Effective Programmatic Analysis of Network Flow Data for Security and Visualization using Higher-order Statistics and Domain Specific Embedded Languages

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Thomas A. Conley

June 2012

© 2012 Thomas A. Conley. All Rights Reserved.
This thesis titled
Effective Programmatic Analysis of Network Flow Data for Security and Visualization
using Higher-order Statistics and Domain Specific Embedded Languages

by
THOMAS A. CONLEY

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

________________________________________
Shawn Ostermann
Associate Professor of Electrical Engineering and Computer Science

________________________________________
Dennis Irwin
Dean of the Russ College of Engineering
ABSTRACT

CONLEY, THOMAS A., M.S., June 2012, Computer Science

Effective Programmatic Analysis of Network Flow Data for Security and Visualization using Higher-order Statistics and Domain Specific Embedded Languages (111 pp.)

Director of Thesis: Shawn Ostermann

The widespread availability of information on networks today, coupled with the potential for exploitation by malicious software, demands constant vigilance by network engineers responsible for information security. Even a moderately sized computer network produces a flow of information that is impossible for a human to watch carefully and understand without tools capable of automatic summation and analysis.

This thesis presents research and engineering that demonstrates the usefulness of network traffic data and presents effective statistical methods and practical mechanisms for analyzing massive amounts of this information for intrusion detection, network forensics, problem alerting and systems monitoring.

We explore how a simple set of network traffic features can be analyzed and used for characterizing behavior on the network. We suggest that statistical measurements, entropy and other higher-order calculations are effective in determining network status or for detecting anomalies. Communication patterns in NetFlow data are summarized for further automatic analysis or for visual interpretation by information security analysts. We examine the potential for identifying overlying networks, such as botnet command and control systems, within a larger complex network of communication. We suggest ways of automating or assisting the manual processes for traffic analysis currently in place at Ohio University through the development of simple tools.

Approved: 

Shawn Ostermann

Associate Professor of Electrical Engineering and Computer Science
# Table of Contents

Abstract .................................................................................................................. 3

List of Tables ......................................................................................................... 7

List of Figures ........................................................................................................ 8

1 Introduction ....................................................................................................... 9
   1.1 The Importance of Traffic Data Analysis .................................................... 9
      1.1.1 Sources of Network Traffic Data ....................................................... 11
      1.1.2 Behavior- Versus Signature- Based Detection ..................................... 12
      1.1.3 Netflow Data Collection at Ohio University ..................................... 13
   1.2 The Focus of this Research ...................................................................... 13
      1.2.1 Netflow Data Versus Raw Packet Data .......................................... 13
      1.2.2 Netflow Analysis Problems ............................................................. 15
      1.2.3 Selected Network Features ............................................................... 16

2 Research Methods ............................................................................................. 20
   2.1 Characterizing and Aggregating Network Traffic Data ............................ 20
      2.1.1 Choosing an Network Feature Set ................................................... 20
      2.1.2 Problems with NetFlow Record Duration ....................................... 20
      2.1.3 Determining Normality and Correlation ........................................ 21
      2.1.4 Aggregation Techniques ................................................................. 21
   2.2 Detecting Network Traffic Anomalies ....................................................... 23
      2.2.1 Measures of Entropy ................................................................... 23
      2.2.2 Anomalies as a Deviation from Normal Traffic ............................. 23
   2.3 Visualizing Network Security Data ............................................................ 24
      2.3.1 Higher-Order Statistics .................................................................. 25
      2.3.2 Multivariate Distance .................................................................... 27
   2.4 Signal Filtering and Pattern Recognition ................................................ 30
      2.4.1 Moving Average Frequency ............................................................. 30
      2.4.2 Moving Standard Deviation ............................................................. 31
      2.4.3 Digital Signal Processing Techniques ............................................. 32
      2.4.4 Calculating Entropy On The Fly ...................................................... 32
      2.4.5 Binning Strategies ......................................................................... 34
      2.4.6 Clustering Techniques ................................................................. 36
   2.5 Entropy and Multivariate Statistics ........................................................... 36
   2.6 Anomalies as Deviation from Normal ....................................................... 37
3 Implementation ....................................................... 39
  3.1 History of Network Traffic Analysis at Ohio University ...... 39
    3.1.1 Noise Floor Alerting ...................................... 40
    3.1.2 Ad Hoc Filtering ......................................... 40
    3.1.3 BinMap and Research Tools ............................... 41
  3.2 Design Considerations ......................................... 41
    3.2.1 ACL Filtering .............................................. 42
    3.2.2 The problem with ACL Filtering .......................... 42
  3.3 Development and Production Environments ...................... 43
    3.3.1 Perl ....................................................... 43
    3.3.2 MatLab ..................................................... 44
    3.3.3 Graphics Packages ....................................... 44
    3.3.4 C++ ....................................................... 44
  3.4 The Applications ............................................... 46
    3.4.1 The flow-macros script .................................... 47
    3.4.2 The flow filtering application ........................... 50
    3.4.3 The flow-map application ................................ 55
    3.4.4 The flow-top application ................................ 58
    3.4.5 The flow-dot utility ...................................... 60
    3.4.6 The flow-nessus application .............................. 61
  4 Results and Observations ......................................... 63
    4.1 NetFlow Fields as Indicators of Network State .............. 63
    4.2 Normality and Correlation of NetFlow Data .................. 68
    4.3 Identifying Anomalies Through State Change ................ 71
    4.4 Ad Hoc ACL Filtering using Boolean Logic .................. 76
    4.5 Identifying Overlying Networks of Communication .......... 77
    4.6 Analysis Language for Alerting and Automation .............. 83
  5 Conclusions ....................................................... 87
    5.1 Netflow Data is Not Sufficient for Attribution of Traffic .. 87
    5.2 NetFlow Data Can be Analyzed in Near-Real Time ............ 88
    5.3 IP Packet Capture is Better for Precise Timing Analysis .... 89
    5.4 Average Port May be Similar to Entropy in Usefulness ....... 90
    5.5 A Priori Knowledge Helps to Improve Network Analysis Tools . 90
    5.6 Methods Described in this Research are Useful for Ad Hoc Data Analysis . 91
  6 Future Work ....................................................... 93
    6.1 Language Improvements ....................................... 93
    6.2 Alerts Logging to the SEIM, Nessus Scanner, or Footprints .. 93
    6.3 The use of Filter Watchlists can be expanded ................ 93
Appendix A: A Smart Pointer Template Class for C++ . . . . . . . . . . . . . . . . . . . . . . . . . . 100
Appendix B: Embedded Language Grammar for ACL Creation . . . . . . . . . . . . . . . . . . . . 106
Appendix C: Embedded Language Grammar for Flow Alerting . . . . . . . . . . . . . . . . . . . . 108
Appendix D: Source Listing for Flow-Dot Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
LIST OF TABLES

1.1 Typical Daily Network Flow Statistics ........................................ 14
1.2 Protocol Usage Statistics .......................................................... 18

4.1 Monotonic Ascension and Record Duration in NetFlow Data ............... 67
4.2 Selected Network Features by MSE Value .................................... 69
4.3 Entropy and Median Values for Spikes in Entropy Graph .................. 75
4.4 Sample of Alerts Firing During an Event of Interest ....................... 86
# List of Figures

1.1 Directed Graph of Residential Network Communications ........................................ 10
1.2 C++ Definition of a Netflow Tuple for Selected Network Features .......................... 17

2.1 Sample Parallel Axis Graph ...................................................................................... 25
2.2 Circular Graph of Port Communications ................................................................... 26
2.3 Visualization of Higher-order Statistics for Network Traffic .................................... 27
2.4 Calculation of Moving Standard Deviation ............................................................... 28
2.5 ADAP Filter Calculation in MatLab ........................................................................... 33
2.6 Multi-dimensional Data Strategy for Stochastic Analysis ......................................... 35

3.1 Overview of flow data processing. ............................................................................. 47
3.2 Flow Application Defined Macros ........................................................................... 48
3.3 Flow Application Help Screen .................................................................................. 49
3.4 Grammar Defining Boolean to ACL Conversion ....................................................... 51
3.5 Sample Cisco ACL File from Boolean Expression .................................................... 53
3.6 Sample Cisco ACL File from Expression Using Parentheses .................................... 54
3.7 Data Dimensions of Flow-Map .................................................................................. 55
3.8 Inheritance Diagram for Stat_Map Class .................................................................... 57
3.9 Help Screen for Flow-map or Flow-top Applications. ................................................ 58
3.10 Sample Flow-map Rules File Showing the use of Port and IP Macros. ....................... 59
3.11 Curses style console output of the flow-top application ........................................... 60
3.12 C++ Function to Open Embedded Pipe for Converting Flow Input. ......................... 62

4.1 Argus Start and End Times ......................................................................................... 65
4.2 Analysis of JFlow record times .................................................................................. 66
4.3 Graphical Comparison of Measured- vs. Digitally Processed-signals .......................... 70
4.4 Port Usage Across the Range of all Ports. ................................................................ 72
4.5 Snapshot of Internal Port NoiseFloor Scrolling Graph .............................................. 73
4.6 Snapshot of External Port NoiseFloor Scrolling Graph .............................................. 74
4.7 Snapshot of Shannon’s Entropy Graph ...................................................................... 76
4.8 Auto-generated ACL File for Flow Filtering ............................................................. 78
4.9 BotNet Patterns of Communication on the Residential Network .............................. 79
4.10 TOR Network Patterns of Communication .............................................................. 80
4.11 Resnet Internal Communication without Network Equipment .................................. 81
4.12 Directed Graph of Identified Clique .......................................................................... 82
4.13 Sample Alerting Rule Configuration File ................................................................. 84
1 INTRODUCTION

This thesis presents research and engineering that demonstrates the usefulness of network traffic data and presents effective statistical methods and practical mechanisms for analyzing massive amounts of this information for intrusion detection, network forensics, problem alerting and systems monitoring.

Every computer on the Internet is connected by many paths to every other computer on the Internet. This web of connectivity, based on IP addressing, port channels and protocols is known as the “Web”. Many other virtual networks are defined by the actual sharing of information over these physical connections. These overlying patterns include peer to peer (P2P) file-sharing networks, botnet command and control networks and IP-based telephone services like Skype. Network engineers and information security analysts are tasked with identifying and understanding these networks in order to protect network resources and enforce information security policies. In many cases, these networks attempt to avoid detection by obscuring their communications. Figure 1.1 shows a directed graph of all internal communications on the Ohio University residential housing network and illustrates some of these overlying patterns which are of interest to network engineers and information security analysts. This graph and other results will be discussed at length in Chapter 4.

1.1 The Importance of Traffic Data Analysis

The widespread availability of information on networks today, coupled with the unprecedented potential for exploitation by malicious software, demands constant vigilance by network engineers responsible for information security. Even a moderately sized computer network is capable of producing a flow of information that is impossible for a human to watch carefully and understand without tools capable of automatic summation and analysis. Network security professionals are faced with the daunting task
Figure 1.1: Directed Graph of Internal Traffic from Ohio University’s Residential Network showing Patterns of Overlying Networks

of analyzing network traffic in order to detect attacks from botnets, worm outbreaks, and reconnaissance scans before they become serious threats to the confidentiality, integrity and availability of the network.

The Information Security Office at Ohio University relies on NetFlow information for much of their daily analysis tasks. A systematic and documented research effort is needed to characterize network traffic and to automate the onerous manual processes currently in
place. The practicality of using existing NetFlow data sources was a deciding factor for this research as NetFlow data is already being collected at Ohio University for a variety of other purposes and establishing a raw packet capture on a large volume network would require much effort and expense.

1.1.1 Sources of Network Traffic Data

Information flowing across Internet Protocol (IP) networks is encapsulated in a fundamental unit of data called a packet. The composition of individual packets, and their transmission sequences, are governed by the various communication protocols employed on the network. This mixture of content and protocol is the foundation of a modern computer network. While it is often necessary to analyze the specific content and sequences of individual packets when debugging network problems, or when looking for specific well-known network anomalies, such in-depth analysis of network content is not the intention of this study. Instead, this research focuses on identifying and understanding the overlying patterns of communication in a computer network, regardless of the underlying protocol or packet content.

A fundamental unit of network traffic is called a “flow” or “NetFlow”. A flow is defined as a series of one- or two-way communications between a source and destination computer on a particular network channel. The two computers are identified by unique IP addresses and the channel consists of a unique combination of source and destination ports. Thus defined, a flow can be identified by a four-part key consisting of source IP, source port, destination IP and destination port. A single flow record contains aggregate statistics regarding the duration and volume of packets exchanged between the source and destination computers. The total communication between two computers may consist of many flows.
It is important to note that NetFlow data only contains statistical information about communications on the network. No actual content of the communication is retrieved and none is available for inspection. For instance, NetFlow data can identify to what servers a particular computer has connected and how much information was exchanged, but nothing can be said about any specific substance of the communication. It is possible to infer the nature of the communication, however, based on NetFlow features such as which ports are used. Email traffic, for instance, may be distinguished from web browsing traffic because of the port used. However, an inference based on port usage is never certain since ports can be used in non-standard ways.

1.1.2 Behavior- Versus Signature- Based Detection

Tools which use NetFlows cannot make any decisions based on the specific content of network traffic. Instead, there are many other tools which fill that niche. Network security devices such as firewalls, spam or virus filters, and Intrusion Detection Systems (IDS) make decisions based on a specific pattern or signature found in the content of packets. For example, when a new virus is discovered, a specific sequence of bytes from the virus payload is entered into a firewall rule as a signature for that malware. The firewall can then identify traffic from that virus. These signature-based detection schemes are very effective when dealing with established or well-known threats. Although some effort has been taken to provide flexible or changing signatures based on traffic content [36], signature based methods will fail to detect new attack patterns, known as “zero-day” exploits.

Some detection methods are based solely on statistical behavior calculated from short term traffic observations of NetFlow data [5]. Approaches such as this rely on finding malicious patterns of network behavior and do not suffer from the problem of changing payload signatures. Instead, an attacker may try to hide the malicious traffic by changing its timing or network behavior. Methods of obscuring malicious traffic include using
well-known “legitimate” ports, or flooding the network with seemingly legitimate traffic in an attempt to confuse the detectors.

1.1.3 Netflow Data Collection at Oho University

Netflow statistics are retrieved from network routers and switches that are capable of producing the aggregated records directly, or from software programs such as Argus which can access and aggregate the underlying packet traffic. Cisco netflows, Juniper JFlows and Argus log files are similar enough to be used interchangeably for traffic analysis [31, 38]. Indeed, any data source such as firewall logs which contains the unique four-part key can be converted for use in network flow analysis. Since its creation in 2002, the Information Security Office (ISO) at Ohio University has relied heavily on network traffic data from Argus collectors at the campus Internet border. The University now collects an average of 233 megabytes of compressed network traffic data per hour from the university’s core routers. Overall traffic statistics from a typical day is shown in Table 1.1.

1.2 The Focus of this Research

A systematic and documented research effort is needed to characterize and automate the onerous manual NetFlow analysis tasks currently undertaken by the Information Security Office at Ohio University.

1.2.1 Netflow Data Versus Raw Packet Data

This research has considered two basic options when deciding on a data source for network traffic analysis. Firstly, it is possible to analyze the raw packet data captured by a collector program such as Tcpdump using the Libpcap library and compile the aggregate flow statistics directly. This may provide better efficiency than existing aggregators since the data capture and analysis could focus specifically on the information of interest and
uninteresting traffic or data fields could be filtered out very early in the process. The raw packet data also has more information that could be used, such as the actual content of the communication. This approach however, would require redoing much of the functionality of existing NetFlow aggregators and would produce a less flexible and extensible system.

The second option provides ease of use and flexibility by using open-source or existing NetFlow aggregators to provide ready-made statistics. Although the provided statistics may contain more information than needed for a particular analysis, the efficiency of this method can be improved by filtering the data with separate executable
programs before final processing by the analysis program. This chain of programs uses standard input and output mechanisms that are common to many data analysis utilities. Using standard input and output means that the analysis engine is not limited to a particular style of NetFlow data, such as Argus or Cisco NetFlow, and it allows network engineers to easily integrate any new analysis tools with preexisting utilities. For example, some firewall logs may be easily converted for integration into NetFlow data without modifying the analysis programs.

