2003-06

Stream splitting in support of intrusion detection

Judd, John David

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

STREAM SPLITTING
IN SUPPORT OF INTRUSION DETECTION
by

John D. Judd

June 2003

Co-Advisors: James Bret Michael
John McEachen

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Stream Splitting In Support Of Intrusion Detection			**5. FUNDING NUMBERS**
6. AUTHOR(S) John D. Judd			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

One of the most significant challenges with modern intrusion detection systems is the high rate of false alarms that they generate. In order to lower this rate, we propose to reduce the amount of traffic sent a given intrusion detection system via a filtering process termed stream splitting. Each packet arriving at the system is treated as belonging to a connection. Each connection is then assigned to a network stream. A network stream can then be sent to an analysis engine tailored specifically for that type of data. To demonstrate a stream-splitting capability, both an extendable multi-threaded architecture and prototype were developed. This system was tested to ensure the ability to capture traffic and found to be able to do so with minimal loss at network speeds up to 20 Mb/s, comparable to several open-source analysis programs. The stream splitter was also shown to be able to correctly implement a traffic separation scheme.

14. SUBJECT TERMS Intrusion Detection System, Stream Splitting, Fuzzy Logic			**15. NUMBER OF PAGES** 186
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 **standard Form 298 (Rev. 2-89)**

Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

STREAM SPLITTING IN SUPPORT OF INTRUSION DETECTION

John D. Judd
Ensign, United States Navy
B.S., Eastern Michigan University, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2003

Author: John David Judd

Approved by: James Bret Michael
Co-Advisor

John McEachen
Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

One of the most significant challenges with modern intrusion detection systems is the high rate of false alarms that they generate. In order to lower this rate, we propose to reduce the amount of traffic sent to the intrusion detection system via a filtering process termed stream splitting. Each packet arriving at the system is treated as belonging to a connection. Each connection is then assigned to a network stream. A network stream can then be sent to an analysis engine tailored specifically for that type of data. To demonstrate a stream-splitting capability both an extendable multi-threaded architecture and prototype were developed. This system was then tested to ensure the ability to capture traffic and found to be able to do so with minimal loss at network speeds up to 20 Mb/s. The stream splitter was also shown to be able to correctly implement a traffic separation scheme.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank all of those extremely intelligent people who helped me with this project:

Dr. George Dinolt, for pointing me to Objective-C.

CDR. Chris Eagle for helping me with all manner of coding problems.

My thesis advisors, Dr. John McEachen and Dr. Bret Michael, who both guided and supported me in the building of the stream splitter.

A special thank you to my wife Brooke, who was always there for me with moral support and understanding.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The increasing dependence on computer networks in both the civilian and military communities have caused these networks to become enticing targets for information warriors. The existing techniques for dealing with attacks --Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), and anomalous network monitors-- all have weaknesses, such as generation of false alarms, resource exhaustion, and an inability to perform efficiently in the presence of high network traffic loads. One way to address these weaknesses is to reduce the workload on a given IDS without sacrificing the ability to accurately identify attacks. With a reduction in traffic being analyzed by any one IDS or IPS, the impact of many of the aforementioned weaknesses will, in theory, be reduced.

To limit the traffic flowing into a computing system, a new device called a stream splitter was conceived. This device processes network traffic as a collection of directed connections. Each connection is then analyzed by a series of sensors inside the stream splitter and associated with a stream. Each stream is sent to an IDS for analysis, allowing for each IDS to be configured to analyze a specific type of traffic.

This thesis details the building of a multi-threaded extendable architecture, called a stream splitter, for implementing a traffic separation scheme on a network. The stream splitter was tested for the ability to capture network traffic efficiently and also for the ability to separate network streams from the captured network traffic. The stream splitter is able to efficiently capture network traffic at speeds up to 30Mb/s; at higher network speeds capture engine packet loss becomes excessive. The stream splitter, through the use of a generic switch, is able to route streams to their intended destination by using the media access control (MAC) address of the destination interface. The stream splitter is also able to dynamically adjust the traffic separation scheme at runtime through the addition of a new stream isolation sensor to the stream splitter architecture. The stream splitter allows a network administrator to control the type of data that is sent to each IDS in the network's detection scheme.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

In general there are two types of Intrusion Detection Systems (IDS), signature and anomaly-based systems. A signature-based IDS relies on a database of known attacks with which to compare network traffic in an effort to determine if an attack is in progress. Anomaly-based IDS's rely on the ability to determine what characterizes normal traffic and compare sensed network traffic to what is expected. There are several challenges associated with improving the performance and capabilities of both types of intrusion detection systems. One of the challenges identified in the results of the study reported in [2] is to reduce the number of false alarms generated by such systems, thereby relieving system administrators and other personnel from expending resources to respond to false alarms.

One possibility for addressing the aforementioned challenge is to apply a type of filtering termed "IDS stream splitting," which consists of classifying each packet as a member of a stream when it is encountered between the sniffer and the IDS. To do this network traffic is viewed as a collection of connections. For the purpose of this thesis a connection is a data path between a system on the outside of the network to be protected and a system inside the network. A connection can be characterized by the source IP address, the destination IP address, and the service that is in use. Each packet can then be associated with either an existing active connection, or a new Never-Before-Seen (NBS) connection. This classification allows for the stream of network traffic to be split up into sub-streams based on type of service(i.e. web, e-mail, ftp, etc...) or some other defined metric.

The results of a study published in *Network World* indicate that the traffic on a production-level network caused many IDS's to fail [5]. These systems consume all available resources with logging processes or false alarms. By using a stream splitter to reduce the amount of traffic going to each IDS, the amount of total network traffic that can be inspected prior to a system collapse would increase. In the worst case, all traffic would be of the same type and as such would be sent to a single IDS. If the splitter can

distribute the traffic to any extent, all other things being equal, system up-time should in theory be as good or better than that of the single-channel approach. In other words, the single channel introduces a choke point, whereas the splitter can reduce the amount of traffic that has to pass through the choke point.

The overall detection scheme could be implemented using primarily Commercial Of The Shelf (COTS) technology, with the exception of the implementation for the splitter; this is in line with the Department of the Navy's acquisition policy of acquiring information systems by integrating reusable components, especially COTS products.

B. RELATED EFFORTS

In [7] previous efforts to reduce the false alarm rates of IDS are listed; this list includes placing the IDS behind a firewall, tuning the signatures used for detection, and using network analysis to filter out false alarms from the alarms that are generated. Placing the IDS behind a firewall is one of the easiest reduction techniques to implement. Performing network analysis on generated alarms is both time consuming and requires a detailed understanding of the network that is to be protected. The effect of using a stream splitter is similar to that produced by placing a firewall between the IDS and the network stream, albeit an intelligent firewall. However, a major difference between a firewall and our splitter is that no traffic will be dropped: it can only be diverted.

The results of the 1999 DARPA Intrusion Detection Evaluation performed by Lincoln Laboratory at MIT brought to light some of the problems that plague modern IDS [2]. The signature-based IDS tested were able to alert the operator for a number of the data set attacks. Unfortunately, both recognizing these attacks in the presence of heavy network-traffic loads, or recognizing a legitimate alarm amongst a sea of false alarms remains a challenge. We hypothesize that by using a stream splitter the performance of a COTS IDS can be improved. The "black box" nature of the stream splitter allows for ease of deployment in a wide range of detection schemes. Through the use of a stream splitter, it is likely that operators will be able to more easily detect attacks on high traffic networks and also see a reduction in the number of false alarms due to both the reduction in traffic and the ability to more finely tune their existing IDS to the correct type of traffic each IDS is monitoring.

C. STREAM SPLITTER OVERVIEW

In our daily lives we constantly break information up into categories. An anomaly in one category is normal in another. A formula-1 race car next to you on the freeway is an anomaly, as is a school bus on a formula-1 race track. If the two were switched they would fit in with what was expected, be in the correct context and no longer be anomalies. As humans we are constantly interpreting what we see in our daily lives in terms of the context that we obtain the information. The splitter is able to separate network traffic into streams of information that contain similar data. This allows network traffic to be analyzed according to what type of traffic it is.

The splitter operates through the use of sensors. Each sensor is given full network traffic. There are two types of sensors, active and passive sensors. Active sensors must examine a packet and then communicate the results of the examination back to a sensor control structure where as passive sensors simply receive traffic and act on the information unilaterally.

The use of these two types of sensors allows for the splitter to act as a stream isolator or assign a trust level to packets and route a packet based on its trust, or as a combination of the two. In order to route packets based on trust, the splitter examines each packet as part of a connection and assigns a trust ranking to it from the range [0..1] using a fuzzy logic model. Fuzzy logic (see, for example, [6] for a primer on this subject) has been chosen due to the ability to partially associate an object with a set of objects. In [7], fuzzy set theory was used to determine the confidence of the system that an event had been correctly classified. For network intrusion detection the issue becomes one of how trustworthy or un-trustworthy the traffic is. To be successful each packet must be assigned a trust level. The trust level will be based on the type of connection and the number of times the connection has been seen in a given time window. If no operator action is taken, then over time as the connection is seen more often, the connection will be assigned ever increasing levels of trust until it is no longer sent to the IDS evaluating un-trusted traffic. All NBS connection traffic is viewed as suspicious. Traffic not sent to the un-trusted traffic IDS will be sent to a trusted-traffic IDS to ensure that all network traffic is continuously monitored.

When the splitter is acting as a stream isolator all traffic is sent to a sensor for isolation. If the traffic matches what the sensor is isolating then further analysis is done. If the traffic does not match then it is simply ignored by the sensor. In this way multiple sensors can be employed to look at the same data and only those sensor that find the data useful will act on it. Isolators route traffic themselves so there is no additional information that must be communicated back to the sensor control structure.

The idea for investigating the technical feasibility of using splitters is founded on the principles of Huffman coding[1]. The fewer the number of times a specific pattern is detected, be it a connection type or a specific connection, the more information is present simply by the existence of that item. For example a mail server that connects to a network every day to deliver mail is not as suspicious as a NBS connection using telnet. Consequently, traffic associated with this mail server can be directly forwarded. There are COTS products that will examine e-mail so there is no need to examine the same traffic with an IDS at the network level.

A stream splitter also allows for a detection scheme to grow. For instance a more comprehensive network monitoring system could be built with the addition of an anomaly detection system. This would provide a network anomaly detection capability in addition to the ability to monitor network traffic for attack signatures.

D. OBJECTIVES OF STUDY

COTS routers can split traffic based on service or destination but the stream splitter can also route traffic based on the number of times the connection has been seen or a number of other metrics. That is the splitter acts essentially like an intelligent router. This allows for the use of an IDS tailored for a specific subset of network traffic. This ability to split based on a number of metrics allows for the use of a number of COTS IDS. This reduces the cost of deployment of the system and allows for a detection scheme to grow over time so that as new hardware is added to the system, the additions may be more effectively used.

The specific objective of this thesis is to develop an extensible architecture for network stream splitting and design a functional prototype. This prototype is evaluated for functional accuracy and performance.

If the number of false alarms can be reduced without sacrificing accuracy then network operators can be more efficient in discovering attacks on their networks. This increase in efficiency will result in a reduction in the number of personnel required to monitor a network. This will lead to a reduction in the cost of operation for the Department of the Navy without a drop in the level of protection for naval computer networks.

E. ORGANIZATION OF THESIS

This thesis is organized such that chapter two explains the high level design of the stream splitter and design considerations. Emphasis is placed on why design decisions were made and other options that were available.

Chapter three talks of the implementation of the stream splitter and how each of the major components were coded. Differences between the C++ and Objective-C implementations are emphasized. Data type decisions are also detailed in this chapter.

Each of the sensors that was developed as well as splitter specific storage types are discussed in chapter four. Many of the sensors were ported from C++ to Objective-C with minor modification. Differences in the sensors between the two implementations are discussed as well.

Testing of the splitter is covered in chapter five. Both implementations of the splitter were tested for capture efficiency and compared with Snort. Once the prototype is shown to be able to capture traffic, a traffic separation scheme is correctly demonstrated by the prototype.

THIS PAGE INTENTIONALLY LEFT BLANK

II. THE STREAM SPLITTING PARADIGM

A. STREAMS

A router takes a network stream as an input and then splits the stream by routing packets in different directions according to where the packet is ultimately destined. The result of this is that through the use of a stream splitting mechanism, overall system performance is increased. Every computer on the Internet has no need to see the traffic of every other computer on the Internet. Indeed if this was so, nothing would get accomplished as the system would be in a constant state of saturation. The split in effect makes the existence of such a massive system possible.

In the context of a computer network, traffic can be split into streams by a number of different metrics. The split could be based on source, destination, type of service, protocol, or many other factors. The ability to split a traffic stream into sub-streams makes the problem of traffic analysis more manageable.

Information can be categorized and through this categorization the information becomes more valuable. Once the information can be placed in some sort of context it becomes easier to use. For the stream splitter, each connection is viewed as a category. A packet then belongs to a connection and through this association additional information can be assumed about the packet.

It is this additional information that the stream splitter focuses on, whether doing fuzzy classification or stream isolation, this is what separates the stream splitter from a router. Current routers will not make a judgement as to the trustworthiness of the packets it is routing. The stream isolators apart from doing standard router splitting can also split traffic based on how often the connection has been seen. In this way a low data rate infrequent connection can be separated out from the main network stream and undergo a more rigorous evaluation. This ability to single out slow, infrequent connections, such as a stealth scan, distinguishes the stream splitter from other network analysis tools.

B. HIGH LEVEL DESIGN

To get the desired level of performance, a multi-threaded design was necessary. In general there are three parts to the system. Each part is given its own thread of execution.

- The Packet Capture Engine
- The Packet Analysis Engine
- The Packet Injection Engine

The packet capture engine captures traffic from a network. This traffic is then passed to the analysis engine where it can be analyzed by a number of sensors. The final step is to route the packet based on analysis results. The overall flow of information can be seen in Figure 1.

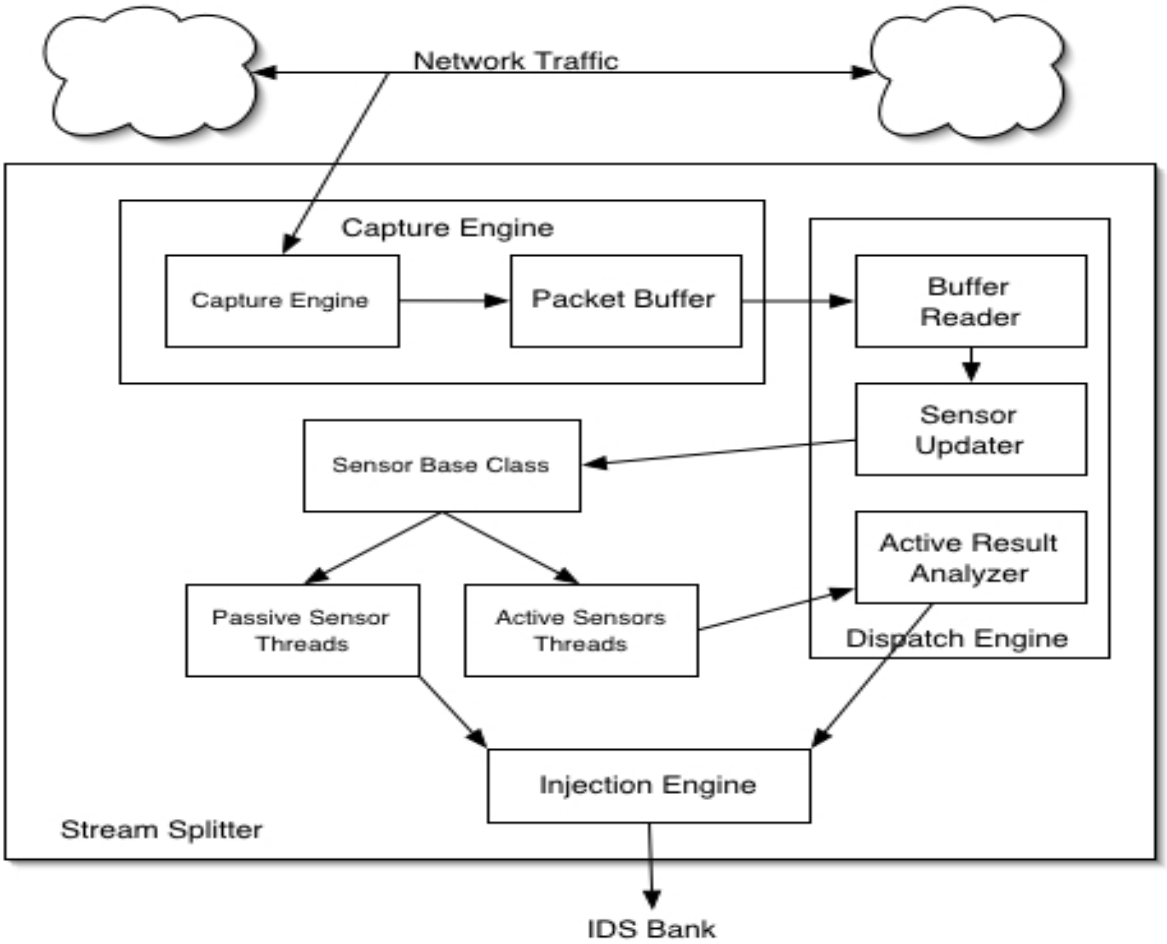

Figure 1. Stream Splitter Information Flow

Figure 2 is a class interaction diagram showing how the major components interact with each other. It is important to note that there can be a number of both active and passive sensors. The stream splitter was designed as an architecture to be used to implement a traffic separation scheme. To meet this design requirement additional

sensors must be easily added to the splitter. This allows for a dynamic separation scheme and reuse of code for unique network configurations.

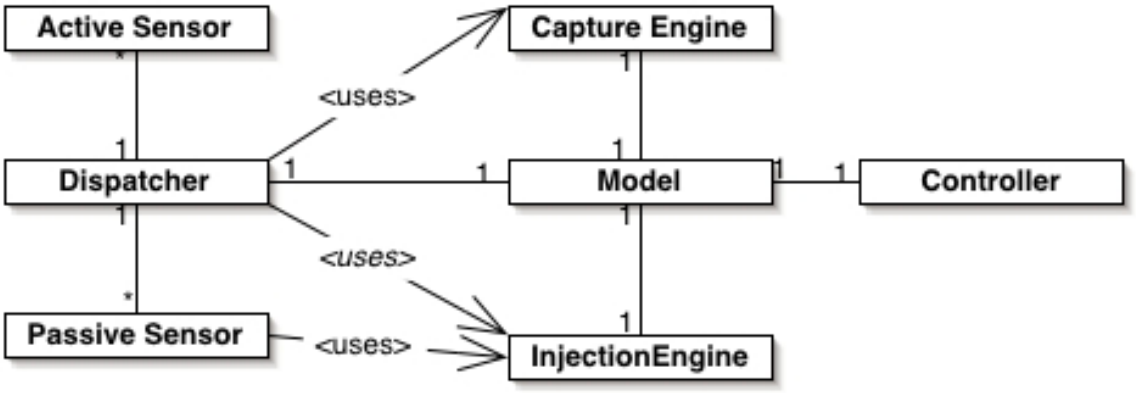

Figure 2. Class Interaction Diagram.

1. Packet Capture

In order to make use of a threaded architecture the packet capture engine must be implemented in a way such that it can run in its own thread. The analysis engine takes longer to analyze each packet than the capture engine takes to capture it from the wire. Further, the capture engine must be able to keep up with network traffic. The goal of the design was to be able to capture as many packets as Snort (www.snort.org), since snort is a well- known network analysis engine.

In order to keep up with network traffic the capture engine must read frames from the wire and place the frames into a buffer in an efficient manner. Once a frame is placed in the buffer the capture engine is finished with it and can then capture the next frame. This allows the system to handle bursts of traffic and drop fewer packets. Packets are then removed from the buffer by the analysis engine. To accomplish this the capture engine must have a buffer and a lock for that buffer that are both shared with the analysis engine. The sequence of events for the capture of a packet is shown in figure 3.

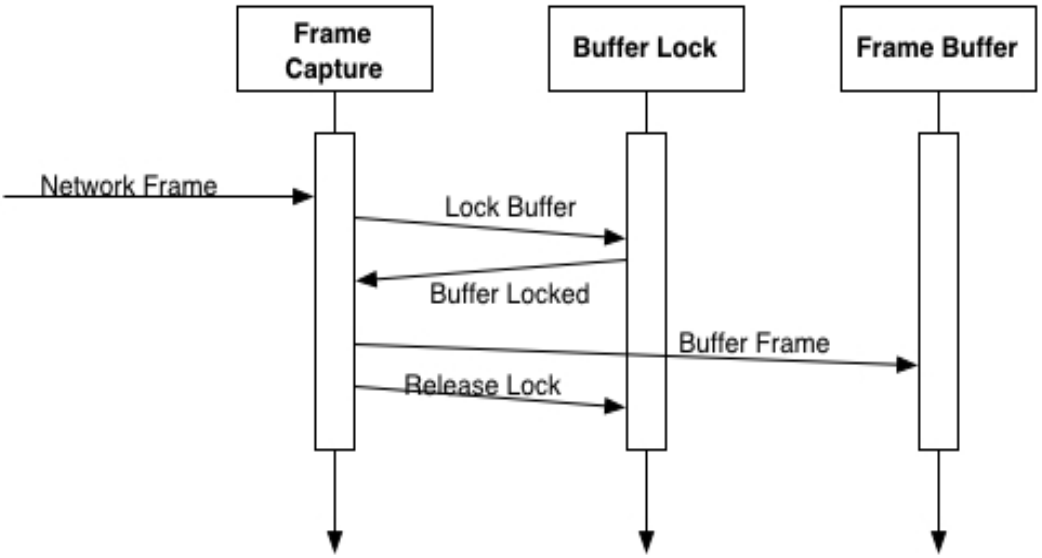

Figure 3. Frame Capture Sequence

2. Analysis Engine

At the heart of the stream splitter are a number of sensors. Each sensor looks for a particular metric in the stream of information. If the sensor finds what that the current set of data is part of the sub-stream it is intended to evaluate then it takes the appropriate action. The use of sensors allows for a fine granularity in the type of information that is used to split a stream. The remainder of this paper will be concerned with stream splitting in a network environment. If the interest is in web traffic, a sensor that evaluates all packets that have a source or destination port of 80 would accomplish this.

