Tools, Data, and Flow Attributes for Understanding Network Traffic without Payload

Timothy Furlong

Supervisor: Prof. Paul Van Oorschot

April 20, 2007

Abstract

The classification of network traffic based on the application that generated it is a relatively new field; it appears to be feasible, but the available tools and data are not yet adequate to pursue it effectively. There has also been some attention paid in the last few years to doing this without payload information, i.e. based only on packet header information, due to challenges such as encryption and privacy issues. This thesis describes a new software tool for computing flow attributes, values derived from network traffic that can be used for classifying it, focused on flow attributes that can be computed in the absence of payload information. It is flexible and powerful, and can compute a wide range of measurements on network traffic. We also perform a qualitative evaluation of the capabilities of the tool and study the behaviour of some flow attributes.

Contents

List of Figures

List of Tables

Acknowledgements

I would first like to thank my supervisor, Prof. Paul Van Oorschot, for his support and guidance in this process.

I'd also like to thank those who influenced my thinking in this process: Prof. Anil Somayaji for the idea to focus on the representation and features as a critical component, Prof. Shirley Mills for support on the statistical side of things, Prof. Tony White for the suggestion of using queueing theory and generative models for application behaviours, Mike Collins for e-mail discussions on the philosophy of network traffic and an advance copy of his ESORICS paper, Prof. Fabian Monrose for an advance copy of his JMLR paper, and the rest of the Carleton Computer Security Lab, for feedback in the early stages of the work.

Also, I offer my respects to the authors of the many tools I used: CAIDA for the Coral-Reef suite, used to convert many of the esoteric data formats, Christian Kreibich for libnetdude, used to demultiplex flows, Evan Hughes for libqcap, used to preprocess trace files and reassemble IP packets, Shawn Ostermann for TCPtrace, used to profile TCP connections, and the TCPdump team for libpcap and tcpdump, the major library used for handling network trace files and the invaluable Swiss Army Knife of network traces; research would be far more difficult without those who have built and shared these tools, and I’ll consider myself fortunate if my contribution proves to be even a tenth as useful as the least of these.

Last but certainly not least, I want to thank my fianceé, Alka, for her moral support and for not sulking (too much) over time spent working on the thesis instead of with her.

Glossary and concept index

This glossary contains expansions of abbreviations and definitions of technical terms used in this thesis. It also functions as a concept index; where appropriate, the page number of the definition of the abbreviation or term within the body of the thesis is provided in parentheses, where one can find more discussion of the term and its context in this work. Also, for convenience, the reference citation of the document defining a protocol is given in the glossary entry for the protocol. Note that commonly-abbreviated terms are described under the heading of the abbreviation, rather than having an entry for the abbreviation and a separate one for the expansion. Terms or phrases used in glossary definitions which are themselves defined elsewhere in the glossary are noted by ‘q.v.’ (Latin *quod vide*, “which see”).

ACK The ‘acknowledgement’ flag in a TCP packet header; also, a TCP packet with the ACK flag set to 1.

ADU Application Data Unit, defined by Hernández-Campos et al. [HCNSJ05] as a unit of data corresponding to a request-response pair in a network flow, where the request is an uninterrupted stream of data from a client and a response is an uninterrupted stream of data from a server in response to a request. (p. 26)

aggregate see packet aggregate.

application behaviour Activities of a networked application that produce observable patterns in network traffic. Examples of application behaviours are file transfer (e.g. FTP, HTTP file download), chat (e.g. MSN Messenger, AIM, ICQ, Unix ‘talk’), and interactive command shell activity (e.g. telnet, SSH). (p. 36)

behavioural distortion An effect on a network flow that changes the values of flow attributes; specifically an effect not directly related to the application itself. For example, we consider fragmentation at the Ethernet maximum transmission unit limit of 1500 bytes to be a distortion, as it is an artefact of the transmission process unrelated to the application-layer requirements. (p. 38)

bidirectional inter-packet delay The length of time between two packets in a half-flow (q.v.). See also inter-packet delay, unidirectional inter-packet delay. (p. 47)

bulk data transfer Data transfer that attempts to send the data as quickly as the network will allow. (p. 36)

client One party in a “client-server” architecture, the one which creates the connection in order to make use of the server’s service. (p. 17)

client-server architecture A programming paradigm for networked applications whereby one part of the application takes the role of a “server” (q.v.) which provides a service, and the other part takes the role of a “client” that uses the service. (p. 17)

command shell A process that serves as an interface between a user terminal and the operating system. (p. 19)

command-shell interactive behaviour Network activity produced by a human on one node interacting with a command shell on another node. (p. 37)

cross-validation A technique from machine learning using multiple trials to estimate the real error rate of a classifier. In each trial, the classifier is trained and tested on disjoint sets of data, which helps avoid overfitting. (p. 12)

distortion see behavioural distortion.

explanatory variable A variable in a regression problem, also known as an independent variable or experimental variable, that explains changes in the response variable, and that is manipulated in an experiment. (p. 13)

feature In machine learning, some variable or value computed from a sample used to classify that sample. (p. 10)

feature vector An array or list of variable values associated with a data sample; used in classification. (p. 10)

flow see network flow.

flow attribute A measurement based on observations about a network flow, or derived from other such measurements. (p. 41)

forward direction One direction of a bidirectional flow defined to be “forward”; in this thesis, the direction from the client to the server. For non-client-server applications, an alternate criteria will need to be defined. (p. 40)

FTP File Transfer Protocol, defined in RFC959 [PR85]. (p. 19)

half-flow A unidirectional flow that is one side of a bidirectional network flow; e.g. a bidirectional HTTP flow between 10.0.0.1:35535 and 10.0.0.2:80 is made up of two half-flows, one from 10.0.0.1:35535 to 10.0.0.2:80, and the reverse half-flow from 10.0.0.2:80 to 10.0.0.1:35535. Note that a half-flow is semantically considered to be a network flow (q.v.) itself; the term "half-flow" is merely used to distinguish the unidirectional flows from the bidirectional flow. (p. 17)

header Information prepended to a data packet by a lower level protocol, to be consumed by the corresponding protocol instance on the remote node. For instance, an instance of the IP given a packet by TCP to be sent will prepend an IP header that will be read by IP instances on every node between source and destination in order to determine the next hop in the route.

host see network node.

HTTP HyperText Transfer Protocol, a protocol for the structured exchange of arbitrary data over a network connection that underlies the operation of the World Wide Web. Version 1.0 is defined in RFC1945 [BLFF96] and version 1.1 is defined in RFC2616 [FGM$^+$99]. (p. 20)

IETF Internet Engineering Task Force, not-for-profit body dedicated to "making the Internet work better"; their mission statement is documented in RFC3935 [Alv04]. (p. 19)

Internet As "an internet" or, more commonly, "an internetwork", a network of (often heterogeneous) networks; as "the Internet", a particular global internetwork known as the Internet.

IP Internet Protocol; a network-layer protocol for routing packets over a network, defined in RFC0791 [Pos81a].

ITU-T International Telecommunications Union - Telecommunication Standardization Sector.

inter-packet delay The length of time between two packets in a packet aggregate; specifically, the unidirectional or bidirectional inter-packet delay (q.v.). (p. 47)

keystroke The byte or bytes resulting from a user pressing a key on their keyboard, particularly such bytes being sent across a network.

linear regression A statistical technique that attempts to use a straight line to describe, based on a data sample, how a responding variable is affected by another factor or factors. (p. 14)

logistic regression A type of linear regression technique commonly used for the analysis of categorical data. (p. 12)

machine-driven interactive behaviour Interactive network activity between two programs with little or no human intervention. (p. 37)

MTU Maximum Transmission Unit: The largest packet size that a particular network protocol will allow. Often used in the context of Ethernet, which has an MTU of 1500.

natural context For a particular packet, the packet aggregate that has been defined to be particularly meaningful. In this thesis, unless otherwise noted, the natural context of a packet is the half-flow (q.v.) in which it is contained. (p. 45)

network address A data value used to identify a network node (q.v.) at the network layer, e.g. an Internet Protocol (IP) (q.v.) address. In this work, we deal mostly with IPv4 addresses.

network flow A packet aggregate (q.v.) generated by a single networked application; used in this thesis to mean an aggregation of packets using a single transport layer protocol (e.g. TCP or UDP) between two endpoints (host/port pairs). Flows can be unidirectional or bidirectional; a unidirectional flow contains packets from one host/port pair to the other, a bidirectional flow also contains packets in the reverse direction. (p. 17)

network host see network node.

network jitter Variations in the time taken for a packet to travel through a network from a certain source node to a certain destination node, usually due to changing congestion conditions in the network. (p. 17)

network latency The amount of time taken for a packet to travel through a network from a certain source node to a certain destination node. (p. 17)

network layer The layer of a layered protocol model responsible for routing a packet through an internetwork (q.v.). (p. 15)

network node Also “node”, “network host”, “host”. A computing device with network connectivity; in this thesis, a network node will typically refer to a personal computer or a server. (p. 15)

network session An episode of network activity generated by a networked application performing some task, and using the network to communicate with a single remote process. (p. 17)

network trace a recording of network data, e.g. as captured by an Ethernet card in promiscuous mode sniffing a local network.

network traffic Activity on a network, or packets sent across a network. (p. 16)

networked application A computer program or set of computer programs that communicate using a network to accomplish some goal. (p. 18)

NLANR National Laboratory for Applied Network Research (p. 141)

node see network node.

octet 8-bit group of data; equivalent to byte in most contexts. Often used in networking discussions as opposed to byte to avoid ambiguity with other sizes of bytes.

packet A sequence of data communicated over a network, e.g. a TCP packet or an Ethernet frame. Can be nested (e.g. an Ethernet frame carrying an IPv4 packet carrying a TCP packet) or fragmented (e.g. a TCP segment split up across multiple IPv4 packets). ‘Packet’ is used somewhat ambiguously both as a generic term as above, and more specifically to refer to data units from layer 3 and 4 protocols such as IP and TCP. (p. 16)

packet aggregate A sequence of packets, such as a network flow (q.v.). (p. 39)

payload The part of a packet which “belongs” to a higher-layer protocol than the protocol being discussed. For example, when discussing an IP packet that is part of a TCP connection, the payload is the data after the IP header, and contains the TCP header and application data. (p. 16)

peer-to-peer A type of network application architecture where many nodes in the network communicate with each other, rather than with a central server. Peer-to-peer architectures are considered only peripherally in this thesis. (p. 24)

POP3 Post Office Protocol, version 3 is a protocol designed to allow an end user to retrieve their e-mail from an e-mail server; it is defined in RFC1939 [MR96]. (p. 20)

port Integer value used to demultiplex incoming packets (q.v.) to the correct process on a host (q.v.). E.g. TCP port, UDP port. (p. 16)

POS Packet over SONET/SDH (q.v.), a physical layer protocol often used to transmit packets such as Ethernet frames over a fiber optic link.

regression A statistical technique for estimating, based on a data sample, the relationship between a dependent (or responding) variable and another factor or factors. See also linear regression, logistic regression. (p. 13)

response variable The variable in a regression, also known as the dependent variable, that is not controlled and is believed to be responding to changes in the explanatory variable(s). (p. 13)

reverse direction The direction in a bidirectional flow opposite the primary, or forward, direction. (p. 40)

RFC Request For Comments: one of a series of documents concerning the workings of the Internet, submitted to the IETF (q.v.) and made public for review and discussion. (p. 19)

sensor A network node (q.v.) that collects network traffic (q.v.) for analysis, often dedicated to this task.

server one party in a client-server architecture, the one which provides a service to be consumed by clients. (p. 17)

session see network session.

shell see command shell.

SMTP Simple Mail Transfer Protocol, a protocol for forwarding e-mail between e-mail servers, defined by RFC821 [Pos82]. (p. 20)

streaming media Streaming media applications are those which read some sort of media, usually audio or audio/video, remotely and display it as it is received, rather than where the user downloads the entire media file and displays it locally. Also streaming audio, streaming video. Streaming media is considered only peripherally in this thesis. (p. 24)

SONET/SDH Synchronous Optical NETwork / Synchronous Digital Hierarchy: a standard for transmitting data over a fiber optic link; defined by ANSI standard T1.105 and ITU-T standards G.707 and G.783.

TCP Transmission Control Protocol; a transport-layer protocol for reliable connections over an Internet Protocol (q.v.) network, defined in RFC793 [Pos81b].

Telnet Protocol providing an interface between terminal devices and terminal processes, defined in RFC854 [PR83]. (p. 19)

trace see network trace.

traffic see network traffic.

transport layer The layer of a layered protocol model responsible for communication between processes on communicating hosts. (p. 15)

UDP User Datagram Protocol; a transport-layer protocol for connectionless data transfer over an Internet Protocol (q.v.) network, defined in RFC768 [Pos80].

unidirectional inter-packet delay The time between two packets in a network flow (q.v.). See also inter-packet delay, bidirectional inter-packet delay. (p. 47)

Chapter 1

Introduction and overview

It is often useful to be able to identify the nature of a networked application by analyzing the network traffic that it generates. Some examples are enforcing network policy, detecting malicious activity, and improving quality-of-service. Recent work has shown the possibility of performing such classification without using the application-layer payload of the network traffic, but much more work can be done in making these approaches more practical. We have developed tools for measuring network traffic and dealing with network data in order to support this research.

Network traffic classification is useful in several contexts. As one example, some organizations disallow peer-to-peer file sharing; in order to enforce such a policy, the network administrators need some way to detect it, even though it may be disguised as web surfing or other traffic. Similarly, if a remote attacker is using the network to communicate with compromised hosts on the network, the administrators need some way to find that activity. Another example is enforcing quality-of-service; in a case where several distinct applications use the same port and protocol (for example, web surfing, streaming video, chat, and peer-to-peer file sharing, all over HTTP on port 80), it is useful to be able to distinguish between them in order to ensure that no one application consumes all the available bandwidth.

We are particularly interested in classifying network traffic without inspecting the application-layer payload data. Such data can be hidden by encryption or unavailable due to privacy and policy reasons; for research purposes in particular, it is difficult to obtain samples of network traffic that include the application-layer payload. There are, however, publically available data sets containing packet headers from a variety of contexts; such varied data would seem to be important for developing and testing general approaches. Another disadvantage to using approaches based on application-layer data is that there are often a variety of applications used for any given purpose, which can differ widely in the format of their payload data. We expect that measurements based on information such as packet lengths and inter-packet delays will better show general patterns of behaviour among applications serving the same basic purpose.

In this chapter, we explain the motivation behind our work, outline the structure of this thesis, and summarize our contributions to the field of network traffic classification.

1.1 Motivation

The approaches to network traffic classification in the literature generally attempt to determine from the network traffic what application generated that traffic. This work is motivated by a desire to see a shift towards the use of application behaviours, patterns in network traffic caused by particular uses of the network, as an intermediate step for classification. Our work focuses on developing tools in support of a shift to a behaviour-based traffic classification; here, we explain the reasoning behind this shift, the path that we think it should take, and why we have focused on tools to support it.

We see two reasons for studying application behaviours rather than applications: many groups of applications exhibit similar behaviours to one another, and some applications exhibit multiple distinct behaviours, even within a single flow. Both result in difficulties in classification; it is difficult to distinguish between two applications that behave similarly to one another, and it is difficult to characterize an application that behaves inconsistently. Hernández-Campos et al. [HCNSJ05] identified a need to study "the impact of common uses of [applications]" rather than specific applications, based on the observation of different applications using the network in similar manners (e.g. HTTP, FTP, and peer-to-peer file sharing all being used for unidirectional file transfer). Many applications are too complex to be easily expressed in terms of the network traffic that they generate (e.g. Telnet exhibiting distinct phases of command-shell interactive traffic and bulk data transfer); this is suggested by Nguyen and Armitage's [NA06] work on classifying game traffic using sub-flows (fragments of network flows) as well as by our own observations.

The concept of application behaviours could be used as an intermediate step between network traffic and applications. Application behaviours are basic types of activity such as bulk data transfer and command-shell interactive activity; they are more meaningful and identifiable at the level of network traffic than applications, which can be expressed in terms of these behaviours. For example, an application could be modelled as a state machine where each state is a behaviour that that application can produce.

Ultimately, we wish to be able to describe the activity of a networked application in terms of application behaviours. For example, consider Telnet, which is an application that allows a user to interact with a command shell on a remote server. In the simple case mentioned earlier, a Telnet session may consist of a user sending short commands and receiving short replies, with occasional commands that result large bursts of data being sent back to the user; we could model this as a finite state machine with two states, corresponding to application behaviours: command-shell interactive activity and bulk data transfer. These application behaviours would be expressed in terms of meaningful flow attributes that would allow them to be identified directly from network traffic; for example, bulk data transfer can be defined in terms of data sent per unit time and mean packet length, while command-shell interactive behaviour can be expressed in terms of the proportion of packets having a certain length and certain timing characteristics.

In this thesis, we focus on building tools for the analysis of network traffic to identify and quantify application behaviours, in support of a shift to behaviour-based network traffic classification. Our particular focus is on flow attributes, which are measurements of network traffic that can be used to study and describe application behaviours. We focused on the tools to build flow attributes, rather than on building a complete set of flow attributes, as the latter goal was too ambitious with the existing tools, and because we expect new flow attributes will need to be created to cope with applications attempting to evade the existing ones.

Our original motivation for this work was in discriminating between normal web surfing activity and malicious software masquerading as such. In that scenario, it must be assumed that an attacker will attempt to evade network traffic characterization. We do not directly address evasion here, leaving it for future work. However, evasion does motivate this work in that we expect such a detection effort to become an arms race, and so we have built a tool that allows the rapid development of new flow attributes, to allow defenders to more quickly adapt to the attackers' changes. This indirectly supports the use of a variety of flow attributes as a countermeasure to evasion; efforts to mimic a small number of attributes will often disturb others (e.g. padding packets to give a certain mean packet length will increase the data rate and total amount of data sent). We do not, however, address evasion in any formal manner.

1.2 Structure

The structure of this thesis is as follows. First, in Chapter 2 ("Background"), we present some information on a variety of topics that we will use in this work, including machine learning, networking and networked applications, network traffic classification, and the existing tools for handling network traffic.

A detailed exploration of the representation of network traffic and a description of the tool we have developed is presented in Chapter 3 ("Flow attributes"). The chapter starts with a description of networked application behaviours in section 3.1. In section 3.3.1, we describe a notation for defining flow attributes, and then in section 3.3, we use our notation to define a number of flow attributes from the literature.

Chapter 4 ("The ANTARES tool") describes the Advanced Network Traffic Analysis Research and Exploration Suite (ANTARES), which is a tool designed to allow researchers to easily define a wide range of flow attributes. It is a component library that aggregates and processes network traffic, and provides mechanisms for a user to define the attributes that they wish to be computed on that traffic. It allows attributes to be defined in terms of other attributes, and will be used to implement an interpreter for a high-level language, based on our notation, for defining flow attributes.

We perform a qualitative evaluation of ANTARES and of the flow attributes we have defined in Chapter 5 ("Evaluating flow attributes"). The overall conclusions of the thesis and our analysis of the future work enabled by it are given in Chapter 6 ("Conclusions and future work").

We have also included appendices with additional information that we believe will be of use to other researchers. Appendix A ("Data conversion") describes a large set of data available from the National Laboratory of Applied Network Research (NLANR, now subsumed by the Collaborative Association for Internet Data Analysis, or CAIDA), and documents the process necessary to convert the data to a more widely usable format. In that process, we used tools that we have developed and included in our toolkit; these tools serve a variety of purposes, such as breaking down network capture files by application or into timeslices, and extracting network flows as samples. We have found these to be quite useful in our investigations, and hope that other researchers can benefit from the effort that we have put into them. Appendix B ("Error tables") includes detailed tables of error rates for classifiers that we used in our evaluation of flow attributes; it helps illustrate the behaviour of the flow attributes and applications that we studied.

1.3 Summary of contributions

This thesis focuses on building the tools to support future research in network traffic classification without the use of application-layer payload data. Our main contributions are a notation for expressing flow attributes, and ANTARES, a software tool designed to facilitate the computation and evaluation of flow attributes. We have developed a number of other tools for manipulating network traffic and bundled them together with ANTARES into a toolkit;[1] we have also made available a data set consisting of flow attribute values computed from publically available packet header traces, converted to a widely usable format. We consider these additional tools and data to be a minor contribution.

We define a notation for precisely expressing flow attributes and use it to define a selection of flow attributes from the literature. This notation is designed to allow the unambiguous definition of flow attributes using a small set of basic operations, so that they can then be computed by a tool such as ANTARES.

We also present ANTARES, a C++ object library for computing flow attributes. ANTARES is designed to be flexible and extensible, and its primary goal is to support the rapid development of new flow attributes for experimentation. We perform a qualitative evaluation of the tool by using it to compute flow attributes on publically available network data from a variety of networks, and find that it meets its requirements, though more rigorous testing should be performed as future work. This tool is similar in purpose to NetMate [ZS06]; the primary advantage of our tool over the latter is that ours allows a researcher to combine flow attributes to create other attributes, in order to facilitate the process of

[1]For clarity, we will refer to our main tool for processing network traffic as the ANTARES tool or simply ANTARES, and to the larger toolkit as the ANTARES toolkit.

developing and experimenting with such attributes.

We have made available a collection of data processing tools for use in network traffic experiments, as well as a data set consisting of flow attribute values computed on publically available packet header traces using ANTARES. The tools facilitate the process of selecting samples from network data, by providing routines for dividing data traces into timeslices and by transport-layer port, among other tasks. The data set consists of 8400 packet header traces from connections of different applications in different network contexts, along with a variety of flow attribute values computed from these traces. The data is in comma-separated value (CSV) format, which is usable by most mathematical and statistical software packages.

Chapter 2

Background

In this chapter we present some background to establish the context of this work. We consider some statistical techniques that we will use in evaluating candidate flow attributes, and discuss networking and network traffic in general, in order to better situate our discussion. We then present a survey of prior research into classifying network traffic, and discuss the issues with them, particularly in their data sets and testing methodology, that motivated many of the decisions in this work.

2.1 Machine learning and statistical data analysis

We draw upon techniques from the fields of machine learning and statistical data analysis in our exploration of behavioural flow attributes, and so we present here some background on these techniques. Specifically, we examine classification and logistic regression. Classification, in this context, is a machine learning technique for developing systems to discriminate between classes of data points. Logistic regression is a statistical technique used for inferring a quantitative relationship between an experimental variable and an outcome.

In this work, these techniques are central to our analysis of behavioural flow attributes. We use pairwise classifiers to discriminate between different application behaviours based on a flow attribute; the classifiers use logistic regression to infer the relationship between the flow attribute and the application behaviours in question, which will be used as the basis of classification.

2.1.1 Classification

Classification, or supervised machine learning, is the discipline concerned with the automatic or semi-automatic generation of algorithms that assign labels to data points based on previously observed data. Given a set of sample data where each sample is labelled, a classification algorithm will generate a classifier, which is a function, or set of rules, or similar construct, that it can use to predict for a previously unseen sample what label that sample should be given. Research in network traffic classification uses machine learning-based classification techniques, and we generate classifiers in this work to gauge the discriminative power of flow attributes, so we explain here some relevant concepts from the field of classification. This discussion draws from Frank and Witten [WF99].

A common model of classification involves predicting a class label from a *feature vector*, which is a one-dimensional array or list of variable values computed from a data sample. The experimenter decides on a set of variables, or *features* that are to be used to train the classifier, and for each data sample, the value of those variables are computed and entered into such a feature vector. These feature vectors are fed into a classification algorithm,

	Actual class	
Classification	Class C	Not C
Class C	True positive	False positive
Not C	False negative	True negative

Table 2.1: Classification results

which produces a classifier that can be used to predict the label that it expects to be associated with a feature vector. Ideally, a classifier will be able to assign the correct label to vectors that were not part of its training set.

A classifier can be tested for accuracy by having it test a set of samples with known classes, and comparing its output labels with the actual known labels. The testing should preferably be done on a set of samples that is distinct from those used to train the classifier in order to prevent *overfitting*, the phenomenon where the classifier learns the test data too well and is unable to work on the general problem.

There are two common measurements of the performance of a classifier, accuracy and recall. Given a target class C, *accuracy* is a measure of how often the classifier is correct when it labels a sample as being of class C, and *recall* measures the proportion of the samples that were actually in class C were identified as such by the classifier.

A classifier's performance can also be evaluated in terms of true and false positives and negatives. Table 2.1 shows these concepts graphically; a positive is a sample that the classifier identifies as belonging to class C, and a negative is one that the classifier identifies as not in class C. A false positive is a sample that does not belong in class C that the classifier identifies as being in class C, and a false negative is one that does belong in class C that the classifier does not label as being in class C.

The accuracy and recall of a classifier can perhaps most easily be explained in the terms of table 2.1. Accuracy is equal to the number of true positives divided by the sum of the true and false positives. Recall is measured as the number of true positives divided by the sum of the true positives and false negatives.

Cross-validation is a technique used to evaluate the performance of a classification algorithm. A classification algorithm will learn how to classify the training data it is given, so simply testing it on the same training data will give an optimistic estimate of its performance on data it has not previously seen. To compensate for this, k-fold cross-validation is used, where k is a parameter. In k-fold cross-validation, the experimenter divides their data into k roughly equal parts. They then train and test a classifier k times; each time, one of the k parts is not used for the training, but is used to test the classifier trained on the other $k-1$ parts. This gives a more accurate estimate of how well the classifier will do on previously unseen data.

We have briefly presented here some key concepts from the field of classification, primarily as background to many of the other works in traffic classification which make use of these techniques. Our own classifiers are fairly simple; we use only one feature at a time, and we use logistic regression (described in section 2.1.2) to find the threshold that separates the target class from the alternate class.

2.1.2 Logistic regression

Logistic regression is a statistical technique that is often used to analyze categorical data and find the relationship between a variable of interest and the odds of the data point belonging to a particular category. This technique is used in this thesis to generate classifiers, and so we describe it here, though not in great detail. This discussion draws on Neter, Wasserman, and Kutner [NWK85].

Regression attempts to determine the quantitative relationship between a response (or dependent) variable and one or more explanatory (or independent) variables. In an experiment, the independent variables are manipulated by the experimenter to produce changes in the dependent variable; in an observational study, the experimenter studies the values of the response variable in cases with different values of the explanatory variables. A regression task normally involves choosing a response function (such as a straight line in linear regression), and estimating the parameters of that function such that it describes the relationship.

As an example, suppose we wanted to explain the observations of a response variable Y as a linear function of an explanatory variable x, with some error. We could express this as:

$$Y_i = mx_i + b + \varepsilon_i i = 1, \ldots, n$$

This simply means that each observation Y_i is on a line described by $mx_i + b$, except for being off by some error term ε_i; if this model is valid, we expect the error terms to be normally distributed, with a mean of 0. More generally, we replace the line by a function $f(x)$; in the above example, m and b are parameters of the function. This would be as follows:

$$Y_i = f(x_i) + \varepsilon_i i = 1, \ldots, n$$

An estimation method is used to iteratively attempt to find values for the parameters of $f(x)$ such that the error terms are minimized. A popular estimation method is least squares, where the function to be minimized is:

$$Q = \sum_{i=1}^{n} [Y_i - f(x)]^2$$

In linear regression, where the response function is a straight line, the parameters are the slope of the line and the y-intercept (*m* and *b* in the previous examples). In logistic regression, the response function is what is called the log-odds or logit of the probability. That is, rather than try to fit a function representing the probability of a sample belonging to a target class, we try to fit a function representing the logarithm of the odds, which is the ratio of the probability of a sample belonging to the target class over the probability of the sample not belonging to the target class. More formally, if the probability of a data sample *i* being a member of the target class is given by π_i, then the logit π'_i is given by $\pi'_i = log(\frac{\pi_i}{(1-\pi_i)})$.

Logistic regression is often appropriate for regression on categorical or indicator dependent variables, those where the value is either 0 or 1 depending on e.g. whether the sample belongs to a particular class. This is because the logit response function has a number of properties that are desirable for regression on categorical variables, particularly that the probability predicted by a logit function will be in the interval (0,1), and thus will not predict probabilities less than 0 or greater than 1, which is a potential problem with simple linear models.

This thesis uses logistic regression to generate classifiers, so we have given here an overview of how the technique works. In our case, the dependent variable will be a class, specifically a class of network traffic, and the independent variable will be one of the flow attributes that we wish to evaluate. Given samples of two classes of network traffic, the regression problem is to fit the log-odds line to best describe, for that flow attribute, what the probability will be that a sample with a given value for the flow attribute will be of the target class vice the alternate class.

2.2 Networking and network traffic

This thesis is focused on an exploratory analysis of network traffic generated by applications, and so some background on networking and network traffic is appropriate. This section will review some terms and concepts that will be important for understanding the remainder of this work. This discussion draws on Peterson and Davie [PD00].

Network traffic is transmitted by a *network node* or *network host*. We use both terms almost interchangeably, the difference being that we use the term host to place emphasis on the host itself and its use of the network, whereas the use of the term node emphasizes its role as part of the network.

The network protocols that we deal with in this thesis are arranged in *layers*; a commonly used reference model is the *OSI stack*, a seven-layer model consisting of physical, data-link, network, transport, session, presentation, and application layers. The layers relevant to this work are the *network layer*, which takes responsibility for routing a packet over a network, and the *transport layer*, which takes responsibility for managing communication between software processes on the communicating hosts. We also refer to the *application* *layer*, which generally refers directly to the application that is using the network.

In a layered protocol stack, each layer is mostly independent from the others, and the data sent by a protocol belonging to a given layer on one node is processed by the same layer on the receiving node. This is often accomplished using a technique known as encapsulation. This technique involves each protocol prepending and/or appending its own information to the *payload* from a higher layer. Many such protocols will prepend a *header* to the payload to carry its own information to the corresponding protocol at the remote node. That remote protocol instance can then remove the header and process the information it contains before passing the unmodified (or restored) payload to the appropriate higher-layer protocol at the remote node.

We use the term *network traffic* to refer generally to data that is sent across a network, whether it is being sent, in transit, or being received, or whether it has been captured and stored in some static format. A unit of such data is a *packet*; we use this term in a very general sense, to mean a bundle of data sent across a network. Different layers of the stack refer to such bundles by different names; in the physical and data link layers, they are often called frames. When talking about the network layer, *packet* is the common term. At higher layers, various protocols will refer to packets, messages, segments, and datagrams; we will use the term packet, and explicitly identify what type of packet we are discussing in the given context.

A common identifier used by protocols such as TCP and UDP is the *port number*, or port, an integer used to multiplex and demultiplex network traffic originating from or destined to a particular node. In common usage, a port acts as a subaddress on a particular node.

The terms *network session* and *network flow*, as used in this thesis, are closely related; we use both to refer to data sent across a network in the course of an application's operation, usually data sent to accomplish a single goal and involving a single remote process. The main difference is perspective; a network session refers to such a transfer of data from the perspective of the application, whereas a network flow refers more directly to the network traffic generated by such a network session, being more focused on the perspective of one intercepting such traffic.

The concepts of network latency and jitter are related to the time taken by packets to traverse the network. Supposing we have a node A and a node B, latency describes the time taken for a packet to reach B after being sent by A. The term latency is generally used to describe the "normal" amount of time taken. Jitter refers to the variations in the amount of time taken. For example, if one were to send a dozen packets from A to B and measure the time taken by each to be received after being sent, the mean of those measurements would be a measure of the latency, and the standard deviation would be a measure of the jitter.

Another concept used repeatedly in this work is that a session involves a client and a server. The applications that we consider employ the *client-server architecture*, in which one node, the *server*, advertises a service that it provides, and the other node, the *client*, connects to the server and uses the service.

The applications we deal with in this work are built on the client-server architecture, and when discussing traffic generated by these applications, we distinguish between client-side and server-side traffic. Client-side traffic is traffic sent by the client to the server, and server-side traffic is traffic sent by the server to the client. The distinction is necessary because the roles of the client and the server are generally quite different, and thus the traffic generated by them is asymmetric.

For the purposes of our work, we will assume that we know a priori which side of a network flow is which, but any practical implementation of a traffic classification system will need to be able to deal with uncertainty in the direction of the traffic.

2.3 Networked applications

Programs and suites of programs that use computer networks (particularly the Internet) in the course of their primary purpose are referred to here as networked applications. In this work, we use samples of network traffic data produced by several networked applications; we describe these applications here and discuss their relevance to this thesis.

One of the most important factors in selecting applications was that we wanted to focus on applications for which we could obtain samples from all of the data sets we used; this requirement limited the set of applications that we had to choose from. However, we have obtained a basic set of applications for testing the tools and techniques that we have developed. The applications of interest are: the File Transfer Protocol (FTP), which is subdivided into FTP-control and FTP-data; Telnet; the Simple Mail Transport Protocol (SMTP); the HyperText Transfer Protocol (HTTP); and the Post Office Protocol version 3 (POP3). Strictly speaking, these are not applications themselves, but protocols that are implemented by various applications; however, for our purposes, we will assume that the protocol is designed for a particular purpose, and that the purpose drives the behaviour more than the implementation does.

In this section, we describe the applications and the known port numbers associated with them. The standards for network interactions, or protocols, of these are defined in documents called Request For Comments (RFC), published by the Internet Engineering Task Force (IETF). The IETF develops and organizes documentation such as protocol standards, best practices documents, and other documents for the purpose of improving the Internet [Alv04]; the RFC system is a major vehicle to this end.

The File Transfer Protocol (FTP), as its name suggests, is a protocol designed to allow the transfer of files between nodes on an internetwork. In this context, we are specifically concerned with FTP as used by standard FTP client and server programs such as the ftpd server and FTP client normally included in Linux, Unix, and BSD operating systems, used interactively by a human user rather than an automated process. FTP is defined in RFC959 [PR85]; this discussion draws primarily on that document. An FTP session consists of two communications channels, the control channel and the data channel. The two channels are used for different purposes and are expected to exhibit entirely different behaviours, so for our work we consider them to be two separate applications, which we will refer to as FTP-control and FTP-data, respectively. FTP-control is registered as using port TCP/21, and FTP-data as using port TCP/20 [Aut06].

The Telnet protocol is a protocol designed to provide an interface between terminal devices and processes such as terminals slaved to a mainframe and the mainframe terminal process; on the Internet, it is often used for remote access to a command shell,[1] which is the use with which we are concerned. We expect that the major implementation of telnet that we will be dealing with will be that of the major telnet servers and clients, such as the BSD

[1]A command shell (or simply shell) is a program which serves as a command-line interface between a user and the operating system on a computer system

server and client, or the telnet client included with Windows operating systems. It is also used for the control channel of FTP. Telnet is defined in RFC854 [PR83], and is registered as using port TCP/23 for communication [Aut06]. It is of particular interest in this thesis, as we expect an attacker who successfully gains unauthorized access to a node would want to interface with a shell, and telnet is the best widely-available proxy to such shell activity.

The Simple Mail Transfer Protocol (SMTP) is a protocol used to implement the handling of e-mail on an internetwork. SMTP is defined in RFC821 [Pos82]; RFC2821 is a proposal to replace that standard, currently designated as a “proposed standard” by the IETF [Kle01]. The latter is a compilation of the original SMTP standard plus several of its major extensions [Kle01]. SMTP is registered as using port TCP/25 [Aut06].

The HyperText Transfer Protocol (HTTP) is a protocol designed to allow the distribution of content via an internetwork. The content distributed via HTTP is not limited, but often consists of documents in HyperText Markup Language (HTML) and images. HTTP version 1.0 is defined in RFC1945 [BLFF96], and version 1.1 is defined in RFC2616 [FGM$^+$99]. Both versions are registered as using port TCP/80 [Aut06].

The Post Office Protocol, version 3, is a protocol designed to allow e-mail clients, used directly by a human user, to interface with the mailhosts responsible for forwarding mail across an internetwork (via a protocol such as SMTP, described in section 2.3). POP3 is defined in RFC1939 [MR96], and registered as using port TCP/25 [Aut06].

2.4 Network traffic classification

There is a growing body of research, driven by several distinct motivations, regarding the classification of network traffic. The major questions addressed by this research are to identify what a given type of network traffic "looks" like, in general, and how to use that information to be able to distinguish between different types of traffic. These questions are relevant in several contexts, such as network security and administration, network provisioning, protocol design, and network simulation. This section provides an overview of the existing literature in traffic classification, focusing on that which uses non-payload information to classify traffic.

The existing work in this area shows that measurements of network traffic offer enough information to classify the traffic by application, though the existing approaches are not yet accurate enough for practical purposes. We note that some of the researchers discussed in this section, e.g. Frank [Fra94] and Karagiannis et al. [KPF05], created groupings of applications that they considered similar, though with no quantitative basis for these groups. Also, most only considered the subset of information available from the tools they chose to use.

One segment of somewhat related work not dealt with here is research on classifying network traffic based on resources consumed, such as Estan and Varghese's [EV03] work on distinguishing between "elephants and mice" – i.e. network flows of significant and insignificant volumes of traffic, respectively. While that is certainly a form of network traffic classification, our work is focused on classifying traffic based on the nature of the flow and the type of activity that generated it, regardless of whether it is an elephant or a mouse.

Where error rates are reported, unless otherwise noted, these are combined error rates, i.e. the number of false positives and false negatives (as described in section 2.1.1) divided by the total number of samples. The results in the literature are presented in various different forms, so we have rephrased the reported error rates or accuracies in terms of combined error rates where possible, in an attempt to facilitate comparisons of the different approaches.

Frank [Fra94] applied feature selection algorithms to the classification of flows from a small set of classes, attempting to improve the performance by selecting only the most useful features for the task. He used some basic flow attributes, plus the probability, according to Heberlein's Network Security Monitor [HDL$^+$90], that the flow was malicious. The classes he used were “login”, “shell”, and “SMTP”, classified by hand, though he did not explain what constituted a login or shell flow. He found that all three of the algorithms used performed well to find a good set of features, although the most complex one did better than the others on the task of distinguishing shell flows, and concluded that in many cases, simple feature selection algorithms perform decently well. He reports error rates (computed using test data independent from the training data) of about 3% or less.[2]

Zhang and Paxson [ZP00] described a number of techniques for detecting what they refer to as backdoors, which are unauthorized access mechanisms in a system installed by an attacker. One detection technique they use for identifying interactive traffic is based on the proportion of consecutive small (less than 20 bytes of payload) packets in the flow.

[2]The percent signs in the error rates that Frank himself reported are superfluous and should be ignored, according to Dunigan and Ostrouchov, who report that this was confirmed to them in an e-mail from Frank [DO01].

They report results for their general detection algorithm for interactivity (which uses packet size and timing information) that correspond to a combined error rate of about 0.77%; they mention, however, that most of the false positives were in fact e-mail protocols used interactively. Disregarding those, the combined error rate would instead be about 0.44%. They note, however, that the data set used to evaluate false positives had many high-volume applications such as HTTP, NNTP, and the data channel of FTP filtered out; had HTTP in particular been present, it might also have led to higher numbers of false positives.

