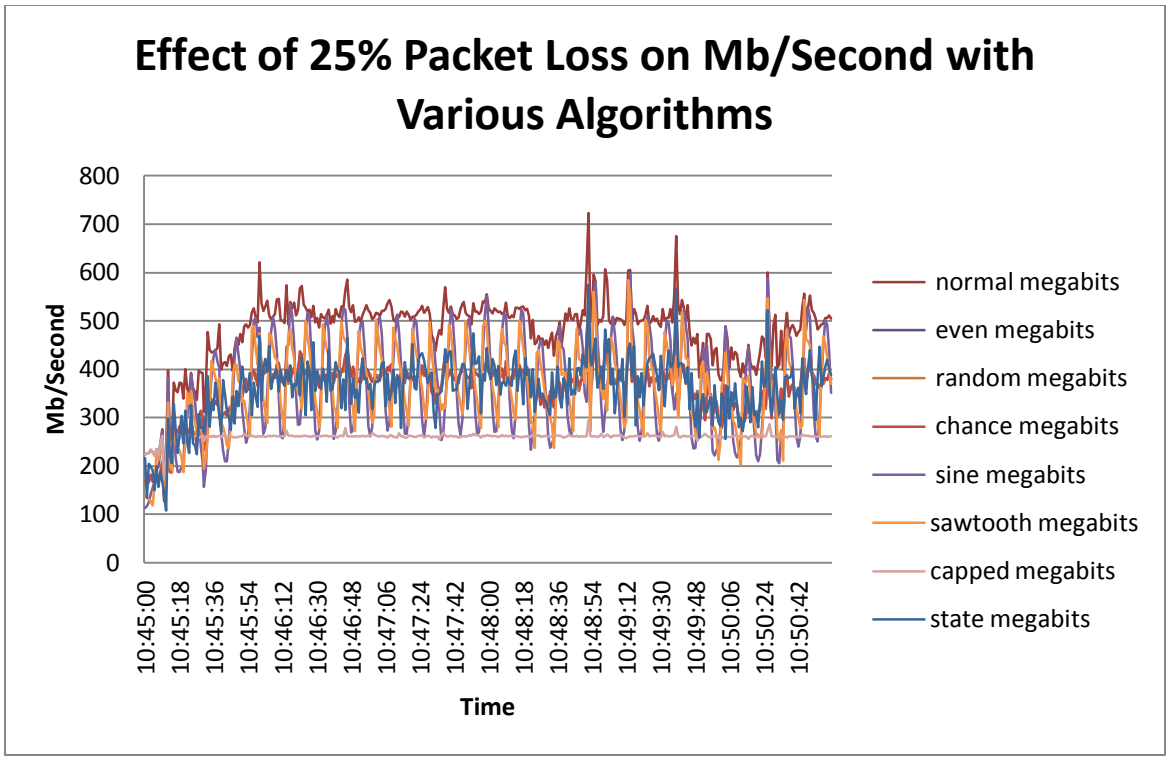

Figure 25 Effect of 25% Packet Loss on Mb/Second with Various Algorithms

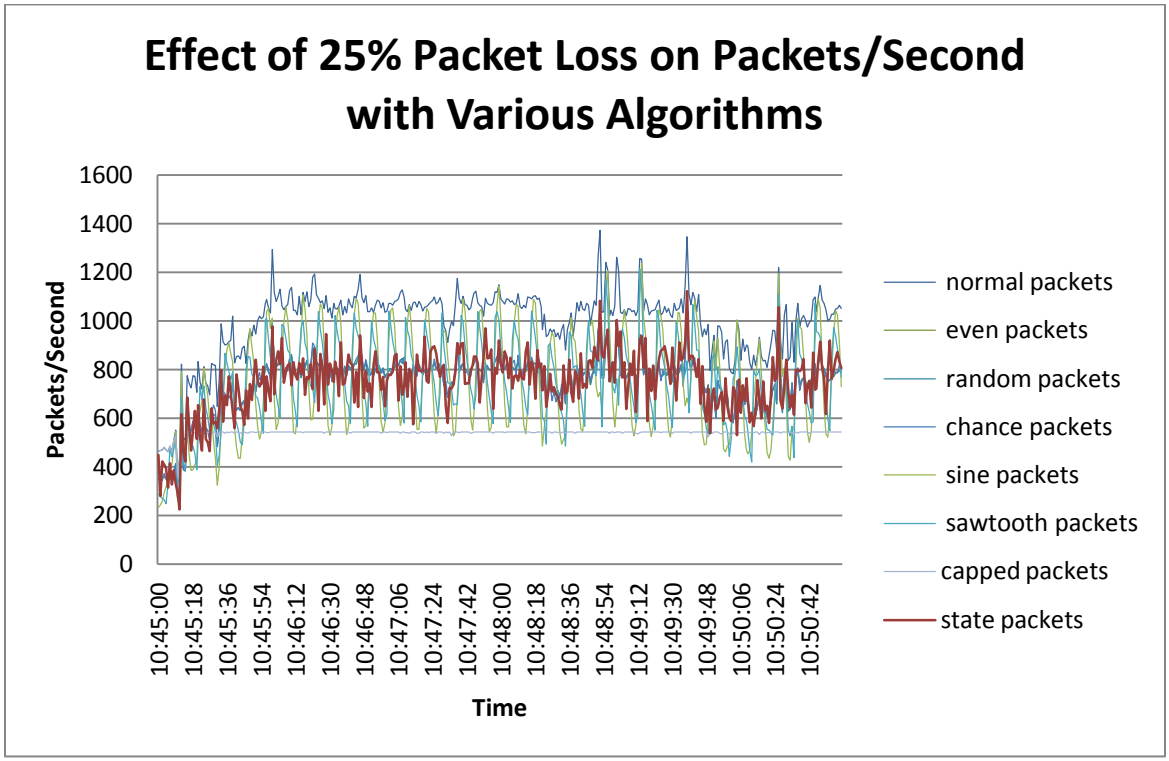

Figure 26 Effect of 25% Packet Loss on Packets/Second with Various Algorithms

Results

Validity vs. PCAP Datafile Size

Once the packet dropper's deterministic algorithm was working, we applied it to an hour of data from the CDX 2009 (Cyber Defense eXercise of 2009) dataset. Figure 27 is a chart mapping alert vs. drop rate for that hour which contained only 33 alerts. Notice the very irregular curve. We will see that as we look at more traffic containing more alerts this curve becomes significantly more regular. This implies that these results will not necessarily scale down well and should not be applied for small numbers of packets and small numbers of alerts.