Sensors can be either active or passive. An active sensor would send data to a shared memory location where the system would then have to act upon the result of the sensor. A trust scheme where a sensor evaluates a packet for trust might return a trust value. A stream isolator on the other hand would simply route the packet and the system would not need to know anything about what the sensor had done.

a. Active Sensors

Active sensors are used for trust classification. These sensors are active because they do not route the packet but must send back information to some control loop that will then make the decision about routing the packet. Each active sensor looks at a

particular metric and communicates back a trust value based on that metric. Several active sensors may work together to perform a more thorough evaluation of a packet.

When more than one active sensor is used a control structure is needed to gather the results from the sensors and to then make a routing decision based on gathered data. Fuzzy set theory is used in both the individual sensor analysis and also to make the final routing decision. Fuzzy logic is used for the speed with which a decision can be reached. More precision may be possible using Bayesian statistics, however the speed of fuzzy logic is of greater concern than added accuracy of a Bayesian model. This is a case where close is good enough.

b. PassiveSensors

Though similar to active sensors passive sensors are used for stream isolation. These sensors simply process each packet, if the packet does not match the criteria the sensor is looking for no action is taken. If the packet matches what the sensor is looking for further analysis is done. For instance a web traffic isolator would look for all tcp traffic with a source or destination port of 80. Once the packet is identified as being of interest to the isolator additional analysis may then be done. Currently rate analysis is done on traffic that matches the criteria of the isolator. Routing of the packet is then done based on this additional rate analysis.

During the design of the system it made sense to separate the isolation scheme along the same lines as the OSI model. There is a layser-3 isolator that will separate layer-3 traffic out of a stream. There is also a layer-4 isolator that will separate out layer-4 traffic. There was no need to perform isolation at layer-2 since this data changes as packets travel throughout the network. Any isolation above layer-4 would be application specific and would require in depth knowledge of the applications running on the network the splitter was going to be implemented on. Adding this functionality would simply be a matter of extending the layer-4 class to accommodate the type of traffic being isolated.

The analysis engine consists of the Dispatcher and Sensors shown in figure 1. The Dispatcher is responsible for querying the buffer and then updating all the sensors with the new data that was obtained from the buffer. To ensure that all the sensors are looking

at the same information they must all work in lock step. That is to say that though the sensors perform their analysis independently, one sensor may not begin to work on the next frame until all sensors have completed their analysis of the current frame. Figure 4 demonstrates this concept.

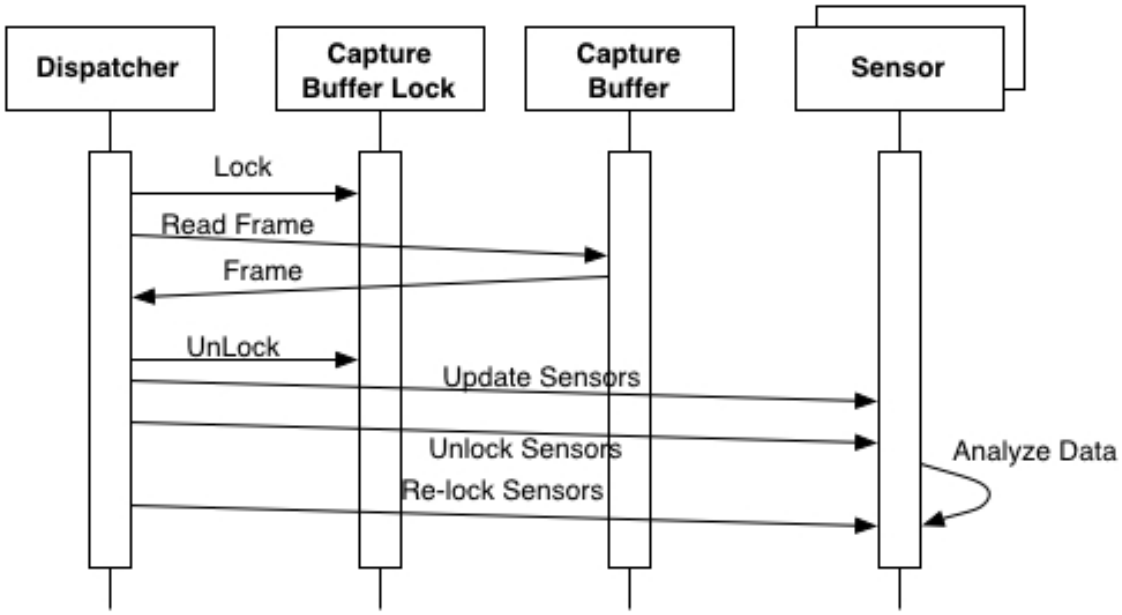

Figure 4. Analysis Sequence

A way to achieve this is to use a lock for each sensor similar to the buffer lock used in the capture engine. The dispatcher locks all the sensors and then updates them with the latest data from the capture buffer. The dispatcher then unlocks the sensors which allows them to lock themselves and process the new information. Upon completion of their analysis each sensor will then unlock their lock allowing the dispatcher to once more take control of each sensor. To ensure that the sensor has processed the information there must be a variable that is reset each time the sensor is updated and then set by the sensor upon completion of analysis. This is necessary because there is no guarantee that each sensor thread will run before the dispatcher tries to relock the sensors. Figure 5 demonstrates this problem situation where the sensor is re-locked prior to the analysis being completed. This will result in the sensor loosing synchronicity with the other sensors. The dispatcher, thinking that all sensors have completed the analysis of the previous packet, will try and send the next packet. This will result in the second packet not being analyzed.

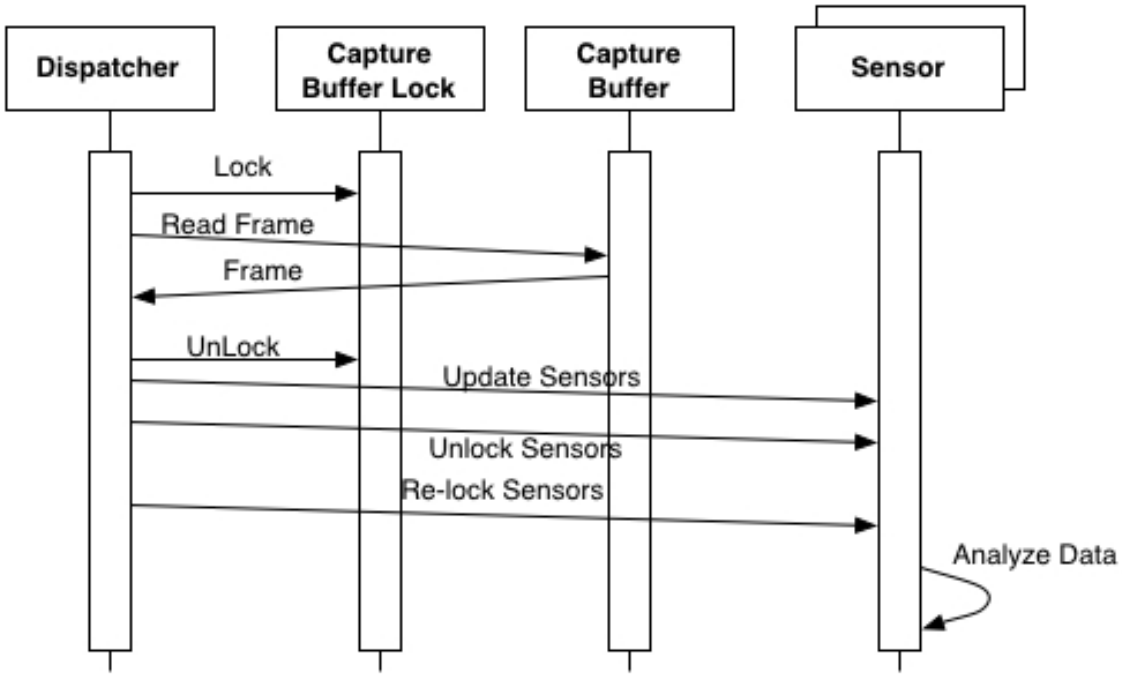

Figure 5. Incorrect Analysis Sequence

3. Injection Engine

Once a frame has been captured and analyzed it must then be replayed back to the network. There are two methods of doing the injection. First, the frame can be simply replayed out of an interface and maintain the integrity of the layer-2 data. This is a simple solution with the only down side being that this requires a separate interface for each stream that is used. An example of this type of injection is shown in figure 6.

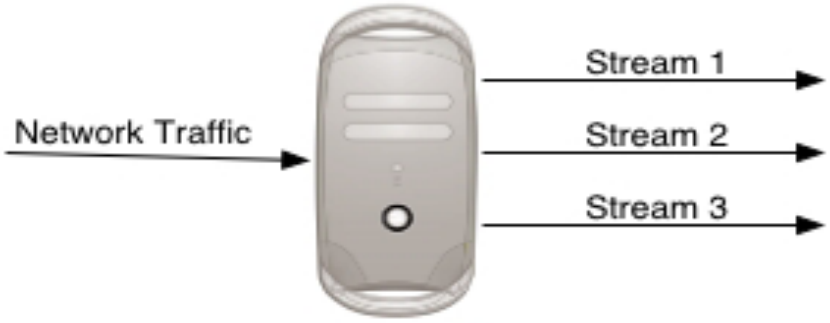

Figure 6. One Interface Per Stream

Another method for replaying and routing outbound traffic is to adjust the destination MAC address. By adjusting the destination MAC address to the address of the

interface on the IDS the stream is destined for, several streams may be sent out the same interface and then fed into a switch that will handle the routing. Figure 7 illustrates this.

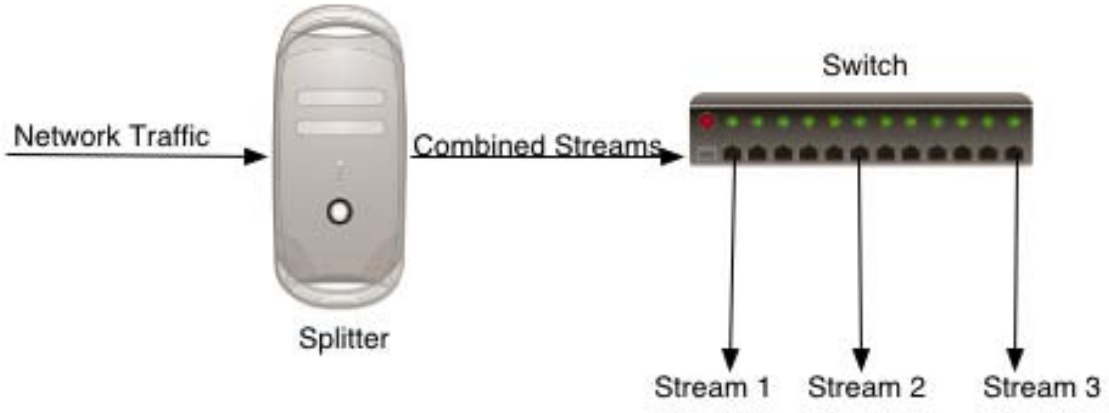

Figure 7. Multiple Streams Using One Interface.

Use of this method requires a buffer for the outgoing frames as one frame may belong to several streams. Once a sensor has decided the destination for a packet and the MAC is over written, the frame is then placed into a buffer from which the injection engine reads and replays Ethernet frames. A lock on the buffer ensures that only one thread is accessing the buffer at a time allowing all sensors requiring the ability to inject traffic to use the same injection engine. Using one injection engine alleviates the possibility that more than one entity will try to control the injection mechanism and also removes the need for an additional mutual exclusion lock surrounding the injection device.

In this chapter, the high level design of the prototype stream splitter was discussed along with reasoning for design decisions. The three major components of the stream splitter were identified and the purpose of each defined. Additionally, two methods of stream replay were discussed, one using multiple network interface cards for injection, the other using a single NIC for injection. The next chapter will discuss the implementation of the prototype and rationale for major decisions made during implementation.

III. IMPLEMENTATION

There are two implementations of the stream splitter. The first was written in C++ and is detailed in this and the following chapter. The second, written in Objective-C, uses the same architecture but does not implement the trust analysis. Additionally, the Objective-C splitter routes all streams out a single interface and lets a switch do the routing to the particular IDS. Objective-C does not make as many virtual table look ups as C++ due to run time method cacheing, This results in a significant speed increase that is discussed in later chapters. Both implementations use a similar packet capture engine and a similar injection engine.

A. PACKET CAPTURE

Packet capture is accomplished through the use of the libpcap library. The libpcap library is a cross platform C library takes much of the tedium of dealing with packet capture out of the hands of the programmer. The splitter uses a call back function that is called each time a packet is encountered. The libpcap library keeps the packet intact and gives a pcap packet header and a pointer to the captured packet to the programmer. Since the packet is kept intact it is much simpler to send the packet out on the wire during the injection phase.

When a packet is encountered by the system it is immediately placed in a buffer where the dispatcher thread can later send it to the sensors. The packet is duplicated since the pcap library reuses the memory for subsequent packet captures.

B. PACKET ANALYSIS

As mentioned earlier there are two methods for analysis employed, trust analysis and stream isolation. For trust analysis the goal is to split the sniffed network stream into trusted and suspect streams. Trust analysis is used if the splitter is going to be placed in front of only two IDSs and the administrator is not interested in doing single service analysis, perhaps the IDS used already does single stream analysis.

Stream isolation is used to separate a stream of interest from the main network stream. This is accomplished through the use of a passive sensor that upon encountering a packet belonging to the stream of interest is then analyzed and routed according to the

rules of the stream sensor. The difference between the stream splitter and an intelligent switch is the use of trust or rate anlysis in the stream splitting decision.

As mentioned earlier, once a stream is isolated it can either all be sent out one interface or it can be sent out a number of interfaces. For instance if the goal is to isolate web traffic then the stream would be split according to port 80. Connections using port 80 may then be evaluated using the same trust scheme as in trust analysis and then be routed accordingly. Using this method the result is a splitting of the network stream into service sub-streams which are then split into trusted and suspected streams. Of course this process can be interrupted at any level. If simply isolating the stream is enough for an analysis scheme, the splitter can do thatas well.

Active sensors use the idea of trust to analyze packets. To develop this notion of trust each packet must be sent to a series of sensors. Each sensor implements a fuzzy membership function for both the Normal set and the Suspect set of traffic. The results are communicated back to the main control loop of the program via shared memory where all the results are analyzed and a best guess as to the trust of the packet is computed. Each sensor that is used has a weight associated with it so that if one sensor is found to be more reliable it can be weighted accordingly.

This notion of trust is used in both trust analysis --if the stream is simply being split according to trust-- and also in stream isolation. When used in conjunction with stream isolation the result is a set of sub-streams based on a metric, for instance web traffic, one of which is considered normal traffic, the other suspect traffic.

1. Sensor Calls

When each sensor is instantiated it launches its own thread that will run as long as the application runs. This is done to increase the performance of the splitter and in an effort to make use of multiple CPU's in the system if they are available. Threads are used instead of a separate process to allow for extensive use of shared memory. Threads also allow for faster context switching than a separate process would allow.

Each sensor sees every packet the splitter captures. This is done so that the various metrics of each sensor can continue to be updated by the packets. For instance since a timestamp is only added to an ongoing connection once a set time interval has

elapsed, the connection sensor must see each packet so that the timestamp can be added at the appropriate time. The source sensor works in a similar manner. There is a drop in performance for continuing to evaluate every packet but this also allows for a stream to be re-classified. If this in-stream evaluation did not take place then the trust for the entire stream would be based on the first packet.

2. Fuzzy Membership Functions

The membership functions assign a number that is the computed trust value that will be associated with the packet. Since we can't account for every possible connection that the system will see we must come up with a way of quantifying the trust value. One approach is to look at the packet and then increase or decrease its trust based on the packet parameters. A packet enters the system and is given a membership of .5 in each of the sets. When the packet is sent to the sensors each sensor will either increase or decrease the packets membership in each of the sets based on the information in the packet. For instance if the connection sensor feels that the packet is to be trusted then it will multiply the current trust by some number greater than1.0 this will increase the trust of the packet. The other case is that the packet is not to be trusted. In this case the current trust will be multiplied by a number less than 1.0.

The problem with the above scheme is that the membership functions will return trust values greater than 1. This value is a trust value and sensor weight all in one. A better scheme is to have the sensor return a membership value and then weight that value in the result analysis function. By returning a membership value debugging becomes much easier and the results from the sensors are more intuitive.

Once the packet has a membership value for each set there are a couple ways that we can decide which way to send it. The splitter can either simply send the packet to a given system if its membership value exceeds a set value or route the packet to the larger of the two membership values. By routing a packet if its membership exceeds a threshold value of membership it will become easier to test the system, with a threshold of 0 for each set all traffic will be sent out both NICs. If traffic is only allowed to be sent out one NIC then the total amount of traffic out of the system can be decreased.

This approach gives rise to the question of whether the set of connections belonging to the suspect set is mutually exclusive of the set of connections belonging to the normal traffic set. This is to say that $\mu_{suspect}(x) = 1 - \mu_{normal}(x)$. Enforcing this policy would ensure that all traffic will be sent to one of the IDSs. If this restriction is relaxed then we can still maintain that all traffic will be sent to at least one of the IDSs by our routing rules.

If ($\mu_{suspect}(x)$ > suspect_setpoint) Then send packet to suspect IDS

If ($\mu_{normal}(x)$ > normal_setpoint) Then send packet to normal IDS

If (!($\mu_{suspect}(x)$ > suspect_setpoint) AND ! ($\mu_{normal}(x)$ > normal_setpoint))

Then by default send traffic to suspect IDS

This would ensure that traffic will be sent to at least one IDS. There is the possibility of overlap between the two outputs, however this may not necessarily be a bad thing. It would be very obvious when a connection is moved from the suspect stream to the trusted stream since there would be duplicated information sent out on both streams.

3. Stream Interruption

When a packet enters the system there are two possibilities it is either associated with an existing connection or recognized as a new connection. Each packet entering the system is sent to all the sensors each of which analyze the packet according to their unique perspectives. If the packet is not part of a previously identified connection then a connection node must be setup. It is only after the examination of this first packet that a trust value can be associated with the connection. This is also true for connectionless protocols such as UDP. Remember that a connection consists of a source IP, destination IP , and a service. There is also the issue of what to do with the first packet of a connection.

There are a couple of ways that connection-based trust can be determined. First, the system may be designed such that any packet belonging to a connection that is not specifically diverted from the suspect IDS will be sent to the suspect IDS. By doing this the system can work out the trust value for a connection and then update the rule set of the splitter. Packets belonging to a trusted connection will no longer be sent to the low-

trust IDS. Further, the trusted IDS will no longer see all the data for a given connection so the possibility of missing an attack on a trusted connection increases with this option.

Another option is to hold each packet when it enters the system until a determination as to the trust of the connection to which it belongs can be made. This would mean that a buffer would have to be used to hold packets under examination. This would open the system up to the possibility that if flooded with new connections the system could crash or be slowed to the point that an attack could be launched while the system was busy analyzing new connections. The addition of a buffer for incoming packets helps with this but does not remove the possibility of a denial of service attack.

An additional way of viewing these two solutions is that the first assumes that the fuzzy classification of inbound connections is done separately from the splitting of network traffic. That is traffic is diverted from a default path only after a decision is made. Prior to that traffic will flow through the system on the default path. The splitter simply responds to rules that are dynamically updated by the fuzzy classifier.

In the second option network traffic is detained until analysis of the connection is complete. Several packets may be needed to establish that a connection is actually active. This means that the fuzzy classification mechanism must be integrated into the splitter such that no routing may take place until each packet entering has been analyzed. This introduces a bottleneck into the system.

To get around the problem of holding packets until a connection can be determined the splitter simply evaluates on a packet basis. This was done since there is still the opportunity for sensors that want to do “connection” analysis to keep the data that the sensor needs but the overall system is not slowed while waiting for the sensor to gather data. The splitter still waits on the results of every sensor but it is hoped that through extensive use of threads the amount of dropped traffic can be kept to a minimum. Each sensor that does more extensive analysis must be implemented in such a way as to not introduce a situation in which the sensor is waiting for the next packet before reporting its results for the current one. If the sensors are implemented correctly each sensor should check to see if the packet is of interest to the sensor prior to doing any CPU

intensive calls. If the packet is of the wrong type the sensor should just return a value indicating that it should be ignored. No sensors in the prototype have this problem.

C. PACKET INJECTION

There is a large difference in the way packets are injected depending on the type of sensor that is in use. Active sensors must report their findings to a control structure that then handles packet injection. Stream isolators inject packets themselves. Regardless of where the logic for packet injection is located all packet injection makes use of the same underlying library to actually write the packets to the wire, libnet.

The easiest and cleanest way to accomplish packet injection was to use the libnet library written by Mike D. Schiffman. The source code for libnet can be found at http://www.packetfactory.net/libnet/ The Libnet library provides a way to do layer-2 packet injection on a linux/unix system.

When Libpcap captures a packet, the packet is stored as an array of u_char elements. It so happens that the libnet advanced write function takes a u_char pointer as a total length and then writes the wire ready packet to a specified interface. Libpcap supplies all this information in the pcap packet header so it is a simple matter to write the packet out to the wire with the libnet_write function.

As mentioned earlier an alternative to the use of a separate network card for each stream to be sent out of the splitter is to use the MAC address of the destination IDS. Layer-2 is no longer maintained under this scheme but it does allow for multiple streams to be sent out a single NIC. If the output interface is plugged directly into a switch then the destination IDS can still be used with their interfaces in promiscuous mode. If a switch is not used then the IDS will have to do some additional filtering. The Objective-C implementation of the splitter uses this MAC address routing.

In this chapter the implementation of the stream splitter was described. Rationale for major decisions made during the implementation was given, external libraries used in the stream splitter were described, and the fuzzy logic model introduced. In addition the stream interruption problem was discussed. The next chapter will talk about each of the sensors used in the stream splitter will be discussed along with abstract data types developed to support the prototypes.