The practicality of using existing NetFlow data sources was a deciding factor for this research. This data is already being collected at Ohio University for a variety of other purposes and establishing a raw packet capture on large volume network connections requires much effort and expense.

1.2.2 Netflow Analysis Problems

The time duration of the statistics for each NetFlow record is variable. Consequently, the end time of any particular record may be later than the starting time of the following record. In other words, the record timeframes overlap. In addition, the beginning times of the records are not monotonically ascending within the series of records. In short, NetFlow data is not strictly a time series. When compiling statistics over time from a continuous stream of data it is necessary to choose some time range or “bin” for storing the statistics.

Other researchers have faced the problem that extant NetFlow records do not fit nicely into time bins because of varying durations. Sen and Wang attacked this problem by fragmenting longer duration records and spreading their values over a range of bins [29]. The same approach was used in this research for the early analysis however, this approach makes processing in near real-time difficult since information from long-term records must be kept in memory until the data can be spread over the longer time period.
As this research focuses on near real-time analysis that is fathomable by a human, we ignore variations in the time series and record durations. Instead, we analyze records in their presented order regardless of the duration, and minimize the effect of temporal variations by using the start time of the flow, which has fewer exceptions to being monotonically ascending. Additionally, we select an analysis time window that is no less than the maximum duration of a NetFlow record as discussed in detail in Section 3.1.3.

NetFlow records contain measurements of multiple independent variables and so require either a multivariate analysis or a simple analysis of multivariate distance. Previous research has confronted this problem with a novel approach that looks for local maxima and minima in multivariate space using second- and fourth-order derivatives of Fisher Linear Discriminant (FLD) of total entropy as a measure of distance. This approach has been shown to be effective in identifying network anomalies in bursty nature of traffic data over small time periods without additional low-pass filtering [4].

1.2.3 Selected Network Features

A NetFlow collection device or Argus collection program may be configured to supply a variety of traffic features as described in RFC5237[2] and Argus documentation[3], however certain basic fields are available from any configuration. For purposes of this research and software implementation, we limit ourselves to this small set of easily measured features; thus we define a NetFlow object as an 8-tuple defined in C++ as shown in Figure 1.2. Note that the fields stime, proto, saddr, sport, sbytes, daddr, dport and dbytes are start time, source address, source port, source bytes, destination address, destination port and destination bytes respectively. These fields are chosen as the minimal set available from any collector source, and the names are chosen to correspond with names used by the Argus NetFlow collector and the flow-map application described in Chapter 3.
```
struct NetFlow {
    time_t stime; // start time
    protocol_t proto; // protocol
    address_t saddr; // source address
    port_t sport; // source port
    count_t sbytes; // source bytes
    address_t daddr; // destination address
    port_t dport; // destination port
    count_t dbytes; // destination bytes

    NetFlow();
    NetFlow(const std::string&, bool=false);
    bool read_argus(const std::string&);
    bool read_netflow(const std::string&);
};
typedef CSmartPointer<NetFlow> NetFlow_t;
```

Figure 1.2: C++ Definition of a Netflow Tuple for Selected Network Features

Accurate time measurements are crucial for forensic investigation and network troubleshooting where the specific time of an event must be determined and when one log source must be correlated with another. Flow records are time stamped with a start and end time covering the statistics which they contain. However, this may not be enough timing information to accurately describe the communication, particularly when the connection times are large. For instance, a record may have a time stamp indicating a connection time of 1 hour. The single field showing the number of bytes transferred gives no information about when the bytes were transmitted within that hour.

The Internet Protocol (IP) address field is used to locate a source or destination computer on an IP network but it should not be considered the “identity” of the computer. The true source or destination of a communication may change over time, or a single true source may be represented by multiple IP addresses. The true identity of a computer is held by an underlying network protocol; in most cases this is an Ethernet network which identifies a computer by a Media Access Control (MAC) address. The MAC address is not typically available in NetFlow records. The management of IP addresses and the true
identity of computers is in the domain of the network routers, switches, name servers (DNS) and dynamic host (DHCP) servers.

This research primarily deals with IP Version 4 (IPv4) addresses, as this is the basis for the Ohio University Network at this time. Many of the utilities developed as part of this research must read the addresses in the form of dotted decimal strings. The Perl language is very effective in dealing with strings such as these; however, when handling large amounts of data, or when great speed is needed, we rely on accessing the 32 bit number directly using C++. More discussion about the efficient processing of these 32 bit IPv4 addresses and the potential for handling 128 bit IPv6 numbers can be found in Chapter 3.

The protocol field contains the protocol number as defined in RFC5237[2]. Along with the port information the value of this field is used to infer the type of communication or service being employed and protocol tells us something about how the underlying connection is made. Table 1.2 shows typical usage of different protocols on Ohio University’s network by percentage.

Table 1.2: Protocol Usage Statistics on Ohio University Network on a Typical Day, 2012-01-16

<table>
<thead>
<tr>
<th>Number</th>
<th>Protocol</th>
<th>Description</th>
<th>Flows</th>
<th>Octets</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>TCP</td>
<td>Transmission Control</td>
<td>286,332,140</td>
<td>8,176,666,226,808</td>
<td>9,691,651,112</td>
</tr>
<tr>
<td>17</td>
<td>UDP</td>
<td>User Datagram</td>
<td>176,528,821</td>
<td>1,598,493,439,273</td>
<td>2,854,409,868</td>
</tr>
<tr>
<td>1</td>
<td>ICMP</td>
<td>Internet Control Mesg.</td>
<td>7,777,741</td>
<td>10,116,292,887</td>
<td>37,805,636</td>
</tr>
<tr>
<td>50</td>
<td>ESP</td>
<td>Encap. Security Payload</td>
<td>54,048</td>
<td>1,338,973,300</td>
<td>4,555,037</td>
</tr>
<tr>
<td>41</td>
<td>IPv6</td>
<td>IPv6 encapsulation</td>
<td>211,365</td>
<td>1,955,042,610</td>
<td>4,292,455</td>
</tr>
<tr>
<td>47</td>
<td>GRE</td>
<td>General Routing Encap.</td>
<td>3621</td>
<td>642,012,319</td>
<td>1,692,272</td>
</tr>
<tr>
<td>112</td>
<td>VRRP</td>
<td>Virtual Router Redun.</td>
<td>2789</td>
<td>7,268,200</td>
<td>181,705</td>
</tr>
<tr>
<td>80</td>
<td>ISO-IP</td>
<td>ISO Internet Protocol</td>
<td>3</td>
<td>1778</td>
<td>7</td>
</tr>
<tr>
<td>46</td>
<td>RSVP</td>
<td>Reservation Protocol</td>
<td>9</td>
<td>1044</td>
<td>9</td>
</tr>
<tr>
<td>169</td>
<td>unassigned</td>
<td></td>
<td>1</td>
<td>46</td>
<td>1</td>
</tr>
</tbody>
</table>
The bytecount field is sometimes referred to as “octets” and holds the number of bytes transferred during the flow record time period. This field is useful as an indication of volume of information being transferred, as such it is often used as the measure of a particular activity that is inferred from looking at the other fields. Argus NetFlow records include bytecounts for both source and destination since the record represents a bidirectional flow. Juniper netflows contain a single field.

This port field contains a channel identifying number as defined in RFC6335[12]. The port number along with the protocol type is an indicator of the service or channel that is being used for communication, analogous to a telephone number – but “port value” has no meaning as a measurement. The “frequency” of occurrence of a particular port however, is a useful measurement. These fields are often used to infer the nature of the communication represented by the record without having access to the actual content of the communication. This information is often taken at face value without any verification because the actual content of the underlying communication is not available in the NetFlow record. However, an inference based on port usage is never certain since ports can be used in non-standard ways. The use of port in filtering is discussed at length in Chapters 2 and 4.
2 Research Methods

2.1 Characterizing and Aggregating Network Traffic Data

Network traffic data is widely used by researchers in the field of security, forensics, signal processing, and pattern recognition. In particular, a research team in the Russ College of Engineering at Ohio University has focused recent research on stochastic analysis and pattern recognition in network flow data[4, 5]. This research has focused on treating network flow data as a time series. They support this emphasis by verifying the pair-wise correlation among network features before proceeding to filter the traffic using advanced statistical models [5, 27]. Much of the background research and engineering described in this chapter is in direct support of Russ College research and was published in IEEE Transactions on Information Forensics & Security. in June 2010.

2.1.1 Choosing an Network Feature Set

Although flow data is often used to draw conclusions regarding behavior on computer networks, the reasons for selecting a particular feature set and the statistical validity of using such data is seldom addressed in research. In research by Celenk et al.[5], we have shown this validity using both statistical and graphical analysis.

2.1.2 Problems with NetFlow Record Duration

Previous research has struggled with the problem that extant NetFlow records do not fit nicely into time bins because of varying durations. Sen and Wang attacked this problem by fragmenting longer duration records and spreading their values over a range of bins [29]. The same approach has been used in this research, however this approach can be difficult since information from long-term records must be kept in memory until the data can be spread over the longer time period. As this research focuses on near real-time analysis which is fathomable by a human being, we ignore variations in the time series
and record durations. Instead, we analyze records in their presented order regardless of the duration, and minimize the effect of temporal variations by using the start time of the flow, which has fewer exceptions to being monotonically ascending as shown in Table 4.1 and Figure 4.1.

2.1.3 Determining Normality and Correlation

Determining the normality of research data is particularly important when making decisions based on mean ($\mu$) and standard deviation ($\sigma$). If the data is not normal, you must normalize it or use methods that do not rely on normality. One way to determine the normality of data is to generate random data with the same mean and standard deviation as your data and compare histograms for similarity. Previous research took the approach of comparing the periodograms as well as histograms for similarity[5, 13, 14]. Varying levels of normality within the feature set has aided researchers in choosing the most appropriate fields for further study[4].

In addition to verifying the normal probability distribution of the selected corpus of data, it is important to understand the correlation among the various features being studied. We have supported the use of statistical analysis by verifying the pair-wise correlation among network features before proceeding to filter the traffic using advanced statistical models [5, 27]. Both Sang and Celenk [5, 28] rely on an auto-regressive moving average (ARMA) model for data smoothing.

2.1.4 Aggregation Techniques

Much research centers on the aggregation of network flows into clusters and sub-clusters based on IP address, port and protocol, where the highest level of clustering is defined as having a fixed source and destination IP [7, 20, 23]. The choice of keys for sub-clustering is dynamic based on the remaining features, the relative sample size of the sub-cluster compared with the parent cluster, and the weighting of the features used to
produce the sub-cluster. Random dimensions of records in a sub cluster are lost, but the entropy measure of the random dimension is maintained [20].

There is also research devoted to statistically predicting or detecting network anomalies through the analysis of network data. Thottan and Chuanyi claim that “network anomalies are characterized by correlated transient changes in measured network data that occur prior to or during an anomalous event” [33]. Some researchers attempt to find covert network channels or hidden overlying networks which try to be undetectable. In particular Gianveccio devotes research to finding covert timing channels [17].

Celenk et al. employed a calculation of average port to characterize overall network usage. Using a numeric average calculation on port numbers, which are identifiers whose value has no intrinsic meaning, was a novel approach. Although the actual value of the average port is meaningless, their research posited that the value should not change dramatically over time in a consistent steady-state network [5]. This contrived measurement of network state was transformed and expanded into a general measure of network entropy in later research [4].

Other researchers have investigated entropy as a means of identifying patterns in network traffic [15, 19, 24, 35]. Wagner and Plattner showed a strong connection between a worm outbreak and the overall entropy of NetFlow data. Although it is obvious that NetFlow data would increase in volume during a high volume network attack, this research team showed that entropy would decrease, as indicated by an increase in the percent of compression achieved while archiving NetFlow logs [34].

While Wagner and Plattner [34] used standard compression tools to easily calculate overall network entropy, other researchers struggled with the resource intensive computations necessary to produce entropy measurements of specific features by employing random sampling or periodic aggregation of results [24, 35].
2.2 Detecting Network Traffic Anomalies

Some research attempts to categorize the behavior of network users based on traffic features found in NetFlow or packet captures. Acharyya and Ghosh present a formalized method of ranking web browser traffic using entropy calculations based on the analysis of pages and links [1]. However most research taking a statistical approach will avoid the deep packet inspection needed for this kind of analysis either because the packet contents are not available or in order to save processing time.

2.2.1 Measures of Entropy

In common speech the terms entropy, probability, predictability and randomness can be used interchangeably. The probability of making an accurate prediction depends on the randomness of the event. For this reason entropy is often considered when attempting to predict anomalous events in network traffic.

2.2.2 Anomalies as a Deviation from Normal Traffic

By definition, any network traffic which does not match a normal pattern is an anomaly which is of interest to network security professionals and engineers. This simple notion demands that some network traffic can be considered normal. Many researchers assume that network traffic is Gaussian [21, 28, 36] while Celenk et al. have shown this to be true using a periodogram analysis [5]. A large relative difference between a baseline probability distribution and the observed traffic has been used as an indication of a network anomaly [18].

A slightly different research approach examines network traffic as a signal and uses statistical filters to transform the data for anomaly identification based on a threshold [5, 21]. Celenk et al. have advanced this method by devising an adaptive digital anomaly
detector (ADAP) which includes a feedback control mechanism to dynamically adjust statistical prediction parameters and thresholds [5].

### 2.3 Visualizing Network Security Data

Basic patterns of communication are easily visualized using directed graphs such as the one shown in Figure 1.1 and discussed at length in Chapter 4. Many researchers have addressed the problem of transforming massive amounts of data into meaningful security information through visualization and many of these methods are explored in the work of Conti[11]. The parallel axis graph is a common tool that is used by many to show complicated communication patterns [11, 22, 37]. Figure 2.1 is an example of a parallel axis graph illustrating a pattern of IP, port and protocol usage for a particular server at Ohio University. The traffic from any server thus graphed will have a distinctive pattern that may be recognizable to an analyst and may be used as a visual alerting mechanism[11].

Much more information is included in the circular axis graphs used by Foresti et al. [16]. These graphs show at-a-glance, the relative volume of information being communicated between a limited number of items. As an example, Figure 2.2 shows the relative volumes of communication between categorized ports at Ohio University.

Research by Celenk et al. takes a novel approach to graphing the time-varying relationship between entropy, mean, variance, and Fisher Linear Discriminant by using a symmetric vertical time axis graph with overlaying bar graphs and impulse point line graphs [4] as shown in Figure 2.3. In these graphs, the time series is shown vertically and the four calculations are shown graphically in changing colors to emphasize the relative magnitudes as a way of visually alerting the observer.
2.3.1 Higher-Order Statistics

For purposes of this discussion we employ a simple definition of higher-order statistics similar to other research[30]. The order is defined by the number of times you typically must read the entire data set to calculate the result. To calculate a mean, for example, you typically read the whole data set once. Two passes are required to calculate the deviation; since each value must be compared with the result of the first pass - the average. The calculation of standard deviation requires yet a third pass, to calculate the average of the second pass. The need for multiple passes through a data set seems to preclude the ability to analyze a stream of data where the later records are not yet
available. However, this kind of on the fly analysis is vital for investigating network traffic in near-real-time, where we define near-real-time processing as any data analysis that takes less time than the data represents. In other words, if we can process one-second worth of data in less than one second, then we can analyze a stream of time stamped information without falling behind. To overcome the problem of requiring multiple passes we borrow from the field of digital signal processing (DSP) and calculate our higher-order
statistics based on methods used in moving average filters[32]. Figure 2.4 shows the method of calculating higher-order statistics on the fly in C++.

2.3.2 Multivariate Distance

Much network traffic research involves observing a stream of multivariate data for changes over time or between different filter criteria. We are therefore faced with the question of determining how different are the two observations, or how much change has
there been over time. A comparison such as this, between two multiple feature sets, requires a meaningful measure of multivariate distance.