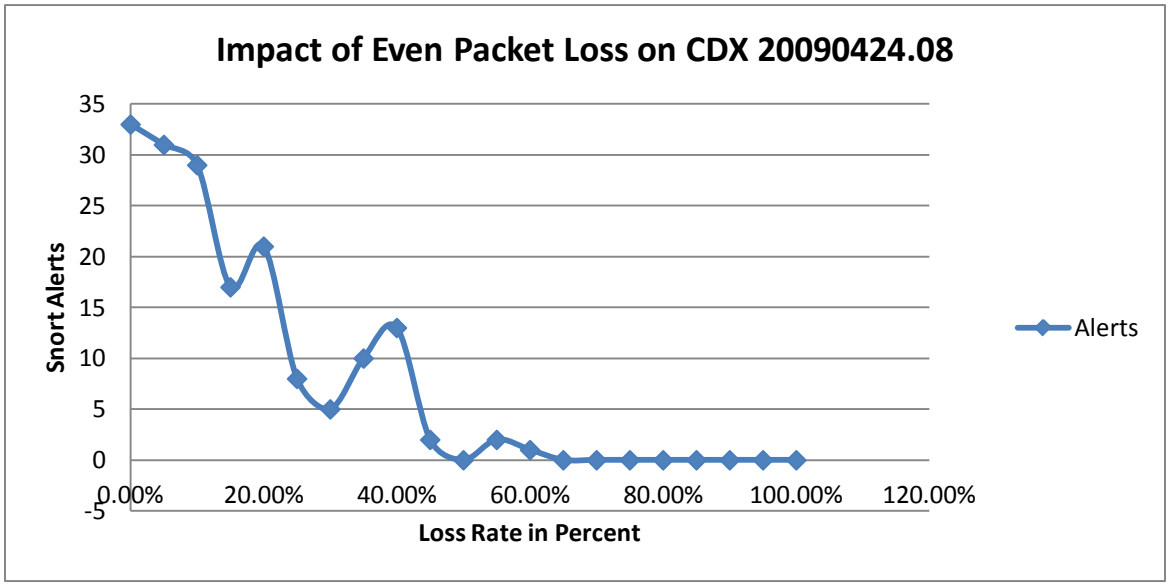

Figure 27 Impact of Deterministic Packet Loss on Alerts using the CDX 20090424.08 dataset

DARPA 1998 Training Dataset

In 1995 Rome Laboratory and the Department of Defense Advanced Research Projects Agency contracted with Massachusetts Institute of Technology's Lincoln Laboratory to conduct a study on Evaluating Intrusion Detection Systems (Lippmann, et

al., 2000). As part of their evaluation they created a training dataset that contained both malicious traffic and background traffic. Each session in the training data was labeled identifying whether it was part of an attack or not. The training dataset contains seven weeks of labeled data in PCAP format. The simulated network used to construct the 1998 training dataset was composed of UNIX workstations. Although this dataset is over twenty years old, it is still one of the best fabricated datasets available for research in intrusion detection (Brugger & Chow, 2007). The content of network traffic has changed significantly since 1998 with the advent of Web 2.0 and very popular sites like Facebook launched in 2004, YouTube launched in 2005, and Internet Video streaming like the NetFlix "Watch Instantly" service launched in 2008. One of the reasons that we find this data set so interesting for this research is we want to see if our findings about the impact of packet loss will still be valid several years after the research is completed, or will the experiments need to be repeated as the character of network traffic changes. This dataset contains 40,667,322 packets and about 16 Gigabytes of data. Snort detected 5,165 alerts for a distribution of about one alert for every 7,900 packets.

The charts in Figures 28 and 29 were created by running snort 2.6.8 against the dataset using the various dropping algorithms and drop rates ranging from 0% to 100% in 5% intervals. The first chart graphs raw alerts versus PLR. The second chart graphs ALR. The hope is that the charts graphing ALR vs. PLR will be normalized and may be compared against different datasets.

In Figures 27 and 28 notice how that for all of the dropping algorithms except the capped algorithm the PLR to ALR curve is very linear and that there is very little difference between the impacts of these algorithms. The conjecture is that this is because

although the deterministic, random, random, sinusoidal, sawtooth, and state algorithms all drop packets very differently, they still drop packets across the dataset without any respect to the data itself. The capped algorithm whether capping by packet/second or by bits/second incorporates the volume of traffic into the equations, and it is believed that is the cause of the difference in their results. It is important to note that the results of the deterministic, random, random, and to a lesser degree state algorithms appears very similar to the findings of Schaelicke and Freeland (compare Figure 4 to Figures 10, 12, 14, and 24), but the capped algorithms best reflect our own findings (compare Figure 6 to Figure 21).