IV. SENSOR DESIGN

There are a number of sensors used in the splitter. Each sensor looks at a particular aspect of network traffic. The number of sensors is dependent on the system administrator's needs for traffic separation. A simple stream isolator may have only one passive sensor, where as a trust analysis splitter may use an extensive array of both active and passive sensors.

Sensors rely on several supporting data structures to function efficiently. There are defined structures for IP headers, TCP headers, and UDP headers. There are also specialized data structures used for the nodes in the Adelson-Velskii and Landis blanced binary search trees (AVL trees) found in a number of the sensors. There is also a buffernode used to encapsulate a newly arrived packet and ready it for queuing in the captured packet buffer. This chapter will discuss the use of these data structures and features integrated into the design of the data types.

A. THREADING THE SENSORS

As stated earlier in an attempt to increase system performance each sensor is run in its own persistent thread. The thread for each sensor is launched at the beginning of the program and executes until the main thread exits. The main thread instantiates all the sensors, initializes the libpcap and libnet libraries, launches the sensor threads, and launches the dispatcher thread.

Each sensor is a derived class of the "Sensor" base class. The base class has a mutex of type pthread_mutex_t as well as a conditional variable of type pthread_cond_t. These variables protect the packet information as well as the membership variables that the sensor uses to communicate the results of the sensor back to the dispatcher thread. The dispatcher thread acts as the control structure for both passive and active sensors. Active sensors require a control structure to correlate the results they generate and to make the final decision as to where to route the packet. Passive sensors merely pass back a "complete" signal, and only need to be told when a new packet is ready for analysis.

When a packet is encountered, main instantiates a new buffer node. Two calls to memcopy copy the packet and the associated pcap packet header into the new buffer node. A timestamp is then added and the buffer mutex is locked. With the buffer mutex locked, the node is placed into the buffer and the number of packets in the buffer is incremented. The mutex is then unlocked and if the number of packets in the buffer is greater than or equal to one the dispatcher thread is signaled. This last check of packets in the buffer reduces the number of unnecessary signals sent to the dispatcher thread.

The dispatcher thread is then awoken by a signal from the capture thread and removes the first packet on the list. The dispatcher then locks all the sensor mutexes and updates the packet information in the sensors to point to the data in the newly removed buffer node. With this done, the dispatcher unlocks all the sensor's mutexes and broadcasts via each sensor's conditional variable to signal that the packet information is new and must be processed.

Once a sensor is woken up it checks to see that the return locations for the membership functions have been reset. This check ensures that a spurious wake up will not cause a packet to be analyzed twice. Providing that the return values have indeed been reset, the sensor then calls the derived class's "analyzePacket" function. When this function returns, the sensor will have analyzed the packet and the return variables updated. The sensor then loops back to the top of the execution loop and, since the return variables are not in a reset condition, the sensor goes back to sleep to await a new signal from the dispatcher.

By waiting on the conditional variable the sensor unlocks it's mutex allowing the dispatcher thread to gain access to the results of the sensor's analysis. The dispatcher waits for control over all sensors before continuing. Once it has all the sensor mutexes locked it can then analyze the results and route the packets accordingly.

The exception to this is with stream isolators. This type of sensor routes packets itself. This is done since there may be many types of sub-streams and each one will have to be routed differently. It is more efficient to have each isolator route packets itself. This also allows for a trust scheme to be employed on the traffic as well as a stream isolation scheme.

Once the analysis function returns, the dispatcher checks to see if there are more packets to be analyzed. If there are then it locks the buffer mutex, decrements the number of packets on the buffer by one, removes the first packet from the list, and then the process repeats.

B. ADDRESS NODES

The address nodes consist of a 32 bit source address and a list of timestamps. The natural ordering of this set is imposed by viewing the address as an unsigned 32 bit integer. The list of timestamps is used to keep track of how often the source IP has been seen. Once a timestamp has been added to the list of timestamps it will stay there for a period of time defined by the system administrator prior to execution. As a protective measure against a Denial Of Service(DOS) attack a time stamp can only be added once every time interval where the time interval is defined in the address node class. The amount of time a timestamp remains on the list and the amount of time between timestamps need not be the same.

When an attempt to add a timestamp is made the first thing that is done is a check to ensure that enough time has elapsed to allow the addition to take place. If it has not been long enough to add the timestamp, it is not added. With each call to add a timestamp the list of timestamps is also purged of old timestamps.

C. CONNECTION NODES

Connection Nodes are similar to address nodes in that they perform the function of keeping track of what the sensor has seen. A connection node contains source and destination addresses in the form of 32 bit unsigned integers, a service port, and a list of time stamps. The connection node keeps track of the direction of the connection. It is the responsibility of the sensor to determine the direction and service being used.

The ordering of this set comes from looking at the larger of the addresses. Ties of the larger of the IP addresses are handled by comparing the second addresses. If this also results in a tie then the service will break the tie. If all three match then the nodes are equal. Though ephemeral ports could be used to break ties the stream splitter is concerned only that the connection belongs to two physical computers. Two simultaneous

connections between two computers would be viewed as the same connection by the stream splitter.

Connection node timestamps are handled in an identical manner to address node time stamps. A time stamp can only be added if a set amount of time has elapsed since the last time stamp was added and the time stamps only live on the list for finite amount of time. As with the address node timestamps this adds a bit of protection against a DOS attack.

D. CONNNECTION SENSOR

There are two views that can be taken for the connection sensor. One is that the connection sensor should see all the other sensors in an attempt to get a better idea of the trust of the connection and also act as the control structure for the other sensors. This would lead to a hierarchy of the sensors with the connection sensor at the top of the list and all other sensors subordinate to it. I have taken a different approach, each sensor makes its decision independent of the other sensors. This puts all sensors in a flat hierarchical structure. In the final step of the trust evaluation process all sensor results are weighted, this gives the ability to put more value on one sensor than another. If the connection sensor were allowed to call all of the other sensors this would no longer be possible.

The connection sensor tries to match all TCP packets to an existing connection. If the packet can't be matched to an existing connection then a new connection has been identified and must be setup using a new connection node. The connection sensor ignores all non-TCP traffic by returning an ignore code in the "suspectMembership" and "normalMembership" variables.

The connection sensor makes use of an AVL tree to store all connections the system has seen in the form of connection nodes. An AVL tree is used for its speed of access. It may take a bit longer to add a new node but once a node is added it will not be removed so emphasis is placed on lookup speed and not necessarily insertion speed.

The connection sensor uses a very simple algorithm for its fuzzy membership functions. Trust is based on how often a particular connection has been seen. The number of times a connection has been seen is determined by the number of timestamps in the

connection node for a particular connection. A connection is determined to be trusted if it has been seen a certain number of times within a defined trust window. The system administrator must define the window and trust threshold. The fuzzy membership functions are shown in figure 8 and figure 9.

$$\mu_{Normal} = \frac{\#\,timestamps}{trustThreshold}$$

Figure 8. Normal Set Membership Function

$$\mu_{Suspect} = 1 - \frac{\#\,timestamps}{trustThreshold}$$

Figure 9. Suspect Membership Function

These values are then returned to the sensor control structure, in the case of the prototype dispatcher thread.

The "analyzePacket" function is called from the sensor base class and starts the analysis process for each packet the system captures. The connection sensor first verifies that the packet is a TCP packet, next it builds a temporary connection node that will be used to search the AVL tree. The temporary node is built as a directed connection. This is to say that based on the location of the service port a determination as to the direction of overall information flow can be established. As an example, if the source port was 80 then the sensor would assume that the packet is a response from a web server.

A search of the AVL tree for the temporary node will find the node and set a pointer to the node in the tree, or a new node will be added to the tree and then the search repeated to get a pointer to the newly added node. This is done to ensure that the node has been successfully added to the tree.

Upon finding the connection node to which the packet belongs the node is updated with the current timestamp and old time stamps are removed from the timestamp list. The pointer to the node is then sent to the fuzzy membership functions. The fuzzy membership functions return the membership of the connection node in the set of normal and suspect traffic sets. These membership values are then stored in "normalMembership" and "suspectMembership", both of which are located in the sensor

base class. Once the membership variables have been updated the mutex covering the connection sensor can safely be unlocked to allow the control structure to retrieve the results.

E. SOURCE SENSOR

The source sensor works in a manner very similar to the connection sensor. The source sensor looks at the source of every packet and keeps track of how many times it has seen a particular source through the use of address nodes stored in an AVL tree.

Like all sensors the start of analysis is when the "analyzePacket" function is called. The sensor first extracts the source IP from the packet. A temporary address node is made with the newly acquired IP address for purpose of searching the AVL tree. A search of the existing AVL tree yields two possibilities, either the node exists or it does not. If the node does not exist it is created and added to the tree. If the node had to be added then an additional search is done to ensure that the node can be recalled from the tree. Upon finding the node in the AVL tree a call is made to "addCurrentTimestamp()" which adds the current timestamp to the node. In the process of adding the current timestamp old timestamps that have exceeded their lifetime on the list are removed.

Once the node has been updated then a pointer to the node is passed to both the normal and suspect membership functions; normalMembershipFunc(), and suspectMembershipFunc() respectively. This sensor makes use of simple fuzzy membership functions similar to the connection sensor. The membership functions store their results in "normalMembership" and "suspectMembership", located in the sensor base class. With the membership values computed the sensor is finished with analysis and the mutual exclusion lock covering the sensor can now safely be unlocked.

F. UDP SENSOR

The UDP sensor does for UDP packets what the connection sensor does for TCP packets. The basic functionality is the same. The "analyzePacket" is the function that is called by the sensor base class and starts the analysis process within the sensor.

The first thing that happens in the sensor is that the packet is verified to be a UDP packet. If the packet is non-UDP then ignore codes are returned in place of membership values via the "normalMembership" and "suspectMembership" variables.

The UDP sensor also makes use of directed connections. Once a packet is recognized as being a UDP packet then a temporary connection node is built in such a manner that the source and destination addresses are in agreement with the flow of information. This is accomplished by looking at the location of the service port. A port is considered to be a service if it is less than 1024. Currently no attempt to handle service ports greater than 1024 is made. If the destination port is a service port, then the sensor assumes that the orientation of the source and destination addresses are correct. If the source port is a destination port, then the source and destination addresses for the temporary node will be reversed.

Once the temporary node is built then the AVL tree is searched for a node matching the temporary. If a node can not be found in the tree then the temporary node is added and then the tree is searched again to ensure the node was added and to obtain a pointer to it.

Having established a pointer to the node in the AVL tree, the fuzzy membership functions are called. The UDP sensor makes use of the same fuzzy membership functions as the connection sensor. The results of the membership functions are stored in "suspectMembership" and "normalMembership", both found in the sensor base class. Analysis accomplished, the sensor mutex is unlocked and the sensor waits for the next packet.

G. LAYER-3 ISOLATOR

The layer-3 isolator is the first of the stream isolators. Stream isolators are passive sensors. Stream isolators do not report back results to a controlling structure. Instead isolators simply analyze traffic and when a packet is of the correct type additional analysis is done and ultimately a routing decision is made.

Like active sensors the layer-3 isolator is a sub-class of the sensor base class. This ensures that the isolator is run in its own thread. This also allows the isolator to run in parallel with active sensors and be launched from the same control structure that is controlling the active sensors. To do this the isolator must return an ignore code to the control structure using "suspectMembership" and "normalMembership" from the sensor

base class. This ignore code must be passed in order to alert the sensor control structure that the isolator has completed its analysis.

The layer-3 isolator is currently the lowest level on the OSI model that the splitter is currently implemented to support. That is to say that the splitter can isolate a stream based solely on the layer-3 protocol of the packet. The layer-3 isolator is designed such that stream isolators for higher OSI layers will build on top of the layer-3 isolator. With this in mind the layer-3 isolator holds an AVL tree that consists of a set of isolator nodes. In keeping with this idea of forming a base for other isolators, the layer3synch function checks to ensure that that the packet is of the correct layer-3 type. This type is passed to the isolator by way of the layer-3 isolator constructor at initialization. If the packet is of the correct type, layer3synch will return true, the Ethernet header pointer will be pointing to the start of the Ethernet frame and the currently implemented IP header will be pointing at the start of the IP packet held in the "pdata" data structure. With the intention for the layer-3 isolator to serve as only a base class for an isolator, there is no "analyzePacket" function defined.

An AVL tree is used for the storage of the isolator nodes for its speed of access and also to avoid the need for using an additional storage class in the system. With all of the active sensors using AVL trees for the storage of their respective nodes it made sense to continue the trend.

H. LAYER-4 ISOLATOR

The layer-4 isolator is a passive sensor. Like the layer-3 isolator the layer-4 isolator is designed to be used as a base class for all isolators that need to isolate a layer-4 stream. The layer-4 isolator is a derived class from the layer-3 isolator. This allows the layer-3 isolator class to perform the checks of the packets at layer-3 through a call to layer3synch(). If the packet matches at layer-3 then layer4synch is called. The layer4synch function call checks to ensure that the packet is of the correct layer-4 type. The layer-4 packet type as well as the layer-3 type is passed in to the layer-4 constructor. The layer-4 constructor in turn calls the layer-3 constructor giving the layer-3 constructor the correct packet type information.

Like the layer-3 isolator the layer-4 isolator is designed such that it is only a base class for the development of layer-4 isolators. The web traffic isolator talked about later in this chapter is an example of how the layer-4 and layer-3 isolators work together to form a base for stream isolators.

The layer-4 isolator constructor takes a layer-3 type, layer-4 protocol, and a port number as arguments. Alternatively this information can be supplied later by only specifying a layer-3 type to the constructor. Once initialized the layer4synch function becomes available for use. This does a very similar function to the layer3synch function in that it tests the incoming packet to ensure that it is of the correct type that the isolator is looking for in terms of proper protocol and service. An additional feature available at layer-4 is the ability to apply a mask to the addresses being examined. This mask is applied to the IP addresses before the isolator node is built. This gives the isolator the ability to view traffic in terms of belonging to a subnet mask. The layer-4 isolator class also provides the buildTempNode() function that is used to build the temporary node that is used in searching the AVL tree.

I. WEB TRAFFIC ISOLATOR

The web traffic isolator is an example of how the layer-3 and layer-4 isolators work together to form the base for a higher-level stream isolator. The purpose of the traffic isolator is to sort out all port 80 web traffic, classify it in terms of directed connections, and finally route packets according to the frequency with which the packets belonging to a connection are seen by the system. The idea is to be able to separate out low data rate connections from normal traffic connections.

The web traffic isolator looks for all TCP packets that have a source or destination port of 80. As with active sensors the web traffic isolator has an analyzePacket function that is called from the sensor base class for each packet that is captured by the system. The web isolator makes use of the layer-4 isolator as well as the layer-3 isolator for identifying the packet as being of interest.

The analysis of a packet by the web isolator is done in a similar manner to that of the active sensors. The packet is tested against the layer3synch function, and upon success is tested against the layer4synch function. If both of these functions return true

the packet is of interest to the isolator. This shows how the web traffic isolator builds on top of the layer-4 isolator, which in turn builds on top of what the layer-3 isolator does.

Once a packet is identified as the correct type for the isolator, a temporary isolator node is built with the information from the packet. Next, the AVL tree from the layer-3 isolator is searched for the node. If a match is made a pointer to the node in the AVL tree is established. If a match is not made then the temporary node is added to the AVL tree and another search is done to ensure that the node has indeed been added to the tree. In testing it was found that if this additional check was not done the node would occasionally not be added to the tree.

With a pointer to the isolator node in the AVL tree, the node is then "touched" to increment the counter in the node. The value of the current time bin is then used to decide which interface to send the packet out. To make this decision a simple threshold is used. If the current time bin value is greater than the threshold for slow traffic it is sent out the normal traffic interface. When a connection is first seen the value of the previous time bin is set to –1 so that a new connection is not sent to the slow traffic until it is seen enough to be sent to the normal traffic interface. This is done so that a normal connection does not mistakenly get sent to the slow interface simply because it had not been seen before.

J. OBJECTIVE-C SENSORS

The Objective-C implementation makes use of a very similar sensor architecture. The differences are noted here. The Objective-C implementation uses NSThreads instead of pthreads. This is of little consequence since NSThreads are built on top of pthreads. The pthread_cond_t data type is replaced with a NSConditionLock data type. The AVL trees have been replaced with an NSDictionaries. The Dictionary uses a hash table as its underlying data type giving faster insertion than an AVL tree. There is no fuzzy logic functionality in the Objective-C implementation. The BufferNode, and IsolatorNode data types are essentially the same only ported to Objective-C. The logic for each is identical to its C++ equivalent. The same is true for the Layer-4 isolator in the Objective-C implementation.

In this chapter the sensors and major abstract data types of the prototypes were discussed in detail. Additionally, multi-threading of the prototype was also discussed.

Source code for all the sensors and abastract data types is included in the this thesis in the form of an appendix. The next chapter details the testing of the stream splitter and conclusions reached upon completion of the testing.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIMENT AND RESULTS

The goal of this thesis was to show that a robust architecture for traffic separation could be implemented. To do this there were two things that had to be shown: (1)the splitter must be able to keep up with network traffic and (2)it must also implement a separation scheme that would not be possible with a router. If both of these criteria are not met then the value of the splitter is greatly diminished. Two tests were devised that would show each of these necessities, a capture efficiency test and a test to show accurate traffic separation.

A. CAPTURE EFFICIENCY

To test the ability of the splitter to capture network traffic,Tcpreplay (tcpreplay.sourceforge.net) was used to replay a tcpdump file from the 1999 DARPA IDS Evaluation done at MIT. Specifically, the week one Tuesday inside dump data was used. Tcpreplay gives the option of specifying how fast to replay the file. The testing started at 5 Mb/s and then increased in 5Mb/s increments to 75 Mb/s. After the file had been completely replayed the capture engine could be examined to see how many packets it had captured. This test was run with the Objective-C version of the splitter, the C++ version of the splitter, and also with Snort 1.9 and Snort 2.0. The base configuration of snort was used in both cases. Snort 1.9 was tried with both ascii logging and binary logging, while snort 2.0 used only binary logging. All tests were run on a Macintosh dual 1.42Ghz G4 with 2 gigabytes of RAM. The test setup is shown in Figure 10.

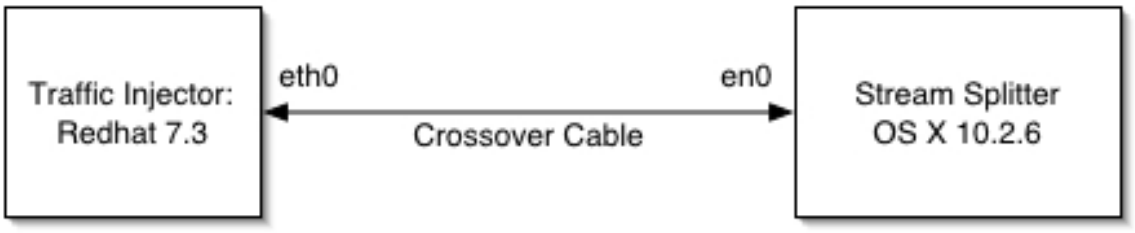

Figure 10. Packet Capture Test Setup

This first test looked only at how many packets the splitter could capture out of the stream of packets that was sent to it. Analysis of the packets takes longer than the capture process so once the stream had been sent, analysis was halted and the number of packets sent to the buffer of the splitter was taken to be the number of packets that was captured. This does not reflect the number of packets that can be captured and processed during continuous operation. When the splitter was stopped there were often a significant number of packets in the buffer waiting to be processed. Only packets that were analyzed by snort were counted since if the packet is dropped prior to analysis it will not be analyzed at all as snort does not buffer packets as the stream splitter does.

Snort, which is implemented in C, outperformed both the Objective-C version of the splitter and the C++ version of the splitter. The Objective-C splitter outperformed the C++ version. This makes sense because in C++ there will be virtual table lookups that will take time to complete whereas with Objective-C, method calls are cached at run time to increase performance. The results of this test are detailed in figure 11. After running this test it became apparent that a hardware solution to capture traffic is necessary for any bandwidth greater than 30 Mb/s, which corresponds to roughly 10,000 packets per second.