This research considers a set of $n$ network traffic features $P = (p_1, p_2, \ldots, p_n)$ as a point in $n$–dimensional feature space. The FLD measurement of previous research[4] is replaced with a simple multivariate Euclidean distance defined as

$$ E_{distance} = \sqrt{\sum_{i=1}^{n} (w_i p_i - w_i q_i)^2} \quad (2.1) $$

which gives the actual distance between two points in multidimensional space based on the Pythagorean theorem. The feature values in sets $P$ and $Q$ may be weighted by $W = (w_1, w_2, \ldots, w_n)$ or normalized so that each set receives the same bias adjustments. Similar to the discussion of “average port” in Section 4.1, the actual value and magnitude of the distance only has meaning in comparison with similar measurements from other sets. Since the distance measurement is only used for comparison with other sets, the distance formula may be simplified by removing functional constants. Therefore, the Euclidean formula is simplified to

$$ E_{simple} = \sum_{i=1}^{n} (p_i - q_i) \quad (2.2) $$

Figure 2.4: Calculation of Moving Standard Deviation in C++, given a NetFlow Record nfs[i], Window Size (N), and Weighting Factor (F)
for relative comparisons. Inherent in this distance formula is the measure of multivariate magnitude for any particular time-frame or bin with feature set $P = (p_1, p_2, \ldots, p_n)$.

$$E_{\text{magnitude}} = \sum_{i=1}^{n} (p_i)$$  \hspace{1cm} (2.3)

In words, we can simply add all the measurements together for comparison with a similar feature set.

This magnitude measurement can be used for stochastic analysis of network traffic data when seen as a time-series, or it can be used as a multivariate measure of noise-floor. In this way, a simple noise-floor can be weighted to include consideration of multiple features. However, we do need to be careful when relying on a measure of Euclidean distance in a non-Euclidean feature space. In Cartesian space, the units of all the dimensions are the same. Thus, the resulting measurement has meaningful value in that space. Using a feature set with different units of measure, e.g., frequency, bytecount, port and protocol, yields a measure of distance whose value has no actual meaning.

Similar to “average port”, this measure of distance can indicate a change in state without giving any specific information about the nature of the change. Two points in the feature space may change position drastically but remain the same distance apart, just as two measurements may change value and have the same average. Actual state changes that reflect no change in the state measurement are a chance events and as such do not hinder the statistical analysis of this data since random effects are by definition statistically normal.

The limitations of state change measurements are addressed in Chapter 3 where we employ domain-specific embedded languages (DSL) to interpret changes in individual features.
2.4 Signal Filtering and Pattern Recognition

In digital signal processing, it is necessary to calculate higher order statistics on the fly using a sliding window of measurements. Statistics must be updated as each new measurement is received since the signal, or stream of data may be unending or you may need to act upon completed calculations without waiting for the signal to end. For this reason we calculate average and standard deviation on the fly or “moving” as described below; this also results in more efficient computer programs.

2.4.1 Moving Average Frequency

The common way of calculating an average for a window, or sample of size, of \( N \) is shown in equation 2.4 and is well known in digital signal processing[32].

\[
\text{simple moving average} = \frac{\sum_{j=1}^{N} \text{measurement}[i+j]}{N} \tag{2.4}
\]

Note that this calculation is only valid if \( N \) measurements have been recorded. Implementation of this simple approach may be problematic when large window sizes are needed as it requires that \( N \) samples be kept in memory in order to be summed and divided by \( N \). Instead we have employed another method which requires very little storage and is easily modified for weighting the average toward the beginning or end or the of the window. Equation 2.5 shows the calculation where \( N \) is the window size and \( W \) is a weighting factor less than \( N \).

\[
\text{average}_i = \frac{\text{average}_{i-1} \times (N-W)}{N} + \frac{\text{measurement}_i \times W}{N} \tag{2.5}
\]

The approach used in Equation 2.5 is better suited to software applications where memory space may be limited and window sizes are large, since only one previous measurement must be retained. A weighting factor \( (W) \) of 1 simulates the value received from Equation 2.4 as it gives equal importance to all measurements, including the last one. Smaller weighting factors tend to smooth out spikes in the average as a single drastically
different value has less effect (weight) in the calculation. As in Equation 2.4 the
calculation in only valid after \( N \) samples have been measured. In a sense, the operation
using widow size (\( N \)) and weighting factor (\( W \)) simulates a pass through the entire
sample, which does not have to be kept in memory.

It is important to consider the size of the time window when calculating statistics.
Another consideration is the increment used to move a window. The increment can range
from 1 unit to the size of the window itself. Moving the window by an increment larger
than the window itself may be a valid research goal as it may require less computing
power, but it results a sampling of the data since some records will be skipped.

In particular, the selection of window size affects the “per-time” functions of the
implemented NetFlow filter language discussed in Chapter 3. Division by time values
greater than the chosen window will be valid only by extrapolation. For example, if you
want to calculate bytes per minute, a window size of 1 minute gives the most accurate
results, a window size of 10 minutes gives you an average per minute value and a window
size of 1 second gives you a per minute calculation based on extrapolation. It is best to
pick filter functions to match your window size.

### 2.4.2 Moving Standard Deviation

The calculation of standard deviation typically requires 3 passes through the data;
pass one calculates the average of all measurements, pass two calculates the deviation of
each measurement from the average, and pass three calculates the average or standard
deviation. By successively applying the operations involving weighting factor \( W \) and
window size \( N \) defined in Equation 2.5 we can calculate the moving standard deviation in
one pass through the data. The resulting value will be available continuously as additional
measurements are observed. The result is valid only after the minimum window size of
measurements has been recorded but the estimate until that time may be improved by
seeding the result with an appropriate value at the start. This calculation of a moving standard deviation is shown in Figure 2.4 as implemented in C++.

2.4.3 Digital Signal Processing Techniques

The principles of moving-average and window size have been used extensively by Ohio University network engineers and security analysts to detect anomalous behavior on the network. More recently, researchers at the Russ College of Engineering have taken the simple notion of moving average and created a more sophisticated Adaptive Digital Anomaly Detector (ADAP)[5]. Figure 2.5 shows the implementation of the ADAP filter in MatLab. Previous research at Ohio University has tested detection methods based solely on statistical behavior calculated from short term traffic observations of NetFlow data [4, 5]. Approaches such as this rely on finding malicious patterns of network behavior and do not suffer from the problem of changing payload signatures. Instead, an attacker may try to hide the malicious traffic by changing its timing or network behavior. Methods of obscuring malicious traffic could include using well known “legitimate” ports, or flooding the network with seemingly legitimate traffic in an attempt to confuse the detectors.

Network flow data has been extensively studied as a time series. Although, most efforts simply assume correlation among network features, Celenk et al. have supported this emphasis by verifying the pair-wise correlation among network features before proceeding to filter the traffic using advanced statistical models [5, 27].

2.4.4 Calculating Entropy On The Fly

Although entropy is often used to analyze network traffic[17, 18, 23, 24] it can be difficult to calculate, having large space and time requirements depending on the type of information being quantified. Real numbers must be binned with a loss of information, and countable values require storage for each possible value; i.e., calculating entropy for
% Adaptive Digital Anomaly Predictor (ADAP)
% Ohio University, Electrical Engineering and Computer Science (2008)

function [filtered,estimate] = adap(signal)

ww = 100; % Wiener Window
ws = 10; % Window Size for cross correlation
pc = 5; % Parameter Count (ARMA coefficients)
ss = numel(signal); % Signal Size
filtered = wiener2(signal, [ww 1]);
estimate = zeros(ss, 1);
for i = pc:ss-ws-1,
    % calculate the cross correlation
    xc = xcorr(filtered(i:i+ws,:), pc, 'coeff');

    % Just use the right half of the coefficients starting with "1"
    % (This is done only to simplify the reading of the next line)
    ac = xc(pc+1:-1:1);

    % Calculate the estimated value at i+1
    estimate(i+1) = ((ac(1)*filtered(i) + ac(2)*filtered(i-1)
                     + ac(3)*filtered(i-2))/3);
end

return;

Figure 2.5: Russ College of Engineering ADAP Filter Calculation in MatLab, showing use of Wiener Filtering and Cross Correlation[5]

an 8 bit value requires 255 storage bins and 16 bit values such as port, require 65,535 storage locations to calculate entropy. A true entropy calculation for a 32 bit value such as IPv4 address becomes unwieldy as over 4 billion frequencies must be stored.

Since the calculation of entropy $E(X)$ as shown in Equation 2.6 requires that the probability $P_x$ be known for each of $N$ possible values of $X$, we can think of no way to calculate entropy on the fly, without storing all of these probabilities.

$$E(X) = - \sum_{i=0}^{N-1} P_x \log_2 P_x$$

(2.6)

The computational difficulty of computing entropy for complex data such as NetFlow information can be avoided by considering the data as a simple stream of bytes without regard to the actual structure of the data it contains. Calculating entropy on this simple data representation is easy, requiring only 255 bytes of storage, and may be used as a
indicator of overall entropy for purposes of indicating drastic state change in a system[34] although the entropy of specific features is lost.

2.4.5 Binning Strategies

A binning strategy is needed when analyzing measurements that are recorded as real numbers, or when frequency is calculated for countable data types with a large number of possible values. The network traffic data used in previous research was processed using a C++ class called BinMap which later became the flow-map class and application described in Section 3.4.3[4, 5, 6]. Both of these classes are standard library maps of time value to NetFlow record. These classes have been painstakingly optimized to pre-process NetFlow data into a multidimensional structure as illustrated in Figure 2.6. After considerable code optimization and logical organization these classes allow analysis of data in near-real time for the amount of traffic passing the Ohio University Internet boundary.

BinMap uses this binning strategy to collect statistics over time for stochastic analysis, and flow-map stores information in a object we are calling a “Flow” which is keyed by an identifying feature of the network traffic. The feature most often used is source address; however the final application allows switching to destination address and future implementations may allow complex keys like source and destination combination. Each Flow object in flow-map is analogous to a bin map as described by Celenk et. al.[4, 5, 6].

The flow-map application was designed to analyze a stream of information and alert as the data is received from the collector; therefore, we do not consider record durations in this implementation. The issue of record duration is mitigated with the implementation of a configurable time-frame window in flow-map. Analysts typically choose a window size spanning an entire file or multiple files when doing ad hoc analysis. Moreover, running
Figure 2.6: Multi-dimensional Data and Binning Strategy for Stochastic Analysis Research

flow-map with a smaller window-size is specifically meant to identify and trigger on acute events, which by definition, do not include the long duration records.

Other researchers have faced the problem that extant NetFlow records do not fit nicely into time bins because of varying durations. Some research attacks this problem by fragmenting longer duration records and spreading their values over a range of bins [29]. The same approach was used early on in this research and for stochastic analysis.

However, this approach makes processing in near real-time difficult since information from long-term records must be kept in memory until the data can be spread over the longer time period.
2.4.6 Clustering Techniques

Some research centers on the aggregation of network flows into clusters and sub-clusters based on IP address, port and protocol, where the highest level of clustering is defined as having a fixed source and destination IP [7, 20, 23]. Often the emphasis is placed on distinguishing between the “normal” and “anomalous” patterns or behavior in computer networks.

Often NetFlow records are aggregated into clusters based on a common key such as a unique combination of source and destination address. Clustering allows for maintaining cluster statistics but information from specific records, which is not incorporated into the cluster key, is lost. The choice of keys for clustering may be dynamic based on the remaining features, the relative sample size of the sub-cluster compared with the parent cluster, and the weighting of the features used to produce the sub-cluster. Random dimensions of records in a sub cluster are lost, but the entropy measure of the random dimension is maintained [20].

Some research attempts to categorize the behavior of network users based on traffic features found in NetFlow records or packet captures. Acharyya and Ghosh present a formalized method of ranking web browser traffic using entropy calculations based on the analysis of pages and links [1]. However most research taking a statistical approach will avoid the deep packet inspection needed for this kind of analysis either because the packet contents are not available or in order to save processing time.

2.5 Entropy and Multivariate Statistics

NetFlow records contain measurements of multiple independent variables and so require either a multivariate analysis or a simple analysis of multivariate distance. Previous research has confronted this problem with a novel approach that looks for local maxima and minima in multivariate space using second- and fourth-order derivatives of
Fisher Linear Discriminant (FLD) of total entropy as a measure of distance. This approach allows the identification of anomalies in traffic data over small time periods without additional low-pass filtering [4].

Celenk et al. employed a calculation of average port to characterize overall network usage. Using a numeric average calculation on port numbers, which are identifiers whose value has no intrinsic meaning, was a novel approach. Although the actual value of the average port is meaningless, their research posited that the value should not change dramatically over time in a consistent steady-state network [5]. This contrived measurement of network state was transformed and expanded into a general measure of network entropy in later research[6].

Other researchers have investigated entropy as a means of identifying patterns in network traffic [15, 19, 24, 35]. Wagner and Plattner showed a strong connection between a worm outbreak and the overall entropy of NetFlow data. Although it is obvious that NetFlow data would increase in volume during a high volume network attack, this research team showed that entropy would decrease, as indicated by an increase in the percent of compression achieved while archiving NetFlow logs[34].

While Wagner and Plattner [34] used standard compression tools to easily calculate overall network entropy, other researchers struggled with the resource intensive computations necessary to produce entropy measurements of specific features by employing random sampling or periodic aggregation of results [24, 35].

2.6 Anomalies as Deviation from Normal

By definition, any network traffic that does not match a normal pattern is an anomaly which is of interest to network security professionals and engineers. This simple notion demands that some network traffic can be considered normal. Many researchers assume that network traffic is Gaussian [21, 28, 36] while Celenk et al. have shown this to be true
using a periodogram analysis [5]. Large relative differences between a baseline probability
distribution and the observed traffic are an indication of a network anomaly [18].

A slightly different research approach looks at network traffic as a signal and uses
statistical filters to transform the data for anomaly identification based on a threshold
[4, 21]. Celenk et al. have advanced this method by devising an adaptive digital anomaly
detector (ADAP) which includes a feedback control mechanism to dynamically adjust
statistical prediction parameters and thresholds [4].
3 IMPLEMENTATION

In this chapter, we describe the practical and theoretical problems encountered in the design and implementation of several tools for research in the Russ College of Engineering and for practical use by Information Security Office at Ohio University.

An effective tool for analyzing network traffic must be capable of processing massive amounts of data while providing complex statistical analysis and ad hoc logical constructs faster than the collectors can provide the data and fast enough to allow engineers to try many different analyses. These tools must also be flexible enough to allow ad hoc queries and experimentation by the college or for security investigations by the Information Security team.

This research effort and software implementation will focus on NetFlow data from Argus collectors, and Juniper JFlow collectors, although it has been tested with other data such as firewall logs which may be converted to an input stream matching the NetFlow 8-tuple defined in Figure 1.2.

3.1 History of Network Traffic Analysis at Ohio University

During the outbreak of the “SQL slammer” computer worm in 2003 the security team at Ohio university was tasked with identifying infected computers on the network. A simple Perl script called “src-by-flows” was developed to identify infected machines by recording the rate at which each computer was using port 1434, the Microsoft SQL port associated with the infection. Src-by-flows simply kept statistics on the use of that port, sorted the results and presented them to the analyst. The highest users of this port were known to be infected. This simple script was the only means available to identify infected computers on the network. Even in its simplicity it was only able to handle small portions of the network at a time due the massive amounts of traffic being generated by the worm. This outbreak prompted the development of a simple, but more flexible tool designed to
alert upon a drastic change in the volume of any identified type of traffic. The new application implemented in Perl, was called “NoiseFloor”.

3.1.1 Noise Floor Alerting

NoiseFloor was very flexible in that it could maintain usage statistics and alerting on any type of network traffic that could be identified by a regular expression when applied to a single NetFlow record. The use of regular expressions to describe a traffic feature allowed an infinite set of possible filters, that is, a filter for any set of flow data that can be described by a regular language. The NoiseFloor script used a calculation of moving average frequency of matches to the regular expression as a measurement of noise floor; a calculation that the author had previously used in private industry and is described in Section 2.4.1. These statistics are calculated over a sliding window in a manner that is efficiently accomplished by software implementations. The idea of using a sliding window to calculate higher-order statistics is the basis for much of the background research and publications leading to this thesis.

