Calling All Nodes:
Classifying Skype Overlay Control Flows

by

Carl Brett Meyer

(Under the direction of Kang Li)

Abstract

Peer-to-Peer botnets are of particular concern in the world of network security because of the difficulty involved in identifying the botmaster node in the network. This paper seeks to address this issue incrementally by developing a statistical model for the control, or signaling, network flows of the most popular P2P VoIP application, Skype, as a first step toward identifying known P2P applications for the purposes of whitelisting them in a network trace. Through construction of a dataset containing real-world Skype traces and real-world traces for four other popular P2P file-sharing programs, a statistical model is created which incorportates the flow behaviors of Skype control flows. This statistical model is tested using four classification algorithms, and the results show a very high accuracy and low false positive rate for successfully identifying Skype control flows against the control flows of the other P2P applications.

Index words: Skype, supervised learning, machine learning, network security, botnet

Calling All Nodes:
Classifying Skype Overlay Control Flows

by

Carl Brett Meyer

B.A., University of Georgia, 2002

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2011

©2011

Carl Brett Meyer

Calling All Nodes:

Classifying Skype Overlay Control Flows

by

Carl Brett Meyer

Approved:

Major Professor: Kang Li

Committee: Roberto Perdisci
Khaled Rasheed

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2011

Acknowledgments

This thesis is dedicated to my loving wife Vanessa, who has been supportive and understanding of my time and effort to complete this work. I would also like to thank Dr. Li, Dr. Perdisci, and Dr. Rasheed for their guidance and inspiration with this project.

Contents

Chapter 1

Introduction

A botnet is a collection of compromised machines that act under the remotely controlled influence of an attacker. The compromised hosts, or "bots", are controlled through the use of a command and control (C&C) channel by an attacker known as a "botmaster". Botnets are commonly used for such purposes as sending spam, Distributed Denial of Service (DDoS) attacks, click fraud, and identify theft.

Botnets have been a source of consternation to network security professionals for several years now, and like any malicious application, they have evolved to overcome previous successful attempts to subvert and control them. While botnets originally used a centralized architecture wherein a C&C server would communicate commands over an IRC channel, the most current permutation perplexing security researchers is the use of a Peer-to-Peer architecture to distribute the location of the botmaster node, making the controlling unit of the botnet much harder to locate and shut down. While other types of network intrusion detection use signature-based mechanisms or blacklists, creating these types of structures for P2P bots is extremely difficult due to the proliferation of valid P2P programs, and because many P2P applications use encryption or obfuscation, both well-known to be used in bot networks as well.

Early P2P botnet research involved observation of particularly prevalent botnets which employ a P2P architecture in order to identify particular unique features of the behaviors of these botnets so that they could later be identified. In [22], Grizzard et al examine the behavior of the *Trojan.Peacomm* botnet, and Holz et al examine a permutation of this botnet known as *Storm* in [26]. Both of these works are interested in outlining the distinguishing features of the system-level and network-level behaviors of the botnet, and they offer an insight into the statistical features that can be used to detect these types of bots.

More recently, P2P botnet research has focused on identification and classification based mostly on clustering algorithms to group flows and classify botnet traffic based on its flow characteristics. In [23], Gu et al use what they term "A-plane" clustering and "C-plane" clustering to cluster the actions and communication patterns of botnet traffic. After these initial clustering steps, they employ "Cross-plane correlation" which is used to create a more general classification scheme for identifying bots within a botnet. In [43], the authors use clustering of the communication behaviors of P2P botnets to identify them against legitimate P2P traffic based on the premise that P2P bots must be present for longer durations than regular P2P application usage in order to maintain the malicious functionality of the botnet, and to issue and search for commands.

Current research on P2P botnets also includes projections for future growth and mutation of the P2P botnet phenomenon. One such example is the work done in [34] by Nappa et al, where the authors describe the potential construction of a botnet on top of the Skype network. By employing Skype's native overlay network and encryption mechanisms through the use of Skype development packages, the authors are able to outline the measures by which a botnet could be embedded within the Skype overlay network in order to prevent detection. The authors emphasize that these types of "parasitic overlay" botnets are likely to be found in the wild soon due to their resilience to detection.

As Haq, et al obviates in [25], the rise in popularity of P2P applications in the past several years has fostered concern over detecting valid P2P programs since there has been a corresponding rise in malware which employs this same overlay network technique, most considerably botnets. Therefore, this work is concerned with detecting valid P2P programs as a first measure towards disallowing botnet traffic by identifying and only allowing non-malicious applications in firewall infrastructures such as Intrusion Prevention Systems (IPS). However, since there are a plethora of valid P2P-based applications to choose from, we limit our scope to detecting one application, in the hopes that the procedures used in this work will aid in detecting others.

The application chosen for examination in this work is the popular VoIP application Skype. Skype was chosen for this work because unlike many popular P2P file-sharing programs, Skype does not have any great conflict with Internet Service Providers (ISP) the way that the file-sharing programs do. This is because typically ISPs are unhappy about bandwidth usage incurred by most P2P file-sharing programs, and due to an overwhelming number of copyright infringement issues that have arisen in previous years, ISPs are not enthusiastic about the use of P2P file-sharing programs in their networks. Skype, on the other hand, is an application that generates innocuous VoIP and instant message traffic, and does not use a great deal of bandwidth to do so. Therefore, Skype seemed a very appropriate choice for use with this work, since a reliable identification method for Skype would satisfy the objective to identify a P2P application without bringing into question issues of copyright law and bandwidth usage.

In this paper, we seek to create a statistical signature for the Skype overlay control protocol that can be used to whitelist the Skype application in a network trace. Created in 2003 by the engineers behind KaZaA, Skype is the most popular VoIP application worldwide, and it uses a distributed, P2P architecture to achieve high throughput and reliability for voice, video, and instant message data. However, Skype uses a proprietary encryption scheme that

is enormously difficult reverse engineer (there have not been any fully successful attempts so far), and is thus very hard to classify, making it extremely difficult to create a whitelist signature for use in an intrusion detection system. In order to overcome this obstacle, this paper focuses on the control or signaling protocol used by Skype to maintain its P2P overlay network. Since the nodes in the Skype network must communicate regularly in order to advertise that they still exist, the regularity of these transactions can be used to determine what statistical features are exhibited by these keep-alive messages. In addition, Skype is configured by default to start up when a client starts their machine, and therefore Skype can be left running in the background, possibly without the client's knowledge, while still transmitting overlay control protocol but not voice, video, or instant message data.

Skype has two main classifications of nodes in its overlay network, supernodes and regular nodes. Supernodes are machines that contain powerful system resources, high bandwidth, and are not blocked by a NAT or firewall. The supernodes also form the actual P2P layer of the Skype topology, since they are organized to facilitate routing between regular nodes, and they maintain the structure of the network with regards to themselves and the regular nodes which communicate with them. Therefore, the purpose of using these supernodes is to use their generous resources to perform maintenance in the overlay network and to reroute data traffic between nodes to improve the throughput for data streams. This is how Skype is able to guarantee a high transfer rate with little delay and also provide reliability for their service. However, this presents another challenge in the case of observing the control protocol, namely that if an observed node is a supernode, it likely is transmitting data content from other nodes as well as control traffic. In lieu of this obstacle, this paper addresses the issue of examining only the overlay control protocol by observing that the aforementioned keep-alive messages should have a very regular delay interval over a long period of time and a fairly regular distribution for the size of the packets transmitted, whereas data traffic should have

"bursts" or periods of clustered packets followed by long periods of inactivity and variable sized packets.

To facilitate observation of Skype control flows in action, a dataset containing network traces of the Skype application was created that mimicked real-world Skype user behavior. In addition, a dataset containing network traces of several P2P file-sharing applications was also created, since not only was the intention to identify Skype control flows in a network trace, but also to ensure that they were not similar enough to the control flows of other P2P applications to create false positives given an unknown network trace. Therefore, the P2P file-sharing applications were used to ensure that the stastical model created for the Skype control flows was particular to Skype, and not statistically similar to other P2P control flows.

Using the regularity of the keep-alive messages in the Skype network, we were able to identify many IPs which had persistent communication and also had very regular delay intervals and packet distributions, allowing for the creation of a statistical model for these control messages. This statistical model was used to create a dataset used for classifying Skype control flows with several different classification algorithms: Decision Trees, Naïve Bayes, k-Nearest Neighbor, and Support Vector Machines. Using these classifiers, we were able to show high accuracy and low false positive rates when classifying Skype control flows against the control flows of the other P2P applications in the dataset.

The contributions of this work are the two datasets containing network traces of two experimental testbeds configured to perform as real-world P2P clients, one using Skype, and one using four popular P2P file-sharing applications, and a statistical model for identifying the Skype control flows in a network trace. In addition, the datasets used during classification can be used for further classifcation schemes.

Chapter 2

Background and Related Work

2.1 Port-based Classification

The most basic form of classification for a network application is the identification of well-known port numbers assigned by or registered with the Internet Assigned Numbers Authority (IANA) [1]. The port numbers can then be examined to determine if a particular application which employs a standard port number is running on a host machine. However, according to Karagiannis, et al in [28], while originally port-based classification was possible in early versions of P2P programs, most P2P programs now either configure a random port upon initial installation, or allow the user to specify a port of their choice. In either scenario, classification based on port number becomes impossible if the port number used by the application is unknown.

Madhukar, et al show in [31] that when port-based classification was used for their classification studies on traces from 2003 to 2005, 10% to 30% of the traffic in their mixed application traces belonged to unknown applications, and by Spring 2005, those numbers had increased to 30% to 70%. Madhukar, et al attribute this unknown traffic to P2P applications which existed in the traces used but were unable to be identified. As Madhukar, et

al were also quick to point out, P2P applications were evolving rapidly during the time of their study, which was evidenced by the increased amount of unknown traffic over the time their traces were taken. P2P applications are eager to avoid detection for a variety of reasons, including conflicts with ISPs over bandwidth usage and copyright issues with content prolifically distributed with these applications, so the use of non-standard port numbers and payload encryption has become commonplace for many P2P applications.

Port-based classification is particularly ineffective for Skype because during its initial configuration upon installation, Skype chooses a random port to bind to in the upper port range for its communication to other Skype nodes. Since the choice of port is random, there is no heuristic way to determine which port may be using Skype. Also, due to the mechanisms by which Skype performs NAT and firewall traversal, Skype may use port 80 or 443 for its communication, further obfuscating it with respect to classification by port.

2.2 Payload-based Classification

The employment of packet payloads for the purposes of classifying network applications comes in the form of different varieties of deep packet inspection (DPI). In this method, the packet payload is examined for either regular expression matches for different application-layer protocols or content, or the packet payload is used to create a statistical signature for the application-layer content of the packet. Regular expression searches tend to be much faster than statistical signatures, due to the lack of overhead with respect to calculation of the signature, but in general, DPI methods tend to be slow and create potential bottlenecks for traffic in high-bandwidth networks when used for real-time applications, making them unsuitable for real-time in-network classification.

In [33], Moore, et al propose a process by which a non-automated, iterative procedure is used to classify traffic flows in a 24-hour trace. The authors start by classifying by IANA

port number, but show that by using this method 30% of the packets they observed could not be attributed to a particular application or were misclassified as belonging to the wrong application. The procedure then iterates through progressively more complex inspection of the packet payload, beginning with the packet headers and eventually analyzing all of the bytes in a given packet. Using this methodology, Moore, et al are able to show an extremely high accuracy in classifying different types of applications, including P2P traffic, although the last step of their iterative procedure involves human intervention to correctly classify the traffic flow. Despite this high success rate, this work was done in 2005, and with the dramatic increase in the amount of encrypted protocols for use with P2P applications since then, even human inspection of the traffic flows may not render such good results today.

Another DPI method that has displayed successful results in the past is the creation of regular expression signatures for known protocols in order to produce a match in packet payloads. To illustrate the effectiveness of regular expression matching, in [38], Sen, et al create regular expression signatures for eDonkey, KaZaA, BitTorrent, Gnutella, and DirectConnect, based on features in the application-level headers of TCP streams. The results are very impressive, reaching a correct detection rate of over ninety-nine percent for most of the applications in two network traces taken in 2003. However, since this work was done in 2004, the application-level headers for the applications studied in this work have been modified, and most now use some form of encryption.

In [31], Madhukar, et al use the regular expression signatures created in [38] by Sen, et al to perform a comparison of port analysis, application signatures, and transport-layer methods. The authors state that despite the accuracy of packet payload signature methods, there are three primary issues with employing regular expression signatures for P2P applications. First, payloads may be unavailable for signature generation due to privacy regulations which make accessing these payloads illegal. Second, the only way that payload signature matching is effective is if the classifier knows what it is looking for, and since new P2P pro-

tocols are being created all the time, the classifier would need to be constantly updated with new signatures. Third, encryption renders signature matching completely ineffective, and many P2P applications now employ encryption to avoid detection. This last reason is why a payload-based signature method is ineffective for use with Skype. Skype uses a proprietary encryption mechanism to obfuscate the contents of the packets transmitted between nodes in the Skype overlay network, making a payload signature impossible.

2.3 Flow-based Classification

In the absence of information from packet payloads, network flow behaviors have proven to be a viable method to classify applications in a trace. Since P2P applications are based on a distributed overlay network of nodes, this methodology proposes that the behaviors of the network connections made between the nodes display some regular properties which can then be used to identify them as P2P traffic.

Karagiannis, et al propose a method in [28] that specifically examines the flow behaviors of P2P applications in order to perform classification. The authors used a trace containing a variety of well-known P2P file-sharing applications to select relevant flow features, and then derived a methodology for classifying these applications based on their flow behaviors. In the first stage of their classification mechanism, the authors show that P2P applications typically use both TCP and UDP transport-layer protocols for communication between nodes, so they identify IP addresses which employ both of these protocols. In the second stage, they look at the ratio of distinct IP addresses to which a host is connected to distinct port numbers to which a host is connected in order to find IP address, port number pairs that have a one-to-one ratio. Because P2P applications communicate with a large number of other nodes in the overlay network and typically only have one connection to each node, the authors choose the IP address to port number ratio as distinguishing feature since one IP address connected

to a single port is representative of this single connection behavior. Using this classification scheme, Karigiannis, et al show that they are able to classify 90% of the packets seen in a traces from May 2003 to April 2004 as P2P traffic, and that they are able to classify 99% of the P2P flows correctly.

In [29], Karagiannis, et al extend the aforementioned work to create a general framework for network applications that considers their behaviors at what the authors term the social level, the functional level, and the application level. The social level takes into account a host's behavior with respect to other hosts, specifically how popular the host is and the community of other hosts to which it is connected. The functional level examines if the host behaves as a consumer or provider of services, or both in the case of P2P applications. The application level deals with the transport-layer communications exhibited by a host, and flow characteristics of the host's behavior. The behaviors at the application layer are captured in graphs, which the authors call "graphlets", that are used to more accurately classify applications based on their host behaviors. Using these three facets of host behavior, the authors are able to classify 80% to 90% of the flows seen in three mixed application traces from 2003 and 2004 with a 95% accuracy rate.