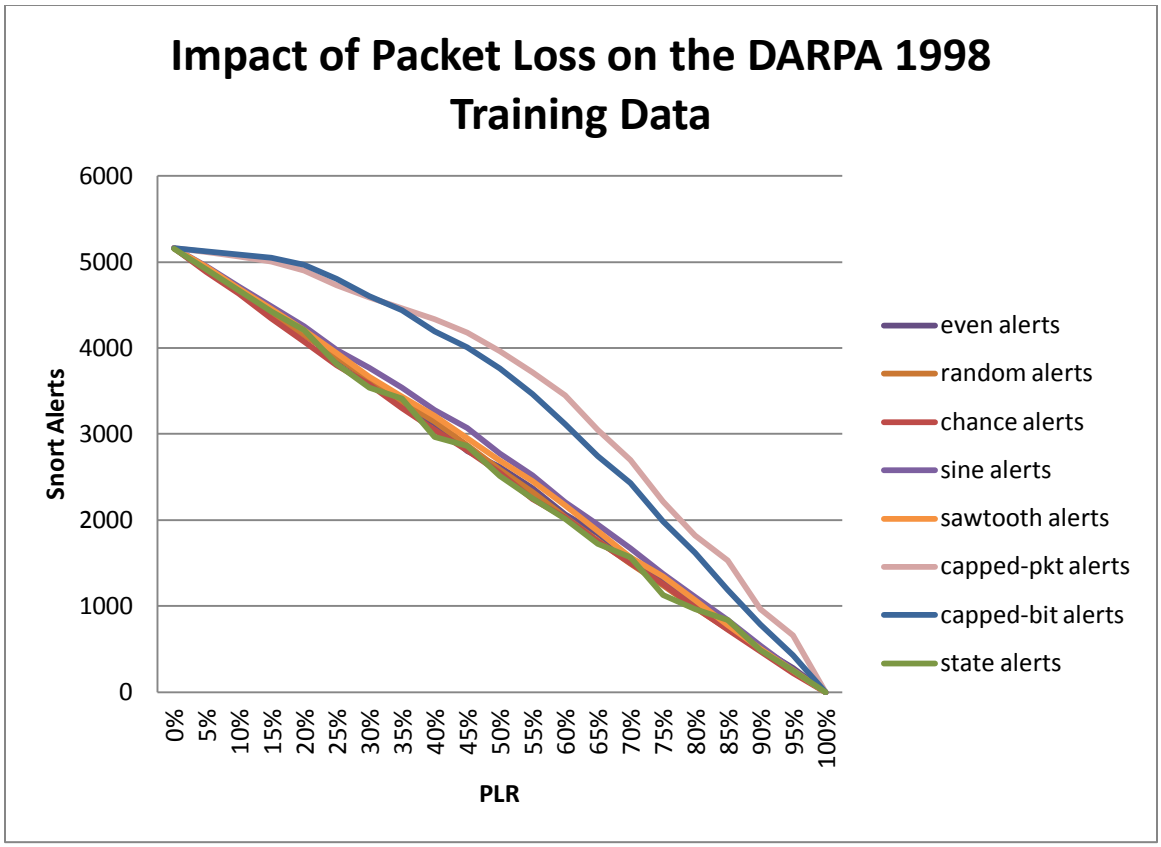

Figure 28 Impact of Packet Loss on the DARPA 1998 Training Data

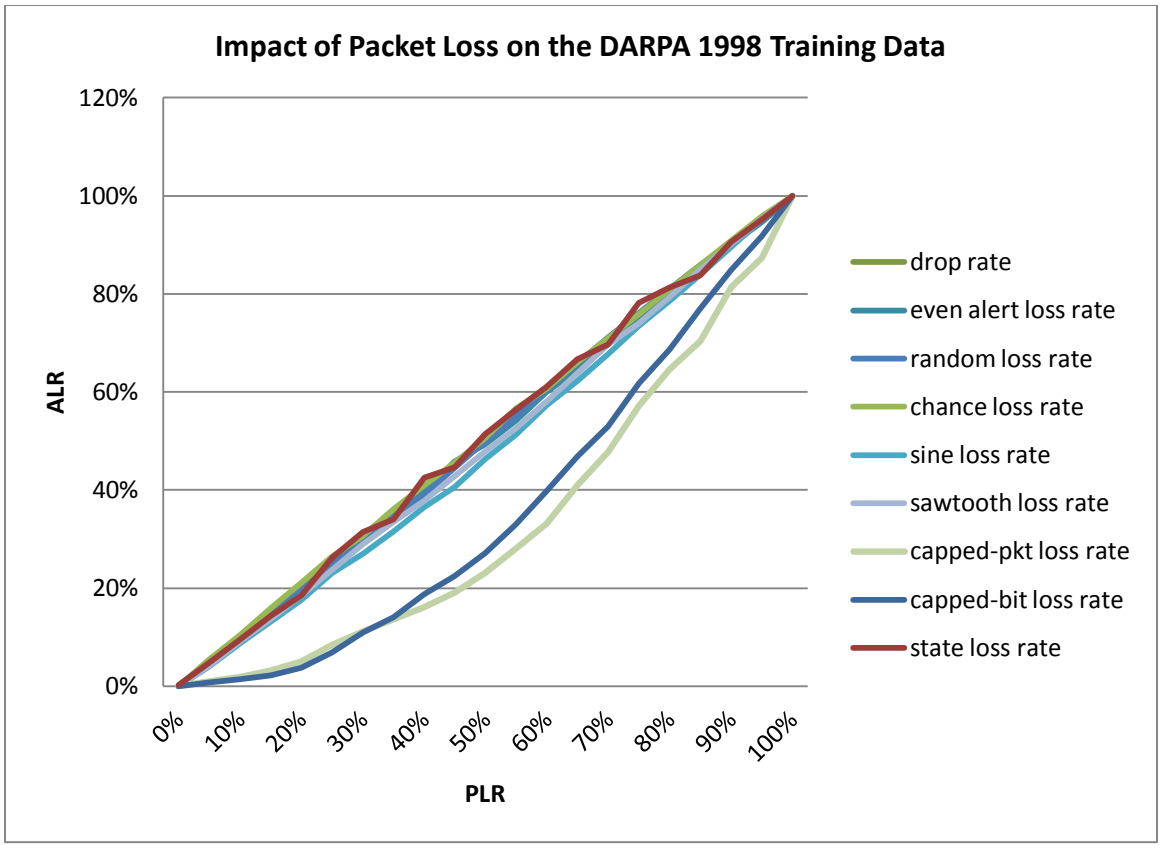

Figure 29 Impact of Packet Loss on the DARPA 1998 Training Data

DARPA 1998 Test Data

During the same study Lippman et.al. created two weeks of test data (Lippmann, et al., 2000). This data was very similar to the training data however it was released without the labels. After the evaluation was completed, the labels were released. This data is valuable for the same reason that the training data had value. This dataset contains 23,341,583 packets and about 5.8 Gigabytes of data. Snort detected 2,173 alerts for a distribution of about one alert for every 10,000 packets.

In Figures 30 and 31 we see very similar graphs to what we saw in Figures 27 and 28, but with greater deviation. It seems reasonable that the deviation can be attributed to the smaller size of the dataset and the smaller number of alerts as we previously discussed in the section on the validity of our results on smaller datasets.