3.1.2 Ad Hoc Filtering

This simple filtering script was used to summarize and sort simple flow statistics for ad hoc analysis of traffic. The Information Security Office used the program called Source-By-Flows (src-by-flows.pl) extensively for this purpose. Implemented in the Perl programming language, both NoiseFloor.pl and src-by-flows.pl were limited in their speed and capacity if not their flexibility. The SQL Slammer outbreak taught us that a timely response to changes in network state was vital; a single instance of SQL Slammer could disable the network at that time. The interpreted language, Perl, did not offer the power and low-level capabilities needed to analyze Ohio University’s network traffic in the time frames necessary for responding to these violent outbreaks.
The next network analysis tools would need to be implemented in a lower level, more powerful language like C++.

3.1.3 BinMap and Research Tools

Researchers at the Russ College of Engineering decided to look at the possibility of detecting or predicting network anomalies based on the simple analyses described so far. The apparent random nature of the traffic measurements and the researchers’ notion that average port looked like entropy, led the team to investigate network anomalies using higher-order statistics and classical measurements of entropy. Unlike average port and average frequency, a measurement of entropy and other higher-order statistics require more information about the distribution of measurements in the past. The time window for the moving calculations is virtual, defined by the constant window size N and weighting factor W as in Equation 2.5. For the more sophisticated measurements needed by the Russ College researchers we employ an actual sliding window with the unavoidable increase in processor time and memory space that such a strategy requires.

The result of this long development life cycle is a set of tools including a NetFlow filtering utility called “flow”, and a application called flow-map which calculates statistics and keeps a map of IP and port connections over a sliding window. The application allows ad hoc filtering and alerting based on a domain specific embedded language and custom-built rules for identifying behavior of interest.

3.2 Design Considerations

Netflow analysis tools should run on the server that holds the data because the volume and sensitive nature of the information makes transmission across a network problematic. For this reason, most utilities operate with standard console input and output typical of UNIX applications. The flow-top application employes a curses based full screen interface similar to the UNIX “top” command because that offers the most
flexibility in a console environment. Research tools or analyst utilities, which require more detailed graphic visualization rely on GnuPlot, the DOT language of GraphViz, or the WxWidgets library for generating graphics.

3.2.1 ACL Filtering

An important part of any analysis of network traffic data is using successive ad hoc filters as a way of figuring out what is in the stream and what needs to be investigated further. It is also vital that unwanted information be discarded for efficiency before other tools such as flow-map or flow-top perform further in-depth analysis.

A simple and effective filtering mechanism is essential, however the NetFlow filtering mechanism employed by the standard flow-tools are difficult to use on an ad hoc basis. Filtering flows using the standard flow tools requires writing and maintaining complicated ACL files. We need a way to easily write dynamic ACL files, however Cisco ACL files are not easily manipulated by a manual process, which is prone to human error. For this reason, we developed a simple utility which is able to convert a user-friendly filter expression to the more complicated Cisco ACL file. Much of the research describe thus far has contributed to the development of such a filtering application called “flow”.

3.2.2 The problem with ACL Filtering

Based on Cisco style ACL files, this filter language has limitations because it acts on a single flow record at a time, deciding whether to permit or deny that record. The comparison is based on the static contents of the ACL file and has no facility for variable comparisons or decisions based on cumulative calculations. An example of a variable comparison would be a rule that denied records where the source port equaled the destination port; this rule is theoretically possible using static definitions but would require over 4 billion separate rules. Dynamic rules require an additional filtering mechanism to
be applied after the initial filtering and employ a wide range of methods ranging from simple Perl scripts to complex statistical analysis using MatLab or C++ applications.

One of the goals of flow-map is to be able to use a rule like what is shown in Equation 3.1 which is used to find machines on the network which may be sending spam or the rule shown in Equation 3.2 which is used to identify Skype super-nodes. Both of these rules require a program to keep track of port counts beyond the capabilities of a simple ACL file.

\[
\text{spam: } \text{dport}(25)/\text{min}>1000 \text{ || } \text{sport}(25)/\text{min}>1000 \tag{3.1}
\]

\[
\text{skype: } \text{count}(\text{daddr})/\text{min}>1000 \text{ && } \text{dport}(443)>1 \tag{3.2}
\]

### 3.3 Development and Production Environments

Various aspects of this research project are required to run on multiple systems and with disparate tool sets and capabilities. Some programs are ad hoc research tools that are only run once to gather data, other programs are developed for higher performance or reliability. In this section, we describe the considerations of development and production environments.

#### 3.3.1 Perl

The Perl language is very flexible and as such, it is a good choice for manipulating data quickly, for prototyping applications or for simple utility scripts. Perl has automatic data conversion and memory garbage collection that happen “under the hood” and are effectively beyond the control of the programmer. Additionally, there is a great deal of public sector Perl code available, which makes writing of complex applications very easy.

The lack of control over what is happening at a very low level is also what makes Perl less than desirable for some tasks. This research uses Perl extensively for one-off data experiments, prototyping and utilities, however we have found that for large complex
applications where memory manipulations must be precisely controlled, or when high performances is needed, that C++ is a better choice.

3.3.2 MatLab

For research requiring statistical and mathematical heavy lifting, a more specialized language is required. We used MATLAB for research and prototyping various statistical analysis of NetFlow data. Like Perl, this language is used for ad hoc tests and prototyping of applications which are later implemented in C++ code.

3.3.3 Graphics Packages

Much of the analysis for research and security office investigations is based on a visual representation of the data at hand. This is particularly important in the early stages of investigation when analysts are still trying to understand the nature of the data being studied or for finding informational artifacts whose existence is unknown. For instance, when visually examining graphs of network traffic it is easy to pick out overlying networks of communications or cliques that may be difficult to find computationally. These graphic representations are accomplished using the WxWidgets cross-platform C++ library, the Dot language format and the GraphViz utility, and the GnuPlot application.

3.3.4 C++

The difficult task of efficiently manipulating massive amounts of multi-dimensional data requires powerful programming tools and strict adherence to safe programming methods. Our experience has shown that this complexity is best managed using the standards enforced by C++ template programming, scope control, polymorphism and object oriented design.

We employ C++, as opposed to C code because C++ helps the programmer handle the complexity of code needed for large complex tasks. The bundling of data and code,
and the total control over the scope and lifetime of variables, including template programming and the Boost template library, provide the programmer with unprecedented power and flexibility.

As an example, the C++ language offers the power of references. A C program tries to be safe by checking that a pointer is not null before using it. In fact, there is no assurance that a pointer is valid even if it is not null. If it is valid, there is no assurance that it points to valid data. The pointed-to data may have already been deleted or it may be a type of data that is unexpected. Instead, the use of a C++ reference is preferred. A reference is really nothing but a pointer, however the programmer knows that a reference points to valid data and the data is exactly the type that he or she expects.

In fact, the haphazard use of pointers can be problematic for code maintainability, safety and even efficiency. Often it is too difficult to keep track of what code segment is still using a pointer or to know who should delete it when they are finished with it. Often, rather than bear the responsibility of determining who owns a pointer, a copy of the data will be made and passed onward instead. This is a very inefficient practice caused by difficulty in controlling the scope and lifetime of variables. Instead, we use a smart pointer, or reference counting pointer, which keeps track of how many code segments are using the actual pointer; this object acts like a pointer but can delete itself when the reference count reaches zero. In many cases the better option is to use a C++ reference instead of a pointer; a reference cannot exist at all if no one is using it. These built-in controls over the scope and lifetime of memory make C++ a language which, if it compiles, has a better chance of running safely.

This research makes extensive use of a reference counting template class called SmartPointer, which was written by myself circa 1997. The original class included multitasking functionality, which was removed for clarity and brevity in college coursework in 2004. The current version is listed in Appendix A. This class is an
encapsulation of a regular pointer. It implements a reference counted object similar to “auto_ptr” and “counted_ptr”, which didn’t make the ISO/IEC C++ standard in 1998[8]. The object maintains an external counter on an external pointer and deletes the memory when no more copies of the object are referring to it. This is a safe way to use real pointers and it prevents the need for copying actual objects. Copying only the encapsulated pointers drastically improves the actual run-time of programs.

Although the reason for this performance improvement seems obvious, it is described in some detail in the ISO/IEC Technical Report on C++ Performance [9]. All operations in this implementation are constant time for asymptotic complexity (O(1)). The space complexity is also constant since memory is only allocated once per instance created. In-fact the memory is static so, multiple instances of the class use the same memory for counters. There is memory associated with the actual objects that the pointer points to, but that is not part of this class. Again, this is just a wrapper around a pointer.

The real performance improvement in C++ comes from the overall toolset that lets the programmer simplify code to manage complexity. In order to achieve efficient, extensible and platform-independent applications, we make extensive use of C++ templates, container classes, the standard template library (STL). In addition, we employ the Boost template library in all C++ code described here. The benefits of using a powerful template library like Boost are illustrated by the fact that the ISO/IEC has chosen to include many of these functions as part of the core C++ language in 2011 as described in the new ISO/IEC Standard C++11[10].

3.4 The Applications

The research discussed thus far has aided in the development of several utilities that are currently used by the Information Security Office for ad hoc analysis of network traffic
data and for automatic alerting and logging of anomalous activity on the network. Figure 3.1 shows an overview of the data flow for typical network traffic analysis.

![Flow Diagram](image)

**Figure 3.1: Overview of flow data processing.**

### 3.4.1 The flow-macros script

A simple Perl script is used to download watchlists or rule sets from various security organizations that collect and maintain information about infected machines, botnet command and control networks or known malicious agents on the Internet. This script, called “flow-macros” can parse and compile the various watchlist formats into a macro configuration file suitable for use with flow filtering application. Flow-macros will download watchlists from the SANS Internet Storm Center, NESSUS rules from the Emerging Threats security website, and watchlists from the Research and Education Networking Information Sharing and Analysis Center(REN-ISAC). REN-ISAC collects information that is only available to members via user name and password.

Filter macros may be included in the data section of the Perl script itself or be kept in an external macro configuration file that is specified on the command line. These external
Figure 3.2: Flow Application Output showing macros defined internally and in the all.macros configuration file.

macro files are maintained automatically with separate scripts to update them from various security organizations, such as SANS, REN-ISAC, and Emerging Threats. In the case of Emerging Threats for example, we download a set of rules for the Nessus vulnerability scanner and convert these into the flow macro format. The internal space for storing
usage: flow [options] [<Date Spec>] [<Filter Spec>]

- Generate ACL file and run a flow-cat command with filter.
- Default time spec searches is the most recent file.
- Options can be included in any order in the list.
- Use option (-d) to debug the generated command and ACL file.
- Prefix items with "s:" or "d:" for source and destination.
- Date format is yyyymmddhh or with wildcards like 2011-12-0[56]-*
- Macros include: internal, facstaff, resnet, citynet
- Date can be any parsable format like 'today', 'yesterday' etc.

items include:

- IP addresses or networks in CIDR notation
- DNS name specifications (e.g., google.com, facebook.com)
- Port numbers (e.g., 80, 443, 6881)
- Protocols (tcp, udp, icmp) or proto:[number]
- Any defined macro describing a list of IPs or Ports

options include:

-h : help, shows this message
-d : debug, just print generated command and ACL file
-e : export fields for use with FlowMap
-c : console output for pipe to other flow tools
-m <macro file> : specify file containing external macro definitions
--show-macros : display internally and externally defined macros

examples:

- flow -d today 1.1.1.1
- flow -d 2011-09-24-12 1.1.1.1 2.2.2.0/23
- flow -d 2011-09-*-12 s:1.1.1.1 d:2.2.2.2
- flow -d s:1.1.1.1 2.2.2.2 and 80 2011-09-30-*

Understanding the embedded filter language:

The list of items is 'or'ed together left to right by default. Inserting the 'and' keyword causes the next item to be 'and'ed instead. This syntax is translated into the ACL filter file which can be viewed using the -d (debugging) option.

In addition, any items not prefixed by 's:' (source) or 'd:' (destination) are translated as 'either' source or destination. Source and destination doesn't apply to protocol.

Figure 3.3: Flow Application Help Screen

macros in the Flow application itself is meant for IP or port lists that do not change very often.
The resulting macro file is available to the flow application as an external file loaded
with the “-m” command line option. The macros currently available are displayed with the
“–show-macros” option as illustrated in Figure 3.2.

3.4.2 The flow filtering application

The flow application was designed to 1) ease the locating of the appropriate log files
based on a user friendly date specification, 2) allow the input of filter criteria using logical
Boolean expressions 3) handle complicated lists of IP addresses using simple macro
specifications and 4) run the generated command for console display or for further
processing by other UNIX style utilities. Several date formats are accepted and a Boolean
filter expression is comprised of IP addresses, DNS host name specifications, CIDR
network specifications, or predefined macros.

The real strength of the flow application is in taking these complex parameters in a
user-friendly form and converting them into an even more complicated Cisco ACL file.
We deal with this complexity by using a formally designed grammar, shown in Figure 3.4
which describes a NetFlow filtering language. The language supports fully recursive
Boolean expressions whose predicates describe the essential parts of a ACL file including
CIDR notation of IPv4 addresses, IP port numbers, protocol specification, DNS lookups
and macro definitions of IP address lists.

These items are logically combined using the keywords “and”, “or”, and “not” along
with parenthesis to specify operator precedence. Each item, with the exception of
protocol, may be prefixed with “s:” for “source” or “d:” for destination indicating the
direction of the data flow. Additionally, the filtering utility assists the user in the onerous
task of manipulating the date and time specification in order to glob the correct log files.

An important feature of this tool is the “debug” option which is invoked with a “-d”
on the command line, since the input Boolean logic and the Cisco ACL file output can be
Figure 3.4: Simple grammar for converting logical filter expressions to Cisco ACL files in Perl.

```
start : expr eos { $item[1] }
expr : conjunction { $item[1] }
    | disjunction { $item[1] }
    | factor { $item[1] }

conjunction : factor /and/oi expr { filter::And($item[1],$item[3]) }
disjunction : factor /or/oi expr { filter::Or ($item[1],$item[3]) }
    | factor expr { filter::Or ($item[1],$item[2]) }

factor : /not/ factor { $item[2]->Negate() }
    | item { $item[1] }
    | '(' expr ')' { $item[2] }

item : /not/ item { $item[2]->Negate() }
    | IPCIDR { $item[1] }
    | PROTO { $item[1] }
    | DNSNAME { $item[1] }
    | PORT { $item[1] }
    | MACRO { $item[1] }

# support cider (/slash) notation of IP addresses
IPCIDR : /'(s|src|d|dst):)?\d+\./\d+\./\d+\./\d+\/\d\d\d\d/oi
{ new filter('cidr', $item[1]) }

# common protocol tags or "proto:number"
PROTO : '/(tcp|udp|icmp|:\?proto|:\d+)/io
{ new filter('ip-protocol', $item[1]) }

# reasonable length words and dots are DNS names
DNSNAME : '/'(s|src|d|dst):)?\w(2,30)\./\w(1,10)\./?/o
{ new filter('dns', $item[1]) }

# lone numbers are ports
PORT : '/'(s|src|d|dst):)?\d\d\d\d/oi
{ new filter('ip-port', $item[1]) }

# any other word could be a macro definition
MACRO : '/'(s|src|d|dst):)?[a-zA-Z][a-zA-Z_0-9]*/oi
{ new filter('macro', $item[1]) }

eos : /\Z/}
```

quite complicated. This is particularly true when the user does not explicitly specify
source and destination tags for items in the logic as these items are automatically
expanded into explicit “or” statements. The debug option reproduces the input Boolean
expression in a fully expanded canonical form so the user will know exactly how the input
was interpreted. It also shows the produced ACL file so the user can check that this was the intended filter operation.