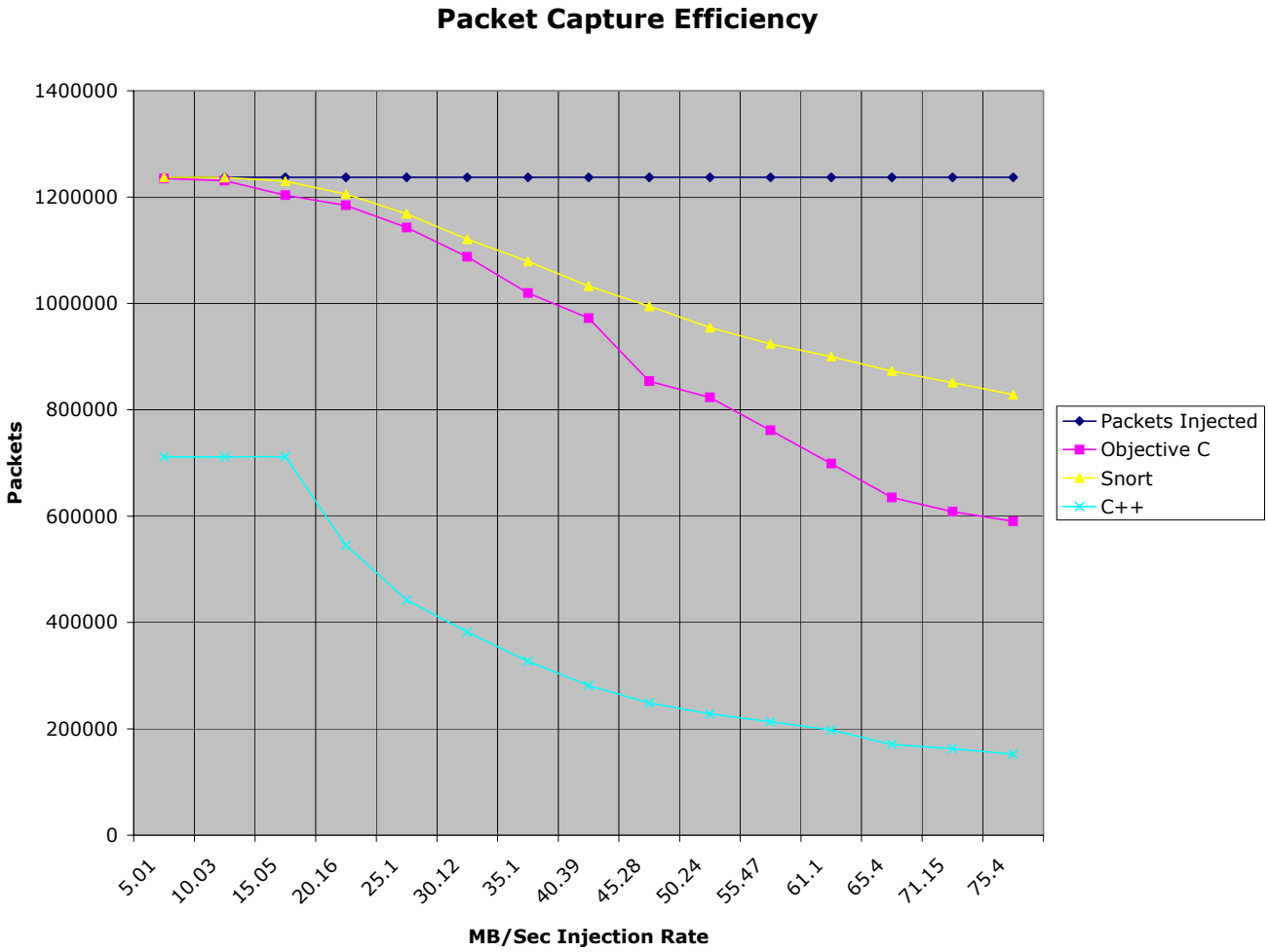

Figure 11. Packet Capture Efficiency

As mentioned earlier one of the differences between the two implementations is that the C++ version makes use of AVL trees, the Objective-C version uses hash tables in the form of NSDictionaries. During testing it became clear that the balancing of the AVL trees was too expensive. The program would periodically appear to hang. After attaching a thread monitor to the program the sensor thread was continuously operating but the dispatch thread was not. This meant that the sensor thread was trying to analyze just one packet. For the experiment the only analysis being done was simply finding the correct node in the AVL tree and then incrementing a counter. If the node did not exist in the tree it must be added to the tree. Adding a node causes the tree to be rebalanced. I also found that during testing occasionally the AVL tree would not find a node that had been added

to the tree and would attempt to re-add it. This would result in an additional rebalance of the tree.

B. TRAFFIC SEPARATION

To show that the splitter can implement a traffic separation scheme an experiment was devised that made use of the IDS test bed already set up at the Naval Postgraduate School. This test bed uses a Smart Bits 6000 chassis and six two-port traffic generation blades. Each blade can be configured to send out a variety traffic. For this experiment, two ports were used on the Smart Bits 6000 chassis, one for fast traffic and one for slow traffic. The output of the splitter was passed to an eight port Linksys Switch. Also connected to the switch were interface three and interface four of a Dell 2650 server running Windows 2000 Server. A diagram of the network is shown in figure 12.

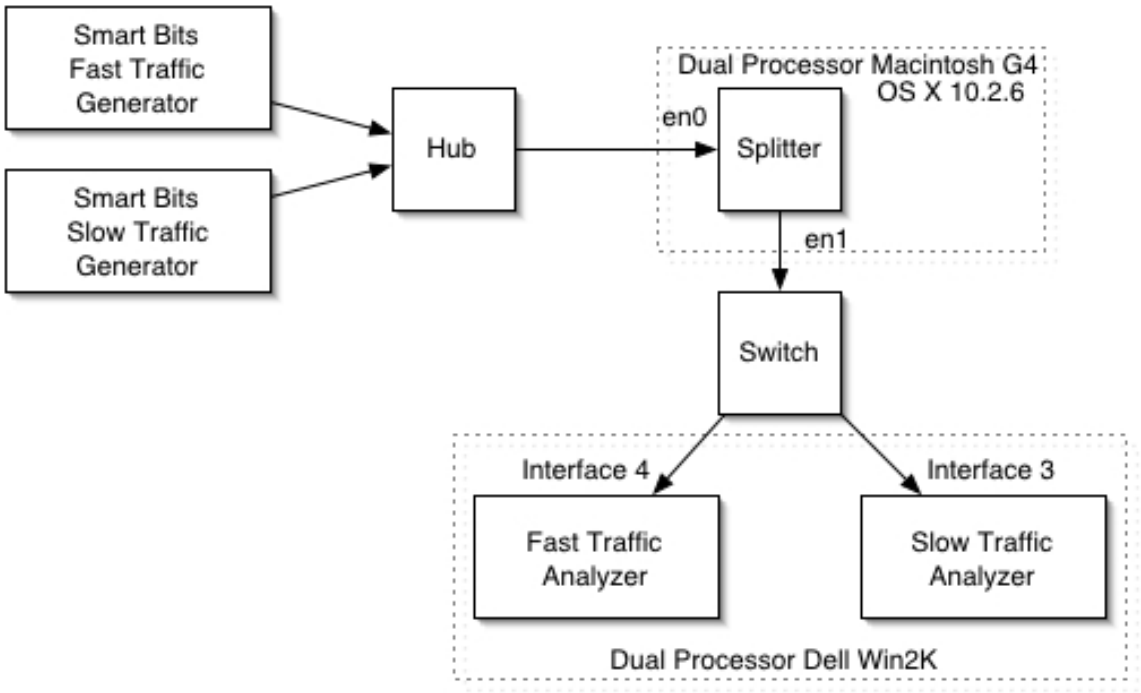

Figure 12. Test Bed Setup

The splitter was configured so that it would look for TCP traffic utilizing port 80. The window size was set to ten seconds with a high/low traffic threshold of 100 packets. The IP addresses were chosen so that when viewed in a large table they would stand out from one another. The test traffic is detailed in table 1. MAC information is not shown in table 1 as the splitter overwrites the destination MAC address in the process of routing the traffic.

Traffic Stream	Source IP	Destination IP	Source Port	Destination Port
Fast	10.10.10.1	10.10.10.2	80	11000
Slow	192.168.10.10	190.168.10.20	80	12000

Table 1. Traffic Stream Description

Since the splitter and both generators could not be started at the same time the first step in the experiment was to start the traffic generators. With the traffic generators running the stream splitter was then configured and started. Configuration of the splitter consisted of setting the following parameters:

- MAC addresses for the two interface on the windows server.
- A threshold level of 100.
- A service port of 80.
- A ten second time window.
- A protocol type of TCP.

The traffic generators were then supplying traffic to the splitter and the splitter was processing the traffic. Ethereal was then allowed to capture traffic on both of the monitored interfaces for sixty seconds. This experiment was run three times. The results are shown in table 2.

Test Number	Fast Traffic Packets	Slow Traffic Packets
1	6000	120
2	5900	120
3	5900	120

Table 2. Traffic Capture Test Results

These results are what was expected since the starting and stopping of Ethereal could not be synchronized with the traffic generators the fast traffic should be within 100 packets of 6000 and the slow traffic should be within 2 packets of 120.

It should be noted that in order for the switch to be able to route the packets by the destination MAC address, each interface had to send data through the switch prior to the

experiment. This was accomplished by simply pinging out each interface on the Dell computer to a nonexistent IP address. This allows the switch to learn the interfaces that are connected to it. When a packet containing the MAC address of a connected interface arrives at the switch it is routed through the correct port to the attached interface. In this case one of two interfaces on the Dell server.

C. SUMMARY

The two experiments conducted in this chapter demonstrated the ability of the prototype splitter to achieve the objectives of traffic capture and separation in environments typically seen at organizational ingress points. The next chapter summarizes the design findings of the splitter and discusses future areas of related research.

VI. CONCLUSION AND RECOMMENDATIONS

Through the design of two software-based prototypes and follow on experimentation, we have shown that it is possible to isolate a stream of interest from a larger network stream. Further, through the development of the prototype, a robust architecture was established and was shown to be capable of isolating a network stream. Future work will undoubtedly center on optimization of the architecture and additional uses for the stream splitter.

A. OPTIMIZATION

1. Internal Data Structures

The current Objective-C stream splitter is a proof of concept prototype. As such it was coded for correctness not for speed. Hash tables were used for the internal data structures that hold connection descriptions. These structures can grow quite large and there may be a better data structure that would lead to an improvement in system performance. A better performing data structure would allow for an increase in system performance.

All of the buffers used in the splitter are linked lists. Like the hash table these too may not be the most appropriate choice for this type of data storage. The decision to use a linked list was based on the ability of the linked list to grow as more packets are captured. Use of the linked list allows the buffers to grow constrained only by the system. This was done because during testing it became clear that the capture buffer was going to grow quite large with network speeds greater than 20Mb/s.

2. Data Capture

The packet capture test showed that if the splitter is to be used in a modern operational network then the packet capture functionality would have to be improved. The limiting factor for network capture is currently the pcap packet capture library. In order to make better use of the network interface a faster method for packet capture must be found. Until then there will always be lost packets so any meaningful analysis of a connection must take into account the packets that were missed by the capture engine. For instance when a connection is made the third part of a three-way handshake may be

missed. This condition must be dealt with or there will not be a way to accurately model the network and perform analysis on it.

If the data capture functionality could be moved out of software and into a hardware implementation, system performance would be greatly improved. Further analysis of the libpcap library could lead to an answer as to why all the snort test results were so similar. This could be a result of the libpcap library being the limiting factor for system performance.

B. FUZZY LOGIC

The fuzzy logic model was not implemented in the Objective-C stream splitter. Future work could port the C++ fuzzy model to Objective-C and include it in the objective-c stream splitter. The fuzzy logic model in its current form only looks at frequency of an IP address and the frequency of the connection. This could be expanded to look at other aspects of network traffic.

If the stream splitter is used for a different type of system the fuzzy logic model could prove to be valuable. The fuzzy logic allows the splitter to do more intelligent routing of streams. An increase in the complexity of the fuzzy logic model could yield better results.

C. ADDITIONAL USES

This paper dealt with a network architecture where the basic data unit is the Ethernet frame. This architecture can be expanded to any system in which a base data type can be defined. For instance a hard drive may be scanned in using the file as the basic data type. Using this splitter scheme the hard drive may be mined for any type of data. The payoff is that the hard drive only has to be read in once. After being read the files are then separated into similar types. In this manner a large amount of information may be processed.

Using a similar technique to the hard drive problem above, the splitter could be adapted to a variety of data mining applications. Wading through log files comes to mind as an area of use. Set the splitter at the log collection facility and then let it sort through all the incoming logs. With the ease of configuration it could make classifying and sorting logs a much easier task.

LIST OF REFERENCES

[1] Cormen, Thomas, Leisserson, Charles, Rivest, Ronald, Stein, Clifford, Introduction to Algorithms Second Edition, McGraw-Hill, Boston, 2001

[2] Joshua W. Haines, Richard P. Lippmann, David J. Fried, Eushiuan Tran, Steve Boswell, Marc A. Zissman, "1999 DARPA Intrusion Detection System Evaluation: Design and Procedures", MIT Lincoln Laboratory Technical Report, Technical Report 1062

[3] Joshua Haines, Lee Rossey, Rich Lippmann and Robert Cunnigham, "Extending the 1999 Evaluation", In the Proceedings of DISCEX 2001, June 11-12, Anaheim, CA.

[4] Lippmann, R.P., Fried D.J., Graf, I., Haines, J.W., Kendall, K.R., McClung, D., Weber, D., Webster, S.E., Wyschogrod, D., Cunningham, R.K., Zissman, M.A., DARPA Information Survivability Conference and Exposition, 2000. DISCEX 00. Proceedings , Volume: 2 , 1999 Page(s): 12 -26 vol.2

[5] Newman, David, Snyder, Joel, Thayer, Rodney "Eight IDSs fail to impress during the monthlong test on a production network.", http://www.nwfusion.com/techinsider/2002/0624security1.html, Network World, 06/24/02

[6] Pedrycz, Witold, Fuzzy Modelling Paradigms and Practice, Kluwer Academic Publishers, Massachusetts, 1996

[7] Timm, Kevin "Strategies to Reduce False Positives and False Negatives" http://online.securityfocus.com/infocus/1463 , September 11, 2001

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. C++ AVLTREE.H

```
// AVL Tree Implementation
// (c) Copyright 2002 William A. McKee.  All rights reserved.
//
//
// The template class AVL::Node (used in AVL::Tree) requires the following:
//
//   T::T (const T &);
//   T::~T ();
//   bool T::operator < (const T & rhs) const;
//   bool T::operator == (const T & rhs) const;
//
// For printing purposes one must declare:
//
//   ostream & operator << (const ostream &, const T &);
//

//
// iostream is required for printing only.
//
#ifndef __AVLTREECODE_H
#define __AVLTREECODE_H

#include <iostream>

//
// Use "namespace" to make sure the class names don't conflict with other code.
//

namespace AVL {

//
// The basic unit of currency in a tree are the nodes that comprise it.
//
template <class T>
class Node
{
public :

// This is were we keep the data we want to store in each node.
// It is const because if you change it while it is in the tree structure
// you compromise the integrity of the tree.
// It is public because the Tree class must have access to it in order
// to return it after being found with the found_node function.
```

```
    T data;

private :

// Each node has two children: left and right.  If they are both NULL then
// the node is a leaf node.  Otherwise, it's an interior node.

   Node<T> * left, * right;

// The height is computed to be: 0 if NULL, 1 for leaf nodes, and the maximum
// height of the two children plus 1 for interior nodes.
// This is used to keep the tree balanced.

   int height;

   void compute_height ()
   {
      height = 0;
      if (left != NULL && left -> height > height)
         height = left -> height;
      if (right != NULL && right -> height > height)
         height = right -> height;
      height += 1;
   }

// The constructor is private because the nodes are self allocating.

   Node (const T & inData)
      : data (inData), left (NULL), right (NULL), height (1)
   {
   }

public :

// Recursively delete the children if this node is being nuked.

   ~Node ()
   {
      delete left;
      delete right;
   }

// Recursively insert some data into the tree then balance it on the way up.

   Node<T> * insert_node (const T & inData)
```

```
    {
      if (this == NULL)
         return new Node<T> (inData);

      if (inData < data)
         left = left -> insert_node (inData);
      else
         right = right -> insert_node (inData);
      return balance ();
    }

// Recursively find some data in the tree and if found return a pointer
// to the node containing the data.  If not found then return NULL.

    Node<T> * find_node (const T & inData) //const
    {
      if (this == NULL)
         return NULL;

      if (inData == data)
         return this;

      if (inData < data)
         return left -> find_node (inData);
      else
         return right -> find_node (inData);
    }

// Recursively search the tree for some data and if found remove (delete) it.
// When you remove an interior node the right child must be place right of
// the right most child in the left sub-tree.
// Remember to balance the tree on the way up after removing a node.

    Node<T> * remove_node (const T & inData)
    {
      if (this == NULL)
         return NULL;

      // we found the data we were looking for

      if (inData == data)
      {
         // save the children

         Node<T> * tmp = left -> move_down_righthand_side (right);
```

```
            // by setting the children to NULL, we delete exactly one node.

            left = NULL;
            right = NULL;
            delete this;

            // return the reorganized children

            return tmp;
         }

         if (inData < data)
            left = left -> remove_node (inData);
         else
            right = right -> remove_node (inData);
         return balance ();
      }

// Recursively print out all nodes in order (left to right).

      void print_node (std::ostream & co) const
      {
         if (this == NULL)
            return;

         left -> print_node (co);

         co << data << " ";

         right -> print_node (co);
      }

private :


// move_down_righthand_side is the remove_node helper function:
//
// Recursively find the right most child in a sub-tree and put
// the "rhs" sub-tree there.
// Re-balance the tree on the way up.

      Node<T> * move_down_righthand_side (Node<T> * rhs)
      {
         if (this == NULL)
            return rhs;
```

```
      right = right -> move_down_righthand_side (rhs);
      return balance ();
   }

//
// Balancing a tree (or sub-tree) requires the AVL algorithm.
//
// If the tree is out of balance left-left, we rotate the node to the right.
// If the tree is out of balance left-right, we rotate the left child to the
// left and then rotate the current node right.
// If the tree is out of balance right-left, we rotate the right child to the
// right and then rotate the current node left.
// if the tree is out of balance right-right, we rotate the node to the left.
//

   Node<T> * balance ()
   {
      int d = difference_in_height ();

      // only rotate if out of balance
      if (d < -1 || d > 1)
      {
         // too heavy on the right
         if (d < 0)
         {
            // if right child is too heavy on the left,
            // rotate right child to the right
            if (right -> difference_in_height () > 0)
               right = right -> rotate_right ();

            // rotate current node to the left
            return rotate_left ();
         }
         // too heavy on the left
         else
         {
            // if left child is too heavy on the right,
            // rotate left child to the left
            if (left -> difference_in_height () < 0)
               left = left -> rotate_left ();

            // rotate current node to the right
            return rotate_right ();
         }
      }
```

```
        // recompute the height of each node on the way up
        compute_height ();

        // otherwise, the node is balanced and we simply return it
        return this;
    }

// ** balancing helper functions **

    Node<T> * exchange_left (Node<T> * & r, Node<T> * node)
    {
        r = left;
        left = node -> balance ();
        return balance ();
    }

    Node<T> * exchange_right (Node<T> * & l, Node<T> * node)
    {
        l = right;
        right = node -> balance ();
        return balance ();
    }

    int difference_in_height ()
    {
        int left_height = (left != NULL) ? left -> height : 0;
        int right_height = (right != NULL) ? right -> height : 0;
        return left_height - right_height;
    }

    Node<T> * rotate_left ()
    {
        return right -> exchange_left (right, this);
    }

    Node<T> * rotate_right ()
    {
        return left -> exchange_right (left, this);
    }

};

//
// Cover class for maintaining the tree.
//
// Since Node<T> is self allocating and self deleting, the Tree<T> class
```

```
// ensures that only qualified calls are made.
//
// Tree<T> is the public interface to the AVL Tree code.
// Node<T> is not meant to be used by the public.
//
// This code makes use of the somewhat dubious practice of calling a member
// function with a NULL "this" pointer.  We will not run into problems since
// we have no virtual member functions in Node<T>.
//

template <class T>
class Tree
{
private :

   Node<T> * root;

public :

   Tree ()
   {
      root = NULL;
   }

   ~Tree ()
   {
      delete root;
   }

   void insert (const T & inData)
   {
      root = root -> insert_node (inData);
   }

   T * find (const T & inData) const
   {
      Node<T> * found = root -> find_node (inData);
      if (found != NULL)
         return & found -> data;
      else
         return NULL;
   }

   void remove (const T & inData)
   {
      root = root -> remove_node (inData);
```

```
    }

    void print (std::ostream & co) const
    {
        root -> print_node (co);
    }

};

//
// Declare a useful extention to the output stream convention for
// the Tree<T> class.
//

template <class T>
std::ostream & operator << (std::ostream & co, const Tree<T> & tree)
{
    tree.print (co);
    return co;
}

// end of namespace AVL

}
#endif
//__AVLTREECODE_H
```

APPENDIX B. C++ KASHA.H

```
#ifndef __PCAPCLASS_H
#define __PCAPCLASS_H

extern "C"{
//#include <pcap.h>

}

#include <libnet.h>
#include "Sensor.h"

#include <iostream>
#include <iomanip>
#include <stdio.h>
#include "net/ethernet.h"

#include <pthread.h>

#include "KashaHeaders.h"
#include "KashaBufferNode.h"
#include <list.h>
/*Sensor Headers*/
#include "KashaStats.h"
#include "KashaSrcSensor.h"
#include "KashaConnectionSensor.h"
#include "KashaUDPSensor.h"
#include "KashaTCPServiceIsolator.h"
#include "KashaWebIsolator.h"
using namespace std;
class Kasha {
public:
   Kasha();
   ~Kasha();

   static void myCallback(u_char *user,struct pcap_pkthdr *pph, u_char *pdata);
   void bufferPacket(struct pcap_pkthdr *pph, u_char *pdata);
private:

   int sendPacketEth2();

   int sendPacketEth1();
```

```
void sensorInit();

float suspectMembershipFunc();

float normalMembershipFunc();

void forwardPacket();


char errbuf[LIBNET_ERRBUF_SIZE];

void processPacket();

void analyzeResults();

static void * dispatcherStart(void * arg);

void dispatcherRun();

//libnet initializations
libnet_t * eth2;
libnet_t * eth1;

//packet info
u_char * pdata;
struct pcap_pkthdr * pph;
struct ether_header * eh;
struct my_tcp * tcp;
struct my_ip * ip;

int numSensors;
unsigned long numPackets;
long int packetsQueued;
time_t tempTime;

pthread_t threadID;
pthread_mutex_t * allocMutex;

Sensor * sensors[1];
int sensorWeight[1];

pthread_mutex_t * bufferMutex;
pthread_cond_t * bufferCond;
list<KashaBufferNode *> buffer;
KashaBufferNode * bufferNodePtr;
```

```
    float normalMembership;
    float suspectMembership;


};

#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. C++ KASHA.CPP

```
#include "Kasha.h"
/*Constructor*/
Kasha::Kasha(){
   pthread_mutex_init(allocMutex, NULL);
   eth1 = libnet_init(LIBNET_LINK_ADV,"eth1",errbuf);
   eth2 = libnet_init(LIBNET_LINK_ADV,"eth2",errbuf);
   normalMembership = -1;
   suspectMembership = -1;
   numPackets = 0;
   numSensors = 1;//3;
   packetsQueued = 0;
   sensors[0] = new KashaWebIsolator(2);
   sensorWeight[0] = 1;
   for(int i = 0;i<numSensors;i++)
      pthread_mutex_lock(sensors[i]->mutex);

   bufferMutex = new pthread_mutex_t;
   bufferCond = new pthread_cond_t;

   pthread_mutex_init(bufferMutex, NULL);
   pthread_mutex_init(allocMutex, NULL);
   pthread_cond_init(bufferCond, NULL);
   sensors[0]->setAllocMutex(allocMutex);
   pthread_create(&threadID,NULL,dispatcherStart,(void*)this);
}