Hu, et al also find success when classifying P2P applications using flow statistics in [27] by building application profiles which can be used to identify a particular application. The authors propose a method by which a rule-based profile is built using flow characteristics of P2P applications, and then applications are classified based on a host-level and flow-level match. They point out that when constructing their rule sets, that there are two essential difficulties with using flow properties to identify P2P applications. First, since P2P applications use network flows for varying purposes, for example to get peer information or to transfer data, different flows in the same application may exhibit different flow characteristics. Second, flows in a particular application may not have flow statistics specific to that application, making per-flow statistics ambiguous for identifying a P2P application. Despite

these challenges, the authors are able to correctly classify both BitTorrent and PPLive in network traces from October 2006 to November 2006 and September 2007 to October 2007 with over 92% accuracy in the case of TCP flows, and over 96% accuracy for UDP flows.

While flow behaviors are certainly related to the work on Skype being presented in this paper, the methods for classifying applications above rely on human interaction in a non-automated classification scheme to determine the correct application to which a flow belongs. The goal of this paper, however, is to create a statistical signature for Skype control flows that can be used in an automated system to identify Skype control traffic. Also, whereas the previously mentioned approaches consider all types of traffic generated by an application, both control traffic and data stream traffic, this work focuses only on the control traffic due to Skype's use of proprietary encryption measures.

2.4 Machine Learning Methods

Machine Learning encompasses a variety of different statistical methodologies used for classification, ranging from clustering techniques that determine related features, to decision trees that determine the gain ratio of the features, to Naïve Bayes classifiers which use a probabilistic method to give a weighted probability to elements in the feature vector. One of the most important elements of any Machine Learning technique is that of feature selection for the problem domain. According to Li, et al, "a feature is a descriptive statistic to characterize an object; and, ideally, each object exhibits different feature values depending on the category to which it belongs."[30] After the relevant features for the problem domain have been chosen, these features can then be used to construct a model for a particular object, and subsequently a classifier for recognizing this model among a given dataset.

One method that can aid in the discovery and selection of features to use for classification is clustering, an unsupervised learning method which groups objects with similar features

into clusters based on a distance metric. In [37], Roughen, et al use the Nearest Neighbor and Linear Discriminant Analysis classification techniques to group traffic flows into Classes-of-Service for the purpose of examining flow behaviors to guarantee Quality-of-Service. Using the average packet size and average flow duration as their target flow statistics, the authors are able to successfully cluster the traffic flows from traces taken in 2001, 2003, and 2004 into four categories: Interactive, e.g. telnet, Bulk data transfer, e.g. FTP-data and KaZaA, Streaming, and Transactional, e.g. DNS and HTTPS.

In [42], Zander, et al use the Autoclass Bayesian classifier to cluster network flows for identifying network applications. They also examine which features are best suited to use for classification, and use a set of features that includes packet inter-arrival time, packet length mean and variance, flow size, and duration, and use sequential forward selection to evaluate the contribution of each of the features for classification. While clustering the flows into their respective applications, including FTP data, Telnet, SMTP, DNS, HTTP, AOL Messenger, Napster, and the game Half-Life, the authors find that packet length is the most significant of the features for the applications they chose. The authors achieve accuracy rates of over 85% on average when clustering the flows in to their respective application groups.

Erman, et al also facilitate a clustering process to classify network applications into categories using network flow statistics in [20]. The authors employ the use of K-Means clustering to group network flows into clusters and then map them to their respective application categories using a distance measure heuristic. The results of their offline classification on traces from 2006 are mixed, as the classifier performs well for HTTP, and Email, but performs poorly for P2P traffic in one of the traces taken at a university campus. The authors state that the low accuracy is due to the fact that there is a very small percentage of traffic present in the trace that belongs to P2P applications, and that not enough examples were present in order for the clustering mechanisms to work well.

Li, et al in [30] do a comparison study of port-based classification, DPI classification, Naïve Bayes classifier for traffic flows, and a C4.5 tree for traffic flows. Their work focuses on grouping network applications into categories rather than specific applications, since they describe how protocols can be used to implement many applications of a given type. Using a dataset of four day-long traces taken from two different sites and over a period of four years from 2003 to 2007, their study finds that C4.5 trees produce far better results than the other methods. However, the accuracy of this method declines overall by almost 10% when comparing the day in 2003 to the day in 2006, especially in the category of "Bulk" traffic, which includes file transfer data, where it dropped to as low as 35%. Since the authors point out that their definition of "Bulk" traffic originated in [37], and since the definition given in [37] includes FTP and KaZaA traffic, it would seem that this category would also surely include misclassified P2P traffic which uses encryption mechanisms or unknown protocols. Also, when using the model built with the 2006 dataset, the accuracy of classifying P2P traffic in the 2007 dataset drops to 58%, and the authors attribute this reduction to the fact that the P2P hosts in the 2007 trace are likely using newer versions of the P2P client software.

Williams, et al compare five Machine Learning algorithms for classifying network flows in [40], implemented using the Weka Machine Learning [2] suite. The authors test Bayesian Network, C4.5 Decision Tree, Naïve Bayes using both discretisation (NBD) and kernel density estimation (NBK), and Naïve Bayes Tree for classifying six applications in traces taken in 2001 and 2003. The features chosen to use with these methods were flow statistics which included the protocol, flow duration, flow volume in bytes and packets, packet length (minimum, mean, maximum and standard deviation), and the inter-arrival time between packets, and these features are used to identify FTP, Telnet, SMTP, DNS, HTTP, and the game Half-Life. The results produced by the algorithms show very high accuracy, over 95% for most all but the Naïve Bayes classifiers. However, this work has some serious limitations since the

traces used were anonymized, so the authors had no way to verify that their classifications of the applications were correct. Also, the authors chose the applications to classify based on well-known port numbers, making the results inapplicable to current network applications such as P2P file-sharing applications.

Since Machine Learning techniques have shown in the preceding literature to be viable methods when coupled with flow characteristics to classify network applications, this work employs a network flow-based approach which facilitates the use of a supervised learning classifier to identify Skype control flows. However, unlike the aforementioned work, this paper seeks to generate a flow signature specific to Skype control flows, not a category of applications.

2.5 Classification of Skype

Previous research on Skype has been focused on both identification and classification, since the encryption and obfuscation methods employed by Skype make the traffic produced by the application very hard to decipher. However, none of these previous approaches attempts to create a classification mechanism for the control traffic which maintains the P2P infrastructure of Skype, but rather focus on other features for classification.

In terms of identification of Skype control flows, the amount of research is fairly slim. In [15], Baset, et al describe the nature of the structure and operations of the Skype overlay network, and discuss the methods by which Skype control traffic is able to traverse NATs and firewalls, but offers little else about its behavior or content. While describing the nature of Skype relay traffic wherein voice data is routed through an intermediary supernode, Suh, et al briefly mention the control traffic in [39], only to state that they are explicitly ignoring it for their measurements. Similarly, in [24], Guha, et al concern themselves with observing the behavior of traffic routed through supernodes in order to better measure the Skype overlay

network, but again ignore the control traffic in favor of the actual data packets routed to the supernodes.

The most significant amount of work done on identifying the properties of the Skype control traffic are several works presented by Mellia, Meo, Rossi and others in [17], [36], [18], and [35]. [36] and [35] are particularly dedicated to understanding the behaviors of Skype control traffic, referred to in these sources as signaling traffic, while [17] and [18] offer a more breadth-oriented view of Skype behavior in general with sections that summarize the measurements presented in [36] and [35]. While these contributions offer much in the way of understanding the behavior of Skype control traffic, the authors rely on a classification technique they developed in [19] to identify Skype traffic flows and therefore determine which flows do not belong to a data stream based on the bitrate and number of packets sent. The classification scheme which they developed, however, is designed for use with payloads and statistical qualities of Skype data traffic in mind, and therefore is not tailored to identify Skype control flows.

The issue of classification is across the board focused solely on either packet payloads or flow statistics that are related to the data transfer performed by Skype during voice, video, or instant message streams. As previously mentioned, in [19] a Chi-Squared test on packet payload and a Naïve Bayes classifier on the packet length and inter-packet gap are used to classify Skype data traffic with good results, but these methods do not consider properties of the control traffic for classification. In [21], Freire, et al measure the effectiveness of using a Chi-Squared test on Skype traffic that has been re-routed through ports 80 and 443, presumably because of firewalls or other impediments to the regular Skype flow, in order to distinguish Skype from other web traffic. This work also focuses on data packet payload, and does not consider control traffic perhaps because of the acknowledgement in [15] that Skype control traffic is more robust to NATs and firewalls than its data traffic. Angevine, et al use AdaBoost and C4.5 to classify Skype traffic in [14] using statistical measurements

of Skype traffic flows, but explicitly rule out any flows consisting of one packet, thereby nullifying any chance of categorizing the control packets since in [36] and [35] the authors show that single packet control notifications are not uncommon in the Skype overlay network. A similar methodology is used in [13] by Alshammari, et al, but this time with a whole gamut of machine learning techniques including the aforementioned AdaBoost and C4.5, but also Support Vector Machines, Naïve Bayes, and RIPPER, a rule-based algorithm. Here the single packet flows are not ruled out, but since the classifiers are using input data that includes an overwhelming majority of data flows and since the features selected include elements such as the number of packets per flow (incoming and outgoing) and packet length, the control flows, being vastly different in these respects from the data flows, will surely be misclassified. Considering all of these methodologies and their general disregard for the control traffic produced by Skype, the issue of classifying Skype control traffic is still very open for further investigation.

Chapter 3

Data Collection

3.1 Skype testbed

To accomplish the goal of creating a statistical signature for Skype control flows, the effort to collect Skype traffic was required in order to generate a dataset that could be used for reliable analysis. The goal of the setup for this dataset was to create as close to real-world behaviors of the Skype clients in the testbed environment as possible, but in a controlled manner that could be easily observed so that these behaviors could be used for classification. Also, it was necessary to run Skype without interaction with other clients in order to distinguish the control traffic from traffic generated during data streams.

In order to simulate a real-world Skype client's environment, a total of nine machines were used, configured into groups that would mimic typical user behavior. Since most Skype users have some group "friend" connections that their Skype client communicates with, we decided to recreate this situation through the formation of a graph structure used to define the associations in the groups of "friend" clients. Shown in Figure 3.1, this graph was designed to replicate a very small social network of nodes. In the graph, each vertex represents one of the machines used in the experiment, and the edges represent "friend"

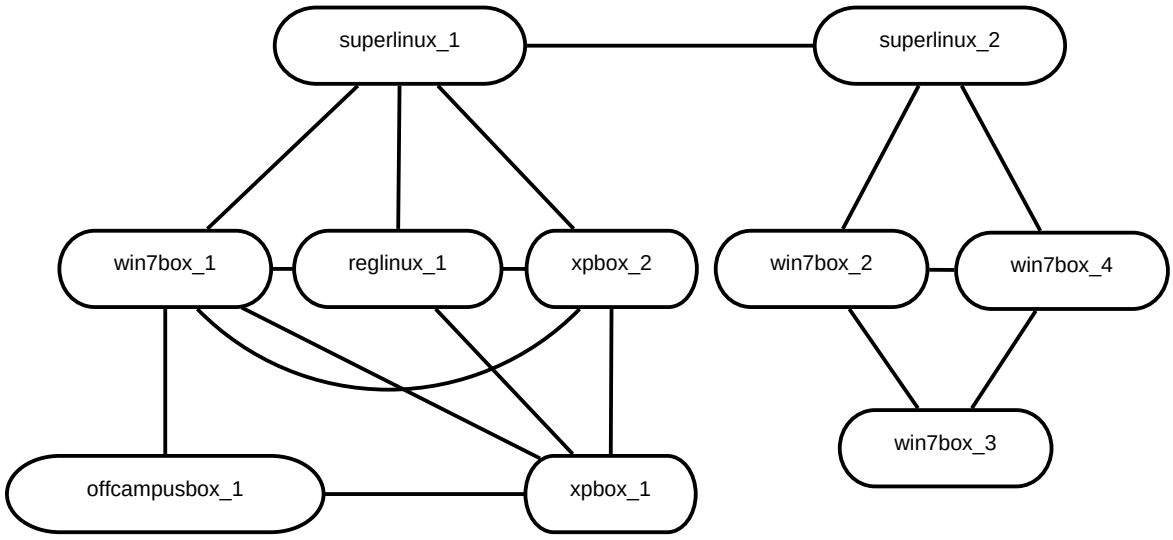


Figure 3.1: Friend associations in the Skype testbed environment.

connections between the machines. The boxes in the testbed environment were configured to be "friends" with each other according to this particular graph structure in order to generate traffic representative of the network of contacts a typical Skype user would have.

This graph structure was also used for several other purposes. The first was to discover in the case that the machines designed to be supernodes actually became supernodes, if the rest of the nodes in the graph would use these two machines as their point of contact with the supernode overlay network. The second reason for the choice of this graph was that in both subgraphs there exists a node which is not "friends" with the likely supernode connected to the rest of the nodes in the subgraph. The facet of interest here was whether these nodes not connected to the likely supernodes would still communicate with them, especially if they were in fact supernodes, and if, in that case, they would use them as their default supernode contact. The third reason for using this graph was that if the likely supernodes were indeed promoted to supernodes, to examine whether the traffic routed through either of these supernodes contained traffic from the other subgraph not connected to it.

The two machines used as candidates for promotion to supernodes in the Skype overlay network both had Intel i7 processors and 8 GB of RAM each and were given public IP addresses in the university address space with the university firewall disabled. These measures were used in order to increase the chances that these machines would become supernodes due to their ample system resources, considerable bandwidth, and no firewall restrictions. Both of these machines had a clean installation of Ubuntu 10.04 64-bit, and all automatic updates, printer services, and all other applications disabled. These two machines are designated superlinux_1 and superlinux_2 in Figure 3.1.

The other seven machines were configured behind a university-firewalled box configured to create a Network Address Translation (NAT) interface to the other machines connected to it through a switch. The machines behind the NAT were assigned addresses in different subnets, and `iptables` rules were instituted to prevent these machines from communicating with each other behind the NAT, in order to force them to use an outside node for communication on the Skype network. This was done to again ensure typical real-world behavior, since most users behind a NAT-ed router or other device typically communicate to nodes outside their own NAT-ed address space. Four of the boxes ran Intel Core2 Duo processors with 2 GB of RAM each, and had a clean install of Windows 7 32-bit with automatic updates disabled. These machines are labeled win7box_1, win7box_2, win7box_3, and win7box_4 in Figure 3.1. Another box had these same system resources, but had a clean install of Ubuntu 10.04 32-bit with automatic updates, printer services, and all other applications disabled. This machine is labeled reglinux_1 in Figure 3.1. The last two boxes had Pentium 4 processors with 1 GB of RAM each and had Windows XP SP3 32-bit installed with automatic updates disabled. These last machines are labeled xpbox_1 and xpbox_2 in Figure 3.1.