Dunigan and Ostrouchov [DO01] found that they could discriminate among eight different applications using flow attributes based on packet sizes, inter-packet delays, and packet directions. They divided the possible space of these flow attributes up into bins, and treated each bin as a separate flow attribute unto itself; for example, one feature could be "the count of packets with a length of 60 bytes, a delay since the last packet of between 1 ms and 800 ms, and where both that packet and the previous packet were from the server to the client".

They tested their approach, using statistical methods to select three such flow attributes and using those attributes to estimate probability density functions for each of the applications, then classified flows by selecting the function that gave the greatest probability based on the flow attributes of the flow. The worst error rate they report, based on testing and training on the same data set, was 6.41%, for e-mail, and the other applications were classified with error rates of 4.28% or better (most of them better than 1%). They also report, however, that preliminary testing on a second data set containing e-mail flows results in the classification error rate more than doubling.[3]

[3]Dunigan and Ostrouchov reported their results as a confusion matrix, where it was not clear which axis

Early et al. [EBR03] developed an anomaly detection approach based on flow attributes. They used frequencies of various TCP flags and mean packet inter-arrival times computed across windows of flows to determine, among five common applications, to which the flow belongs. They used the C5.0 decision tree algorithm [Qui93] to develop their classifier, and found that it could classify the flows with recall of 82% or better for SMTP, and with recall of 96% or better for most other applications,[4] including no errors for some applications. Their primary results are based on the 1999 Lincoln Labs/DARPA data set [LHF$^+$00], which has some known issues [McH00, MC03], which they acknowledge. As a result, they also used data collected from their own networks, and report that their system performed equally well on that data set.

Roughan et al. [RSSD04] performed a set of experiments aimed at showing the potential of discriminating between flows from different applications based on flow attributes; their motivation was the improvement of quality-of-service schemes. In one of their experiments, they developed classifiers using several classification algorithms to distinguish between applications in several sets. They chose the average packet size and duration of the flows as data features to distinguish between Telnet, FTP-data, Domain Name Service (DNS), and streaming video.[5] They also worked with a seven-class version, adding in HTTP, HTTP Secure (HTTPS), and KaZaA.[6] The best error rates they obtained were 5.1%

represented real classes and which represented assigned classes, so we have simply computed these combined error rates based on the values in that table. It is not clear whether error rate that they reported for the second data set was based on false negatives or on false positives.

[4]They report their performance as "accuracy", but their description indicates that the values are actually recall values.

[5]Streaming media , such as video, is a class of network application where the client obtains data from the server as it renders it, rather than obtaining all of the data prior to rendering.

[6]KaZaA is a peer-to-peer file sharing application; peer-to-peer is a decentralized network application architecture where endpoints communicate directly with one another, with little or no reliance on centralized server nodes.

and 9.4% for these classification problems, respectively, using a 3-Nearest-Neighbour classification algorithm.[7] Rather than using attributes computed over individual flows, however, they used daily averages computed from the flows seen in a day; some of our observations suggest that these may have led to optimistic results, though we by no means establish that.

Borders and Prakash [BP04] developed an anomaly detection system they called “WebTap”, which uses flow attributes to detect anomalous behaviour in HTTP sessions. Their main focus is on information being smuggled out of a network via HTTP, such as by spyware.[8] Their approach operates on HTTP requests and responses, rather than on packets, and they use attributes such as inter-request delay, request sizes, and outbound data volumes. They found that their approach was successfully able to detect actual unauthorized software on their network, as well as some hidden channel software that they installed to test the system.

Wright et al. [WMM04] used Hidden Markov Models[9] (HMMs) and k-Nearest Neighbour (kNN) learning algorithms to classify flows and aggregates of flows generated by different applications, based on packet lengths, inter-packet delays, and packet directions. Their focus was on classifying traffic with minimal information, such as might be available from traffic in an encrypted tunnel; they went so far as to alter the packet lengths to simulate

[7]3-Nearest-Neighbour is a common machine learning algorithm which classifies a data point by looking at the three closest data points for which a class is known [WF99].

[8]Spyware is malicious software that gathers information from the host computer about the user; often it is designed to pick up sensitive information such as credit card numbers, social insurance identifiers, or such things as game activation codes.

[9]An HMM is a model that represents a finite state machine where only the outputs can be observed, and the internal states and transitions are unknown [Rab89]; this is appropriate to the task, as network traffic can be viewed as the output of a hidden finite state machine (or set of such machines) – the application in question.

the effect of a block cipher being used on the data. This is a particularly difficult problem; most other approaches discussed here assume that at least the packet headers are available.

Their kNN approach to identifying single-protocol aggregates was based on features similar to those used by Dunigan and Ostrouchov [DO01], in that they were based on relative proportions of packets falling into bins based on packet lengths and direction, though they used only four bins. For this task, they found that for some applications they could get good (often perfect) recall, but for some low-volume applications, the performance was much poorer, as their features were computed for epochs of constant duration. They used more fine-grained bins for their HMM-based approaches for identifying individual flows, and also incorporated timing information in some of them. Although they had trouble with FTP-data, their full classifier (using packet size, timing, and direction) gave recall values of 76% or better for other applications.

Hernández-Campos et al. [HCNSJ05] emphasized the need to focus on application behaviours rather than on applications. They performed clustering on flow attributes for flows from the ABILENE-I data set from the National Laboratory for Applied Network Research (NLANR), based on some simple and less simple flow attributes. Rather than computing these attributes on packets, though, they computed them on what they called Application Data Units (ADUs). These ADUs were, using the client-server paradigm, request-response pairs, where packets comprising a request (e.g. an HTTP request) were the first part of the ADU, and packets comprising a response to that request (e.g. an HTTP response) were the second part of the ADU. They then computed their attributes on the amount of data in each part of the ADU, and the delays between ADUs. They also showed that the size of both parts of the ADU, at least for TCP flows, could be deduced from the sequence and acknowledgement numbers from just one side of the flow.

They found that the clusters qualitatively separated the traffic out into interesting classes, and they demonstrated a visualization technique using “heat maps” – graphical representations of attributes values – to analyze the clustering. They report that traffic with ports associated with peer-to-peer applications separated itself out into two clusters, which they believe are different modes of operation for the peer-to-peer applications, that traffic with ports associated with HTTP applications separated out into another cluster, and that there were indications of homogeneity in the other clusters as well.

Karagiannis et al. [KPF05] propose a system they call BLINC, for BLINd Classification, that focuses on patterns in communication between nodes to classify each node based on what application activity that node is engaged in. Rather than examining attributes of the flows themselves, they develop data structures, which they call graphlets, based on patterns in the network activity in terms of other hosts contacted, different ports and protocols used, and relationships between those pieces of information. They then compare the graphlet generated by a host’s behaviour to a library of graphlets of known application behaviours in order to determine what application is running on the host. Though this approach does not use flow attributes, it is nonetheless related, and could be a useful alternate approach to complement a flow attribute based approach.

DeMontigny-LeBoeuf [DL05] described a wide range of flow attributes that can be used to classify flows by application, and built a hand-tuned classifier from them. She organized them based on the higher-level qualitative features that they attempt to measure, such as interactivity, conversationality, and regularity. An advantage of this approach of linking flow attributes to higher-level features is that her system is able to generate a qualitative description of a sample flow in terms that may be more useful to a human analyst than the raw flow attributes would be. Her hand-built classifier distinguishes among 9 different applications. Its worst error rate is 36% (for HTTP), but for most of the applications, its error rates are around 10% - 20%; note that these error rates, unlike those of many of the other approaches mentioned here, are for a data set that was not used in building the classifiers [DL06].

Moore and Zuev [MZ05] used Naïve Bayes classifiers,[10] with a Fast Correlation-Based Filter for feature selection, to classify network traffic into groups of applications (e.g. bulk data transfer, interactive, database). Their best classifiers obtained combined error rates[11] as low as 3.7% (across all application groups) on data from the same timeframe,[12] with error rates of 6.3% on data collected 12 months after the training data was obtained. However, their approach apparently incorporated the server-side port as a flow attribute; we wish to ignore that value when classifying traffic. Other attributes that they found useful included the number of packets with a particular flag in the TCP header set, the initial values for the TCP window sizes,[13] and average TCP payload length.

Collins and Reiter [CR06] used NetFlow data[14] to distinguish BitTorrent traffic[15] from that of FTP-data, SMTP, and HTTP. They designed tests to distinguish BitTorrent traffic based on four measurements: a failed connections heuristic, the bandwidth (data rate) of the

[10]A Naïve Bayes classifier is a common machine learning algorithm that expresses an outcome, such as a flow belonging to a particular application, as a probabilistic model of a set of factors, making the naïve assumption that those factors are independent of one another [WF99].

[11]What they report as accuracy is effectively the complement of the combined error rate; they also report a "trust" measure, which corresponds to the accuracy as defined in section 2.1.1.

[12]Their error rate may even be overestimated, as they trained on $\frac{1}{10}$ of their data set and tested on the remainder, inverse to the more common 10-fold cross-validation method described in section 2.1.1.

[13]The TCP window size is a field in the TCP header used for flow control.

[14]NetFlow is data collection system and data format designed by Cisco that provides summaries of network flows by sampling packets, in order to keep up with high-data-rate links [EKMV04].

[15]BitTorrent is a popular file sharing application [IUKB$^+$04].

flow, the histogram of the packet lengths, and the logarithm of the number of packets. They report that they obtained a 72% true positive rate with no false positives (i.e. a combined error rate of 28%) by using a voting scheme among these tests.

Another thread of interesting work is by Zander, Nguyen, Williams, and Armitage [ZNA05b, ZNA05a, ZWA06, NA06] regarding the automated classification of network traffic using statistical measures of packet lengths and inter-packet delays.

Zander, Nguyen, and Armitage [ZNA05b, ZNA05a] clustered network flows and then compared the resulting clusters with the applications that generated the flows, based on port numbers. Specifically, they reported the homogeneity of the clusters – i.e. the percentage of flows in the cluster belonging to the dominant application for that cluster. Their candidate flow attributes were the means and variances of inter-packet delay and packet length, total data volume in bytes, and duration, where every attribute except duration is computed for both sides of each flow. They found that, by labelling each cluster according to the dominant application and then using the clusters to classify the data, they obtained a mean recall (which they termed accuracy) of 86.5% [ZNA05a]. Note that that is likely overestimated, as they appear to have trained and tested on the same data.

They also report the influences of the various flow attributes they used. Some of their findings were that packet lengths were preferred over inter-arrival delays for the applications they were attempting to classify, that duration was not strongly preferred by their feature selection algorithm (in contrast to e.g. Roughan et al. [RSSD04]), and that the two most useful flow attributes were the variances in the packet lengths for each direction of a flow [ZNA05a].

Zander, Williams, and Armitage [ZWA06] then applied the earlier work in application classification to examining the applications present in historic Internet traces, under the term “Internet Archaeology”. Specifically, they examined how feasible it would be to identify peer-to-peer and games traffic in historic traces based on flow attributes. They found that the flow attribute values for the applications of interest had not changed significantly between the different time periods of the data sets they examined, and that different applications, even similar applications, could be distinguished with high accuracy (better than 90%, evaluated using 10-fold cross-validation) using these attributes.[16]

Zander et al. [ZNA05b, ZNA05a, ZWA06] used a traffic measurement tool developed by Zander and Schmoll [ZS05, ZS06] called NetMate, which is similar to our own tool; had we come across it earlier, we likely would have tried to leverage it for building our own tools. It is an application built in C++ for computing flow attributes on network traffic, which allows attributes to be defined using eXtensible Markup Language (XML), a machine-parsable text format.

More recently, Nguyen and Armitage [NA06] examine the effect of using sub-flows for classification, rather than looking at entire flows. This work is particularly interesting for the purposes of this thesis, as it hits upon the issue of non-homogeneous applications and hints at the need to deal with application behaviours rather than dealing with applications holistically. They use a Naïve Bayes classifier to attempt to identify traffic from a 3D networked game based on a sliding window of the most recent packets. Their findings strongly suggest that the game traffic was generated differently at different times, and that a classifier trained on the appropriate sub-flow (e.g. near the start or in the middle of each

[16]It is not clear, however, how they computed these “accuracy” values, and whether it was the same metric as reported in the earlier work [ZNA05b, ZNA05a], which is more commonly termed recall; informally, that seems likely from their phrasing.

training flow) would do much better at identifying traffic from a similar stage of the application. This is of particular interest to us, as it is one of the few related works that deal with variations within an application, let alone within flows of an application.

2.4.1 Summary

There are many indications in the literature that classification of network traffic can be done with reasonable accuracy without using payload data, though sufficiently high for practical purposes. Different researchers have used a variety of different flow attributes, generally using what was available from the tools that they were using. We have built a tool for computing flow attributes and used it to implement a variety of attributes from the literature, which should facilitate applying these approaches to a wider set of flow attributes. Of the approaches presented here, our work is most closely related to that of DeMontigny-LeBoeuf [DL05] and that of Zander and Schmoll [ZS05]; we will compare our work to the former here, and the latter is discussed in detail in section 4.1.

In table 2.2, we present an overview of the types of measurements used by the different approaches. We used a "/" to denote types of measurements where the approach used just one or two of many possibilities, and an "X" to indicate that the approach used several measurements of that type. Time is the class of timing-based attributes, primarily duration and inter-packet delay, discussed in section 3.3.2.[17] Len and LenH are the classes of measurements based on packet lengths; the Len class are general measurements such as averages, described in section 3.3.3, whereas the LenH measurements are heuristics based on

[17]We used "X" for approaches that used both duration and inter-packet delay, and "/" for those that used only one of the two.

Approach	Time	Len	LenH	Vol	Flag
Frank [Fra94]	/			X	
Zhang and Paxson [ZP00]	/		X		
Dunigan and Ostrouchov [DO01]	/		X		
Early et al. [EBR03]	X	X			X
Roughan et al. [RSSD04]	/	X		X	
Borders and Prakash [BP04]	/	X		X	
Wright et al. [WMM04]	/	/			
Hernández-Campos et al. [HCNSJ05]	X	X		X	
DeMontigny-LeBoeuf [DL05]	X	X	X	X	
Moore and Zuev [MZ05]	X	X		/	
Collins and Reiter [CR06]		X		X	
Zander, Nguyen, and Armitage [ZNA05b]	X	X		X	
Zander, Williams, and Armitage [ZWA06]	X	X		X	
Nguyen and Armitage [NA06]	/	X			

Table 2.2: Table of measurement classes used in surveyed approaches

packet lengths, described in section 3.3.5. Vol is the class of measurements based on data and packet volumes, described in section 3.3.4. Flag represents heuristics based on packet flags, described in section 3.3.5. The classes are quite coarse; even between two approaches with an "X" in the same column, the actual attributes used often vary considerably.

The flow attributes that we define in section 3.3 are drawn in most part from the approaches described in this section. We note where the attributes have previously been used when we define them. We do not concern ourselves in our work with creating new attributes, but more on building the mechanisms by which new attributes may be created.

As did DeMontigny-LeBoeuf [DL05], we conduct a survey of the flow attributes used in the literature and implement several of them. In contrast to that work, however, we have made our tool publically available. Also, in designing our tool, we have generalized several previously described types of flow attributes and designed a notation to facilitate the expression of flow attributes, which will be used to create a better interface to the tool than is presently available.

In addition, we take a different approach to evaluation than did DeMontigny-LeBoeuf; whereas she manually created signatures for different applications and reported the overall classification performance for distinguishing among them, we focus on evaluating the performance of individual flow attributes for discriminating pairs of applications. This approach gives us more insight into the behaviour of the individual flow attributes, which is our focus.

Chapter 3

Flow attributes

The focus of this thesis is on flow attributes, measurements and calculations based on network flows, and on ways to compute and use them. In this chapter, we attempt to more clearly define and explain the concept of these attributes, and discuss the tool that we have developed. We first discuss application behaviours, the types of activity that we wish to measure and characterize, in section 3.1. We discuss how we aggregate packets into flows in section 3.2, and in section 3.3, we describe the flow attributes of interest, with a notation for defining them.

3.1 Networked application behaviour

One of the principle motivations behind the development of ANTARES is the desire to better understand network traffic, in order to improve classification accuracy. Our intuition is that one of the major tasks that will need to be done to do this is to shift from the current paradigm of associating network traffic directly with applications, and instead use network traffic to identify application behaviours, and then identify applications from these behaviours. In this section, we more fully explain our concept of application behaviours, which influences much of this work.

Other researchers have touched on the concepts around application behaviours. One key observation in the idea of application behaviours is that applications are not necessarily homogeneous; that is, a single application can exhibit different behaviours when performing different tasks, even changing behaviour within a single connection. Nguyen and Armitage [NA06] found that, at least for some applications, the performance of an application-based classifier depends on the the portion of a network flow used to train it and the portion that it is used to classify. In their example, a classifier trained against the beginning of a flow of traffic produced by a particular game was much better at identifying traffic from the beginning of another flow than that from the middle of the other flow, and vice versa. Hernández-Campos et al. [HCNSJ05] described the need to look at uses of the network rather than applications, though they focused on grouping applications into behavioural classes rather than fully separating behaviours from applications. Collins and Reiter [CR06] looked at treating an application as a composite of several different types of flow, which they called Short Flows, Messages, and File Transfers, based on the length of the flows.

We use the term *application behaviour* to refer to the activities of a networked application that generate particular types of network traffic. For example, *bulk data transfer* is an application behaviour in which one network node sends a (relatively) large amount of data to another network node, where the transfer proceeds as quickly as the network will allow; common examples of bulk data transfer behaviour are FTP-data (the data transfer channel of the File Transfer Protocol, described in section 2.3) and Simple Mail Transfer Protocol (SMTP, described in section 2.3) where large attachments are being sent along with e-mail.

Another example of an application behaviour that is interesting in this context is *command-shell interactive behaviour*; we discuss two slightly different types of this: keystroke interactive behaviour, and command-line interactive behaviour. Both are generated by a human entering commands at one end of a network connection, and a command shell responding to those commands at the other end. The difference between the keystroke and command-line interactive behaviour is that in keystroke interactive behaviour, every keystroke of the user is transmitted as it occurs, whereas in command-line interactive behaviour, the keystrokes are buffered locally by the user's host and only sent once a complete command has been entered, such as when a carriage return is entered by the user.

We also consider *machine-driven interactive behaviour*, by which we mean automated interactions between two programs. In this work, we consider SMTP and POP3 as examples, where both generally perform some automatic negotiations and then transmit some data.

Figure 3.1 shows an example of a flow that appears to include both of these behaviours. It shows a non-homogeneous Telnet flow from a server to a client broken into ten second timeslices, with mean payload length and total number of payload bytes (i.e. not counting packet headers) computed for each timeslice. For most of the time plotted, the packets and data volume are low, apparently conforming to our expectations of command-shell interactive behaviour. However, near the end of the displayed time, there is a burst of large packets, probably corresponding to the user running a program that produces a large amount of output. This is not uncommon in the data we have examined, and tends to distort

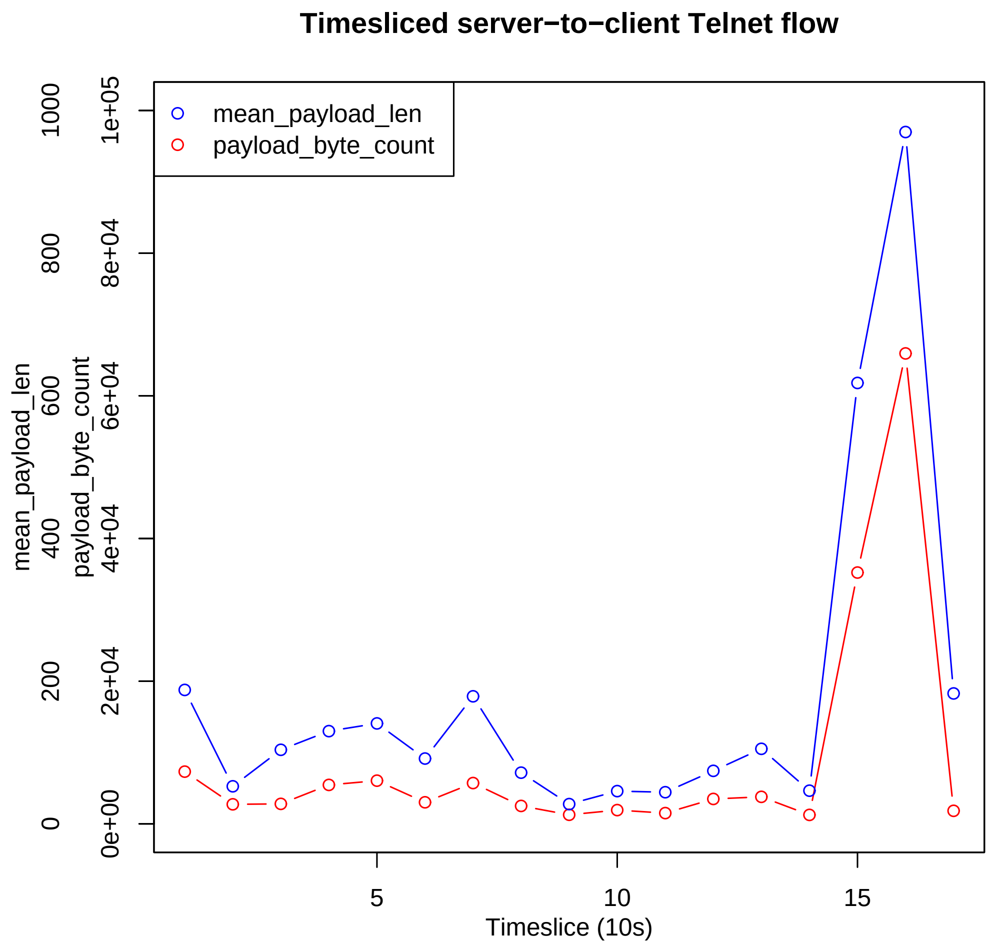

Figure 3.1: Mean payload length and data volume of sample non-homogeneous Telnet flow

flow attributes that are influenced by the entire flow. In this case, we would prefer to express this flow as exhibiting two distinct behaviours: command-shell interactive for most of it, and bulk data transfer for the burst of large packets.

In addition to application behaviours, we also expect to have to contend with behavioural distortions. These are artefacts of network protocols or network conditions, that are visible in the same way as application behaviours but not generated by the application itself. A good example of a distortion is the fragmentation of network data by transport layer protocols such as the Transmission Control Protocol (TCP). TCP will take a stream of data given to it for transmission across a network and break it up into smaller chunks, often less than 1500 bytes to accommodate Ethernet's Maximum Transmission Unit (MTU), often 1460 bytes (which allows 40 bytes for IP and TCP headers). This will lead to sequences of consecutive packets carrying 1460 bytes of payload, which is an effect of the TCP protocol rather than of the underlying application behaviour. However, in this case, the distortion can be useful, as the proportion of such packets can be used as a proxy for a large amount of data being transferred at once, such as with bulk data transfer.

We do not make extensive use of the concepts of application behaviours and distortions in this work; we present them to help explain our interest in studying, rather than classifying, network traffic, and in evaluating flow attributes themselves. Our work does not go far enough to make any concrete statements about such behaviours, but we hope that the tools that we have made available will make it far easier to do so.

3.2 Network flows

An important concept in this work is that of aggregations of network traffic, or *packet aggregates*. These are sequences of packets, often associated by fields in the packet headers such as network addresses and ports. The network flow is the packet aggregate used in this thesis. The flow attributes described in this chapter and summarized in table 3.1 are computed based on a sequence of packets being aggregated together into a logically meaningful unit. We describe here more precisely our use of this term.

The term network flow, as we use it, is defined as a sequence of packets using the same transport layer protocol, and having the same two endpoints, where an endpoint is a network layer address paired with a transport layer protocol. A network flow consists of packets from port *a* on node *A* to port *b* on node *B*, and packets from port *b* on node *B* to port *a* on node *A*. Further, each network flow can be split into two *half-flows*, or directions; in the previous example, one half-flow would be from port *a* on node *A* to port *b* on node *B*, and the other half-flow would be from port *b* on node *B* to port *a* on node *A*. For convenience, we refer to the half-flow from the client to the server as the *forward direction* of the flow, and the other half-flow as the *reverse direction*.[1] This convention will be useful when discussing attributes computed on half-flows, as the values of the attributes are often quite different between the two directions of the flow.

We also use a timeout mechanism, where a sequence of this type that has a gap of more 64 seconds[2] with no packets is considered to be separate flows divided by that gap.

3.3 Flow attributes

In this section, we discuss flow attributes, explaining what they are and how we intend to use them, introducing some qualifiers that we use for different types of flow attributes, and describing a notation. The bulk of this section is descriptions of the different types of flow attributes that we will use in this thesis. We also describe how these flow attributes, and other related attributes, have been used in the previous research discussed in section 2.4. We summarize the flow attributes that we use in this thesis in table 3.1.

[1]Note that we do not deal with peer-to-peer applications in this work; distinguishing between "forward" and "reverse" for such traffic would be more difficult. If the behaviour is symmetric, then this is not an issue, but otherwise the "direction" would have to be assigned by using some other criterion.

[2]Here we follow the example of DeMontigny-LeBoeuf [DL05]; 64 seconds was reported to be the most effective timeout by claffy et al. [cBP95].

A *flow attribute* is a value that represents a network flow in some way, particularly a measurement of a network flow or a calculation based on other such attributes. For example, one flow attribute is the payload data rate of a network flow, i.e. the number of bytes of payload data carried by a network flow per unit time.

Ideally, flow attributes reflect some aspect of the behaviour of the networked application being observed, but they are often affected by peripheral effects. These behavioural distortions (as described in section 3.1) include such things as the load on the host sending the traffic or congestion in the network, which can alter the timing characteristics of the traffic, or more consistent effects such as fragmentation of large messages at the TCP layer, which alters packet length characteristics.

In this thesis, we focus on flow attributes that can be computed in linear time. Although we are not particularly concerned with the performance of our implementations of the flow attributes discussed here,[3] we focus on flow attributes that could be implemented in a practical implementation that deals with network traffic in real time. Flow attributes with an inherent complexity of greater than linear time are not good candidates for such a future implementation, and so we do not deal extensively with them. We do expect that such flow attributes are likely to be useful for the study of network traffic in general, however, and describe them where appropriate. Table 3.1 lists the different attributes that we have implemented using our tool and have also used in our evaluation.[4]

In general, we have selected these attributes because they reflect many of the types of attributes used in the literature. Some, such as *duration*, *mean_payload_len*, *pkt_count*, and

[3]See section 4.2 for a discussion of our requirements for the tool.

[4]Chapter 4 contains two similar tables, table 4.1 and table 4.2, which also include flow attributes implemented using our toolkit but that are not included in the evaluation.

Attribute name	Section	Description
duration	3.3.2	duration of flow
mean_delay	3.3.2	mean delay between packets
mean_pkt_len	3.3.3	mean packet length
mean_payload_len	3.3.3	mean payload length
mean_nonempty_payload_len	3.3.3	mean payload, not counting empty packets
pkt_count	3.3.4	number of packets
nonempty_count	3.3.4	number of packets with payload
pkt_byte_count	3.3.4	total volume of data sent
payload_byte_count	3.3.4	total volume of payload sent
mean_payload_datarate	3.3.4	mean payload sent per unit time
dir_data	3.3.4	ratio of data sent fwd to rev dir
sp_alpha	3.3.5	small packet heuristic α
sp_beta	3.3.5	small packet heuristic β
sp_gamma	3.3.5	small packet heuristic γ
sp_delta	3.3.5	small packet heuristic δ
lp_alpha	3.3.5	large packet heuristic α
lp_beta	3.3.5	large packet heuristic β
lp_gamma	3.3.5	large packet heuristic γ
flag_X	3.3.5	prop. of packets with flag X

Table 3.1: Summary of flow attributes

pkt_byte_count are commonly used in many approaches, and some, such as *mean_nonempty_payload_len* and *payload_byte_count* are variants of these, chosen to illustrate how such variants can be expressed using the same mechanisms as the basic attributes. Other attributes, such as the small and large packet heuristics, are less commonly used, but are useful for identifying certain types of traffic (e.g. small packet heuristics are useful for identifying command-shell interactive traffic [ZP00]). Many other candidate attributes exist in the literature; investigation of these has been left for future work.

3.3.1 Notation

Before we describe the flow attributes, we need to define a notation that we will use in explaining how to calculate them. We mix mathematical and programming-language notations as necessary, and apologize in advance to anyone who is offended by such blasphemies. For example, we use dot notation to denote properties of a complex object, e.g. $p.len$ for the length of a packet (p), and subscripts to denote a particular object or sequence of objects, e.g. we may use p_{first} for the first packet of a flow, or P_{ack} to denote the sequence of acknowledgement packets in a flow.

The main reasons for using program-style variable names and conventions is that we expect them to be familiar and easy to understand for those familiar with either object-oriented programming languages such as C++ and Java, and to those familiar with Wireshark [Ct06], a popular application for displaying and parsing network traffic. Also, variable names are clearer and more memorable than single-character variables, which is important given the number of flow attributes being discussed.

These conventions also allow us to define our flow attributes in a similar manner to the way they are defined using ANTARES, hopefully making it easier to relate the discussion here to the actual implementation.

We will use F to denote a network flow, or more specifically to denote the sequence of packets that make up the flow. For the half-flows that comprise the two directions of a bidirectional flow, we will use F_{half} for an arbitrary half-flow, and F_{fwd} and F_{rev} for those in the forward and reverse directions, respectively. For reference, recall from section 3.2 that the "forward" direction is from the client to the server.[5]

We will use p to denote a packet, and P for a packet aggregate. A packet, as used here, is any sequence of octets, and can refer to a subsequence of another packet. For example, a

[5]We do not deal with other types of architectures, such as peer-to-peer, in this thesis; for those, different criteria will be needed and will depend on the actual architecture.

TCP packet is a sequence of octets that is usually contained within an IP packet, which is a sequence of octets that is often contained within an Ethernet packet, and so forth. These sequences of octets are generally handled contiguously in memory, whereas sequences of packets (e.g. a sequence of IP packets) refer to logical sequences.

Note that a network flow is, for all intents and purposes, simply a packet aggregate. We use a separate notation for a flow to highlight that it is a packet aggregate with special meaning to a human analyst. Also, some properties, such as the inter-packet delay, are contextual; the "special status" of the flow (and its constituent half-flows) serves as a basis for specifying the context of these properties. This is discussed in more detail below, in the explanation of the "consecutive" filter and natural context.

We will use set notation when discussing packets and packet aggregates; e.g. we will refer to all the packets in a particular packet aggregate P as being $p \in P$. Note, though, that we are dealing with sequences rather than sets, as packets are ordered chronologically according to their arrival time. Thus, we will talk about packet $p_k \in P$ being the kth packet in packet aggregate P, where $\forall_{j<k}\, p_j.time < p_k.time$, where $p.time$ is the arrival time of the packet, as discussed in section 3.3.1.

The notation $P\{filter\,expression\}$ represents a packet aggregate defined as the packets resulting from a filter being applied to a packet aggregate P. The *filter expression* asserts something about the properties of a packet and can be used to select particular packets from a packet aggregate. It is a relation concerning packet properties; e.g. $transport.payload.len$ $>= 5$ is a filter that selects packets with a transport-layer payload of 5 bytes or more. Note that the packet is implicit, so the filter is NOT given as $p.transport.payload.len >= 5$. We also allow compound filter expressions with "AND" and "OR", such as $(transport.payload.$

$len >= 5)\, AND\, (transport.payload.len <= 20)$ to describe a filter that matches a packet with a transport-layer payload length between 5 and 20 bytes, inclusive. For example, $F' = P\{transport.len == 20\}$ defines a packet aggregate P' consisting of the packets in the packet aggregate P in which the transport-layer packet is exactly 20 bytes long (such a 20-byte TCP header with no options and no payload, or a UDP packet with its 8-byte header followed by 12 bytes of payload).

We define a special filter expression "consecutive", which allows the selection of naturally consecutive packets from a previously filtered expression. This requires the definition of a packet's *natural context*, which is a packet aggregate that has been selected to be particularly significant to the analysis being performed. In this thesis, unless otherwise specified, the natural context of a packet is the half-flow to which it belongs. The "consecutive" filter matches every packet in an aggregate for which the previous packet in that flow is also the previous packet in the natural context (or a packet that has no previous packet in either the current flow or in the natural context). It does not include the first packet in a series of consecutive packets unless it is also the first packet in both aggregates; it is intended to be evaluated inline (i.e. evaluated on packets as they arrive), and thus "looking ahead" at the next packet is not allowed. Care must be taken when using this expression, as it can be confusing.

Properties

Properties are loosely defined as some information about a packet. A property has a type associated with it depending on what it refers to; it can itself be a packet (a TCP packet encapsulated within an IP packet can be a property of the IP packet), or it can be a numerical value (integer- or real-valued).

The part of a packet that exists at a given layer of the network stack can be expressed explicitly as a property; e.g. $p.transport$ refers to the data in a packet that pertains to the transport layer (so for a TCP packet, $p.transport$ refers to the data from the start of the TCP header to the end of the application-layer data that is carried as the payload of the TCP packet). The layers of interest for this work are $network$ and $transport$, though $datalink$, $session$, $presentation$, and $application$ may be useful in other contexts. Similarly, for packets that are known to contain headers from a particular protocol, those parts of the packet can be addressed directly, e.g. $p.tcp$ for a TCP packet, which would be equivalent to $p.transport$ for that packet.

The properties $header$ and $payload$ refer to the data in the header and the payload section of a given packet, respectively.[6] It is worth noting that, e.g. for IP and TCP, $p.network.payload = p.transport$.

The len property will denote the total length of a packet, so $p.header.len$ and $p.payload.len$ will refer to the header and payload length of the transport-layer section of the packet, respectively; for the protocols we consider, $p.len = p.header.len + p.payload.len$.[7] Note that in the flow attributes we discuss, we will more commonly specify the layer as well, e.g. $p.transport.payload.len$ for the length of the payload of the transport-layer section of the packet, which for TCP will commonly be the length of the application-layer data carried in the packet (as the session and presentation layers are not commonly used).

We will use $time$ to refer to the arrival time of a packet, and $time\_delta$ to refer to

[6]This could easily be extended to protocols with footers by defining a $footer$ property, but we do not deal with such protocols in this work.

[7]The obvious cases where this would not be true are for protocols which have a ‘trailer’ – data following the payload.

inter-packet delay, the amount of time that has passed since the previous packet in its natural context. For our purposes, the arrival time of a packet at a network monitor is the time that the monitor finished receiving the last byte of that packet. We will deal with two types of inter-packet delay: the *unidirectional inter-packet delay*, where the natural context is the half-flow (i.e. $time\_delta$ is the length of time since the last packet in the same direction), and the *bidirectional inter-packet delay*, where the natural context is the flow (i.e. $time\_delta$ is the length of time since the last packet in either direction). Unless otherwise noted, $time\_delta$ and inter-packet delay refer to the unidirectional inter-packet delay.

Specific protocol headers contain fields that can be named as a property; these will be defined where appropriate. As an example to make this a bit clearer, consider the TCP flags octet. A TCP header contains an octet, each bit of which has some significance; we could refer to that octet as $p.tcp.flags$. Furthermore, we could define properties for each bit, for example $p.tcp.flags.ack$ for the 'Acknowledgement' (ACK) flag. This would be a property with the value of 1 for a packet where the ACK flag is set, and 0 for one where it is not set. This notation is used extensively by WireShark [Ct06], and should be quite familiar to users of that tool.

Functions

A function is a notation that indicates a computation that is to be performed on its arguments. Functions will be used here mostly to express computations across the packets of a packet aggregate. We describe here the functions that will be used in this work when specifying flow attributes.

$pkt\_count(P)$ denotes the number of packets in a packet aggregate P. In this thesis, the aggregates we deal with are at the transport layer and the applications are all carried over TCP, so more specifically $pkt\_count(P)$ will refer to the number of TCP packets in the aggregate. If TCP packets are fragmented across network-layer packets (e.g. IP fragmentation), only the reassembled transport-layer packets are considered.

It should be noted that since only reassembled packets are counted, approaches using such flow attributes can potentially be evaded by fragmentation. Evasion by fragmentation is a known issue in the context of intrusion detection, as discussed by Handley et al. [HPK01], though the implications in this context are slightly different. Handley et al. were concerned mostly about an attacker fragmenting traffic in such a way as to have a sensor misread the content of the session, such as by sending fragments with overlapping payloads that would be reassembled differently by the sensor and by the node under attack. In the context of network traffic classification without payload, an attack would involve the attacker fragmenting traffic in order to cause a sensor to miscompute some flow attributes, e.g. by carefully orchestrating the arrival times of the packets. We expect that the techniques discussed in the former work would not be generally effective against these flow attributes, but we also expect that similar techniques can (and, if this work progresses well enough, will) be devised that are effective. We do not address that issue here, instead relegating it to our “Future work” discussion in section 6.2.

$sum(property, P)$ indicates the sum of some numeric property of the packets of a packet aggregate P. For example, $sum(transport.payload.len, P)$ denotes the sum of the lengths of the transport layer packets of P.

We will also use normal arithmetic operations on numeric properties and on functions that evaluate to a numeric value. For instance, the mean length of the application-layer payloads of the packets in a flow F, *mean_payload_len* can be expressed as in equation 3.1.

$$mean\_payload\_len(F) = \frac{sum(transport.payload.len, F)}{pkt\_count(F)} \tag{3.1}$$

3.3.2 Timing attributes

One prominent feature of network traffic that can be measured is the arrival time of a packet, and thus the inter-packet delays (i.e. the length of time from the arrival of one packet to the arrival of the next). In this section, we discuss flow attributes derived from the arrival times and inter-packet delays of a network flow.

One issue with many timing attributes are the distortions due to network latency and jitter. These distortions are sometimes mentioned as a possible explanation for the observation that machine learning algorithms tend to prefer non-timing-related flow attributes over timing-related ones, but the impact of such distortions has not yet been studied.

Duration

A simple flow attribute which is commonly used is the *duration* of a flow, the length of time it lasts. The *duration* of a flow F consisting of n packets is computed as:

$$duration(F) = p_n.time - p_1.time \tag{3.2}$$

Some of the approaches we discussed earlier use duration in their classification efforts. Frank [Fra94] uses the duration of a flow as one of his features, which was selected as useful for all of his classification problems. Roughan et al. [RSSD04] found that the duration of flows was one of the most useful attributes for distinguishing between the applications that they studied; we discuss this aspect of their work further in section 5.2.