Kasha::~Kasha(){
   for(int i=0;i<numSensors;i++){
     delete sensors[i];
   }
   pthread_mutex_destroy(bufferMutex);
   pthread_cond_destroy(bufferCond);
   pthread_mutex_destroy(allocMutex);
   while(buffer.size() > 0){
      buffer.pop_front();
   }
}

void Kasha::myCallback(u_char *user,struct pcap_pkthdr *pph, u_char *pdata){
   Kasha *ptr = (Kasha*)user;
   ptr->bufferPacket(pph, pdata);
```

```
}

inline void Kasha::processPacket(){
for(int i = 0;i<numSensors;i++){
        sensors[i]->reset();
        pthread_mutex_unlock(sensors[i]->mutex);
        pthread_cond_broadcast(sensors[i]->cond);
    }
    for(int i=0;i<numSensors; ){
        pthread_mutex_lock(sensors[i] -> mutex);
        while(sensors[i] -> normalMembership == -1)
            pthread_cond_wait(sensors[i]->cond, sensors[i]->mutex);
        i++;

    }
}


/*****************************************************************/
/*These are the functions used to route the packet once the      */
/*the packet has been analyzed and a trust decision has been made*/
/*****************************************************************/
int Kasha::sendPacketEth2(){
  return libnet_adv_write_link(eth2,pdata,pph->len);
}

int Kasha::sendPacketEth1(){
    return libnet_adv_write_link(eth1,pdata,pph->len);
}

void Kasha::forwardPacket(){
int status1;
int status2;
float normalThreshold = .5;
float suspectThreshold = .5;
    if(normalMembership >= normalThreshold )
        status1 = sendPacketEth1();
    if(suspectMembership >= suspectThreshold)
        status2 =  sendPacketEth2();
    if(!(normalMembership >= normalThreshold)&&
       !(suspectMembership >= suspectThreshold))
       status2 = sendPacketEth2();
}
/*****************************************************************/
/*Currently there is no need for any initialization other than   */
/*simply declaring the sensors that are to be used but if you    */
```

```
/*need to do some sort of initialization this would be a good    */
/*place to do it :-) Don't Forget Your towel!                    */
/*****************************************************************/
void Kasha::sensorInit(){}

/*****************************************************************/
/*These two functions compile all the results from the sensors   */
/*****************************************************************/
float Kasha::suspectMembershipFunc(){
   float result = 0;
   float totalWeight = 0;
   // cout<<"Suspect Input ";
   for(int i =0;i<numSensors;i++){
      //cout<<sensors[i]->suspectMembership<<" ";
      if(sensors[i]->suspectMembership != SENSOR_IGNORE){
         totalWeight += sensorWeight[i];
         result += sensors[i]->suspectMembership * sensorWeight[i];
      }
   }
  // cout<<endl;
   return result/totalWeight;
}

float Kasha::normalMembershipFunc(){
   float result = 0;
   float totalWeight;
   //cout<<"normal input: ";
   for(int i =0;i<numSensors;i++){
     // cout<<sensors[i]->normalMembership<<" ";
      if(sensors[i]->normalMembership != SENSOR_IGNORE){
         totalWeight += sensorWeight[i];
         result = result + sensors[i]->normalMembership * sensorWeight[i];
      }
   }
   //cout<<endl;
   return result/totalWeight;
}

/*****************************************************************/
/*This is the function that compiles all the results from the    */
/*sensors. When it is finished it will unlock all mutexes.       */
/*MUTEXES MUST BE LOCKED BEFORE CALLING THIS FUNCTION!!!!        */
/*****************************************************************/
void Kasha::analyzeResults(){
   normalMembership = normalMembershipFunc();
   suspectMembership = suspectMembershipFunc();
```

```
}

//void Kasha::bufferPacket(struct pcap_pkthdr *pph, const u_char *pdata){
inline void Kasha::bufferPacket(struct pcap_pkthdr *pph, u_char *pdata){
   numPackets++;
   pthread_mutex_lock(allocMutex);
   KashaBufferNode* node = new KashaBufferNode(pph,pdata);
   pthread_mutex_unlock(allocMutex);

   pthread_mutex_lock(bufferMutex);
   packetsQueued++;
   buffer.push_back(node);
   pthread_mutex_unlock(bufferMutex);

   if(packetsQueued == 1)
   pthread_cond_broadcast(bufferCond);

}

void* Kasha::dispatcherStart(void * arg){
Kasha * ptr = (Kasha *)arg;
ptr->dispatcherRun();
return NULL;
}

inline void Kasha::dispatcherRun(){
KashaBufferNode * node;
u_int32_t waiting = 0;
cout<<"Dispatcher Running"<<endl;

while(true){
   pthread_mutex_lock(bufferMutex);
   while(buffer.front() == NULL){
      pthread_cond_wait(bufferCond,bufferMutex);
   }

   node = buffer.front();
   pthread_mutex_unlock(bufferMutex);
   while(node!=NULL){
   waiting++;
   cout<<numPackets<<endl;

   Sensor::setPacketData(&node->pktHdrPtr, node->packetPtr, node->arrivalTime);
   processPacket();
   pthread_mutex_lock(bufferMutex);
   buffer.pop_front();
```

```
    packetsQueued--;
    node = buffer.front();
    pthread_mutex_unlock(bufferMutex);
    }

}//end while
}//end dispatcher Run


//end of file
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. C++ KASHAADDRNODE.H

```
/*
 *  KashaAddrNode.h
 *  Kasha
 *
 *  Created by John Judd on Sat Feb 15 2003.
 *
 *TO DO:
 *implement purge of list
 *implement membership functions
 */

#ifndef KashaAddrNode_h
#define KashaAddrNode_h
#include <stdio.h>
#include <netinet/in.h>
#include "time.h"
#include <list.h>
#include <stdlib.h>
#include <ostream.h>
#include <libnet.h>
class KashaAddrNode{
public:
   KashaAddrNode(u_long inputAddress){
      address = inputAddress;
   }

   ~KashaAddrNode(){

      while(IPlist.size()>0){
         IPlist.pop_front();
      }
   }
//IP address that is used as the key for the node
   u_long address;

//overloaded operators
   bool operator == (const KashaAddrNode & rhs) const{
      return address == rhs.address;
   }
   bool operator < (const KashaAddrNode & rhs) const{
      return address < rhs.address;
   }
   bool operator > (const KashaAddrNode & rhs) const{
```

```
        return address > rhs.address;
    }
    bool operator >= (const KashaAddrNode & rhs) const{
        return address >= rhs.address;
    }
    bool operator <= (const KashaAddrNode & rhs) const{
        return address <= rhs.address;
    }
   friend ostream& operator<<(ostream & outs, KashaAddrNode & ptr){
          outs<<libnet_addr2name4(ptr.address, LIBNET_DONT_RESOLVE)<<"
"<<ptr.IPlist.size()<<endl;
        return outs;
   }


 //adds current time to the list of times on this node most
//current timestamp is added to the end of the list.
    void addCurrentTimeStamp(time_t &tempTime){
        purgeList(-1,tempTime);
        if(difftime(tempTime, IPlist.front())> 20*60){
            IPlist.push_back(tempTime);
        }
    }

//removes timestamps that are out of the window
    bool purgeList(float timeInterval, time_t tempTime){
    //if timeInterval passed in is 0 then use 30 days
        if (timeInterval == -1)
            timeInterval = (30*24*60*60);
       while(difftime(tempTime, IPlist.front())> timeInterval
              && IPlist.size() > 0){
            IPlist.pop_front();
        }
        return true;
    }

    list< time_t > IPlist;
    list< time_t >::iterator iter;


};

#endif
```

APPENDIX E. C++ KASHABUFFERNODE.H

```
/*
 *  KashaBufferNode.h
 *  Kasha
 *
 *  Created by System Administrator on Thu Mar 13 2003.
 *.
 *
 */
#ifndef KASHABUFFERNODE_H
#define KASHABUFFERNODE_H
#include "KashaHeaders.h"
extern "C"{
#include <pcap.h>
}
#include <iomanip>
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
using namespace std;
class KashaBufferNode{
public:
   KashaBufferNode();
   KashaBufferNode(struct pcap_pkthdr * pph, u_char * data);
   ~KashaBufferNode();
   u_char * packetPtr;
   time_t arrivalTime;
   struct pcap_pkthdr pktHdrPtr;

};
#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. C++ KASHABUFFERNODE.CPP

```
/*
 *  KashaBufferNode.cpp
 *  Kasha
 *
 *  Created by John Judd on Thu Mar 13 2003.
 *
 */

#include "KashaBufferNode.h"

KashaBufferNode::KashaBufferNode(){}

KashaBufferNode::KashaBufferNode(struct pcap_pkthdr * pph, u_char * data){
   memcpy(&pktHdrPtr, pph,sizeof(struct pcap_pkthdr));
   packetPtr = new u_char[pktHdrPtr.caplen];
   memcpy(packetPtr, data, pktHdrPtr.caplen);
   time(&arrivalTime);
}

KashaBufferNode::~KashaBufferNode(){
   if(packetPtr != NULL)
      delete [] packetPtr;

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G. C++ KASHACONNECTIONNODE.H

```
/*
 *  KashaConnectionNode.h
 *  Kasha
 *
 *  Created by John Judd on Wed Feb 19 2003.
 *.
 *
 */

#ifndef __KASHACONNECTIONNODE_H
#define __KASHACONNECTIONNODE_H

#include <time.h>
#include <list.h>
#include "KashaHeaders.h"
#include <libnet.h>
using namespace std;
class KashaConnectionNode{
public:
    KashaConnectionNode();
    KashaConnectionNode(int inputAddWindow, int inputLiveWindow);
    bool purgeList(int,time_t);
    void addCurrentTimestamp(time_t & tempTime);
    bool operator == (const KashaConnectionNode & rhs) const;
    bool operator < (const KashaConnectionNode & rhs) const;
    friend ostream & operator << (ostream &outs, const KashaConnectionNode & ptr);
    u_long src;
    u_long dest;
    u_int16_t service;

    int size;
    list< time_t >::iterator iter;
    list<time_t> timestampList;

private:
    //seconds before next timestamp can be added to list
    int timestampAddWindow;
    //seconds for timestamp to live on list
    int timestampLiveWindow;
};

#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX H. C++ KASHACONNECTIONNODE.CPP

```
/*
 *  KashaConnectionNode.cpp
 *  Kasha
 *
 *  Created by John Judd on Wed Feb 19 2003.
 *
 */

#include "KashaConnectionNode.h"

KashaConnectionNode::KashaConnectionNode(){
   src = 0;
   dest = 0;
   service = 0;
   size = 0;
   //private members
   timestampAddWindow = 1;//20*60; //20 minutes
   timestampLiveWindow = 30*24*60*60;//30 days
}

KashaConnectionNode::KashaConnectionNode(int inputAddWindow, int
inputLiveWindow){
   src = 0;
   dest = 0;
   service = 0;
   size = 0;
   //private members
   timestampAddWindow = inputAddWindow;
   timestampLiveWindow = inputLiveWindow;
}

bool KashaConnectionNode::operator < (const KashaConnectionNode & rhs) const{
   //order IPs for both nodes
   //comparison is SRC>DEST>SERVICE
   u_long aBig = src;
   u_long aSmall;
   if(aBig < dest){
      aBig = dest;
      aSmall = src;
   }
   u_long bBig = rhs.src;
   u_long bSmall;
   if(bBig < rhs.dest){
```

```
        bBig = rhs.src;
        bSmall = rhs.src;
    }
    if (aBig == bBig){
        if (aSmall == bSmall){
                return service < rhs.service;
            }
        else{ return aSmall < bSmall;}
    }
    else{
    return aBig < bBig;
    }

    cout<<"error computing < KashaConnectionNode";
    return src < rhs.src;
    }// end bool <

bool KashaConnectionNode::operator == (const KashaConnectionNode & rhs) const{
    return (src == rhs.src &&
    rhs.dest == rhs.dest &&
     service == rhs.service)
            ||
    (dest == rhs.dest &&
    dest == rhs.src &&
    service == rhs.service);
}

ostream & operator << (ostream &outs, const KashaConnectionNode & ptr){
    outs<<"Source "<<libnet_addr2name4(ptr.src,LIBNET_DONT_RESOLVE)<< " "<<
          libnet_addr2name4(ptr.dest,LIBNET_DONT_RESOLVE)<< " "
          <<" numtimestamps "<<ptr.timestampList.size()<<endl;
    return outs;
}


//adds a timestamp to the list if it can be added.
//A time stamp will only be added as long as timstampAddWindow
//seconds have gone by since the last time stamp was added
void KashaConnectionNode::addCurrentTimestamp(time_t & tempTime){
    //iter = timestampList.begin();
    purgeList(200000,tempTime);
    cout<<difftime(tempTime,timestampList.front() )<<endl;
    if(difftime(tempTime,timestampList.front() )> 10 ||
    timestampList.size() == 0){
        size++;
        timestampList.push_back(tempTime);
```

```
    }
    cout<<"added Timestamp "<<tempTime<<" "<<timestampList.size()<<endl;
}

//removes timestamps that are out of the window
bool KashaConnectionNode::purgeList(int purgeInterval,time_t tempTime){
    while(difftime(tempTime, timestampList.front())> purgeInterval
          && timestampList.size() > 0){
          timestampList.pop_front();
          size--;
       }
       return true;
    }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX I. C++ KASHACONNECTIONSENSOR.H

```
/*
 *  KashaConnectionSensor.h
 *  Kasha
 *
 *  Created by John Judd on Tue Feb 18 2003.
 *
 * Problems: How to check for connection failure
 * I am thinking watchdog timer.
 *
 */
#ifndef __KASHAHEADERS_H
#define __KASHAHEADERS_H

#include <libnet.h>
#include <assert.h>
#include "Sensor.h"
#include "KashaHeaders.h"
#include "KashaConnectionNode.h"
#include "avltree.h"
#include <pthread.h>

//#include <stdlib.h>

#include "net/ethernet.h"

class KashaConnectionSensor: public Sensor{

public:
   KashaConnectionSensor();
   virtual ~KashaConnectionSensor();
   //analysis function called by Kasha.cpp

   void buildTempNode();

   void syncLocalVariables();

protected:
   //temporary connection variables
   u_long srcAddr;
   u_long destAddr;
   u_short service;
   u_short protocol;
```

```
    int purgeInterval;
    int timeBetweenListEntries;
    int listWindowSize;

    long int numPackets;
    //pointer to data structures
    struct my_tcp * tcpHeader;
    struct my_ip * ipHeader;
    struct ether_header * eh;
/******tree for storing connections******/
    AVL::Tree<KashaConnectionNode> connectionTree;

//temp node used for searching trees
    KashaConnectionNode temp;
//pointer to object returned by tree search
    KashaConnectionNode  * ptr;


private:
    float normalMembershipFunc(KashaConnectionNode *);
    float suspectMembershipFunc(KashaConnectionNode *);
    void analyzePacket();

};

#endif
```

APPENDIX J. C++ KASHACONNECTIONSENSOR.CPP

```
/*
 *  KashaConnectionSensor.cpp
 *  Kasha
 *
 *  Created by John Judd on Tue Feb 18 2003.
 *
 */
#include "KashaConnectionSensor.h"

KashaConnectionSensor::KashaConnectionSensor():Sensor(){
   srcAddr = 0;
   destAddr = 0;
   purgeInterval = 30*24*60*60;
   timeBetweenListEntries = 20*60;
   listWindowSize = 24*3;
   service = 0;
   protocol = 0;
   normalMembership = 0;
   suspectMembership = 0;
   numPackets = 0;

   output.open("connectionSensor.txt", ios::out);
}


KashaConnectionSensor::~KashaConnectionSensor(){
}

void KashaConnectionSensor::analyzePacket(){
   numPackets++;
   sensorBail();
   return;
   ipHeader = (struct my_ip *)(pdata + sizeof(struct ether_header));
   protocol = ipHeader->ip_p;
   if(protocol == TCP){
      syncLocalVariables();
      buildTempNode();
      ptr = NULL;
      ptr = connectionTree.find(temp);
      if(ptr != NULL){
         ptr->addCurrentTimestamp(tempTime);

         normalMembership = normalMembershipFunc(ptr);
```

```
        suspectMembership = suspectMembershipFunc(ptr);

    }//end 2nd level if
    else{
    /*because when we build temp node we set the direction
    *based on the location of the service port in the packet
    *If we make it to this code it is either a new packet
    *or it is using a service port for both the source and Destination
    *port numbers and we will have to handle both of these cases.
    */
        if(tcpHeader -> tcp_src_port > 1024 && tcpHeader-> tcp_dest_port > 1024){
            //this would be where we lookup the service in the efemeral table.
            //if it existed. If match then we add to the connection tree if not
            //then we assume the packet is bad and should skew the result that way
            normalMembership = .9;
            suspectMembership = 1.1;
            /*
            output<<"unusual ports! "<<tcpHeader -> tcp_src_port<< " " <<
            tcpHeader-> tcp_dest_port<<endl;
            */
        }//end 3rd level if
        else{
        /*the packet is part of a new connection this is
        *the only place an add to the tree is done this
        *should be threaded here since this will be threaded
        *for now I am just going to add the new temp object.
        */
        //add the time to the node
        connectionTree.insert(temp);
        ptr = connectionTree.find(temp);
        while(ptr == NULL){
        connectionTree.insert(temp);
        ptr = connectionTree.find(temp);
        }
        ptr->addCurrentTimestamp(tempTime);
        //insert the node into the tree

        normalMembership = normalMembershipFunc(ptr);
        suspectMembership = suspectMembershipFunc(ptr);

        ptr = &temp;

        }
    }//end 2nd level else

}//end 1st level if
```

```
   else{
   //not a tcp packet
      normalMembership = SENSOR_IGNORE;//need to do something here
      suspectMembership = SENSOR_IGNORE;//need to do something here
      //output<<"non TCP"<<endl;
   }//end 1st level else
}//end analyzePacket



void KashaConnectionSensor::buildTempNode(){
   if(htons(tcpHeader -> tcp_src_port) < 1024 ||
      htons(tcpHeader-> tcp_dest_port) < 1024){

       if(tcpHeader -> tcp_src_port > tcpHeader-> tcp_dest_port){
         temp.service = tcpHeader->tcp_dest_port;
         temp.src = srcAddr;
         temp.dest = destAddr;
       }//end second level if
       else{
         temp.service = tcpHeader -> tcp_src_port;
         temp.src = destAddr;
         temp.dest = srcAddr;
       }//end 2nd level else
   }//end 1st level if
   else{
   /*the service is an efemeral port
   *We should now check the efemeral service table that
   *doesn't exist yet!
   *for now just put in something
++++THIS IS A PROBLEM HERE!!! FIX ME !!!++++
   */
      temp.service = tcpHeader -> tcp_src_port;
      temp.src = destAddr;
      temp.dest = srcAddr;
   }//end  1st level else

}

void KashaConnectionSensor::syncLocalVariables(){
   tcpHeader = (struct my_tcp*)(pdata + sizeof(struct ether_header) +
      ((ipHeader->ip_vhl)&htons(0x0f))*4);
   srcAddr = ipHeader->ip_src.s_addr;
   destAddr = ipHeader->ip_dest.s_addr;
   service = ipHeader->ip_tos;
```

```
}

float KashaConnectionSensor::normalMembershipFunc(
KashaConnectionNode * nodePtr){
   if(nodePtr == NULL){
   cerr<<"bad ptr membership func"<<endl;
   exit(1);
   }
   float size = nodePtr -> timestampList.size();
   if(size > 0)
      if(size/listWindowSize < 1)
         return size/listWindowSize;
      else
         return 1;
   else
   return .1;
}

float KashaConnectionSensor::suspectMembershipFunc(
KashaConnectionNode * nodePtr){
   if(nodePtr == NULL){
         cerr<<"bad ptr membership func"<<endl;
         exit(1);
   }

   float size = nodePtr -> timestampList.size();
   if(size > 0)
      if (listWindowSize/size > 1)
         return 1;
      else
         return listWindowSize/size;
   else
   return 1.5;

}
```

APPENDIX K. KASHAHEADERS.H

```
/*
 *
 *
 *
 *  Created by John Judd on Fri Jan 24 2003.
 *   Thanks to Chris Eagle for the formats and the start of this
 *    file.
 */
#ifndef kashaheaders_h
#define kashaheaders_h
#include <arpa/inet.h>
#include <netinet/in.h>
#include <time.h>
#ifndef LOG
#define LOG
#endif
struct my_ip {
        u_int8_t        ip_vhl;         /* header length, version */
#define IP_V(ip)        (((ip)->ip_vhl & 0xf0) >> 4)
#define IP_HL(ip)       ((ip)->ip_vhl & 0x0f)
        u_int8_t        ip_tos;         /* type of service */
        u_int16_t       ip_len;         /* total length */
        u_int16_t       ip_id;          /* identification */
        u_int16_t       ip_off;         /* fragment offset field */

#define IP_DF 0x4000
   /* dont fragment flag */
#define IP_MF 0x2000
   /* more fragments flag */
#define IP_OFFMASK 0x1fff
   /* mask for fragmenting bits */

#define TCP 6
#define UDP 17

        u_int8_t        ip_ttl;         /* time to live */
        u_int8_t        ip_p;           /* protocol */
        u_int16_t       ip_sum;         /* checksum */
        struct in_addr ip_src,ip_dest;  /* source and dest address */
};

struct my_tcp {
        u_int16_t       tcp_src_port;
```

```
        u_int16_t       tcp_dest_port;
        u_int32_t       tcp_seq_num;
        u_int32_t       tcp_ack_num;
        u_int8_t        tcp_dataOff;
#define TCP_DATAOFF(tcp)        ((tcp)->tcp_dataOff >> 2)
        u_int8_t        tcp_flags;
#define TCP_UF 0x20
   /* urgent flag */
#define TCP_AF 0x10
   /* ack flag */
#define TCP_PF 0x08
   /* push flag */
#define TCP_RF 0x04
   /* reset flag */
#define TCP_SF 0x02
   /* syn flag */
#define TCP_FF 0x01
   /* fin flag */
#define IP_MF 0x2000
   /* more fragments flag */

        u_int16_t       tcp_window;
        u_int16_t       tcp_checksum;
        u_int16_t       tcp_urgptr;
};

struct my_udp {
   u_int16_t       udp_src_port;
   u_int16_t       udp_dest_port;
   u_int16_t        udp_length;
   u_int16_t        udp_checksum;

};

struct KashaBufferNode_t{
  u_char * packetPtr;
  time_t arrivalTime;
  struct pcap_pkthdr * pktHdrPtr;
};

/*
Sensor defines
*/
#define SENSOR_IGNORE -2
#define SENSOR_RESET -1
```

```
#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX L. C++ KASHAL3ISOLATOR.H