On all of the machines in the testbed environment, Skype was the only application running, other than the scripts used to generate user behavior, in order to ensure no interference from other applications so that the traffic generated by Skype could be isolated. Also, since

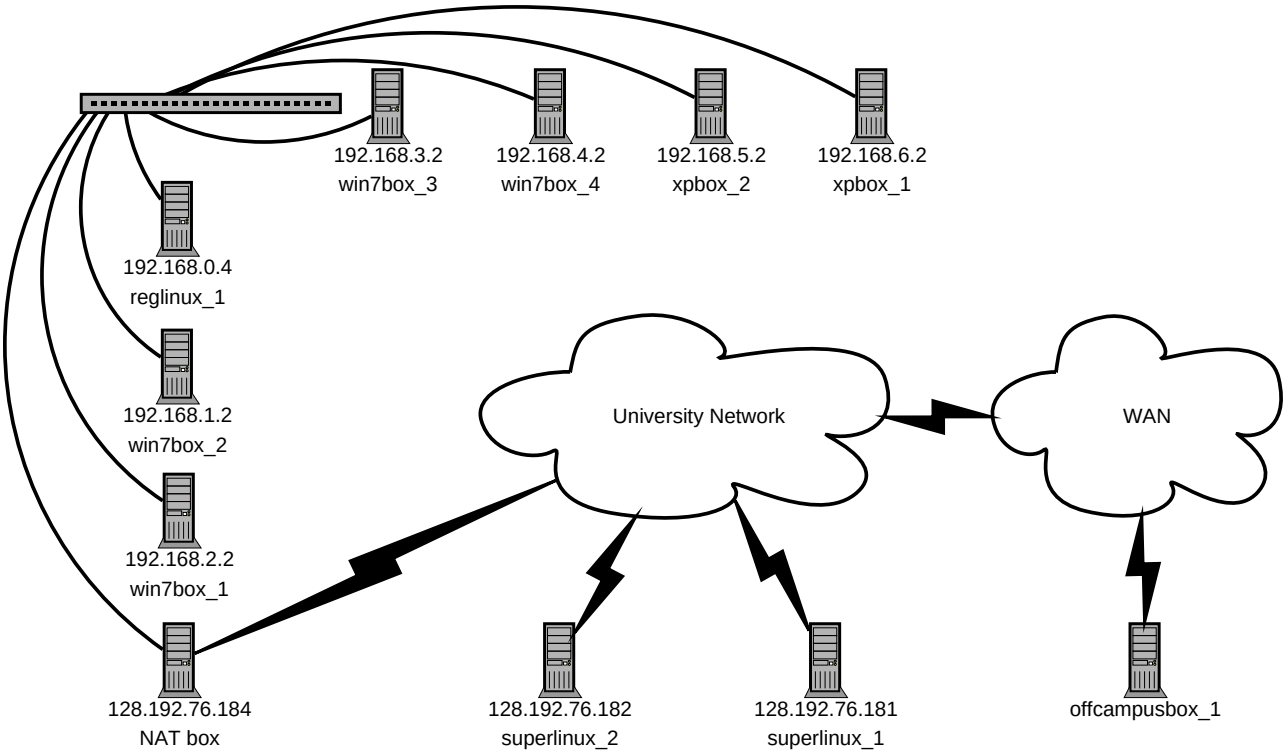


Figure 3.2: Physical topology of the Skype testbed environment.

automatic updates and other services were disabled on all of the machines, this further ensured that the network traces would likely only include traffic from Skype, and thereby reducing noise from other applications in the traces for the dataset.

One other box was configured for use with this experiment, which was located off campus, and was used to receive calls from the boxes in the testbed environment. This box had Windows XP SP3 installed, and ran a script which answered incoming calls and subsequently used iTunes to play talk radio from a local Atlanta station through the microphone to simulate normal VoIP vocal communication. This machine is labeled offcampusbox_1 in Figure 3.1. The physical topology of the Skype testbed environment is shown in Figure 3.2.

Boxes in the NAT were configured so that 2 boxes made calls to the off campus machine, 2 boxes sent instant messages to each other, and the rest simply ran Skype without any other action. The idea was to capture the VoIP behavior of the Skype traces as well as the instant messaging behaviors, since these are the typical functions for which users employ Skype. By examining these behaviors against the control traffic, the assumption was that the control traffic would become easier to isolate since it presumably does not exhibit the same behaviors as the data stream traffic.

The challenge, then, was to artificially create client behavior that was believable enough to make the traffic seem "real". For this purpose, several scripts were written using AutoIt [3], a Visual Basic-based scripting tool on the Windows platform which can be configured to interact with graphical user interfaces. These scripts mimicked two types of normal usage by Skype users, calls and instant messages. The two boxes in the testbed environment that were designated to make VoIP calls did so at two hour intervals, and for either five or ten minutes per call. The choice for these values was somewhat arbitrary, but seemed to be a reasonably average length for a typical phone conversation. Since the box located off campus was playing talk radio while the call was active, the boxes that initiated the calls had their microphones positioned to pick up the speaker output from the call and play it back to the receiving client, so that voice data was being carried over the data stream between the boxes, as it would for a normal VoIP conversation. The two boxes that communicated with each other through the use of instant messages did so at varying intervals, and sent differing lengths of messages ranging from a few lines to around thirty lines. These two scripts were also implemented so that the messages were delivered between the clients at fairly interleaved intervals, so that it would more accurately represent the typical flow of an instant message communication.

After all of the Skype machines were configured and running, the data collection process suffered several plights. The most significant was a failure of the campus network over

intermittent intervals for two days, which not only interrupted the data collection process, but resulted in a re-tooling of the AutoIt scripts to make them more robust to network failures of this type. Also, this interruption prompted the installation of a script on one of the likely supernode boxes that ran a periodic `ping` message in order to monitor network availability. The other problems mostly involved configuration of the scripts, since nuances of the Skype client interface made scripting troublesome for some actions. Skype employs very few keyboard shortcut combinations, meaning that the scripts had to interact with various portions of the client interface using directed mouse clicks to the GUI. Since the Skype GUI is designed to be dynamically interactive with users, accommodations had to be made in the scripts for elements that were repositioned during certain actions or behaved differently if they were inactive for the periods between actions in the scripts. Despite these troubles, the testbed was eventually brought to a state in which it could continue to collect data indefinitely in an automated fashion, or until a Skype update or unexpected hardware failure occurs.

3.2 P2P file-sharing testbed

Four applications were chosen to use for the creation of the P2P file-sharing dataset, based on the protocols they employ: μTorrent and Vuze, which both use BitTorrent, FrostWire, which uses Gnutella, and eMule which uses eDonkey. Although other P2P file-sharing protocols exist, the BitTorrent, Gnutella, and eDonkey protocols are the most popular for freely distributed client software. The clients chosen for the three protocols were selected because they are currently the most popular of the clients for each of the respective protocols, and more than one BitTorrent client was chosen since BitTorrent is currently the most widely used of the protocols, and we wanted to examine different implementations to identify any discrepancies in network behavior between clients which use the BitTorrent protocol.

The BitTorrent protocol uses the distribution of parts of files in the "swarm" of nodes in the BitTorrent network to achieve high availability of files for download. The protocol relies on the use of .torrent files which contain information about the peers in the "swarm" which currently have parts of a file available for download. These .torrent files are propagated through the use of web servers called "trackers" which serve the .torrent files, usually on ad-laden websites, and keep track of which peers are seeding which pieces of the file in the network. Downloading a file in a BitTorrent network is a matter of searching trackers and finding the desired file, then downloading the corresponding .torrent file, at which point the BitTorrent client is able to identify from which peers in the swarm to download parts of the file.

The eDonkey protocol is another server-based protocol, in which files are split in to chunks and identified based on an MD4 sum, and made available on servers in the network. Any client can install the server software on their machine, making the eDonkey network, also known as eDonkey 2000, or eD2k, a decentralized architecture since it does not rely on a few centralized servers to keep information about the files in the network. As an addendum to the eDonkey protocol, eMule also uses an implementation of the Kademlia distributed hash table (DHT) for locating files on other peers in the network, and the use of this DHT allows for faster lookup of files in the network. Unlike BitTorrent, peers to make a direct file-transfer connection to download an entire file from another peer, instead of downloading pieces of a file from many peers.

The Gnutella protocol, on the other hand, does not rely on centralized tracker servers. It instead uses a fully decentralized architecture, and much like Skype, each client stores a list of active "ultrapeers", usually numbering around three, with which it communicates. These ultrapeers store the address of other ultrapeers with which to perform file queries. When a client searches for a desired file, it queries the active ultrapeers in its internal list to find desired files, and if none of these ultrapeers have the desired file, they subsequently

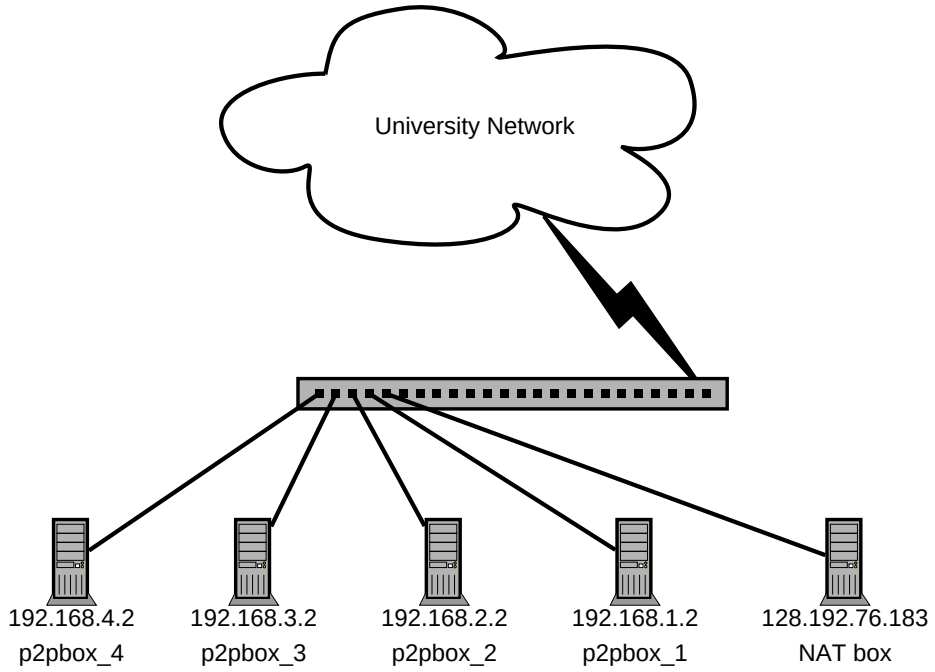


Figure 3.3: Physical topology of the P2P file-sharing testbed environment.

query their active sibling ultrapeers, which then query the non-ultrapeers they communicate with. The peers which have the file then make a file transfer connection with the peer that originally queried for the file, and like eDonkey, download the entire file from the other peer.

Five machines were used to create the testbed for P2P file-sharing applications, all of which had Intel Core2 Duo processors and 2GB of RAM. The four machines used to act as P2P clients each had a clean install of Windows XP SP3 with automatic updates disabled, and the machine used as the NAT for the rest of the machines had a clean install of Ubuntu Server 10.10. The physical topology of the P2P file-sharing testbed environment is shown in Figure 3.3, and the machines used as the clients for the applications are labeled p2pbox_1, p2pbox_2, p2pbox_3, and p2pbox_4.

Table 3.1: Fake multimedia files created for use with the P2P clients

Name	size (bytes)
Archer.S02E05.HDTV.Rip.avi	190840832
born.this.way.lady.gaga.mp3	3276800
COD-Black-Ops-Keygen.exe	470244
DeadSpace2KeyGen.exe	1052747
Forget_You_Cee_Lo.mp3	3276800
Glee.S02E13.HDTV.Rip.avi	182452224
glee_baby.mp3	3276800
Glee_Somebody_To_Love.mp3	3276800
House.S07E10.HDTV.Rip.avi	202375168
Justin-Bieber-Never-Say-Never.mp3	3276800
katy.perry.firework.mp3	3276800
Rihanna-S&M.mp3	3276800
Sipser.Theory.of.Computation.Solutions.Manual.pdf	6553600
wowkeygen.exe	907389
Yogi_Bear_Movie_2010.avi	759169024

The original intention was for each application to run on the client machines for 7 days, and to use AutoIt scripts on the client machines to simulate real-world user behavior of file downloads. In order to avoid copyright issues, the first method for sharing and downloading files used was to create fake files that resembled media commonly sought by other peers before the data collection began by manipulating public domain research papers from arxiv.org [4]. These .pdf files downloaded from arxiv.org were then renamed as popular media in order to entice other file-sharing peers to download them in order to create more traffic during the experiment. Using the Unix `dd` function, the file sizes were then padded to resemble realistic file sizes of the media they spoofed. In all, 15 files were created, and their names and sizes are listed in Table 3.1. Although this seemed to be a very beneficial approach, it introduced a few challenges. First, one of the machines had to be dedicated as a "seeding" client who simply uploaded files to the other clients, while the rest of the clients downloaded

files at random intervals. The reason for this was to ensure two things: first, that the files were actually seeded and available over the network when the scripts running on the clients attempted to download them, and second, in order to verify that these non-copyrighted files were indeed the correct files, so that copyrighted material was not accidentally downloaded instead.

Despite overcoming these limitations during the initial implementation, these measures were quickly revised after initial setup of the testbed environment, because after the testbed was configured and two days' worth of μTorrent data had been collected, the university implemented new IPS rules in the university firewall which blocked all incoming .torrent file downloads into the network. The decision was then made that if a solution could be found to allow exceptions in the firewall, that to avoid any further complications, only open-source software would be downloaded and shared from the clients in the testbed environment. The scripts were then re-written to accommodate this alteration, except for FrostWire, due to the limited availability of reliably non-malware files found during searches for open-source software on this client. After a lengthy ongoing discussion with the administrators of the university firewall for several weeks, eventually some modifications had to be made to the setup of the machines in the testbed environment due to time constraints. The issue with the firewall rules disallowing .torrent file downloads was not resolved, but a workaround was developed for the BitTorrent clients, and both the eMule and FrostWire clients were able to perform their regular functionality despite the firewall issues. The decision was made to run these applications side by side in the testbed environment, with two boxes running each application, in order to collect as much traffic as possible for each application in the time given for the project. The eMule and FrostWire clients were therefore run in parallel for 7 days, and the BitTorrent clients were also run in parallel for 7 days. Below are detailed accounts for the configuration of each of the clients.

3.2.1 μTorrent

μTorrent is the most popular BitTorrent client available today, and it promotes itself as "a (very) tiny BitTorrent client". This was the first client configured for the testbed environment, and it was during the setup and initial days of data collection of μTorrent that the aforementioned hurdles with the AutoIt scripts and faked .torrent files were addressed and overcome. The faked .torrent files were uploaded to two different prominent trackers, and the URLs for the .torrent files from these trackers were used in the scripts. One of the boxes was configured to simply seed the files so that the trackers would not remove the links from their sites, as they are apt to do in the case of a file which no peers are currently distributing. The other three boxes were scripted to download the files at varying intervals, and the scripts were written to take into account the fact that as more of the boxes in the testbed environment downloaded and subsequently seeded the files, more parts of the files would be available to the box downloading the file.

As mentioned previously, after two days of data collection, the university firewall IPS rules were instated, and all subsequent .torrent file downloads were canceled. Therefore, in accordance with the desire to avoid further issue with the university's firewall policies, the scripts were rewritten to download only open-source content, including several versions of Ubuntu Linux, several versions of Fedora Linux, Debian Linux, and Open Office. In the revised scripts, all four boxes downloaded these files at varying intervals, and the URLs for the .torrent files were provided from the same well-known trackers to which the faked .torrent files were uploaded. After the files downloaded, the clients then seeded this content to the rest of the network to generate upload traffic on the client.