Inter-packet delay

The inter-packet delay is the length of time between the arrival of one packet and the next. Some flow attributes can be computed using this measurement; we discuss a few of them here. Recall from section 3.3.1, there are two types of inter-packet delay: unidirectional (between packets in the same direction), and bidirectional (between any two packets in a flow). In this work, we mainly consider unidirectional inter-packet delays, as we expect that this will reveal more about the activities of the application generating the traffic, and not be distorted as strongly by network congestion and similar effects.

The most straightforward flow attribute based on inter-packet delay is the *mean inter-packet delay* of a flow F, $mean\_delay(F)$. This is expressed as:

$$mean\_delay(F) = \frac{duration(F)}{pkt\_count(F)} \tag{3.3}$$

Early et al. [EBR03] used the mean inter-packet delay in modelling the behaviour of server flows in order to detect anomalies. They note issues with a single large delay between packets can drastically change the mean inter-packet delay and make a flow much more difficult to classify; they term this the *water cooler effect*, as a user leaving their terminal for some time and returning later to the previous activity could easily cause such a delay.

Another flow attribute that could be used is the *inter-packet delay variability*, a measure of how widely varied the inter-packet delays are within a flow. We do not use it in this work, however, as we have not found a definition that can be computed in linear time.

Roughan et al. [RSSD04] did use a metric based on the variability of inter-packet delays to help distinguish between FTP-data and streaming media. The metric they used was based on the standard deviation of inter-packet delays divided by the mean inter-packet delay, where the delay was between any two packets in the bidirectional flow (rather than between two packets in the same direction). They found that this measure, combined with average packet length, was able to distinguish fairly well between the two types of traffic; combining their variability metric with duration and mean packet length, they found they could distinguish quite well among FTP-data, HTTP, and streaming media, with error rates of 0 for several of the classifiers that they used. As with the other attributes they considered, they appear to have used daily averages rather than values from individual flows in their experiment.

3.3.3 Packet lengths

One of the most basic measurements available from network traffic is the length of the packets that make up a flow. We consider here flow attributes based on packet lengths.

Three flow attributes based on packet length, for a flow F, are the *mean packet length*, $mean\_pkt\_len(F)$, the *mean payload length*, $mean\_payload\_len(F)$, and the *mean nonempty payload length*, $mean\_nonempty\_payload\_len(F)$. The first is computed as:

$$mean\_pkt\_len(F) = \frac{sum(transport.len, F)}{pkt\_count(F)} \tag{3.4}$$

$mean\_payload\_len(F)$ is the same metric except that it uses the property $transport.payload.len$ in place of $transport.len$. The third attribute, $mean\_nonempty\_payload\_len$, is defined as:

$$P_{nep} = F\{transport.payload.len > 0\} \tag{3.5}$$

$$mean\_nonempty\_payload\_len(F) = \frac{sum(transport.payload.len, P_{nep})}{pkt\_count(P_{nep})} \tag{3.6}$$

P_{nep} is thus the sequence of non-empty packets (packet which contain some application-layer data), which allows us to ignore empty packets such as TCP ACKs that are not directly generated by the application and could be considered to be distortions.

The mean packet length and similar flow attributes are used fairly often in traffic classification approaches. Roughan et al. [RSSD04] considered the mean packet length, along with duration, to be one of the most useful attributes among those they examined. We discuss their use of this attribute further in section 5.2.

The mean packet size doesn't give an indication of how regular or irregular the packet sizes are; for this, measures of the variability of the packet sizes could be used as flow attributes. However, we are focusing on flow attributes that can be computed in linear time, and we have not found an appropriate way to compute these measures, so we do not use them in this thesis.

A common measurement of variability is standard deviation, so one possible flow attribute would be the *standard deviation of packet size*.

In addition to the flow attributes described above, there are a number of heuristics based at least in part on packet lengths that we will use. These are described in section 3.3.5.

Wright et al. [WMM04] used packet lengths as well, in several different ways. They compute a flow attribute from the packet lengths, they trained a Hidden Markov Model using packet lengths as an input, with a fuzz factor to simulate the obscuring effect of a block cipher, where packets are padded out to an integer multiple of the block length.

3.3.4 Data volume

Data volume attributes are concerned with the amount of data in a network flow; this category includes attributes based on total data volume of a flow and on data rates. Total data volume attributes pertain to an entire network flow (up to the point at which the flow attribute is being computed), whereas data rate attributes focus on the amount of data being transferred per unit time. The most intuitive measure of data volumes is the number of bytes being sent, but we also consider here attributes based on the number of packets being sent.

Total data volume

A simple class of flow attributes are total data volume attributes, which measure the overall amount of activity involved in a network flow. We describe here packet counts and byte counts, and discuss some of the variations on each theme.

The *packet count* of a network flow is simply the number of packets that make up the flow, e.g. for a flow F, the packet count is $\mathit{pkt\_count}(F)$ (defined in section 3.3.1).

The *nonempty packet count*, $\mathit{pkt\_count\_nonempty}(F)$ of a network flow F is the count of transport-layer packets that carry application data, i.e. those packets p where $p.\mathit{transport}.$

$payload.len > 0$. In our notation, this is expressed as:

$$pkt\_count\_nonempty(F) = pkt\_count(F\{transport.payload.len > 0\}) \tag{3.7}$$

Frank [Fra94] used the forward and reverse packet counts in his feature selection experiment, finding that both were used in most of the selected feature sets, with the reverse packet count used for all of the classifiers.

Another group of flow attributes that seem promising are byte counts. The major question with byte counts is deciding which bytes to count. We have the option of counting only the bytes in the payload of the transport-layer packets; this can be useful, as it generally includes only data actually sent by the application itself. However, it can sometimes be necessary to consider all of the data from the network-layer up, particularly if there is some sort of encrypted tunneling mechanism being used that obscures the transport-layer header and thus prevents the calculation of the payload length of the packet.

We consider the *packet byte count* and the *payload byte count*; these two flow attributes will be computed in a similar manner, except that in the former case, the length of the entire transport-layer packet will be used, and in the latter, the length of only the payload part of the transport-layer packet will be used. The packet byte count of a flow F, $pkt\_byte\_count(F)$ and the payload byte count of a flow F, $payload\_byte\_count(F)$ are defined by:

$$pkt\_byte\_count(F) = sum(network.length, F) \tag{3.8}$$

$$payload\_byte\_count(F) = sum(transport.payload.length, F) \tag{3.9}$$

Frank [Fra94] used forward and reverse payload byte counts in his feature selection experiment, finding that the forward payload bytes was used mostly for classifying SMTP, and that the reverse payload bytes was rarely used at all. However, this may be due to the limited set of traffic types he used.

Data rate

Closely related to data volume flow attributes are data rates; i.e. measurements of the amount of data that an application is sending and receiving per unit time. There are several distinct flow attributes that can be computed based on data rates. They have been shown to be useful in traffic classification experiments, and though they can be evaded, such evasion is likely to significantly affect other flow attributes. Here, we describe the data rate flow attributes in question and give an analytic evaluation of them.

There are two parameters to be considered in defining data volume attributes: the *time granularity*, and the data of interest. The time granularity is the length of each interval over which the data rate is calculated, e.g. per second or per five seconds. Note, however, that the data rate will be reported in units of bytes per second, regardless of the time granularity. The data of interest indicates what data we are measuring, generally at what layer we're measuring it. For example, the data rate as calculated at the IP layer will be different from the data rate calculated in terms of TCP payload (the amount of data in payload portions of TCP packets); in this example, the former would include not only the extra data in the IP and TCP headers, but would also count TCP acknowledgement packets, which may be a distortion in many contexts.

We consider several data-rate-based flow attributes.

The *mean data rate* of a flow F, $mean\_datarate(F)$, is simply the amount of data of interest sent in bytes divided by the duration of the flow in seconds, as formalized in equation 3.10. Note that we use the payload byte count as defined in section 3.3.4, i.e. the amount of data sent by the application, not including network or transport layer headers; we could alternately use the packet byte count if the payload byte count was not available, but we do not include that here.

$$mean\_datarate(F) = \frac{payload\_byte\_count(F)}{duration(F)} \tag{3.10}$$

The *directionality of data* is, for a bidirectional flow, a ratio between the average data rates of its half-flows; for example, a bidirectional flow with a forward average data rate of 5Kb/s and a reverse average data rate of 20Kb/s would have a directionality of data of 0.25 (in the forward direction). We define it as:

$$dir\_data(F) = \frac{payload\_byte\_count(F_{fwd})}{payload\_byte\_count(F_{rev})} \tag{3.11}$$

Note that the result is undefined if there is no application data on the reverse side, a case which has to be handled carefully.

The logarithm of the directionality of data is used by Hernández-Campos et al. [HCNSJ05] in their work on clustering network flows.

Another interesting set of possibilities for flow attributes based on data rates are those designed to detect regularity in the data rate, such as those used by DeMontigny-LeBoeuf [DL05] to characterize streaming media. These would give values based on the amount of variability in the data rate, and would probably be well-suited for detecting applications that attempt to maintain a certain fixed rate of data flow, particularly streaming media applications. However, we do not address them here, as we have not found a way to compute them in linear time.

3.3.5 Packet proportion heuristic attributes

We encountered a number of heuristic attributes in the literature that were aimed at identifying particular types of behaviour. In this section, we will discuss general forms of the most common types of heuristics: those based on small or large packets, and those based on packet flags.

A number of the heuristics which have been used in the literature are based on computing the proportion of the packets within a network flow having a particular characteristic. Heuristics based on packet flags, Zhang and Paxson's [ZP00] heuristics for detecting interactivity, and many of DeMontigny-LeBoeuf's [DL05] heuristics fall into this category, as do the bins of Dunigan and Ostrouchov [DO01]. We present a generalized description of these types of heuristics and review a number of possible flow attributes based on them.

In general, such a flow attribute will be a calculation h based on two packet count flow attributes c_1 and c_2 as described in section 3.3.4, where $c_2 \leq c_1$; it is computed as $h = c_2/c_1$, so $0.0 \leq h \leq 1.0$. For example, if we consider any packet with a payload length of 60 or less to be a small packet, we could compute the proportion of small packets: Let c_{small} be the number of small packets in a given flow, let c_{total} be the total number of packets in a flow, then the proportion of small packets in the flow h_{small} would be $h_{small} = c_{small}/c_{total}$.

Small packet heuristics

Some heuristics that have been seen to be effective at detecting command-shell interactive behaviour are *small packet heuristics* [ZP00]. These are measurements of the proportion of packets in a flow that meet some criterion for being small, e.g. all packets with 20 bytes or less of payload.

The most notable examples of small packet heuristics are those used to detect command-shell interactive activity by Zhang and Paxson [ZP00], and DeMontigny-LeBoeuf's [DL05] adaptation of them. The approach taken by Dunigan and Ostrouchov [DO01] is also related; they effectively attempt to develop a deterministic method for creating such heuristics.

Zhang and Paxson combined two metrics, which they called α and Γ, to detect back-doors. Both are ratios computed based on the occurrences of consecutive small packets, where a consecutive small packet is a packet that is below some length threshold, which we will call λ_{max_len} (20 bytes, in their work) and that follows another packet that is also a small packet. They do not specify whether that is consecutive in the same direction, so we assume that it considers the previous packet in either direction. For our heuristics, we incorporate an optimization from DeMontigny-LeBoeuf's work, which is to not consider packets with no transport-layer payload (e.g. TCP ACK packets) as being small packets. DeMontigny-LeBoeuf also defined β and δ heuristics to complement the above two.

The α heuristic indicates how many of the delays between consecutive small packets are between 10ms and 2s (though they mention that the upper bound could just as well be 100s or more). We will use λ_{min_delay} and λ_{max_delay} to refer to the lower and upper bounds on the inter-packet delay, respectively. The intuition behind using the inter-packet delay is to exclude consecutive small packets resulting from machine-driven activity (which, they reason, would be sent with very short inter-packet delays). We express below α for a flow F as $sp\_alpha(\lambda_{max\_len}, \lambda_{min\_delay}, \lambda_{max\_delay}; F)$ using our notation.[8] We first define some subsequences of F, the sequence of nonempty packets P_{nep}, the sequence of small packets P_{sp}, the sequence of consecutive small packets P_{sp}, and the sequence of consecutive small packets with certain delays, P'_{csp}.

$$P_{nep} = F\{transport.payload.len > 0\} \tag{3.12}$$

$$P_{sp} = P_{nep}\{transport.payload.len \leq \lambda_{max\_len}\} \tag{3.13}$$

$$P_{csp} = P_{sp}\{consecutive\} \tag{3.14}$$

$$P'_{csp} = P_{csp}\{time\_delta \geq \lambda_{min\_delay} AND time\_delta \leq \lambda_{max\_delay}\}) \tag{3.15}$$

$$sp\_alpha(\lambda_{max\_len}, \lambda_{min\_delay}, \lambda_{max\_delay}; F) = \frac{pkt\_count(P'_{cs})}{pkt\_count(P_{cs})} \tag{3.16}$$

This heuristic indicates how much of the consecutive small packet activity appears, by timing, to be interactive human-driven activity; if the result is undefined due to there not being any consecutive small packets, we simply use a value of 0 to indicate that there is no apparent human-driven activity based on such packets.

It is important to recall from section 3.3.1 that the property $p_k.time\_delta$ is the time delay of packet p_k since the previous packet in its natural context, the original half-flow F_{fwd} or F_{rev}, not since the previous packet in the packet aggregate P_{csp}. Similarly, recall from section 3.3.1 that the filter expression "consecutive" is a special term that only matches a packet if the previous packet to be tested against that term was consecutive to that packet in

[8] Note that if there are no consecutive small packets in F, this value is undefined.

its context (again, in the original half-flow F_{fwd} or F_{rev}, not in P_{sp}). When computing one of these small packet heuristics, we will consider the natural context of a packet to be the bidirectional flow if the heuristic is computed on an aggregate with packets in both directions (this will usually be the bidirectional flow itself), or the half-flow if it is computed on an aggregate with packets in only one direction (this will usually be the half-flow itself).

The β heuristic is simply the proportion of small packets among the non-empty packets in the flow, defined, with P_{nep} and P_{sp} computed as above, as:

$$sp\_beta(\lambda_{max\_len}; F) = \frac{pkt\_count(P_{sp})}{pkt\_count(P_{nep})} \tag{3.17}$$

This is the simplest of these heuristics, merely aimed at representing the overall proportion of small packets. Note that it can be undefined if there are no non-empty packets in the flow; however, if all the packets in the flow are empty, we can state that this heuristic does not show any evidence of interactivity, and use a value of 0.

The γ heuristic[9] is the proportion of consecutive small packets; we use DeMontigny-LeBoeuf's version, which uses the proportion of consecutive small packets among non-empty packets, rather than among all packets. It is defined as:

$$sp\_gamma(\lambda_{max\_len}, \lambda_{min\_delay}, \lambda_{max\_delay}; F) = \frac{pkt\_count(P_{csp})}{pkt\_count(P_{nep})} \tag{3.18}$$

As with the β heuristic, we take a lack of nonempty packets to be a lack of evidence for interactivity and use the value 0 to replace an undefined value.

[9]Zhang and Paxson used a capital gamma (Γ), whereas DeMontigny-LeBoeuf used a lowercase gamma (γ). We also use the lowercase gamma for consistency with the other Greek letters.

The δ heuristic is a slight modification of the β heuristic; it uses the proportion of small packets among small and empty packets. The rationale is to distinguish applications that send mostly small non-empty packets (such as interactive applications) from those that send many empty packets as well as small packets (such as some machine-driven applications). It is defined, using $P_{ep} = F\{transport.payload.len == 0\}$ and P_{sp} defined as for the sp_alpha heuristic, as follows:

$$sp\_delta(\lambda_{max\_len}, \lambda_{min\_delay}, \lambda_{max\_delay}; F) = \frac{count(P_{sp})}{count(P_{ep}) + count(P_{sp})} \qquad (3.19)$$

If there are neither empty nor small packets in the flow, we once more consider that to be a lack of evidence of interactivity, and use the value 0 instead of an undefined value.

Note that these metrics are based on the $transport.payload.len$ property, which requires that the transport-layer header be accessible. Similar metrics could be defined based on $network.len$, for example, with suitably higher values for the λ_{max_len} threshold,[10] for situations in which the transport-layer header is not available.

When we use small packet heuristics in this thesis, we compute them on the half-flows F_{fwd} and F_{rev} rather than on the full bidirectional flow. We are looking for interactive activity from at least one side of the communication, so we wish to look at each side in isolation. As an example to clarify, suppose that a bidirectional flow F contains three packets p_{k-2}, p_{k-1}, and p_k, where p_{k-2} and p_k are in the forward direction and p_{k-1} is in the reverse direction, and that the $transport.payload.len$ for the three packets are 17, 1460, and 15, respectively. We wish to compute sp_gamma with $\lambda_{max\_len} = 20$; we will ignore

[10]e.g. 40 bytes more, 20 for the IP header and 20 for the TCP header, or more if header extensions appear to be in use

the timing constraints for this example. If we wish to compute *sp_gamma* for F, then both p_{k-2} and p_k would be in P_{sp}, but since they are separated by p_{k-1}, neither is in P_{csp}, as they are not consecutive in their natural context of the bidirectional flow, as discussed in the description of sp_alpha. However, if we instead compute *sp_gamma* for F_{fwd}, they are again both in P_{sp}, and this time the natural context of the packets is the half-flow F_{fwd}, so p_{k-1} is ignored (as it is in the half-flow F_{rev}) and they are consecutive. Thus, at least p_k (and possibly p_{k-2}, depending on the preceding traffic) would be in P_{csp}.

Large packet heuristics

Analogous to the small packet heuristic is the *large packet heuristic*. This is a class of potential flow attributes that are aimed at identifying bulk data transfer behaviour.

The heuristics presented here are those of DeMontigny-LeBoeuf [DL05]. We define here her file-transfer heuristics α, β, and γ in our own notation. These use the concept of consecutive large packets, which are packets above a certain length threshold that follow other packets above that threshold. They also look for short inter-packet delays to indicate machine-driven activity (contrast that with the small packet heuristics, which look for longer delays to exclude such activity). We use the terms λ_{min_len} and λ_{max_delay} to denote the minimum length and maximum delay parameters, respectively. DeMontigny-LeBoeuf used 225 bytes and 50 ms for these parameters, respectively.

The α heuristic computes the proportion of consecutive large packets having short delays. The intuition behind this is that applications performing bulk data transfer are often sending data without interaction with the remote host, and thus will tend to send packets with very short delays between them. We define this heuristic for a flow F as

$lp\_alpha(\lambda_{min\_len}, \lambda_{max\_delay}; F)$, first defining the sequence of large packets P_{lp}, the sequence of consecutive large packets P_{clp}, and the sequence of consecutive large packets with short delays P'_{clp}:

$$P_{lp} = F\{transport.payload.len \geq \lambda_{min\_len}\} \tag{3.20}$$

$$P_{clp} = P_{lp}\{consecutive\} \tag{3.21}$$

$$P'_{clp} = P_{clp}\{p_k.time\_delta \leq \lambda_{max\_delay}\} \tag{3.22}$$

$$lp\_alpha(\lambda_{min\_len}, \lambda_{max\_delay}; F) = \frac{pkt\_count(P'_{clp})}{pkt\_count(P_{clp})} \tag{3.23}$$

Similarly to the sp_alpha heuristic, this takes an undefined value if there are no consecutive large packets; we consider this to be a lack of evidence of bulk transfer and substitute the value 0 for an undefined result from this calculation.

The β heuristic computes the proportion of large packets in a flow. Using the sequence of large packets P_{lp} defined above, and defining the sequence of nonempty packets P_{nep}, we express this for a flow F as:

$$P_{nep} = F transport.payload.len > 0 \tag{3.24}$$

$$lp\_beta(\lambda_{min\_len}; F) = \frac{pkt\_count(P_{lp})}{pkt\_count(P_{nep})} \tag{3.25}$$

The γ heuristic used indicates the proportion of consecutive large packets in a flow. Using the sequence of consecutive large packets P_{clp} as above, we express this as:

$$lp\_gamma(\lambda_{min\_len}; F) = \frac{pkt\_count(P_{clp})}{pkt\_count(F)} \tag{3.26}$$

A notable issue with large packet heuristics is fragmentation, particularly at the transport layer. Fragmentation leads to large chunks of data sent by an application being split into many packets of the same maximum size; while this distortion can be useful as an indicator of large chunks of data being sent simultaneously, it makes it difficult to analyze the sizes of the chunks of data actually sent by an application. The approach of Hernández-Campos et al. [HCNSJ05] to reassembling Application Data Units (ADUs) from transport-layer packets based on changes in direction is one way to get at the sizes of the data chunks actually sent by the application; we do not address this issue further in this thesis, but leave it to future work.

Dunigan and Ostrouchov's [DO01] approach of binning the space of possible lengths, timings, and directionalities was intended to provide a deterministic way to discover features such as these, without relying on experts to construct heuristics. Their selection of boundaries for the bins, however, was itself based on studying the traffic. We suggest that the same goal could be achieved with the heuristics presented here by using data to search for optimal values for the parameters (λ_{min_len}, λ_{max_len}, λ_{min_delay}, and λ_{max_delay}), e.g. using machine learning or genetic algorithms.

Wright et al. [WMM04] used a technique called vector quantization to define bins, which could function as the basis for an alternate method to develop small and large packet heuristics. To oversimplify, this technique uses a clustering algorithm on a particular data set to discover meaningful regions in the space of packet lengths, inter-packet delays, and directions. They use it for a different purpose, to quantize a multi-dimensional space of possible packets down to a smaller, finite number of transitions for a Hidden Markov Model, but it may be applicable to this purpose as well. We have not used either this or the previous approach in this work for establishing the parameters for the various heuristics, but suggest that future work that is more focused on such flow attributes would want to investigate them.

Packet flag heuristics

A set of flow attributes that has been used by several approaches [EBR03, TAF01] is based on analysis of the TCP packet flags of the packets making up the flow. These flags are used by TCP to signal control information to the TCP implementation at the remote end; though this doesn't seem like it would be directly related to the application itself, it could potentially reflect the way in which the application is using TCP.

Such a heuristic would simply be the number of packets with the specified flag, e.g. ACK or PSH, divided by the total number of packets in the flow.[11] As with many such heuristics, packet flag heuristics can only be used in a traffic classification system if the system has access to the TCP header.

Early et al. [EBR03] included heuristics based on packet flags in modelling server flows for intrusion detection; though they do not comment on their usefulness, example rules that they provided appeared to use them extensively.

[11]ACK is the acknowledgement flag, used by TCP to indicate data received from the remote node, and PSH indicates that the data in the packet should be sent to the receiving application immediately rather than being buffered with subsequent data, similar to flushing a buffer [Pos81b].

Chapter 4

The ANTARES tool

To facilitate the study of flow attributes and application behaviours, we have developed a tool, the Advanced Network Traffic Analysis Research and Exploration Suite, or ANTARES. This tool is designed to provide powerful mechanisms for implementing the measurement of a wide variety of flow attributes and for handling network traffic in a flexible manner. In this section, we describe the design and architecture of the tool and its features, and we present an overview of how it works, to better explain how flow attributes can be constructed and computed. We first give an overview of ANTARES and some of the design decisions we made during its construction, and explain how it differs from NetMate, a similar tool built by Zander and Schmoll [ZS05]. We then describe its architecture and how some of the basic mechanisms work. Finally, we walk through the process of implementing some of the flow attributes listed in table 3.1 with the tool, as a guide to others who would use it, and discuss the functionality of the program that we used to produce the flow attribute values used in Chapter 5.

ANTARES is written in C++, which seemed to be the best compromise of design and performance. Java was considered, but we rejected it due to concerns that networking researchers would not be willing to use a Java tool, out of fear of poor performance. We also wished to use an object-oriented architecture as it seemed appropriate to the problem; the structures involved seemed to map naturally onto class hierarchies (TCP and UDP packets are transport-layer packets, transport- and network-layer packets are packets; different types of flows and sequences of packets are all aggregates, etc.), and encapsulating functionality with data was useful when designing flow attribute objects.

The architecture of our tool allows arbitrary code to be included in the computation of a flow attribute, using a "listener" paradigm. Flow attributes are notified when a packet is added to an aggregation, and they then perform computations on the added packet. These attribute objects can be arbitrarily complex, and they can be stateful; thus, arbitrarily complex flow attributes can be computed, so long as they only require information from the stream of packets that they see. Care must be taken to manage resources, of course, as there are currently no safeguards to avoid having the code of an attribute consume arbitrary amounts of resources. Once these flow attributes are set up and the data processed, the program using the tool can request values from a flow attribute using a descriptive string, the meaning of which is interpreted by the code of the flow attribute. This mechanism is described in more detail in section 4.4.

ANTARES is not yet as user-friendly as it is intended to be; the creation of new flow attributes currently requires that the user write C++ code to access functions implemented in a library. This is generally just a matter of creating instances of classes that do the work and providing them with appropriate parameters, however; examples of this process are given in section 4.4. Efforts have been made to simplify the process of creating new flow attributes and to make it easier for less experienced C++ programmers to avoid some of the common pitfalls. Our goal, however, is to make it possible to define a wide range of flow attributes using a configuration file rather than requiring users to modify the source code. Although the notation used in this thesis coincides with the programming constructs used by the tool, the process of translating from the former to the latter is not yet as easy as we would like; as future work, a parser should be created to do this translation, such that flow attributes expressed using our notation could be compiled directly to modules for the tool.[1]

ANTARES was written as a research tool intended to help with the understanding of network traffic and application behaviours, and not as a production traffic monitoring system, so performance has not been a primary consideration. The performance is adequate for the samples we have been dealing with (consisting of 5 minutes of a single flow), but has suffered with the addition of more complex flow attributes, particularly time-based attributes that require a large number of signals being passed between components. Prior to the introduction of these time-based attributes, the performance had been reasonable even on large (about 1Gb) network traces; with some tuning, that should be possible again.

Our impression after using ANTARES to implement many of the flow attributes described in section 3.3 is that it does make it straightforward for a programmer who understands the tool to implement a wide variety of such attributes. The major drawback at this point is the internal mechanisms of the data structure that handles network traffic are more complex than we would like, partially because it was designed with several different purposes in mind, such as being used for interactive traffic analysis, which added complex-

[1]More accurately, the parser would take as input flow attributes expressed using a variant of our notation adapted to be expressed in plain text.

ity; however, this complexity is fairly well hidden from a programmer mainly interested in creating flow attributes. The interface for defining the attributes is somewhat inelegant at the moment as well; it was designed to be driven by a parser reading a configuration file that defines the flow attributes. This will be discussed in more detail in section 4.4.

ANTARES is available as a SourceForge project [Fur06],[2] licensed under the GNU Public License [Pro06a].

4.1 Comparison of ANTARES and NetMate

Zander and Schmoll [ZS05] built NetMate, a tool for computing the values of flow attributes; it is somewhat unfortunate that we did not discover this tool earlier, as it appears that it would have been useful as a basis for ANTARES. However, there are fundamental differences between the approaches taken by each of these tools; as a result, ANTARES offers more flexibility and expressive power than NetMate, and the interface is aimed at programmers familiar with an object-oriented programming style. As NetMate appears to be the tool most similar to ANTARES, we present here a discussion of these differences and the relative merits of ANTARES.

The basic difference between NetMate and ANTARES is the way in which each handles flows and flow attributes. NetMate is an application for computing independent flow attributes; each attribute or set of attributes is developed as a module that can be plugged into the application and that performs its calculations directly on packets passed to it by the application. In ANTARES, on the other hand, flows are expressed as aggregates of pack-

[2] SourceForge is a publically accessible Internet site devoted to collaborative open source software development.

ets, and flow attributes are attached to these aggregates as listeners; these flow attributes are given names and can be referenced by other flow attributes.

The approach taken by ANTARES permits greater flexibility and power in defining flow attributes than that of NetMate. One advantage is that it becomes easier to adapt a set of flow attributes to a different definition of a flow when only a few of the attributes are directly examining packets, and the other attributes are using the results of those lower-level attributes; in that case, only those lower-level attributes need to be changed. Another is that if a researcher wishes to introduce a new flow attribute that compares two separate attributes, those existing attributes need not be merged into a single module first, the new attribute needs to simply reference the existing attributes. Furthermore, ANTARES allows attributes attached to different but related aggregates to interact; for example, if a researcher wished to define an attribute for a bidirectional flow that used values of attributes attached to its component single-directional flows (e.g. the forward and reverse directions of a TCP session), they could attach a new attribute to the bidirectional flow and reference the attributes attached to the single-directional attribute, rather than having to re-implement those values in the bidirectional flow.

ANTARES was designed to emphasize an object-oriented approach to creating flow attributes and dealing with network flows. Flows and flow attributes are treated as first-class objects within ANTARES, which allows flow attributes to be expressed in a manner that we expect to be more intuitive to some researchers than the procedural paradigm of NetMate. Also, the components of ANTARES are loosely coupled and generally self-contained. For example, the mechanisms by which packets are processed, reassembled, and sorted into aggregates can be altered or replaced independently of the other components; it is thus straightforward to change the definition of a flow being used by a particular application built using ANTARES.

We deliberately emphasized power over performance when designing ANTARES. The performance requirements that we have set for it are not stringent; for a set of flow attributes of a similar size and complexity of those listed in tables 4.1 and 4.2 and running on a commodity computing platform, it must be able to process a 100kB sample within 5 seconds for debugging purposes, and it must be able to process an appreciable data set of 2GB overnight (i.e. within 12 hours).

As discussed, the primary distinction between ANTARES and NetMate is that ANTARES sacrifices efficiency of implementation for flexibility and power of expression. Because of this trade-off, we consider our tool to be more appropriate for defining experimental flow attributes in order to evaluate their potential, and NetMate to be more appropriate for implementing proven flow attributes in a production setting.

4.2 Requirements

We present in this section an analysis of the requirements of the ANTARES toolkit; these inform the design of the library and will be referenced as motivating the various design decisions in section 4.3, which describes its architecture. We first document the use case of interest, that is, the context in which we expect the library to be used, and for what purpose we expect it to be used in that context. We then describe the requirements for the toolkit derived from that use case; the functional requirements, which describe the concrete tasks that the toolkit is to perform, and the nonfunctional requirements, which are qualitative goals guiding the design of the toolkit.

4.2.1 Use case

The main tasks for which ANTARES was developed are those where a researcher defines flow attributes that they wish to study, and that where they compute the values of those flow attributes on sample network flows. These fit within the researcher's broader task of attempting to find meaningful flow attributes for distinguishing between two or more different types of network traffic. We describe here in more detail the overall analysis process and the role played by these tasks.

Figure 4.1 depicts the analysis tasks in the analysis process and indicates where ANTARES fits into it, using a Unified Modeling Language (UML) use case diagram.[3] Prior to the use of ANTARES, the researcher would first have defined the types of traffic among which they wish to distinguish (**UC-1**), and obtained samples of those types of traffic (**UC-2**) in the network context or contexts of interest, for example by using a network capture tool such as tcpdump [Dev06a]. They would then analyze the traffic samples (**UC-3**), possibly by using an analysis tool such as Wireshark [Ct06], and label them with the corresponding traffic types (**UC-4**).

Having established a data set consisting of labelled samples of traffic, they would then turn those into labelled feature vectors. They would first have to decide on a set of flow attributes to be used as the elements of the feature vectors and define them (**UC-A1**), and then compute the values of those attributes for their labelled samples (**UC-A2**). These two

[3]UML is a modelling language for software design and architecture defined at the Object Management Group [Obj07].

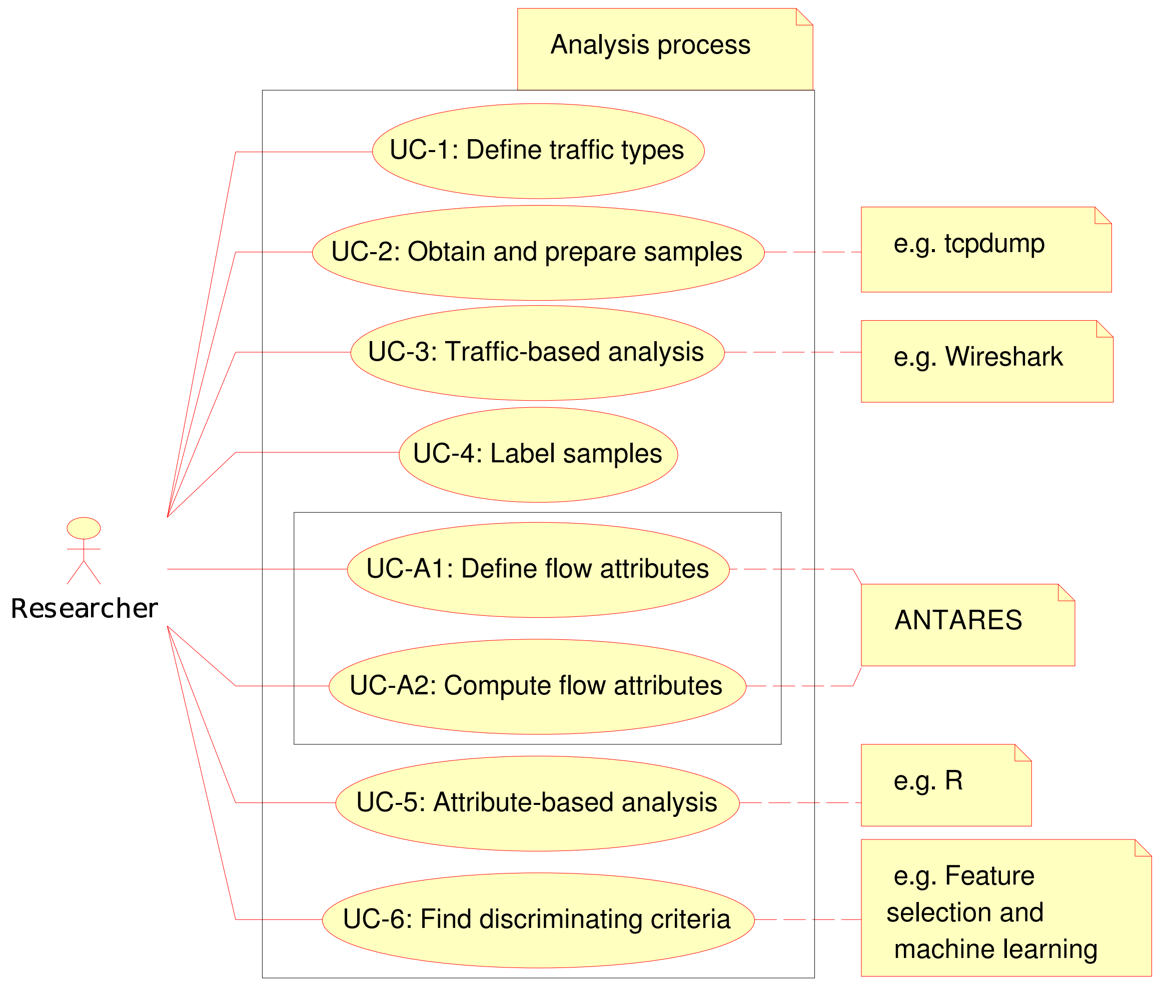
Analysis process
UC-1: Define traffic types
UC-2: Obtain and prepare samples
e.g. tcpdump
UC-3: Traffic-based analysis
e.g. Wireshark
UC-4: Label samples
Researcher
UC-A1: Define flow attributes
UC-A2: Compute flow attributes
ANTARES
UC-5: Attribute-based analysis
e.g. R
UC-6: Find discriminating criteria
e.g. Feature selection and machine learning

Figure 4.1: UML Use Case diagram depicting the tasks involved in the analysis process

tasks are the use cases for which ANTARES is intended to be used, and the remainder of the design process will focus on them.

The researcher would then use these feature vectors to determine how to discriminate between the types of traffic. They might analyze the vectors using data modelling and statistical tools, such as R [Dev05], to develop a qualitative understanding of the data (**UC-5**). The researcher would then use quantitative tools, such as feature selection algorithms and machine learning algorithms described in section 2.1.1, to find a mechanism for distinguishing among the different types of traffic (**UC-6**).

Although this process has been presented as a linear sequence, it would not necessarily proceed that way in reality. For example, a researcher who finds that their feature vectors do not adequately distinguish the types of traffic may well define and compute additional flow attributes; similarly, one who finds that their samples are not sufficiently representative may obtain additional samples and repeat the process on those. The analysis steps in particular (**UC-3** and **UC-5**) would likely be performed repeatedly at different stages of the process to inform choices of parameters and algorithms used in other tasks, and to judge the success of those other stages.

The two main tasks for which ANTARES is designed are the specification of flow attributes (**UC-A1**) and the computation of attributes so specified (**UC-A2**). The other use cases presented here are beyond the scope of this tool, although we intend for some components of the library to be capable of being reused for some of them, in particular for presenting the computed flow attributes for analysis (**UC-5**). The remainder of this section will document the requirements for these tasks. These requirements will be referenced later, in our discussion of the architecture of the tool, to explain the rationale behind the design decisions that were made during its development.

4.2.2 Functional requirements

We list here the functional requirements motivating the design of ANTARES. These requirements proceed from the use cases **UC-A1** and **UC-A2**, defining flow attributes to be computed and actually computing the values of those flow attributes, respectively. We also include some future functional requirements; these are not intended to be met by the current implementation of ANTARES, but they influence the design decisions taken, as it is expected that these will eventually need to be met by the tool and its components.

UC-A1: Define flow attributes

These requirements follow from the use case **UC-A1**, in which the researcher specifies the flow attributes to be computed. At this point in time, this is accomplished by writing a program that does this by instantiating objects of the appropriate classes via the ANTARES Application Programmer Interface (API), but the requirements are written to allow this to later be done by writing a configuration file in a special-purpose grammar and invoking a parser (which would be part of ANTARES) to read that file.

FR-A1-1 The user must be able to specify attributes to be computed.

FR-A1-2 The user must be able to specify attributes in terms of other attributes within the same flow or subflows thereof.

FR-A1-3 ANTARES must provide a mechanism for identifying and referencing packet properties.

FR-A1-4 ANTARES must provide the user the ability to define several basic types of attributes with minimal effort (e.g. invoking a constructor). These are:

- **FR-A1-4.1** An attribute that counts packets.
- **FR-A1-4.2** An attribute that keeps a running sum of a specified (numerical) property of packets (e.g. length).

FR-A1-4.3 An attribute that performs an arithmetic operation (addition, subtraction, multiplication, or division) on two values; each of these values can be either the value of another attribute or a constant.

FR-A1-4.4 An attribute that stores values of packet properties or of other attributes as specified by the user.

FR-A1-5 Attributes can have sub-attributes (attributes subordinate to the containing attribute), which should be accessible by anything that can access the attributes itself.

FR-A1-6 Any attribute must be able to have a filter specified for it; packets not matching that filter must not affect the value of the attribute or its sub-attributes.

FR-A1-7 ANTARES must provide a way to specify that a packet filter is to be applied to a packet sequence.