```
/*
 *  KashaIsolator.h
 *  Kasha
 *
 *  Created by John Judd on Tue Mar 18 2003.
 *
 */
#ifndef KASHAISOLATOR_H
#define KASHAISOLATOR_H

//#include <stdio.h>
#include "KashaHeaders.h"
#include "Sensor.h"
#include "KashaIsolatorNode.h"
#include "avltree.h"

class KashaL3Isolator: public Sensor{
public:
   KashaL3Isolator(u_short type);
   bool layer3SyncEH();
   bool layer3Sync();
   void analyzePacket();
   AVL::Tree<KashaIsolatorNode> tree;
   KashaIsolatorNode tempNode;
   KashaIsolatorNode * nodePtr;

protected:
   struct ether_header * eh;
   struct my_ip * ip;
   u_int32_t src;
   u_int32_t dest;
   u_short etherType;
   u_int32_t mask;

private:

};
#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX M. C++ KASHAL3ISOLATOR.CPP

```
/*
 *  KashaL3Isolator.cpp
 *  Kasha
 *
 *  Created by John Judd on Tue Mar 18 2003.
 *
 */

#include "KashaL3Isolator.h"

KashaL3Isolator::KashaL3Isolator(u_short type){
   etherType = type;
   mask = 0xffffffff;

}

inline bool KashaL3Isolator::layer3SyncEH(){
   eh = (struct ether_header *)(pdata);
   if(eh != NULL)
      return true;
   return false;
}

bool KashaL3Isolator::layer3Sync(){
   /*it would be nice to check for IP version here*/
   if (!layer3SyncEH())
      return false;
   if(eh->ether_type != ntohs(etherType))
      return false;
   ip = (struct my_ip *)(pdata + sizeof(struct ether_header));
   if(ip != NULL)
      return true;
   return false;
}

void KashaL3Isolator::analyzePacket(){
//do something here Bebblebrox or inherit from this class
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX N. C++ KASHAL4ISOLATOR.H

```
/*
 *  KashaL4Isolator.h
 *  Kasha
 *
 *  Created by John Judd on Wed Mar 19 2003.
 *
 */

//#include <Carbon/Carbon.h>
#ifndef KASHAL4ISOLATOR_H
#define KASHAL4ISOLATOR_H
#include <stdio.h>//needed for u_char
#include "net/ethernet.h"
#include "KashaL3Isolator.h"



class KashaL4Isolator: public KashaL3Isolator {
public:
   KashaL4Isolator():KashaL3Isolator(ETHERTYPE_IP){}
   KashaL4Isolator( u_short layer3type, u_int8_t protocolIn, u_int16_t portIn);
   bool layer4Sync();
   void setMask(u_int32_t newMask);
   void analyzePacket();

protected:
   void buildTempNode();
   u_int8_t protocol;
   u_int16_t port;
   struct my_tcp * tcp;
   /*Mask for ip addresses*/
};

#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX O. C++ KASHAL4ISOLATOR.CPP

```
/*
 *  KashaL4Isolator.cpp
 *  Kasha
 *
 *  Created by John Judd on Wed Mar 19 2003.
 *
 */

#include "KashaL4Isolator.h"

KashaL4Isolator::KashaL4Isolator(u_short layer3type, u_int8_t protocolIn, u_int16_t
portIn)
    :KashaL3Isolator(layer3type){
    protocol = protocolIn;
    port = portIn;
}


void KashaL4Isolator::setMask(u_int32_t newMask){
    mask = newMask;
}
/*returns true so long as all properties match Layer 2-4*/
bool KashaL4Isolator::layer4Sync(){
    if(!layer3Sync())
        return false;
    if(ip->ip_p != protocol)
        return false;
    tcp = (struct my_tcp *)(pdata +  sizeof(struct ether_header) +
        ((ip->ip_vhl)&0x0f)*4);
    if(tcp->tcp_src_port != port && tcp->tcp_dest_port != port)
        return false;
    return true;
}

inline void KashaL4Isolator::buildTempNode(){
   if(tcp->tcp_src_port==port){
        tempNode.dest = (ip->ip_src.s_addr)&mask;
        tempNode.src = (ip->ip_dest.s_addr)&mask;
    }
      tempNode.src = (ip->ip_src.s_addr)&mask;
      tempNode.dest = (ip->ip_dest.s_addr)&mask;
}
```

```
void KashaL4Isolator::analyzePacket(){
   //sensor is passive
   //add to counter

}
```

APPENDIX P. C++ SENSOR.H

```
/*
 *  Sensor.h
 *  Kasha
 *
 *  Created by John Judd on Tue Mar 04 2003.
 *
 */
#ifndef __SENSOR_H
#define __SENSOR_H

extern "C"{

#include <pcap.h>
}
#ifndef LOG
#define LOG
#endif

//file io includes
#include <iostream>
#include <fstream.h>
#include <assert.h>

#include <time.h>
#include "avltree.h"
#include "KashaHeaders.h"
#include "net/ethernet.h"

#include <semaphore.h>
#include "net/ethernet.h"
using namespace std;
class Sensor{
public:
   Sensor();

   virtual ~Sensor();
   static void * threadFunc(void * arg);

   static void setPacketData(const struct pcap_pkthdr * hdr,
                  u_char * pkt, time_t myTime);
   void run();
   virtual void analyzePacket(){}
```

```
    void reset();

    void sensorBail(){normalMembership = 1;suspectMembership = 1;}
    void setAllocMutex(pthread_mutex_t * am){allocMutex = am;}
    float normalMembership;
    float suspectMembership;
    pthread_t threadID;
    pthread_mutex_t * allocMutex;
    pthread_mutex_t * mutex;
    pthread_cond_t * cond;
    ofstream output;
protected:
    static const struct pcap_pkthdr * header;
    static u_char * pdata;
    static time_t tempTime;
};
#endif
```

APPENDIX Q. C++ SENSOR.CPP

```
/*
 *  Sensor.cpp
 *  Kasha
 *
 *  Created by John Judd on Tue Mar 04 2003.
 *
 */

#include "Sensor.h"
#include <pthread.h>


Sensor::Sensor(){
    cond = new pthread_cond_t;
    mutex = new pthread_mutex_t;
    pthread_mutex_init(mutex,NULL);
    pthread_cond_init(cond,NULL);
    pthread_create(&threadID,NULL,threadFunc,(void*)this);
    normalMembership = 0;
}

Sensor::~Sensor(){
output.close();
pthread_cond_destroy(cond);
pthread_mutex_destroy(mutex);
}
void * Sensor::threadFunc(void * arg){
    Sensor * ptr = (Sensor *)arg;
    ptr->run();
    return NULL;
}

void Sensor::setPacketData(const struct pcap_pkthdr * hdr,
                u_char * pkt,time_t myTime){
    header = hdr;
    pdata = pkt;
    tempTime = myTime;

}

void Sensor::reset(){
    normalMembership = SENSOR_RESET;
```

```
    suspectMembership = SENSOR_RESET;
}
void Sensor::run(){
    while(true){
    pthread_mutex_lock(mutex);
        while(normalMembership != SENSOR_RESET)
            pthread_cond_wait(cond, mutex);
    analyzePacket();
    pthread_mutex_unlock(mutex);
    pthread_cond_broadcast(cond);
    }
}


const struct pcap_pkthdr * Sensor::header;
u_char * Sensor::pdata;
time_t Sensor::tempTime;
```

APPENDIX R. C++ KASHASRCSENSOR.H

```
/*
 *  KashaSrcSensor.h
 *  Kasha
 *
 *  Created by John Judd on Tue Feb 04 2003.
 *
 */
#ifndef KASHASRCSENSOR_H
#define KASHASRCSENSOR_H


#include <iostream>
#include <fstream.h>
#include <assert.h>

#include <time.h>
#include "avltree.h"
#include "net/ethernet.h"

#include "KashaAddrNode.h"
#include "KashaHeaders.h"
#include <iomanip>

#include "Sensor.h"
using namespace std;
/*
*To DO:
*1. implement a tree wide purge on a set interval
*2. get/set for purgeInterval
*3. get/set for listWindowSize
*4. overloaded constructor
*/
class KashaSrcSensor : public Sensor {
public:
   KashaSrcSensor();
   ~KashaSrcSensor();
// analysis function called by Kasha.cpp
   void analyzePacket();

   void setPurgeInterval(float interval);


private:
```

```
    void setLastPurge();
    void extractIP();

//Membership Functions
    float suspectMembershipFunc( KashaAddrNode *);
    float normalMembershipFunc( KashaAddrNode *);

//AVL tree that stores all IP addresses
    AVL::Tree<KashaAddrNode> tree;

// IP address that is used to search tree
    u_long IP;

//time stamp that is set when a purge of list is run
    time_t lastPurge;

//sliding window for purging timestamps contained in nodes
    float purgeInterval;

//size of list window for a trusted connection
//72 by default, a connections seen every 20 minutes for a day.
    int listWindowSize;

    long int numPackets;
   //ofstream output;

};

#endif
```

APPENDIX S. C++ KASHASRCSENSOR.CPP

```
/*
 *  KashaSrcSensor.cpp
 *  Kasha
 *
 *  Created by John Judd on Tue Feb 04 2003.
 *
 */

#include "KashaSrcSensor.h"

KashaSrcSensor::KashaSrcSensor():Sensor(){
//pass in 0 so that we get the default 30 days
    purgeInterval = 0;
//set listWindowSize to default of 72 or a connection
//seen every 20 minutes for a day
    listWindowSize = 72;
    numPackets = 0;
   output.open("srcSensorLog.txt",ios::out);

}

KashaSrcSensor::~KashaSrcSensor(){
}
/*Membership functions*/
float KashaSrcSensor::suspectMembershipFunc( KashaAddrNode * node){
    //what to do if not seen often, how to calculate
    //72 is 24 * 3; so a connection seen every 20 minutes for a day
    float size = node->IPlist.size();

    if(size > 0)
        if (listWindowSize/size > 1)
            return 1;
        else
            return listWindowSize/size;
    else
    return 1.5;
}

float KashaSrcSensor::normalMembershipFunc( KashaAddrNode * node){
    //what to do if seen often
    float size = node->IPlist.size();
    if(size > 0)
        if(size/listWindowSize < 1)
```

```
            return size/listWindowSize;
        else
            return 1;
    else
    return .1;
}

void KashaSrcSensor::setPurgeInterval(float interval){
    purgeInterval = interval;
}

void KashaSrcSensor::setLastPurge(){}

void KashaSrcSensor::analyzePacket(){
    numPackets++;
    sensorBail();

    output<<numPackets<<endl;
   // return;
    extractIP();
    KashaAddrNode temp(IP);
    KashaAddrNode * ptr;
    ptr = tree.find(temp);
    if(ptr != NULL){
        //IP has been seen before and is in the tree

        ptr->addCurrentTimeStamp(tempTime);


output<<tempTime<<"Update|"<<
libnet_addr2name4(IP,  LIBNET_DONT_RESOLVE);
output<<" | listLength: "<<ptr->IPlist.size()<<endl;
     }
    else{
    //First sighting of this IP
        temp.addCurrentTimeStamp(tempTime);
        tree.insert(temp);
        ptr = tree.find(temp);
    }
    //update return location;
    normalMembership = normalMembershipFunc(ptr);
    suspectMembership = suspectMembershipFunc(ptr);
}

void KashaSrcSensor::extractIP(){
     struct my_ip * ip = (struct my_ip *)(pdata + sizeof(struct ether_header));
```

```
IP = (u_long)ip->ip_src.s_addr;
        }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX T. C++ KASHAWEBISOLATOR.H

```
/*
 *  KashaWebIsolator.h
 *  Kasha
 *
 *  Created by John Judd on Mon Mar 24 2003.
 *
 */

#ifndef KASHAWEBISOLATOR_H
#define KASHAWEBISOLATOR_H
#include "KashaL4Isolator.h"
#include <libnet.h>
#include <iostream>

class KashaWebIsolator:public KashaL4Isolator{
public:

   KashaWebIsolator(u_int16_t numInterfaces);

   /*called by sensor base class. main analysis function*/
   void analyzePacket();

   /*Builds the temp node to search the (L-3)tree for*/
   void buildTempNode();

   /*Writes data from node to a file named according
      to the IP addresses in the current Isolator node
      */
   void writeToFile();

   /*Writes packet to wire depending on the bins in the current
      isolator node that is being pointed at.*/
   void writeToWire();

   /*useed for libnet error messages*/
   char errbuf[LIBNET_ERRBUF_SIZE];

   /*used for file name when writing data to file*/
   char charPtr[37];

   /*character pointers used to get string representation of the
      IP addresses used in the file name*/
   u_char * s1;
```

```
    u_char * s2;

private:
    /*libnet instances for packet injection*/
    //libnet_t * * interfaceArray;
    libnet_t * fast;
    libnet_t * slow;

    /*Threshlod value*/
    u_int32_t threshold;
    u_int32_t nodeCount, nodeLastBin;
};
#endif
```

APPENDIX U. C++ KASHAWEBISLOATOR.CPP

```
/*
 *  KashaWebIsolator.cpp
 *  Kasha
 *
 *  Created by John Judd on Mon Mar 24 2003.
 *
 */

#include "KashaWebIsolator.h"
KashaWebIsolator::KashaWebIsolator(u_int16_t
numInterfaces):KashaL4Isolator(ETHERTYPE_IP, TCP, 80){
  mask = 0xffffffff;
  slow = libnet_init(LIBNET_LINK_ADV,"en1",errbuf);
  fast = libnet_init(LIBNET_LINK_ADV,"en2",errbuf);
  threshold = 100;
  nodeCount = 0;
  nodeLastBin = 0;
}

inline void KashaWebIsolator::buildTempNode(){
   if(tcp->tcp_src_port==port){
        tempNode.dest = (ip->ip_src.s_addr)&mask;
        tempNode.src = (ip->ip_dest.s_addr)&mask;
        return;
    }
      tempNode.src = (ip->ip_src.s_addr)&mask;
      tempNode.dest = (ip->ip_dest.s_addr)&mask;
}

inline void KashaWebIsolator::analyzePacket(){
    normalMembership = SENSOR_IGNORE;
    suspectMembership = SENSOR_IGNORE;
    return;
    if(!layer4Sync()){
        //cout<<"-";
        //cout<<"----"<<header->len<< " "<<header->caplen<<endl;
        return;
        }
   // cout<<endl;
    //cout<<"webIsolatorMatch"<<endl;
    buildTempNode();
    nodePtr = NULL;
```

```
   nodePtr = tree.find(tempNode);
   if(nodePtr == NULL){
      //must add a new node
      cout<<"fail lookup"<<endl;
      //char c;
      //cin >> c;
      while(nodePtr == NULL){
         tree.insert(tempNode);
         nodePtr = tree.find(tempNode);
      }
   }//end if
   //found existing node
   nodePtr->touchNode(tempTime);
   nodeCount = nodePtr->timeBin[nodePtr->currentBin];
   nodeLastBin = nodePtr->lastBin;
   //if(nodePtr->listSize > 1)
   //    writeToFile();
   //writeToWire();
}

/*
based on current node writes all unwritten data in node to file
 Status: complete
 */
inline void KashaWebIsolator::writeToFile(){
   s1 = libnet_addr2name4(nodePtr->src,LIBNET_DONT_RESOLVE);
   int ct=0;
   for(int i =0;s1[i] != NULL; i++){
      ct++;
   }
   s2 = libnet_addr2name4(nodePtr->dest,LIBNET_DONT_RESOLVE);
   for(int i =0;s2[i] != NULL; i++){
      ct++;
   }
   int ct2 =0;
   for(int i =0;s1[i] != NULL; i++){
      charPtr[i] = s1[i];
      ct2++;
   }
   charPtr[ct2] = '-';
   ct2++;
   charPtr[ct2] = '>';
   ct2++;
   for(int i =0;s2[i] != NULL; i++){
      charPtr[ct2] = s2[i];
      ct2++;
```

```
    }
    charPtr[ct2] = '.';
    ct2++;
    charPtr[ct2] = 'x';
    ct2++;
    charPtr[ct2] = 'l';
    ct2++;
    charPtr[ct2] = 's';
    ct2++;
    charPtr[ct2] = NULL;
 //   pthread_mutex_lock(allocMutex);
    ofstream tempOutput;
    tempOutput.open(charPtr,ios::app);

//    pthread_mutex_unlock(allocMutex);

    int temp = 0;
    while(nodePtr->listSize > 0){
        temp = nodePtr->binList.front();
        nodePtr->binList.pop_front();
        //cout<<"adding "<<temp<<" "<<charPtr<<endl;
        tempOutput<<temp<<",";
        nodePtr->listSize--;
    }
    tempOutput.close();
}

/*Writes the current packet to the wire sending to either
interface 1 or 2 based on the current information in the isolator node
being pointed at.
*/
inline void KashaWebIsolator::writeToWire(){
    int code = -1;
    if(nodeCount < threshold && nodeLastBin < threshold && nodeLastBin != 0){
      // cout<<"<<->>"<<charPtr<<" C: "<<nodeCount<<" B: "<<
//nodePtr->numBinsProcessed<<
      // " T: "<<difftime(tempTime, nodePtr->timeStart)<<" LB: "<<nodeLastBin<<endl;
       //if(header->len > 1500)
          //code =libnet_adv_write_link(slow, pdata, 1500);
       //else
          //code = libnet_adv_write_link(slow, pdata, header->len);
       //if(code <0){
        //   cout<<code<<" "<<libnet_geterror(slow)<<errbuf<<endl;
        //   cout<<header->len<<" "<<header->caplen<<endl;
        //   cout<<ip->ip_len<<endl;
      // }
```

```
    }
    else
    {
       //cout<<"+";
       //cout<<"++++"<<charPtr<<" C: "<<nodePtr->timeBin[nodePtr->currentBin]
//<<" B: "<<nodePtr->numBinsProcessed<<
       //" T: "<<difftime(tempTime, nodePtr->timeStart)<<" L: "<<header->len<<endl;

       //if(header->len > 1500)
          // code = libnet_adv_write_link(fast, pdata, 1500);
       // else
         //  code = libnet_adv_write_link(fast, pdata, header->len);

       //if(code <0){
         //  cout<<code<<" "<<libnet_geterror(fast)<<errbuf<<endl;
         //  cout<<header->len<<" "<<header->caplen<<endl;
         //  cout<<ip->ip_len<<endl;
       //}
    }

}
```

APPENDIX V. C++ KASHAUDPSENSOR.H

```
/*
 *  KashaUDPSensor.h
 *  Kasha
 *
 *  Created by John Judd on Wed Feb 26 2003.
*
 */

#ifndef __KASHAUDPSENSOR_H
#define __KASHAUDPSENSOR_H

#include "KashaHeaders.h"
#include "KashaConnectionNode.h"
#include "avltree.h"
#include "Sensor.h"

//#include <stdlib.h>

#include "net/ethernet.h"


class KashaUDPSensor: public Sensor{
public:
   KashaUDPSensor();
   virtual ~KashaUDPSensor();
   void analyzePacket();
private:
   void buildTempNode();
   void logNewPacket();
   void logUpdate();
   float normalMembershipFunc(KashaConnectionNode *);
   float suspectMembershipFunc(KashaConnectionNode *);
   //temporary connection variables
   u_long srcAddr;
   u_long destAddr;
   short protocol;
   KashaConnectionNode temp;
   KashaConnectionNode * ptr;

   //pointer to data structures
   struct my_udp * udpHeader;
   struct my_ip * ipHeader;
```

```
    int timeBetweenListEntries;
    int purgeInterval;
    long int numPackets;
    int listWindowSize;
    AVL::Tree<KashaConnectionNode> tree;

};
#endif
```

APPENDIX W. C++ KASHAUDPSENSOR.CPP

```
/*
 *  KashaUDPSensor.cpp
 *  Kasha
 *
 *  Created by John Judd on Wed Feb 26 2003.
 *
 */
#include "KashaUDPSensor.h"
KashaUDPSensor::KashaUDPSensor (): Sensor(){
   timeBetweenListEntries = 20*60;
   purgeInterval = 30*24*60*60;
   numPackets = 0;
   listWindowSize = 72;
   output.open("UPDSensorLog.txt",ios::out);
}

KashaUDPSensor::~KashaUDPSensor(){

}
void KashaUDPSensor::analyzePacket(){
   numPackets++;
   sensorBail();
   return;
   ipHeader = (struct my_ip *)(pdata + sizeof(struct ether_header));
   srcAddr = ipHeader->ip_src.s_addr;
   destAddr = ipHeader->ip_dest.s_addr;
   protocol = ipHeader->ip_p;
   ptr = NULL;
   if(protocol == UDP){
      udpHeader = (struct my_udp *)(pdata + sizeof(struct ether_header) +
((ipHeader->ip_vhl)&ntohs(0x0f))*4);
      buildTempNode();
      ptr = tree.find(temp);
      if(ptr != NULL){
         ptr->addCurrentTimestamp(tempTime);
         logUpdate();
      }
      else{
      temp.addCurrentTimestamp(tempTime);
      tree.insert(temp);
      ptr = &temp;
      logNewPacket();
      }
```

```
        normalMembership = normalMembershipFunc(ptr);
        suspectMembership = suspectMembershipFunc(ptr);
    }
    else{
    /*
    #ifdef LOG
    output <<tempTime<< " Non UDP Packet "<<endl;
    #endif
    */
    normalMembership = SENSOR_IGNORE;
    suspectMembership = SENSOR_IGNORE;
    }
}

void KashaUDPSensor::buildTempNode(){
    u_int16_t srcPort, destPort;
    srcPort = udpHeader -> udp_src_port;
    destPort = udpHeader-> udp_dest_port;
    if(srcPort < 1024 || destPort < 1024){
        if(srcPort > destPort){
          temp.service = destPort;
          temp.src = srcAddr;
          temp.dest = destAddr;
        }//end second level if
        else{
          temp.service = srcPort;
          temp.src = destAddr;
          temp.dest = srcAddr;
        }//end 2nd level else
    }//end 1st level if
    else{
    /*the service is an efemeral port
    *We should now check the efemeral service table that
    *doesn't exist yet!
    *for now just put in something
++++THIS IS A PROBLEM HERE!!! FIX ME !!!++++
    */
   // cout<<"greate than 1024! "<<endl;
        temp.service = udpHeader -> udp_src_port;
        temp.src = destAddr;
        temp.dest = srcAddr;
    }//end  1st level else