The debug option is also helpful in the input Boolean logic. Equation 3.3 shows a command which might not produce the output that is expected because of the way the Boolean expression is parsed with operator precedence left-to-right and because the unspecified source and destination of the two macros, “facstaff” and “resnet”.

\[
\text{flow facstaff or resnet and 80 -d} \quad (3.3)
\]

\[
\text{flow (facstaff or resnet) and 80 -d} \quad (3.4)
\]

Figure 3.5 shows the result of debugging the command in Equation 3.3. The canonical expression on line 3 shows that each item is explicitly expanded to be an “or” of source and destination, and items are grouped in parentheses from left to right. This is probably not what the user expected since the ACL will permit anything to or from facstaff regardless of port as is clearly seen from lines 36 and 38 in Figure 3.5. Equation 3.4 shows a corrected command that is less confusing and produces the desired output as shown in Figure 3.6. Note the use of parenthesis in making the command less confusing. Parentheses may need to be prefixed, or “escaped” using the slash (\) in order to be accepted by the UNIX command line. Alternatively, the entire filter expression can be quoted.

The Flow application creates a temporary ACL file as shown in Figures 3.5 and 3.6 and generates a system command to read this file and filter NetFlow records for display to the screen or for further analysis by other tools. The complete flow help screen is displayed in Figure 3.3.
Figure 3.5: Flow Debug Results with Canonical Expression and ACL File Produced.
# source expr -> (facstaff or resnet) and 80
# canonical expr -> (((s:facstaff|d:facstaff)|(s:resnet|d:resnet))&(s:80|d:80))

filter-primitive s:facstaff
  type ip-address-prefix
  permit 132.235.0.0/16
  default deny

filter-primitive d:facstaff
  type ip-address-prefix
  permit 132.235.0.0/16
  default deny

filter-primitive s:resnet
  type ip-address-prefix
  permit 64.247.64.0/18
  default deny

filter-primitive d:resnet
  type ip-address-prefix
  permit 64.247.64.0/18
  default deny

filter-primitive s:80
  type ip-port
  permit 80
  default deny

filter-primitive d:80
  type ip-port
  permit 80
  default deny

filter-definition default
  match ip-source-address s:facstaff
  match ip-source-port s:80
  or
  match ip-source-address s:facstaff
  match ip-destination-port d:80
  or
  match ip-destination-address d:facstaff
  match ip-source-port s:80
  or
  match ip-destination-address d:facstaff
  match ip-destination-port d:80
  or
  match ip-source-address s:resnet
  match ip-source-port s:80
  or
  match ip-source-address s:resnet
  match ip-destination-port d:80
  or
  match ip-destination-address d:resnet
  match ip-source-port s:80
  or
  match ip-destination-address d:resnet
  match ip-destination-port d:80

Figure 3.6: Flow Debug Results with Canonical Expression and ACL File Produced.
3.4.3 The flow-map application

The flow-map application uses an object oriented class structure to calculate and maintain a map of all connections along with higher order statistics for each connection. Written in C++, this program processes output from the flow application and produces data for further analysis by information security analysts. Each unit of communication described by a single NetFlow record is encapsulated in a smart-pointer object. These encapsulated NetFlow records are used to create “Flow” objects which are kept in a C++ standard library map using the source IP address as a key. The flow-map class structure efficiently maintains multiple dimensions of Network Traffic data as illustrated in Figure 3.7 which is an annotated screen print of the flow-top application.

![Figure 3.7: Data Dimensions of flow-map data structure as shown by the flow-top application screen.](image-url)
The real work of maintaining the flow statistics and the map of communications is handled by template member objects of type “stat_map”. Using a C++ template methodology, the stat_map class is able to maintain information about any integer type traffic feature. Real number type features may be summarized by stat_map if the values are binned or rounded into integers. For example a stat_map of network feature “IP address” will keep a map with counts of every unique IP address that is encountered. For the reasons discussed in Section 2.2.1 this is the minimum amount of information that is required for calculating mean, standard deviation, and entropy for any traffic feature. The stat_map also keeps track of the time-frame for the set of NetFlow records encountered and so can calculate statistics on a per-second, -minute or -hour basis. Figure 3.8 shows the basic structure and inheritance of the stat_map class.

The flow-map program structure also incorporates a domain specific embedded language for recognizing patterns of behavior in network traffic. The grammar shown in Appendix B defines this language, which is used to write user-defined rules for matching patterns of network behavior. These rules are evaluated in the context of the accumulated statistics and any successful parsing of a rule signifies a match for that behavior pattern. The results of these pattern matches are displayed to the screen in flow-top, are used to trigger automatic actions by other scripts, or may be relayed to a security information and event manager (SEIM) device for further action or analysis.

The user-defined rules used by flow-map are defined in a text file that is passed as a parameter on the application command line. A rule consists of a rule name followed by a colon (:) followed by a Boolean expression defined by the grammar shown in Appendix C. Equation 3.2 shows an example rule which might be used to identify Skype supernodes on the network; in addition, it is possible define macros which are used to label subsets of ports or IP addresses with more meaningful names. For instance, rather than writing a rule
that specifies the ports ranging from 6881 to 6999, you may want to define a macro specifying that range and labeling it “bittorrent” as it represents the ports typically used by that file sharing program. Similarly, a macro like “resnet” could be created to define the IP address range (64.247.64.0/18) used by the Ohio University residence hall network. These name tags can be used in rules in place of the actual port or IP values as shown in Figure 3.10.
Ohio University -- Information Security Office -- utilities -- flow-map

usage: flow-map [options] "<input command>"

Generic options:
-h [ --help ] produce help message
-f [ --config-file ] arg specify configuration file
-r [ --rules-file ] arg specify rules file
-t [ --tags-file ] arg specify tags file
-p [ --ports-file ] arg specify ports file

Configuration:
-w [ --watchlist-file ] arg specify watchlist file
-i [ --input-command ] arg specify input command
-n [ --sort-column ] arg sort by column
-s [ --window-seconds ] arg (=0) window size in seconds
-W [ --who-is ] show whois domain names
-d [ --destination-ips ] show destination IPs
--no-display no display, just run rules

examples:
flow-map "ra -r /data/argus/archive/dcnets/netmon2/argus.*.gz - net 192.235.0.0/16"
flow-map "ra -r /data/argus/archive/netscan2/argus.*.gz"
flow-map "ra -r /data/argus/archive/netmon2/argus.091224*.gz"
flow-map "ra -r 'argus-recent.pl'"
flow-map --window-seconds 86500 "ra -r 'argus-today'"

aliases:
src-by-flows = flow-map "ra -r 'argus-recent.pl'"
dst-by-flows = flow-map -d "ra -r 'argus-recent.pl'"

Figure 3.9: Help Screen for Flow-map or Flow-top Applications.

3.4.4 The flow-top application

The flow-top application does the same statistical analysis and mapping of connections as the flow-map application but displays the results continuously to a full console screen as shown in Figure 3.11. The screen is updated at a default specified time interval of two seconds.

Since flow-top is meant to run continuously, the time window feature (windowseconds) is important for this application. This feature causes the application to calculate statistics for a sliding window of time instead of accumulating the statistics continuously. This is helpful and necessary when looking for specific events in time. The
Sample Alerting Rules for flow-map or flow-top

Many connections on X-Windows port per minute
xwindows : tagmap(xwindows)/minute > 1000 || tagmap(xwindows)/minute > 1000

count connections on IRC - Direct Client to Client for BotNet Control
irc-dcc : tagmap(ircdcc)/minute > 100 || tagmap(ircdcc)/minute > 100

node == connections to a lot of different machines in short time
node : count(daddr)/minute > 1000

generic supernode == could be p2p on an unknown port or skype or gaming
supernode : count(daddr)/minute > 10000

skype == lots of connections, not a lot of bandwidth, port 443
skype : count(daddr)/minute > 1000 && ( dport(443) > 1 || sport(443) > 1 )

Bandwidth rules:

volume : ( sbytes + dbytes )/minute > 50m
bandwidth : ( sbytes + dbytes ) > ( total(sbytes) + total(dbytes) ) / 50

This example rule fires like the rule above but only for bursts of usage
greater than the threshold (100) per second
burst : (((sbytes + dbytes)/second) > ((total(sbytes)+total(dbytes))/10) )

more than one tenth of the traffic is IPv6
IPv6 : proto(IPv6) > flows/10

these silly rules test the parser for parentheses, not, add, mult and div etc.
silly1 : (((sbytes/second+((dbytes/second)/(2+1*2/2))))+1/1 > 40k)
silly2 : count(sport)==count(dport) || !(1==1)

Figure 3.10: Sample Flow-map Rules File Showing the use of Port and IP Macros.

user-defined alerting rules are evaluated for the flows contained in the current time window. The results of the evaluation are shown on the screen as they occur or are logged and transmitted to the SIEM for further analysis.
### 3.4.5 The flow-dot utility

The flow-dot utility is a simple Perl script for converting raw NetFlow data into directed graphs like the one shown in Figure 1.1 and discussed further in Chapter 4. As described in the listing in Appendix D, the program keeps track of all IP address connections and produces a “dot” language file which can be displayed using a graphing utility such as GraphViz. Equation 3.5 shows the command which generated the directed graph source (resnet.dot) for Figure 1.1.

```
flow (s:resnet and d:resnet) -c | flow-dot > resnet.dot
```

Figure 3.11: Curses style console output of the flow-top application
3.4.6 The flow-nessus application

The flow-nessus application is an example of post-processing the results of flow filtering and flow map analysis. A particular filtering and analysis based on user defined ACL files and alerting rules will produce a list of IP addresses that need further analysis. This application, which is currently under development, receives the flow output and launches a nessus vulnerability scan on machines that have been flagged. The results of the scan can be used to verify the diagnosis of the pre-analysis or can be used to identify false positive results.

This example of post-processing relies on the ability of applications to interpret the results of previous flow analyses. This is accomplished using embedded system calls which open a pipe to the flow-export application as shown as a Perl command in Equation 3.6 and as a C++ function in Figure 3.12. Using this method, any application can be created to post-process results of the tools described in this research.

\[
\text{open STDIN, `\texttt{flow-export -f2 -mSRCADDR,DSTADDR |}`;} \quad \text{(3.6)}
\]
```cpp
bool InputPipe() {
    if ( isatty(fileno(stdin)) ) { return 0; }
    int fd[2]; pipe(fd);
    if (!fork()) {
        close(1);   // close normal stdout
        dup(fd[1]); // make stdout same as fd[1]
        close(fd[0]); // we don't need this
        execlp("flow-export", "flow-export", "-f2",
               ",-mUNIX_SECS,DOCTETS,SRCADDR,DSTADDR,SRCPART,DSTPORT,PROT", NULL);
    } else {
        close(0);   // close normal stdin
        dup(fd[0]); // make stdin same as fd[0]
        close(fd[1]); // we don't need this
        return 1;
    }
    return 0;
}
```

Figure 3.12: C++ Function to Open Embedded Pipe for Converting Flow Input.
4 RESULTS AND OBSERVATIONS

4.1 NetFlow Fields as Indicators of Network State

This research has focused on a minimal set of network traffic features that is readily available from various data sources. Here we describe how these features have been used in this analysis. The NetFlow field “port” contains Internet Assigned Numbers Authority (IANA) registered port numbers as defined in RFC6335[12]. The port number only has meaning as an identifier – it indicates the service or channel that is being used for communication, analogous to a telephone number – but “port value” has no meaning as a measurement. The fact that port 22 is close to port 25 says nothing about the relationship between those two services. By similar logic, one may argue that the average of port values is a meaningless value. It would be like taking the average of two telephone numbers. When we average our two example ports (22 and 25) we get a value of 23.5 which is not apparently meaningful and not even a valid port. However, we contend that average port may be a valid calculation when used in relative comparisons as an indicator of change in the state of a network over time. Even though the calculated value has no meaning, the fact that it is changing indicates a change in the behavior on the network.

As an indicator of a state change, “average port” may be similar to entropy in usefulness. A change in the value of entropy indicates some change in the network without offering any specific explanation for the change. Though entropy is often used for NetFlow analysis[17, 18, 23, 24], we have found it to be difficult to calculate, having large space and time requirements depending on the type of information being quantified. Real numbers must be binned, with a loss of information, and countable values require storage for each possible value, i.e., 8-bit byte entropy calculations require 255 storage bins and 16 bit port entropy requires 65,535 storage locations. A true entropy calculation for a 32-bit value such as IPv4 address becomes impractical as over 4 million frequencies must
be maintained. Comparatively the calculation of an average port is very easy and may server as a good substitute for entropy.

When analyzing network traffic we often need to calculate the average frequency of a measurement as opposed to the average of the measurement itself. For example, in noise floor calculations we require a moving “average frequency”, not the moving “average”. That is, we need the average of how many of a particular measurement we are seeing, not the average of the measurement itself. The average and “deviation from average” (standard deviation) is also calculated on the fly using a weighted moving average frequency calculation as shown in Figure 2.4. The “frequency” of occurrence of a particular port however, is a useful measurement. An increase in the use of a particular port is often seen as a red flag by network security engineers. For example, experience has shown that inordinate amounts of outbound traffic on the port used for email transmission is a indication of a virus infection, as the infected computer begins to send large amounts of spam. An automated tool using simple statistics can easily detect when this simple frequency measurement exceeds a predetermined threshold and can take action to remove the machine from the network or to notify the owner of a policy violation.

Timestamps within network flow data are by necessity a statistical artifact, since it is the collector that must decide when to start and stop a record. These start and stop times do not necessarily reflect the start and stop of connections that may exist in the underlying protocols. This phenomenon of artifacts is most evident when analyzing NetFlow log files. When a collector is closing a log file, it must arbitrarily stop recording statistics in a particular record so it can write that record with the current end time. When starting up, the collector must start new records with arbitrary start times, even though the data represents a pre-existing underlying connection.
Figure 4.1: Plot of Argus record start and end times showing historical time artifacts (left) and more recent managed time stamps (right).

Time-based analysis of NetFlow records requires a start and stop time indicating the duration of the statistics for that record. Figure 4.1 shows Argus NetFlow record start and stop times where start times are offset 600 seconds earlier to distinguish them from the stop times. For this reason the start times appear as a line below the stop times, though in fact they overlap in the scale of these graphs. In the left-side figure, the varying distribution of stop times above the distinct line of start times indicate widely varying record time spans. The distinct horizontal limit to stop times is an artifact of the log file closing on the hour. The start times however appear to be monotonically ascending and this was verified programatically. The two distinct lines of start and stop times on the right hand side show more recent records from an Argus collector which was configured to produce more consistent record time frames. Both the start and stop times appear to be somewhat monotonically ascending; however the thickness of the apparent lines show that the times are not strictly monotonically ascending. The time spans for any record are limited to 60 seconds, as indicated by the thickness of the lines on the right graph and verified programatically.
Figure 4.2: Analysis of JFlow record times showing given values Record Time (RT), Record Up time (UTR), First Measurement Up time (UTF) and Last Measurement Up time (UTL).

Recorded times in Juniper Netflow records also require some explanation as the actual times for the beginning and end of the interval are not given. Instead, the records contain measures of ‘up time’ or the number of milliseconds since the start-up of the collector system. This is not readily apparent and any custom time-based analysis must calculate record duration explicitly. As illustrated in Figure 4.2 the given values include ‘up time’ at the first record in the interval \( (UTF_{ms}) \), ‘up time’ at the last record \( (UTL_{ms}) \) and ‘up time’ when the record is written \( (UTR_{ms}) \). In addition, the ‘actual time’ \( (RT_s) \) in seconds is recorded when the record is written. The actual interval times, in seconds \( (First_s, Last_s) \) must be calculated from these given measurements using the formula shown
in Equations 4.1 and 4.2.

\[ First_s = \left( RT_s - \frac{UTR_{ms}}{1000_s} \right) + \frac{UTF_{ms}}{1000_s} \]  

\[ Last_s = \left( RT_s - \frac{UTR_{ms}}{1000_s} \right) + \frac{UTL_{ms}}{1000_s} \]  

Table 4.1: Monotonic Ascension and Record Duration in NetFlow Data

<table>
<thead>
<tr>
<th>Record Type</th>
<th>Monotonically Ascending</th>
<th>Monotonically Ascending</th>
<th>Max. Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start Times?</td>
<td>End Times?</td>
<td></td>
</tr>
<tr>
<td>Historical Argus</td>
<td>Yes</td>
<td>No</td>
<td>3600 seconds</td>
</tr>
<tr>
<td>Recent Argus</td>
<td>No</td>
<td>No</td>
<td>60 seconds</td>
</tr>
<tr>
<td>JFlow Netflows</td>
<td>No</td>
<td>No</td>
<td>75 seconds</td>
</tr>
</tbody>
</table>

Table 4.1 illustrates that neither the recent Argus records nor the current JFlow records have monotonically ascending start times. As shown in this table, the maximum duration of each file type should determine the minimum window size used for any time based analysis.