In order to still collect traffic for the BitTorrent clients in lieu of the university firewall rules, the .torrent files for the open source programs listed above were downloaded off campus and then transferred to the BitTorrent clients manually. Since the firewall rules only blocked incoming .torrent file downloads, the clients were able to function as normal despite the lack

of interaction with a tracker server. There was no discernable difference in the behavior of the clients when the .torrent files were not downloaded from a tracker, so this measure proved to be extremely fruitful for the purposes of data collection.

3.2.2 Vuze

Vuze is a popular Java-based BitTorrent client which was formerly known as Azureus. The AutoIt scripts written for the Vuze clients were designed to perform the same basic actions as the μTorrent clients, downloading the same open-source files from the same tracker URLs, but using the semantics of the Vuze application. Due to the university firewall rules preventing the download of .torrent files, the .torrent files were downloaded off campus and then manually transferred to the clients as was done with the μTorrent clients, since both applications share the same basic functionality.

3.2.3 eMule

eMule is the most popular client application which employs the eDonkey network. It also uses the Kademlia DHT to locate files in the network. Unlike the other clients which have only one listening port for incoming connections, eMule uses two ports, a TCP listening port for the eD2k network of eDonkey servers, and a UDP listening port for the Kademlia network. The Kademlia DHT is used for significantly reducing the search time to find files in the network, since only $O(\log(n))$ peers in the network are queried for the existence of a desired file.

eMule's operation was not affected by the university firewall after some adjustments were made, so the scripts were written to download the same open-source software distributions listed above, Ubuntu, Fedora, Debian, and Open Office. These files were scripted to be

downloaded at varying intervals, and after their completion, would then seed these files to rest of the eMule network, creating upload traffic on the client.

3.2.4 FrostWire

FrostWire uses a combination of the Gnutella network and BitTorrent, and is the most popular remaining Gnutella-based client. It is a sibling software to the very popular LimeWire file-sharing application, which was the most popular Gnutella-based client for several years. Originally, LimeWire was intended for use with this project, but as of October 26, 2010, a federal court order shut down distribution of the LimeWire client software. In lieu of this action, FrostWire was selected for use with this work.

However, using FrostWire proved to be very challenging when designing scripts to mimic user behavior. The controls on FrostWire's graphical user interface were not recognizable to the AutoIt script, and the AutoIt script could not be configured to direct input to the FrostWire GUI window. This condition therefore preempted the use of scripts for mimicking user behavior, but this turned out to be a fitting correlation with another of the troubles with FrostWire, namely that when developing the scripts, finding open-source files for download was difficult at best, and when some were found, they turned out to be malware, as reported by the anti-virus software installed on the machine used to develop the scripts. The decision was then made to alter some of the conditions of the dataset, and allow for the downloading of potentially copyrighted content, but only capture 150 bytes of packet payload so as to not capture the packets in their entirety during the trace, and to erase the files from disk immediately upon download. This way, the content could not be reconstructed from the packet payloads found in the dataset, and the content would not be hosted to other peers from the machines in the testbed environment. Since the statistical flow behaviors were of main concern during the analysis of the traces, not capturing the entire packet content did not present a significant setback while analyzing the datasets.

Table 3.2: Size of Data Collected for Each Testbed Environment

Application	Machine	Amount of Data Collected	Dates Collected
Skype	NAT	11GB	1/31/2011-4/22/2011
	superlinux_1	3.3GB	
	superlinux_2	3.0GB	
eMule	p2pbox_1	19GB	3/26/2011-4/3/2011
	p2pbox_3	20GB	
FrostWire	p2pbox_2	6GB	3/26/2011-4/3/2011
	p2pbox_4	5.1GB	
μTorrent	p2pbox_1	33.1GB	4/3/2011-4/11/2011
	p2pbox_3	19.9GB	
Vuze	p2pbox_2	10.6GB	4/3/2011-4/11/2011
	p2pbox_4	8.3GB	

Various video content was downloaded based on availability in the network, and upon completion, the files were deleted from the disk. The faked files mentioned previously were used to create upload traffic by sharing them in the designated folder of the FrostWire clients. This circumvented the issue of distributing copyrighted material over the FrostWire network, and it gave a useful purpose for the created files.

3.3 Amount of Traffic Collected

Table 3.2 shows the totals for the traffic collected in the Skype and P2P file-sharing testbed environments. The totals comprise 11 weeks worth of Skype network traces, and 7 days of network traces for each of the P2P file-sharing applications. As is apparent from the totals, the Skype clients generate far less network traffic even when data streams are present than the file-sharing programs, and despite the fact that only 150 bytes are being captured from each packet in the file-sharing traces.

Table 3.3: Total Packets Collected in Each Trace

Application	**Machine**	**TCP**	**UDP**
Skype	reglinux_1	323203	148913
	win7box_1	857842	27865148
	win7box_2	1050746	67959
	win7box_3	908681	157083
	win7box_4	406976	149384
	xpbox_1	958740	27895117
	xpbox_2	321413	271306
	superlinux_1	1439502	1173215
	superlinux_2	410145	164684
eMule	p2pbox_1	112146047	3374431
	p2pbox_3	119109333	3993447
FrostWire	p2pbox_2	31701557	7392566
	p2pbox_4	26256448	6744909
μTorrent	p2pbox_1	33399975	204060033
	p2pbox_3	32993651	114145758
Vuze	p2pbox_2	41691671	20480443
	p2pbox_4	31235095	19935152

In Table 3.3, the total number of packets seen in each trace are shown, divided by protocol. According to the packet totals, Skype uses primarily TCP connections for instant message traffic, and UDP streams for VoIP calls. The P2P file-sharing applications generate far more traffic than the Skype clients, despite the fact that they ran for only 7 days each, as opposed to 11 weeks of Skype traffic. However, since the P2P file-sharing applications were searching for and downloading files from many different peers in their networks, it stands to reason that they should generate a great deal more traffic than the Skype clients, since at most the Skype clients are actively communicating with only one other peer in the case of the boxes performing calling and instant messaging behaviors.

Chapter 4

Data Analysis

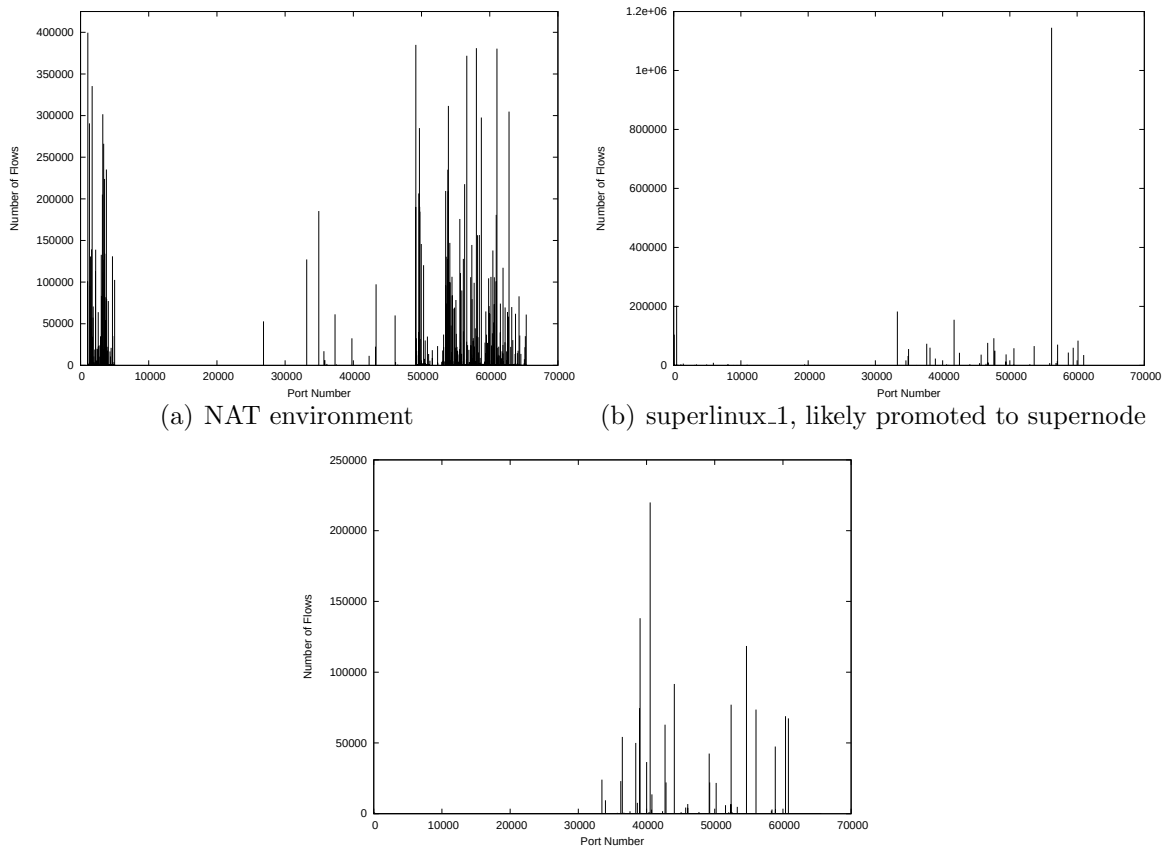
4.1 Behavioral Analysis

While analyzing the network traces, a few facets of the Skype testbed environment were of particular interest, namely if there was any indication that the machines designed to be promoted to supernodes displayed network traffic indicative of a supernode status, and which boxes inside the group behind the NAT communicated with these likely supernodes. In order to examine the Skype traces for finding the features we were looking for, a Python script which extracted values for the mean and standard deviation of the bytes per packet and delay between packets for all flows in the traces which contained more than 100 packets was run on the traces in order to observe the general behaviors exhibited by the flows. Upon studying the output of this script, it became clear that while one of designed supernode boxes (superlinux_1) displayed a highly increased number of connections with many more IPs than the other boxes, the other designed supernode (superlinux_2) displayed the same behaviors as the two machines in the NAT which did not run any Skype data traffic. Also, whereas superlinux_1 showed many flows to and from the NAT, superlinux_2 showed no flows to or from the NAT. Therefore, we concluded that while it is likely that superlinux_1 was

promoted to supernode status, superlinux_2 was not, and that due to the promotion of the machine to supernode status and the observation of flows to and from the NAT, some of the overlay control traffic for the NAT machines was created by or routed through superlinux_1.

In addition to this observation, we also identified several types of likely control flows that existed across the traces for the machines in the testbed environment. The first of these control flows were TCP flows which transmitted a few bytes per packet and had very regular intervals, and were transmitted from a variety of different university IP addresses. The second type of control flow we observed were UDP flows which carried a very consistent packet size, were transmitted in regular intervals, and like the TCP flows, came from university IP addresses. However, these UDP flows also existed between the boxes in the NAT that were "friends" with the machine promoted to supernode status, whereas there were no TCP connections between the boxes. Therefore, it seemed likely that since "friends" in the Skype network must communicate their status as online or offline to the other "friends", that these UDP flows transmitted this information. On the Windows machines, there was another consistent TCP flow which exibited very similar behavior to the other TCP flows, but came from IP addresses registered to Skype.

Another property of the network traces of interest was the port usage for Skype and the P2P file-sharing programs. Figure 4.1 shows the port usage for the machines running Skype in the NAT, and the two designed supernodes. As is evidenced by these figures, the port usage for the Skype application is typically split between the lower port ranges and the 40000 to 60000 range. This is due to the fact that Skype uses ports 80 and 443 to do NAT and firewall traversal, but also sets a static listening port in the upper port range upon initial setup. Thus, in the case of the two machines not behind the university firewall, there is very little traffic on the lower port range, and there is one port which sees much more traffic than the others. In the case of the NAT, there are a small handful of ports which are clearly preferred for usage, and a very large number at 80 and 443. However, there is



(a) NAT environment

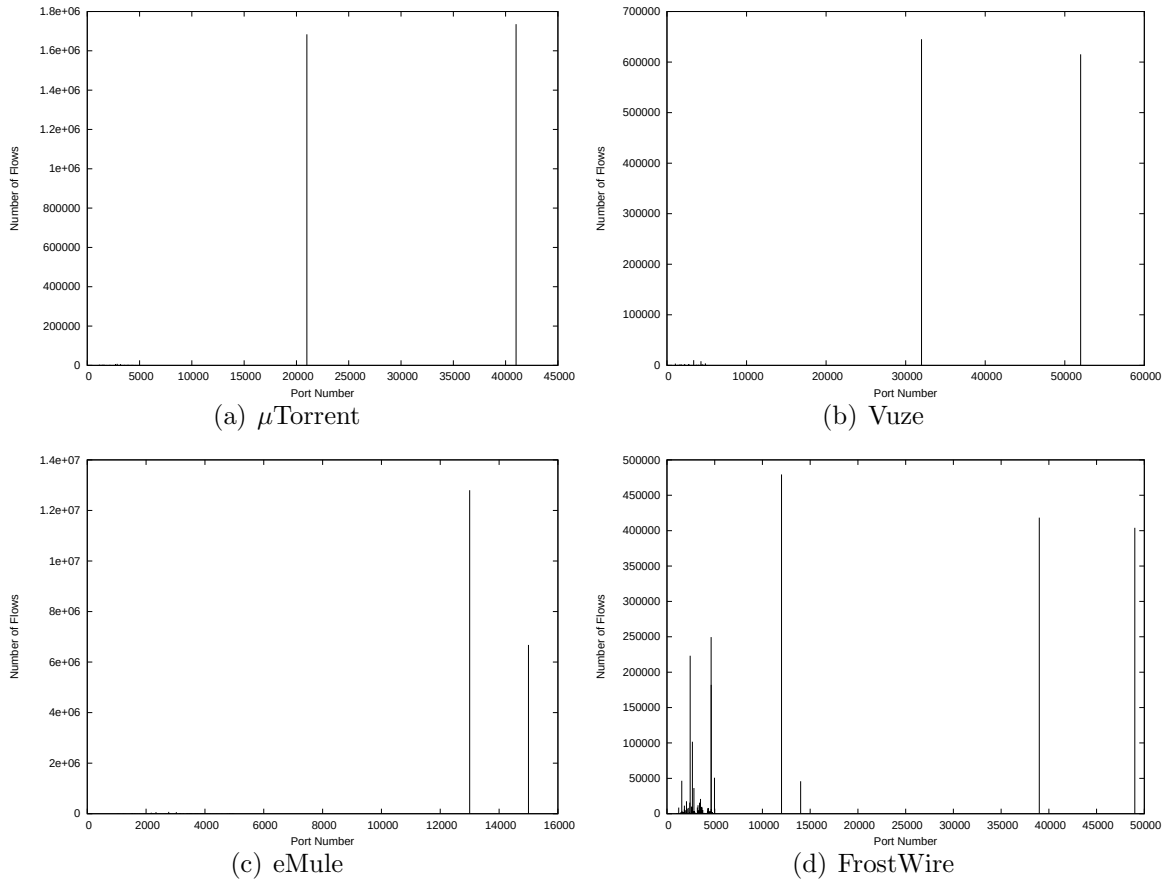
(b) superlinux_1, likely promoted to supernode

(c) superlinux_2, not promoted to supernode

Figure 4.1: Port usage of the machines running Skype.

also a generous amount of traffic on ports in the lower range other than 80 and 443, and it is unclear from observation of the traffic why this is so, but since it occurs only for the boxes behind the NAT, it seems likely to be an artifact of Skype's NAT and firewall traversal functions.