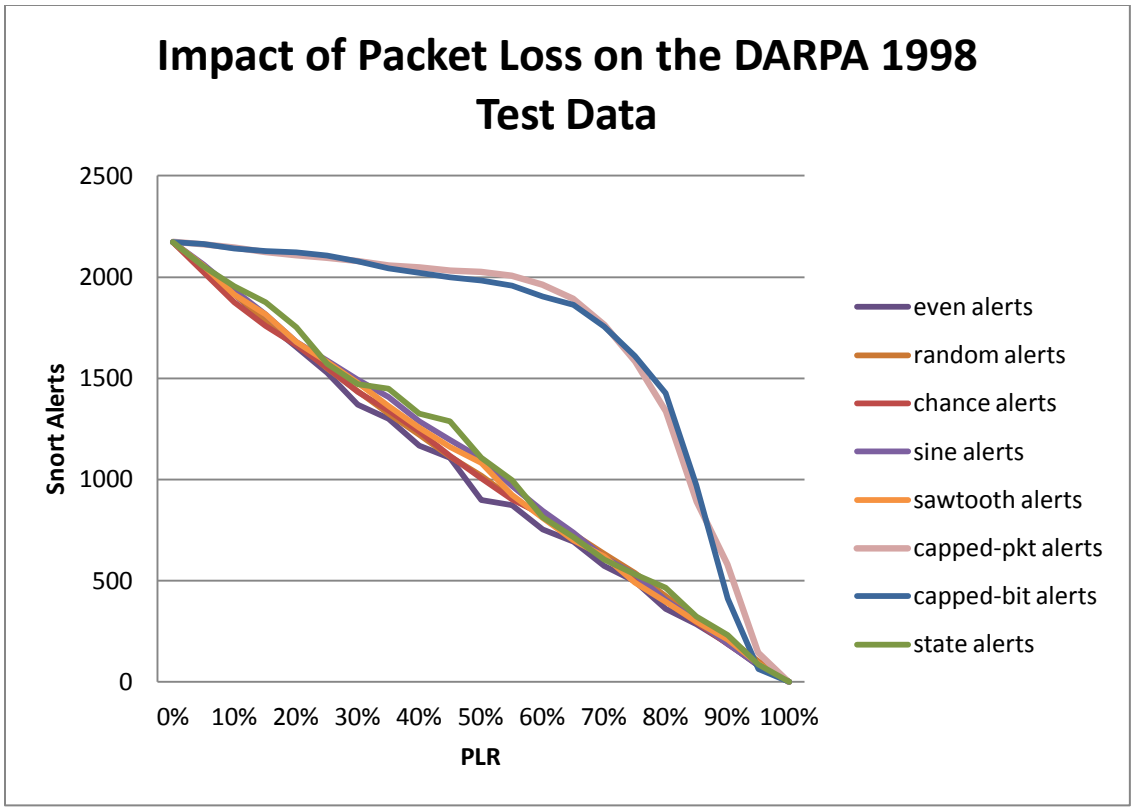

Figure 30 Impact of Packet Loss on the DARPA 1998 Test Data

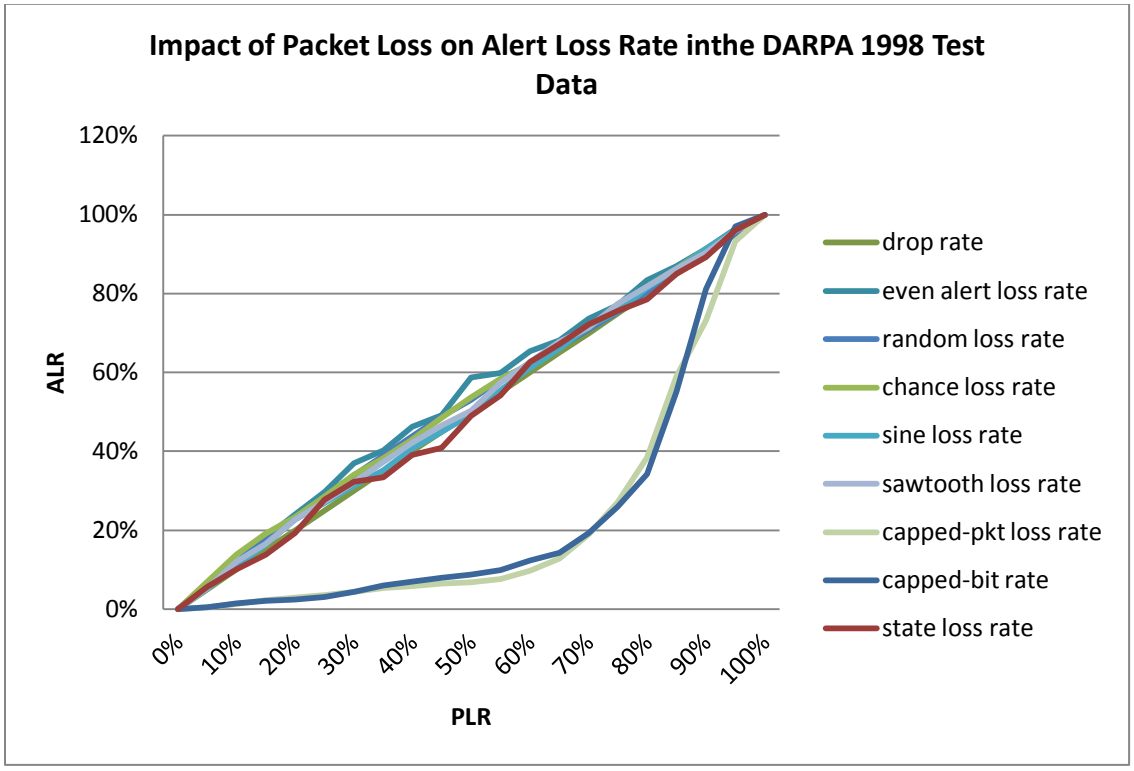

Figure 31 Impact of Packet Loss on ALR in the DARPA 1998 Test Data

DARPA 1999 Data Set

Later Lincoln Labs release a dataset containing seven days of traffic from a simulated network with systems running Microsoft Windows operating systems (Haines, Lippman, & Cunningham, 2001). This dataset contains 84,327,351 packets and about 18 Gigabytes of data. Snort detected 6,420 alerts for a distribution of about one alert for every 13,000 packets.

Notice that the graphs in Figures 32 and 33 look almost identical to Figures 28 and 29. Therefore, it is reasonable to conclude that this shows that the results remain the same whether the traffic is dominated by UNIX traffic or Microsoft Windows traffic.