The following commentary from the QoSient website explains a little bit about problems with Argus timing[3]:

Argus is pretty lazy as to when it will print out its records. This is so Argus will have maximum cycles for packet processing, rather than data output. Argus can be easily tuned to be more timely in reporting audit events, but without that tuning, Argus could take as long as 30-120 seconds to print out a particular record, depending on the load of the Argus, the protocol and when the last packet was seen.[3]
4.2 Normality and Correlation of NetFlow Data

Previous research supporting the selection of network features for study has investigated the normality of traffic features in terms of mean squared error (MSE) when compared with similar normal data. A lower MSE indicates greater accuracy when compared with random data of a similar distribution. As shown in Table 4.2, a feature that has a small MSE, such as Average Port, is more random or “normal” and so is better suited to statistical analyses. On the other hand, a feature with high MSE such as ‘Peer Factor’ provides less accuracy when compared with random data and therefore it is not selected for statistical analysis.

We begin with an expanded set of frequency based network features as shown in Table 4.2 from Celenk et al.[6], which describes the normality of network traffic features in terms MSE. As shown in Table 4.2, a feature that has a small MSE, such as Average Port, is statistically normal and is better suited to statistical analyses. On the other hand, a feature with high MSE such as “Peer Factor” provides less accuracy when compared with random data and therefore it is not selected for statistical analysis. We view this table as a listing of selected features ranging from very random at the top to very non-random at the bottom.

This interpretation of randomness is supported by experience when we consider the difference between average port at the top of the table, and peer factor at the bottom. We have already discussed how average port is observed to be a random value without any real meaning, thus the low MSE for average port is explained. In this research, ’Peer Factor’ is a measure of how often we observe a connection with the same source and destination port. Two machines using the same port is clearly not a random event, as the probability of this happening by chance are quite small[5]. Thus the high MSE result for “Peer Factor” is explained. In a client-server relationship, one port is a somewhat
Table 4.2: MSE sorted feature set used for normal density approximation and anomaly detection[5]

<table>
<thead>
<tr>
<th>%MSE</th>
<th>Measured Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.014403</td>
<td>Average port</td>
<td>Avg. port number as indicator of usage</td>
</tr>
<tr>
<td>0.055449</td>
<td>High-ports</td>
<td>Percentage of port numbers &gt; 10000</td>
</tr>
<tr>
<td>0.105316</td>
<td>Total ports</td>
<td>Number of ports seen</td>
</tr>
<tr>
<td>0.105888</td>
<td>Flow records</td>
<td>Count of flow records</td>
</tr>
<tr>
<td>0.119697</td>
<td>Total bits</td>
<td>Bits per second load on network</td>
</tr>
<tr>
<td>0.137958</td>
<td>Destination bits</td>
<td>Destination bits per second load</td>
</tr>
<tr>
<td>0.148073</td>
<td>Source bits</td>
<td>Source bits per second load</td>
</tr>
<tr>
<td>0.183783</td>
<td>Packets per second</td>
<td>Total packets per second</td>
</tr>
<tr>
<td>0.194217</td>
<td>Dest. packets</td>
<td>Destination packets per second</td>
</tr>
<tr>
<td>0.229815</td>
<td>Server factor</td>
<td>Measure of typical port usage</td>
</tr>
<tr>
<td>0.257600</td>
<td>Mid-range ports</td>
<td>Percentage of ports &gt; 1024 and &lt; 10000</td>
</tr>
<tr>
<td>0.284241</td>
<td>Low-range ports</td>
<td>Percentage of ports &lt; 1025</td>
</tr>
<tr>
<td>0.301142</td>
<td>Source packets</td>
<td>Source packets per second</td>
</tr>
<tr>
<td>2.109031</td>
<td>Total bytes</td>
<td>Total bytes per second</td>
</tr>
<tr>
<td>2.244572</td>
<td>Destination bytes</td>
<td>Destination bytes per second</td>
</tr>
<tr>
<td>4.569049</td>
<td>Source bytes</td>
<td>Source bytes per second</td>
</tr>
<tr>
<td>39.264246</td>
<td>Peer factor</td>
<td>Measure of same port usage</td>
</tr>
</tbody>
</table>

randomly selected port that is negotiated after the server is contacted on the known port. It is important to understand the correlation among the various features being studied and to verifying the normal probability distribution of the selected corpus of data. Celenk et al. have supported the use of statistical analysis by verifying the pair-wise correlation among network features before proceeding to filter the traffic using advanced statistical models
Both Sang and Celenk [5, 28] rely on an auto-regressive moving average (ARMA) model for data smoothing.

The shape of the frequency histogram is often used as a visual confirmation of the normality of data. Examination of the periodogram and analysis of autocorrelation have
also been used[5]. Figure 4.3 by Celenk et al[4] shows comparative graphs of the analysis of four features using these three visual methods. These results support the findings from Table 4.2 as the histograms for Average Port (E) appear to be more similar to normal than the graphs for ‘peered ports‘ (F). This research explains the difference by looking more closely at the population sample for “port”. While the histogram in (E) is generally bell shaped, it appears to be comprised of three separate bell shaped curves, indicating three separate normal populations within the total population. The graph in Figure 4.4 provides the explanation for this variance from normal as a whole. This graph shows a measure of port usage for every possible port from 1 to 65535. Distinct ranges appear because of different operating systems selecting their ephemeral ports based on different settings or algorithms. For instance, as of Windows Vista, Microsoft uses the range of 49152 to 65535, for its ephemeral ports, whereas older versions of Windows use the range 1024 through 4999[25, 26]. The three major populations shown in Figure 4.4 correspond to the three combined bell shaped curves in Figure 4.3. A fourth distinct population, ports less than 1024 do not show up in the graph at the scale shown but is revealed when the original data is plotted at a larger scale.

4.3 Identifying Anomalies Through State Change

Early use of netflows by the Information Security Office relied heavily on simple system scripts, which filtered and counted results for analysis by network security analysts. The first statistical analysis of this data by network security analysts at Ohio University started around 2004 with the idea of using network traffic noise floor calculations for multiple events. We define NoiseFloor as the average frequency of the event measured over a given time window. The measurement is most interesting when used to compare level of activity at different times or between different feature selection sets. For instance, a significant change in the amount of port 25 traffic (inferring email)
may be require investigation by network security analysts. Network engineers may need to understand how the amount of video streaming traffic changes during the course of a day. Overall noise floor or level of activity is not that interesting; instead, we must calculate individual noise floor measurements, for specific services or networks within the greater mass of traffic. This calculation must be made on the fly or “moving” so that alerts can be sent in a timely manner when events of interest are detected.

Figures 4.5 and 4.6 are a snapshots of a scrolling video graph using “average port” as a measure of noise floor. An event of interest is shown in the center of the graph where the measure of average port drops well below -3 standard deviations as indicated by the straight horizontal line at a value of just over 30,000.
Port entropy as defined by Equation 2.6 and discussed in Section 2.5 is plotted in Figure 4.7 for the same time period as Figures 4.5 and 4.6. A visual inspection shows that there is a decrease in entropy at the same time as decrease in average port. The decrease in average internal port is a consequence of an increase in the proportional usage of a lower port or ports; similarly, the increase in the measurement of external port average is created by an increase in higher external ports. This change in behavior is reflected by the change in entropy.

We can further verify the nature of the event by examining the median port values at the time of the low entropy. Table 4.3 shows entropy and median values corresponding to
the entropy spikes, below a threshold of 8.0 bits, in Figure 4.7. As predicted, the lowest entropies indicate a median value with a high count. The lowest entropy value in the table, 4.68, is calculated from a set of 1000 records comprised of 600 port 2967 measurements and 600 port 6000 measurements; each record has 2 ports, a source and destination. All other entries show less repetition, therefore higher probability, and therefore lower entropy.

The two ports implicated in the low entropy measurements are known to be used or exploited by malware. Port 6000 is for XWindow systems and is used by some trojans and port 2967 is an IRC communication port sometimes used to control BotNets. This data

![Average External Port Measurements](image.png)
Table 4.3: Entropy and Median Values for Spikes in Figure 4.7 showing Top 5 Port Values and Counts per 1000 Records - for Entropy Values < 8.0 During an Event of Interest

<table>
<thead>
<tr>
<th>Entropy</th>
<th>Top 5 Median Values and Counts (port:count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.33</td>
<td>(80:378),(12200:231),(445:92),(53:91),(443:30),(0:24)</td>
</tr>
<tr>
<td>4.68</td>
<td>(2967:600),(6000:600),(80:113),(445:59),(53:29),(443:14)</td>
</tr>
<tr>
<td>6.78</td>
<td>(2967:347),(6000:347),(80:177),(53:38),(443:37),(0:18)</td>
</tr>
<tr>
<td>7.84</td>
<td>(22:442),(80:177),(53:67),(443:18),(0:16),(46830:12)</td>
</tr>
<tr>
<td>7.97</td>
<td>(22:404),(80:170),(53:62),(443:33),(46830:15),(0:14)</td>
</tr>
<tr>
<td>7.92</td>
<td>(22:450),(80:149),(53:51),(0:26),(443:19),(46830:16)</td>
</tr>
<tr>
<td>7.63</td>
<td>(22:534),(80:145),(53:40),(443:28),(0:20),(21166:10)</td>
</tr>
<tr>
<td>7.94</td>
<td>(22:330),(80:264),(53:77),(443:29),(0:24),(20784:11)</td>
</tr>
<tr>
<td>7.93</td>
<td>(22:421),(80:168),(53:46),(0:28),(28229:25),(443:20)</td>
</tr>
<tr>
<td>7.94</td>
<td>(9649:405),(80:169),(0:42),(443:33),(53:29),(3074:21)</td>
</tr>
<tr>
<td>7.61</td>
<td>(80:246),(8000:194),(12200:194),(445:58),(0:46),(53:43)</td>
</tr>
<tr>
<td>7.98</td>
<td>(80:275),(3134:202),(27015:64),(53:64),(0:48),(59451:22)</td>
</tr>
<tr>
<td>7.71</td>
<td>(80:309),(2967:146),(6000:146),(53:95),(46830:35),(0:26)</td>
</tr>
<tr>
<td>7.82</td>
<td>(80:306),(16001:191),(0:134),(53:86),(445:46),(443:19)</td>
</tr>
<tr>
<td>7.95</td>
<td>(80:394),(53:136),(0:104),(46830:22),(443:21),(30356:20)</td>
</tr>
<tr>
<td>7.93</td>
<td>(80:454),(0:104),(53:85),(443:25),(1513:19),(45508:14)</td>
</tr>
<tr>
<td>7.98</td>
<td>(80:403),(0:110),(53:76),(443:68),(35690:27),(25:17)</td>
</tr>
<tr>
<td>7.97</td>
<td>(80:419),(0:98),(53:86),(443:40),(445:39),(59451:24)</td>
</tr>
</tbody>
</table>
provide enough information to warrant further investigation by security analysts. The other high median ports indicated by the entropy spikes (80,22) are well known ports for transferring data and may or may not represent legitimate traffic.

### 4.4 Ad Hoc ACL Filtering using Boolean Logic

The Information Security Office at Ohio University filters NetFlow traffic for forensic analysis and network trouble-shooting on a daily basis. Even a simple task such as filtering out a particular subset of IP addresses and ports requires creation of a specific ACL file. The flow application described in Section 3.4.2 makes this task easy enough to be done on an ad hoc basis.
As an example, suppose an analyst wants to know which machines on campus, other than known OIT servers, appear to be acting as web servers based on their port usage. The command shown in Equation 4.3 would be used to filter the NetFlow stream to produce a subset of the data for further analysis. This command relies on internal macros defined in the flow program. In this case, the macro “internal” produces a filter primitive describing the networks on campus, macro “web” produces a filter primitive for commonly used web ports, and macro “oit_servers” contains a list of networks controlled by OIT. Figure 4.8 gives the output from the debugging option (-d) showing the Cisco ACL file produced by the filter parser in this example.

\[
\text{flow d:internal and d:web and not d:oit_servers}
\]  \hspace{1cm} (4.3)

4.5 Identifying Overlying Networks of Communication

Every computer on the Internet is connected by many paths to every other computer on the Internet. This web of connectivity, based on IP addressing, port channels and protocols is known as the “Web”. Many other virtual networks are defined by the actual sharing of information over these physical connections. These overlying patterns include peer to peer (P2P) file-sharing networks, botnet command and control networks and IP-based telephone services. Network engineers and information security analysts are tasked with identifying and understanding these networks in order to protect network resources and enforce information security policies. In many cases, these networks attempt to avoid detection by obscuring their communications.

Given a source of information about botnet command and control channels and servers we are able to automate the creation of macros compatible with the flow
application for filtering traffic and identifying communications with botnets. Figure 4.9 shows the result of the filter specified in Equation 4.4 showing all traffic between the residential hall network and identified botnet controllers.

```
flow -m all.macros resnet and botcc -c | flow-dot > botcc.dot
```

(4.4)

Figure 4.10 is another example of an overlying network of communication identified by simple filtering. This directed graph shows all communications between Ohio
University networked computers and known TOR network routers as created with the command shown in Equation 4.5.

\[
\text{flow -m all.macros tor -c | flow-dot > tor.dot}
\]
This filtering and graphing technique is also helpful to the OIT and the Information Security Office for understanding our own network and patterns of communication for which we are responsible. Figure 1.1 showed all inter residence hall communication for a particular day and revealed a few patterns of communication that were not readily apparent before running the graphing experiment. Of particular interest was the large number of connections centering on few relatively few machines and the highly connected
clique of 6 machines shown in the graph. Closer examination of IP addresses of the dense clusters of communication showed them to be network infrastructure devices performing their normal duty. Re-running the graph without network infrastructure devices gives the less cluttered graph shown in Figure 4.11.

Figure 4.11: One hour sample of internal residence network communication without network equipment.

The remaining cluster of many connections which was revealed by this further filtering was shown to be an OIT network vulnerability scanner performing its normal...
duty. Notice that the 6-tuple clique is still seen after filtering was found to be an overlying network of building control computers which should indeed remain a clique i.e., these computers should not be connecting with the rest of the residential network. One final check on this communication clique verified that these control computers were not being connected to from unknown computers. A filter was created for just these 6 IP addresses and all communication to them was graphed as shown in Figure 4.12. An examination of the machines in this larger clique showed that they are indeed part of the building control system for the University.

Figure 4.12: Directed graph of clique identified in the residential network.
4.6 Analysis Language for Alerting and Automation

Filtering NetFlow data based on ACL files alone is not sufficient for all analysis needs. ACL filter analysis is based on a single NetFlow record at-a-time so there is no capacity for filtering or triggering alerts based on information accumulated from multiple records over time. Responding to changes in the overall state of the traffic flow is a common requirement and a change in percentage use of a particular port is a strong indication of possibly malicious activity.

Another drawback in ACL filtering is the lack of variable analysis within a record. To illustrate this limitation, suppose we want to identify all connections where the source port and destination port are the same, regardless of what the port is; that is not possible with a static ACL file. The ACL file would need a static entry for each possible port. However, this is interesting data since, if both machines are using the same port, then it is not a random occurrence. We consider this traffic to be peer-to-peer (p2p) writ large, as it indicates an a priori agreement between the machines so neither machine can claim the server or client status. This arrangement is not necessarily for file sharing applications – a common meaning for the term p2p. Network Time Protocol (NTP) is an example of a p2p type service; all machines use the same port (123).

To overcome the constraints of ACL filtering we developed additional mechanisms that are capable of analysis over time, or filtering based on comparisons with multiple records as described in Sections 3.4.3 and 3.4.4. Additionally, the flow-map and flow-top utilities are able to accommodate a list type variable or macro since network features such as IP address and port are subject to constant change. List variables or macros can be maintained automatically without affecting the filtering and alerting logic.