}

float KashaUDPSensor::normalMembershipFunc(KashaConnectionNode * nodePtr){
```

```
    float size = nodePtr->timestampList.size();
    if(size > 0)
        if(size/listWindowSize < 1)
            return size/listWindowSize;
        else
            return 1;
    else
    return .1;
}

float KashaUDPSensor::suspectMembershipFunc(KashaConnectionNode * nodePtr){
    float size = nodePtr->timestampList.size();
    if(size > 0)
        if (listWindowSize/size > 1)
            return 1;
        else
            return listWindowSize/size;
    else
    return 1.5;
}

void KashaUDPSensor::logNewPacket(){
     return;

}

void KashaUDPSensor::logUpdate(){
    return;
    #ifdef LOG
    output<<tempTime<<" UDP updated "<<
libnet_addr2name4(ptr->src,LIBNET_DONT_RESOLVE<<
 " "<<libnet_addr2name4(ptr->dest,LIBNET_DONT_RESOLVE)<<
 " "<<" #timestamps “ <<ptr->timestampList.size()<<endl;
    #endif

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX X. C++ MAIN.CPP

```
#include <iostream>
#include <stdio.h>
#include <signal.h>
extern "C"{
#include <pcap.h>
}

#include <pthread.h>
#include <net/ethernet.h>

#include "Kasha.h"

void catch_int(int sig_num);
int main (int argc,  char ** argv[]) {
   char errbuf[PCAP_ERRBUF_SIZE];
   pcap_t* descr;
   char *dev;
   Kasha pcapObj;
   signal(SIGINT,catch_int);
   dev = pcap_lookupdev(errbuf);
   /*Look up the device to capture on*/
   dev = "en0";
   descr = pcap_open_live(dev,BUFSIZ, -1,10,errbuf);
   /*Open a capture session*/
   pcap_loop(descr, -1, (pcap_handler)Kasha::myCallback, (u_char*)&pcapObj);
   /*register our call back function*/
   return 0;
}
void catch_int(int sig_num){
   //set signal mask here or race condition may occur
   /******THIS IS NOT DONE YET*****/
   cout<<"caught stop signal"<<endl;
   exit(1);
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX Y. C++ MAKEFILE

all:Kasha.h Kasha.cpp Sensor.h Sensor.cpp KashaStats.cpp KashaStats.h KashaSensor.cpp KashaSrcSensor.cpp KashaSrcSensor.h KashaConnectionSensor.h KashaConnectionSensor.cpp avltree.h KashaConnectionNode.h KashaTCPServiceIsolator.cpp KashaTCPServiceIsolator.h KashaConnectionNode.cpp KashaUDPSensor.h KashaUDPSensor.cpp

g++ -o KashaMake Kasha.cpp Sensor.cpp KashaStats.cpp KashaSrcSensor.cpp KashaConnectionSensor.cpp main.cpp KashaConnectionNode.cpp KashaUDPSensor.cpp KashaTCPServiceIsolator.cpp -lpthread -lnet -lpcap -Wno-deprecated -lstdc++

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX Z. OBJ-C BUFFERNODE.H

```
//
//  BufferNode.h
//  tranquility
//
//  Created by John Judd on Wed Apr 30 2003.
//

#import <Foundation/Foundation.h>
#import <pcap.h>
#import "Kashaheaders.h"


@interface BufferNode : NSObject {
@public
    //captured Packet from pcap
    NSMutableData * data;

    //time of arrival
    time_t arrivalTime;

    //pcap packet header
    struct pcap_pkthdr pktHdr;

    //Variable used for replay in injection engine
    int interface;
}
-(id)initWithPacket:(const u_char *)packetIn
    andPktHdr:(struct pcap_pkthdr *)pktHdrIn;
-(id)init;
-(id)initWithBufferNodePtr:(BufferNode *)node;

@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX AA. OBJ-C BUFFERNODE.M

```
//
//  BufferNode.m
//  tranquility
//
//  Created by John Judd on Wed Apr 30 2003.
//

#import "BufferNode.h"


@implementation BufferNode

-(id)initWithPacket:(const u_char *)packetIn
        andPktHdr:(struct pcap_pkthdr *)pktHdrIn{

   [super init];

   //copy pcap packet header
   memcpy(&pktHdr, pktHdrIn,sizeof(struct pcap_pkthdr));

   //initialize packet mem space
   data = [[NSMutableData alloc]initWithBytes:packetIn length:pktHdr.caplen];

   //set arrival time
   time(&arrivalTime);

   interface = 0;
   return self;

}

-(id)init{return self;}

-(id)initWithBufferNodePtr:(BufferNode *)node{
   return [self initWithPacket:[node->data bytes]
                  andPktHdr:&node->pktHdr];
}
@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX BB. OBJ-C CONTROLLER.H

```
/* Controller */

#import <Cocoa/Cocoa.h>
#import "Model.h"
#import <libnet.h>
#import "KashaHeaders.h"
@interface Controller : NSObject
{
    IBOutlet id binSize;
    IBOutlet id fastTrafficDestination;
    IBOutlet id model;
    IBOutlet id numberOfBins;
    IBOutlet NSTextField *packetsBuffered;
    IBOutlet id packetsCaptured;
    IBOutlet id packetsProcessed;
    IBOutlet id portField;
    IBOutlet id protocol;
    IBOutlet id slowTrafficDestination;
    IBOutlet id splitterStatus;
    IBOutlet id threshold;
}
- (IBAction)activateNewSensor:(id)sender;
- (IBAction)Reset:(id)sender;
- (IBAction)startSplitter:(id)sender;
- (IBAction)stopSplitter:(id)sender;
- (IBAction)updatePacketsBuffered:(id)sender;
- (IBAction)updatePacketsCaptured:(id)sender;
- (IBAction)updatePacketsProcessed:(id)sender;
@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX CC. OBJ-C CONTROLLER.M

```
#import "Controller.h"

@implementation Controller

- (IBAction)activateNewSensor:(id)sender
{
   float binSizeInput = [binSize floatValue];
   NSString * fast = [[NSString alloc]
         initWithString:[fastTrafficDestination stringValue]];
   NSString * slow = [[NSString alloc]
         initWithString:[slowTrafficDestination stringValue]];
   unsigned long numBins = [numberOfBins intValue];
   unsigned int proto = [protocol intValue];
   unsigned int thresh = [threshold intValue];
   unsigned int port = [portField intValue];
   [[model dispatcher] addSensor:[[L4Isolator alloc] initWithL4Type:proto
                                    andInjectionBuffer:[model injectionBuffer]
                                       andBufferLock:[model injectionBufferLock]
                                         andFastDest:fast
                                         andSlowDest:slow
                                          andBinSize:binSizeInput
                                       andNumberOfBins:numBins
                                      andFastThreshold:thresh
                                      andSlowThreshold:thresh
                                               andPort:port]];
}

- (IBAction)startSplitter:(id)sender
{
   NSString * string = [model startSplitter];
   [splitterStatus setStringValue:string];
}

- (IBAction)stopSplitter:(id)sender
{
   NSString * string = [model stopSplitter];
   [splitterStatus setStringValue:string];
}

- (IBAction)updatePacketsBuffered:(id)sender
{
   [packetsBuffered setIntValue:[model getNumberOfPacketsBuffered]];
}
```

```
- (IBAction)updatePacketsCaptured:(id)sender
{
   [packetsCaptured setIntValue:[model getNumberOfPacketsCaptured]];
}

- (IBAction)updatePacketsProcessed:(id)sender
{
   [packetsProcessed setIntValue:[model getNumberOfPacketsProcessed]];
}
- (IBAction)Reset:(id)sender{
   [[model captureEngine] numberOfPacketsCaptured:0];
   NSMutableArray * ap = [model captureBuffer];
   NSConditionLock * al = [model captureBufferLock];
   [al lock];
   [ap removeAllObjects];
   [al unlockWithCondition:NODATA];
   [packetsCaptured setIntValue:0];
   [packetsBuffered setIntValue:0];
}
@end
```

APPENDIX DD. OBJ-C MODEL.H

```
/* Model */

#import <Cocoa/Cocoa.h>
#import "CaptureEngine.h"
#import "InjectionEngine.h"
#import "Dispatcher.h"
#import "BufferNode.h"
#import "L4Isolator.h"
@interface Model : NSObject
{
@public
    CaptureEngine * captureEngine;
    InjectionEngine * injectionEngine;
    Dispatcher * dispatcher;

    NSMutableArray * captureBuffer;
    NSMutableArray * injectionBuffer;

    L4Isolator * sensor;
    //this is the lock that is given to both
    //the dispatcher and the capture engine
    NSConditionLock * captureBufferLock;
    NSConditionLock * injectionBufferLock;
}
-(NSString *)startSplitter;
-(NSString *)stopSplitter;
-(void)stub;
-(void)dealloc;
-(unsigned long)getNumberOfPacketsCaptured;
-(int)getNumberOfPacketsBuffered;
-(unsigned long)getNumberOfPacketsProcessed;
-(NSMutableArray *)injectionBuffer;
-(NSConditionLock *)injectionBufferLock;
-(NSMutableArray *)captureBuffer;
-(NSConditionLock *)captureBufferLock;
-(Dispatcher *)dispatcher;
-(CaptureEngine *)captureEngine;
@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX EE. OBJ-C MODEL.M

```
#import "Model.h"

@implementation Model
-(id)init{
   //initialize the buffer lock
   captureBufferLock =[[NSConditionLock alloc] initWithCondition:NODATA];
   injectionBufferLock = [[NSConditionLock alloc]initWithCondition:NODATA];

   //initialize buffer for captured packets
   captureBuffer= [[NSMutableArray alloc] init];

   //initialze injectionBuffer
   injectionBuffer = [[NSMutableArray alloc] init];

   //initialize the capture engine
   captureEngine = [[CaptureEngine alloc] initWithBuffer:captureBuffer
                                        andLock:captureBufferLock];

   //initialize injection engine
   injectionEngine = [[InjectionEngine alloc]initWithBuffer:injectionBuffer
                                    andInjectionLock:injectionBufferLock];
   //initialize dispatch engine
   dispatcher = [[Dispatcher alloc] initWithBuffer:captureBuffer
                                    andLock:captureBufferLock];

   //launch the stub thread to ensure isMultiThreaded is set
   [NSThread detachNewThreadSelector:@selector(stub) toTarget:self withObject:Nil];

   //launch capture thread
   [captureEngine start];
   [injectionEngine start];

   [dispatcher start];
   return self;
}

-(NSString *)startSplitter{
   [captureEngine startCapture];
   return @"running";
}

-(NSString *)stopSplitter{
   [captureEngine stopCapture];
```

```
    return @"stopped";
}

-(void)stub{
    NSAutoreleasePool * localPool = [[NSAutoreleasePool alloc] init];
    if([NSThread isMultiThreaded])
        NSLog(@"entering Multithreaded Mode");
    [localPool release];
}

-(void)dealloc{
    [captureEngine release];
    [captureBuffer release];
    [injectionBuffer release];
    [captureBufferLock release];
    [super dealloc];
}

-(unsigned long)getNumberOfPacketsCaptured{
    return [captureEngine numberOfPacketsCaptured];
}

-(int)getNumberOfPacketsBuffered{
    return [captureEngine currentBufferSize];
}

-(unsigned long)getNumberOfPacketsProcessed{
    return [dispatcher numberOfPacketsProcessed];
}

-(NSMutableArray *)injectionBuffer{
    return injectionBuffer;
}

-(NSConditionLock *)injectionBufferLock{
    return injectionBufferLock;
}

-(NSMutableArray *)captureBuffer{
    return captureBuffer;
}

-(NSConditionLock *)captureBufferLock{
    return captureBufferLock;
}
```

```
-(Dispatcher *)dispatcher{
    return dispatcher;
}

-(CaptureEngine *)captureEngine{
    return captureEngine;
}

@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX FF. OBJ-C INJECTIONENGINE.H

```
//
//  InjectionEngine.h
//  tranquility
//
//  Created by John Judd on Thu May 01 2003.
//
//

#import <Foundation/Foundation.h>
#import "libnet.h"
#import "BufferNode.h"
#import <net/ethernet.h>
#import "kashaheaders.h"
@interface InjectionEngine : NSObject {
@public
    NSMutableArray * buffer;
    NSConditionLock * bufferLock;

    libnet_t * injector;
    libnet_t * injector1;

    //for libnet error messages
    char errbuf[LIBNET_ERRBUF_SIZE];
    char errbuf1[LIBNET_ERRBUF_SIZE];

    //needed to recover data structures for injection
    struct ether_header * eh;
}
-(id)init;
-(id)initWithBuffer:(NSMutableArray *)bufferInput
   andInjectionLock:(NSConditionLock *)lockInput;
-(void)start;
-(void)run;
-(void)addNode:(BufferNode *)node;
-(BOOL)inject:(BufferNode *)node;
-(void)setBuffer:(NSMutableArray *)buff;
@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX GG. OBJ-C INJECTIONENGINE.M

```
//
//  InjectionEngine.m
//  tranquility
//
//  Created by John Judd on Thu May 01 2003.
//
//

#import "InjectionEngine.h"

@implementation InjectionEngine
-(id)init{
    injector = libnet_init(LIBNET_LINK,"en1",errbuf);
    injector1 = libnet_init(LIBNET_LINK,"en2",errbuf1);
    if(injector == NULL){
        NSLog(@"error initializing libnet");
        printf(errbuf);
    }
    if(injector1 == NULL){
        NSLog(@"error initializing libnet");
        printf(errbuf);
    }
    return self;
}

-(id)initWithBuffer:(NSMutableArray *)bufferInput
   andInjectionLock:(NSConditionLock *)lockInput{
    [super init];
    buffer = bufferInput;
    bufferLock = lockInput;
    return [self init];
}

-(void)setBuffer:(NSMutableArray *)buff{
    buffer = buff;
}

-(void)addNode:(BufferNode *)nodeIn{
    [bufferLock lock];
    [buffer addObject:nodeIn];
    NSLog(@"inj buffer size: %d" ,[buffer count]);
    [bufferLock unlockWithCondition:HASDATA];
}
```

```
-(BOOL)inject:(BufferNode *)node{
    eh = (struct ether_header *)[node->data bytes];
    int i;
    //build the frame for injection using libnet
    i = libnet_build_ethernet(
        (u_char *)eh->ether_dhost, /*Destination MAC*/
        (u_char *)eh->ether_shost, /*Source MAC*/
        eh->ether_type,            /*Layer-2 Type*/
        (u_char *)([node->data bytes]+sizeof(struct ether_header)),/*Layer-3 Data pointer*/
        node->pktHdr.caplen-sizeof(struct ether_header)-4, /*size of data*/
        injector,                  /*Libnet descriptor*/
        0);                                /*optional ptag_t*/
    if(i< 1){
        printf(libnet_geterror(injector));
        return NO;
    }
    //inject the frame using libnet
    i = libnet_write(injector);
    if(i< 1){
        printf(libnet_geterror(injector));
        return NO;
    }
    //NSLog(@"inject: %d ",i);

    //clear libnet memory this is important or the new data
    //will be appended to the old data and result in
    //a frame too large to send
    libnet_clear_packet(injector);
    return YES;
}

-(BOOL)inject1:(BufferNode *)node{
    eh = (struct ether_header *)[node->data bytes];
    int i;
    //build the frame for injection using libnet
    i = libnet_build_ethernet(
        (u_char *)eh->ether_dhost, /*Destination MAC*/
        (u_char *)eh->ether_shost, /*Source MAC*/
        eh->ether_type,            /*Layer-2 Type*/
        (u_char *)([node->data bytes]+sizeof(struct ether_header)),/*Layer-3 Data pointer*/
        node->pktHdr.caplen-sizeof(struct ether_header)-4, /*size of data*/
        injector1,                 /*Libnet descriptor*/
        0);                                /*optional ptag_t*/
```

```
        if(i< 1){
            printf(libnet_geterror(injector1));
            return NO;
        }
        //inject the frame using libnet
        i = libnet_write(injector1);
        if(i< 1){
            printf(libnet_geterror(injector1));
            return NO;
        }
//
       // NSLog(@"inject1: %d ",i);
        //clear libnet memory this is important or the new data
        //will be appended to the old data and result in
        //a frame too large to send
        libnet_clear_packet(injector1);
        return YES;
}

-(void)start{
    [NSThread detachNewThreadSelector:@selector(run)
                             toTarget:self withObject:nil];
    NSLog(@"injection Engine started");
}

-(void)run{
    NSAutoreleasePool * localPool = [[NSAutoreleasePool alloc] init];
    double count = 0;
    BufferNode * node;

    while(1){
        //NSLog(@"running");
        [bufferLock lockWhenCondition:HASDATA];
        if(count == 0){
            count = [buffer count];
        }
        if(count > 0){
            node = [buffer objectAtIndex:0];
            if(node->interface == 0)
                [self inject:node];
            else
                [self inject1:node];
            [buffer removeObjectAtIndex:0];
            count--;
        }
        if(count == 0)
```

```
            [bufferLock unlockWithCondition:NODATA];
        else
            [bufferLock unlockWithCondition:HASDATA];


    }
    [localPool release];

}

@end
```

APPENDIX HH. OBJ-C CAPTUREENGINE.H

```
//
//  CaptureEngine.h
//  tranquility
//
//  Created by John Judd
//
//

#import <Cocoa/Cocoa.h>

//#import <Foundation/Foundation.h>
#import "BufferNode.h"
//for pcap capture library
#import <pcap.h>
//for HASDATA and NODATA
#import "kashaHeaders.h"
@interface CaptureEngine : NSObject
{
@public
    //buffer for captured packets along with lock
    NSMutableArray * buffer;
    NSConditionLock * captureBufferLock;

    //error buffer for pcap
    char errbuf[PCAP_ERRBUF_SIZE];

    //pcap descriptor
    pcap_t* descr;

    //name of interface to sniff on
    char *dev;

    //struct for use with pcap_pkthdr info
    struct pcap_pkthdr pkthdr;

    //boolean for starting and stopping capture
    BOOL capture;

    //total number of packets captured
    unsigned long numberPacketsCaptured;

    //number of packets in buffer
    unsigned long currentBufferSize;
```

```
}
-(id)init;
-(id)initWithBuffer:(NSMutableArray *)packetBuffer
          andLock:(NSConditionLock *)lock;
-(void)testFunc;
-(void)startCapture;
-(void)stopCapture;
-(void)start;
-(void)dealloc;
-(unsigned long)numberOfPacketsCaptured;
-(void)numberOfPacketsCaptured:(unsigned long)value;
-(unsigned long)currentBufferSize;
@end
```

APPENDIX II. OBJ-C CAPTUREENGINE.M

```
//
//  CaptureEngine.m
//  tranquility
//
//  Created by John Judd
//
//

#import "CaptureEngine.h"
#import <Foundation/Foundation.h>
#import "BufferNode.h"

/*
Callback function used for packet capture
*/
void pcapCallback(u_char *user,struct pcap_pkthdr *pph,const u_char *pdata){
   CaptureEngine * ce = (CaptureEngine*)user;
   if(ce->capture == NO)return;
   BufferNode * node = [[BufferNode alloc] initWithPacket:pdata andPktHdr:pph];
   //start mutex region
   [ce->captureBufferLock lock];
   [ce->buffer addObject:node];
   //NSLog(@"%d", [ce->buffer count]);
   [ce->captureBufferLock unlockWithCondition:HASDATA];
   ce->numberPacketsCaptured++;
   //end mutex region

}

@implementation CaptureEngine

-(void)testFunc{
   NSLog(@"testfunc called");

}

-(void)dealloc{
   [super dealloc];
}

-(id)init{
   //device to capture traffic on
   dev = pcap_lookupdev(errbuf);
```

```
    //initialize the pcap descriptor
    descr = pcap_open_live(dev,BUFSIZ, -1,10,errbuf);

    //ensure pcap initialized correctly
    if(descr == NULL){
        NSLog(@"pcap initialization Error");
        NSString * string =[[NSString alloc]
            initWithCString:pcap_geterr(descr)];
        NSLog(string);
    }

    capture = NO;

    numberPacketsCaptured = 0;
    currentBufferSize = 0;
    return self;
}

-(id)initWithBuffer:(NSMutableArray *)packetBuffer
          andLock:(NSConditionLock *) lock{
    captureBufferLock = lock;
    buffer = packetBuffer;
    [packetBuffer retain];
    return [self init];
}

-(void)start{
    [NSThread detachNewThreadSelector:@selector(run)
                             toTarget:self
                           withObject:nil];
}

-(void)run{
    NSAutoreleasePool * localPool = [[NSAutoreleasePool alloc] init];
    if([NSThread isMultiThreaded])
        NSLog(@"entering Multithreaded Mode");
    NSLog(@"capture thread running");
    while(1){
        pcap_loop(descr, -1, (pcap_handler)pcapCallback, (u_char*)self);
    }
    [localPool release];

}
-(void)startCapture{
    capture = YES;
```

```
}

-(void)stopCapture{
    capture = NO;
}

-(unsigned long)numberOfPacketsCaptured{
    return numberPacketsCaptured;
}

-(void)numberOfPacketsCaptured:(unsigned long)value{
    numberPacketsCaptured = value;
}