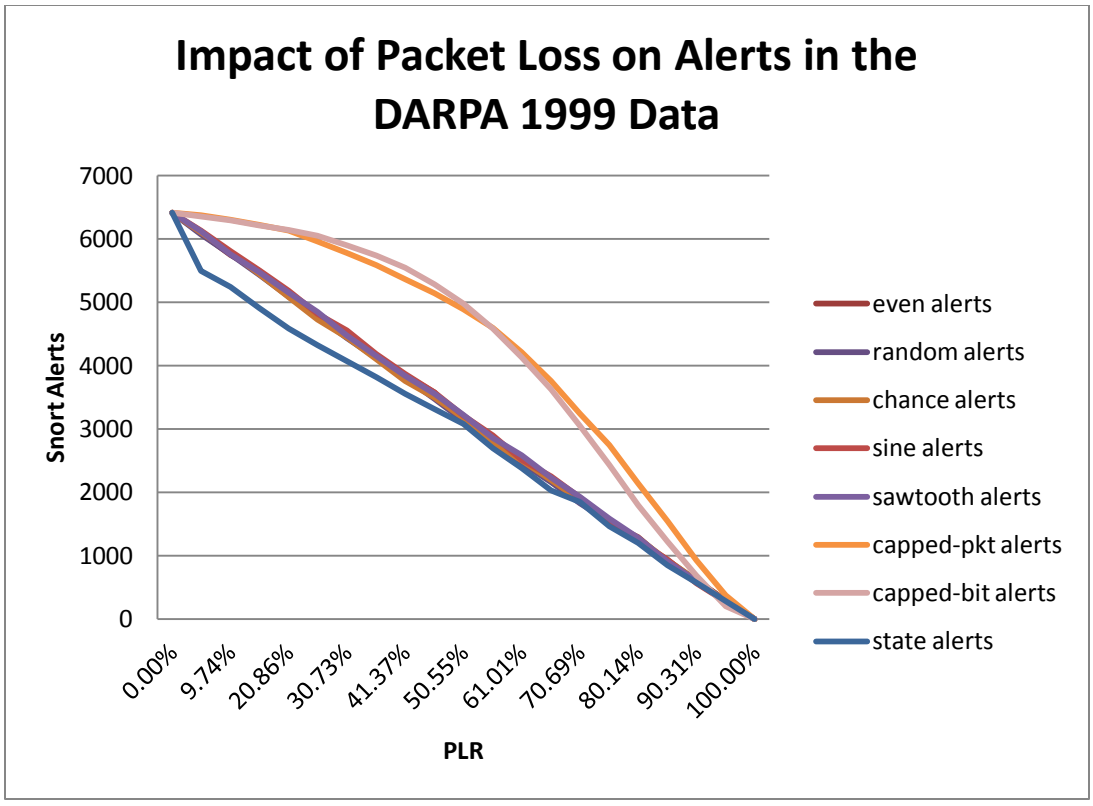

Figure 32 Impact of Packet Loss on Alerts in the DARPA 1999 Data

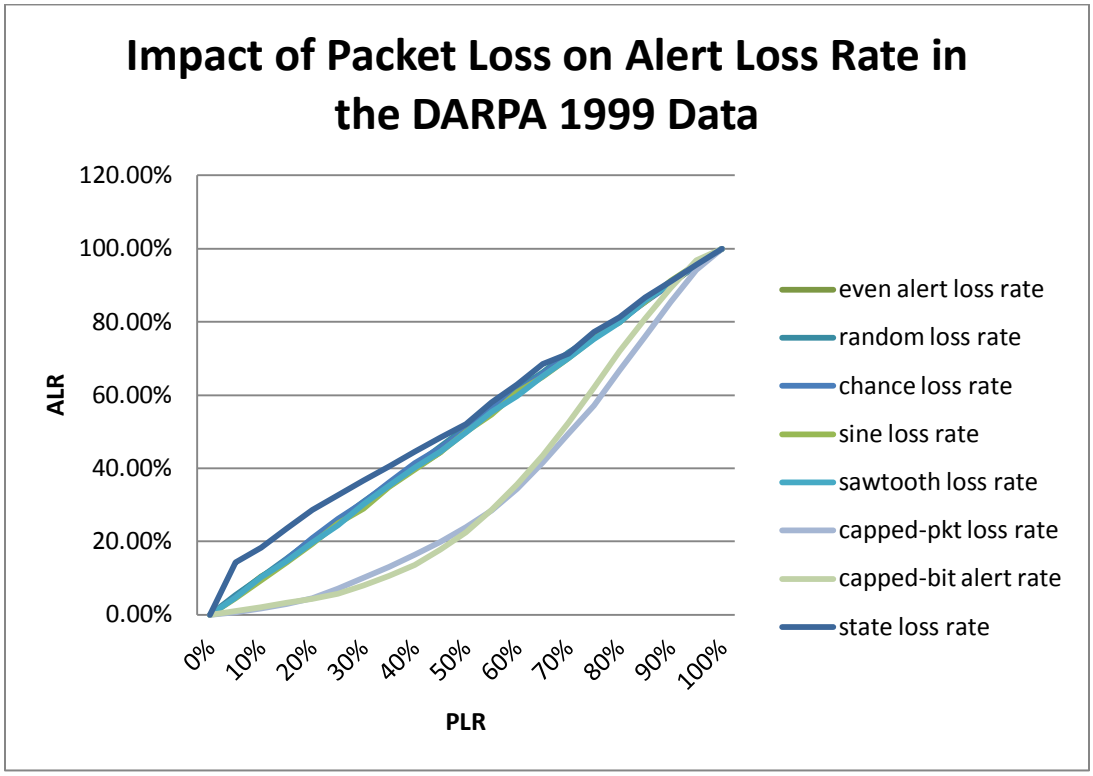

Figure 33 Impact of Packet Loss on ALR in the DARPA 1999 Data

Cyber Defense Exercise 2009

In 2009 The National Security Agency/Central Security Service (NSA/CSS) conducted an exercise pitting teams from the military academies of the U.S. and Canada against teams of professional network specialists to see who could best defend their network (West Point Takes the NSA Cyber Defense Trophy for the Third Straight Year, 2009).

In their paper, Sangster et al. describe their efforts to collect and label traffic from this competition. We were able to obtain this data from https://www.itoc.usma.edu/research/dataset/ (Sangster, et al., 2009). This dataset contains 47,511,801 packets and about 23 Gigabytes of data. Snort detected 2,900 alerts for a distribution of about one alert for every 16,000 packets.

Reviewing Figures 34 and 35, we see a slight bow developing in the relationship between packet loss and alert loss that we did not see in the data from over twenty years ago. This slight curve may indicate that the changes in network activity in the twenty years from 1999 to 2009 have altered the relationship between packet loss to ALRs when packets are dropped consistently across the dataset (as observed by Shaelicke and Freeland and modeled by the deterministic, random, change, and state algorithms). However, this difference may also be attributable to the fact that the DARPA datasets are fabricated in an attempt to reflect the real world traffic they were observing at the time, and the CDX traffic was captured from a competition. The deviation is very similar to what we saw in the DARPA 98 test data, and we believe that this variation is attributable to the fact that these datasets have a similar number of alerts.