UC-A2: Compute flow attributes

These requirements are based on the use case **UC-A2**, in which the researcher computes the flow attributes that they have specified in use case **UC-A1** on samples of network traffic.

FR-A2-1 ANTARES must accept as input data files containing recorded packets, and organize those packets into aggregates according to the commonly used 5-tuple model of a network flow (protocol, network-layer addresses, transport-layer ports).

FR-A2-2 ANTARES must compute the values of the specified attributes on the network traffic input to it.

FR-A2-2.1 ANTARES must be able to compute the values of the basic attributes described in FR-A1-4.

FR-A2-2.2 ANTARES must be able to make available the value of a flow attribute to another flow attribute that references it (e.g. for the attribute described in FR-A1-4.3).

FR-A2-3 ANTARES must enforce an update order among attributes, so that an attribute that is referenced by another attribute is updated before its value is used by the latter attribute.

FR-A2-4 ANTARES must track the history of a packet; that is, for a given type of packet, ANTARES must be able to access the ancestors (lower-layer packets from which the current packet was computed) and the descendants (higher-layer packets including data from the current packet) of that packet. For example, for a TCP packet that was fragmented across multiple IP packets, ANTARES must be able to access all of the IP packets that were reassembled to produce that TCP packet.

Future requirements

The design of ANTARES is also influenced by an intention for the library to eventually support an interactive traffic analysis tool, along the lines of Wireshark [Ct06], but focused on displaying non-payload information as opposed to parsing and displaying the payload data (e.g. a tool appropriate to use case **UC-5**). The functional requirements in this section are not intended to be implemented in the current iteration of ANTARES, but they do motivate several decisions in the library's design, and so we include them here.

FR-F-1 ANTARES will provide the ability to seek to an arbitrary point in time within a trace, where which the values of its attributes will be consistent with having processed packets from the trace up to but not past that point.

FR-F-2 ANTARES will be able to model arbitrary subsequences on existing packet sequences.

FR-F-3 ANTARES will be able to model hierarchical aggregations of network traffic beyond the 5-tuple model (e.g. aggregates based on source address for investigating peer-to-peer filesharing).

4.2.3 Nonfunctional requirements

We list here the nonfunctional requirements guiding the design of ANTARES, which detail the qualitative goals of the design process. These requirements are broken down into subrequirements where necessary.

NFR-1 (Usability) ANTARES must allow a researcher to implement flow attributes in significantly less time than it would take them to do so using other tools, such as libpcap [Dev06a], Netdude [Kre06], or NetMate [ZS06].

NFR-2 (Extensibility) ANTARES must be designed in a way to allow its functionality to be easily extended.

NFR-3 (Reusability) The individual modules in ANTARES must be designed in a generic enough way that they can be used in other related applications, such as in a graphical analysis application such as Wireshark [Ct06].

NFR-4 (Performance) Performance is not a primary driver of the design of ANTARES, but steps must be taken to ensure that it is not unusably slow. For a set of flow attributes such as that listed in tables 4.1 and 4.2, using commodity hardware (e.g. a commodity PC with a 2GHz CPU and 1GB of RAM) a small sample used for debugging purposes (100kB) should be processed within a few seconds, while a larger data set (2Gb) should be processed within 12 hours (e.g. overnight). The target audience is the research community, so runtime is considered secondary to ease of describing new flow attributes.

NFR-5 (Scalability) ANTARES should be able to cope with handling large data files; e.g. files larger than available memory or greater than the 2Gb limit of signed 32-bit file offsets.

4.3 Architecture

We give here an overview of how ANTARES is arranged, and how the various components interact with each other, in order to provide context for our explanations of how flow attributes are computed, and to discuss many of the design decisions taken during the development. The tool is implemented as a library focused around a data structure, which represents hierarchical aggregates of the network traffic being processed. Flow engines populate the data structure, and flow attributes are attached to the aggregates to compute whatever information is required by the researcher. We describe these various components, and describe in some detail how the flow attributes interact with each other and with the data structure discussion of the capabilities of ANTARES.

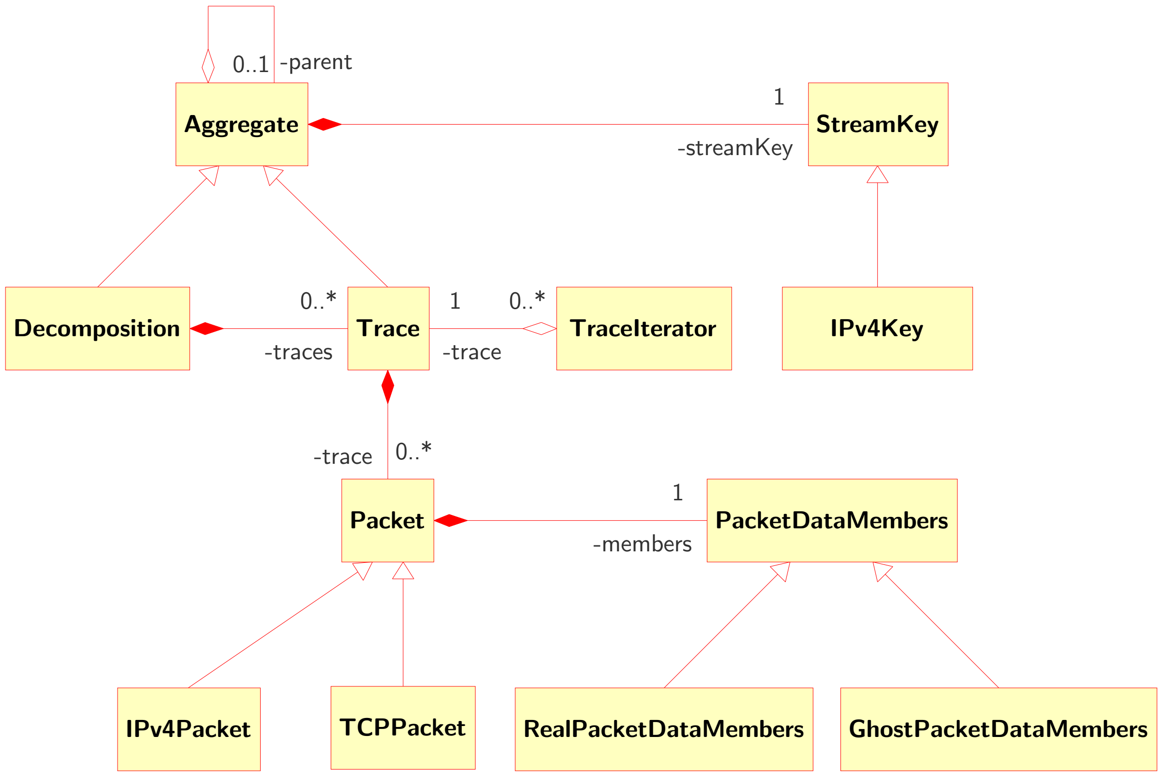

Figure 4.2: ANTARES data structure class diagram

4.3.1 Data structure

The classes that implement ANTARES' data structure are designed to be as generic as possible, to make them more flexible and extensible (**NFR-2**), and so that they could be more easily applied to models of network traffic beyond the 5-tuple flow model used in this work (**FR-F-3**). The data structure consists mainly of packet aggregates and packets. These classes are depicted in figure 4.2; in this section, we describe how the data structure works and explain some of the decisions made in its design in terms of the requirements.

Packet aggregates are handled using the `Aggregate` class and its subclasses, `Decompo-sition` and `Trace`. `Aggregate` represents a generic aggregate, whereas `Decomposition` specifically models an aggregate of other aggregates, and `Trace` models an aggregate of packets. `Decompositions` can thus contain either other `Decompositions` or `Traces`; such a generic treatment is intended to be more reusable (**NFR-3**), and to allow support of new models of network traffic other than those based on the common 5-tuple (**FR-F-3**).

`Aggregates` contain `StreamKeys`, which indicate how subaggregates of a given `Aggregate` are distinguished from one another.[4] For bidirectional 5-tuple flows, the `StreamKey` is an `IPv4Key`, which stores the values of the 5-tuple for that flow. The half-flows belonging to that bidirectional flow would also be keyed using a `IPv4Key`, but for those it would be configured to care about the direction of the flow, whereas the `IPv4Key` for the bidirectional flow would not.

An `Aggregate` can have a `parent`, which is another `Aggregate` that contains it. The simplest example of this is that the half-flows of a bidirectional 5-tuple flow are `Traces` that have as a parent the `Decomposition` representing the bidirectional flow.

The `Packet` class is the base class for the hierarchy that represents packets; packets of particular protocols such as IPv4 and TCP are represented by the appropriate subclasses (e.g. `IPv4Packet` and `TCPPacket` in figure 4.2). New types of packets can be defined by creating new subclasses of `Packet`.

Descendent relationships among packets are tracked by the `Packet` objects (**FR-A2-4**); a given `Packet` object has references to its ancestors (those lower-layer packets that carry data contained in the packet in question) and its descendents (higher-layer packets for which the packet in question carries data). For example, a TCP packet that was carried in its entirety in a single IPv4 packet would have the latter as its only ancestor; however, a TCP packet that was fragmented across several IPv4 packets would have all of those as its

[4]We use the term "stream" as a more generic form of "flow" that could indicate other types of aggregates as necessary.

ancestors.

Properties of packets are accessed via the `getAttrib()` method defined in `Packet`; the name of the desired property is specified as a dot-separated string. `Packet` subclasses return the appropriate values for these properties in the `getAttribLocal()` method; `getAttrib()` implements a searching function that fans out from the packet to check its ancestors and descendents if it is not found in the packet object. For example, the length of the payload portion of the transport-layer packet would be specified as the string *transport.payload .len*. The task of finding the value of the property is thus pushed into the `Packet` class and its subclasses, so that objects using `Packet` objects, such as attributes, need only specify *what* information they need, without needing to know *how* to obtain it. This means that all of the logic required to implement a new type of packet can be contained within a new subclass of `Packet`, which is intended to make the design more easily extensible (**NFR-3**). In addition, the idiom of dot-separated properties is expected to be more familiar to the target audience (and thus more usable, per **NFR-1**), as it mimics the syntax of display filters in Wireshark [Ct06], a popular tool for network traffic analysis.

The ANTARES data structures also implement a mechanism, which we call *packet ghosting*, to create overlays on aggregates (**FR-F-2**) without duplicating entire packets (**NFR-4**); that is, one can create an `Aggregate` that is composed of packets from another `Aggregate`. A ghost packet is an instance of `Packet` where the actual data is contained in an instance of `RealPacketDataMembers` that is contained in a different instance of `Packet` (the original packet that was ghosted). This is accomplished by having a `GhostPacketDataMembers` in the ghost packet that points back to the original; a ghost packet is thus analogous to a reference in C++. In order for this to be transparent to the subclasses and thus make it easier to extend `Packet` (**NFR-2**), `Packet` objects cannot directly access the data corresponding to the packet, and must instead use methods such as `getPacketData()`, which returns all of the packet data, and `getData()`, which returns a specified substring of the data.

`Packet` objects in ANTARES are not intended to be free-floating; instead, it is expected that they will be added to an `Aggregate` as soon as they are created. This is because that any long-lived reference to a packet should be made through a `TraceIterator` object, which acts as a pointer to the `Packet` and represents it in the context of the `Aggregate` containing it. So long as these references are made through `TraceIterator` objects, `Aggregate` objects can write any or all of their packets to a file and be able to read them back in when those packets are accessed, which supports ANTARES' ability to seek within a file (**FR-F-1**) and will allow it to deal with large files by swapping out unused `Packet` objects (**NFR-5**).

4.3.2 Flow attributes

Flow attributes in ANTARES are attached to `Aggregate`s in the data structure as `PacketListener`s; they are notified when a packet is added to the `Aggregate` to which they are attached, and update their state appropriately. Figure 4.3 depicts the classes involved in implementing these flow attributes; this section will explain these classes and how they interact.

Classes that are to be added as listeners to an `Aggregate` must derive from the `PacketListener` interface. `Aggregate`s themselves are `PacketListener`s; this is so that an

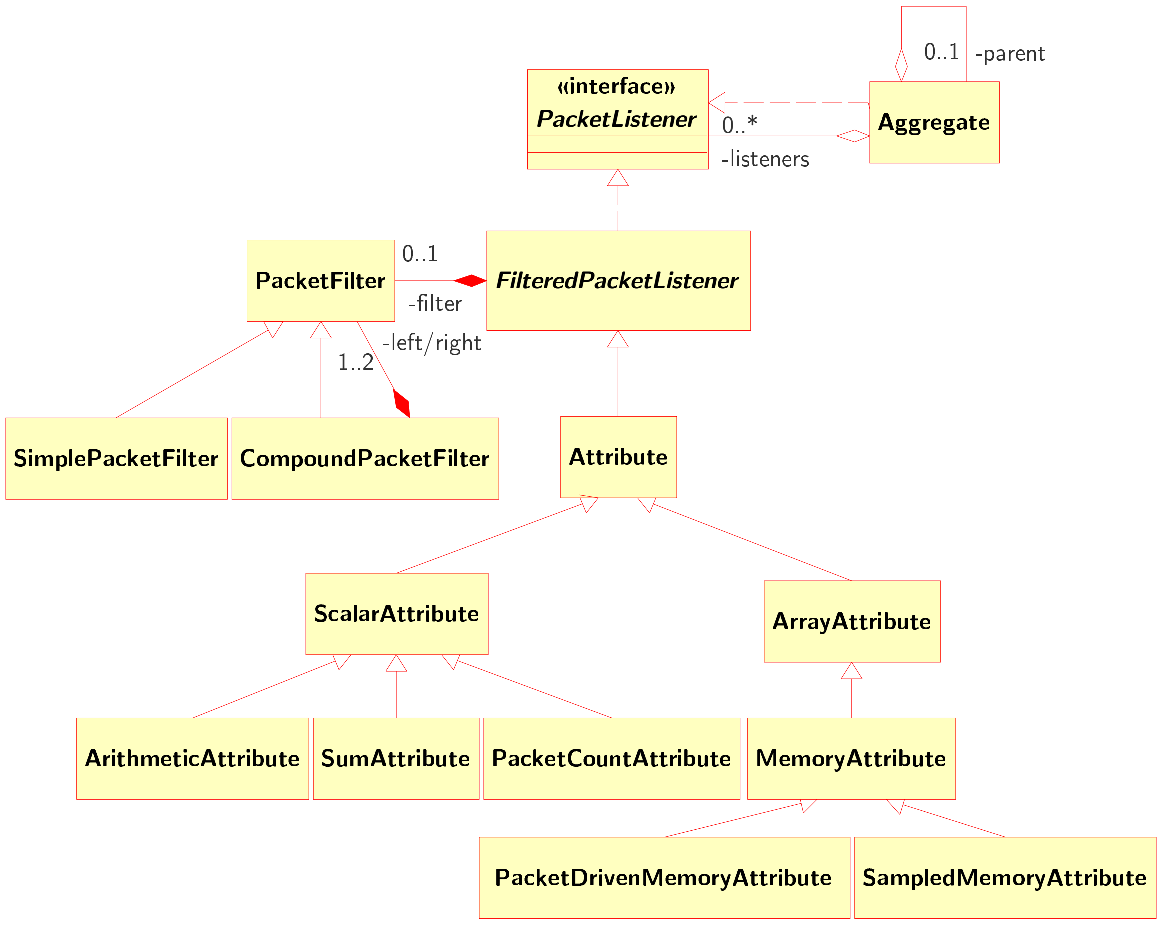

Figure 4.3: ANTARES flow attributes class diagram

`Aggregate`, such as a `Decomposition`, that contains other `Aggregates` can be added as a listener to its child `Aggregates`. Attributes attached to that `Aggregate` are then updated when packets are added to those child `Aggregates` (**FR-A1-2**).

In order to restrict the packets that a particular attribute considers (**FR-A1-6**), ANTARES allows filters to be placed on the listener, using the abstract `FilteredPacketListener` class. These filters, as discussed in section 3.3.1, use a filter expression to determine which packets are passed to the attribute. Filters can be simple or compound (which are implemented by `SimplePacketFilter` and `CompoundPacketFilter`, respectively). Simple packet filters filter based on a single criterion, whereas compound packet filters are logical combinations (e.g. using AND, OR, or NOT) of other packet filters, either simple or compound.

An attribute can refer to the values of other attributes attached to the same aggregate, or to its subaggregates, such as the half-flows belonging to a bidirectional flow (**FR-A1-2**). This is accomplished by using dot-separated strings, as with packet properties. For a bidirectional flow, the strings fwd and *rev* refer to the forward and reverse half-flows, respectively; the *pkt_count* attribute of the forward half-flow can thus be requested from the bidirectional flow as $fwd.pkt\_count$.

Similarly, one can refer to a subattribute of an attribute using a dot-separated string, which is useful for obtaining simple values from a complex attribute. For example, one basic type of attribute is the `MemoryAttribute`, which remembers a certain number of values from packets or from other attributes. The `MemoryAttribute` has a subattribute called *sum*, which is the sum of the values it contains (if they are of an appropriate type, such as an integer). Supposing that a flow contains a `MemoryAttribute` called $payload\_byte\_count\_5s$ that retains the payload lengths ($transport.payload.len$) of all packets in that flow seen in the past 5 seconds, the amount of data transferred as part of that flow in the past 5 seconds could be obtained as $payload\_byte\_count\_5s.sum$.

Attributes are implemented as subclasses of the `Attribute` base class; they are further subdivided into `ScalarAttributes` (attributes which evaluate to a single scalar value), and `ArrayAttributes` (attributes which revolve around an array of values). There are five main attribute classes used by ANTARES: `ArithmeticAttribute`, `SumAttribute`, `PacketCountAttribute`, and `MemoryAttribute`. Most of the flow attributes in table 3.1 were implemented using only these classes, plus a `DurationAttribute` class which computes the duration of a flow or half-flow.[5] These five classes provide enough functionality to define a wide variety of flow attributes.

The `ArithmeticAttribute` class allows other attributes to be combined in arithmetic operations, per **FR-A1-4.3**; it was used frequently in implementing the attributes used in this thesis (those listed in tables 4.1 and 4.2). It currently supports the addition, subtraction, multiplication, and division of other attribute values and constants. One simple attribute implemented as an `ArithmeticAttribute` is the *mean_pkt_len* from section 3.3.3; this is just an `ArithmeticAttribute` that takes the byte count attribute *pkt_byte_count* and divides it by the packet count attribute *pkt_count*.

A `SumAttribute` computes the sum of some property of the packets, per **FR-A1-4.2**; the most obvious application is for attributes such as byte counts. The *pkt_byte_count* and *payload_byte_count* attributes from section 3.3.4 could be implemented as `SumAttribute`s on $network.len$ and $transport.payload.len$, respectively.

A `PacketCountAttribute` is simply an attribute that counts the number of packets that it sees, per **FR-A1-4.1**. The most common such attribute is the *pkt_count*, which is just a `PacketCountAttribute` attached to a flow or half-flow with no filter. Packet proportion heuristics, such as those defined in section 3.3.5, are generally implemented with the help of such attributes. For example, the attribute $pkt\_count(P_{nep})$ used in computing some of the small packet heuristics in section 3.3.5 is computed as a `PacketCountAttribute` with a filter of $transport.payload.len > 0$, as depicted in figure 4.4 in section 4.3.

`MemoryAttribute`s retain some information about the past state of an aggregate, per

[5]The `DurationAttribute` could even have been implemented as an `ArithmeticAttribute` with an associated `MemoryAttribute`, but was implemented separately for simplicity.

FR-A1-4.4. There are two different subclasses of `MemoryAttribute` available for different purposes: `PacketDrivenMemoryAttribute`s and `SampledMemoryAttribute`s. The former records an observation each time a packet is received (either from the packet itself or from some other attribute), whereas the latter records (or samples) an observation at a certain time interval. Both subclasses retain a configurable window of observations, either a fixed number or for a fixed period of time, or for the entire duration of the flow. Obviously, however, the resource requirements of an application using such attributes will be influenced by the memory sizes.

The maximum datarate attributes from section 3.3.4 were implemented using `MemoryAttribute`s. For example, a 5s datarate attribute using a 1s sliding window was implemented with a `PacketDrivenMemoryAttribute` that retained a 5s window of packet lengths, with a `SampledMemoryAttribute` sampling the sum of the lengths in that window every 1s. A `MemoryAttribute` features the subattributes *sum*, which calculates the sum of the observations, and *max*, which calculates the largest observation.

Using these building blocks, a researcher can define a wide range of flow attributes. In cases where these building blocks are not sufficient, they can code their own attributes; those custom attributes could then be used in other computations, retained in memories, etc., just as the attributes described here can. This generality is intended to allow ANTARES to be extended to flow attributes far beyond those that have been designed to date, as our understanding of network traffic matures.

Figure 4.4 shows an example of a flow and its half flows with some attached flow attributes, for illustration. Consider the sp_beta attribute on the lower right side of the figure; recall from section 3.3.5 that the sp_beta attribute is the proportion of small packets

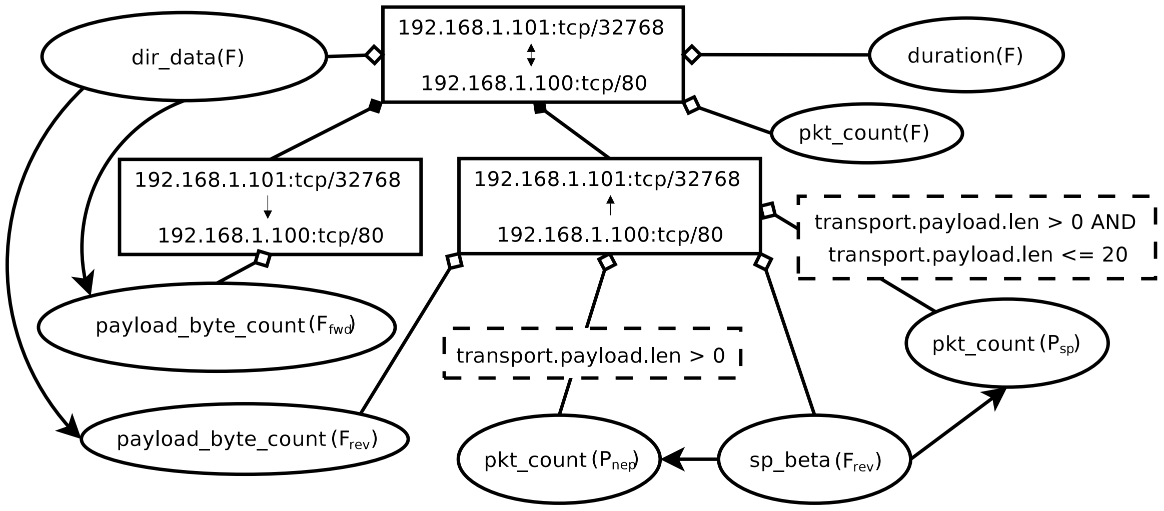
dir_data(F)
192.168.1.101:tcp/32768 ↕ 192.168.1.100:tcp/80
duration(F)
pkt_count(F)
192.168.1.101:tcp/32768 ↓ 192.168.1.100:tcp/80
192.168.1.101:tcp/32768 ↑ 192.168.1.100:tcp/80
transport.payload.len > 0 AND transport.payload.len <= 20
payload_byte_count (F_fwd)
pkt_count (P_sp)
transport.payload.len > 0
payload_byte_count (F_rev)
pkt_count (P_nep)
sp_beta (F_rev)

Figure 4.4: An example of ANTARES attributes

among non-empty packets, for some definition of "small packet". For the reverse half-flow, the *sp_beta* attribute is attached to the half-flow; it uses the values of $pkt\_count(P_{nep})$ (the number of non-empty packets) and $pkt\_count(P_{sp})$ (the number of small packets), each of which is shown with their appropriate filters in dashed boxes. On the left side of the figure, the *dir_data* attribute attached to the bidirectional flow is dependant on the payload byte counts of the half-flows, as it computes its value based on those values. It would use the notations $fwd.payload\_byte\_count$ and $rev.payload\_byte\_count$ to refer to the *payload_byte_count* attribute in each of the forward and reverse half-flows, respectively. In actual use, each flow and half-flow would have the same sets of attributes, but only some are shown in the diagram, for simplicity.

4.3.3 Flow engine

The flow engine being used determines the actual shape taken by the data structure. ANTARES' primary flow engine is session-based; that is, it divides the traffic into half-flows according to the transport-layer protocol (the “next protocol” value in the IP header), source IP address, destination IP address, source port, and destination port; it also pairs those half-flows into bidirectional flows while retaining the distinctions between the two sides of the flow. Figure 4.5 depicts the classes involved in implementing the flow engine; the primary flow engine described above is the `IPv4Sessionizer`.

The session-based flow engine has *template aggregates* that contain the `Attribute` objects that are to be computed for the flows; in the case of `IPv4Sessionizer`, these are the `flowTemplate`, a `Decomposition` containing the `Attribute`s to be computed on the bidirectional flow, and the `halfflowTemplate`, a `Trace` containing the `Attribute`s to be computed on each single-directional half-flow. When the flow engine is to create a new aggregate, it copies the template (using the `clone()` method of the `Aggregate` subclass) along with all of its flow attributes, so that they will computed by the new aggregate. For example, when the `IPv4Sessionizer` creates a new flow, it `clone()`s `flowTemplate` for the bidirectional flow and `clone()`s `halfflowTemplate` for the appropriate single-directional half-flow (and then `clone()`s `halfflowTemplate` a second time when the first packet in the opposite direction is processed). This templating mechanism allows the set of attributes to be defined by the application, independently of the flow engine, while allowing new `Aggregate`s to be created dynamically.

ANTARES uses the libqcap [Hug07] library to access packets. Libqcap is a library based on libpcap [Dev06a]; it reassembles fragmented IP packets, maintaining relationships between the original fragments and the reassembled packets (as per **FR-A2-4**). That library is wrapped with the `QcapAdaptor` class, which uses libqcap to read a network trace, process the data link layer and reassemble fragmented IPv4 packets, and add those pack-

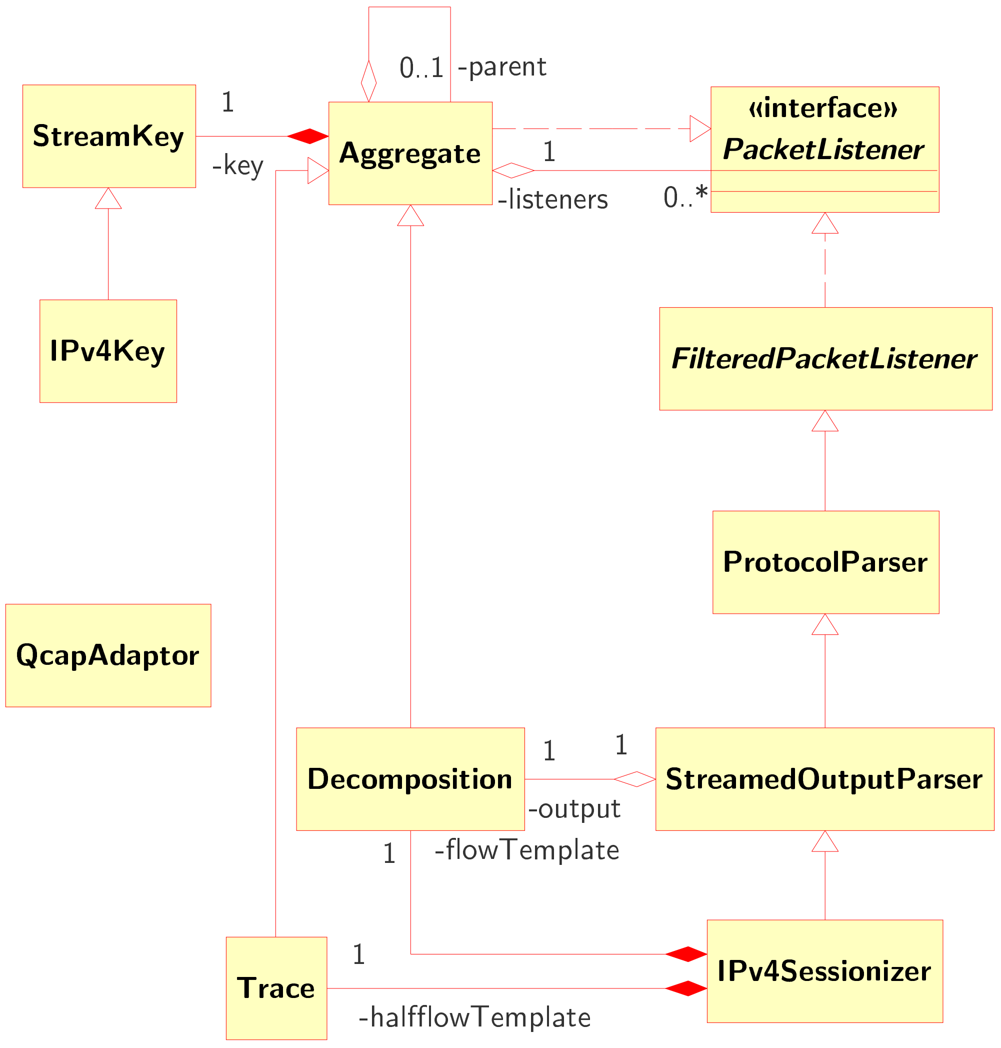

Figure 4.5: ANTARES flow engine class diagram

ets to an IPv4 `Trace`. The `IPv4Sessionizer` listens to that IPv4 `Trace` and outputs to a `Decomposition` (the `output` data member inherited from `StreamedOutputParser`), creating flows as described above, keyed with `IPv4Keys`. An example of a data structure built by the `IPv4Sessionizer` flow engine is shown in Figure 4.6. The object representing the entire network trace is made up of bidirectional flows, each of which is made up of two half-flows.

ANTARES is designed to be flexible in the way that flow engines are used (**FR-F-3**,

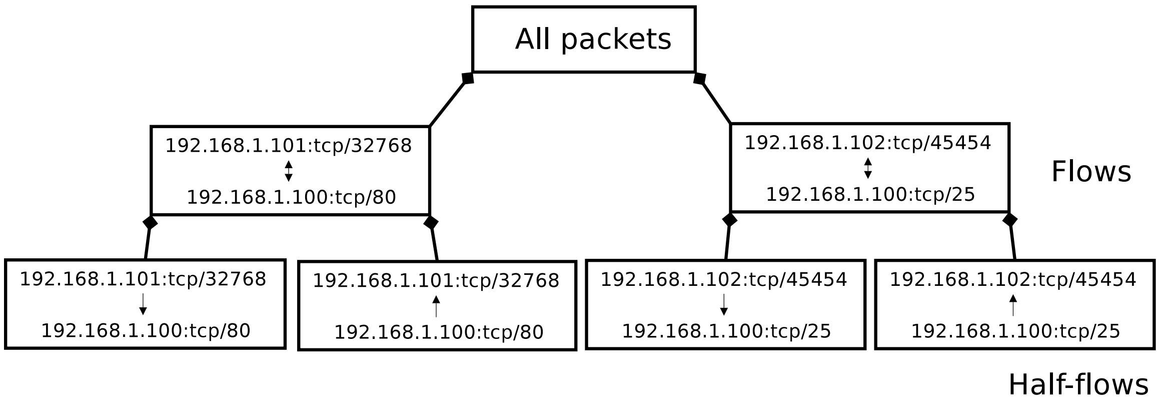
All packets
192.168.1.101:tcp/32768 ↕ 192.168.1.100:tcp/80
192.168.1.102:tcp/45454 ↕ 192.168.1.100:tcp/25
Flows
192.168.1.101:tcp/32768 ↓ 192.168.1.100:tcp/80
192.168.1.101:tcp/32768 ↑ 192.168.1.100:tcp/80
192.168.1.102:tcp/45454 ↓ 192.168.1.100:tcp/25
192.168.1.102:tcp/45454 ↑ 192.168.1.100:tcp/25
Half-flows

Figure 4.6: An example of an ANTARES session-based data structure

NFR-2). The flow engine described here deals with both the network and transport layers, but that is not required by the library. An alternate configuration would have an IPv4 flow engine process incoming IPv4 packets into `Decompositions` by the *next protocol* field, and have separate TCP and UDP flow engines attached as listeners to the $protocol = 6$ and $protocol = 17$ `Decompositions`, respectively (either by creating those `Decompositions` ahead of time, or by having a lookup table for the IPv4 flow engine to create the appropriate transport-layer flow engine, etc.). Similar techniques can be used to implement other types of aggregations, such as by node-pair or by source or destination node.

4.4 Implementing flow attributes

This section describes a program implemented using the ANTARES tool to compute the flow attributes described in Chapter 3. This serves not only to document the capabilities of the program, which is included with the ANTARES tool, but also illustrates how ANTARES is used in practice. We first present an example of implementing an attribute using ANTARES, and then give an overview of the functionality of the `profile_streams_the-`

```
nonempty_count = pkt_count(F{transport.len > 0})
sp_count_1 = pkt_count(F{(transport.len <= 20) AND (transport.len > 0)})
sp_beta_1 = (sp_count_1 / nonempty_count)
```

Figure 4.7: Pseudocode for $sp\_beta(\lambda_{max\_len} = 20; F)$

```
PacketFilter *nonemptyPacketFilter = new SimplePacketFilter(
    PacketAttribute("transport.len"),
    ConcretePacketAttributeValue<int>(0),
    SimplePacketFilter::GREATER );
templ.addAttrib( "nonempty_count",
    PacketCountAttribute( *nonemptyPacketFilter ) );
PacketFilter *sp_filter_1 = new CompoundPacketFilter(
    SimplePacketFilter(
        PacketAttribute( "transport.len" ),
        ConcretePacketAttributeValue<int>(20),
        SimplePacketFilter::LESS_OR_EQUAL ),
    *nonemptyPacketFilter,
    CompoundPacketFilter::AND );
templ.addAttrib( "sp_count_1", PacketCountAttribute( *sp_filter_1 ) );
templ.addAttrib( "sp_beta_1",
    ArithmeticAttribute<int,int,double>(
        "sp_count_1", "nonempty_count",
        ArithmeticAttribute<int,int,double>::OP_DIVIDE ), 1 );
```

Figure 4.8: Actual code for $sp\_beta(\lambda_{max\_len} = 20; F)$

`sis` program, which computes the values of the flow attributes discussed in Chapter 3.

As a concrete example of a flow attribute implemented with ANTARES, we consider $sp\_beta(\lambda_{max\_len} = 20; F)$ (referred to as `sp_beta_1` in the code, as it corresponds to parameter set 1 in our experiment), as shown in figure 4.4 in section 4.3.[6] We present this attribute and those on which it depends, first in pseudocode as it will be written once we have developed a parser for flow attributes in figure 4.7, then as the actual code in figure 4.8.

Figure 4.9 outlines the control flow of the `profile_streams_thesis` program that we

[6]Note that the code shown does not distinguish between a flow and a half-flow; that is done based on where the code is placed. The above actual code was from a function that sets up the template aggregate for half-flows.

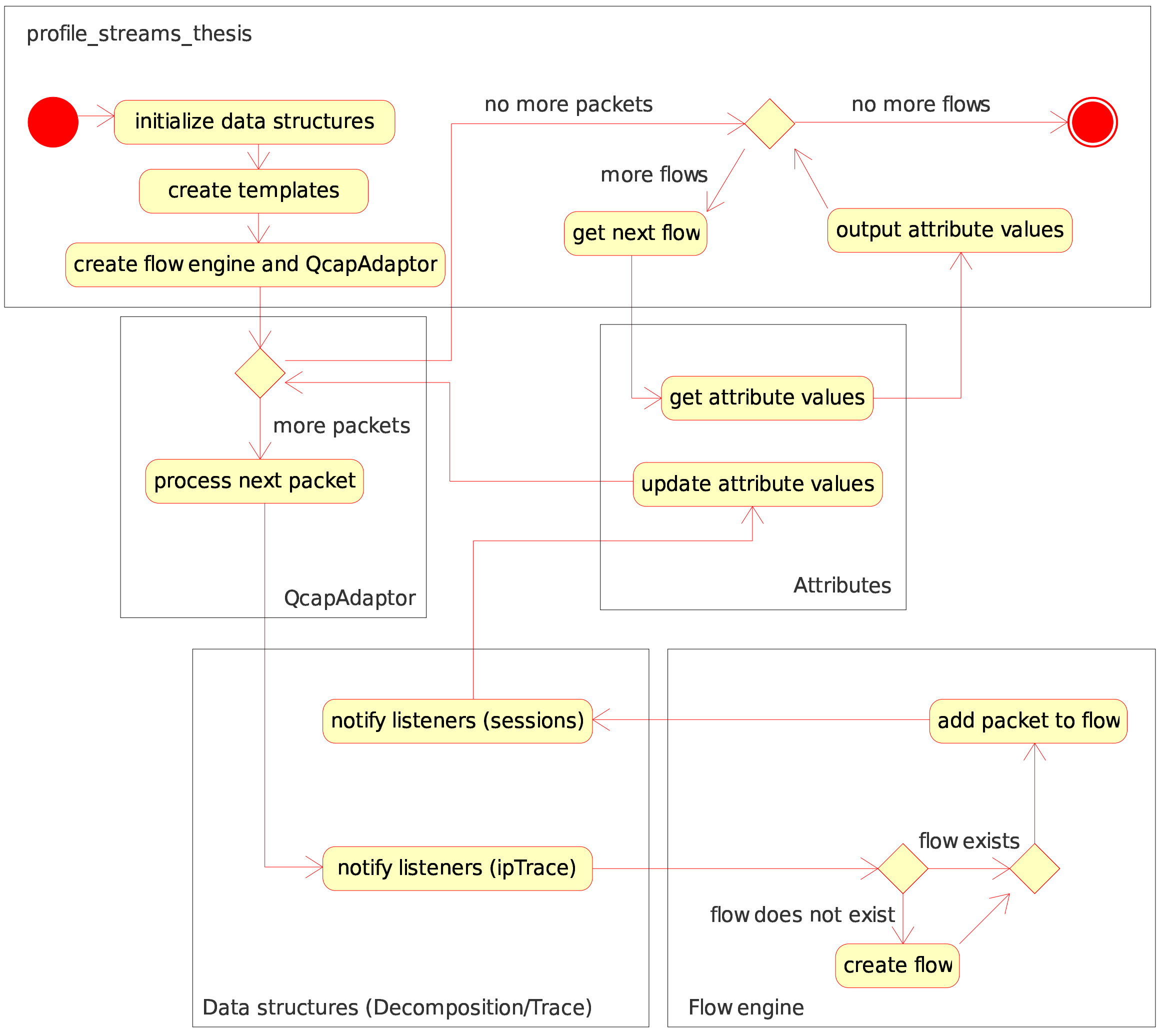

Figure 4.9: UML Activity diagram depicting control flow of `profile_streams_thesis`

used to compute the values used for our evaluation in Chapter 5, and illustrates the interactions between the different components of ANTARES. It first creates and initializes various data structures and processing objects; of note are `ipTrace`, which is the `Trace` into which the `QcapAdaptor` deposits non-fragmented and reassembled IPv4 packets, and `sessions`, which is a `Decomposition` into which the `IPv4Sessionizer` flow engine organizes packets into (session-based) flows. It also creates two templates: `flowTemplate`, containing the attributes to be computed for the bidirectional flows, and `halfflowTemplate`, containing the attributes to be computed for the single-directional flows (the actual attributes are listed below).