Appendix C shows the formal grammar defining an embedded language created for the purpose of expanding NetFlow filtering and alerting capabilities. Some of the
Sample Alerting Rules for flow-map or flow-top

Many connections on X-Windows port per minute
xwindows : tagmap(xwindows)/minute > 1000 || tagmap(xwindows)/minute > 1000

count connections on IRC - Direct Client to Client for BotNet Control
irc-dcc : tagmap(ircdcc)/minute > 100 || tagmap(ircdcc)/minute > 100
	node == connections to a lot of different machines in short time
node : count(daddr)/minute > 1000

generic supernode == could be p2p on an unknown port or skype or gaming
supernode : count(daddr)/minute > 10000

skype == lots of connections, not a lot of bandwidth, port 443
skype : count(daddr)/minute > 1000 && ( dport (443) > 1 || sport (443) > 1 )

Bandwidth rules:

This example rule fires when the total bytes sent and received is greater than 1/50 of the total sent/recvied bandwidth for the entire window where window is set by --window-seconds option

volume : ( sbytes + dbytes )/minute > 50m
bandwidth : ( sbytes + dbytes ) > ( total(sbytes) + total(dbytes) ) / 50

This example rule fires like the rule above but only for bursts of usage greater than the threshold (100) per second
burst : (((sbytes + dbytes)/second) > ((total(sbytes)+total(dbytes))/10) )

more than one tenth of the traffic is IPv6
IPv6 : proto(IPv6) > flows/10

these silly rules test the parser for parentheses, not, add, mult and div etc.
silly1 : (((sbytes/second+((dbytes/second)/(2+1*2/2))))+1/1 > 40k)
silly2 : count(sport)==count(dport) || !(1==1)

Figure 4.13: Sample Alerting Rule Configuration File Showing Expression Parser Capabilities
language capabilities are better shown in the filter rules given in figure 4.13. For example
the rule “xwindows” on line 6 in Figure 4.13 employs a macro defining a certain set of
ports (xwindows), a function to count the occurrences of an identified item (tagmap(…)),
a sliding window calculator (/minute) and comparison and Boolean operators. This
eexample macro will alert whenever the count per minute of connections on any
XWindows port exceeds 1000.

Another basic functionality of the language is associated with the IP and port maps
that are maintained by the application during the analysis. The example on line 14 will
alert when a machine, to be called a “node”, connects to more than 1000 distinct
destination IP addresses in a single minute.

The rules shown in Figure 4.13 were applied to the same data stream which was
graphically analyzed for noisefloor and entropy analysis in Figures 4.5, 4.6 and 4.7. Using
an analysis window size of 60 seconds, the alerting rule set produced selected alert results
from this analysis are shown in Table 4.4 and confirm the analysis supported by
examining the median values in table 4.3. These alerts identify the IP addresses
involved in the incident and the exact time frame. The actual number of alerts that fired
during this event was quite high, over 1200 alerts in approximately 10 minutes, which
indicates that perhaps the rules could use a higher threshold value or a larger window size.
Table 4.4: Sample Alerts Firing During an Event of Interest

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>132.235.100.32</td>
<td>15:22:59</td>
<td>irc-dcc,node, supernode</td>
<td>12398</td>
<td>1123936</td>
<td>0</td>
</tr>
<tr>
<td>64.247.122.179</td>
<td>15:23:13</td>
<td>irc-dcc, volume</td>
<td>1907</td>
<td>14124205</td>
<td>52364216</td>
</tr>
<tr>
<td>218.60.132.111</td>
<td>15:23:22</td>
<td>xwindows,irc-dcc,node</td>
<td>1186</td>
<td>71160</td>
<td>3900</td>
</tr>
<tr>
<td>64.247.125.42</td>
<td>15:24:52</td>
<td>irc-dcc, volume</td>
<td>660</td>
<td>2710602</td>
<td>51837236</td>
</tr>
<tr>
<td>64.247.85.38</td>
<td>15:25:05</td>
<td>irc-dcc, volume</td>
<td>838</td>
<td>52998452</td>
<td>99190783</td>
</tr>
<tr>
<td>132.235.86.47</td>
<td>15:25:14</td>
<td>irc-dcc, volume</td>
<td>1199</td>
<td>13227935</td>
<td>54515072</td>
</tr>
<tr>
<td>132.235.13.51</td>
<td>15:25:37</td>
<td>irc-dcc,node, skype</td>
<td>2168</td>
<td>490490</td>
<td>457707</td>
</tr>
<tr>
<td>132.235.145.106</td>
<td>15:25:58</td>
<td>irc-dcc,node, skype</td>
<td>2994</td>
<td>792520</td>
<td>755392</td>
</tr>
<tr>
<td>64.247.116.149</td>
<td>15:25:58</td>
<td>irc-dcc,node, skype</td>
<td>3590</td>
<td>763741</td>
<td>673730</td>
</tr>
</tbody>
</table>
5 CONCLUSIONS

5.1 Netflow Data is Not Sufficient for Attribution of Traffic

Much of this research was done to support the Information Security Office at Ohio University, which is often tasked with attributing network traffic to a specific individual or computer as part of an official investigation or network troubleshooting. However, NetFlow records alone do not contain enough information to identify a computer or person using the computer.

The Internet Protocol (IP) address field is used to locate a source or destination computer on an IP network but it should not be considered the “identity” of the computer. The true source or destination of a communication may change over time, or a single true source may be represented by multiple IP addresses. The identity of a computer is held by an underlying network protocol; in most cases this is an Ethernet network that identifies a computer by a Media Access Control (MAC) address. The MAC address is not typically available in NetFlow records. The management of IP addresses and the actual identity of computers is in the domain of the network routers, switches, name servers (DNS) and dynamic host (DHCP) servers.

This lack of accountability in network traffic data is particularly acute when dealing with archival data. Any attempt to attribute historical data to a particular person or machine – a task often undertaken by forensic or criminal investigators – requires information about the state of the network at a specific time. This network state is continuously changing as network devices and servers are reconfigured and computers come and go from the network. Keeping this kind of state information is possible for a small network completely within the control of the data collectors; however, it is extremely difficult when examining data encompassing the entire Internet.
It is easy to forget all that is involved in the state of the network. Often when examining older log files, an analyst will look up the host name of an IP address, or check whether an IP address is in a particular watchlist, however the information from queries such as these may not be valid since the state of the name servers and watchlist may have changed since the logs were recorded. It is important that servers and watchlists be updated regularly in order to examine current data, but copies of historical watchlists or server settings are not normally available.

This restriction on analyzing log files applies to the macros and watchlists employed in this research. For processing large amounts of historical data or for scientific experiments that must be reproducible, it is vital that the results of look-ups be stored with the original record because doing a look-up may require too much time or the state of the network may have been lost.

The methods and tools described in this research are necessary for effective ad hoc analysis by information security analysts and network engineers; however, this usefulness is limited by a lack of attribution of traffic on the network. A system of network registration or network access control is needed to be fully capable of responding to alerts triggered by IP address. Often, an event is detected based on IP address that requires some attention, but action cannot be taken in a timely manner because we lack information about where the machine really is.

5.2 NetFlow Data Can be Analyzed in Near-Real Time

A great deal of the effort associated with the Russ College of Engineering research into NetFlow analysis was a result of the problems with time stamps in NetFlow records as discussed in Section 4.1. The inability to control the record durations within the data set lead to complications described in Sections 2.4.1 and 2.1.2. Although much effort went into making the tools perform adequately for any time window, for best results, care
must be taken to choose a time window size that corresponds to the timeframes used in the filtering language.

An analysis of traffic at a single second in real time may not be accurate since there may be information about that second that will not be written to the log file until the collector closes a record at some time in the future. Additionally, the number of records written per time period is not consistent – at times of greater activity there are more records to process – we must write programs that interpret timestamps, aggregate traffic measurements and keep them in memory until we are confident that we have read all the information for that time-frame. The problem with timestamps is a recurring theme when doing statistical analysis and implementing automated tools for this research. We address several methods for dealing with this difficulty in Chapter 3.

The nature of time stamps in NetFlow records also necessitated preprocessing the research data for placement into arbitrary sized time bins. This was done to allow for accurate statistical analysis but also to reduce processing time for subsequent analyses. Because of the extra processing required, the best that could be achieved was near-real-time processing as defined in Section 2.3.1, but there is potential for better performance given complete control over the original data source.

5.3 IP Packet Capture is Better for Precise Timing Analysis

Any additional research focusing on the precise timing of events would benefit from a direct analysis of the underlying packet stream and would be more effective because it would not suffer from the inconsistencies in the timing of NetFlow records. A packet capture program using LibPcap would produce data with precise timing. The IP packet header is easily read directly into a C++ structure and an application could store and process only the information that was needed. Subsequent processing would have much less data to deal with and would be freed to perform calculations more efficiently.
5.4 Average Port May be Similar to Entropy in Usefulness

This thesis has examined more closely a notion proposed by previous research, that a calculation of average port is useful as a measure of network state change. Previous research verified the viability of this idea using statistical and graphical analysis. We have substantiated that research by looking more closely at patterns of port usage as described in Section 4.2. Furthermore, we have shown practical results in identifying network anomalies based on changes in average port as well as entropy. These results are discussed in Section 4.3. We have corroborated the anomaly detection with practical tools using an embedded filter language and presented results in Section 4.6.

Other measures of port were found to be unsuitable for statistical analysis. However, measurements of port usage such as “peer factor” as discussed in Section 4.2 have much to tell us about behavior on the network. In fact, we can imagine any number of non-random measurements that contain useful information; any specific combination of source- or destination-port, byte count or protocol may indicate an event of interest.

These non-Gaussian events must be examined using non-statistical methods such as the linguistic approach described in Chapter 3 where we employ a domain-specific embedded language (DSL) to identify these features. The use of alerting rules, defined by this language, appears to be useful and has been used to verify test results displayed in Figures 4.5 and 4.6.

5.5 A Priori Knowledge Helps to Improve Network Analysis Tools

Much of the research discussed in Chapter 2 is dedicated to identifying anomalous activity without specific knowledge of that behavior; however most of the practical work in this field is dependent on using information obtained from the information security community in the form of vulnerability alerts and watchlists of problematic systems. Despite the significant progress in network analysis based on statistical theory, linguistic
techniques, and practical inference - it is still helpful to have substantive information from
reliable sources. The usefulness of this information is increased by using the knowledge to
improve the analysis methods described in Chapter 2.

Tools capable of full packet inspection and analysis, such as Procera’s PacketLogic,
can be used to verify an inference which was based on a traffic analysis. Furthermore,
these devices can identify classes of devices to be added to the watchlists or contributing
back to the information security community.

Noisefloor measurements are Useful but have Limitations A single measurement of
noisefloor across all devices on the network was found to have limited usefulness when
applied to unfiltered network traffic. Noise floor as described in Sections 3.1 and 3.1.1 has
more significance when measured against a specific set of machines that are expected to
have similar behavior. Machines dedicated to a particular service should display limited or
no activity indicating other services. An email server for instance, should have a
noisefloor threshold of zero for all Web traffic.

There is also some difficulty in determining when a significant change in noisefloor
represents an event of interest to analysts. A dramatic change in traffic to financial systems
may be normal and benign at the start of a business day but might indicate more nefarious
activity when happening after hours or on the weekend.

5.6 Methods Described in this Research are Useful for Ad Hoc Data Analysis

The Information Security Office at Ohio University filters NetFlow traffic for forensic
analysis and network trouble-shooting on a daily basis. Even a simple task such as
filtering out a particular subset of IP addresses and ports requires creation of a specific
ACL file. The more difficult work of maintaining watchlists and building ACL files of
several thousand lines is not feasible with unduly manual processes.
The practical methods described in Section 4.4 and 4.5 and the flow filtering application described in Section 3.4.2 make onerous NetFlow analysis tasks easy enough to be done on a routine ad hoc basis. These tools have proven useful in helping security analysts and network engineers understand complex patterns of communications on the networks for which they are responsible. Section 4.5 illustrates the discovery of an overlying pattern of communication among the University’s building control systems, which once identified can be watched carefully for anomalous activity.

ACL filtering of NetFlow data is very powerful but it lacks the flexibility necessary for statistical analysis or for evaluating the state of the network over time. The analysis tools described in Chapter 3 and summarized in Section 4.6 were developed to meet this need and have been shown to be cable of explaining network anomalies. Network anomalies such as those illustrated in Figures 4.5 and 4.6 have been explained using a simple set of these rules as shown in Figure 4.13 with the results shown in 4.4.
6 Future Work

6.1 Language Improvements

The filtering language implemented in the Flow application should be modified to include IP and port combinations as this information is often available from the security community watchlists. With this change, it would be easier to avoid false positive identifications as the addition of port would more-closely identify suspect communication.

The Flow-Map and Flow-Top applications can be improved in several ways. The alerting language should be expanded to include a weighting system, specific rules could be marked as more or less important with a numeric value, and then a threshold based on frequency times weight could be applied to the firing of rules. This would help limit the large number of rules that fire in the event of a detected anomaly.

6.2 Alerts Logging to the SEIM, Nessus Scanner, or Footprints

Development of automated responses to network alerts is an on-going process. A separate flow-action utility might be useful in brokering response activities to be handled by the SEIM or other systems. Although these tools developed thus far have been shown to be useful for ad hoc queries and investigations, there is much work to be done in automating these systems to work together in an automatic mode.

6.3 The use of Filter Watchlists can be expanded

More community based watch lists can be automated for inclusion into macros. Existing watchlist functions should be expanded to include a port designation as well as IP address as this information is almost always available. This feature would help to avoid false-positives associated with legitimate communication with suspected systems.

Another potential source of watchlist information is the Procera PacketLogic device currently in place on Ohio University’s network. This device functions in real time mode.
without maintaining any significant history, but it could be used to make a snapshot of machines engaged in a particular behavior, based on inspection of packets. The network traffic tools could then use this snapshot or macro to study the patterns of communication with that subset of the network.
REFERENCES


SIGCOMM conference on Internet measurement, pages 1–6, New York, NY, USA. ACM.