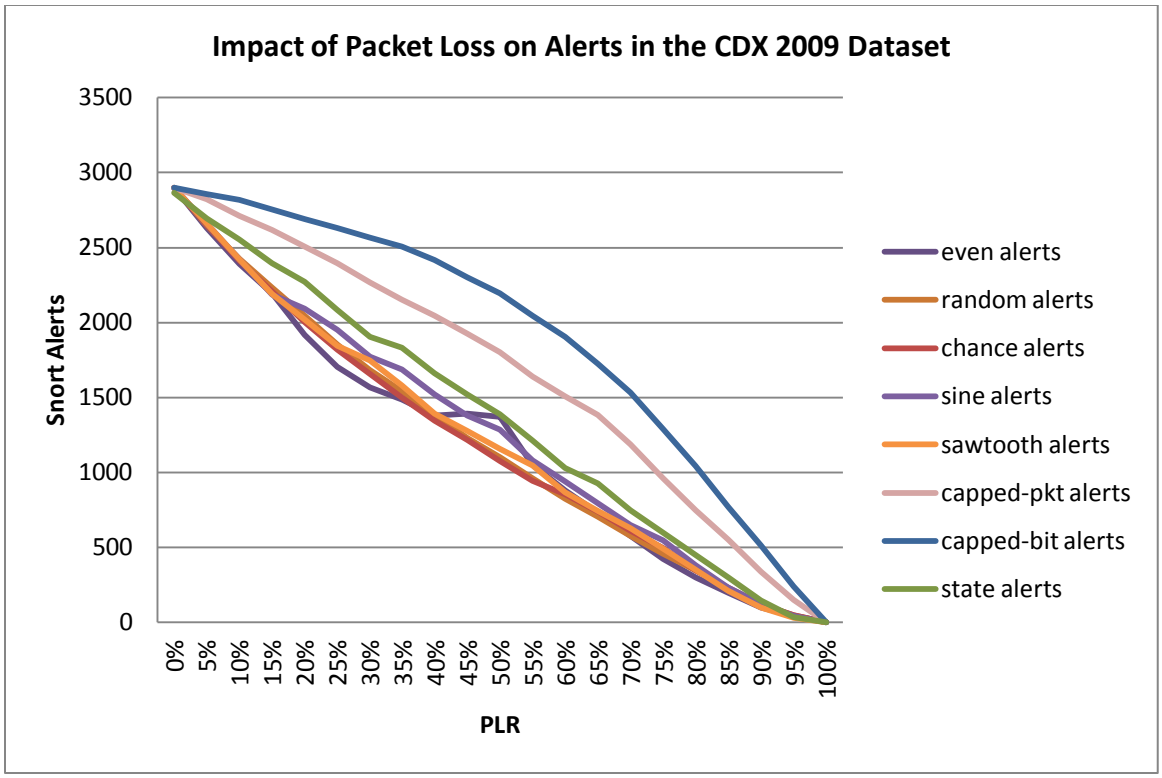

Figure 34 Impact of Packet Loss on Alerts in the CDX 2009 Dataset

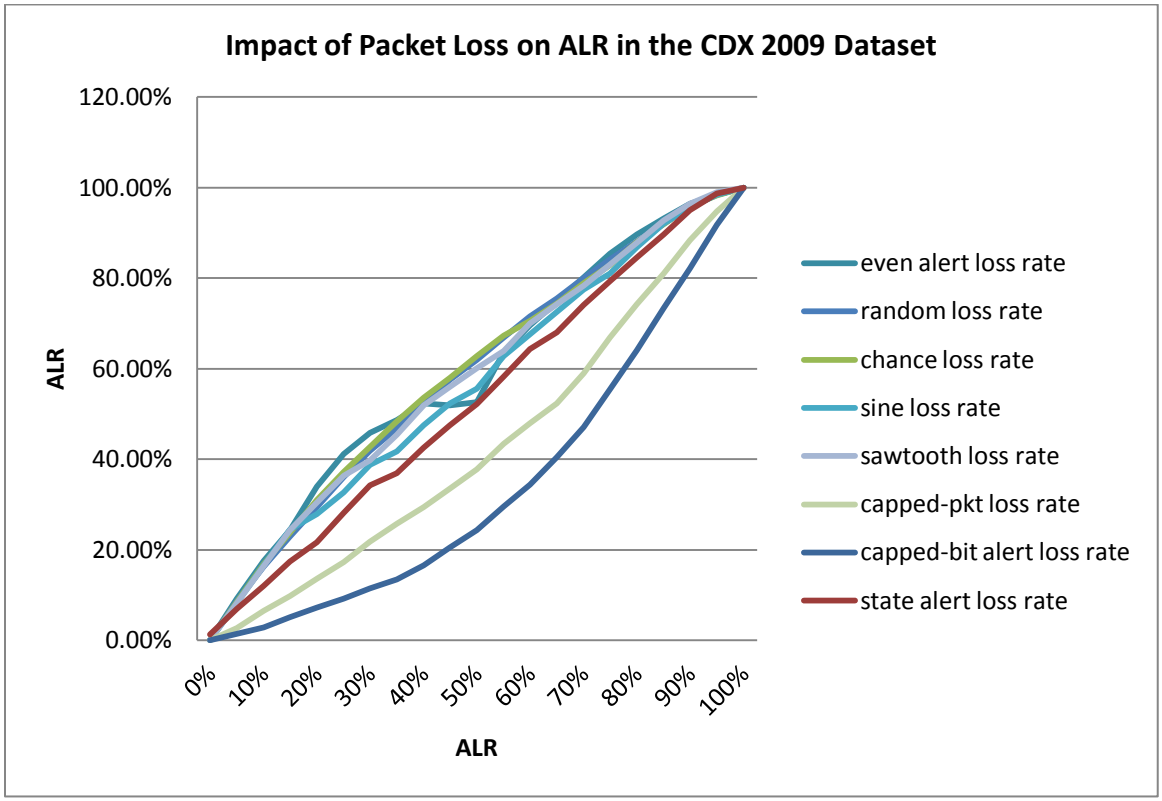

Figure 35 Impact of Packet Loss on ALR in the CDX 2009 Dataset

Collegiate Cyber Defense Competition 2010 (CCDC 2010)

Paul Asadoorian describes his experiences as a Red Team captain in the Mid-Atlantic Regional Collegiate Cyber Defense Competition (CCDC) which pitted blue teams representing Universities and red teams composed of security experts (Asadoorian, 2010). We were able to obtain the packet capture data from CCDC 2010. This dataset contains 264,973,151 packets and about 32 Gigabytes of data. Snort detected 84,913 alerts for a distribution of about one alert for every 3,120 packets.

Looking at Figures 36 and 37 we see further development of the slight curve we first observed in the CDX dataset. For the first time we see a vast divergence between the curve for the capped algorithm when it is capped by packets/second or capped by bits/second with each curving the on the opposite side of the mean. Where the deterministic, random, and random algorithms which best reflect Schaelicke and Freeland observations are plotted almost right on top of each other, we see a divergence of the state, sinusoidal, and saw tooth algorithms that we have not seen in the past.