Once the initialization is done, the program goes into the main processing loop, where the `QcapAdaptor` reads packets, reassembles them, creates `IPv4Packet` objects, and deposits them into `ipTrace`, which notifies its listener, the `IPv4Sessionizer`; that in turn adds the packet to the appropriate trace in `sessions`, creating it first (from copies of `flowTemplate` and `halfflowTemplate`) if necessary. The `Trace` representing the half-flow to which the packet is added notifies its listeners, which are its attributes and also the `Decomposition` representing the flow containing that half-flow; the latter notifies its own attributes. After all the updating is done, control returns to `QcapAdaptor`, which continues this loop until all of the packets have been read. The main program then regains control and iterates through the flows in `sessions`, fetching and outputting the values of the attributes.

Table 4.1 lists the per-flow attributes computed by `profile_streams_thesis`, and table 4.2 lists the per-half-flow attributes that it computes. For the most part, these are the same as those listed in table 3.1. The notable exceptions are the max datarate attributes, which indicate the maximum per-second datarate over varying time windows. The parameter sets mentioned in the descriptions of the small and large packet heuristics in table 4.2 refer to those defined later, in table 5.1 and 5.2 in section 5.1.2; there are three parameter sets for the small packet heuristics, and six for the large packet heuristics, and they are represented by digits from 1–3 and 1–6 in the names of the attributes computed by `profile_streams_thesis`.

Name	**Description**
pkt_count	number of packets
duration	duration in seconds
nonempty_count	count of nonempty packets
pkt_byte_count	sum of packet lengths
payload_byte_count	sum of payload lengths
mean_delay	mean inter-packet delay
mean_pkt_len	mean packet length
mean_payload_len	mean length of nonempty packets
mean_nonempty_payload_len	mean length of nonempty packet payloads
dir_data	directionality of data
mean_pkt_datarate	mean datarate (packet length)
mean_payload_datarate	mean datarate (payload length)
1s_datarate_window.max	max datarate over 1s window
5s_datarate_window.max	max datarate over 5s window
30s_datarate_window.max	max datarate over 30s window
flag_urg	proportion of pkts w/ TCP URG flag
flag_ack	proportion of pkts w/ TCP ACK flag
flag_psh	proportion of pkts w/ TCP PSH flag
flag_rst	proportion of pkts w/ TCP RST flag
flag_syn	proportion of pkts w/ TCP SYN flag
flag_fin	proportion of pkts w/ TCP FIN flag

Table 4.1: Per-flow attributes computed by `profile_streams_thesis`

Name	**Description**
pkt_count	number of packets
pkt_byte_count	sum of packet lengths
payload_byte_count	sum of payload lengths
nonempty_count	count of nonempty packets
mean_delay	mean inter-packet delay
mean_pkt_len	mean packet length
mean_payload_len	mean length of nonempty packets
mean_nonempty_payload_len	mean length of nonempty packet payloads
1s_datarate_window.max	max datarate over 1s window
5s_datarate_window.max	max datarate over 5s window
30s_datarate_window.max	max datarate over 30s window
sp_alpha_x	small packet heuristic α, parameter set x
sp_beta_x	small packet heuristic α, parameter set x
sp_gamma_x	small packet heuristic α, parameter set x
sp_delta_x	small packet heuristic α, parameter set x
lp_alpha_x	large packet heuristic α, parameter set x
lp_beta_x	large packet heuristic α, parameter set x
lp_gamma_x	large packet heuristic α, parameter set x
flag_urg	proportion of pkts w/ TCP URG flag
flag_ack	proportion of pkts w/ TCP ACK flag
flag_psh	proportion of pkts w/ TCP PSH flag
flag_rst	proportion of pkts w/ TCP RST flag
flag_syn	proportion of pkts w/ TCP SYN flag
flag_fin	proportion of pkts w/ TCP FIN flag

Table 4.2: Per-half-flow attributes computed by `profile_streams_thesis`

Chapter 5

Evaluating flow attributes

Given a set of potential behavioural flow attributes, we wished to evaluate how useful they are at discriminating between different types of network traffic and to demonstrate some uses of the Advanced Network Traffic Analysis Research and Exploration Suite (ANTARES), the tool that we have developed. We describe in this chapter an experiment where we investigate the ability of some flow attributes to distinguish between a small set of common applications; it is by no means an exhaustive survey of flow attributes and networked applications. There were three main results from our experimentation. We found that our data was similar to that of Roughan et al. [RSSD04], but that their results were probably optimistic, due to the fact that they used values that were averaged over a day rather than using individual flows. We looked at parameter values for small and large packet heuristics, and found that several were useful for distinguishing between applications. Finally, we found that the error rates of the classifiers corresponded to our expectations; two applications that we expected to exhibit similar behaviour to one another were more difficult to distinguish from one another than from other applications. We discuss in this chapter in more detail the design of the experiment and then present our results.

We chose several common applications for which there are historical network traces available and that we felt exemplified three different application behaviours, and we used several of the discussed flow attributes to build classifiers for discriminating between pairs of these applications. The error rates of the resulting classifiers gave us a combined measure of the discriminating power of the flow attribute and the similarity of the two applications. This combined measure was clarified, at least qualitatively, by comparing the relative performance of different classifiers on the same task. That is, if all of the flow attributes studied had produced poor classifiers for distinguishing between a particular pair of applications, we had some basis to suspect that it was because that pair of applications is fundamentally similar, although this suspicion would by no means be conclusive.[1] We found that the error rates followed our expectations; applications that we considered to be similar were more difficult to distinguish from each other than from those we expected to be dissimilar.

For the experiment, we constructed a composite data set using various data sets from the National Laboratory of Applied Network Research (NLANR). We chose the samples to cover a variety of different traffic, in an effort to capture the "normal" behaviour of each application, independent of a particular network environment, and to avoid skew from particular hosts performing abnormal activities (e.g. running Telnet sessions over ports associated with FTP). The data that we used is already out-of-date, the most recent samples being more than two years old; we leave it to future work to investigate how the characteristics of network traffic change over time. The composite data set is discussed in detail in

[1]We can obviously only consider similarity in terms of the attributes with which we have used to study the attributes; it will always be possible that there may be another flow attribute, not considered, for which the two applications are totally dissimilar. However, we hold that this is true in general for any expression of similarity: it is only valid in a certain context.

section 5.1.3; it is publically available alongside our toolkit [Fur06].

It may seem odd (even hypocritical) that, having proclaimed that the way of the future is to focus on application behaviours rather than applications, we ourselves use the application as a label for our data. The distinction is that we are not building classifiers for the sake of building classifiers, we are building them as a technique to evaluate flow attributes and to illustrate the concept of application behaviours. The applications we use are exemplars of particular types of behaviour; FTP-data and HTTP are exemplars of bulk data transfer, FTP-control and Telnet are exemplars of command-shell interactive behaviour, and POP3 and SMTP are exemplars of machine-driven interactive behaviour. We accept that the data will be noisy, even potentially with some number of flows that were generated by a completely different application than we suppose, though we have taken steps to minimize the noise in preparing the data.

In this section, we present a qualitative evaluation of ANTARES by using it for several analysis tasks. We evaluate the values in the context of past work, analyze the effects of different parameters for flow attributes, and identify potential application behaviours using common applications as exemplars. In the course of this experimentation, we implemented a range of flow attributes using our tool; we found this task was straightforward, which reflects well on the capabilities of the ANTARES tool. This does not take the place of a rigorous evaluation of the toolkit, however; such an evaluation has been left as future work.

5.1 Experimental design

The goals of this experiment are to demonstrate the ANTARES tool and the flow attributes that we use in several contexts: evaluating past work, investigating parameter values for flow attributes, and exploring application behaviours. We describe here in detail the experiment; we specify how the applications, flow attributes, and samples were selected, and then describe how the results were obtained and interpreted.

5.1.1 Applications

In this experiment, we used traffic from six common applications: FTP-data, FTP-control, Telnet, SMTP, HTTP, and POP3. We have made the uncomfortable, though convenient, assumption that we can use the IANA-registered port number as a proxy for application in the historical data, expecting that the few cases in which that is not valid are outweighed by the majority in which the applications are using the defined port. Accordingly, we constructed our data set in such a way as to minimize the number of such stray sessions, as described in section 5.1.3.

Since one of our goals was to examine how the values of the flow attributes listed in table 3.1 vary between data sets, the main criterion for selecting applications was that they all be present in a large enough number of the available data sets. The six applications chosen were all present in sufficient quantities in eight data sets (described in detail in section 5.1.3), which comprise university Internet uplinks, research backbone networks, and a public ISP peering exchange point. These data sets also include multiple samples from the same sites in different years, so that we can examine the changes in the flow attributes over time in the same environment.

We also wanted to choose applications that we expected to exhibit different application behaviours. HTTP and Telnet were chosen to represent bulk data transfer and command-shell interactive activity, respectively; these were the two behaviours we were most interested in from the perspective of distinguishing between web surfing and interactive back-doors. FTP-data was chosen as the most straightforward example of bulk data transfer to complement HTTP. FTP-control was chosen as an alternate type of interactive behaviour to complement Telnet, though we do not expect it to differ significantly from the latter. SMTP and POP3 were considered to be good examples of machine-driven interactive behaviour, although with SMTP, data is pushed from the client to the server, and with POP3, it is pulled by the client.

Some other applications that would have been nice to include, such as peer-to-peer filesharing, streaming media, and online games, were rejected for this experiment simply because they were not sufficiently prevalent in enough data sets, which was a criterion we chose to attempt to get sample sets that represented a variety of network contexts. Although some such traffic could be found in most of the data sets, different applications were seen in different datasets, adding an additional complication that we felt was best left for future work.

5.1.2 Flow attributes

Table 3.1 lists the flow attributes, defined in Chapter 3, that are used in this experiment. We describe here in more detail some of the particular attributes chosen, and the parameters

Parameter set	λ_{max_len} (bytes)	λ_{min_delay} (ms)	λ_{max_delay} (ms)
1	20	10	2000
2	60	25	3000
3	200	250	30000

Table 5.1: Parameter sets for small packet heuristics

Parameter set	λ_{min_len} (bytes)	λ_{max_delay} (ms)
1	225	50
2	1000	50
3	1460	50
4	225	250
5	1000	250
6	1460	250

Table 5.2: Parameter sets for large packet heuristics

used for those that require them.

We arbitrarily chose to try three different maximum datarate attributes, computed on time granularities of 1s, 5s, and 30s. These values were approximated, as the tool is not yet capable of computing these values over all possible time intervals. The approximation was done by sampling the datarate using a sliding window of size t at fractional intervals of $\frac{t}{5}$;[2] for example, to approximate a maximum datarate computed over a time window of size 5s, we took the maximum of the datarates computed over the time windows $[0,5)$, $[1,6)$, $[2,7)$, and so on.

For the dir_data flow attribute, we need to account for the fact that the value could be undefined if the denominator (that is, the payload byte count of the reverse side) is 0. We take the simple approach of adding one to each the numerator and the denominator prior to computing the value of the attribute; this seemed to be a reasonable approach to avoid disturbing the values excessively while avoiding the problem of undefined values.

The parameter values used for the small and large packet heuristics are given in ta-

[2]The choice to use 5 as the fraction was also arbitrary.

bles 5.1 and 5.2, respectively. For small packet heuristics, we used three parameter sets taken from the works of Zhang and Paxson [ZP00] and DeMontigny-LeBoeuf [DL05]: parameter set 1 was the parameters from the former, parameter set 2 was the keystroke-interactive metrics from the latter, and parameter set 3 was the command-line interactive metrics from the latter.[3] For the large packet heuristics, we simply used all combinations of three minimum packet lengths λ_{min_len} and two maximum delays λ_{max_delay}. The parameters $\lambda_{min\_len} = 225$ and $\lambda_{max\_delay} = 50$ were taken from DeMontigny-LeBoeuf's work; $\lambda_{min\_len} = 1460$ was chosen to only consider those packets which carry the typical maximum amount of data for an application using TCP over the Internet,[4]

For both the small and large packet heuristics, it is possible to encounter undefined values. For the small packet heuristics, these can occur if a flow contains no nonempty packets, no small or empty packets, or no consecutive small packets, where the definition of small depends on the parameters of the heuristic in question. For large packet heuristics, they can occur if there are no nonempty packets or no consecutive large packets. These cases are discussed in more detail in section 3.3.5, and by the arguments given there, the most reasonable way to deal with undefined values for those attributes is to consider them to be equal to 0.

Flag proportion attributes were computed for the URG, ACK, RST, PSH, SYN, and FIN flags; the name of the attribute was created accordingly (e.g. $flag\_psh$ for the proportion of packets with the PSH flag). We could have also extended this to include combinations

[3]Note that DeMontigny-LeBoeuf's attributes were designed to identify SSH as well as Telnet data, so our experiment, which does not include SSH, is not really a fair evaluation of these parameters.

[4]The maximum total packet size for a route over the Internet that contains an Ethernet link is 1500 bytes; an IP header with no extensions is 20 bytes [Pos81a], as is a TCP header with no options [Pos81b], which leaves 1460 bytes for the TCP payload.

of flags, e.g. *flag_psh_ack* for all packets with both the PSH and ACK flags set. However, the single-flag heuristics were considered to be sufficient at this time.

5.1.3 Data sets and samples

In order to evaluate flow attributes, it is first necessary to have network traffic on which to compute the attributes. The data used for this experiment was taken from historical network traces from various sources, obtained from the National Laboratory for Applied Network Research (NLANR) [NLA06].[5] This section describes the data sets selected and how we selected samples from them; a more thorough discussion of our preparation of the sample data is presented in Appendix A.

For this experiment, we used data from eight of the NLANR data sets: ABILENE-II, ABILENE-III, ABILENE-V, Auckland-IV, Auckland-VI, Leipzig-I, Leipzig-II, and NZIX-II, as described below. Table 5.3 gives an overview of the data sets used in chronological order. Note that the volume of data in the table indicates the sum of the sizes of the network traces from which we obtained samples, which is often less than the full dataset.

The data was from a variety of different network environments, though it is predominantly from research-affiliated institutions, due to the fact that such institutions were NLANR's primary partners. The ABILENE and NZIX data sets are from network backbone links. The ABILENE data sets are from the Abilene research backbone administered by the Internet2 consortium; the ones used were captured at various times between August

[5]NLANR was supported by funding from the National Science Foundation (cooperative agreement nos. ANI-0129677 (2002) and ANI-9807479 (1998)), but has (as of July 2006) been discontinued; its data, hardware, and website are being maintained by the Cooperative Association for Internet Data Analysis (CAIDA) at the University of California's San Diego Supercomputer Center.

Data set	timeframe	Source
NZIX-II	Summer 2000	New Zealand ISP peering exchange point
Auckland-IV	Winter 2001	University of Auckland Internet uplink
Auckland-VI	Spring 2001	University of Auckland Internet uplink
ABILENE-II	Autumn 2002	ABILENE research backbone
Leipzig-I	Autumn 2002	University of Leipzig Internet uplink
Leipzig-II	Winter 2003	University of Leipzig Internet uplink
ABILENE-III	Summer 2004	ABILENE research backbone
ABILENE-V	August 2004	ABILENE research backbone

Table 5.3: Summary of NLANR data sets used

2002 and August 2004. The NZIX data set is from the New Zealand Internet eXchange, a peering point for both public ISPs and research institutions at the University of Waikato in New Zealand, in July of 2000. The Auckland and Leipzig data sets are from the Internet uplinks of two universities. The Auckland data sets were captured at the University of Auckland in New Zealand, and the Leipzig data sets were captured at the University of Leipzig in Germany. The Auckland data sets used in this work were captured between February and May of 2001, and the Leipzig data sets are from November 2002 and February of 2003.

They are also from several different points in time, as can be seen from table 5.3. Even the most recent data is already more than two years old as of this experiment; however, as we are interested in studying the flow attributes, we felt it was more reasonable to begin with a wide selection of older traffic than to use a narrow selection of more recent traffic, as we were unable to locate such a rich publically available collection of recent traffic as was available from NLANR. We do expect that it will be necessary to establish that observations made on older data are still valid on current traffic.

Some of the approaches discussed in section 2.4 have also used some of these same data sets. Roughan et al. [RSSD04] used the Auckland-IV data set in their work. Hernández-Campos et al. [HCNSJ05] performed clustering on the ABILENE-I data set, which is from the same facility as the ABILENE traces we used, though a different link (they used data from the Indianapolis to Cleveland link). Zander et al. [ZNA05b, ZNA05a] used three of the same data sets we did: Auckland-VI, NZIX-II and Leipzig-II.

We downloaded in excess of 70 gigabytes of compressed network traffic from the NLANR archive; there was far more data available in the data sets we selected, let alone in other available data sets. We then extracted samples from the available data for analysis, using the methodology described in Appendix A and summarized here. We hope that our documentation of this data, along with the data manipulation tools available with ANTARES, will allow other researchers to easily duplicate this sample selection process, either to obtain an independent sample set or to generate a sample set including traffic from applications other than the ones that we have chosen.

Randomly selected samples from each NLANR data set of interest were downloaded and converted to tcpdump [Dev06a] format, and flows of the selected applications were extracted. One of the data sets used consisted of five-minute samples of traces; for consistency, all of the traces were divided into five-minute timeslices prior to sampling. Packets belonging to the target applications were extracted based on TCP ports.

Having obtained trace files of the selected applications from our data sets, we then selected samples from them, where each sample is a network flow (or possibly a partial network flow), as defined in section 3.2. The samples were selected such that, for a given data set, there could be at most two samples involving the same pair of network nodes; this was to ensure that the sample set was diverse, and not strongly influenced by any particular use of a given application.

We balanced our sample set by data set, time-of-day, and application. For each data set, we selected 100 samples per application, per time-of-day period (work hours or off-work hours).[6] The exception to that was the ABILENE-III and ABILENE-V data sets, each of which contained little or no traffic from one of the time periods; for those, we took 100 samples per application from ABILENE-III traces for off-work hours, and 100 samples per application from ABILENE-V for work hours. Overall, we obtained 1400 samples per application: 100 samples from each of two types of time periods from each of seven data sets (seven, not eight, due to ABILENE-III and ABILENE-V not counting for one or the other time period). These were chosen to encompass different network environments at different times from 2000 through 2004, in order to minimize the influence of misleading uses of ports or unusual uses of the applications.

5.1.4 Training classifiers

Once we had selected a set of applications of interest, selected a set of flow attributes of interest, and acquired an appropriate set of sample traffic, we then evaluated the power of the flow attributes (specified in table 3.1) to discriminate between the applications in the samples. We did this by computing the flow attributes for the sample traffic and training a classifier on each flow attribute for discriminating between each pair of applications. In this section, we will explain in more detail how these classifiers were trained.

Values for the selected flow attributes were computed using the Advanced Network

[6]We considered any traffic occurring between 8 AM and 4 PM (local time of the monitored network link) on a non-holiday weekday to be during work hours, and all other traffic to be during off-work hours.

Traffic Analysis Research and Exploration Suite (ANTARES) [Fur06], a tool developed for this research and designed to facilitate the computation of experimental flow attributes. ANTARES is described in more detail in Chapter 4. ANTARES scans through a network trace and aggregates packets into network flows, which are described in section 3.2. Values for the flow attributes are computed by code attached to the aggregations and stored, then retrieved when the network trace has been fully parsed. After these attributes were computed, we used R [Dev05], a popular open-source statistical package, to train the classifiers described here. Specifically we used the `glm` function for general linear models with the `binomial(logit)` family of link functions, which used the least squares method to train a linear regression model using a logistic link function, described in section 2.1.2 and also explained below.

For each pair of applications, we trained one classifier for every flow attribute. The following discussion describes how we trained a single classifier to distinguish between two applications, A and B, using a flow attribute X; we designated application A to be the target class and the other to be the background class for the purposes of training the classifier. The designation of the target class is arbitrary; it does not affect the outcomes. The classes were represented by a categorical (binary) variable Y, which had a value of 1 for the target class and 0 for the background class. For each application, we had 1400 samples, so for each classifier, we used 2800 samples. Each sample was represented by two values; for sample i, X_i was the value of the flow attribute X for that sample, and Y_i was 1 for the samples from application A and 0 for those from application B.

Before training the classifiers, however, we applied a transformation to some of the flow attribute values for the purposes of improving the effectiveness of the regression. The least squares method that we used does not perform well when some of the data points have large numerical values, as it assigns more influence to those data points than those with smaller values. Thus, for flow attributes that can have values exceeding 1.0, we took the logarithm of the value before fitting the model, similar to Wright et al. [WMM04] and Paxson [Pax94]. Unlike those approaches, we did not wish to simply disregard zero values, so to avoid taking the logarithm of zero (which would give negative infinity), we first found the smallest non-zero value among the sample flow attribute values, and increased all of the sample flow attribute values by half that amount. These new values were used as the value of X_i for those flow attributes that we needed to transform.

The addition of one-half of the smallest non-zero value prior to taking the logarithm is based on a common practice in statistics of adding one to each (count) value prior to performing a log transform [Bar47]; we have simply adapted it to work better with our values, many of which range from zero to one. Adding one to a value that is between zero and one would destroy the proportional relationship among values at the low end of the scale; e.g. the difference between 0.01 and 0.02 is 100%, but that between 1.01 and 1.02 is less than 1%. An alternative would be to use a very small constant ε; however, using a value proportional to the smallest non-zero value means that the log-transformed values of the zero values and the smallest non-zero values are also proportional, which avoids large clumps of distant outliers in the transformed data. For example, if a flow attribute had values clustered at 0, 0.25, 0.5, 0.75, and 1, our technique results in clusters of transformed values of -3, -2, -1, -0.5, and 0, whereas using a constant value of about $2 * 10^{-16}$ (`__DBL_EPSILON__`, the smallest double value for the GNU implementation of C++ [GNU07]) would give transformed values of -52, -2, -1, -0.5, and 0. A more appropriate constant could be found for this case, of course, but the resulting constant might not be appropriate in other cases.

For each such classifier, we used a logistic regression model to find a threshold on X that best divided the samples from the two applications; we thus had to produce logit values to use as the responding variable of the model. As discussed in section 2.1.2, logistic regression is a form of linear regression in which the response variable is the log-odds ratio, or logit, of an observation belonging to the target class. That is, if sample k with an explanatory variable of X_k belongs to the target class with probability π_k, the response variable used for the regression is the logit $\pi'_i = log(\frac{\pi_i}{(1-\pi_i)})$. Due to the implementation of this model in R, each sample was treated separately, so each probability π was either 1 or 0 (i.e. $\pi_i = Y_i$).[7] The logit values were produced by taking the values Y_i from the samples, scaling them to the range 0.25–0.75 (this scaling was built into R's logistic regression logic), and computing the logit from these probabilities as described above.

With the logit values computed, the R software fit a regression line to the data. That is, it used the iterative weighted least squares method[8] to compute parameters λ_1 and λ_0 such that the line $\pi' = \lambda_1 X + \lambda_0$ fit the data as closely as possible.

We then used the regression line to find the value of the flow attribute that corresponded to a probability of 0.5 (equal to a logit value of 0) of a sample belonging to the target class. The probability of 0.5 was chosen because there were an equal number of samples in each class for each model, so 0.5 is the probability of a given sample being in the target class in

[7] A more effective method would have been to bin the samples into intervals over the range of the explanatory variable and use the proportion of samples of the target class in each bin j as π_j, with the midpoint of the bin as the explanatory variable X_j; however, as we are interested in the performance of the classifiers relative to each other rather than in absolute terms, we leave this for future work.

[8] The iterative weighted least squares method, from Dobson [Dob90], is a variant of the least squares method described in section 2.1.2, which minimizes the distance between the regression line and the samples.

the absence of any other information.

The value of the flow attribute corresponding to a probability of 0.5 (with the above log transformation of the attribute value reversed) then becomes the threshold x_t for a classification rule, and the sign of the regression coefficient λ_1 indicates whether the target class generally has values greater than or less than the threshold. If $\lambda_1 < 0$, the classification rule is that a sample with a flow attribute value x_s such that $x_s \leq x_t$ is classified as belonging to the target class; if $\lambda_1 \geq 0$, a sample with $x_s \geq x_t$ is classified as belonging to the target class. In the degenerate case where $\lambda_1 = 0$, the classifier would not be useful.

With the classification algorithm defined, we estimated the error rate of the classifiers based on the model using 10-fold cross-validation, as described in section 2.1.1. The selection of samples was stratified to ensure that the classes stayed balanced; i.e. each partition of the sample data had equal numbers of samples from target class and the non-target class.

This experiment allows us to "try out" some flow attributes that we expect to be interesting, to see how useful they are for discriminating among a small set of common applications. We do not claim that this is exhaustive or conclusive in any way; the goal of the experiment was primarily as a test case for ANTARES and our data, and secondarily to provide a preliminary evaluation of these flow attributes. We also do not expect excellent error performance, as we are only using a single flow attribute at a time; if we wished to train high-accuracy classifier, we could use multiple attributes at a time, but doing it this way allows us to study the attributes themselves. Also, we train our classifiers on pairs of applications, rather than trying to distinguish one application from all the others; this is because we do not wish to assume a priori either that each application is in fact distinct from all the others, or to assign them into classes before classifying them.

5.2 Results

This section describes our analysis of the data generated by the above procedure. Our primary goal was to demonstrate ANTARES and the flow attributes described in table 3.1, and our secondary goals were to get a general sense of how useful the various attributes are for distinguishing between a small set of application and to explore whether it would be feasible to establish application behaviours using the NLANR data sets.

Implementing the flow attributes using ANTARES was a straightforward task, though this is at least partially due to the fact that it was developed with many of these attributes in mind. The graphs displayed in this section were produced using the R statistical package [Dev05] to process data produced by ANTARES.

The data that our procedure created is a three-dimensional matrix of classifier error rates, where two of the axes are applications and the third is flow attributes, so that each cell is the error rate of a classifier trained to distinguish between a particular pair of applications using a particular flow attribute. We have included this data in Appendix B as a set of tables.

We present in this section three results from our experimentation. First, we compared the data produced by ANTARES to previous results by Roughan et al. [RSSD04], and found that our data behaved similarly to theirs, but also that their method of averaging flows probably yielded optimistic results. We then used the error rates that we produced to evaluate potential parameters for large and small packet heuristics, and found that in general, a given heuristic could be used for different tasks, but that the most effective parameter values could be quite different for each task. Specifically, the small packet heuristics were useful for distinguishing command-shell interactive behaviour from other types of traffic with one set of parameters, and for distinguishing command-shell and machine-driven interactive behaviour from bulk data transfer with another set of parameters. Finally, we looked at the error rates and identified that they correspond to our expectations of application behaviours; the two applications that we expect to exhibit bulk data transfer behaviour were far easier to distinguish from other applications than from each other, and likewise for Telnet and FTP-control, those that we expect to exhibit command-shell interactive behaviour, and for SMTP and POP3, those we expected to exhibit machine-driven interactive behaviour.

5.2.1 Comparison with Roughan et al.

Among the flow attributes, we consider first the two found most useful by Roughan et al. [RSSD04], described in section 2.4: duration and mean packet length. We found that our data was qualitatively very different from theirs until we applied a transformation to our data that mimicked their methodology of using daily aggregates; with that transformation, our data much more closely resembled theirs. We conclude that their use of daily aggregates may mean that the results of their classification experiment are not applicable for classifying individual flows, though it may be a useful technique in studying the behaviour of the applications themselves.

We examine the data computed by ANTARES in figure 5.1; for each application we randomly selected 70 flows and, for those, plotted mean packet length against duration.[9] We then compared this data to the corresponding figure from Roughan et al.'s [RSSD04]

[9]We chose to use 70 flows for comparison with Roughan et al.'s [RSSD04] work, which used 70 data points per application.

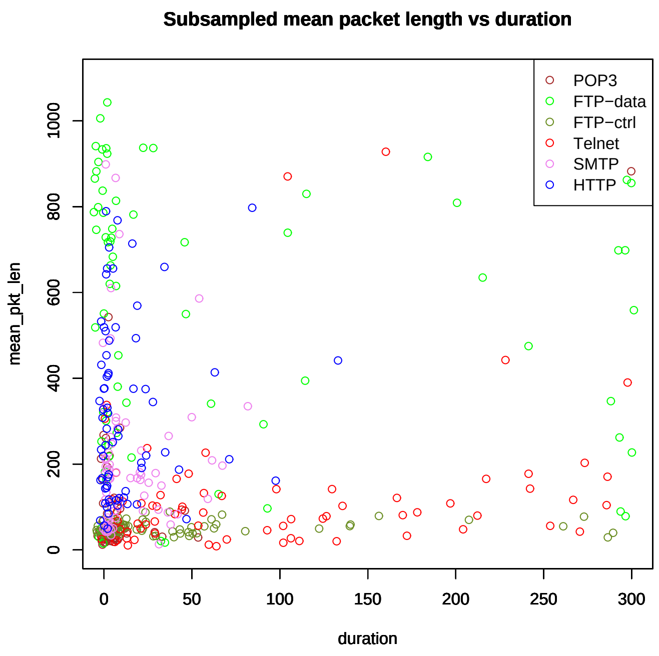

Figure 5.1: Sampled flows from NLANR traces by mean packet length and duration

work, reproduced here as figure 5.2, which shows mean packet length vs. duration for aggregated daily flows for 70 days.[10] The applications used are not the same, but FTP-data, HTTP, and Telnet (ftp-data, www, and telnet in figure 5.2) are common to both. Note that there is significant discrepancy between the two figures.

We explain that discrepancy by plotting points based on averaging our own data, to approximate the aggregate flows used by Roughan et al. [RSSD04]; the results of this transformation are shown in figure 5.3. By averaging the data, we mean that each data point in that figure is computed by selecting 20 sample flows (without replacement) and taking the mean of the durations and the mean of the mean packet lengths of those sample flows as the duration and mean packet length of the averaged data point. We felt that this

[10]Each data point in the figure from Roughan et al. is based on averaging out all of the flows for a given application over an entire day.

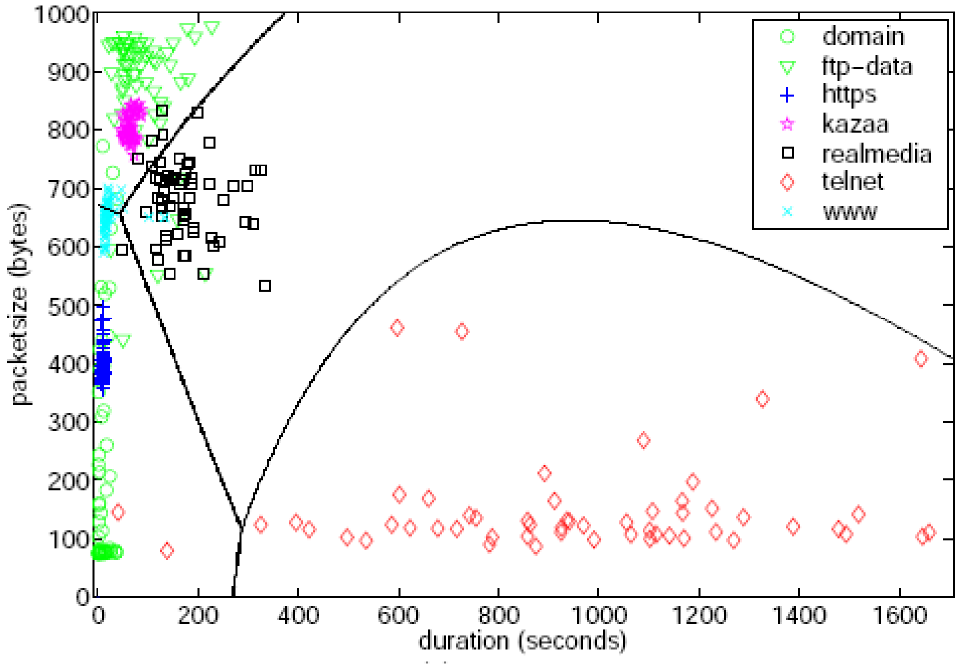

Figure 5.2: Aggregate flows from Roughan et al. [RSSD04] by mean packet length and duration (Figure 2 from [RSSD04])

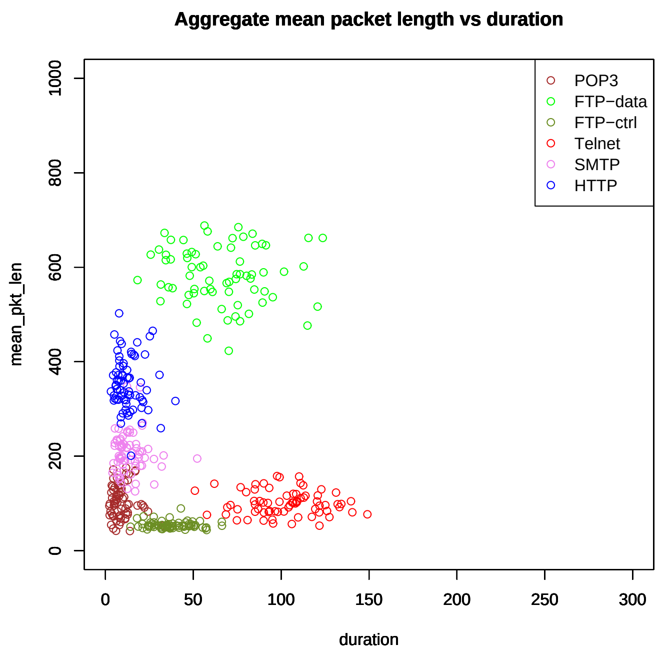

Figure 5.3: Aggregate flows from NLANR traces by mean packet length and duration

was an appropriate way to simulate their data points, each of which was the mean value over all flows seen in a single day. For our figure, the durations, particularly of the telnet sessions, were much shorter, partially because of our 5-minute timeslicing discussed in section 5.1.3. Our mean packet lengths are also somewhat lower, but the general clusters for the three applications we have in common with theirs (FTP-data, Telnet, and HTTP) are in similar places in figure 5.3 as in figure 5.2.

Referring back to the individual flows in figure 5.1, we can see that the attributes of the flows are more variable than those of the averaged data points, suggesting that classifying aggregate flows may be less difficult than classifying individual flows. However, since we are focused on evaluating flow attributes rather than on classifying applications, our quantitative results have no bearing on their results. We simply suggest that the performance of a classifier based on averaged data points such as those used by Roughan et al. [RSSD04] and those in figure 5.3 is not reflective of the performance of that classifier for distinguishing between individual flows. However, it appears that the averaging may be useful in studying and defining application behaviours, as it minimizes the effects of outliers and appears to focus on the "normal" behaviour of traffic for the application; we leave this possibility to be explored as future work.

5.2.2 Parameter selection

We express the small and large packet heuristics, defined in section 3.3.5, in terms of parameters; we describe here an evaluation of the effectiveness of different values of these parameters for distinguishing between the applications that we studied. We did not attempt to optimize them for a particular task, such as distinguishing between bulk data transfer and command-shell interactive behaviour; such an optimization would be done more effectively by using a feature selection algorithm[11] to choose the best parameter set, as we wanted to see if different parameter sets were appropriate for different classification tasks, rather than simply finding the best parameter set for one particular task. We found that the small packet heuristics were able to distinguish between bulk data transfer and other traffic with varying degrees of success with any of the parameter sets we used, and that they were able to distinguish between command-shell interactive behaviour and machine-driven interactive behaviour with certain parameters, while large packet heuristics were best at distinguishing bulk data traffic from other types for the parameters we used.

In the case of small packet heuristics, the parameters are the maximum size of a packet to be considered "small" (λ_{max_len}), and the minimum and maximum inter-packet delay (λ_{min_delay} and λ_{max_delay}, respectively) of packets of interest (to focus only on packets that have timings consistent with human keystroke or command-line entry). For large packet heuristics, the parameters are the minimum length of a "large" packet (λ_{min_length}), and the maximum inter-packet delay between packets (λ_{max_delay}, to focus on packets with timings consistent with a host streaming data quickly over a network).

We performed an evaluation of several parameter sets for small and large packet heuristics, listed in tables 5.1 and 5.2, on classification tasks as listed in table 5.4, and found that certain parameters consistently gave better results than the rest for large packet heuristics, whereas the most effective parameters for the small packet heuristics depended on the

[11]A feature selection algorithm takes a set of candidate features for a labelled data set and determines the subset of those that is most effective for classifying the data, using a given machine learning algorithm.

1	POP3 vs. FTP-data	9	FTP-data vs. HTTP
2	POP3 vs. FTP-control	10	FTP-control vs. Telnet
3	POP3 vs. Telnet	11	FTP-control vs. SMTP
4	POP3 vs. SMTP	12	FTP-control vs. HTTP
5	POP3 vs. HTTP	13	Telnet vs. SMTP
6	FTP-data vs. FTP-control	14	Telnet vs. HTTP
7	FTP-data vs. Telnet	15	SMTP vs. HTTP
8	FTP-data vs. SMTP		

Table 5.4: Classification tasks

context. We used a fairly small set of parameters for each type of heuristic, and our application set was also limited,[12] so we do not consider these results definitive by any means. However, they do provide some insight into the ways in which the applications differ.

For large packet heuristics, parameter sets 1 ($\lambda_{min\_len} = 225$, $\lambda_{max\_delay} = 50$) and 4 ($\lambda_{min\_len} = 225$, $\lambda_{max\_delay} = 250$) proved to be the most useful; in general, the minimum packet size was far more important than the maximum delay. Also, we noted that the large packet heuristics seemed to be more useful when used on the reverse direction half-flows (from server to client). The exception to this was HTTP, for which lp_beta on the forward direction half-flows was fairly useful (perhaps because HTTP requests are significantly larger than client-side requests for the other applications we considered). Figure 5.4 shows the error rates of all six parameter sets, as listed in table 5.2 in section 5.1.2, used for various classification tasks as listed in table 5.4; there are two lines per parameter set, the lower line being the minimum error rate obtained by large packet heuristics with those parameters, and the higher line being the mean error rate of those heuristics.[13] The lines for parameter sets with the same value for λ_{min_len} (1 and 4, 2 and 5, 3 and 6) generally

[12]In particular, a more thorough examination should also include at least streaming media and peer-to-peer filesharing, as we expect those to have packet lengths between those of the interactive and mail applications and less than the bulk data transfer applications that we consider here.