In the case of the P2P file-sharing programs, each program has a listening port in the upper port range configured upon initial setup, and eMule has two separate listening ports configured, one TCP port for the eD2k network, and one UDP port for the Kad network. Figure 4.2 shows the port usage for each of the P2P file-sharing programs, and the BitTorrent



(a) μTorrent (b) Vuze

(c) eMule (d) FrostWire

Figure 4.2: Port usage of the machines running P2P file-sharing applications.

clients clearly use the listening ports assigned to them for almost all of their traffic flows. However, in the case of eMule, the port in the upper range assigned for the UDP listening port for the Kad network seems to not be used at all in favor of the TCP ports assigned to the machines. The other odd piece to the eMule port usage is that no ports in the upper range are used, which is especially puzzling since the UDP listening port is assigned in that range, but the range in which the TCP listening ports are assigned do show port usage. FrostWire also displays some interesting port usage properties, as the two higher ports on the graph, 39000 and 49000 were the listening ports configured for the clients, but there

are other ports which see a lot of traffic, which could be due to search functionality in the Gnutella network, or that FrostWire uses these ports to listen for incoming connections, but initiates connections to other peers on one of the heavily used ports in the graph.

4.2 Statistical Model

In order to create a statistical model for the Skype control flows, we used three basic premises to identify the Skype control traffic in the traces, which included the fact that the size of the packets in the control traffic should be more or less constant, the packets should exhibit a strong periodic behavior, and that the duration of the communication should be long-lived. These premises were derived from the fact that in any P2P overlay network, the nodes must send "hearbeat" or "keep-alive" messages to inform the other nodes in the network that they are still online. Since these "heartbeat" messages are intended only to advertise that the host is alive, the size of the packets should not vary much. Also, the "heartbeat" messages must come at regular intervals, and the communication between hosts must persist for the duration that the hosts are online in order to be aware of each other. Conversely, any data stream traffic trasmitted over the P2P network should exibit the properties of having "bursts" of packets, which may not conform to a particular uniform packet size or periodic behavior, and are likely to be short-lived.

Due to the identification of the classes of flows mentioned above, the assumptions made for these premises seemed to be confirmed by the flow behaviors in the traces. All of the machines in the testbed environment had flows which met the three criteria listed above for our definition of control protocol flows, and the boxes that were trasmitting voice and instant message data additionally contained flows which matched our definition of data stream behavior. Given these confirmations of the premises for identification of the control

protocol flows, we set about extracting a statistical model from the traces to be used for training the classifiers.

The first decision to be made about the statisical model was the time window to be used for capturing statistical flow behaviors. We chose one hour as the decision window for analyzing a flow for two important reasons. The first was that despite the fact that we were using an offline classification method, anything longer than an hour seemed to be too long to make a decision in any classification scheme, since in a typical intrusion detection system (IDS), decisions about flows are made much more quickly than say, a day's worth of flow data. However, using any time window shorter than an hour may have excluded some control flows, and since our classification scheme was based on the flow statistics, not on a regular expression match or payload analysis, an hour seemed to meet the condition for brevity without losing vital data. The second reason was that anything greater than an hour would have reduced the number of labeled instances in the dataset used for training the classifier, and since we hoped to maximize the number of labeled instances used during classification, an hour seemed an appropriate choice given the constraint of requiring enough time to account for statistical flow behaviors.

After selecting a time window for the instances in the statistical model, several considerations had to be made for the features to be included in the dataset. The first features chosen were the protocol, TCP or UDP, the bytes per packet (BPP) and the inter-packet delay (IPD), or time between packets, since these values gave information about the size and periodicity of the "heartbeat" messages. The mean, standard deviation, and median of the BPP and IPD were used in the statistical model in order to capture the distributions of these values over the hour time window. The intuition was that the distributions for these two values in the case of the control traffic should be very similar between instances. The other features chosen were the number of flows in the given hour-long window (FPH), the number of packets per flow (PPF), and the number of bytes per flow (BPF). The mean, standard

Table 4.1: Feature Vector for Skype Control Flows

0) IP
1) Protocol
2) FPH
3) mean(PPF)
4) std(PPF)
5) med(PPF)
6) mean(BPP)
7) std(BPP)
8) med(BPP)
9) mean(IPD)
10) std(IPD)
11) med(IPD)
12) mean(BPF)
13) std(BPF)
14) med(BPF)

deviation, and median of the latter two features were included in the statisical model, again to capture the distributions of these values over the hour time window. The complete feature vector is shown in Table 4.1.

A Python script was implemented to extract the relevant features for the dataset, using the Python dpkt module and Argus [5], a software tool for flow analysis of a network trace. Argus was specifically used for calculating values such as the FPH, PPF, and BPF. This same script was used with the P2P file-sharing traces, since we assumed the same intuitions about the properties of the control traffic for the P2P file-sharing applications, based on the fact that they also employ a P2P architecture and therefore must also rely on "hearbeat" messages to maintain their overlay network. The output of the script were instances of each IP which matched the flow properties of the premises mentioned above, and their values for the features listed above at each hour they matched the criteria.

In order to determine which IPs matched the given criteria, the IPs were filtered based on the flow behaviors they exhibited. First, the IP address and port were used to identify "bursty" flows by putting the inter-packet delay values into bins, and then calculating if these values belonged mostly to one bin. The bin size was set to 5 minutes, and in order to exclude these "bursty" flows, if 90% or more of the values were less than 1 second, the flow was discarded. Next, the flows were examined based only on IP address, and not by port as well, in order to examine the packet sizes of the flows irrespective of whether multiple ports were used during potential control traffic flows. The bytes per packet for one IP were placed into bins of 30 bytes, and the number of packets per bin in relation to the total number of packets was used to decide whether to keep the flows. However, to account for flows that potentially straddled a bin boundary, the bins were aggregated into sets of three, and if 90% or more of the packets were in the same aggregated bin, the flow was kept. Last, the time difference between the first timestamp an IP was seen in the trace and the last timestamp an IP was seen in the trace in the hour window was used to ensure that the flows persisted for at least 90% of the hour time window in order to keep them.

The script was run on the traces for 9 weeks worth of Skype traffic, and was run for all of the flows incoming and outgoing for each box. This was done to ensure that the behavior of the `ping`/`pong` element of the "heartbeat" messages was captured in the statistical model, since the machines have to respond to the "heartbeat" messages sent by the other peers in the network. The script generated 41,069 instances of Skype control traffic flows to be used for classification.

The script was also run on each of the P2P file-sharing traces, and in total processed 7 days' worth of traffic for each application, with 2 boxes per application. During this feature extraction on the traces, 84,680 instances for the P2P applications were derived. The script generated 14,179 instances for μTorrent, 32,062 instances for Vuze, 27,274 instances for eMule, and 11,165 instances for FrostWire.

Table 4.2: Percentage of Total Traffic Belonging to Control Flows for Skype Clients

Client	**TCP**	**UDP**
reglinux_1	81.8%	2.03%
win7box_1	23.05%	0.002%
win7box_2	20.45%	1.02%
win7box_3	22.35%	0.2%
win7box_4	47.71%	1.83%
xpbox_1	20.93%	0.002%
xpbox_2	57.68%	1.57%
superlinux_1	22.03%	3.31%
superlinux_2	60.71%	8.53%

4.3 Control Flow Totals

After the features were extracted, one item of interest was how much traffic was labeled as control flows in the traces for Skype and for the other applications. Table 4.2 shows the total amount of control traffic for TCP and UDP flows in the traces for each of the Skype clients. From Table 4.2 we can see that the boxes running no VoIP or instant messaging data streams have a very large percentage of TCP traffic attributed to control flows, which is to be expected, and the boxes running data streams have considerably lower percentages of TCP control flows. These percentages also aid in confirmation of superlinux_1's promotion to supernode status and that superlinux_2 was not promoted to supernode status, since superlinux_1 exhibits properties similar to the boxes with data streams, and superlinux_2 exhibits properties similar to the boxes only running Skype in the background. Another interesting property of these totals is that UDP data is a very small portion of the control traffic for the boxes behind the NAT, and it is only slightly higher for the boxes outside the NAT. The UDP traffic is also a much higher percentage of the traffic for those boxes not running data streams.

Table 4.3: Percentage of Total Traffic Belonging to Control Flows for P2P File-sharing Clients

Application	**Client**	**TCP**	**UDP**
eMule	p2pbox_1	8.23%	3.71%
	p2pbox_3	11.87%	3.28%
FrostWire	p2pbox_2	7.9%	12.62%
	p2pbox_4	8.2%	11.38%
μTorrent	p2pbox_1	30.32%	0.1%
	p2pbox_3	21.15%	0.12%
Vuze	p2pbox_2	25.37%	45.45%
	p2pbox_4	22.8%	42.75%

Table 4.3 shows the total amount of control traffic for the P2P file-sharing applications for the two machines running the application. According to the results, eMule runs a mix of TCP and UDP control traffic, but has considerably more TCP control traffic. FrostWire also uses a mix of TCP and UDP, but uses UDP more prominently for its control flows. μTorrent uses TCP almost exclusively, and the control traffic is of a much higher percentage for both BitTorrent clients than for the other protocols. Vuze shows much more UDP control flows than TCP control flows, and like the other BitTorrent client, the control traffic comprises a significant portion of the total network flows.

Chapter 5

Classification and Results

5.1 Classifiers

Four classifiers were used to test how well the statistical model was able to distinguish the Skype control traffic flows from the P2P file-sharing flows, and the classifiers were chosen because they have shown to be effective for classifying network flows in previous work. These selected classifiers were implemented through the use of the Weka Machine Learning package [2], and included Decision Trees, Naïve Bayes, k-Nearest Neighbor, and Support Vector Machines (SVM). Since Weka has no native SVM package, the libSVM package [6] was used, imported into Weka.

Decision Trees use a gain ratio measure to determine which feature in a feature vector most evenly divides all of the instances, and then repeatedly uses this scheme with all of the features to create a tree in a very short amount of time that is human-readable and fast to traverse. Naïve Bayes classifiers measure the probability that a feature in the feature vector is a positive or negative example given some set of training data, and use the "naïve" assumption of conditional independence for each trial, or feature, in the feature vector to calculate overall probability that the instance is positive or negative. k-Nearest

Neighbor classifiers are "lazy" learning classifiers which do clustering based on a weighted vote between the k nearest points to the instance being classified. Support Vector Machines map a highly dimensional feature vector to a linearly separable space, and use a Maximum Margin Hyperplane to separate the instances into one category or another based on the points, or Support Vectors, closest to the Maximum Margin Hyperplane.

The dataset used with the classifiers was comprised of all of the Skype instances extracted, along with all of the P2P file-sharing application instances that were extracted. In all, the dataset contained 125,749 instances to be fed into the classifiers. In order to prevent over-fitting to the training data in the classifiers, 10-fold cross-validation was used for each of the classifiers.

Due to the computational complexity of generating a model with the SVMs, a subset of the data was used in order to significantly reduce the time to generate the model. Two days of each P2P file-sharing application were used, which amounted to 3217 instances for μTorrent, 4139 instances for Vuze, 3352 instances for eMule, and 1775 instances for FrostWire. To fairly sample from the Skype data so that all of the clients in the testbed environment were represented in the classification dataset, the 12,482 instances to be sampled were divided equally amongst the Skype clients, and then randomly sampled from each client, half from TCP instances and half from UDP instances. After balancing the dataset, a total of 24,965 instances were fed to the SVM classifier.

Four measures were used to evaluate the classifiers, accuracy, true positive rate, false positive rate, and precision. The accuracy is the number of predictions that are correct divided by the total number of instances, and is important because it tells us how well the classifiers did overall. The true positive rate measures how well the classifiers did in classifying Skype as Skype and P2P traffic as P2P traffic, and similarly, the false positive rate measures how well the classifiers did in not classifying Skype as P2P traffic or P2P

Table 5.1: Skype vs. Non-Skype

Classifier	**True Positive Rate**	**False Positive Rate**	**Precision**	**Overall Accuracy**
J48 Decision Tree	99.8%	0.2%	99.8%	99.8767%
IBK kNN, k=1	99.6%	0.2%	99.6%	99.7702%
Naïve Bayes	98.9%	28.0%	63.1%	80.7513%
SVM	56.2%	0.0%	100%	78.1013%

traffic as Skype. The precision tells us out of the instances classified as belonging to Skype or P2P traffic, how many were correctly classified.

5.2 Results

5.2.1 Classification

Two different initial experiments were run using this dataset on these classifiers. In the first experiment, the instances were labeled as "1" for Skype and "-1" for non-Skype in order to determine how well the classifiers would do in identifying Skype against all of the other P2P traffic. Table 5.1 shows the results of this experiment, and as is apparent, the Decision Tree and the k-Nearest Neighbor classifier produced very similar high overall accuracy and low false positive rates for Skype. The Naïve Bayes classifier performed only slightly worse the Decision Tree and k-Nearest Neighbor for the true positive rate, but the false positive rate was extremely high, and the overall accuracy was the lowest of the classifiers. Naïve Bayes also yielded a poor precision for Skype. The SVM clearly performed much worse than the rest of the classifiers for the true positive rate of Skype, but the instances it classified as Skype were always correct, as indicated by the precision. Given these results, the easiest to

Table 5.2: Each Application Labeled

Classifier	**Application**	**True Positive Rate**	**False Positive Rate**	**Precision**	**Overall Accuracy**
J48 Decision Tree	Skype	99.8%	0.1%	99.8%	97.7336%
	eMule	98.5%	0.6%	97.7%	
	FrostWire	93.9%	0.6%	94.3%	
	Vuze	96.7%	0.9%	97.2%	
	μTorrent	95.7%	0.6%	95.6%	
IBK kNN, k=1	Skype	99.6%	0.2%	99.7%	92.0143%
	eMule	92.1%	1.9%	93.0%	
	FrostWire	77.0%	2.4%	75.7%	
	Vuze	89.5%	3.8%	88.9%	
	μTorrent	87.2%	1.5%	88.0%	
Naïve Bayes	Skype	98.9%	25.8%	65.1%	41.3427%
	eMule	9.6%	0.6%	81.0%	
	FrostWire	60.9%	41.8%	12.4%	
	Vuze	1.6%	1.8%	23.4%	
	μTorrent	10.4%	1.6%	45.9%	
SVM	Skype	51.3%	0.0%	100%	49.9647%
	eMule	29.2%	0.1%	98.8%	
	FrostWire	15.3%	0.1%	96.8%	
	Vuze	99.9%	66.0%	32.8%	
	μTorrent	24.6%	0.0%	99.6%	

use classifiers, Decision Tree and k-Nearest Neighbor, are the clear winners, and performed remarkably well on the dataset.

In the second experiment, each instance was labeled according to which application it belonged, in order to determine if there would be a significant difference in the results if the outcome was not a binary classification. Also, since the more general goal of this work was to create a statistical model to recognize P2P applications, we were interested to see how well the classifiers did on the features extracted for the other P2P applications. The same data reduction measures used in the first experiment for the SVM classifier had to be taken during this experiment due to computational complexity.