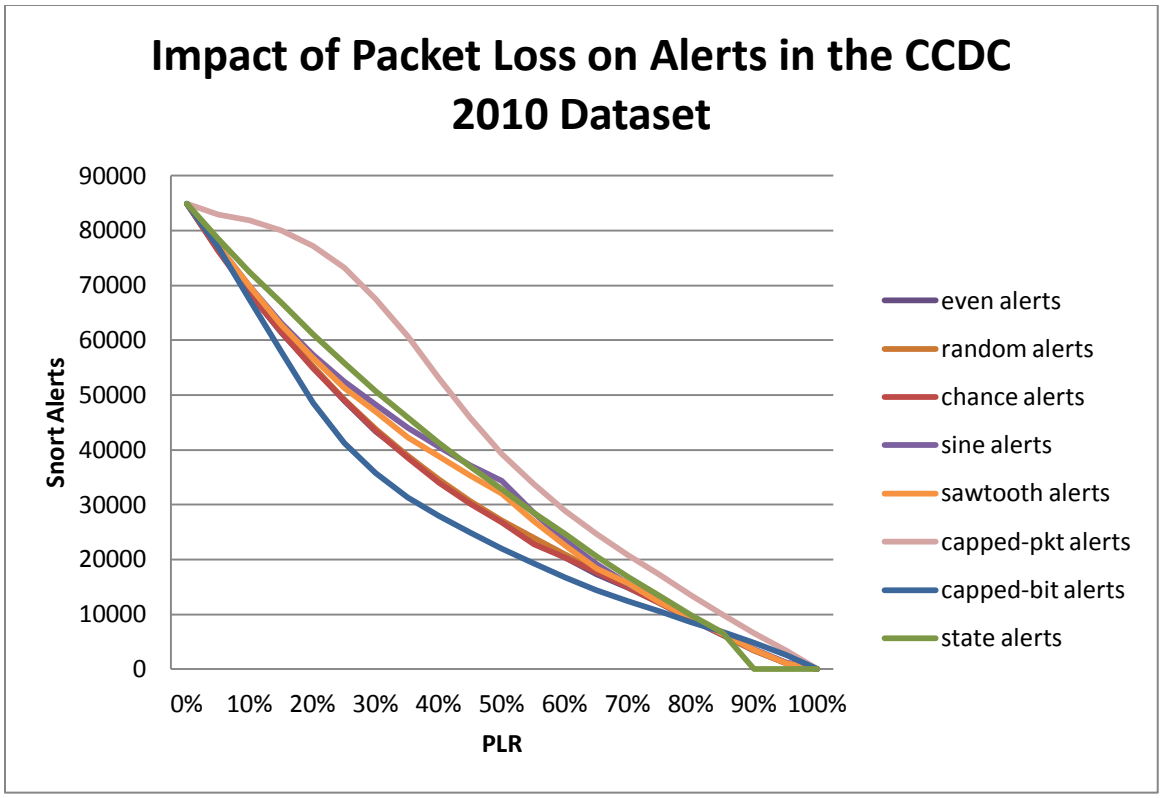

Figure 36 Impact of Packet Loss on Alerts in the CCDC 2010 Dataset

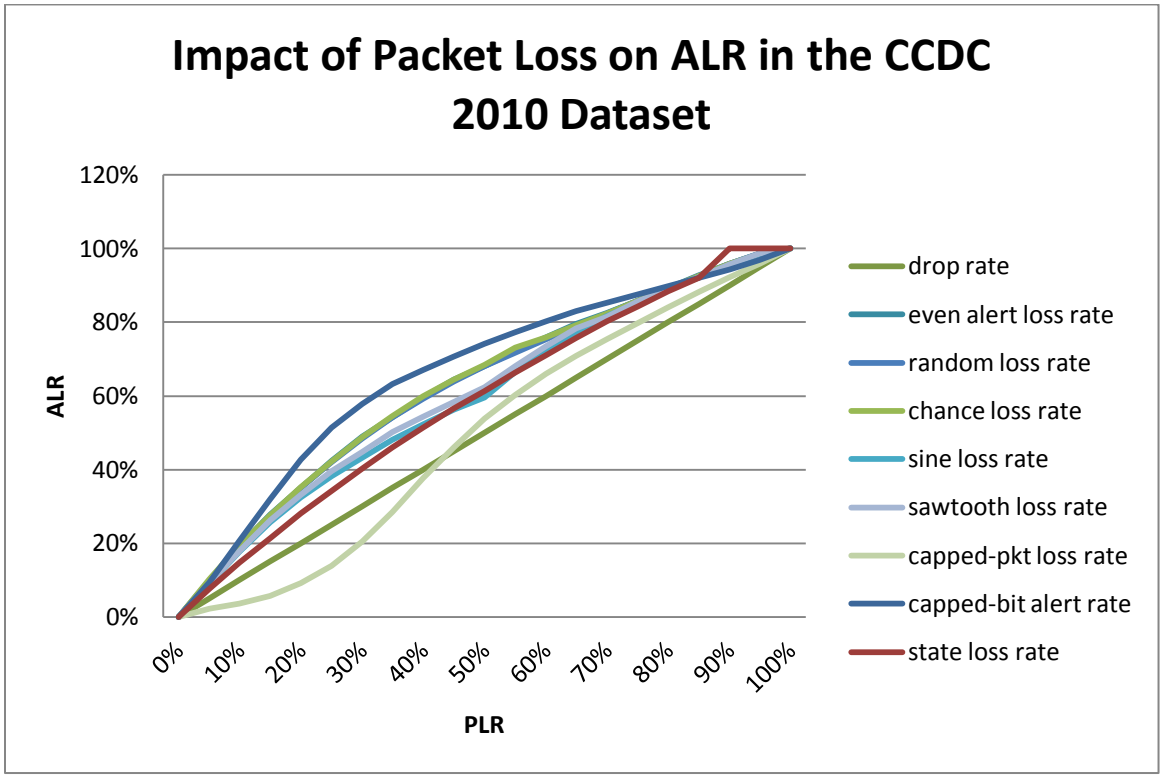

Figure 37 Impact of Packet Loss on ALR in the CCDC Dataset

Summary

We specifically normalized the detection loss by plotting loss rate so that we could compare the results of different datasets. In Figure 38 we have plotted the loss rates of all the datasets and all of the algorithms that we have used in an effort see if there is a consistent pattern we can infer from the data. We see significant deviation; however, the preponderance of the data falls between $ALR = PLR^2$ as a lower bound, and $ALR = \sqrt{PLR}$ as an upper bound. Since we are able to set bounds upon the vast majority of the data, it seems reasonable to conclude that a general formula does exist and with further research we will be able to identify it.

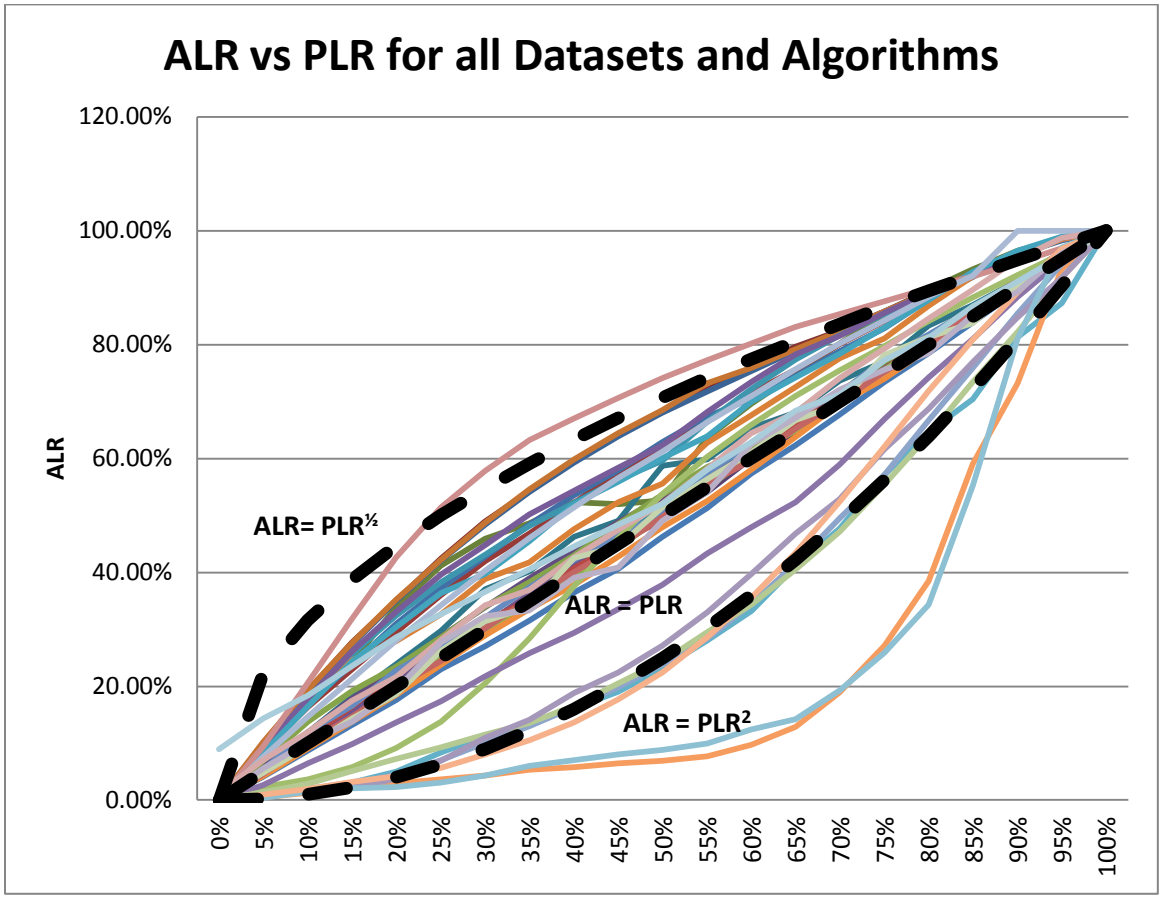

Figure 38 ALR vs. PLR for all Datasets and Algorithms

Discussion

We started by asking several research questions. We will focus our discussion by reviewing each of these questions and relating the answers we have presented.

We asked if there is sufficient regularity in packet loss to allow an algorithm to be developed to model it. Our experiments show packet loss that is very similar to what we saw in the capped algorithms (see Figures 6, 20, and 21). The deterministic, random, and random algorithms produce packet loss which is very similar to that observed by Shaelicke and Freeland. Future research should be explored to determine which models are more indicative of what is seen in the real world; however, it is reasonable to believe that we have enough data to conclude that it may be modeled.

We asked if the impact of packet loss on NIDS performance is sufficiently regular to allow a formula to be developed which will accurately predict the effects. In Figure 38 we see that the vast majority of data points generated by evaluating NID performance may be bounded by the equations $ALR = PLR^2$ as a lower bound and $ALR = \sqrt{PLR}$ as an upper bound. Since we have found formulas that provide reasonable upper and lower bounds for the impact of packet loss on ALR allowing us to predict this impact with some level of confidence, it is reasonable to believe that formula exists and that with further research we can close the gap and provided an deterministic greater level of confidence.

We asked if the same rate of packet loss would induce the same loss of sensor alerts regardless of the strategy used to induce the packet loss. Figures 25 and 26 show that our selection of dropping algorithms do a very good job of covering the space; so that if the way packets are dropped impacts the results of the NIDS we will see it. For most of the algorithms the differences between the algorithms was about the same or less

than the differences between datasets. The capped algorithms are the exception. Most of the algorithms show a linear relations or perhaps a slight curve; however, the capped algorithms all show a clear curve. Therefore, we must conclude that the algorithm used to drop the packets does impact the results.