[13]All heuristics for a given parameter set are considered in these measurements, i.e. lp_alpha, lp_beta, and lp_gamma on each of the forward and reverse half-flows.

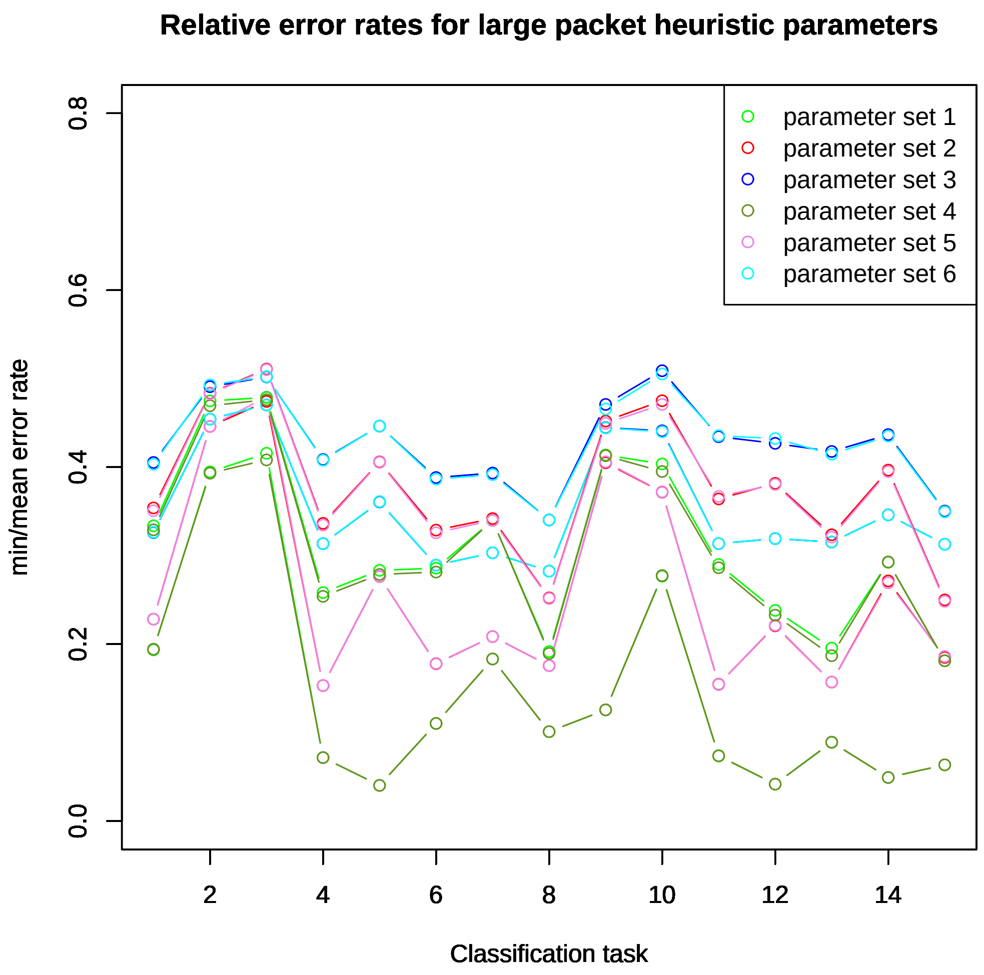

Figure 5.4: Mean and minimum error rates for large packet heuristics by parameter set

overlay each other, illustrating that for each distinct λ_{min_len} parameter, both choices for λ_{max_delay} perform equally well on the applications we examined.

The small packet heuristics were more varied, in that none of the parameter sets were better than the others across all of the classification problems; the performance of the parameter sets depended on the classification task being attempted. Figure 5.5 shows the performance of the various heuristics using each of the three parameter sets listed in table 5.1 in section 5.1.2 on various classification tasks as listed in table 5.4; again the mean

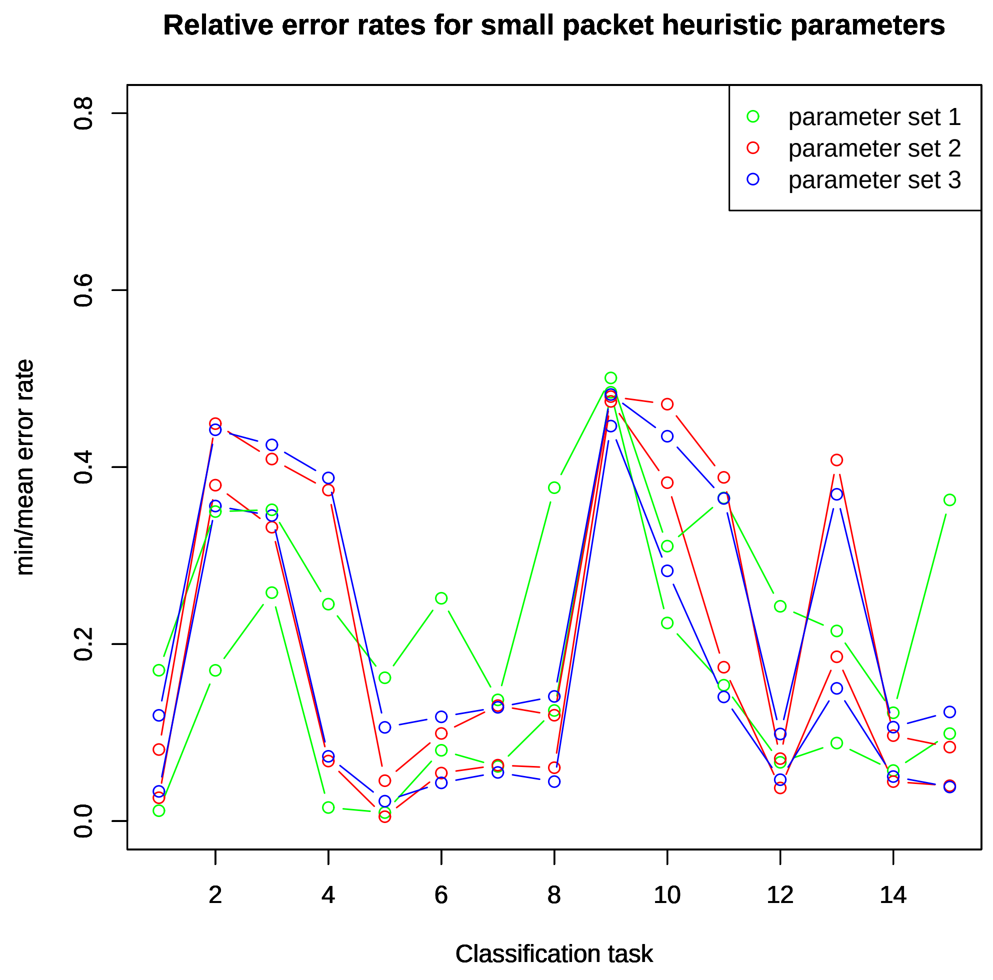

Figure 5.5: Mean and minimum error rates for small packet heuristics by parameter set

and minimum error rates for each parameter set across all of the small packet heuristics are given. Parameter set 1 ($\lambda_{max\_len} = 20, \lambda_{min\_delay} = 10ms, \lambda_{max\_delay} = 2s$) did better on classification tasks 2, 3, 4, 10, and 13 (POP3 vs. FTP-control, POP3 vs. Telnet, POP3 vs. SMTP, FTP-control vs. Telnet, and Telnet vs. SMTP) – distinguishing between applications that generally used many small packets, whereas the other parameter sets (parameter set 1 was $\lambda_{max\_len} = 60, \lambda_{min\_delay} = 25ms, \lambda_{max\_delay} = 3s$, and parameter set 2 was $\lambda_{max\_len} = 200, \lambda_{min\_delay} = 250ms, \lambda_{max\_delay} = 30s$) performed similarly to each other, and generally better than parameter set 1 on distinguishing between applications with many small packets and applications with larger packets. Of the two, parameter set 3 generally did slightly better. Unsurprisingly, none of the small packet heuristics performed particularly well at classification task 9, distinguishing between FTP-data and HTTP.

As we mentioned, we do not consider this an exhaustive evaluation by any means; our results are only valid for the small set of applications we consider, and a proper evaluation should use many more candidate parameter values. However, with the proper data set and values, implementing all of the flow attributes for the candidate parameters with ANTARES would be straightforward.

5.2.3 Distinguishing behaviours

We also examined the results of the classifiers, which are presented in Appendix B; we will summarize and illustrate here some of the more interesting observations. We present the error rates in tabular form in the appendix, with colours to bring out patterns in them; we found this to be a useful technique for identifying general trends, and we explain in the appendix how to interpret the data. For many pairs of applications, there were at least a few flow attributes that could distinguish them with a combined error rate of 0.10 or better. In general, we found that the error rates were consistent with our expectations; the most difficult pairs of applications to distinguish between were those that we considered to represent the same application behaviour, particularly those representing bulk data transfer (HTTP and FTP-data) and command-shell interactive behaviour (Telnet and FTP-control). SMTP and POP3, representing machine-driven interactive behaviour, were not difficult to distinguish from one another, but that was primarily due to the fact that they operate in different directions, SMTP pushing the data from client to server and POP3 pulling it from server to client.

Telnet and FTP-control, the applications representing command-shell interactive behaviour, are difficult to distinguish. The three flow attributes which gave the best combined error rates for distinguishing between the two were *mean_nonempty_ payload_len_fwd*, *mean_payload_len_fwd*, and *flag_syn*, which had error rates of 0.140, 0.163, and 0.214, respectively. The first two attributes are quite similar, so we disregard the second here. It is somewhat counterintuitive that the proportion of SYN flags should be useful, but since there are usually a small number of SYN flags,[14] it may be used as a proxy for the length of the session.[15] Figure 5.6 and figure 5.7 show these attributes as box-and-whisker diagrams for each application.[16] As can be seen from the figures, there is a significant overlap between the two.[17]

The best flow attributes to distinguish between FTP-data and HTTP took advantage of the directionality of the data. For FTP-data, the data is almost exclusively in one direction, and with HTTP, requests are sent in one direction, and responses in the other. The best flow attribute was the *mean_nonempty_payload_len_fwd*, as with Telnet and FTP-control, shown in figure 5.6, with a combined error rate of 0.084. This is probably because there is

[14]There are normally two SYN packets for a TCP connection, one from each node, not accounting for resent packets or SYN packets that were not within the 5-minute timeslice

[15]Though this, of course, does not explain why *pkt_count* gave a worse error rate of 0.266.

[16]A box-and-whisker diagram, or boxplot, plots data by drawing a line at the median value, a box around the middle half of the data (from the first quartile to the third quartile), and "whiskers" extending to the extreme points; outliers not included in the whiskers are indicated with circles. Such diagrams have been found to be useful for summarizing multiple features of a data sample [Dev95]. Figures 5.6 and 5.7 are examples of such plots.

[17]There is also significant overlap among other pairs of applications in these figures, but there were other flow attributes that could distinguish between those. The significant point here is that these two flow attributes were the most effective at distinguishing between Telnet and FTP-control.

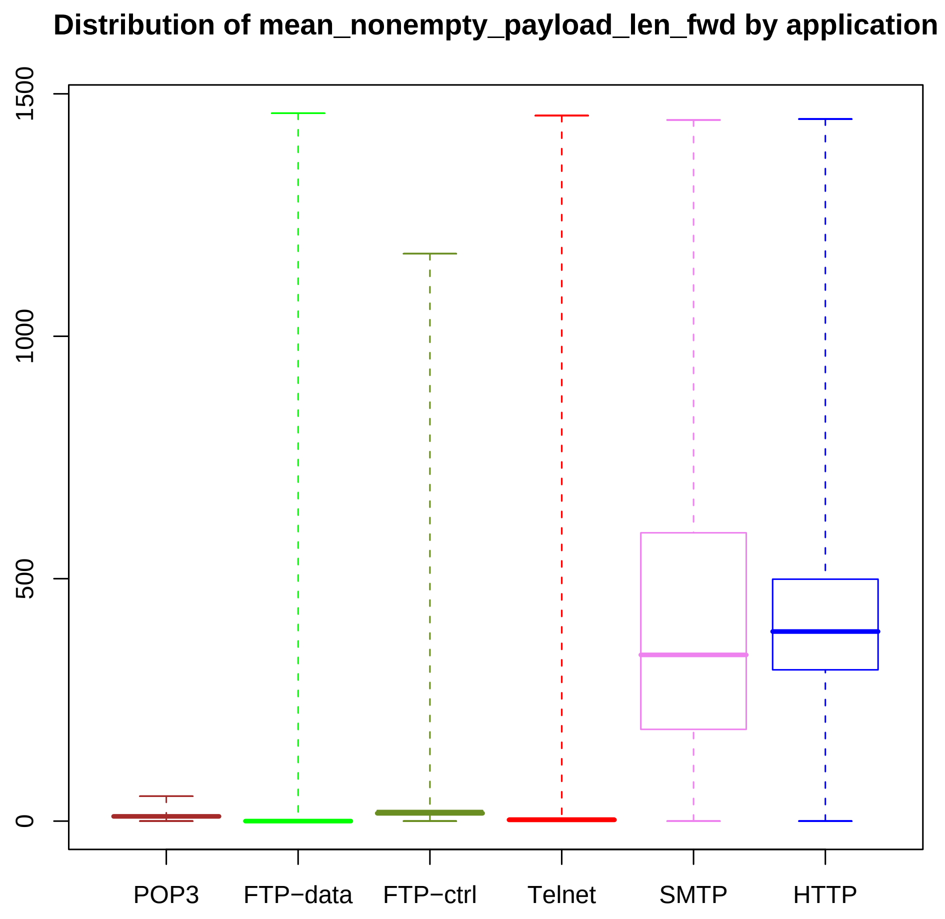

Figure 5.6: Distribution by application of *mean_nonempty_payload_len_fwd*

very little data in the forward direction of most FTP-data flows; if the direction were not known in advance, this one attribute would be less powerful. Having said that, using values from both directions together should still be useful even when the client-to-server direction is not known.[18]

SMTP and POP3 were similarly easy to distinguish from one another by virtue of the

[18]Preliminary experiments suggest that in such a case, the two classes are not linearly separable (i.e. cannot be separated with a single line), with one class "bracketed" by the other – a classification algorithm like *k*-nearest-neighbour, which does not require that the data be linearly separable, might be more effective.

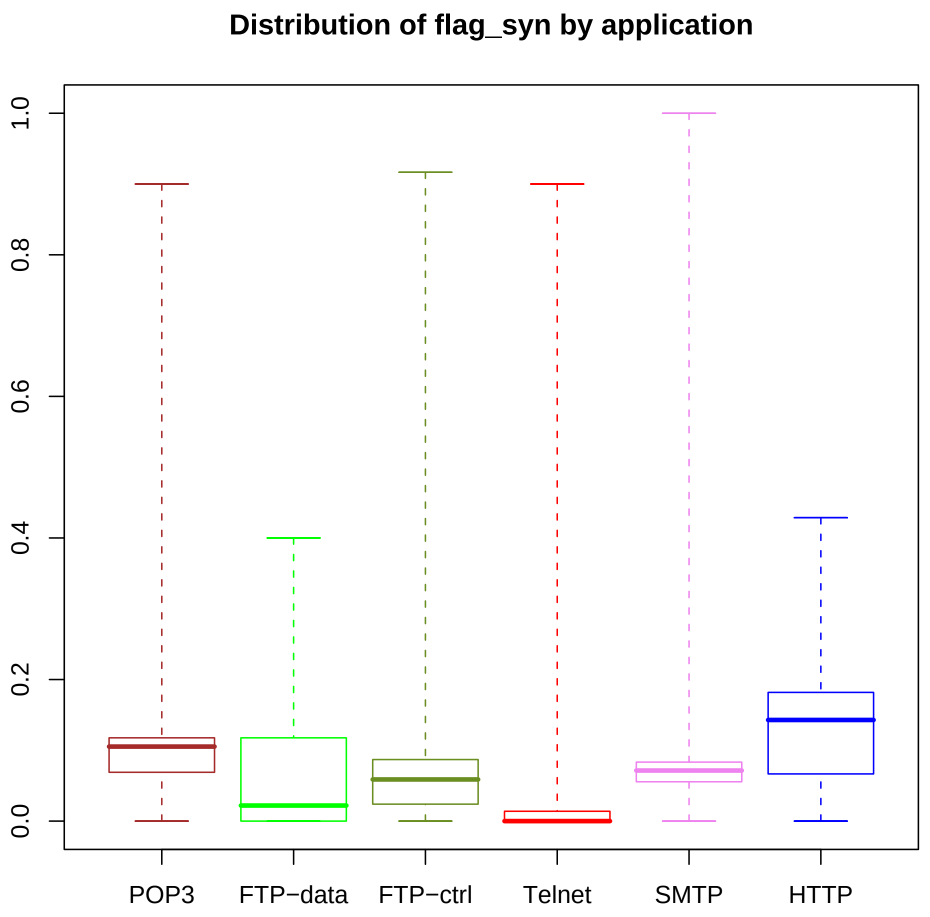

Figure 5.7: Distribution by application of $flag\_syn$

direction of the data. With SMTP, the bulk of the data is sent from the client to the server, whereas with POP3, most of the data is sent from the server to the client. This is clearly visible in figure 5.6. We expect that it would be more difficult to distinguish between the two applications if they were re-oriented so that the data flow was in the same direction.

Chapter 6

Conclusions and future work

In this thesis, we document ANTARES, a tool for computing flow attributes and tools for converting and preprocessing network data sets, and we perform some basic analysis using our tool, to demonstrate the range of flow attributes that it can compute and to illustrate the concepts of flow attributes and application behaviours. We describe here the conclusions that we have drawn based on this investigation, and describe the numerous avenues that we see for future work on traffic classification.

6.1 Conclusions

We used our tool to implement a range of flow attributes from the literature, and found that, although the interface is not particularly user-friendly, a knowledgeable user can easily build flow attributes using its powerful mechanisms. The performance leaves much to be desired, but we expect that it can be significantly improved with a moderate amount of effort. We also found that specifying the flow attributes using the notation presented in this thesis and translating those notations to the syntax of the tool were both straightforward, and we expect that one could implement a parser to do that, making ANTARES easier to use than it is now.

We compared our measurements to those of Roughan et al. [RSSD04], and found that ours were similar to theirs, but only when we averaged our data points to simulate their use of values based on daily averages. We noted that such a treatment of the data would seem to make it easier to distinguish between applications, and thus we questioned whether their classification results were overly optimistic because of this treatment. We note, however, that such a treatment may be appropriate for studying general patterns of application behaviour, and should be considered as a potential tool in itself.

We evaluated a small set of parameters for small and large packet heuristics, and found that while two similar parameter sets were consistently the most useful for the large packet heuristics, the most useful parameter set for the small packet heuristics depended much more on the applications to be discriminated. We also noted that for large packet heuristics, the timing-based criteria we used were not nearly as significant as the packet-length-based criteria.

We found that it was far easier to distinguish between intuitively dissimilar applications, such as between HTTP and Telnet, than to distinguish between intuitively similar applications, such as between HTTP and FTP-data or between Telnet and FTP-control. This is, of course, in the context of the flow attributes that we used, and it is entirely possible that there are other, far more effective, flow attributes, for these purposes. We feel there is far more work to be done in this area, both in searching for more effective flow attributes and for exploring application behaviours, which is why we have focused on building tools in support of these efforts.

We believe that in order to accurately classify traffic from network applications, those applications will have to be expressed in terms of application behaviours that can be identified from network traffic using flow attributes. It is our hope that the notation, the tool, and the data pre-processing tools we have made available will facilitate an exploration of these attributes and behaviours that will lead to a far greater understanding of them and to a far more effective approach to application classification.

6.2 Future work

We see this thesis as supporting the development of behaviour-based network application classification, but there is still a great deal of work to be done, even just to show that such an approach is feasible. Here, we will first discuss the work left to be done in the short term to improve the tools we have developed, and then we will describe the way that we believe behaviour-based classification can be realized.

For ANTARES, there are still several important improvements to be made, and rigorous testing should be done. The interface needs to be finished by creating a parser that can take our notation for flow attributes and translate it into code to drive the computation of those attributes in the tool. Its performance must also be addressed, particularly with respect to those attributes that use a clock, as the introduction of that mechanism caused a significant slowdown. Also, it requires more attribute classes to support the development of other promising attributes from the literature, such as DeMontigny-LeBoeuf's [DL05] conversationality heuristics.

The notation described in this thesis needs to be developed into a programming language by creating a grammar that is appropriate to a text configuration file (as opposed to the grammar of the notation, which is more appropriate to $\LaTeX$mathematical typesetting) and constructing a domain model to provide its semantics; the domain model will depend on the flow engine being used. Using the five-tuple flow engine described in this thesis as a partial example, the top-level concept could be for the *flows*, which would be an aggregate of the different flows encountered in parsing a trace file. Each flow in that aggregate would consist of a *key* containing the fields of the five-tuple (protocol, source IPv4 address, destination IPv4 address), *attributes* corresponding to the flow attributes, both built-in and user-defined, and *subflows* representing the two single-directional flows making up the bidirectional flow. The grammar and domain model would then be used to create a parser that could take a text configuration file containing attribute definitions and use that to create the templates that define the attributes to be computed.

More rigorous testing of ANTARES should be done to ensure that the calculations are correct; in addition to comprehensive unit testing, system testing should be performed by generating network traffic with known characteristics that can be measured by ANTARES. Existing network traffic generators can probably be leveraged in this process, but these are likely to require modifications. Many of the flow attributes of network traffic that ANTARES considers appear to be outside of the scope of traditional network traffic analysis and network traffic generation, so we expect that proper testing will require either significant modifications to existing traffic generation tools, or the development of new traffic generation tools.

For behaviour-based network application classification as a whole, there are three main components that will be needed: meaningful flow attributes, models of application behaviours in terms of the flow attributes, and models of applications in terms of application behaviours. Meaningful flow attributes and application behaviours are those which reflect the purpose for which the network is being used. A particular application behaviour should correspond to a distinct type of activity; examples might include bulk data transfer, interactive command shell activity, streaming media, gaming (i.e. synchronization of virtual environments), polling of a remote resource (e.g. of an email account), or automated transactions (e.g. sending an e-mail via SMTP). A model of an application in terms of such behaviours could take the form of a finite state machine, or a more complicated construct.

The most difficult part is likely to be finding the best flow attributes and application behaviours to use; we expect this to be an iterative process of finding flow attributes that represent a proposed set of application behaviours, then inspecting the samples that do not fit that set of behaviours, and revising the set of behaviours appropriately. That process would seem to require the use of data sets that include the payload data, to better understand those outliers; however, we suggest that a broader set of data without payload would play a crucial role in evaluating the generality of these attributes and behaviours, as we expect it will continue to be difficult for a researcher to obtain data sets with payload from network environments other than their own.

We also believe that the process of developing flow attributes and application behaviours should use fragments of flows rather than full flows, as we expect full flows from many applications to contain multiple behaviours. One interesting topic of research would be techniques for fragmenting flows; for instance, flows could be fragmented adaptively by finding change points in various flow attributes.

Application behaviours spawned by this process can be used as the basis of models of applications themselves. A simple approach would be to take flows belonging to a certain application, break them down into fragments, label those fragments using a model of application behaviours, and use them to infer a model of the application (e.g. "Telnet consists of proportion x of bulk data transfer, proportion y of interactive command shell behaviour, and proportion z of inactivity, where $0 \leq x \leq 0.2$, $0 \leq y \leq 0.95$, $0 \leq z \leq 0.15$, and $x+y+z=1$"). A more complex approach would be to define each behaviour as a state in a finite state machine, and to learn the transitions between the states. These models that could then be used to classify network traffic by application, and the results of such classifications could be evaluated.[1]

An alternate approach that has been suggested would be to create generative models, using queueing theory [Kle75, GH85] for example, to simulate traffic from particular applications, then compare the traffic generated by these models with the traffic observed on actual networks. This would seem to bypass application behaviours, but what we would expect to happen is that for a given application, several distinct models (corresponding to distinct application behaviours) would be required to generate all of the traffic for that application, and that those models could be reused between applications that exhibit similar behaviours. We agree that this is certainly a valid approach, and could probably be used in conjunction with the observational approaches described above.

Further work is required to build a better understanding of the domain of possible flow attributes. In Chapter 3, we presented flow attributes divided into classes, but this was

[1]Ideally these would be evaluated using techniques such as 10-fold cross-validation for more accurate results, and the terms used to report the results, such as accuracy or error rate, would be clearly defined.

not a rigorous treatment, it was merely for organizational and conceptual purposes. We suggest that a more comprehensive treatment with concrete criteria would be useful for understanding the domain. In addition to the type of information on which an attribute is based (e.g. time, packet lengths, data volume), another criterion that could be used is the level of the information (packet-level, flow-level, connection-level, intra-flow, multi-flow) used by Roughan et al. [RSSD04], or the way in which it is computed (e.g. mean, sum, minimum/maximum, heuristic).

Evasion is a major issue that will also need to be addressed explicitly. Our intuition is that a system using a variety of complementary flow attributes will be more difficult to evade than one that uses only a few flow attributes, but there is much room for work in determining how to select sets of attributes that complement one another. The example given earlier was that of packet length and packet count; if an attacker having a certain set amount of data to send tries to evade detection by sending smaller packets, they will have to send more packets. Some mechanism for quantifying these relationships will be needed, in order to use “resistance to evasion” as a criterion in an algorithm that searches for optimal feature sets of flow attributes.

There is also a great deal of work that could and should be done in visualization using these flow attributes, which will be vital in support of this work. As we discussed in Chapter 4, the ANTARES tool was designed with the possibility of being used for interactive traffic analysis, along the lines of Wireshark [Ct06], but focused on flow attributes rather than on parsing payload. We present classifier error rates in tables in Appendix B, colourized for readability, but we feel there is a great deal of room for improvement in displaying such information in a more easily usable format. Hernández-Campos et al. [HCNSJ05], for example, developed techniques for visualizing the results of their clustering approaches; tools based on these and other techniques will likely be very useful in this field.

Chapter 7

References

[Alv04] H. Alvestrand. A Mission Statement for the IETF. RFC 3935 (Best Current Practice), October 2004.

[Aut06] Internet Assigned Numbers Authority. Port numbers. Web resource, November 2006. http://www.iana.org/assignments/port-numbers, last accessed Nov/2006.

[Bar47] MS Bartlett. The Use of Transformations. *Biometrics*, 3(1):39–52, 1947.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[BP04] Kevin Borders and Atul Prakash. WebTap: Detecting covert web traffic. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, pages 110–120. ACM Press New York, NY, USA, October 2004.

[cBP95] k. claffy, H. W. Braun, and G. Polyzos. A parameterizable methodology for internet traffic flow profiling. *IEEE Journal on Selected Areas in Communications*, 13(8):1481–1494, October 1995.

[CR06] M. Collins and M. Reiter. Finding peer-to-peer file-sharing using coarse network behaviours. In *Proceedings of the 11th European Symposium on Research in Computer Security*, volume 4189/2006, pages 1–17. Springer Berlin/Heidelberg, September 2006.

[Ct06] G. Coombs and the Wireshark Development Team. Wireshark. Software package, available online, September 2006. http://www.wireshark.org, last accessed Sept/2006.

[Dev95] J. L. Devore. *Probability and Statistics for Engineering and the Sciences*. Duxbury Press, 4th edition, 1995.

[Dev05] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. ISBN 3-900051-07-0.

[Dev06a] TCPdump Development Team. TCPdump. Software package, available online, September 2006. http://www.tcpdump.org, last accessed Sept/2006.

[Dev06b] Wireshark Development Team and User Community. Wireshark wiki: Ethernet (IEEE 802.3). Web resource, November 2006. http://wiki.wireshark.org/Ethernet, last accessed Nov/2006.

[DL05] A. DeMontigny-LeBoeuf. Flow attributes for use in traffic characterization. Technical report CRC-TN-2005-003, Communications Research Center, Industry Canada, December 2005.

[DL06] A. DeMontigny-LeBoeuf. Personal communication, September 2006.

[DO01] T. Dunigan and G. Ostrouchov. Flow characterization for intrusion detection. Technical Report TM-2001/115, Oak Ridge National Laboratory, November 2001.

[Dob90] A.J. Dobson. *An Introduction to Generalised Linear Models*. Chapman & Hall, 1990.

[EBR03] J. P. Early, C. E. Brodley, and C. Rosenberg. Behavioral authentication of server flows. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, pages 46–55, December 2003.

[EKMV04] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 245–256. ACM Press New York, NY, USA, 2004.

[EV03] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3):270–313, August 2003.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[fIDAC06] Cooperative Association for Internet Data Analysis (CAIDA). CoralReef. Software package, available online, November 2006. http://www.caida.org/tools/measurement/coralreef/, last accessed Nov/2006.

[Fra94] J. Frank. Artificial intelligence and intrusion detection: Current and future directions. In *Proceedings of the 17th National Computer Security Conference*, October 1994.

[Fur06] T. Furlong. ANTARES project page. Software package, available online, December 2006. http://antares-net.sourceforge.net/.

[GH85] D. Gross and C.M. Harris. *Fundamentals of queueing theory*. John Wiley & Sons, Inc. New York, NY, USA, 1985.

[GNU07] GNU. The gnu compiler collection. Software package, available online, 2007. http://gcc.gnu.org/, last accessed Jan/2007.

[HCNSJ05] F. Hernández-Campos, A. B. Nobel, F. D. Smith, and K. Jeffay. Understanding patterns of TCP connection usage with statistical clustering. In *Proceedings of the 13th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 35–44, September 2005. Atlanta, GA.

[HDL$^+$90] L. T. Heberlein, G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 296–304, May 1990.

[Hei93] Juha Heinanen. Multiprotocol Encapsulation over ATM Adaptation Layer 5. RFC 1483 (Proposed Standard), July 1993. Obsoleted by RFC 2684.

[HPK01] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 2001 USENIX Security Symposium*, 2001.

[Hug07] E. Hughes. Qcap. Software package, available online, 2007. http://qcap.sourceforge.net, last accessed Jan/2007.

[IUKB$^+$04] M. Izal, G. Urvoy-Keller, E.W. Biersack, P. Felber, A. Al Hamra, and L. Garces-Erice. Dissecting BitTorrent: Five Months in a Torrent's Lifetime. In *Proceedings of the Passive and Active Measurement workshop (PAM)*. Springer, 2004.

[Kle75] L. Kleinrock. *Queueing systems. Vol. 1, Theory*. Wiley, 1975.

[Kle01] J. Klensin. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), April 2001.

[KPF05] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel traffic classification in the dark. *ACM SIGCOMM Computer Communication Review*, 35(4):229–240, 2005.

[Kre06] C. Kreibich. Netdude. Software package, available online, September 2006. http://netdude.sourceforge.net, last accessed Sept/2006.

[LHF$^+$00] R. Lippmann, J.W. Haines, D.J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, 2000.

[MC03] M. V. Mahoney and P. K. Chan. An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, pages 220–237, 2003.

[McH00] J. McHugh. The 1998 Lincoln Laboratory IDS evaluation (a critique). In *Proceedings of Recent Advances in Intrusion Detection*, pages 145–161, 2000.

[MR96] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939 (Standard), May 1996. Updated by RFCs 1957, 2449.

[MZ05] A. W. Moore and D. Zuev. Internet traffic classification using Bayesian analysis techniques. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 50–60. ACM Press New York, NY, USA, 2005.

[NA06] T. T. T. Nguyen and G. Armitage. Training on multiple sub-flows to optimise the use of machine learning classifiers in real-world IP networks. In *Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN)*, November 2006. Tampa, Florida, U.S.A.

[NLA06] NLANR. Main project page. Web resource, November 2006. http://www.nlanr.net/, last accessed Nov/2006.

[NWK85] J. Neter, W. Wasserman, and M. H. Kutner. *Applied Linear Statistical Models*. Richard D. Irwin, Inc., Chicago, 1985.

[Obj07] Object Management Group. Uml resource page. Web resource, 2007. http://www.uml.org/, last accessed Jan/2007.

[Ost06] S. Ostermann. tcptrace. Software package, available online, November 2006. http://jarok.cs.ohiou.edu/software/tcptrace/, last accessed Nov/2006.

[Pax94] V. Paxson. Empirically derived analytic models of wide-area TCP connections. *ACM Transactions on Networking*, 2(4):316–336, August 1994.

[PD00] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 2nd edition, 2000.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[Pos81a] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[Pos81b] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[Pos82] J. Postel. Simple Mail Transfer Protocol. RFC 821 (Standard), August 1982. Obsoleted by RFC 2821.

[PR83] J. Postel and J.K. Reynolds. Telnet Protocol Specification. RFC 854 (Standard), May 1983.

[PR85] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773.

[Pro06a] GNU Project. The GNU general public licence. Web resource, November 2006. http://www.gnu.org/copyleft/gpl.html, last accessed Nov/2006.

[Pro06b] The GNU Project. coreutils. Software package, available online, November 2006. http://www.gnu.org/software/coreutils/, last accessed Nov/2006.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. San Mateo, CA.

[Rab89] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, February 1989.

[RSSD04] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification. In *Proceedings of the 4th ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 135–148. ACM Press New York, NY, USA, 2004.

[Sim94] W. Simpson. The Point-to-Point Protocol (PPP). RFC 1661 (Standard), July 1994. Updated by RFC 2153.

[TAF01] C. Taylor and J. Alves-Foss. NATE - Network Analysis of Anomalous Traffic Events, a low-cost approach. In *Proceedings of the New Security Paradigms Workshop*, 2001.

[WF99] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[WMM04] C. Wright, F. Monrose, and G. M. Masson. HMM profiles for network traffic classification. *Proceedings of the ACM workshop on Visualization and Data Mining for Computer Security*, pages 9–15, 2004.

[ZNA05a] S. Zander, T. T. T. Nguyen, and G. Armitage. Automated traffic classification and application identification using machine learning. In *Proceedings of the 30th IEEE Conference on Local Computer Networks (LCN)*, November 2005. Sydney, Australia.

[ZNA05b] S. Zander, T.T.T. Nguyen, and G. Armitage. Self-learning IP traffic classification based on statistical flow characteristics. In *Proceedings of the Passive and Active Measurement Workshop (PAM)*. Springer, 2005.

[ZP00] Y. Zhang and V. Paxson. Detecting backdoors. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[ZS05] S. Zander and C. Schmoll. NETMATE – a flexible, extensible, and high-performance passive software meter. Technical Report TR-2005-1110-Meter-NetMate, Fraunhofer FOKUS (Institute for Open Communication Systems), Fraunhofer-Gesellschaft e.V., Germany, 2005.

[ZS06] S. Zander and C. Schmoll. NetMate. Software package, available online, November 2006. http://netmate-meter.sourceforge.net/, last accessed Nov/2006.

[ZWA06] S. Zander, N. Williams, and G. Armitage. Internet archeology: Estimating individual application trends in incomplete historic traffic traces. Technical Report 060313A, Centre for Advanced Internet Architectures, Swinburne University of Technology, March 2006.

Appendix A

Data preparation

The National Laboratory for Applied Network Research (NLANR)[1] [NLA06] was a group funded by the National Science Foundation (NSF) to support and analyze the NSF's high-performance networks. One part of NLANR was the Measurement and Network Analysis team (NLANR/MNA), and one of their functions was to collect and make available traces of network headers from various networks. Many of the traces they have made available are from sources other than the NSF's networks, through partnerships with various institutions around the world. This means that their repository contains a rich variety of traffic from different sources taken at different times, from 1999 through to 2005, but it also means that they have traces in many different formats and with various idiosyncrasies. This appendix documents lessons we learned while using their 'Special' data sets, in hopes that future researchers might benefit from them and to make it easier to reproduce the experiments in this thesis. In the course of our work, we converted traces from these data sets into the more widely usable tcpdump format (also known as "pcap" format, after a common file extension used for such files); we describe in detail in this appendix how this conversion was performed, for reference by other researchers.

A.1 Technologies

The networks from which the NLANR network traces were collected used two common physical-layer protocols, Packet over SONET/SDH (PoS) and Asynchronous Transfer Mode (ATM), which we very briefly describe here. Headers from these protocols appear in the traffic and have to be dealt with appropriately when converting the data to tcpdump format.

Physical and data link protocols

Several different physical and data link protocols were used by the links from which the NLANR data sets were captured. We list the protocols and a quick description of each.

[1]NLANR was supported by funding from the National Science Foundation (cooperative agreement nos. ANI-0129677 (2002) and ANI-9807479 (1998)), but has (as of July 2006) been discontinued; its data, hardware, and website are being maintained by the Cooperative Association for Internet Data Analysis (CAIDA) at the University of California's San Diego Supercomputer Center.

PoS One physical layer protocol commonly seen in the data sets was packet over SONET/SDH (PoS), where SONET/SDH stands for Synchronous Optical NETwork/Synchronous Digital Hierarchy, a standard enabling the transmission of data over optical networks. PoS is a protocol for transmitting packets, such as Ethernet frames, over an optical network.

ATM/AAL5 For the network traces from fiber-optic links in the data sets that we used, the lower layers were generally Asynchronous Transfer Mode with ATM Adaptation Layer 5 (ATM/AAL5); the adaptation layer is specified in ITU recommendation I.363, according to RFC1483 [Hei93].

LLC/SNAP Logical Link Control/Sub-Network Access Protocol (LLC/SNAP) was used as the datalink layer over ATM/AAL5. This is a protocol defined in the Institute of Electrical and Electronics Engineers' (IEEE) 802.2 standard, according to RFC1483 [Hei93]. The LLC/SNAP header includes a type field that indicates the type of data being carried; we are only concerned with frames having a type field of 0x0800, which indicates an IPv4 packet.

Ethernet Data traces that had PoS as a physical layer protocol sometimes used Ethernet (more specifically, IEEE 802.3) as a data link layer. Ethernet frames consist of a 14-byte header that includes the Media Access Control (MAC), or physical, addresses of the source and destination network transceivers, and a type field that indicates what type of data is being carried; we only concern ourselves with frames having a type field of 0x0800, which indicates an IPv4 packet, as with LLC/SNAP [Dev06b].

Cisco HDLC Some PoS traces used a Cisco protocol called High-level Data Link Control (HDLC) at the datalink layer rather than Ethernet. This is a fairly simple protocol, consisting of a 4-byte header and a 4-byte trailer; the header includes a type field where 0x0800 is again the value used to indicate that the frame carries an IPv4 packet.

PPP Another datalink-layer protocol seen on PoS traces was the Point-to-Point Protocol (PPP), specified in RFC1661 [Sim94].