The results of the second experiment are shown in Table 5.2, and the Decision Tree again performed the best on this dataset. The Decision Tree was surprisingly effective at not only

Table 5.3: Skype vs. Non-Skype with Correlation-Based Feature Selection

Features Used: Protocol, FPH, med(PPF), med(BPP)

Classifier	**True Positive Rate**	**False Positive Rate**	**Precision**	**Overall Accuracy**
J48 Decision Tree	99.1%	0.3%	99.4%	99.5101%
IBK kNN, k=1	99.1%	0.3%	99.4%	99.5149%
Naïve Bayes	93.6%	45.3%	50.1%	67.4304%

classifying the Skype traffic correctly, but also the other P2P traffic as well. Unfortunately, the other classifiers did not fare quite as well. Although the k-Nearest Neighbor classifier performed almost as well as the Decision Tree on Skype, it had some trouble distinguishing between the other applications, most notably FrostWire. Since FrostWire had the smallest number of instances in the dataset, this is perhaps not so surprising. The other two classifiers performed very poorly with this dataset, most notably Naïve Bayes, which fared much worse than the other classifiers. The low accuracy of the Naïve Bayes classifier and the SVM clearly shows that these classifiers are not a very good choice for use with this type of flow data. However, since the Decision Tree performed the best for this experiment and the previous experiment, it is clearly effective for both binary and multi-class classification for identifying these control flow signatures.

5.2.2 Feature Selection

In order to determine whether all of the features in the feature vector were relevant in classifying Skype control flows, two different types of Feature Selection were performed. The first method was Correlation-Based Feature Selection, which was performed by using the associated package in Weka, and a subset of four features was returned by this Feature Selection mechanism as the necessary features for classification: Protocol, Flows Per Hour,

Table 5.4: Skype vs. Non-Skype with Features Selected From First Three Levels of the Decision Tree

Features Used: Protocol, mean(BPP), mean(IPD), std(IPD), mean(BPF), med(BPF)

Classifier	**True Positive Rate**	**False Positive Rate**	**Precision**	**Overall Accuracy**
J48 Decision Tree	99.7%	0.1%	99.7%	99.8107%
IBK kNN, k=1	99.4%	0.3%	99.3%	99.5555%
Naïve Bayes	98.9%	50.7%	48.6%	65.4963%

Median of Packets Per Flow, Median of Bytes Per Packet. Table 5.3 shows the results of running the classifiers on the Skype vs. Non-Skype dataset with only these four features included. Due to the computational complexity of running the SVM, and due to its poor results, the SVM was omitted for the subsequent experiments. Since the results for the other classifiers are very similar to the results gained by using all of the features, using this feature subset is proven successful.

The second feature selection method used was based on the Decision Tree created during the first experiment using the Skype vs. Non-Skype dataset with all of the features included; this Decision Tree is shown in Appendix C. In order to find the relevant features, the features found in the first three levels of the Decision Tree were used, and included Protocol, Mean of Bytes Per Flow, Median of Bytes Per Flow, Mean of Bytes Per Packet, Standard Deviation of Inter-packet Delay, and Mean of Inter-packet Delay. The results of running the Skype vs. Non-Skype dataset with these features is shown in Table 5.4, and as with the other feature selection method, the results are very similar to the results with all features included, again verifying that this is a successful choice for feature selection.

Table 5.5: Skype vs. Non-Skype with Varying Time Window Values

Time Window	Classifier	True Positive Rate	False Positive Rate	Precision	Overall Accuracy
5 minutes	J48 Decision Tree	99.8%	0.4%	99.8%	99.7436%
	IBK kNN, k=1	99.8%	0.9%	99.6%	99.6011%
	Naïve Bayes	98.2%	46.7%	82.6%	84.371%
2 minutes	J48 Decision Tree	98.8%	0.3%	98.5%	99.5564%
	IBK kNN, k=1	96.4%	1.2%	94.3%	98.4368%
	Naïve Bayes	95.8%	67.1%	22.0%	43.3038%
60 seconds	J48 Decision Tree	-	-	-	-
	IBK kNN, k=1	-	-	-	-
	Naïve Bayes	-	-	-	-

5.2.3 Time Window Selection

Another property of interest for classification was the time window used to extract the features. Since we were interested in long-lived connections, the natural question was, how long is long-lived? Since we wanted to determine whether our time window was an appropriate choice, we first extracted features from a subset of the data with the time window set to five minutes, and then ran the classifiers again. The results are shown in Table 5.5, and are very similar to the results for the one hour time window, so it seems the selection for the time window may have been too generous. We wanted to test the limit to the size of the time window, so we reduced it further, this time to two minutes, and ran the classifiers again. The results in Table 5.5 again indicate that since the results are very similar to those with the one hour time window that a time window of as little as two minutes captures enough behaviors of the Skype control flows to be able to classify successfully. We then reduced the time window even further, to sixty seconds, but this time the feature extraction resulted in no meaningful data. This is likely due to the fact that according to our analysis of the Skype control flows, the time interval between control flows is between 1.5 to 2 minutes. Therefore, by setting the feature extraction time window too small to capture these flows, our statistical model cannot effectively capture the control flow behavior at this time win-

dow. A time window of two minutes therefore seems to be the optimal value with which to do classification of Skype control flows.

5.3 Discussion

The high accuracy rates for the the Decision Tree and k-Nearest Neighbor classifiers are indicative that our statistical model is robust enough to effectively identify Skype control flows. This is essential to our goal of creating a statistical model for Skype control flows, since the high accuracy rates indicate that our statistical model is effective in identifying Skype. The false positive rates for the Decision Tree and k-Nearest Neighbor classifiers is also particularly reassuring because it shows that Skype can be very accurately classified using our statistical model, but is also not often confused with the other P2P applications in the dataset. This is important since our overall goal is to create a statistical model for the Skype control flows, and a low false positive rate ensures that if a statistical model such as this one was used for detecting future Skype control flows in the case of a firewall or IPS, that other applications would not be misclassified as Skype and therefore allowed to continue potentially malicious behavior. Also, since the Decision Tree performs well for the other P2P applications on the multi-labeled dataset, this indicates that creating statistical model for other P2P applications should be possible.

A small subset of the features is also effective for identifying the Skype control flows, and a time window of 2 minutes is sufficient to accurately identify them as well. These values could be optimized further, but this is left to future work. Also left to future work is the issue of why Naïve Bayes and the SVM performed so poorly with these datasets. Since their accuracy was far lower than the other classifiers, perhaps measures can be taken to adjust the parameters of these two classifiers to produce better results with these datasets.

Chapter 6

Conclusion

In this paper we have presented a classification scheme for identifying Skype overlay control flows in a network trace. By constructing a dataset containing real-world Skype traces, we were able to analyze the behavioral features of the Skype control flows and create a statistical model for the flow behaviors. Using this statistical model, we created a dataset which could be used to perform classification of these control flows. In order to ensure the robustness of the statistical model, we also created a dataset containing real-world P2P file-sharing traces, so that we could verify that the statistical model for the Skype control flows was particular to the Skype application. During classification, we employed Decision Trees, Naïve Bayes, k-Nearest Neighbor, and Support Vector Machines to determine how effective our statistical model was with respect to identifying the Skype control flows against other P2P control flows. We showed very accurate results, with a 99.87% accuracy and 0.2% false positive rate, indicating that the statistical model created here effectively captures the behavior of the Skype overlay control flows.

Since this statistical model has proven effective with Skype overlay control flows, the likelihood of creating similar models for other P2P applications seems promising. This makes the task of identifying known valid P2P applications a tangible reality, and presents the first

step towards using identification of known P2P applications to disallow potentially malicious P2P traffic, specifically traffic generated by P2P botnets. This type of identification method can also be used to perform a behavioral comparison between P2P botnet network activity and valid P2P network activity, in order to more accurately identify P2P botnet channels, but this is left to future work.

Bibliography

[1] http://www.iana.org.

[2] http://www.cs.waikato.ac.nz/ml/weka/.

[3] http://www.autoitscript.com/site/autoit/.

[4] http://arxiv.org.

[5] http://www.qosient.com/argus/.

[6] http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

[7] http://www.skype.com.

[8] http://www.utorrent.com.

[9] http://www.vuze.com.

[10] http://www.limewire.com.

[11] http://www.frostwire.com.

[12] http://www.emule-project.net.

[13] Alshammari, R., and Zincir-Heywood, A. N. Machine learning based encryption traffic classification: Identifying ssh and skype. In *CISDA '09: Proceedings of the Second*

IEEE International Conference on Computational Intelligence for Security and Defense Applications (Piscataway, New Jersey, 2009).

[14] Angevine, D., and Zincir-Heywood, A. N. A preliminary investigation of skype traffic classification using a minimalist feature set. In *ARES '08: Proceedings of the Third International Conference on Availability, Reliability and Security* (Washington, DC, 2008).

[15] Baset, S. A., and Schulzrinne, H. An analysis of the skype peer-to-peer internet telephony protocol. In *IEEE Infocom '06* (Barcelona, Spain, April 2006).

[16] Biondi, P., and Desclaux, F. Silver needle in the skype.

[17] Bonfiglio, D., Mellia, M., Meo, M., and Ritacca, N. Tracking down skype traffic. In *IEEE INFOCOM '08* (Phoenix, Arizona, April 2008).

[18] Bonfiglio, D., Mellia, M., Meo, M., and Rossi, D. Detailed analysis of skype traffic. *IEEE Transactions on Multimedia 11*, 1 (January 2009).

[19] Bonfiglio, D., Mellia, M., Meo, M., Rossi, D., and Tofanelli, P. Revealing skype traffic: When randomness plays with you. In *ACM SIGCOMM '07* (Kyoto, Japan, August 2006).

[20] Erman, J., Mahanti, A., Arlitt, M., Cohen, I., and Williamson, C. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation 64* (October 2007).

[21] Freire, E., Ziviani, A., and Salles, R. Detecting skype flows in web traffic. In *IEEE Network Operations and Management Symposium* (August 2008).

[22] Grizzard, J. B., Sharma, V., Nunnery, C., Kang, B. B., and Dagon, D. Peer-to-peer botnets: Overview and case study. In *USENIX HotBots* (2007).

[23] Gu, G., Perdisci, R., Zhang, J., and Lee, W. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security* (2008).

[24] Guha, S., Daswani, N., and Jain, R. An experimental study of the skype peer-to-peer voip system. In *5th International Workshop on Peer-to-Peer Systems* (Santa Barbara, California, Feburary 2006).

[25] Haq, I. U., Ali, S., Khan, H., and Khayam, S. A. What is the impact of p2p traffic on anomaly detection? *Recent Advances in Intrusion Detection: Lecture Notes in Computer Science 6307/2010* (2010), 1–17.

[26] Holz, T., Steine, M., Dahl, F., Biersack, E., and Freiling, F. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. In *USENIX LEET* (2008).

[27] Hu, Y., Chiu, D.-M., and Lui, J. C. Profiling and identification of p2p traffic. *Computer Networks 53* (2009), 849863.

[28] Karagiannis, T., Broido, A., Faloutsos, M., and Claffy, K. Transport layer identification of p2p traffic. In *IMC'04* (Taormina, Sicily, Italy, October 2004).

[29] Karagiannis, T., Papagiannaki, K., and Faloutsos, M. Blinc: Multilevel traffic classification in the dark. In *SIGCOMM05* (Philadelphia, Pennsylvania, August 2005).

[30] Li, W., Canini, M., Moore, A. W., and Bolla, R. Efficient application identification and the temporal and spatial stability of classification schema. *Computer Networks 53* (2009), 790809.

[31] Madhukar, A., and Williamson, C. A longitudinal study of p2p traffic classification. In *MASCOTS06* (Monterey, California, August 2006).

[32] Mitchell, T. M. *Machine Learning*. WCB/McGraw-Hill, 1997.

[33] Moore, A. W., and Papagiannaki, K. Toward the accurate identification of network applications. In *PAM05* (Boston, Massachusetts, 2005).

[34] Nappa, A., Fattori, A., Balduzzi, M., DellAmico, M., and Cavallaro, L. Take a deep breath: a stealthy, resilient and cost-effective botnet using skype. In *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment* (2010), DIMVA'10.

[35] Rossi, D., Mellia, M., and Meo, M. Following skype signaling footsteps. In *IEEE Telecommunication Networking Workshop on QoS in Multiservice IP Networks* (April 2008), IT-NEWS.

[36] Rossi, D., Mellia, M., and Meo, M. Understanding skype signaling. *Computer Networks* (November 2008).

[37] Roughan, M., Sen, S., Spatscheck, O., and Duffield, N. Class-of-service mapping for qos: A statistical signature-based approach to ip traffic classification. In *IMC04* (Taormina, Sicily, Italy, October 2004).

[38] Sen, S., Spatscheck, O., and Wang, D. Accurate, scalable in-network identification of p2p traffic using application signatures. In *WWW2004* (New York, New York, May 2004).

[39] Suh, K., Figueiredo, D., Kurose, J., and Towsley, D. Characterizing and detecting relayed traffic: A case study using skype. In *IEEE INFOCOM '06* (Barcelona, Spain, April 2006).

[40] Williams, N., Zander, S., and Armitage, G. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. *ACM SIGCOMM Computer Communication Review 36*, 5 (October 2006).

[41] Witten, I. H., and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques*, second ed. Elsevier Inc., 2005.

[42] Zander, S., Nguyen, T., and Armitage, G. Automated traffic classification and application identification using machine learning. In *LCN05* (Sydney, Australia, November 2005).

[43] Zhang, J., Perdisci, R., Lee, W., Sarfraz, U., and Luo, X. Detecting stealthy p2p botnets using statistical traffic fingerprints. In *DSN-DCCS '11* (Hong Kong, China, June 2011).