We asked if the results are independent of the composition of the network traffic. We looked a diverse set of datasets from 1998, 1999, 2009, and 2010. About half of this data was fabricated and the other half was captured from Cyber Defense Competitions. It would be valuable to run our tests against data captured from a live network, but as Figure 27 illustrates we would need a significant amount of data with a significant number of alerts for our results to be meaningful. The datasets are not only different is age, but also in type. The older data are fabricated and the newer data are collected from actual competitions. Although more research is warranted to determine whether the difference in the data is more attributable to age or to type, we cannot say the results are similar enough that conclusions based upon the older data sets would be valid for the newer data sets.

Although much work has been done to reduce packet loss, very little work has been done to characterize or model packet loss and its impact upon NIDS performance and actual intrusion detection. In this research we discovered that a capped algorithm models the packet loss that we are seeing in our experiments. We find that the formula $ALR = PLR^2$ provided a reasonable lower bound for the impact of PLR on ALR. We discovered that the deterministic, bounded random, and random algorithms best model the observations of Shaelicke and Freeland. We find that the formula $ALR = PLR$ provides a reasonable lower bound for the impact of PLR on ALR.

With future research we will be able to continue to experimentally study how packets are dropped in a laboratory environment and compare that to our theoretical experiments. We may also examine data captured from live networks for evidence of packet loss to characterize how this happens in practice. This additional data will allow us to improve the Packet Dropper application. Also it will allow us to prune unrealistic algorithms from the study. Once unrealistic and duplicate algorithms are pruned, we will be able to apply rigorous regression techniques to provide more accurate predictive formulas with confidence intervals.

References

Asadoorian, P. (2010, March 18). *The Mid-Atlantic Regional CCDC 2010 Event - Part I*. Retrieved March 22, 2013, from tenable netowrk security: http://www.tenable.com/blog/the-mid-atlantic-regional-ccdc-2010-event-part-i

Brugger, S. T., & Chow, J. (2007). An Assessment of the DARPA IDS Evaluation Dataset Using Snort. UCDAVIS department of Computer Science 1.

Chung, B.-H., Kim, J.-N., Sohn, S.-W., & Park, C.-h. (2004). Kernel-level intrusion detection system for minimum packet loss. *Advanced Communication Technology, 2004. The 6th International Conference on*, (pp. 207-212).

Gilbert, E. N. (1960). Capacity of a Burst-Noise Channel. *The Bell System Technical Journal* , 1253-1265.

Haines, J. W., Lippman, R. P., & Cunningham, R. K. (2001). Extending the DARPA off-line intrusion detection evaluations. *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX '01. Proceedings , vol.1* (pp. 35,45). DISCEX.

Kim, N.-U., Park, M.-W., Park, S.-H., Jung, S.-M., Eom, J.-H., & Chung, T.-M. (2011). A study on effective hash-based load balancing scheme for parallel NIDS . *Advanced Communication Technology (ICACT), 2011 13th International Conference on* , (pp. 886-890).

Lippmann, R., Fried, D., Graf, I., Haines, J., Kendall, K., McClung, D., et al. (2000). Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation. *DARPA Information Survivability Conference and Exposition* (pp. 12-26). DISCEX '00.

Mizrak, A. T., Savage, S., & Marzullo, K. (2009). Detecting Malicious Packet Losses. *Parallel and Distributed Systems, IEEE Transactions on* , 191-206.

O'Neill, T. (2007, August 23). *SPAN Port or TAP? CSO Beware* . Retrieved February 21, 2012, from LoveMyTool: http://www.lovemytool.com/blog/2007/08/span-ports-or-t.html

Paxson, V., & Floyd, S. (1997). Why We Don't Know How To Simulate the Internet. *Proceedings of the 1997 Winter Simulation Conference*, (pp. 1037-1044). Atlanta.

Salah, K., & Kahtani, A. (2009). Improving Snort performance under Linux. *IET Communications* , 1883-1895.

Sangster, B., O'Connor, T. J., Cook, T., Fanelli, R., Dean, E., Adams, J., et al. (2009). Toward Instrumenting Network Warfare Comptetions to Generate Labeled Datasets. *USENIX Security's Workshop on Cyber Security Experimentation and Test (CST).*

Schaelicke, L., & Freeland, J. C. (2005). Characterizing sources and remedies for packet loss in network intrusion detection systems. *Workload Characterization Symposium, 2005. Proceedings of the IEEE International* (pp. 188-196). Austin, Texas: IEEE Conference Publications.

Song, B., Yang, W., Chen, M., Zhao, X., & Fan, J. (2010). Achieving Flow-Level Controllability in Network Intrusion Detection System. *SNPD '10 Proceedings of the 2010 11th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* (pp. 55-60). Washington, DC: IEEE Computer Society.

Tcpreplay. (n.d.). Retrieved April 2, 2013, from Syn Fin dot Net: http://tcpreplay.synfin.net

Vasiliadis, G., Polychronakis, M., & Ioannidis, S. (2011). MIDeA: a multi-parallel intrusion detection architecture. *Proceedings of the 18th ACM conference on Computer and communications security* (pp. 297-308). New York: ACM.

Wei, C., Fang, Z., Li, W., Liu, X., & Yang, H. (2008). The IDS Model Adapt to Load Characteristic under. *CORD Conference Proceedings*, (pp. 1-4).

Welcome. (n.d.). Retrieved April 02, 2013, from TCPDUMP&LIBPCAP: http://www.tcpdump.org

West Point Takes the NSA Cyber Defense Trophy for the Third Straight Year. (2009, April 28). Retrieved March 22, 2013, from Natioinal Security Agency Central Security Service: http://www.nsa.gov/public_info/press_room/2009/cyber_defense_trophy.shtml

Yueai, Z., & Junjie, C. (2009). Application of Unbalanced Data Approach to Network Intrusion Detection. *First International Workshop on Database Technology and Applications* (pp. 140-143). IEEE.

Curriculum Vita

NAME: Sidney Charles Smith

PERMANENT ADDRESS: 1600 Cynthia Court, Jarrettsville, MD 21084

PROGRAM OF STUDY: Computer Science

DEGREE AND DATE TO BE CONFERRED: Master of Science 2013

Secondary education: North Harford High School, Pylesville, MD 1984

Towson University	2010 – 2013	Master of Science
Towson State University	1988 – 1990	Bachelor of Science
Harford Community College	1985 – 1987	None

Major: Computer Science

Professional publications: N/A

Professional Certifications:

Certified Information Systems Auditor 1081888

Certified Accreditation Professional (CAP) CN: 98606

Security + COMP001006858518

Certified Information Systems Security Professional (CISSP) CN: 98606

Professional positions held:

01 Jan 2010 to Present, Computer Scientist, Team Leader of the Product Integration and Test Team (PITT) for the US Army Research Laboratory 2800 Powder Mill Road, Adelphi, MD 20783

14 May 2006 to 31 Dec 2009, Computer Scientist, Information Assurance Program Manager for the US Army Research Development & Engineering Command 3071 Aberdeen Blvd., Aberdeen Proving Ground, MD 21005

01 Oct 2003 to 14 May 2006, Computer Scientist, Information Assurance Network Manager for the US Army Research Development & Engineering Command 3071 Aberdeen Blvd., Aberdeen Proving Ground, MD 21005

05 Dec 1999 to 01 Oct 2003, Computer Scientist, Information Assurance Network Manager for the US Army Soldier and Biological Chemical Command 5183 Black Hawk Road, APG MD 20010

23 Apr 1990 to 05 Dec 1999. Computer Scientist, Systems Administrator for the Chemical Biological Defense Command 5183 Black Hawk Road, APG MD 20010