DAG file format

All of the NLANR trace files we used in this thesis were originally in a format called "DAG", which was developed by Endace for their network capture hardware (we were unable to determine what, if anything, the acronym represents). There are apparently several versions of the DAG format that have been used over time; we refer to some of them as "legacy" DAG formats, following terminology from CAIDA's CoralReef. These are older DAG formats that do not contain header information identifying the protocol layers on the captured link or the snapshot length (the amount of data captured from each packet seen), and thus this information must be found (in our case, by reverse-engineering combined with some trial-and-error) to properly convert the file.

A.2 Conversion and processing tools

We used a collection of different tools to massage the trace data into a more easily usable format; links to the tools used and a brief description of what they are and how we used them are presented here. Commands shown here and in the remainder of the appendix use angle brackets to delimit a variable argument (e.g. `<sourcefile.gz>` indicates a place where the input filename, having a `.gz` extension, is to be inserted in the command). We also describe a number of tools used in the further preparation of the data, particularly timeslicing and demultiplexing it into sample files containing individual flows.

CoralReef crl_to_pcap

http://www.caida.org/tools/measurement/coralreef/

The Cooperative Association for Internet Data Analysis (CAIDA) [fIDAC06] developed a suite of tools for manipulating network traffic called CoralReef. One tool that we have found particularly useful in our research is called `crl_to_pcap`; it is a utility for converting between different network capture file formats. We use it primarily to convert assorted different DAG file formats[2] to tcpdump format.

We used the version of `crl_to_pcap` included in CoralReef version 3.7.5.

TCPTrace

http://jarok.cs.ohiou.edu/software/tcptrace/

Shawn Ostermann's [Ost06] `tcptrace` is an application for analyzing captures of TCP traffic and modelling the interactions of the protocols. We use it to quickly count the packets in each direction of a TCP flow, though this is a tiny fraction of its functionality. We also used more of its functionality while conducting analysis of the traffic.

We used version 6.6.1 of `tcptrace`.

Wireshark utilities

http://www.wireshark.org

Wireshark [Ct06], formerly called Ethereal, is a network traffic display and parsing GUI application, and includes a number of tools for the manipulation of network traces. Of particular use in this thesis were the `editcap` and `mergecap` utilities; the former performs various transformations and divisions of a trace file, while the latter merges multiple trace files into a single one. We used `editcap` mainly to subdivide network traces by date, for creating 5-minute timeslices of network traffic, and we used `mergecap` to combine unidirectional traces into bidirectional traces, as many of the data sets had traffic intercepted in each direction separately.

We used the version of these utilities included in the source distribution of Ethereal 0.99.0.

[2]DAG is a format developed by Endace for their network capture hardware

ANTARES utilities

http://antares-net.sourceforge.net/

Our ANTARES toolkit [Fur06] includes a number of scripts that we used to semi-automate the processing of data sets. Many of these are discussed in the context of that processing; in this context, we only wish to specifically mention `dag_eth_to_pcap`, which is a utility for taking Ethernet traces in a legacy DAG format and converting them to tcpdump format.

The version of the ANTARES toolkit used for this thesis is available on the `antares-net` SourceForge project as version 0.1.

dd and cat

`dd` and `cat` are core utilities from the GNU project [Pro06b], probably included in most Linux and Unix distributions. They are not tools specifically for use with trace files; dd is a low-level file copy command and cat outputs a file on standard output, and both happen to be useful in dealing with large files. Many of the tools listed here will fail with error if invoked on files that are too large ($> 2Gb$ on many Linux systems); however, some of them (`crl_to_pcap` and `dag_eth_to_pcap`) are capable of reading from standard input, which provides a way around the file size limit. Two examples are given here of how to do this.

Converting a `gzip`–compressed legacy DAG “CHDLC over POS” trace (48 bytes of packet data per record) to pcap with `dd` and Coralreef’s `crl_to_pcap`:

```
dd if=<sourcefile.gz> bs=4K | crl_to_pcap
    -C "src dag:- phy=POS,proto=CHDLC,48" -o <outputfile.pcap>
```

Note that this command is intended to be issued on a single line; we have broken the line here to fit the margins of the page.

Converting a `gzip`–compressed legacy DAG Ethernet trace (64-byte overall record length) to pcap with `cat` and `dag_eth_to_pcap`:

```
cat <sourcefile.gz> | dag_eth_to_pcap -z - <outputfile.pcap>
```

We used `dd` and `cat` from the GNU coreutils package, version 5.2.1.

A.3 Converting NLANR data to tcpdump format

Before using the NLANR datasets, we converted them to tcpdump format, which required that we first identify the protocols being used. We document here those findings and the procedure for converting the data format, as we found that this information was not readily available, and other researchers wishing to use these data sets will likely need to perform the same conversions. We also developed a tool for converting the NZIX data set, which we have made available as part of the ANTARES toolkit; the `dagtools` package recommended

Data set	Timeframe	Source	Volume (Gb)
NZIX-II	Summer 2000	New Zealand ISP peering point	x
Auckland-IV (ATM)	Winter 2001	University of Auckland uplink	x
Auckland-VI (Eth)	Spring 2001	University of Auckland uplink	x
ABILENE-II	Autumn 2002	ABILENE research backbone	x
Leipzig-I (PPP)	Autumn 2002	University of Leipzig uplink	x
Leipzig-II (Eth)	Winter 2003	University of Leipzig uplink	x
ABILENE-III	Summer 2004	ABILENE research backbone	x
ABILENE-V	August 2004	ABILENE research backbone	x

Table A.1: Summary of NLANR data sets used

Group	Physical layer	Datalink layer	Snap length
ABILENE	PoS	Cisco HDLC	48
Auckland-ATM	ATM/AAL5	LLC/SNAP	48
Auckland-Eth	None	Ethernet	54(64)
Leipzig-PPP	PoS	PPP	48
Leipzig-Eth	None	Ethernet	48
NZIX	None	Ethernet	54(64)

Table A.2: Technical details of NLANR datasets for conversion

by NLANR for this purpose does not appear to be publically available anymore, and is apparently only available from Endace with the purchase of network capture hardware.

In table A.1 we repeat the table of datasets used in our experiments from section 5.1.3; table A.2 lists them in groups, showing the lower layers of protocols in use and the snap length (the amount of data captured from each packet). The parameters in the latter table are needed for the conversion, and will be described in more detail in the section for the appropriate data set. Note that for the Ethernet traces, a second length is given in parentheses in the snap length column; when using the `dag_eth_to_pcap` tool to convert the, the parenthesized length, which including the DAG header, is used as the record length. The unparenthesized length in these rows is the actual amount of packet data contained in each record, including the 14-byte Ethernet header.

ABILENE

ABILENE-I *http://pma.nlanr.net/Traces/long/ipls1.html*

ABILENE-II *http://pma.nlanr.net/Traces/long/ipls2.html*

ABILENE-III *http://pma.nlanr.net/Special/ipls3.html*

ABILENE-IV *http://pma.nlanr.net/Special/ipls4.html*

ABILENE-V *http://pma.nlanr.net/Special/ipls5.html*

The ABILENE data sets were collected at a router at the Indianapolis router node of the Internet2 ABILENE backbone[3]; specifically, this work uses traffic from the link from Indianapolis to Kansas City (IPLS-KSCY).

The traffic from these data sets is packet-over-SONET (POS) at the physical layer, and Cisco HDLC at the datalink layer. The files are in a legacy Endace DAG format, which means that they do not contain information about the framing and protocols, so that information must be supplied to the `crl_to_pcap` tool in order to process them correctly.

We used CoralReef's `crl_to_pcap` tool to convert these trace files to pcap, using the following command:

```
crl_to_pcap -C "src dag:<sourcefile> phy=PoS,proto=CHDLC,48"
-o <outputfile>
```

Auckland

Auckland-II *http://pma.nlanr.net/Traces/long/auck2.html*

Auckland-IV *http://pma.nlanr.net/Traces/long/auck4.html*

Auckland-VI *http://pma.nlanr.net/Traces/long/auck6.html*

Auckland-VII *http://pma.nlanr.net/Traces/long/auck7.html*

Auckland-VIII *http://pma.nlanr.net/Special/auck8.html*

The Auckland data sets were collected at the Internet uplink of the University of Auckland, in New Zealand.

There were two distinct types of traces among the Auckland data sets: ATM, and Ethernet. These are shown in table A.2 as Auckland-ATM and Auckland-Eth, respectively. Auckland-II, Auckland-IV, and Auckland-VII are ATM, Auckland-VIII is Ethernet, and Auckland-VI gives both. ATM trace file names end in `-0` and `-1` while Ethernet trace file names for Auckland-VI end in `-e0` and `-e1` (for two different collection points). The Auckland-VIII traces have no particular extension. The ATM traces were collected on the outside of the Internet-facing router, while the Ethernet traces were collected on the inside of it.

We used the CoralReef `crl_to_pcap` tool to convert the Auckland-IV ATM trace files to tcpdump format, using the following command:

```
crl_to_pcap -C "src dag:<sourcefile> phy=ATM,proto=ATM_RFC1483,48"
-o <outputfile>
```

For the Auckland-VI data set, we used the `-e1` data files, converting them to pcap using the `dag_eth_to_pcap` tool that we developed as part of the ANTARES toolkit; we simply used the following command:

[3]Internet2 is a not-for-profit consortium of academic and industrial parties collaborating in developing advanced networking technologies; ABILENE is a 10 gigabit national fiber backbone network in the United States built and maintained by Internet2

```
dag_eth_to_pcap -z -l 64 <sourcefile.gz> <outputfile>
```

The `-l 64` argument is not strictly necessary, as the tool defaults to a 64 byte record length. The `-z` argument indicates that the input file is compressed with `gzip`, so that one does not need to explicitly decompress the files before converting them.

Leipzig

Leipzig-I *http://pma.nlanr.net/Special/leip1.html*

Leipzig-II *http://pma.nlanr.net/Special/leip2.html*

The Leipzig data sets were collected at the Internet uplink of the University of Leipzig in Germany. Both Leipzig-I and Leipzig-II include PPP-over-PoS traces, collected on the outside link of the border router, while Leipzig-II also includes Ethernet traces from the inside link of the router; these are denoted in table A.2 as Leipzig-PPP and Leipzig-Eth, respectively. The PPP trace filenames end in `-0` and `-1`, not including the `.gz` extension, and the Ethernet traces end in `-e`.

We used the PPP traces from both, converting them to tcpdump format with the following command:

```
crl_to_pcap -C "src dag:<sourcefile.gz> phy=PoS,proto=PPP,48"
-o <outputfile.pcap>
```

From Leipzig-II the Ethernet traces are in a more modern DAG file format than those of Auckland-VI, which include enough information that `crl_to_pcap` does not need to be manually configured. They can be converted to tcpdump format as follows:

```
crl_to_pcap -C "src dag:<sourcefile.gz>" -o <outputfile.pcap>
```

NZIX

NZIX-II *http://pma.nlanr.net/Traces/long/nzix2.html*

The NZIX-II dataset is from the New Zealand Internet Exchange, a peering point for several public Internet Service Providers in New Zealand, hosted at the University of Waikato.

The trace files for this data set are in a legacy DAG Ethernet format; we used our `dag_eth_to_pcap` tool to convert them to tcpdump format as follows:

```
dag_eth_to_pcap -z -l 64 <sourcefile.gz> <outputfile.pcap>
```

A.4 Preparing data sets

In this section, we present a more detailed description of how we generated a sample data set from the converted NLANR traces, as outlined in section 5.1.3. Our goal was to come up with a balanced set of samples that represented the usual behaviour of the applications being studied, and minimized the effects of anomalous uses. We took five-minute timeslices of the traces, divided them by data set, time period (work hours vs. non-work hours), and application (by port), selected samples from each division in such a way as to minimize the influence of any one pair of hosts, and took one hundred samples from each division. Unless otherwise noted, the tools discussed here are part of the ANTARES toolkit, available in the `tools/` directory of the distribution.

One of our data sets consisted of five-minute samples, so we timesliced all of our datasets into five-minute intervals, for consistency. The traces were timesliced using our tool `timeslice_trace.pl`; this tool is a wrapper for the *editcap* tool from the WireShark toolkit, formerly known as Ethereal [Ct06]. For example, to divide a one-hour trace file that begins at 12:00:00 PM on December 12th, 2003 into five-minute timeslices, we would use the command:

```
timeslice_trace.pl <inputfile.pcap> "2003-12-12 12:00:00" 5m 12
```

where `5m 12` indicates twelve five-minute timeslices.

Note that time zones are an issue with the timeslicing; the time has to be given relative to local time, so if a capture file begins at 12 PM GMT and the tool is being run on a system in time zone GMT-5, the time given should be "7:00:00" – usually. However, due to a peculiarity of the `editcap` tool, if the date of the capture file falls within daylight savings time for the local timezone, an additional hour must be subtracted (so in the example, one would use "6:00:00" in place of "7:00:00"). These issues are unfortunate, but avoiding them would require modifying `editcap`.[4]

The partitioning of the traces into five-minute timeslices means that no sample flow will be longer than five minutes in duration. We consider this to be an advantage for the purposes of studying application behaviours, as we feel five-minute timeslices of longer network flows are more likely to be homogeneous than the full flows would have been; i.e. for an application that is capable of performing distinct activities, a five-minute flow is more likely to be generated by a single type of activity than a thirty-minute flow would be. However, the timeslicing is a factor that may have affected our results in unexpected ways.

We then separated the timesliced trace files for each data set by time period; timeslices falling between 8 AM and 4 PM local time (local to the collection point) were considered to be during *work* hours, and timeslices outside those times were considered to be during *off-work* hours. This was intended to allow us to explore whether there were differences in the uses of network applications in different periods of the day.

The tool used to divide the data sets by application was `decompose_trace.pl`; the default configuration file `etc/portlist.txt` will extract the same set of applications used in this thesis. The tool is a wrapper for the `tcpdump` tool [Dev06a]. More specifically, the

[4]We have asked the Wireshark development team to consider allowing an explicit time zone to be specified, which would allow us to avoid this ugliness.

configuration file lists “known” ports; a flow which involves a port in that list is considered to be an instance of the associated application. If a flow involved two ports that are both on the list, it is considered to be an instance of the application corresponding to the lower of the two ports. When using the default portlist, the tool can be executed simply as:

```
decompose_trace.pl <inputfile.pcap>
```

Dividing the data into *cells* by data set, time period, and application, we used a tool on each cell that decomposed the traces into network flows and selected a subset of those flows as samples. The samples were selected using our tool `sample_traffic.pl`; this tool is a wrapper around the `demux` module of the libnetdude library, of the Netdude suite [Kre06]. Specifically, it ensured that no more than two samples involved the same pair of IP addresses (one pair in each direction, i.e. $A \rightarrow B$ and $B \rightarrow A$). This restriction was intended to reduce the chance that the samples would be strongly influenced by a small group of hosts performing some unusual activity involving a port of interest. We used the command:

```
sample_traffic.pl -l 10 -t 64 -o <output dir> <inputfile.pcap>
```

where `-l 10` indicated we only wanted samples with at least 10 packets, and `-t 64` indicated that we wanted to use a timeout of 64 seconds to close off inactive flows. This extracted each sample flow to its own file in tcpdump format.

Finally, having divided the data into 84 cells (7 effective data sets, 2 time periods, and 6 applications: $7*2*6=84$), we used a simple tool called `select_samples.pl` to randomly select 100 samples from the potential samples in each cell with the command:

```
select_samples.pl -n 100 -o <output dir> <input dir list>
```

Each sample file was then processed with a tool, `profile_streams_thesis` built with the ANTARES toolkit; that tool is available in the `src/testing` directory of the toolkit. That output the flow attribute values of interest, which we formatted in comma-separated value (CSV) format using the `profile_to_csv.pl` tool as:

```
profile_to_csv.pl -p <server port>
    -l "<data set>, <time period>, <application>"
    -o <outputfile.csv> <inputfile.profile>
```

where `-p <server port>` allowed us to specify which port was the server port (e.g. 23 for Telnet), so that the forward and reverse directions could be labelled, and the `-l` switch and arguments allowed us to label the row with the data set, time period, and application for later analysis. The CSV files produced in this manner were then loaded into the R statistical processing package [Dev05], and classifiers were trained as detailed in section 5.1.4.

We have explained here in detail the process by which we prepared our data sets for analysis in hopes that other researchers will benefit from our tools and methods.

Appendix B

Error tables

In this appendix, we present the error tables for classifiers trained to distinguish between two classes of network traffic using a single flow attribute. These classifiers were developed with logistic regression as described in section 5.1.4. Each section of this appendix contains a set of tables for one application of interest; each table shows the combined error rates of classifiers trained to distinguish network traffic from the application of interest from those of one other class of network traffic (the columns) using a single flow attribute (the rows). The tables are broken up in loose groupings such that each would fit on a page; the first table in each section contains classifiers trained using flow attributes computed on both directions of the flow, the second has those using flow attributes computed separately on each half-flow, and the third and fourth focus on small and large packet heuristics, respectively. The flow attributes themselves are those listed in table 3.1 and described in section 3.3.

Each cell represents the combined error rate of a classifier trained to distinguish between two applications using a single flow attribute. We will explain the tables, using as an example table B.1. All of the error rates in that table pertain to classifiers trained to distinguish POP3 from other applications; the other applications are in columns, e.g. the leftmost column of error rates in the table consists of classifiers trained to distinguish between POP3 and FTP-data. Each row contains error rates of classifiers trained using a particular flow attribute; for table B.1, the top row of error rates correspond to classifiers trained using *pkt_count*, the total number of packets. So the top left error rate in table B.1 is the combined error rate of a classifier trained to distinguish between POP3 and FTP-data using the *pkt_count*, evaluated using 10-fold cross-validation as described in section 2.1.1.[1]

In an attempt to make the tables somewhat more readable, we have coloured each cell according to the error rate; the colors range from green (relatively good classifiers) through yellow (mediocre classifiers) to red (poor classifiers).[2] This makes it easier to identify general trends in the data; a column that is mostly green and yellow indicates a pair of

[1]Recall from section 2.1.1 that the combined error rate is the total number of errors, both false positives and false negatives, divided by the total number of samples. For the example given, this would be the sum of the number of POP3 flows classified as FTP-data and the number of FTP-data flows classified as POP3, divided by the total number of flows of both POP3 and FTP-data.

[2]The coloring is designed so that an error rate of 0 is green and an error rate of 0.5 (no better than random chance) or worse is red, and so that there is an even balance of red and green at an error rate of 0.25, yielding orange.

applications that was generally easy to distinguish, while one that is mostly red indicates a pair of applications that was more difficult to distinguish. Rows that are mostly green indicate a flow attribute that is good at distinguishing a particular application (the one on which that table focuses) from others; often there will be several adjacent rows, pertaining to similar flow attributes, that are all mostly green, which indicates a class of flow attributes that are useful for distinguishing that application from others.

Also interesting are regions of generally good (green) classifiers that have a single column of poor (red) classifiers; we take this to indicate that the application for that column is similar to the target application, at least with respect to those flow attributes. An excellent example of this is table B.7, which shows small packet heuristics for FTP-data; the bulk of the classifiers are green, except for the HTTP column, which we interpret as meaning that FTP-data is easy to distinguish by the proportion of small packets (in this case, its lack thereof), except that HTTP is similar to it in that respect. This is mirrored in table B.23, which suggests that, at least in terms of proportions of small packets, HTTP and FTP-data are similar to each other and dissimilar from all the other applications that we considered.

These tables (in full colour), along with the rest of the thesis, will be available on the SourceForge page for ANTARES [Fur06], the toolkit we have developed.

B.1 POP3

	FTP-data	FTP-ctrl	Telnet	SMTP	HTTP
pkt_count	0.2700	0.3614	0.2261	0.4004	0.4929
duration	0.3404	0.2686	0.1464	0.4225	0.4954
nonempty_count	0.2750	0.3171	0.2264	0.4007	0.4264
pkt_byte_count	0.2221	0.4025	0.2789	0.2568	0.3089
payload_byte_count	0.2157	0.3736	0.2946	0.2261	0.2839
mean_delay	0.4339	0.2846	0.2382	0.4650	0.4932
mean_pkt_len	0.1439	0.5411	0.3821	0.2407	0.2243
mean_payload_len	0.1475	0.5939	0.3461	0.2218	0.2096
mean_nonempty_payload_len	0.1243	0.5446	0.3600	0.2204	0.1054
dir_data	0.1593	0.5168	0.5582	0.0904	0.5793
mean_pkt_datarate	0.2046	0.2932	0.2907	0.3875	0.3189
mean_payload_datarate	0.1804	0.3400	0.3761	0.3336	0.2796
flag_urg	0.5286	0.5200	0.5221	0.5236	0.5121
flag_ack	0.3236	0.5296	0.1511	0.4036	0.2843
flag_psh	0.1071	0.2604	0.3093	0.4518	0.1279
flag_rst	0.4696	0.4189	0.4829	0.4557	0.4425
flag_syn	0.3043	0.2932	0.1400	0.2939	0.4025
flag_fin	0.2982	0.2732	0.1561	0.2957	0.5154

Table B.1: POP3 — per-flow metrics

	FTP-data	FTP-ctrl	Telnet	SMTP	HTTP
pkt_count_fwd	0.2986	0.3279	0.2150	0.3607	0.4879
pkt_byte_count_fwd	0.2736	0.2889	0.2061	0.1179	0.1725
payload_byte_count_fwd	0.1036	0.1607	0.2686	0.0804	0.0332
nonempty_count_fwd	0.1036	0.3129	0.2075	0.2261	0.1286
mean_delay_fwd	0.4393	0.2846	0.2525	0.4839	0.5275
mean_pkt_len_fwd	0.3707	0.2168	0.3604	0.0700	0.0382
mean_payload_len_fwd	0.1068	0.1896	0.1800	0.0550	0.0229
mean_nonempty_payload_len_fwd	0.1025	0.1714	0.1314	0.0346	0.0075
flag_urg_fwd	0.5314	0.5243	0.5179	0.5171	0.5186
flag_ack_fwd	0.1154	0.4393	0.1407	0.3450	0.2864
flag_psh_fwd	0.1018	0.4614	0.4836	0.5121	0.1064
flag_rst_fwd	0.4921	0.4118	0.4814	0.4543	0.4339
flag_syn_fwd	0.3211	0.2839	0.1314	0.2868	0.4436
flag_fin_fwd	0.3196	0.2754	0.1568	0.3093	0.5350
pkt_count_rev	0.2711	0.3736	0.2379	0.4257	0.4929
pkt_byte_count_rev	0.2525	0.4586	0.2868	0.4911	0.3214
payload_byte_count_rev	0.2571	0.4532	0.2964	0.5668	0.3039
nonempty_count_rev	0.3021	0.3282	0.2364	0.4314	0.5100
mean_delay_rev	0.4250	0.2861	0.2086	0.4443	0.5375
mean_pkt_len_rev	0.2246	0.5339	0.3654	0.4904	0.2582
mean_payload_len_rev	0.2404	0.5739	0.3400	0.5568	0.2525
mean_nonempty_payload_len_rev	0.2293	0.5225	0.3546	0.5961	0.1879
flag_urg_rev	0.5200	0.5286	0.5207	0.5207	0.5200
flag_ack_rev	0.2600	0.5246	0.5279	0.5164	0.5161
flag_psh_rev	0.1318	0.2254	0.2500	0.4082	0.1557
flag_rst_rev	0.5032	0.5186	0.5211	0.5125	0.5243
flag_syn_rev	0.3007	0.3136	0.1457	0.3314	0.3800
flag_fin_rev	0.2868	0.2732	0.1582	0.2882	0.3854

Table B.2: POP3 — per-half-flow metrics

	FTP-data	FTP-ctrl	Telnet	SMTP	HTTP
sp_alpha_1_fwd	0.0389	0.3811	0.4279	0.1418	0.0321
sp_alpha_2_fwd	0.0693	0.4657	0.4561	0.4307	0.0307
sp_alpha_3_fwd	0.2339	0.5082	0.5211	0.4711	0.2071
sp_beta_1_fwd	0.0118	0.1704	0.4339	0.0154	0.0096
sp_beta_2_fwd	0.0271	0.4139	0.4171	0.0679	0.0050
sp_beta_3_fwd	0.0336	0.4821	0.4657	0.0732	0.0293
sp_gamma_1_fwd	0.0371	0.1921	0.3061	0.0414	0.0304
sp_gamma_2_fwd	0.0529	0.4296	0.3329	0.1764	0.0261
sp_gamma_3_fwd	0.0589	0.4618	0.3457	0.2061	0.0264
sp_delta_1_fwd	0.0557	0.3214	0.3950	0.1432	0.0446
sp_delta_2_fwd	0.0782	0.5236	0.3975	0.4382	0.0404
sp_delta_3_fwd	0.0493	0.4986	0.4029	0.5100	0.0225
sp_alpha_1_rev	0.3675	0.4350	0.3129	0.4289	0.3596
sp_alpha_2_rev	0.1025	0.4211	0.4425	0.4339	0.0725
sp_alpha_3_rev	0.3039	0.4296	0.3886	0.5350	0.2825
sp_beta_1_rev	0.1186	0.3943	0.3696	0.3304	0.1121
sp_beta_2_rev	0.0907	0.4732	0.4046	0.5054	0.0668
sp_beta_3_rev	0.1096	0.4346	0.4107	0.4157	0.1457
sp_gamma_1_rev	0.3675	0.4518	0.3161	0.4400	0.3557
sp_gamma_2_rev	0.1125	0.4643	0.4814	0.5089	0.0700
sp_gamma_3_rev	0.1082	0.3554	0.4479	0.4664	0.0864
sp_delta_1_rev	0.3675	0.4489	0.2550	0.4239	0.3529
sp_delta_2_rev	0.1132	0.3789	0.3354	0.4107	0.0532
sp_delta_3_rev	0.0596	0.3757	0.4193	0.4471	0.0475

Table B.3: POP3 — small packet heuristics

	FTP-data	FTP-ctrl	Telnet	SMTP	HTTP
lp_alpha_1_fwd	0.4507	0.5150	0.5200	0.1907	0.4804
lp_alpha_2_fwd	0.4675	0.5186	0.5264	0.2918	0.5179
lp_alpha_3_fwd	0.4807	0.5171	0.5189	0.3900	0.5171
lp_alpha_4_fwd	0.4350	0.5200	0.5050	0.1686	0.4561
lp_alpha_5_fwd	0.4582	0.5214	0.5196	0.2818	0.5182
lp_alpha_6_fwd	0.4754	0.5171	0.5293	0.3861	0.5243
lp_beta_1_fwd	0.4307	0.5082	0.4775	0.0718	0.0404
lp_beta_2_fwd	0.4518	0.5243	0.5182	0.1532	0.4746
lp_beta_3_fwd	0.4707	0.5286	0.5207	0.3136	0.5111
lp_beta_4_fwd	0.4307	0.5232	0.4775	0.0718	0.0404
lp_beta_5_fwd	0.4518	0.5264	0.5229	0.1532	0.4746
lp_beta_6_fwd	0.4707	0.5414	0.5139	0.3136	0.5004
lp_gamma_1_fwd	0.4336	0.5254	0.5011	0.1525	0.4329
lp_gamma_2_fwd	0.4561	0.5207	0.5218	0.2736	0.5221
lp_gamma_3_fwd	0.4732	0.5193	0.5132	0.3836	0.5271
lp_gamma_4_fwd	0.4336	0.5254	0.4921	0.1525	0.4329
lp_gamma_5_fwd	0.4561	0.5250	0.5136	0.2736	0.5111
lp_gamma_6_fwd	0.4732	0.5257	0.5261	0.3836	0.5200
lp_alpha_1_rev	0.2750	0.3968	0.4350	0.3843	0.3221
lp_alpha_2_rev	0.2764	0.4493	0.4743	0.4421	0.3275
lp_alpha_3_rev	0.3489	0.4661	0.4714	0.4596	0.3900
lp_alpha_4_rev	0.2629	0.3932	0.4082	0.3807	0.3193
lp_alpha_5_rev	0.2668	0.4479	0.4789	0.4407	0.3261
lp_alpha_6_rev	0.3443	0.4657	0.4718	0.4589	0.3889
lp_beta_1_rev	0.1939	0.4646	0.4146	0.3725	0.1471
lp_beta_2_rev	0.2289	0.4504	0.5425	0.4200	0.2771
lp_beta_3_rev	0.3264	0.4539	0.5004	0.4468	0.3593
lp_beta_4_rev	0.1939	0.4636	0.4129	0.3725	0.1464
lp_beta_5_rev	0.2282	0.4507	0.5511	0.4200	0.2768
lp_beta_6_rev	0.3261	0.4539	0.5004	0.4468	0.3611
lp_gamma_1_rev	0.2189	0.3943	0.5411	0.3779	0.2779
lp_gamma_2_rev	0.2436	0.4461	0.4832	0.4382	0.3168
lp_gamma_3_rev	0.3311	0.4650	0.4711	0.4582	0.3843
lp_gamma_4_rev	0.2186	0.3943	0.5400	0.3779	0.2786
lp_gamma_5_rev	0.2432	0.4461	0.4811	0.4382	0.3171
lp_gamma_6_rev	0.3307	0.4654	0.4693	0.4582	0.3839

Table B.4: POP3 — large packet heuristics

B.2 FTP-data

	POP3	FTP-ctrl	Telnet	SMTP	HTTP
pkt_count	0.2700	0.3239	0.5107	0.2993	0.2864
duration	0.3404	0.4568	0.3296	0.4139	0.3921
nonempty_count	0.2750	0.3468	0.5268	0.3032	0.2904
pkt_byte_count	0.2221	0.1771	0.3350	0.2654	0.3050
payload_byte_count	0.2157	0.1464	0.3229	0.2654	0.3068
mean_delay	0.4339	0.2925	0.2182	0.4296	0.4564
mean_pkt_len	0.1439	0.0593	0.1318	0.2089	0.3761
mean_payload_len	0.1475	0.0582	0.1361	0.2129	0.3711
mean_nonempty_payload_len	0.1243	0.0482	0.0943	0.1621	0.3914
dir_data	0.1593	0.1668	0.2282	0.2475	0.1636
mean_pkt_datarate	0.2046	0.1111	0.0986	0.3011	0.3796
mean_payload_datarate	0.1804	0.0950	0.0964	0.2882	0.3711
flag_urg	0.5286	0.5200	0.5079	0.5293	0.5279
flag_ack	0.3236	0.4457	0.3757	0.4800	0.3043
flag_psh	0.1071	0.0964	0.0889	0.1304	0.3864
flag_rst	0.4696	0.3850	0.5243	0.4221	0.4054
flag_syn	0.3043	0.4646	0.3411	0.3664	0.3168
flag_fin	0.2982	0.5357	0.3600	0.3496	0.3561

Table B.5: FTP-data — per-flow metrics

	POP3	FTP-ctrl	Telnet	SMTP	HTTP
pkt_count_fwd	0.2986	0.3361	0.5271	0.3221	0.3075
pkt_byte_count_fwd	0.2736	0.3193	0.4771	0.4457	0.3614
payload_byte_count_fwd	0.1036	0.1364	0.1850	0.1432	0.1021
nonempty_count_fwd	0.1036	0.1550	0.1518	0.1082	0.1764
mean_delay_fwd	0.4393	0.2989	0.2543	0.4429	0.4732
mean_pkt_len_fwd	0.3707	0.5571	0.4271	0.1493	0.1586
mean_payload_len_fwd	0.1068	0.1182	0.1239	0.1046	0.1054
mean_nonempty_payload_len_fwd	0.1025	0.1193	0.3068	0.1364	0.0836
flag_urg_fwd	0.5314	0.5307	0.5250	0.5171	0.5257
flag_ack_fwd	0.1154	0.1821	0.4168	0.1168	0.1229
flag_psh_fwd	0.1018	0.1075	0.1200	0.1071	0.2064
flag_rst_fwd	0.4921	0.3911	0.4968	0.4339	0.4089
flag_syn_fwd	0.3211	0.5368	0.3321	0.4443	0.3150
flag_fin_fwd	0.3196	0.5246	0.3496	0.5771	0.3768
pkt_count_rev	0.2711	0.2893	0.4896	0.2625	0.2850
pkt_byte_count_rev	0.2525	0.1921	0.3739	0.1375	0.3536
payload_byte_count_rev	0.2571	0.1968	0.4193	0.1343	0.3954
nonempty_count_rev	0.3021	0.3439	0.5175	0.2336	0.3446
mean_delay_rev	0.4250	0.2700	0.1982	0.4125	0.4693
mean_pkt_len_rev	0.2246	0.1304	0.2439	0.1046	0.4271
mean_payload_len_rev	0.2404	0.1536	0.3318	0.1043	0.5496
mean_nonempty_payload_len_rev	0.2293	0.1282	0.2836	0.1014	0.5275
flag_urg_rev	0.5200	0.5264	0.5214	0.5286	0.5250
flag_ack_rev	0.2600	0.2611	0.2689	0.2632	0.2725
flag_psh_rev	0.1318	0.1061	0.1179	0.1793	0.4739
flag_rst_rev	0.5032	0.5164	0.4850	0.5132	0.4989
flag_syn_rev	0.3007	0.4289	0.3768	0.3250	0.3204
flag_fin_rev	0.2868	0.4904	0.3889	0.3111	0.3361

Table B.6: FTP-data — per-half-flow metrics

	POP3	FTP-ctrl	Telnet	SMTP	HTTP
sp_alpha_1_fwd	0.0389	0.1750	0.1300	0.4093	0.5186
sp_alpha_2_fwd	0.0693	0.1121	0.1304	0.1254	0.4789
sp_alpha_3_fwd	0.2339	0.2964	0.2568	0.3068	0.4786
sp_beta_1_fwd	0.0118	0.0800	0.0621	0.1250	0.5071
sp_beta_2_fwd	0.0271	0.0543	0.0632	0.1143	0.4814
sp_beta_3_fwd	0.0336	0.0514	0.0550	0.1125	0.5171
sp_gamma_1_fwd	0.0371	0.1732	0.1379	0.4043	0.5029
sp_gamma_2_fwd	0.0529	0.1068	0.1436	0.1661	0.4793
sp_gamma_3_fwd	0.0589	0.0989	0.1414	0.1361	0.4739
sp_delta_1_fwd	0.0557	0.1729	0.1468	0.4025	0.5118
sp_delta_2_fwd	0.0782	0.1086	0.1607	0.1204	0.4793
sp_delta_3_fwd	0.0493	0.0918	0.1139	0.0639	0.4739
sp_alpha_1_rev	0.3675	0.4321	0.1704	0.4386	0.5032
sp_alpha_2_rev	0.1025	0.1243	0.1368	0.1264	0.4796
sp_alpha_3_rev	0.3039	0.2154	0.1564	0.3004	0.4829
sp_beta_1_rev	0.1186	0.1543	0.1146	0.3254	0.5129
sp_beta_2_rev	0.0907	0.0643	0.1218	0.0604	0.4907
sp_beta_3_rev	0.1096	0.0432	0.1011	0.0443	0.4454
sp_gamma_1_rev	0.3675	0.4111	0.1636	0.4339	0.4964
sp_gamma_2_rev	0.1125	0.1043	0.1429	0.0914	0.4775
sp_gamma_3_rev	0.1082	0.0729	0.1196	0.0904	0.5093
sp_delta_1_rev	0.3675	0.4164	0.1721	0.4636	0.4857
sp_delta_2_rev	0.1132	0.1179	0.1471	0.1521	0.4743
sp_delta_3_rev	0.0596	0.0739	0.0861	0.0743	0.4989

Table B.7: FTP-data — small packet heuristics

	POP3	FTP-ctrl	Telnet	SMTP	HTTP
lp_alpha_1_fwd	0.4507	0.4532	0.4571	0.2379	0.4732
lp_alpha_2_fwd	0.4675	0.4679	0.4686	0.3232	0.4682
lp_alpha_3_fwd	0.4807	0.4807	0.4814	0.4093	0.4807
lp_alpha_4_fwd	0.4350	0.4354	0.4411	0.2368	0.4796
lp_alpha_5_fwd	0.4582	0.4586	0.4593	0.3239	0.4607
lp_alpha_6_fwd	0.4754	0.4754	0.4761	0.4111	0.4757
lp_beta_1_fwd	0.4307	0.4325	0.4482	0.2311	0.1257
lp_beta_2_fwd	0.4518	0.4525	0.4554	0.2868	0.4775
lp_beta_3_fwd	0.4707	0.4707	0.4721	0.3436	0.4771
lp_beta_4_fwd	0.4307	0.4325	0.4479	0.2311	0.1257
lp_beta_5_fwd	0.4518	0.4521	0.4554	0.2807	0.4779
lp_beta_6_fwd	0.4707	0.4707	0.4721	0.3439	0.4771
lp_gamma_1_fwd	0.4336	0.4343	0.4404	0.3164	0.4993
lp_gamma_2_fwd	0.4561	0.4568	0.4582	0.3239	0.4607
lp_gamma_3_fwd	0.4732	0.4732	0.4743	0.4107	0.4736
lp_gamma_4_fwd	0.4336	0.4343	0.4404	0.3168	0.4993
lp_gamma_5_fwd	0.4561	0.4568	0.4579	0.3254	0.4607
lp_gamma_6_fwd	0.4732	0.4732	0.4739	0.4107	0.4736
lp_alpha_1_rev	0.2750	0.1546	0.3246	0.1400	0.4800
lp_alpha_2_rev	0.2764	0.2196	0.2468	0.2061	0.4671
lp_alpha_3_rev	0.3489	0.3129	0.3189	0.3014	0.4682
lp_alpha_4_rev	0.2629	0.1464	0.3389	0.1293	0.4568
lp_alpha_5_rev	0.2668	0.2107	0.2443	0.2007	0.4564
lp_alpha_6_rev	0.3443	0.3082	0.3143	0.2989	0.4629
lp_beta_1_rev	0.1939	0.1107	0.1839	0.1011	0.4864
lp_beta_2_rev	0.2289	0.1779	0.2164	0.1757	0.4339
lp_beta_3_rev	0.3264	0.2889	0.3100	0.2821	0.4632
lp_beta_4_rev	0.1939	0.1104	0.1832	0.1011	0.4875
lp_beta_5_rev	0.2282	0.1786	0.2150	0.1757	0.4339
lp_beta_6_rev	0.3261	0.2893	0.3096	0.2825	0.4632
lp_gamma_1_rev	0.2189	0.1300	0.1868	0.1225	0.4193
lp_gamma_2_rev	0.2436	0.1986	0.2086	0.1961	0.4054
lp_gamma_3_rev	0.3311	0.3032	0.3032	0.2950	0.4443
lp_gamma_4_rev	0.2186	0.1300	0.1854	0.1225	0.4193
lp_gamma_5_rev	0.2432	0.1986	0.2089	0.1961	0.4046
lp_gamma_6_rev	0.3307	0.3032	0.3032	0.2950	0.4446