Appendix A

README for Skype Dataset

```
*****Skype client information*****
Client login names:
superlinux_1
superlinux_2
reglinux_1
win7box_1
win7box_2
win7box_3
win7box_4
xpbox_1
xpbox_2
offcampusbox_1

Password for all clients is Ugalab537
All clients registered to cbmeyer@uga.edu

*****Hardware info*****
superlinux boxes both have Intel i7 vPro QuadCore 2.80GHz processors,
8GB of RAM
All 4 win7 boxes and reglinux have Intel Core2 Duo 2.13GHz processors,
2GB of RAM
Both xp boxes have Intel Pentium 4 3.00GHz processors, 1GB of RAM

*****IP configuration*****
natbox 128.192.76.184
```

```
reglinux_1 192.168.0.4
win7box_1 192.168.2.2
win7box_2 192.168.1.2
win7box_3 192.168.3.2
win7box_4 192.168.4.2
xpbox_1 192.168.6.2
xpbox_2 192.168.5.2

superlinux_1 128.192.76.181
superlinux_2 128.192.76.182

offcampusbox_1 97.81.96.137

Each box in the NAT is on its own subnet to prevent communication
between the clients behind the NAT.

*****iptables rules for natbox*****
iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
iptables -A FORWARD -s 192.168.0.0/16 -d 192.168.0.0/16 -j DROP

Additionally, the eth0 interface is multi-homed to provide
the default gateway for each of the NATed boxes:
eth1 128.192.76.184
eth0 192.168.0.1
eth0:1 192.168.1.1
eth0:2 192.168.2.1
eth0:3 192.168.3.1
eth0:4 192.168.4.1
eth0:5 192.168.5.1
eth0:6 192.168.6.1

*****tcpdump script*****
Running on superlinux_1, superlinux_2, and the NAT box, scheduled as a
cron job to run at midnight every day.

#! /bin/bash
DATE=$(date +"%Y%m%d")

/usr/bin/killall tcpdump
/usr/bin/nohup /usr/sbin/tcpdump -i eth0 -n -nn -X -s 0 -w $DATE.pcap &

*****ping daemon*****
```

Running on superlinux_2 to determine network availability/latency.
Started 2/17/2011.

```
#! /bin/bash
while [ 1 ]
do
ping -c 5 8.8.8.8
sleep 10
done
```

*****AutoIt script configurations*****
Calling boxes:
IMPORTANT NOTE - Make sure to enable hotkeys in Skype
(Tools->Options->Advanced->Hotkeys) for use with the calling scripts.

win7box_1 Upon initiation of the script, enters an infinite loop in
which it places a call for 10 minutes then ends the call,
waits for 2 hours 5 minutes, places a call for 5 minutes
then ends the call, waits for 2 hours 10 minutes, and
repeats. The reason for the additional time in the 2 hour
waiting period is to avoid collisions with the other box
making calls. By waiting during the duration of the other
box's calls, collisions can be avoided.

xpbox_1 Upon initiation of the script, waits for 1 hour 10 minutes,
then enters an infinite loop in which it places a call
for 5 minutes then ends the call, waits for 2 hours 10 minutes,
places a call for 10 minutes then ends the call, waits for
2 hours 5 minutes, and repeats. The collision avoidance
measures are present in the wait times for this script as
in the other so that they work in concert.

Both calling boxes also have microphones hooked up that are positioned
under the speaker on the underside of the machine. The receiving box
off campus will play an iTunes radio feed to simulate voice data over
the Skype network, so the microphones are positioned to pick up the
radio feed so that it will be played back to the receiving box.

Receiving box:
offcampusbox_1 Requires iTunes to run alongside Skype,
specifically with iTunes radio open. Any talk radio station
will do. The script waits in background for an incoming call,

then answers the call and presses play on iTunes radio to start
a radio feed. It waits 10 minutes then presses stop on iTunes
to stop the radio feed.

IM boxes:
win7box_2 Upon initiation of the script, enters an infinite loop in
which it sends a short message, waits 10 minutes, sends a long
message, waits 2 hours, sends a long message, waits 1 hour,
sends a short message, waits 2 hours, and repeats. The timing
measures in this script are designed to be complimentary to
the script on win7box_2, so that they communicate in a
back-and-forth manner.

win7box_3 Upon initiation of the script, enters an infinite loop in
which it sends a short message, waits 10 minutes, sends a long
message, waits 2 hours, sends a long message, waits 1 hour,
sends a short message, waits 2 hours, and repeats. The timing
measures in this script are designed to be complimentary to
the script on win7box_3, so that they communicate in a
back-and-forth manner.

*****Notes*****
1/31/2011-2/1/2011
Forgot to disable auto-update for reglinux_1
- gvfsd-http 91.189.89.31, 91.189.89.106, gnome vfs daemon
- fixed 2:25PM 2/1/2011

2/10/2011
Network failure at offcampusbox_1 from 2:30AM-2:30PM

2/14/2011
Campus network failure from 10:30AM-11:00AM
Slow/stopped again 2:00PM

2/15/2011
Campus network slow/stopped again 2:00PM

2/17/2011
Working on scripts
Ping daemon initiated
tcpdump cron job initiated

2/18/2011-3/21-2011
win7box_2 sending IMs to wrong client, win7box_4 instead of win7box_3

3/28/2011-3/30/2011
win7box_2 offline as of 4:21, CAT5 cord slightly pulled out
fixed 3/30 @ 1:25
in the interim, win7box_3 sending instant messages to win7box_4

3/31/2011-4/1/2011
superlinux_1 offline, possible dhcp failure?
noticed 9:15AM, fixed 9:35AM

4/5/2011-4/6/2011
superlinux_2 offline, same issue as with superlinux_1
from behavior of win7box_2, offline from 1:18PM on 4/5,
fixed 11:16AM 4/6
due to superlinux_2 failure, win7box_2 sending instant messages
to win7box_4

Appendix B

README for P2P File-Sharing Dataset

```
*****Hardware info*****
All 4 boxes have Intel Core2 Duo 2.13GHz processors, 2GB of RAM

*****IP configuration*****
natbox 128.192.76.183
p2pbox_1 192.168.1.2
p2pbox_2 192.168.2.2
p2pbox_3 192.168.3.2
p2pbox_4 192.168.4.2

Each box in the NAT is on it's own subnet to prevent communication
between the clients behind the NAT.

**LOGIN FOR p2pnatbox is natbox

*****iptables rules for natbox*****
**for eMule/FrostWire setup**
*nat
:PREROUTING ACCEPT [2607038:331546498]
:OUTPUT ACCEPT [40:2720]
:POSTROUTING ACCEPT [0:0]
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p tcp --dport 13000 -j DNAT
```

```
--to-destination 192.168.1.2:13000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p udp --dport 31000 -j DNAT
--to-destination 192.168.1.2:31000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p udp --dport 39000 -j DNAT
--to-destination 192.168.2.2:39000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p tcp --dport 39000 -j DNAT
--to-destination 192.168.2.2:39000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p tcp --dport 15000 -j DNAT
--to-destination 192.168.3.2:15000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p udp --dport 51000 -j DNAT
--to-destination 192.168.3.2:51000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p udp --dport 49000 -j DNAT
--to-destination 192.168.4.2:49000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p tcp --dport 49000 -j DNAT
--to-destination 192.168.4.2:49000
-A POSTROUTING -s 192.168.4.2/32 -p tcp --sport 49000 -j SNAT
--to-source 128.192.76.183:49000
-A POSTROUTING -s 192.168.4.2/32 -p udp --sport 49000 -j SNAT
--to-source 128.192.76.183:49000
-A POSTROUTING -s 192.168.3.2/32 -p tcp --sport 15000 -j SNAT
--to-source 128.192.76.183:15000
-A POSTROUTING -s 192.168.3.2/32 -p udp --sport 51000 -j SNAT
--to-source 128.192.76.183:51000
-A POSTROUTING -s 192.168.2.2/32 -p tcp --sport 39000 -j SNAT
--to-source 128.192.76.183:39000
-A POSTROUTING -s 192.168.2.2/32 -p udp --sport 39000 -j SNAT
--to-source 128.192.76.183:39000
-A POSTROUTING -s 192.168.1.2/32 -p tcp --sport 13000 -j SNAT
--to-source 128.192.76.183:13000
-A POSTROUTING -s 192.168.1.2/32 -p udp --sport 31000 -j SNAT
--to-source 128.192.76.183:31000
#
-A POSTROUTING -s 192.168.0.0/16 -j MASQUERADE
# -A POSTROUTING -o eth0 -j MASQUERADE

**for uTorrent/Vuze setup**
*nat
:PREROUTING ACCEPT [2607038:331546498]
:OUTPUT ACCEPT [40:2720]
:POSTROUTING ACCEPT [0:0]
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p tcp --dport 21000 -j DNAT
--to-destination 192.168.1.2:21000
```

```
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p udp --dport 21000 -j DNAT
--to-destination 192.168.1.2:21000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p udp --dport 32000 -j DNAT
--to-destination 192.168.2.2:32000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p tcp --dport 32000 -j DNAT
--to-destination 192.168.2.2:32000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p tcp --dport 41000 -j DNAT
--to-destination 192.168.3.2:41000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p udp --dport 41000 -j DNAT
--to-destination 192.168.3.2:41000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p udp --dport 52000 -j DNAT
--to-destination 192.168.4.2:52000
-A PREROUTING -d 128.192.76.183/32 -i eth0 -p tcp --dport 52000 -j DNAT
--to-destination 192.168.4.2:52000
-A POSTROUTING -s 192.168.4.2/32 -p tcp --sport 52000 -j SNAT
--to-source 128.192.76.183:52000
-A POSTROUTING -s 192.168.4.2/32 -p udp --sport 52000 -j SNAT
--to-source 128.192.76.183:52000
-A POSTROUTING -s 192.168.3.2/32 -p tcp --sport 41000 -j SNAT
--to-source 128.192.76.183:41000
-A POSTROUTING -s 192.168.3.2/32 -p udp --sport 41000 -j SNAT
--to-source 128.192.76.183:41000
-A POSTROUTING -s 192.168.2.2/32 -p tcp --sport 32000 -j SNAT
--to-source 128.192.76.183:32000
-A POSTROUTING -s 192.168.2.2/32 -p udp --sport 32000 -j SNAT
--to-source 128.192.76.183:32000
-A POSTROUTING -s 192.168.1.2/32 -p tcp --sport 21000 -j SNAT
--to-source 128.192.76.183:21000
-A POSTROUTING -s 192.168.1.2/32 -p udp --sport 21000 -j SNAT
--to-source 128.192.76.183:21000
#
-A POSTROUTING -s 192.168.0.0/16 -j MASQUERADE
# -A POSTROUTING -o eth0 -j MASQUERADE

Additionally, the eth0 interface is multi-homed to provide the default
gateway for each of the NATed boxes:
eth0 128.192.76.183
eth0:0 192.168.1.1
eth0:1 192.168.2.1
eth0:2 192.168.3.1
eth0:3 192.168.4.1
```

```
*****tcpdump script*****
Running on natbox

#! /bin/bash

DATE=$(/bin/date +"%Y%m%d%k")
LAST=$(/bin/cat /home/natbox/last.txt)
/usr/bin/killall tcpdump
/usr/bin/nohup /usr/sbin/tcpdump -i eth0 -n -nn -X -s 0 not tcp port 22
-w public_ip.$DATE.pcap &
/usr/bin/nohup /usr/sbin/tcpdump -i eth0:0 -n -nn -X -s 0 not tcp port 22
-w p2pbox1.$DATE.pcap &
/usr/bin/nohup /usr/sbin/tcpdump -i eth0:1 -n -nn -X -s 0 not tcp port 22
-w p2pbox2.$DATE.pcap &
/usr/bin/nohup /usr/sbin/tcpdump -i eth0:2 -n -nn -X -s 0 not tcp port 22
-w p2pbox3.$DATE.pcap &
/usr/bin/nohup /usr/sbin/tcpdump -i eth0:3 -n -nn -X -s 0 not tcp port 22
-w p2pbox4.$DATE.pcap &

/usr/bin/scp /home/natbox/p2pbox*.$LAST.pcap brett@128.192.76.184:~/
/usr/bin/scp /home/natbox/public_ip.$LAST.pcap brett@128.192.76.184:~/

/bin/rm /home/natbox/p2pbox*.$LAST.pcap
/bin/rm /home/natbox/public_ip.$LAST.pcap

/bin/echo $DATE > /home/natbox/last.txt

*****AutoIt script configuration*****
For each the P2P clients, the clients must be configured to use different
ports than the ports used for the other applications.

***eMule***
Run on boxes 192.168.1.2 & 192.168.3.2
The script is named emule, and it searches for the following open source
software on the Kad network, and downloads it:
Ubuntu 10.10
Ubuntu 10.10 server
Ubuntu 10.04
Ubuntu 9.10
Ubuntu 8.04
Ubuntu 7.10
Fedora 10
```

Fedora 11
Fedora 12
Fedora 13
Fedora 14
Debian
Open Office

Under "Options->Connection" the "Capacities" fields were set to 1000KB/s for Download and 512KB/s for Upload.

To use the script, under the "Search" tab, the "Type" field must be specified as "CD-Image" so that only the relevant files being searched for are returned.

Before the script is started, the "Transfers" tab must be selected.

uTorrent
Run on boxes 192.168.1.2 & 192.168.3.2
The script is named "utest", and is configured with the URLs of the torrents to be downloaded, and the URLs are from BTJunkie and The Pirate Bay.

Vuze
Run on boxes 192.168.2.2 & 192.168.4.2
The script is named "vuze", Same script configuration as with uTorrent, but with semantics of Vuze.

FrostWire
Run on boxes 192.168.2.2 & 192.168.4.2
AutoIt script will not interact with the GUI. Must do downloads manually.
Chose a variety of .avi files to download due to size and availability.
Mostly chose search terms "glee", "greys anatomy", and "house".

"Fake" files are served to other users in
C:\Documents and Settings\Administrator\My Documents\FrostWire\Shared
To share files, go to "Library" tab, "Shared" under "Shared Files", right click each file and select "Share File"

Scheduled Task running on boxes 2 & 4, runs at 12:00AM every day.
Administrator password: p2pbox
rmstuff.bat
del "C:\Documents and Settings\Administrator\My Documents\FrostWire\Saved*"

*****Notes*****

3/2/2011
Problems with firewall rules began.

3/26/2011
eMule and FrostWire data collection began

4/3/2011
eMule and FrostWire data collection ended ~10:56AM
uTorrent and Vuze collection began ~12:00PM

4/4/2011-4/5/2011
p2pnatbox not sending data to natbox, no space left on device
began 2:00PM on 4/4, fixed 2:00PM 4/5