APPENDIX A: A SMART POINTER TEMPLATE CLASS FOR C++

A.1SmartPointer.h

```c++
#ifndef _CONLEY_THESIS_SMARTPOINTER_H_
#define _CONLEY_THESIS_SMARTPOINTER_H_
//*******************************************************************
// Class: CSmartPointer<class T> template code
// The original class written by myself circa 1997, included multithreading functionality, which I removed for clarity and brevity in CS361 in 2004.
// Author: Thomas Conley
// Description: This class is an encapsulation of a pointer. It is a reference counted auto_ptr-like class similar to counted_ptr, which didn't make the ISO/IEC 14882 C++ standard in 1998.
// It keeps an external counter on an external pointer and deletes the memory when no more pointers are referring to it.
// It's basically a safe way to use pointers and it prevents the need for coping actual objects (copy pointers instead) which can drastically improves the actual-time of programs.
// Time Complexity: All operations are constant time for asymptotic complexity. O(1).
// Space Complexity: The space complexity is also constant, O(1), since memory is only allocated once per instance created. In-fact the memory is static so, multiple instances of the class use the same memory for counters. There is memory associated with the actual objects that the pointer points to, but that is not part of this class. Again, this is just a wrapper around a pointer.
//*******************************************************************
template <class T> class CSmartPointer {
public:
//*******************************************************************
// default constructor
//*******************************************************************
explicit CSmartPointer (T* p = NULL);
//*******************************************************************
// copy constructor sets info from lhs and ups the count
//*******************************************************************
CSmartPointer(const CSmartPointer<T>& p);
//*******************************************************************
// destructor does nothing but call dispose() which decreases the reference count and only deletes the actual object if the count reached 0
//*******************************************************************
~CSmartPointer();

};
```
// the member-access operators

const T* operator->() const;
T* operator->(void);

//*******************************************************************

// assignment operator just calls dispose to decrement (and delete if
// necessary) then the information is updated

CSmartPointer<T>& operator=(const CSmartPointer<T>& p);

//*******************************************************************

// Equality and inequality operators

bool operator==(const CSmartPointer<T>& p) const;
bool operator!=(const CSmartPointer<T>& p) const;
//bool operator < (const CSmartPointer<T>& p) const;

//*******************************************************************

// The dereference operator is const since not pointers referred to
// by this smart pointer may be modified except by this class

const T* operator*() const;

//*******************************************************************

// The "bool" operator

operator bool() const;

//*******************************************************************

// Is the pointer Null

const bool IsNull() const;

//*******************************************************************

// const access to the embedded pointer

const T* get() const;

//*******************************************************************

// deleting a smart pointer has no meaning, but someone may
// try if they don’t understand

void operator delete(void *);

protected:

//*******************************************************************

// dispose does most of the work, decrement the external counter and
// delete when we reach zero

inline void dispose();

private:

T* ptr; // shared pointer to the value
long* count; // shared number of owners

//*******************************************************************

// Increment and Decrement the external counters

inline long Increment();
inline long Decrement();

}
template <class T>
std::ostream& operator<< (std::ostream&o, CSmartPointer<T> p)
{
  o << (*p.get());
  return o;
}

#include "SmartPointer.cc"
#endif // _CONLEY_THESIS_SMARTPOINTER_H__

A.2  SmartPointer.cc

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

 maman((class T>
CSmartPointer<T>::CSmartPointer (T* p /*= NULL*/ )
  : ptr(p), count(new long(1))
{
}

CSmartPointer<T>::CSmartPointer(const CSmartPointer<T>& p)
{
  assert( p.ptr );
  assert( p.count );

  ptr = p.ptr;
  count = p.count;
  Increment();
  assert( ptr );
}

CSmartPointer<T>::~CSmartPointer()
{
 dispose();
}

// the member-access operator const
// Time and space complexity is O(1)
template <class T>
const T* CSmartPointer<T>::operator->() const
{
    assert(ptr);
    return ptr;
}

template <class T>
T* CSmartPointer<T>::operator->(void)
{
    assert(ptr);
    return ptr;
}

CSmartPointer<T>& CSmartPointer<T>::operator=(const CSmartPointer<T>& p)
{
    if (this != &p)
    {
        dispose();
        ptr = p.ptr;
        count = p.count;
        Increment();
    }
    return *this;
}

bool CSmartPointer<T>::operator==(const CSmartPointer<T>& p) const
{
    // Smart pointers are considered equal if they point
    // to the same thing
    if (ptr == p.ptr )
    {
        return true;
    }
    return false;
}

bool CSmartPointer<T>::operator!=(const CSmartPointer<T>& p) const
{
    return !operator==(p);
}
The dereference operator is const since pointers referred to by this smart pointer may be modified except by this class. Time and space complexity is O(1).

```cpp
template <class T>
const T* CSmartPointer<T>::operator*() const
{
    assert(ptr);
    return ptr;
}
```

The "bool" operator. Time and space complexity is O(1).

```cpp
template <class T>
CSmartPointer<T>::operator bool() const
{
    return (ptr==NULL ? false : true);
}
```

Is the pointer NULL. Time and space complexity is O(1).

```cpp
template <class T>
const bool CSmartPointer<T>::IsNull() const
{
    return ptr==NULL;
}
```

const access to the embedded pointer. Time and space complexity is O(1).

```cpp
template <class T>
const T* CSmartPointer<T>::get() const
{
    return ptr;
}
```

deleting a smart pointer has no meaning, but someone may try if they don’t understand. Time and space complexity is O(1).

```cpp
template <class T>
void CSmartPointer<T>::operator delete(void *)
{
    assert(false);
}
```

dispose does most of the work, decrement the external counter and delete when we reach zero. Time and space complexity is O(1).

```cpp
template <class T>
void CSmartPointer<T>::dispose()
{
    Decrement();
    assert( *count > -1 ); // should never be negative
}
```
if (*count <= 0) // if no more references
    { // to the code then
        delete count; // the memory can be
        delete ptr; // released
        count = NULL;
        ptr = NULL;
    }
}

//*******************************************************************
// Increment the external counter
// Time and space complexity is O(1)
//*******************************************************************
template <class T>
long CSmartPointer<T>::Increment() { return (*count)++; }

//*******************************************************************
// Decrement the external counters
// Time and space complexity is O(1)
//*******************************************************************
template <class T>
long CSmartPointer<T>::Decrement() { return (*count)--; }
APPENDIX B: EMBEDDED LANGUAGE GRAMMAR FOR ACL

CREATION

package acl;
{
use Parse::RecDescent;

$::RD_HINT = 1;

sub WriteFile
{
my @argv = @_;
return '' unless @argv;

my $grammar = q{

start : expr eos { $item[1] }

expr : conjunction { $item[1] } |
| factor { $item[1] }

conjunction : factor /and/ expr { filter::And($item[1],$item[3]) }

disjunction : factor /or/ expr { filter::Or ($item[1],$item[3]) }

| factor expr { filter::Or ($item[1],$item[2]) }

factor : /not/ factor { $item[2]->Negate() } |
| /and/ item { $item[2] } # ignore starting and |
| /or/ item { $item[2] } # ignore starting or |
| item { $item[1] } |
| '(' expr ')' { $item[2] }

item : /not/ item { $item[2]->Negate() }

| IPCIDR { $item[1] } |
| PROTO { $item[1] } |
| DNSNAME { $item[1] } |
| PORT { $item[1] } |
| MACRO { $item[1] }

# support cider (/slash) notation of IP addresses

IPCIDR : /\(s:\|src:\|d:\|dst:\)?\d+\.\d+\.\d+\.\d+/.?\d{1,2}/?/io

[ new filter('cidr',$item[1]) ]

# common protocol tags or "proto:number"

PROTO : /\(tcp|udp|icmp\)\(?:\ proto\)?\d+\)/io

[ new filter('ip-protocol',$item[1]) ]

# reasonable length words and dots are DNS names

DNSNAME : /\(s:\|src:\|d:\|dst:\)?\w\[2,30\]\.\w\[1,10\]\.\w\?/o

[ new filter('dns',$item[1]) ]

# lone numbers are ports

PORT : /\(s:\|src:\|d:\|dst:\)?\(?:\d+\)/io

[ new filter('ip-port',$item[1]) ]

# any other word could be a macro definition

MACRO : /\(s:\|src:\|d:\|dst:\)?\w\[a-z\]\[a-z:\_0-9\]*/oi

[ new filter('macro',$item[1]) ]

eos : /\Z/

};
use File::Temp;

my $parser = Parse::RecDescent->new($grammar);
{
    my $str = join(' ', @ARGV);
    if ( $str )
    {
        $str =~ s/\s+//;
        my $result = $parser->start( $str );
        if ( $result )
        {
            my $res = $result->String();
            my $string = "# source expression -> $str\n" .
                "# canonical expression -> $res\n" .
                "\n";

            my @primatives = $result->Primitives();
            foreach my $p (@primatives)
            {
                if ( $p )
                {
                    $string .= join("\n", @{$p});
                }
            }

            my @blocks = $result->Definition_blocks();
            my $max_index = scalar(@blocks)-1;

            $string .= "filter -definition default\n";
            for my $b (0..$max_index)
            {
                foreach my $i (@{$blocks[$b]})
                {
                    $string .= " $i\n";
                }
                $string .= " or\n" unless $b >= $max_index;
            }

            my $tmpfile = File::Temp->new(
                DIR => '/usr/local/isobin/acls',
                SUFFIX => '.acl',
                UNLINK => 0,
            );
            print $tmpfile $string;
            return $tmpfile;
        }
    }
    return '';
APPENDIX C: EMBEDDED LANGUAGE GRAMMAR FOR FLOW

ALERTING

```c
108

1 top = boolean [ top.val = bind(&Grammar::done)(self, arg1) ];
2
3 boolean = ( // a boolean is a predicate
4   predicate [boolean.val=arg1]
5   // followed by zero or more boolean operators and other predicates
6   >> *
7   ( "&amp;&amp;" >> predicate [boolean.val=BIND(And, boolean.val, arg1)]
8     | ( "||" >> predicate [boolean.val=BIND(Or, boolean.val, arg1)]
9     )
10 )
11 predicate = // predicate is a comparison that evaluates to true or false
12 ( expr [ predicate.left = arg1 ]
13   >> *
14     ( 
15       ( 
16         ( 
17           ( > >> expr [predicate.val=BIND(gt, predicate.left, arg1)]
18         | ( &lt; >> expr [predicate.val=BIND(lt, predicate.left, arg1)]
19         | ( == >> expr [predicate.val=BIND(eq, predicate.left, arg1)]
20     )
21     )
22     | // this is a specific predicate that checks the tag on the source
23     // address and returns true if it matches the passed parameter
24     ( str_p("iptag")
25       >> 
26       identifier
27       [predicate.val = BIND(iptag, arg1)]
28       >> 
29     )
30   | // allow "NOT" of a predicate... why not?
31   ( 
32     ( ! >> predicate [predicate.val = !arg1] )
33   )
34   | // allow parentheses around predicates and booleans
35   ( 
36     ( ( >> predicate [predicate.val = arg1] )
37     | ( ) >> boolean [predicate.val = arg1] )
38 )
39   )
40 expr = // expression is a term
41 term [expr.val=arg1]
42   // plus or minus another term (optional)
43   >> *
44     ( 
45       ( + >> term [expr.val += arg1] )
46     | ( - >> term [expr.val -= arg1] )
47 )
48 );
49 term = // term is a factor
50 factor [term.val=arg1]
51   // multiplied or divided by another factor (optional)
52   >> *
53     ( 
54       ( * >> factor[ term.val *= arg1 ] )
55     | ( / >> factor[ term.val /= arg1 ] )
56 )
57 factor = // factor is a value
58 value [factor.val=arg1]
59   // or a number
```
value = (  // value is a measurement that comes from a Flow object
    str_p("count") >> '(' >> str_p("dport") >> ')') [value.val = BIND(dports)]
    | (str_p("total") >> '(' >> str_p("dport") >> ')') [value.val = BIND(total_dports)]
    | (str_p("total") >> '(' >> str_p("sport") >> ')') [value.val = BIND(total_sports)]
    | (str_p("total") >> '(' >> str_p("daddr") >> ')') [value.val = BIND(total_daddrs)]
    | (str_p("total") >> '(' >> str_p("proto") >> ')') [value.val = BIND(total_protos)]
    | (str_p("count") >> '(' >> str_p("flows") >> ')') [value.val = BIND(flows)]
    | (str_p("total") >> '(' >> str_p("flows") >> ')') [value.val = BIND(total_flows)]
    | (str_p("total") >> '(' >> str_p("sbytes") >> ')') [value.val = BIND(total_sbytes)]
    | (str_p("total") >> '(' >> str_p("dbytes") >> ')') [value.val = BIND(total_dbytes)]
    | (str_p("total") >> '(' >> str_p("evil") >> ')') [value.val = BIND(total_evil)]
    | (str_p("flows") >> '(' >> uint_p [value.val = BIND(dport, arg1)] >> ')')
    | (str_p("sbytes") >> '(' >> uint_p [value.val = BIND(sport, arg1)] >> ')')
    | (str_p("dbytes") >> '(' >> uint_p [value.val = BIND(daddr, arg1)] >> ')')
    | (str_p("proto") >> '(' >> prot [value.val = BIND(proto, arg1)] >> ')')
    | (str_p("tagmap") >> '(' >> identifier [value.val = BIND(tagmap, arg1)] >> ')')
);
APPENDIX D: SOURCE LISTING FOR FLOW-DOT UTILITY

```perl
#!/usr/bin/perl -w
use strict;
use Socket;

# open the input pipe of netflow output from flow-cat
# and convert to expected fields using flow-export
open STDIN, " flow-export -f2 -mSRCADDR,DSTADDR |" or die $!

my $reIP = qr{\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}}oi;
my (%nodes,%links);

while(<>)
{
  last if($done);
  if ( /(\$reIP),($reIP)$/ )
  {
    my ($n1,$n2) = ($1,$2);
    foreach my $n (($n1,$n2))
    {
      $nodes{NodeSpec($n)}++;
    }
    $links{"\"$n1\"->\"$n2\";\n"}++;
  }

  # print the dot file based on the collected data
  print "digraph G { node [shape=none]\n";
  print keys %nodes;
  print keys %links;
  print "}\n";
  exit(0); # end of main

  # ------------------------------------------------------------------ #
  sub NodeSpec
  {
    my $n = shift or die;
    my $attributes = '[fontcolor=gray, shape=none]';
    if($n="/(?:64\.247\.|132\.235\.|\d+)\./")
    {
      return NodeAttr($n);
    }
    return "\"$n\" $attributes;\n";
  }

  sub NodeAttr
  {
    my $ip = shift or die;
    if ( $ip =~ m/(?::0?64\.247\.|\d+)\.|\d+)\./o )
    {
      my $p = $1 ? $1 : $2;
      if ( $p < 64 )
      {
        return "\"$ip\" [fontcolor=chartreuse, shape=none];\n";
      }
      # 64.247.64.0/22 South Green 1 (Ewing-Fenzel-Adams-Nelson-Wray)
      # 64.247.68.0/22 South Green 2 (Crawford-Dougan-True-Weld)
      # 64.247.72.0/22 South Green 3 (Mackinnon-Smith-Obleness-Armb-Smith)
      # 64.247.76.0/22 South Green 4 (Atkinson-Brown-Martzolff)
      # 64.247.80.0/22 South Green 5 (Brough-Cady-Foster-Pickering)
  }
```

elsif($p>=64&&$p<=83)
{
    return "\"$ip\" [fontcolor=blue, shape=none];\n";
}
# 64.247.84.0/22 East Green 1 (Scott-Shively-tiffin-Wolfe-Ullom-Zoology)
# 64.247.88.0/22 East Green 2 (Gamertsfelder-Perkins)
# 64.247.92.0/22 East Green 3 (Biddle-Washington-Bush)
# 64.247.96.0/22 East Green 4 (Jefferson-Johnson-Read-Kantner-Seigfried)
# 64.247.100.0/22 East Green 5 (Bryan-Lincoln-Voight-PhiGammaDelta)
elsif($p>=84&&$p<=103)
{
    return "\"$ip\" [fontcolor=green, shape=none];\n";
}
# 64.247.104.0/22 West Green 1 (Boyd-Treudley-Ryors)
# 64.247.108.0/22 West Green 2 (Sargent-WilsonWest)
# 64.247.112.0/22 West Green 3 (Convo-James)
elsif($p>=104&&$p<=115)
{
    return "\"$ip\" [fontcolor=red, shape=none];\n";
}
# 64.247.116.0/22 Bromley Hall
# 64.247.120.0/23 AlphaDeltaPi - AlphaXiDelta - DeltaZeta - SigmaKappa
# 64.247.122.0/23 University Courtyard
# 64.247.124.0/23 Mill Street Apartments
# 64.247.126.0/23 Hoover Hall
elsif($p>=116&&$p<=127)
{
    return "\"$ip\" [fontcolor=orange, shape=none];\n";
}
# others within range
elsif($p>=127&&$p<=170)
{
    return "\"$ip\" [fontcolor=darkgreen, shape=none];\n";
}
# default
elsif($p>=170)
{
    return "\"$ip\" [fontcolor=darkviolet, shape=none];\n";
}
return "\"$ip\" [shape=none];\n";
}
sub Host
{
    my $ip = shift or die;
    my $iaddr = inet_aton($ip);
    my $name = gethostbyaddr($iaddr, AF_INET);
    return $name;
}
sub NodeAttr
{
    my $ip = shift or die;
    my $iaddr = inet_aton($ip);
    my $name = gethostbyaddr($iaddr, AF_INET);
    if($name)
    {
        if($name=~/\e\g\d\./o) { return "\"$ip\"[fontcolor=blue, shape=none];\n";
        elsif($name=~/\w\g\d\./o) { return "\"$ip\"[fontcolor=green, shape=none];\n";
        elsif($name=~/\n\g\d\./o) { return "\"$ip\"[fontcolor=red, shape=none];\n";
        elsif($name=~/\s\g\d\./o) { return "\"$ip\"[fontcolor=orange, shape=none];\n";
        };
        return "\"$ip\" [shape=none];\n";
    }