Table B.8: FTP-data — large packet heuristics

B.3 FTP-ctrl

	POP3	FTP-data	Telnet	SMTP	HTTP
pkt_count	0.3614	0.3239	0.2657	0.5129	0.3304
duration	0.2686	0.4568	0.3375	0.3439	0.3271
nonempty_count	0.3171	0.3468	0.2775	0.4386	0.2843
pkt_byte_count	0.4025	0.1771	0.2636	0.2625	0.3000
payload_byte_count	0.3736	0.1464	0.2786	0.2171	0.2579
mean_delay	0.2846	0.2925	0.5118	0.3439	0.3436
mean_pkt_len	0.5411	0.0593	0.3629	0.1357	0.0807
mean_payload_len	0.5939	0.0582	0.3750	0.1325	0.0746
mean_nonempty_payload_len	0.5446	0.0482	0.3721	0.1229	0.0189
dir_data	0.5168	0.1668	0.3861	0.1382	0.4771
mean_pkt_datarate	0.2932	0.1111	0.4843	0.2514	0.1979
mean_payload_datarate	0.3400	0.0950	0.4689	0.2211	0.1550
flag_urg	0.5200	0.5200	0.5204	0.5193	0.5257
flag_ack	0.5296	0.4457	0.2350	0.4582	0.3404
flag_psh	0.2604	0.0964	0.4643	0.2650	0.1018
flag_rst	0.4189	0.3850	0.3961	0.4639	0.4757
flag_syn	0.2932	0.4646	0.2139	0.4150	0.2800
flag_fin	0.2732	0.5357	0.3050	0.3596	0.3189

Table B.9: FTP-ctrl — per-flow metrics

	POP3	FTP-data	Telnet	SMTP	HTTP
pkt_count_fwd	0.3279	0.3361	0.2636	0.5146	0.3150
pkt_byte_count_fwd	0.2889	0.3193	0.2982	0.1493	0.3850
payload_byte_count_fwd	0.1607	0.1364	0.4171	0.1157	0.1486
nonempty_count_fwd	0.3129	0.1550	0.2957	0.4739	0.1414
mean_delay_fwd	0.2846	0.2989	0.5121	0.3421	0.3461
mean_pkt_len_fwd	0.2168	0.5571	0.2525	0.0821	0.0764
mean_payload_len_fwd	0.1896	0.1182	0.1629	0.0832	0.0764
mean_nonempty_payload_len_fwd	0.1714	0.1193	0.1404	0.0857	0.0100
flag_urg_fwd	0.5243	0.5307	0.5164	0.5143	0.5146
flag_ack_fwd	0.4393	0.1821	0.2193	0.4582	0.3364
flag_psh_fwd	0.4614	0.1075	0.4900	0.4439	0.1046
flag_rst_fwd	0.4118	0.3911	0.3900	0.4568	0.4721
flag_syn_fwd	0.2839	0.5368	0.2139	0.4243	0.2686
flag_fin_fwd	0.2754	0.5246	0.3004	0.3818	0.3443
pkt_count_rev	0.3736	0.2893	0.2768	0.4486	0.3461
pkt_byte_count_rev	0.4586	0.1921	0.2639	0.3382	0.3079
payload_byte_count_rev	0.4532	0.1968	0.2807	0.3882	0.2857
nonempty_count_rev	0.3282	0.3439	0.2893	0.2121	0.3329
mean_delay_rev	0.2861	0.2700	0.4907	0.3471	0.3218
mean_pkt_len_rev	0.5339	0.1304	0.3471	0.3889	0.1536
mean_payload_len_rev	0.5739	0.1536	0.3464	0.3979	0.1479
mean_nonempty_payload_len_rev	0.5225	0.1282	0.3521	0.5746	0.0629
flag_urg_rev	0.5286	0.5264	0.5179	0.5271	0.5157
flag_ack_rev	0.5246	0.2611	0.5218	0.5229	0.5154
flag_psh_rev	0.2254	0.1061	0.5096	0.1768	0.1368
flag_rst_rev	0.5186	0.5164	0.5189	0.5057	0.5186
flag_syn_rev	0.3136	0.4289	0.2200	0.4146	0.2957
flag_fin_rev	0.2732	0.4904	0.3136	0.3457	0.3100

Table B.10: FTP-ctrl — per-half-flow metrics

	POP3	FTP-data	Telnet	SMTP	HTTP
sp_alpha_1_fwd	0.3811	0.1750	0.4479	0.2693	0.1700
sp_alpha_2_fwd	0.4657	0.1121	0.4896	0.4664	0.0843
sp_alpha_3_fwd	0.5082	0.2964	0.5054	0.4700	0.2564
sp_beta_1_fwd	0.1704	0.0800	0.2671	0.1500	0.0671
sp_beta_2_fwd	0.4139	0.0543	0.5307	0.1743	0.0375
sp_beta_3_fwd	0.4821	0.0514	0.5086	0.1404	0.0468
sp_gamma_1_fwd	0.1921	0.1732	0.4000	0.2586	0.1646
sp_gamma_2_fwd	0.4296	0.1068	0.3846	0.2889	0.0786
sp_gamma_3_fwd	0.4618	0.0989	0.3793	0.2579	0.0707
sp_delta_1_fwd	0.3214	0.1729	0.4354	0.2825	0.1632
sp_delta_2_fwd	0.5236	0.1086	0.4032	0.4382	0.0825
sp_delta_3_fwd	0.4986	0.0918	0.4421	0.4825	0.0664
sp_alpha_1_rev	0.4350	0.4321	0.2393	0.5296	0.4243
sp_alpha_2_rev	0.4211	0.1243	0.5236	0.4839	0.0943
sp_alpha_3_rev	0.4296	0.2154	0.4768	0.4254	0.1625
sp_beta_1_rev	0.3943	0.1543	0.2314	0.4164	0.1504
sp_beta_2_rev	0.4732	0.0643	0.4300	0.4918	0.0421
sp_beta_3_rev	0.4346	0.0432	0.2818	0.4018	0.0600
sp_gamma_1_rev	0.4518	0.4111	0.2232	0.5425	0.4011
sp_gamma_2_rev	0.4643	0.1043	0.5314	0.4639	0.0711
sp_gamma_3_rev	0.3554	0.0729	0.4275	0.4061	0.0614
sp_delta_1_rev	0.4489	0.4164	0.2350	0.4732	0.4014
sp_delta_2_rev	0.3789	0.1179	0.4554	0.3118	0.0743
sp_delta_3_rev	0.3757	0.0739	0.4793	0.3321	0.0621

Table B.11: FTP-ctrl — small packet heuristics

	POP3	FTP-data	Telnet	SMTP	HTTP
lp_alpha_1_fwd	0.5150	0.4532	0.5132	0.1914	0.4807
lp_alpha_2_fwd	0.5186	0.4679	0.5236	0.2921	0.5211
lp_alpha_3_fwd	0.5171	0.4807	0.5143	0.3900	0.5229
lp_alpha_4_fwd	0.5200	0.4354	0.5125	0.1696	0.4564
lp_alpha_5_fwd	0.5214	0.4586	0.5239	0.2821	0.5250
lp_alpha_6_fwd	0.5171	0.4754	0.5175	0.3861	0.5279
lp_beta_1_fwd	0.5082	0.4325	0.4807	0.0736	0.0418
lp_beta_2_fwd	0.5243	0.4525	0.5296	0.1543	0.4750
lp_beta_3_fwd	0.5286	0.4707	0.5114	0.3136	0.5125
lp_beta_4_fwd	0.5232	0.4325	0.4804	0.0736	0.0418
lp_beta_5_fwd	0.5264	0.4521	0.5204	0.1546	0.4750
lp_beta_6_fwd	0.5414	0.4707	0.5150	0.3136	0.5236
lp_gamma_1_fwd	0.5254	0.4343	0.5107	0.1546	0.4336
lp_gamma_2_fwd	0.5207	0.4568	0.5261	0.2743	0.5196
lp_gamma_3_fwd	0.5193	0.4732	0.5211	0.3836	0.5171
lp_gamma_4_fwd	0.5254	0.4343	0.5193	0.1546	0.4336
lp_gamma_5_fwd	0.5250	0.4568	0.5318	0.2746	0.5086
lp_gamma_6_fwd	0.5257	0.4732	0.5354	0.3836	0.5171
lp_alpha_1_rev	0.3968	0.1546	0.3179	0.4886	0.2143
lp_alpha_2_rev	0.4493	0.2196	0.4750	0.5021	0.2768
lp_alpha_3_rev	0.4661	0.3129	0.5107	0.5171	0.3557
lp_alpha_4_rev	0.3932	0.1464	0.2875	0.4875	0.2064
lp_alpha_5_rev	0.4479	0.2107	0.4686	0.4929	0.2739
lp_alpha_6_rev	0.4657	0.3082	0.5086	0.5011	0.3546
lp_beta_1_rev	0.4646	0.1107	0.2950	0.3500	0.0686
lp_beta_2_rev	0.4504	0.1779	0.3725	0.4668	0.2218
lp_beta_3_rev	0.4539	0.2889	0.4396	0.5239	0.3193
lp_beta_4_rev	0.4636	0.1104	0.2946	0.3500	0.0686
lp_beta_5_rev	0.4507	0.1786	0.3714	0.4668	0.2211
lp_beta_6_rev	0.4539	0.2893	0.4407	0.4929	0.3193
lp_gamma_1_rev	0.3943	0.1300	0.2771	0.5004	0.1900
lp_gamma_2_rev	0.4461	0.1986	0.4371	0.5018	0.2689
lp_gamma_3_rev	0.4650	0.3032	0.4989	0.4929	0.3518
lp_gamma_4_rev	0.3943	0.1300	0.2775	0.4818	0.1900
lp_gamma_5_rev	0.4461	0.1986	0.4382	0.5143	0.2689
lp_gamma_6_rev	0.4654	0.3032	0.5139	0.4929	0.3518

Table B.12: FTP-ctrl — large packet heuristics

B.4 Telnet

	POP3	FTP-data	FTP-ctrl	SMTP	HTTP
pkt_count	0.2261	0.5107	0.2657	0.2400	0.2382
duration	0.1464	0.3296	0.3375	0.2025	0.2186
nonempty_count	0.2264	0.5268	0.2775	0.2361	0.2307
pkt_byte_count	0.2789	0.3350	0.2636	0.3821	0.4493
payload_byte_count	0.2946	0.3229	0.2786	0.5457	0.5043
mean_delay	0.2382	0.2182	0.5118	0.3061	0.3154
mean_pkt_len	0.3821	0.1318	0.3629	0.3132	0.2400
mean_payload_len	0.3461	0.1361	0.3750	0.3071	0.2421
mean_nonempty_payload_len	0.3600	0.0943	0.3721	0.2889	0.0950
dir_data	0.5582	0.2282	0.3861	0.1764	0.4396
mean_pkt_datarate	0.2907	0.0986	0.4843	0.2282	0.1704
mean_payload_datarate	0.3761	0.0964	0.4689	0.2343	0.1657
flag_urg	0.5221	0.5079	0.5204	0.5146	0.5250
flag_ack	0.1511	0.3757	0.2350	0.1532	0.1643
flag_psh	0.3093	0.0889	0.4643	0.3136	0.0900
flag_rst	0.4829	0.5243	0.3961	0.4357	0.4143
flag_syn	0.1400	0.3411	0.2139	0.1411	0.1589
flag_fin	0.1561	0.3600	0.3050	0.1650	0.2125

Table B.13: Telnet — per-flow metrics

	POP3	FTP-data	FTP-ctrl	SMTP	HTTP
pkt_count_fwd	0.2150	0.5271	0.2636	0.2482	0.2257
pkt_byte_count_fwd	0.2061	0.4771	0.2982	0.3511	0.3696
payload_byte_count_fwd	0.2686	0.1850	0.4171	0.1379	0.1929
nonempty_count_fwd	0.2075	0.1518	0.2957	0.2954	0.1282
mean_delay_fwd	0.2525	0.2543	0.5121	0.3132	0.3225
mean_pkt_len_fwd	0.3604	0.4271	0.2525	0.0968	0.0771
mean_payload_len_fwd	0.1800	0.1239	0.1629	0.1111	0.0743
mean_nonempty_payload_len_fwd	0.1314	0.3068	0.1404	0.1154	0.0314
flag_urg_fwd	0.5179	0.5250	0.5164	0.5236	0.5264
flag_ack_fwd	0.1407	0.4168	0.2193	0.1525	0.1454
flag_psh_fwd	0.4836	0.1200	0.4900	0.4714	0.1218
flag_rst_fwd	0.4814	0.4968	0.3900	0.4339	0.4057
flag_syn_fwd	0.1314	0.3321	0.2139	0.1443	0.1468
flag_fin_fwd	0.1568	0.3496	0.3004	0.1754	0.2496
pkt_count_rev	0.2379	0.4896	0.2768	0.2389	0.2525
pkt_byte_count_rev	0.2868	0.3739	0.2639	0.1779	0.4611
payload_byte_count_rev	0.2964	0.4193	0.2807	0.1496	0.5104
nonempty_count_rev	0.2364	0.5175	0.2893	0.1500	0.2657
mean_delay_rev	0.2086	0.1982	0.4907	0.3021	0.2729
mean_pkt_len_rev	0.3654	0.2439	0.3471	0.2407	0.3032
mean_payload_len_rev	0.3400	0.3318	0.3464	0.2761	0.3100
mean_nonempty_payload_len_rev	0.3546	0.2836	0.3521	0.3779	0.1986
flag_urg_rev	0.5207	0.5214	0.5179	0.5296	0.5186
flag_ack_rev	0.5279	0.2689	0.5218	0.5164	0.5200
flag_psh_rev	0.2500	0.1179	0.5096	0.2025	0.1457
flag_rst_rev	0.5211	0.4850	0.5189	0.5132	0.5168
flag_syn_rev	0.1457	0.3768	0.2200	0.1375	0.1754
flag_fin_rev	0.1582	0.3889	0.3136	0.1632	0.2089

Table B.14: Telnet — per-half-flow metrics

	POP3	FTP-data	FTP-ctrl	SMTP	HTTP
sp_alpha_1_fwd	0.4279	0.1300	0.4479	0.2211	0.1246
sp_alpha_2_fwd	0.4561	0.1304	0.4896	0.5161	0.1100
sp_alpha_3_fwd	0.5211	0.2568	0.5054	0.4689	0.1832
sp_beta_1_fwd	0.4339	0.0621	0.2671	0.0886	0.0571
sp_beta_2_fwd	0.4171	0.0632	0.5307	0.1864	0.0446
sp_beta_3_fwd	0.4657	0.0550	0.5086	0.1500	0.0504
sp_gamma_1_fwd	0.3061	0.1379	0.4000	0.1754	0.1229
sp_gamma_2_fwd	0.3329	0.1436	0.3846	0.4068	0.1096
sp_gamma_3_fwd	0.3457	0.1414	0.3793	0.4175	0.0993
sp_delta_1_fwd	0.3950	0.1468	0.4354	0.2654	0.1296
sp_delta_2_fwd	0.3975	0.1607	0.4032	0.5414	0.1154
sp_delta_3_fwd	0.4029	0.1139	0.4421	0.3996	0.0871
sp_alpha_1_rev	0.3129	0.1704	0.2393	0.2371	0.1593
sp_alpha_2_rev	0.4425	0.1368	0.5236	0.5032	0.1082
sp_alpha_3_rev	0.3886	0.1564	0.4768	0.3864	0.1139
sp_beta_1_rev	0.3696	0.1146	0.2314	0.2743	0.1075
sp_beta_2_rev	0.4046	0.1218	0.4300	0.3864	0.0921
sp_beta_3_rev	0.4107	0.1011	0.2818	0.2764	0.1475
sp_gamma_1_rev	0.3161	0.1636	0.2232	0.2575	0.1375
sp_gamma_2_rev	0.4814	0.1429	0.5314	0.4786	0.1000
sp_gamma_3_rev	0.4479	0.1196	0.4275	0.4811	0.0946
sp_delta_1_rev	0.2550	0.1721	0.2350	0.2025	0.1389
sp_delta_2_rev	0.3354	0.1471	0.4554	0.2682	0.0954
sp_delta_3_rev	0.4193	0.0861	0.4793	0.3868	0.0739

Table B.15: Telnet — small packet heuristics

	POP3	FTP-data	FTP-ctrl	SMTP	HTTP
lp_alpha_1_fwd	0.5200	0.4571	0.5132	0.1946	0.4839
lp_alpha_2_fwd	0.5264	0.4686	0.5236	0.2929	0.5193
lp_alpha_3_fwd	0.5189	0.4814	0.5143	0.3907	0.5218
lp_alpha_4_fwd	0.5050	0.4411	0.5125	0.1750	0.4618
lp_alpha_5_fwd	0.5196	0.4593	0.5239	0.2829	0.5075
lp_alpha_6_fwd	0.5293	0.4761	0.5175	0.3868	0.5329
lp_beta_1_fwd	0.4775	0.4482	0.4807	0.0893	0.0493
lp_beta_2_fwd	0.5182	0.4554	0.5296	0.1571	0.4779
lp_beta_3_fwd	0.5207	0.4721	0.5114	0.3154	0.5264
lp_beta_4_fwd	0.4775	0.4479	0.4804	0.0893	0.0493
lp_beta_5_fwd	0.5229	0.4554	0.5204	0.1571	0.4779
lp_beta_6_fwd	0.5139	0.4721	0.5150	0.3154	0.5157
lp_gamma_1_fwd	0.5011	0.4404	0.5107	0.1600	0.4407
lp_gamma_2_fwd	0.5218	0.4582	0.5261	0.2754	0.5329
lp_gamma_3_fwd	0.5132	0.4743	0.5211	0.3843	0.5193
lp_gamma_4_fwd	0.4921	0.4404	0.5193	0.1600	0.4407
lp_gamma_5_fwd	0.5136	0.4579	0.5318	0.2754	0.5250
lp_gamma_6_fwd	0.5261	0.4739	0.5354	0.3843	0.5175
lp_alpha_1_rev	0.4350	0.3246	0.3179	0.3025	0.3711
lp_alpha_2_rev	0.4743	0.2468	0.4750	0.4682	0.3007
lp_alpha_3_rev	0.4714	0.3189	0.5107	0.4882	0.3611
lp_alpha_4_rev	0.4082	0.3389	0.2875	0.2704	0.3932
lp_alpha_5_rev	0.4789	0.2443	0.4686	0.4614	0.3036
lp_alpha_6_rev	0.4718	0.3143	0.5086	0.4871	0.3607
lp_beta_1_rev	0.4146	0.1839	0.2950	0.2007	0.1579
lp_beta_2_rev	0.5425	0.2164	0.3725	0.3214	0.2696
lp_beta_3_rev	0.5004	0.3100	0.4396	0.4307	0.3457
lp_beta_4_rev	0.4129	0.1832	0.2946	0.2007	0.1586
lp_beta_5_rev	0.5511	0.2150	0.3714	0.3218	0.2696
lp_beta_6_rev	0.5004	0.3096	0.4407	0.4304	0.3457
lp_gamma_1_rev	0.5411	0.1868	0.2771	0.2268	0.2529
lp_gamma_2_rev	0.4832	0.2086	0.4371	0.4264	0.2829
lp_gamma_3_rev	0.4711	0.3032	0.4989	0.5000	0.3525
lp_gamma_4_rev	0.5400	0.1854	0.2775	0.2261	0.2525
lp_gamma_5_rev	0.4811	0.2089	0.4382	0.4264	0.2825
lp_gamma_6_rev	0.4693	0.3032	0.5139	0.4843	0.3529

Table B.16: Telnet — large packet heuristics

B.5 SMTP

	POP3	FTP-data	FTP-ctrl	Telnet	HTTP
pkt_count	0.4004	0.2993	0.5129	0.2400	0.3125
duration	0.4225	0.4139	0.3439	0.2025	0.4336
nonempty_count	0.4007	0.3032	0.4386	0.2361	0.2339
pkt_byte_count	0.2568	0.2654	0.2625	0.3821	0.4564
payload_byte_count	0.2261	0.2654	0.2171	0.5457	0.4411
mean_delay	0.4650	0.4296	0.3439	0.3061	0.4964
mean_pkt_len	0.2407	0.2089	0.1357	0.3132	0.3400
mean_payload_len	0.2218	0.2129	0.1325	0.3071	0.3471
mean_nonempty_payload_len	0.2204	0.1621	0.1229	0.2889	0.2389
dir_data	0.0904	0.2475	0.1382	0.1764	0.1721
mean_pkt_datarate	0.3875	0.3011	0.2514	0.2282	0.4261
mean_payload_datarate	0.3336	0.2882	0.2211	0.2343	0.4196
flag_urg	0.5236	0.5293	0.5193	0.5146	0.5250
flag_ack	0.4036	0.4800	0.4582	0.1532	0.3021
flag_psh	0.4518	0.1304	0.2650	0.3136	0.1568
flag_rst	0.4557	0.4221	0.4639	0.4357	0.4861
flag_syn	0.2939	0.3664	0.4150	0.1411	0.2471
flag_fin	0.2957	0.3496	0.3596	0.1650	0.3164

Table B.17: SMTP — per-flow metrics

	POP3	FTP-data	FTP-ctrl	Telnet	HTTP
pkt_count_fwd	0.3607	0.3221	0.5146	0.2482	0.3018
pkt_byte_count_fwd	0.1179	0.4457	0.1493	0.3511	0.2079
payload_byte_count_fwd	0.0804	0.1432	0.1157	0.1379	0.1907
nonempty_count_fwd	0.2261	0.1082	0.4739	0.2954	0.1004
mean_delay_fwd	0.4839	0.4429	0.3421	0.3132	0.4814
mean_pkt_len_fwd	0.0700	0.1493	0.0821	0.0968	0.2121
mean_payload_len_fwd	0.0550	0.1046	0.0832	0.1111	0.2189
mean_nonempty_payload_len_fwd	0.0346	0.1364	0.0857	0.1154	0.4050
flag_urg_fwd	0.5171	0.5171	0.5143	0.5236	0.5271
flag_ack_fwd	0.3450	0.1168	0.4582	0.1525	0.2921
flag_psh_fwd	0.5121	0.1071	0.4439	0.4714	0.1186
flag_rst_fwd	0.4543	0.4339	0.4568	0.4339	0.4832
flag_syn_fwd	0.2868	0.4443	0.4243	0.1443	0.2379
flag_fin_fwd	0.3093	0.5771	0.3818	0.1754	0.3729
pkt_count_rev	0.4257	0.2625	0.4486	0.2389	0.3239
pkt_byte_count_rev	0.4911	0.1375	0.3382	0.1779	0.2082
payload_byte_count_rev	0.5668	0.1343	0.3882	0.1496	0.1579
nonempty_count_rev	0.4314	0.2336	0.2121	0.1500	0.3214
mean_delay_rev	0.4443	0.4125	0.3471	0.3021	0.4450
mean_pkt_len_rev	0.4904	0.1046	0.3889	0.2407	0.0786
mean_payload_len_rev	0.5568	0.1043	0.3979	0.2761	0.0779
mean_nonempty_payload_len_rev	0.5961	0.1014	0.5746	0.3779	0.0254
flag_urg_rev	0.5207	0.5286	0.5271	0.5296	0.5164
flag_ack_rev	0.5164	0.2632	0.5229	0.5164	0.5232
flag_psh_rev	0.4082	0.1793	0.1768	0.2025	0.2182
flag_rst_rev	0.5125	0.5132	0.5057	0.5132	0.5111
flag_syn_rev	0.3314	0.3250	0.4146	0.1375	0.2546
flag_fin_rev	0.2882	0.3111	0.3457	0.1632	0.2789

Table B.18: SMTP — per-half-flow metrics

	POP3	FTP-data	FTP-ctrl	Telnet	HTTP
sp_alpha_1_fwd	0.1418	0.4093	0.2693	0.2211	0.4043
sp_alpha_2_fwd	0.4307	0.1254	0.4664	0.5161	0.0989
sp_alpha_3_fwd	0.4711	0.3068	0.4700	0.4689	0.2843
sp_beta_1_fwd	0.0154	0.1250	0.1500	0.0886	0.0996
sp_beta_2_fwd	0.0679	0.1143	0.1743	0.1864	0.0746
sp_beta_3_fwd	0.0732	0.1125	0.1404	0.1500	0.1268
sp_gamma_1_fwd	0.0414	0.4043	0.2586	0.1754	0.3979
sp_gamma_2_fwd	0.1764	0.1661	0.2889	0.4068	0.1068
sp_gamma_3_fwd	0.2061	0.1361	0.2579	0.4175	0.0675
sp_delta_1_fwd	0.1432	0.4025	0.2825	0.2654	0.3971
sp_delta_2_fwd	0.4382	0.1204	0.4382	0.5414	0.0836
sp_delta_3_fwd	0.5100	0.0639	0.4825	0.3996	0.0386
sp_alpha_1_rev	0.4289	0.4386	0.5296	0.2371	0.4307
sp_alpha_2_rev	0.4339	0.1264	0.4839	0.5032	0.1039
sp_alpha_3_rev	0.5350	0.3004	0.4254	0.3864	0.2793
sp_beta_1_rev	0.3304	0.3254	0.4164	0.2743	0.3246
sp_beta_2_rev	0.5054	0.0604	0.4918	0.3864	0.0400
sp_beta_3_rev	0.4157	0.0443	0.4018	0.2764	0.0607
sp_gamma_1_rev	0.4400	0.4339	0.5425	0.2575	0.4246
sp_gamma_2_rev	0.5089	0.0914	0.4639	0.4786	0.0725
sp_gamma_3_rev	0.4664	0.0904	0.4061	0.4811	0.0689
sp_delta_1_rev	0.4239	0.4636	0.4732	0.2025	0.4261
sp_delta_2_rev	0.4107	0.1521	0.3118	0.2682	0.0882
sp_delta_3_rev	0.4471	0.0743	0.3321	0.3868	0.0614

Table B.19: SMTP — small packet heuristics

	POP3	FTP-data	FTP-ctrl	Telnet	HTTP
lp_alpha_1_fwd	0.1907	0.2379	0.1914	0.1946	0.2107
lp_alpha_2_fwd	0.2918	0.3232	0.2921	0.2929	0.2925
lp_alpha_3_fwd	0.3900	0.4093	0.3900	0.3907	0.3900
lp_alpha_4_fwd	0.1686	0.2368	0.1696	0.1750	0.2143
lp_alpha_5_fwd	0.2818	0.3239	0.2821	0.2829	0.2846
lp_alpha_6_fwd	0.3861	0.4111	0.3861	0.3868	0.3864
lp_beta_1_fwd	0.0718	0.2311	0.0736	0.0893	0.1496
lp_beta_2_fwd	0.1532	0.2868	0.1543	0.1571	0.1864
lp_beta_3_fwd	0.3136	0.3436	0.3136	0.3154	0.3204
lp_beta_4_fwd	0.0718	0.2311	0.0736	0.0893	0.1496
lp_beta_5_fwd	0.1532	0.2807	0.1546	0.1571	0.1854
lp_beta_6_fwd	0.3136	0.3439	0.3136	0.3154	0.3204
lp_gamma_1_fwd	0.1525	0.3164	0.1546	0.1600	0.2932
lp_gamma_2_fwd	0.2736	0.3239	0.2743	0.2754	0.2779
lp_gamma_3_fwd	0.3836	0.4107	0.3836	0.3843	0.3839
lp_gamma_4_fwd	0.1525	0.3168	0.1546	0.1600	0.2796
lp_gamma_5_fwd	0.2736	0.3254	0.2746	0.2754	0.2779
lp_gamma_6_fwd	0.3836	0.4107	0.3836	0.3843	0.3839
lp_alpha_1_rev	0.3843	0.1400	0.4886	0.3025	0.2011
lp_alpha_2_rev	0.4421	0.2061	0.5021	0.4682	0.2696
lp_alpha_3_rev	0.4596	0.3014	0.5171	0.4882	0.3493
lp_alpha_4_rev	0.3807	0.1293	0.4875	0.2704	0.1932
lp_alpha_5_rev	0.4407	0.2007	0.4929	0.4614	0.2668
lp_alpha_6_rev	0.4589	0.2989	0.5011	0.4871	0.3479
lp_beta_1_rev	0.3725	0.1011	0.3500	0.2007	0.0636
lp_beta_2_rev	0.4200	0.1757	0.4668	0.3214	0.2125
lp_beta_3_rev	0.4468	0.2821	0.5239	0.4307	0.3129
lp_beta_4_rev	0.3725	0.1011	0.3500	0.2007	0.0636
lp_beta_5_rev	0.4200	0.1757	0.4668	0.3218	0.2125
lp_beta_6_rev	0.4468	0.2825	0.4929	0.4304	0.3129
lp_gamma_1_rev	0.3779	0.1225	0.5004	0.2268	0.1804
lp_gamma_2_rev	0.4382	0.1961	0.5018	0.4264	0.2621
lp_gamma_3_rev	0.4582	0.2950	0.4929	0.5000	0.3457
lp_gamma_4_rev	0.3779	0.1225	0.4818	0.2261	0.1804
lp_gamma_5_rev	0.4382	0.1961	0.5143	0.4264	0.2621
lp_gamma_6_rev	0.4582	0.2950	0.4929	0.4843	0.3457

Table B.20: SMTP — large packet heuristics

B.6 HTTP

	POP3	FTP-data	FTP-ctrl	Telnet	SMTP
pkt_count	0.4929	0.2864	0.3304	0.2382	0.3125
duration	0.4954	0.3921	0.3271	0.2186	0.4336
nonempty_count	0.4264	0.2904	0.2843	0.2307	0.2339
pkt_byte_count	0.3089	0.3050	0.3000	0.4493	0.4564
payload_byte_count	0.2839	0.3068	0.2579	0.5043	0.4411
mean_delay	0.4932	0.4564	0.3436	0.3154	0.4964
mean_pkt_len	0.2243	0.3761	0.0807	0.2400	0.3400
mean_payload_len	0.2096	0.3711	0.0746	0.2421	0.3471
mean_nonempty_payload_len	0.1054	0.3914	0.0189	0.0950	0.2389
dir_data	0.5793	0.1636	0.4771	0.4396	0.1721
mean_pkt_datarate	0.3189	0.3796	0.1979	0.1704	0.4261
mean_payload_datarate	0.2796	0.3711	0.1550	0.1657	0.4196
flag_urg	0.5121	0.5279	0.5257	0.5250	0.5250
flag_ack	0.2843	0.3043	0.3404	0.1643	0.3021
flag_psh	0.1279	0.3864	0.1018	0.0900	0.1568
flag_rst	0.4425	0.4054	0.4757	0.4143	0.4861
flag_syn	0.4025	0.3168	0.2800	0.1589	0.2471
flag_fin	0.5154	0.3561	0.3189	0.2125	0.3164

Table B.21: HTTP — per-flow metrics

	POP3	FTP-data	FTP-ctrl	Telnet	SMTP
pkt_count_fwd	0.4879	0.3075	0.3150	0.2257	0.3018
pkt_byte_count_fwd	0.1725	0.3614	0.3850	0.3696	0.2079
payload_byte_count_fwd	0.0332	0.1021	0.1486	0.1929	0.1907
nonempty_count_fwd	0.1286	0.1764	0.1414	0.1282	0.1004
mean_delay_fwd	0.5275	0.4732	0.3461	0.3225	0.4814
mean_pkt_len_fwd	0.0382	0.1586	0.0764	0.0771	0.2121
mean_payload_len_fwd	0.0229	0.1054	0.0764	0.0743	0.2189
mean_nonempty_payload_len_fwd	0.0075	0.0836	0.0100	0.0314	0.4050
flag_urg_fwd	0.5186	0.5257	0.5146	0.5264	0.5271
flag_ack_fwd	0.2864	0.1229	0.3364	0.1454	0.2921
flag_psh_fwd	0.1064	0.2064	0.1046	0.1218	0.1186
flag_rst_fwd	0.4339	0.4089	0.4721	0.4057	0.4832
flag_syn_fwd	0.4436	0.3150	0.2686	0.1468	0.2379
flag_fin_fwd	0.5350	0.3768	0.3443	0.2496	0.3729
pkt_count_rev	0.4929	0.2850	0.3461	0.2525	0.3239
pkt_byte_count_rev	0.3214	0.3536	0.3079	0.4611	0.2082
payload_byte_count_rev	0.3039	0.3954	0.2857	0.5104	0.1579
nonempty_count_rev	0.5100	0.3446	0.3329	0.2657	0.3214
mean_delay_rev	0.5375	0.4693	0.3218	0.2729	0.4450
mean_pkt_len_rev	0.2582	0.4271	0.1536	0.3032	0.0786
mean_payload_len_rev	0.2525	0.5496	0.1479	0.3100	0.0779
mean_nonempty_payload_len_rev	0.1879	0.5275	0.0629	0.1986	0.0254
flag_urg_rev	0.5200	0.5250	0.5157	0.5186	0.5164
flag_ack_rev	0.5161	0.2725	0.5154	0.5200	0.5232
flag_psh_rev	0.1557	0.4739	0.1368	0.1457	0.2182
flag_rst_rev	0.5243	0.4989	0.5186	0.5168	0.5111
flag_syn_rev	0.3800	0.3204	0.2957	0.1754	0.2546
flag_fin_rev	0.3854	0.3361	0.3100	0.2089	0.2789

Table B.22: HTTP — per-half-flow metrics

	POP3	FTP-data	FTP-ctrl	Telnet	SMTP
sp_alpha_1_fwd	0.0321	0.5186	0.1700	0.1246	0.4043
sp_alpha_2_fwd	0.0307	0.4789	0.0843	0.1100	0.0989
sp_alpha_3_fwd	0.2071	0.4786	0.2564	0.1832	0.2843
sp_beta_1_fwd	0.0096	0.5071	0.0671	0.0571	0.0996
sp_beta_2_fwd	0.0050	0.4814	0.0375	0.0446	0.0746
sp_beta_3_fwd	0.0293	0.5171	0.0468	0.0504	0.1268
sp_gamma_1_fwd	0.0304	0.5029	0.1646	0.1229	0.3979
sp_gamma_2_fwd	0.0261	0.4793	0.0786	0.1096	0.1068
sp_gamma_3_fwd	0.0264	0.4739	0.0707	0.0993	0.0675
sp_delta_1_fwd	0.0446	0.5118	0.1632	0.1296	0.3971
sp_delta_2_fwd	0.0404	0.4793	0.0825	0.1154	0.0836
sp_delta_3_fwd	0.0225	0.4739	0.0664	0.0871	0.0386
sp_alpha_1_rev	0.3596	0.5032	0.4243	0.1593	0.4307
sp_alpha_2_rev	0.0725	0.4796	0.0943	0.1082	0.1039
sp_alpha_3_rev	0.2825	0.4829	0.1625	0.1139	0.2793
sp_beta_1_rev	0.1121	0.5129	0.1504	0.1075	0.3246
sp_beta_2_rev	0.0668	0.4907	0.0421	0.0921	0.0400
sp_beta_3_rev	0.1457	0.4454	0.0600	0.1475	0.0607
sp_gamma_1_rev	0.3557	0.4964	0.4011	0.1375	0.4246
sp_gamma_2_rev	0.0700	0.4775	0.0711	0.1000	0.0725
sp_gamma_3_rev	0.0864	0.5093	0.0614	0.0946	0.0689
sp_delta_1_rev	0.3529	0.4857	0.4014	0.1389	0.4261
sp_delta_2_rev	0.0532	0.4743	0.0743	0.0954	0.0882
sp_delta_3_rev	0.0475	0.4989	0.0621	0.0739	0.0614

Table B.23: HTTP — small packet heuristics

	POP3	FTP-data	FTP-ctrl	Telnet	SMTP
lp_alpha_1_fwd	0.4804	0.4732	0.4807	0.4839	0.2107
lp_alpha_2_fwd	0.5179	0.4682	0.5211	0.5193	0.2925
lp_alpha_3_fwd	0.5171	0.4807	0.5229	0.5218	0.3900
lp_alpha_4_fwd	0.4561	0.4796	0.4564	0.4618	0.2143
lp_alpha_5_fwd	0.5182	0.4607	0.5250	0.5075	0.2846
lp_alpha_6_fwd	0.5243	0.4757	0.5279	0.5329	0.3864
lp_beta_1_fwd	0.0404	0.1257	0.0418	0.0493	0.1496
lp_beta_2_fwd	0.4746	0.4775	0.4750	0.4779	0.1864
lp_beta_3_fwd	0.5111	0.4771	0.5125	0.5264	0.3204
lp_beta_4_fwd	0.0404	0.1257	0.0418	0.0493	0.1496
lp_beta_5_fwd	0.4746	0.4779	0.4750	0.4779	0.1854
lp_beta_6_fwd	0.5004	0.4771	0.5236	0.5157	0.3204
lp_gamma_1_fwd	0.4329	0.4993	0.4336	0.4407	0.2932
lp_gamma_2_fwd	0.5221	0.4607	0.5196	0.5329	0.2779
lp_gamma_3_fwd	0.5271	0.4736	0.5171	0.5193	0.3839
lp_gamma_4_fwd	0.4329	0.4993	0.4336	0.4407	0.2796
lp_gamma_5_fwd	0.5111	0.4607	0.5086	0.5250	0.2779
lp_gamma_6_fwd	0.5200	0.4736	0.5171	0.5175	0.3839
lp_alpha_1_rev	0.3221	0.4800	0.2143	0.3711	0.2011
lp_alpha_2_rev	0.3275	0.4671	0.2768	0.3007	0.2696
lp_alpha_3_rev	0.3900	0.4682	0.3557	0.3611	0.3493
lp_alpha_4_rev	0.3193	0.4568	0.2064	0.3932	0.1932
lp_alpha_5_rev	0.3261	0.4564	0.2739	0.3036	0.2668
lp_alpha_6_rev	0.3889	0.4629	0.3546	0.3607	0.3479
lp_beta_1_rev	0.1471	0.4864	0.0686	0.1579	0.0636
lp_beta_2_rev	0.2771	0.4339	0.2218	0.2696	0.2125
lp_beta_3_rev	0.3593	0.4632	0.3193	0.3457	0.3129
lp_beta_4_rev	0.1464	0.4875	0.0686	0.1586	0.0636
lp_beta_5_rev	0.2768	0.4339	0.2211	0.2696	0.2125
lp_beta_6_rev	0.3611	0.4632	0.3193	0.3457	0.3129
lp_gamma_1_rev	0.2779	0.4193	0.1900	0.2529	0.1804
lp_gamma_2_rev	0.3168	0.4054	0.2689	0.2829	0.2621
lp_gamma_3_rev	0.3843	0.4443	0.3518	0.3525	0.3457
lp_gamma_4_rev	0.2786	0.4193	0.1900	0.2525	0.1804
lp_gamma_5_rev	0.3171	0.4046	0.2689	0.2825	0.2621
lp_gamma_6_rev	0.3839	0.4446	0.3518	0.3529	0.3457

Table B.24: HTTP — large packet heuristics