-(unsigned long)currentBufferSize{
    [captureBufferLock lock];
    currentBufferSize = [buffer count];
    [captureBufferLock unlock];
    return currentBufferSize;
}
@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX JJ. OBJ-C DISPATCHER.H

```
//
//  Dispatcher.h
//  tranquility
//
//  Created by John Judd on Wed Apr 30 2003.
//

#import <Foundation/Foundation.h>
#import "SensorBase.h"
#import "L4Isolator.h"

//for HASDATA and NODATA
#import "kashaheaders.h"
@interface Dispatcher : NSObject {
    //buffer to read buffer nodes from
    NSMutableArray * buffer;

    //lock for reading buffer
    NSConditionLock * bufferLock;

    //array for storing sensors
    NSMutableArray * sensorArray;

    //lock used internally when adding sensors
    NSLock * sensorArrayLock;

    //number of packets processed
    unsigned long numberOfPacketsProcessed;
   }
-(id)init;
-(id)initWithBuffer:(NSMutableArray *)packetBuffer
          andLock:(NSConditionLock*) lock;
-(void)dealloc;
-(void)addSensor:(id)sensor;
-(void)dispatcherRun;
-(void)start;
-(unsigned long)numberOfPacketsProcessed;
@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX KK. OBJ-C DISPATCHER.M

```
//
//  Dipatcher.m
//  tranquility
//
//  Created by John Judd on Wed Apr 30 2003.
//

#import "Dispatcher.h"


@implementation Dispatcher
-(id)init{
   sensorArray = [[NSMutableArray alloc] init];
   sensorArrayLock = [[NSLock alloc]init];
   numberOfPacketsProcessed = 0;
   return self;
}
-(id)initWithBuffer:(NSMutableArray *)packetBuffer
         andLock:(NSConditionLock *)lock{
   buffer = packetBuffer;
   bufferLock = lock;
   return [self init];
}

-(void)dealloc{
   [buffer release];
   [sensorArray release];
}

-(void)addSensor:(id)sensor{
   [sensorArray addObject:sensor];
   [sensor start];
}

-(void)start{
   [NSThread detachNewThreadSelector:@selector(dispatcherRun)
                     toTarget:self withObject:nil];
}

-(void)dispatcherRun {
   NSAutoreleasePool * localPool = [[NSAutoreleasePool alloc] init];
   int i;
   int bufferCount = 1;
```

```
L4Isolator * sensor;
BufferNode * node;
//take control of sensors
for(i = 0 ; i < [sensorArray count] ; i++){
   sensor = [sensorArray objectAtIndex:i];
   [sensor->lock lockWhenCondition:OLDDATA];
}

while(1){
   [bufferLock lockWhenCondition:HASDATA];

   //get first packet in buffer
   node = [buffer objectAtIndex:0];
   [node retain];
   [buffer removeObjectAtIndex:0];
   bufferCount--;
   numberOfPacketsProcessed++;
   if(bufferCount <= 0)
      bufferCount = [buffer count];
   if(bufferCount == 0)
      [bufferLock unlockWithCondition:NODATA];
   else
      [bufferLock unlock];

   /*This lock puts a control around the march of the sensors
      a new sensor can be added so long as this lock is unlocked*/
   [sensorArrayLock lock];

   //update and signal sensors
   for(i = 0 ; i < [sensorArray count] ; i++){
      sensor = [sensorArray objectAtIndex:i];
      [sensor newNode:node];
      [sensor->lock unlockWithCondition:NEWDATA];
   }
   /*see the following for information on how to lock the threads together
      http://cocoadevcentral.com/articles/000061.php
   */
   for(i = 0 ; i < [sensorArray count] ; i++){
      sensor = [sensorArray objectAtIndex:i];
      [sensor->lock lockWhenCondition:OLDDATA];
   }
   // process any information from sensors

   [sensorArrayLock unlock];
   [node release];
```

```
    }//end while
    [localPool release];
}

-(unsigned long)numberOfPacketsProcessed{
    return numberOfPacketsProcessed;
}

@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX LL. OBJ-C L4ISOLATOR.H

```
//
//  L4Isolator.h
//  tranquility
//
//  Created by John Judd on Sat May 03 2003.
//

#import <Foundation/Foundation.h>
#import "BufferNode.h"
#import "IsolatorNode.h"
#import "kashaheaders.h"
#import <net/ethernet.h>
#import <libnet.h>

@interface L4Isolator : NSObject {
    //injectionBufer
    NSMutableArray * injectionBuffer;
    NSConditionLock * injectionBufferLock;

    //layer-3 type
    u_short layer3Type;

    //layer-4 type
    u_int8_t layer4Type;

    //data overlays for packet
    struct ether_header * eh;
    struct my_ip * ip;

    //dictionary for all connections
    NSMutableDictionary * connections;

    //configuration for isolator nodes
    unsigned long numberOfBins;
    unsigned long binInterval;

    //Variables for rate analysis
    int currentCount;
    int highThreshold;
    int lowThreshold;
    int fastInterface;
    int slowInterface;
    NSMutableData * fastMAC;
```

```
    NSMutableData * slowMAC;
    unsigned int port;

    @public
        BufferNode * node;
    NSConditionLock * lock;
}
-(id)init;

-(id)initWithL4Type:(u_int8_t)l4Type
 andInjectionBuffer:(NSMutableArray *)array
      andBufferLock:(NSConditionLock *)bufferLock;

-(id)initWithL4Type:(u_int8_t)l4Type
 andInjectionBuffer:(NSMutableArray *)array
      andBufferLock:(NSConditionLock *)bufferLock
         andFastDest:(NSString *)fast
        andSlowDest:(NSString *)slow
         andBinSize:(unsigned long)binSizeIn
    andNumberOfBins:(unsigned long)numberOfBinsIn
   andFastThreshold:(int)thresholdHigh
   andSlowThreshold:(int)thresholdLow
            andPort:(unsigned int)portIn;

-(void)analyzePacket:(BufferNode *)node;

-(void)newPacket;

-(void)start;

-(void)analyzeUDP;

-(void)analyzeTCP;

-(void)newNode:(BufferNode *)newNode;

-(void)route;
@end
```

APPENDIX MM. OBJ-C L4ISOLATOR.M

```
//
//L4Isolator.m
//  tranquility
//
//  Created by John Judd on Sat May 03 2003.
//

#import "L4Isolator.h"


@implementation L4Isolator
-(id)init{
    lock = [[NSConditionLock alloc] initWithCondition:OLDDATA];
    connections = [[NSMutableDictionary alloc]init];
    numberOfBins = 10;
    binInterval = 10;
    highThreshold = 0;
    lowThreshold = 50;
    fastInterface = 0;
    slowInterface = 1;
    fastMAC = [[NSMutableData alloc]init];
    slowMAC = [[NSMutableData alloc]init];
    port = 0;
    return self;
}
-(id)initWithL4Type:(u_int8_t)l4Type
 andInjectionBuffer:(NSMutableArray *)array
     andBufferLock:(NSConditionLock *)bufferLock{
    injectionBuffer = array;
    injectionBufferLock = bufferLock;
    layer4Type = l4Type;

    return [self init];
}

-(id)initWithL4Type:(u_int8_t)l4Type
 andInjectionBuffer:(NSMutableArray *)array
      andBufferLock:(NSConditionLock *)bufferLock
        andFastDest:(NSString *)fast
        andSlowDest:(NSString *)slow
         andBinSize:(unsigned long)binSizeIn
    andNumberOfBins:(unsigned long)numberOfBinsIn
   andFastThreshold:(int)thresholdHigh
```

```
  andSlowThreshold:(int)thresholdLow
          andPort:(unsigned int)portIn{

    //[super init];
    lock = [[NSConditionLock alloc] initWithCondition:OLDDATA];
    connections = [[NSMutableDictionary alloc]init];
    layer4Type = l4Type;
    injectionBuffer = array;
    injectionBufferLock = bufferLock;
    unsigned int fast_length = [fast length];
    unsigned int slow_length = [slow length];
    unsigned int * fastLength = &fast_length;
    unsigned int * slowLength = &slow_length;
    fastMAC = [[NSMutableData alloc]initWithBytes:
        libnet_hex_aton((u_char *)[fast cString],fastLength) length:*fastLength];
    slowMAC = [[NSMutableData alloc]initWithBytes:
        libnet_hex_aton((u_char *)[slow cString],slowLength) length:*slowLength];
    binInterval = binSizeIn;
    numberOfBins = numberOfBinsIn;
    highThreshold = thresholdHigh;
    lowThreshold = thresholdLow;
    port = portIn;
    return self;
}

-(void)analyzePacket:(BufferNode *)node{

}
-(void)newNode:(BufferNode *)newNode{
//    [newNode retain];
//    [node release];
    node = newNode;
}

-(void)newPacket{
//when this is called node holds a new node to be analyzed

    //check for Layer-3 type
    eh = (struct ether_header *)[node->data bytes];
    if(eh->ether_type != ETHERTYPE_IP)
        return; //we only work with IP for now sorry :-(

    //check for Layer-4 type
    ip = (struct my_ip *)([node->data bytes] + sizeof(struct ether_header));
    if(ip->ip_p != layer4Type)
        return;
```

```
    if(layer4Type == UDP)
        [self analyzeUDP];
    else
        [self analyzeTCP];
}

-(void)run{
    NSAutoreleasePool * localPool = [[NSAutoreleasePool alloc] init];
    NSLog(@"L4 Sensor started %d", port);
    while (1){
        [lock lockWhenCondition:NEWDATA];
        [self newPacket];
        [lock unlockWithCondition:OLDDATA];
    }
    [localPool release];

}

-(void)start{
    [NSThread detachNewThreadSelector:@selector(run)
                             toTarget:self withObject:nil];
}

-(void)analyzeUDP{
    //if()
}

-(void)analyzeTCP{
    BOOL correctDirection;
    struct my_tcp * tcp = (struct my_tcp *)([node->data bytes] +
                                    sizeof(struct ether_header)+
                                    ((ip->ip_vhl)&0x0f)*4);

   if(tcp->tcp_src_port != port &&
      tcp->tcp_dest_port!= port &&
      port != 0){ //zero can be used to look at all tcp traffic
        return;
   }
    //determine direction of connection
    if(tcp->tcp_src_port < 1024)
        correctDirection = YES;
    else
        correctDirection = NO;

    //make a string of source + destination
    NSString * source = [[NSString alloc]initWithCString:
```

```
        libnet_addr2name4(ip->ip_src.s_addr,LIBNET_DONT_RESOLVE)];

    NSString * dest = [[NSString alloc]initWithCString:
        libnet_addr2name4(ip->ip_dest.s_addr,LIBNET_DONT_RESOLVE)];

    NSString * key;
    if(correctDirection == YES){
        source = [source stringByAppendingString:@"->"];
        key = [source stringByAppendingString:dest];
        [source release];
    }
    else{
        dest = [dest stringByAppendingString:@"->"];
        key = [dest stringByAppendingString:source];
        [dest release];
    }

    //search dictionary for node
    IsolatorNode * isolatorNode = [connections objectForKey:key];

    //if not in dictionary make a temp dictionary and add to primary dictionary
    if(isolatorNode == nil){
        NSLog(@"new node");
        isolatorNode = [[IsolatorNode alloc] initWithSource:ip->ip_src.s_addr
                                        andDestination:ip->ip_dest.s_addr
                                       andNumberOfBins:numberOfBins
                                        andBinInterval:binInterval];
        [isolatorNode touch:node->arrivalTime];
        NSDictionary * dict =[NSDictionary dictionaryWithObject:isolatorNode
                                forKey:key];
        [connections addEntriesFromDictionary:dict];
        currentCount = 0;
    }
    else{
    //if in dictionary touch node.
        [isolatorNode touch:node->arrivalTime];
        currentCount = isolatorNode->currentCount;
        if(isolatorNode->lastBin > currentCount)
            currentCount = isolatorNode->lastBin;
        //NSLog(@"%s count: %d",[key cString],isolatorNode->currentCount);
    }
    [self route];

}

-(void)route{
```

```
    BufferNode * newNode = [[BufferNode alloc] initWithBufferNodePtr:node];
    [injectionBufferLock lock];
    eh = (struct ether_header *)[newNode->data bytes];
    if(currentCount > lowThreshold){
        newNode->interface = fastInterface;
        memcpy(&eh->ether_dhost, [fastMAC bytes], [fastMAC length]);
    }
    else{
        newNode->interface = fastInterface;
        memcpy(eh->ether_dhost, [slowMAC bytes], [slowMAC length]);

    }
    [injectionBuffer addObject:newNode];
    [injectionBufferLock unlockWithCondition:HASDATA];
}
@end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX NN. OBJ-C ISOLATORNODE.H

```
//
//  IsolatorNode.h
//  tranquility
//
//  Created by John Judd on Sat May 03 2003.
//

#import <Foundation/Foundation.h>


@interface IsolatorNode : NSObject {
    /*
     Source and Destination address as a 32 bit
     representation of a dotted decimal may be
     a class of network or an individual IP
     by changing the subnet mask in the isolator
     */
    unsigned long src;  //32 bit source address
    unsigned long dest; //32 bit destination address

    /*number of bins to use*/
    unsigned long numBins; //32 bit number of bins
    NSMutableArray * timeBinArray;

    /*TIme interval for each bin*/
    unsigned long timeBinInterval;

    /*current value*/
    unsigned long currentBin;

    /*start time of current bin*/
    time_t timeStart;

    /*end time of current bin*/
    time_t timeEnd;

    /*number of bins remaining to be written to file*/
    unsigned long listSize;

    /*Number of bins processed*/
    unsigned long numBinsProcessed;
@public
```

```
    /*count for current bin*/
    unsigned long currentCount;

    /*List of bins*/
    NSMutableArray * binArray;

    /*value of last non zero bin*/
    long lastBin;

}
-(id)init;
-(id)initWithSource:(unsigned long)source
     andDestination:(unsigned long)destination
    andNumberOfBins:(unsigned long)numberOfBins
     andBinInterval:(unsigned long)interval;
-(void)touch:(time_t)timestamp;
@end
```

APPENDIX OO. OBJ-C ISOLATORNODE.M

```
//
//  IsolatorNode.m
//  tranquility
//
//  Created by John Judd on Sat May 03 2003.
//

#import "IsolatorNode.h"


@implementation IsolatorNode
-(id)init{
    if(src == Nil){
        src = 0;
        dest = 0;
        numBins = 10;
        timeBinInterval = 30;
    }
    timeBinArray =  [[NSMutableArray alloc]init];
    currentBin = 0;
    timeStart = NULL;
    timeEnd = timeStart + timeBinInterval;
    numBinsProcessed = 0;
    binArray = [[NSMutableArray alloc]init];
    currentCount = 0;
    return self;
}
-(id)initWithSource:(unsigned long)source
     andDestination:(unsigned long)destination
    andNumberOfBins:(unsigned long)numberOfBins
     andBinInterval:(unsigned long)interval{
    src = source;
    dest = destination;
    numBins = numberOfBins;
    timeBinInterval = interval;
    return [self init];
}

-(void)touch:(time_t) timestamp{
    if(timeStart == NULL){
        timeStart = timestamp;
        lastBin = 0;
    }
```

```
    if(difftime(timestamp, timeStart) <= timeBinInterval){
        currentCount++;
        return;
    }
    else{
        while(difftime(timestamp, timeStart)>timeBinInterval){
            [binArray addObject:[[NSNumber alloc] initWithUnsignedLong:currentCount]];
            numBinsProcessed++;
            if(currentCount != 0){
                lastBin = currentCount;
            }
            currentCount = 0;
            currentBin = (numBinsProcessed)%numBins;
            timeStart= timeStart + timeBinInterval;
        }
        currentCount++;
    }
    return;

}
@end
```

APPENDIX PP. CAPTURE RESULTS

Packets Injected:	1237119				
Bytes Injected:	321607620				
Injection Speed Mb/s	Packets Captured	Packets Missed	%Missed	Packets/sec	Seconds
Kasha (C++)					
5.01	711396	525723	42.49575021	2524	490.2
10.03	711563	525556	42.4822511	5058	244.6
15.05	711813	525306	42.46204286	7588	163
20.16	545108	692011	55.93730272	10163	121.7
25.1	442326	794793	64.24547679	12653	97.8
30.12	381819	855300	69.13643716	5187	81.46
35.1	327177	909942	73.55331217	17695	69.91
40.4	281530	955589	77.24309464	20365	60.75
45.29	248384	988735	79.92238418	22831	54.19
50.24	228118	1009001	81.56054511	25328	48.84
55.48	213298	1023821	82.75848968	27967	44.23
61.05	197681	1039438	84.02085814	30790	40.18
65.39	170393	1066726	86.22662816	32970	37.52
71.15	162715	1074404	86.84726368	35875	34.48
75.4	152561	1084558	87.66804164	38018	32.54

Packets Injected:	1237119				
Bytes Injected:	321607620				
Injection Speed Mb/s	Packets Captured	Packets Missed	%Missed	Packets/sec	Seconds
Tranquility (Obj-C)					
5.01	1235113	2006	0.162150933	2524	490.2
10.03	1230763	6356	0.513774342	5058	244.6
15.05	1203644	33475	2.705883589	7588	163
20.16	1184538	52581	4.250278267	10163	121.7
25.1	1142743	94376	7.628692147	12653	97.8
30.12	1087780	149339	12.07151454	15187	81.46
35.1	1019600	217519	17.58270627	17695	69.91
40.39	972276	264843	21.40804563	20365	60.75
45.28	853840	383279	30.98157898	22831	54.19
50.24	823341	413778	33.44690365	25328	48.84
55.47	761430	475689	38.45135351	27967	44.23
61.1	698723	538396	43.5201464	30790	40.18
65.4	635225	601894	48.65287818	32970	37.52
71.154	608454	628665	50.81685755	35875	34.48
75.4	590268	646851	52.2868859	38018	32.54

Packets Injected:	1237119				
Bytes Injected:	321607620				
Injection Speed Mb/s	Packets Captured	Packets Missed	%Missed	Packets/sec	Seconds
Snort 1.9 -I					
5	1237129	-10	-0.00080833	2524	490.2
10.03	1236919	200	0.016166594	5058	244.6
15.05	1229982	7137	0.57690489	7588	163
20.16	1205649	31470	2.543813489	10163	121.7
25.1	1168826	68293	5.520325854	12653	97.8
30.12	1120709	116410	9.409765754	15187	81.46
35.1	1079037	158082	12.77823718	17695	69.91
40.39	1032616	204503	16.53058437	20365	60.75
45.28	994764	242355	19.59027385	22831	54.19
50.25	954577	282542	22.83870832	25328	48.84
55.48	923868	313251	25.32100792	27967	44.23
61.1	900041	337078	27.24701504	30790	40.18
65.39	872856	364263	29.44445926	32970	37.52
71.16	851271	385848	31.18923887	35875	34.48
75.4	828374	408745	33.04007133	38018	32.54

Packets Injected:	1237119				
Bytes Injected:	321607620				
Injection Speed Mb/s	Packets Captured	Packets Missed	%Missed	Packets/sec	Seconds
Snort 1.9 No Console Output -ci					
5	1237123	-4	-0.000323332	2524	490.2
10.03	1236802	317	0.025624051	5058	244.6
15.05	1227793	9326	0.753848256	7588	163
20.16	1203300	33819	2.73369013	10163	121.7
25.1	1160748	76371	6.173294566	12653	97.8
30.12	1110990	126129	10.19538137	15187	81.46
35.1	1066682	170437	13.77692849	17695	69.91
40.39	1024366	212753	17.19745635	20365	60.75
45.28	988199	248920	20.1294229	22831	54.19
50.25	953397	283722	22.93409122	25328	48.84
55.48	918929	318190	25.72024195	27967	44.23
61.1	886486	350633	28.34270592	30790	40.18
65.39	863850	373269	30.17244097	32970	37.52
71.16	847200	389919	31.51830988	35875	34.48
75.4	823895	413224	33.40212219	38018	32.54

Packets Injected:	1237119				
Bytes Injected:	321607620				
Injection Speed Mb/s	Packets Captured	Packets Missed	%Missed	Packets/sec	Seconds
Snort 1.9 tcpdump logging -bci					
5	1237127	-8	-0.000646664	2524	490.2
10.03	1236918	201	0.016247426	5058	244.6
15.05	1227600	9519	0.769449018	7588	163
20.16	1203076	34043	2.751796715	10163	121.7
25.1	1164426	72693	5.875990911	12653	97.8
30.12	1116998	120121	9.709736897	15187	81.46
35.1	1069109	168010	13.58074688	17695	69.91
40.39	1025093	212026	17.13869078	20365	60.75
45.28	990391	246728	19.94375642	22831	54.19
50.25	953658	283461	22.91299382	25328	48.84
55.48	925929	311190	25.15441118	27967	44.23
61.1	891623	345496	27.92746696	30790	40.18
65.39	873789	363330	29.36904211	32970	37.52
71.16	845407	391712	31.66324339	35875	34.48
75.4	841508	395611	31.97841113	38018	32.54

Packets Injected:	1237119				
Bytes Injected:	321607620				
Injection Speed Mb/s	Packets Captured	Packets Missed	%Missed	Packets/sec	Seconds
Snort 2.0 -ci					
5	1237123	-4	-0.000323332	2524	490.2
10.03	1236824	295	0.023845725	5058	244.6
15.05	1205291	31828	2.572751692	7588	163
20.16	1203528	33591	2.715260213	10163	121.7
25.1	1165556	71563	5.784649658	12653	97.8
30.12	1123894	113225	9.152312752	15187	81.46
35.1	1070066	167053	13.50338973	17695	69.91
40.39	1030748	206371	16.68158035	20365	60.75
45.28	991255	245864	19.87391674	22831	54.19
50.25	958324	278795	22.53582719	25328	48.84
55.48	923057	314062	25.38656346	27967	44.23
61.1	892361	344758	27.86781223	30790	40.18
65.39	868645	368474	29.78484689	32970	37.52
71.16	847626	389493	31.48387504	35875	34.48
75.4	829966	407153	32.91138524	38018	32.54

All capture results were taken on the following machine:
Macintosh Dual 1.42 Ghz G4
2Gb RAM
OS X 10.2.6

Traffic was generated using:
tcpreplay 1.0.1
Dell Dimension L667r 667Mhz
512Mb RAM
Red Hat 7.3

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Bret Michael
   Naval Postgraduate School
   Monterey, California

4. Dr. John McEachen
   Naval Postgraduate School
   Monterey, California

5. Dr. Wen Su
   Naval Postgraduate School
   Monterey, California

6. Dr. George Dinolt
   Naval Postgraduate School
   Monterey, California

7. LCDR Chris Eagle
   Naval Postgraduate School
   Monterey, California

THIS PAGE INTENTIONALLY LEFT BLANK