4/11/2011
uTorrent and Vuze collection ended ~12:00PM

Appendix C

Decision Tree Created From J48

Decision Tree Classifier

```
J48 pruned tree
------------------

proto = tcp
|  meanbpf <= 1522.5
|  |  meanipd <= 101.139433
|  |  |  fph <= 41
|  |  |  |  stdipd <= 56.83446
|  |  |  |  |  medbpp <= 54.5: 1 (35290.0)
|  |  |  |  |  medbpp > 54.5
|  |  |  |  |  |  stdbpp <= 23.084597: -1 (12.0)
|  |  |  |  |  |  stdbpp > 23.084597: 1 (44.0)
|  |  |  |  stdipd > 56.83446
|  |  |  |  |  stdbpp <= 6.462752
|  |  |  |  |  |  meanbpp <= 47.818182
|  |  |  |  |  |  |  stdppf <= 2.484619: 1 (4.0)
|  |  |  |  |  |  |  stdppf > 2.484619: -1 (2.0)
|  |  |  |  |  |  meanbpp > 47.818182: -1 (108.0)
|  |  |  |  |  stdbpp > 6.462752
|  |  |  |  |  |  stdipd <= 248.27924
|  |  |  |  |  |  |  medbpp <= 57.5
|  |  |  |  |  |  |  |  medbpp <= 47: 1 (22.0)
```

```
|  |  |  |  |  |  |  |  medbpp > 47
|  |  |  |  |  |  |  |  |  meanipd <= 82.63717
|  |  |  |  |  |  |  |  |  |  stdbpp <= 156.07976
|  |  |  |  |  |  |  |  |  |  |  medipd <= 0.478459: 1 (30.0)
|  |  |  |  |  |  |  |  |  |  |  medipd > 0.478459
|  |  |  |  |  |  |  |  |  |  |  |  medbpp <= 52.5
|  |  |  |  |  |  |  |  |  |  |  |  |  stdppf <= 6.898683
|  |  |  |  |  |  |  |  |  |  |  |  |  |  meanbpf <= 329.833333: 1 (4.0/1.0)
|  |  |  |  |  |  |  |  |  |  |  |  |  |  meanbpf > 329.833333: -1 (11.0)
|  |  |  |  |  |  |  |  |  |  |  |  |  stdppf > 6.898683: 1 (4.0)
|  |  |  |  |  |  |  |  |  |  |  |  medbpp > 52.5: 1 (10.0)
|  |  |  |  |  |  |  |  |  |  stdbpp > 156.07976: -1 (6.0)
|  |  |  |  |  |  |  |  |  meanipd > 82.63717: -1 (16.0)
|  |  |  |  |  |  |  |  medbpp > 57.5: -1 (17.0)
|  |  |  |  |  |  |  stdipd > 248.27924
|  |  |  |  |  |  |  |  medppf <= 12.5: 1 (193.0/1.0)
|  |  |  |  |  |  |  |  medppf > 12.5
|  |  |  |  |  |  |  |  |  stdppf <= 6.22703: 1 (9.0)
|  |  |  |  |  |  |  |  |  stdppf > 6.22703: -1 (2.0)
|  |  |  fph > 41
|  |  |  |  medppf <= 10.5
|  |  |  |  |  stdbpp <= 21.821075: -1 (783.0)
|  |  |  |  |  stdbpp > 21.821075
|  |  |  |  |  |  fph <= 56
|  |  |  |  |  |  |  stdipd <= 64.398123: 1 (10.0)
|  |  |  |  |  |  |  stdipd > 64.398123: -1 (6.0)
|  |  |  |  |  |  fph > 56: -1 (61.0)
|  |  |  |  medppf > 10.5
|  |  |  |  |  medbpp <= 70
|  |  |  |  |  |  fph <= 79: 1 (24.0)
|  |  |  |  |  |  fph > 79: -1 (2.0)
|  |  |  |  |  medbpp > 70: -1 (41.0)
|  |  meanipd > 101.139433
|  |  |  stdbpp <= 8.202697
|  |  |  |  medbpp <= 59: -1 (2790.0/1.0)
|  |  |  |  medbpp > 59
|  |  |  |  |  medbpp <= 61
|  |  |  |  |  |  medppf <= 6: 1 (3.0)
|  |  |  |  |  |  medppf > 6: -1 (2.0)
|  |  |  |  |  medbpp > 61: -1 (59.0)
|  |  |  stdbpp > 8.202697
|  |  |  |  medipd <= 29.50867: -1 (109.0)
```

```
|  |  |  |  medipd > 29.50867
|  |  |  |  |  meanbpp <= 76.405405
|  |  |  |  |  |  medipd <= 120.504646
|  |  |  |  |  |  |  fph <= 21: 1 (130.0)
|  |  |  |  |  |  |  fph > 21
|  |  |  |  |  |  |  |  medppf <= 6: 1 (21.0)
|  |  |  |  |  |  |  |  medppf > 6: -1 (4.0)
|  |  |  |  |  |  medipd > 120.504646: -1 (9.0/1.0)
|  |  |  |  |  meanbpp > 76.405405: -1 (35.0)
|  meanbpf > 1522.5
|  |  medbpf <= 1322
|  |  |  medppf <= 10
|  |  |  |  meanbpf <= 1652.9625
|  |  |  |  |  stdppf <= 14.159251: -1 (18.0)
|  |  |  |  |  stdppf > 14.159251: 1 (13.0)
|  |  |  |  meanbpf > 1652.9625
|  |  |  |  |  fph <= 35
|  |  |  |  |  |  meanipd <= 32.22501
|  |  |  |  |  |  |  meanbpf <= 4135.736434: 1 (8.0)
|  |  |  |  |  |  |  meanbpf > 4135.736434: -1 (18.0)
|  |  |  |  |  |  meanipd > 32.22501: -1 (89.0)
|  |  |  |  |  fph > 35: -1 (311.0)
|  |  |  medppf > 10
|  |  |  |  medbpf <= 743: 1 (62.0/1.0)
|  |  |  |  medbpf > 743
|  |  |  |  |  medppf <= 17.5
|  |  |  |  |  |  meanbpf <= 3188.792553
|  |  |  |  |  |  |  meanppf <= 17.405941: -1 (9.0)
|  |  |  |  |  |  |  meanppf > 17.405941: 1 (7.0)
|  |  |  |  |  |  meanbpf > 3188.792553: -1 (67.0)
|  |  |  |  |  medppf > 17.5: 1 (5.0)
|  |  medbpf > 1322
|  |  |  stdbpf <= 10753.29333
|  |  |  |  meanbpp <= 75.182692
|  |  |  |  |  meanipd <= 5.599241
|  |  |  |  |  |  meanbpf <= 36359.7381
|  |  |  |  |  |  |  medbpp <= 62.5: 1 (15.0)
|  |  |  |  |  |  |  medbpp > 62.5: -1 (2.0)
|  |  |  |  |  |  meanbpf > 36359.7381: -1 (4.0)
|  |  |  |  |  meanipd > 5.599241
|  |  |  |  |  |  stdppf <= 0.934267
|  |  |  |  |  |  |  fph <= 6: 1 (4.0)
```

```
|  |  |  |  |  |  |  fph > 6: -1 (2.0)
|  |  |  |  |  |  stdppf > 0.934267
|  |  |  |  |  |  |  stdbpp <= 49.132382: -1 (94.0/1.0)
|  |  |  |  |  |  |  stdbpp > 49.132382
|  |  |  |  |  |  |  |  meanppf <= 12.666667: 1 (2.0)
|  |  |  |  |  |  |  |  meanppf > 12.666667: -1 (3.0/1.0)
|  |  |  |  meanbpp > 75.182692: -1 (1385.0)
|  |  |  stdbpf > 10753.29333: -1 (7003.0)
proto = udp
|  meanbpp <= 327.920635
|  |  stdipd <= 65.90275
|  |  |  medbpp <= 67
|  |  |  |  medbpf <= 245
|  |  |  |  |  fph <= 2
|  |  |  |  |  |  meanbpf <= 135.5: -1 (50.0)
|  |  |  |  |  |  meanbpf > 135.5
|  |  |  |  |  |  |  meanbpp <= 44: -1 (16.0)
|  |  |  |  |  |  |  meanbpp > 44
|  |  |  |  |  |  |  |  meanipd <= 3337.122679
|  |  |  |  |  |  |  |  |  stdbpp <= 1: 1 (35.0/1.0)
|  |  |  |  |  |  |  |  |  stdbpp > 1
|  |  |  |  |  |  |  |  |  |  meanbpp <= 60.744027: 1 (4.0)
|  |  |  |  |  |  |  |  |  |  meanbpp > 60.744027: -1 (2.0)
|  |  |  |  |  |  |  |  meanipd > 3337.122679
|  |  |  |  |  |  |  |  |  meanbpf <= 137.2: -1 (4.0)
|  |  |  |  |  |  |  |  |  meanbpf > 137.2
|  |  |  |  |  |  |  |  |  |  meanbpf <= 142.25: 1 (6.0)
|  |  |  |  |  |  |  |  |  |  meanbpf > 142.25
|  |  |  |  |  |  |  |  |  |  |  stdbpp <= 6.249865: -1 (5.0)
|  |  |  |  |  |  |  |  |  |  |  stdbpp > 6.249865: 1 (3.0/1.0)
|  |  |  |  |  fph > 2
|  |  |  |  |  |  medipd <= 849.98734
|  |  |  |  |  |  |  medbpf <= 182: -1 (340.0)
|  |  |  |  |  |  |  medbpf > 182
|  |  |  |  |  |  |  |  meanbpf <= 200.428571: -1 (6.0)
|  |  |  |  |  |  |  |  meanbpf > 200.428571: 1 (31.0/1.0)
|  |  |  |  |  |  medipd > 849.98734: -1 (6944.0/2.0)
|  |  |  |  medbpf > 245
|  |  |  |  |  medbpf <= 259: 1 (139.0)
|  |  |  |  |  medbpf > 259
|  |  |  |  |  |  stdbpf <= 2.357023: 1 (21.0)
|  |  |  |  |  |  stdbpf > 2.357023
```

```
|  |  |  |  |  |  |  meanipd <= 20.964318
|  |  |  |  |  |  |  |  medipd <= 18.032503: -1 (24.0)
|  |  |  |  |  |  |  |  medipd > 18.032503: 1 (40.0)
|  |  |  |  |  |  |  meanipd > 20.964318
|  |  |  |  |  |  |  |  fph <= 3
|  |  |  |  |  |  |  |  |  meanppf <= 2.25
|  |  |  |  |  |  |  |  |  |  meanbpf <= 303: -1 (3.0)
|  |  |  |  |  |  |  |  |  |  meanbpf > 303: 1 (19.0)
|  |  |  |  |  |  |  |  |  meanppf > 2.25: -1 (2.0)
|  |  |  |  |  |  |  |  fph > 3: -1 (264.0/2.0)
|  |  |  medbpp > 67
|  |  |  |  stdipd <= 5.700382
|  |  |  |  |  fph <= 2
|  |  |  |  |  |  meanbpf <= 256
|  |  |  |  |  |  |  medbpp <= 146: -1 (465.0)
|  |  |  |  |  |  |  medbpp > 146
|  |  |  |  |  |  |  |  meanbpf <= 244: -1 (17.0)
|  |  |  |  |  |  |  |  meanbpf > 244: 1 (8.0/1.0)
|  |  |  |  |  |  meanbpf > 256: -1 (2240.0)
|  |  |  |  |  fph > 2: -1 (35238.0)
|  |  |  |  stdipd > 5.700382
|  |  |  |  |  fph <= 6
|  |  |  |  |  |  meanipd <= 849.037902
|  |  |  |  |  |  |  medbpp <= 126: -1 (111.0)
|  |  |  |  |  |  |  medbpp > 126
|  |  |  |  |  |  |  |  medbpf <= 294.5: 1 (99.0)
|  |  |  |  |  |  |  |  medbpf > 294.5: -1 (13.0)
|  |  |  |  |  |  meanipd > 849.037902: -1 (1373.0/3.0)
|  |  |  |  |  fph > 6
|  |  |  |  |  |  stdbpp <= 11.228639: -1 (10355.0/1.0)
|  |  |  |  |  |  stdbpp > 11.228639
|  |  |  |  |  |  |  stdipd <= 22.105046
|  |  |  |  |  |  |  |  medbpf <= 361
|  |  |  |  |  |  |  |  |  meanipd <= 49.111879: 1 (36.0)
|  |  |  |  |  |  |  |  |  meanipd > 49.111879: -1 (18.0)
|  |  |  |  |  |  |  |  medbpf > 361: -1 (235.0)
|  |  |  |  |  |  |  stdipd > 22.105046: -1 (1428.0)
|  |  stdipd > 65.90275
|  |  |  fph <= 6
|  |  |  |  meanipd <= 1093.239499
|  |  |  |  |  stdbpp <= 4.066442
|  |  |  |  |  |  medipd <= 659.451454
```

```
|  |  |  |  |  |  |  meanppf <= 2.083333
|  |  |  |  |  |  |  |  meanbpf <= 217.055556
|  |  |  |  |  |  |  |  |  stdipd <= 77.574621
|  |  |  |  |  |  |  |  |  |  meanbpf <= 192.5: -1 (2.0)
|  |  |  |  |  |  |  |  |  |  meanbpf > 192.5
|  |  |  |  |  |  |  |  |  |  |  stdipd <= 77.078171
|  |  |  |  |  |  |  |  |  |  |  |  medipd <= 603.224144: -1 (2.0)
|  |  |  |  |  |  |  |  |  |  |  |  medipd > 603.224144: 1 (34.0)
|  |  |  |  |  |  |  |  |  |  |  stdipd > 77.078171: -1 (2.0)
|  |  |  |  |  |  |  |  |  stdipd > 77.574621: 1 (143.0)
|  |  |  |  |  |  |  |  meanbpf > 217.055556: 1 (2469.0/1.0)
|  |  |  |  |  |  |  meanppf > 2.083333
|  |  |  |  |  |  |  |  meanppf <= 3: 1 (35.0)
|  |  |  |  |  |  |  |  meanppf > 3: -1 (3.0)
|  |  |  |  |  |  medipd > 659.451454
|  |  |  |  |  |  |  meanbpf <= 218
|  |  |  |  |  |  |  |  fph <= 5: 1 (2.0)
|  |  |  |  |  |  |  |  fph > 5
|  |  |  |  |  |  |  |  |  stdipd <= 86.207937: -1 (8.0)
|  |  |  |  |  |  |  |  |  stdipd > 86.207937: 1 (2.0)
|  |  |  |  |  |  |  meanbpf > 218: 1 (35.0/1.0)
|  |  |  |  |  stdbpp > 4.066442
|  |  |  |  |  |  stdbpp <= 4.964345
|  |  |  |  |  |  |  meanppf <= 11.25: 1 (2.0)
|  |  |  |  |  |  |  meanppf > 11.25: -1 (2.0)
|  |  |  |  |  |  stdbpp > 4.964345: -1 (21.0)
|  |  |  |  meanipd > 1093.239499
|  |  |  |  |  meanipd <= 1646.836216
|  |  |  |  |  |  medbpp <= 61: 1 (3.0)
|  |  |  |  |  |  medbpp > 61
|  |  |  |  |  |  |  medbpp <= 128: -1 (38.0)
|  |  |  |  |  |  |  medbpp > 128
|  |  |  |  |  |  |  |  medbpf <= 259.5: 1 (4.0)
|  |  |  |  |  |  |  |  medbpf > 259.5: -1 (10.0)
|  |  |  |  |  meanipd > 1646.836216: -1 (163.0)
|  |  |  fph > 6
|  |  |  |  stdbpf <= 8.430747
|  |  |  |  |  stdbpp <= 0.207512: -1 (589.0)
|  |  |  |  |  stdbpp > 0.207512
|  |  |  |  |  |  stdbpf <= 0: -1 (23.0)
|  |  |  |  |  |  stdbpf > 0
|  |  |  |  |  |  |  medbpf <= 192.5: -1 (11.0)
```

```
|  |  |  |  |  |  |  medbpf > 192.5
|  |  |  |  |  |  |  |  medbpf <= 344.5: 1 (31.0)
|  |  |  |  |  |  |  |  medbpf > 344.5: -1 (3.0)
|  |  |  |  stdbpf > 8.430747
|  |  |  |  |  meanbpp <= 51
|  |  |  |  |  |  medbpp <= 48.5: -1 (29.0)
|  |  |  |  |  |  medbpp > 48.5: 1 (7.0)
|  |  |  |  |  meanbpp > 51: -1 (10189.0/2.0)
|  meanbpp > 327.920635
|  |  meanbpp <= 328
|  |  |  meanipd <= 422.680863: 1 (1902.0)
|  |  |  meanipd > 422.680863
|  |  |  |  stdbpp <= 7.5: 1 (21.0)
|  |  |  |  stdbpp > 7.5: -1 (4.0)
|  |  meanbpp > 328
|  |  |  fph <= 2
|  |  |  |  meanbpp <= 605.552426
|  |  |  |  |  meanbpp <= 441.333333: -1 (41.0)
|  |  |  |  |  meanbpp > 441.333333: 1 (19.0)
|  |  |  |  meanbpp > 605.552426: -1 (84.0)
|  |  |  fph > 2: -1 (1228.0)

Number of Leaves  :  126

Size of the tree :  251
```