Queen Mary University of London

# Discriminators for use in flow-based classification

Moore, Andrew; Zuev, Denis; Crogan, Michael

ISSN 1470-5559

# Discriminators for use in flow-based classification

Andrew Moore, Denis Zuev and Michael Crogan

Queen Mary
University of London
Department of Computer Science

RR-05-13

updated
August 2005

# Discriminators for use in flow-based classification*

Andrew W. Moore,
Queen Mary, University of London, Department of Computer Science, †

Denis Zuev,
University of Oxford, Mathematical Institute,‡

and

Michael L. Crogan§

August 2004, updated August 2005

**Abstract**

Any assessment of classification techniques requires data. This document describes sets of data intended to aid in the assessment of classification work. A number of data sets are described; each data set consists a number of objects, and each object is described by a group of features (also referred to as discriminators). Leveraged by a quantity of hand-classified data, each object within each data set represents a single flow of TCP packets between client and server. The features for each object consist of the (application-centric) classification derived elsewhere and a number of features derived as input to probabilistic classification techniques. In addition to describing the features, we also provide information allowing interested parties to retrieve these data sets for use in their own work. The data sets contain no site-identifying information; each object is only described by a set of statistics and a class that defines the causal application.

## 1 Introduction

This work describes sets of data that we have provided to the research community in order to allow them to assess their classification techniques on real data. The intention of this work is to provide the provenance of the discriminators that describe each object as well as to provide background on the formation of the objects and subsequent limitations of this data set.

We created these data sets as training sets to allow assessment of probabilistic classification techniques. The information in the features is derived using packet header information alone, while the classification-class has been derived using a content-based analysis. The content-based classification process is described in [1, 2].

We have used data collected by the high-performance network monitor described in [3]. We use its loss-limited, capture to disk providing timestamps with resolution of better than 35 nanoseconds. The site, **B**, is a research facility host to about 1,000 users connected to the Internet via a full-duplex Gigabit Ethernet link. Our data is based upon a 24-hour, full-duplex trace of this research facility.

Section 2 describes how we subsample the 24-hour period, creating 10 separate data sets each from a different period of the 24-hour day. Section 3 describes how we filter the data of each period to create the sets of objects (TCP flows) we subsequently characterize. Section 4 describes each of the discriminators (features) that are used to characterize the objects of each data set. Section 5 notes the address from which this data may be retrieved, summarizes the features and limitations of these data sets and notes where future work may take us.

*Contact author: andrew.moore@dcs.qmul.ac.uk

†This work was completed when Andrew Moore was supported by a research fellowship from the the Intel Corporation.
‡This work was completed when Denis Zuev was employed by Intel Research, Cambridge.
§This work was completed when Michael Crogan was employed by Intel Research, Cambridge.

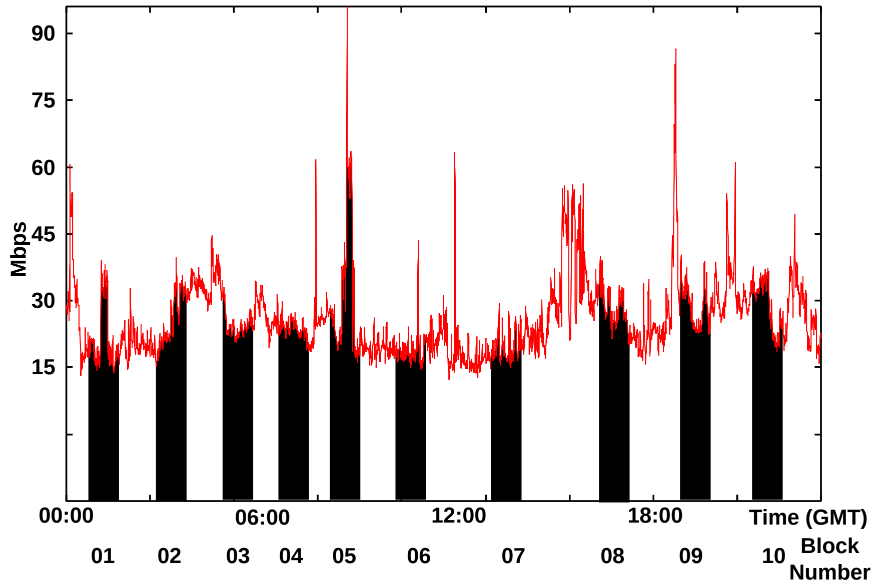


Figure 1: Heuristic illustration of how data blocks were obtained. The line represents the instantaneous bandwidth requirements during the day, while the dark regions represent the data of each data set.

| Data-set | Start-time | End-time | Duration | Flows (Objects) |
|---|---|---|---|---|
| entry01 | 2003-Aug-20 00:34:21 | 2003-Aug-20 01:04:43 | 1821.8 | 24863 |
| entry02 | 2003-Aug-20 01:37:37 | 2003-Aug-20 02:05:54 | 1696.7 | 23801 |
| entry03 | 2003-Aug-20 02:45:19 | 2003-Aug-20 03:14:03 | 1724.1 | 22932 |
| entry04 | 2003-Aug-20 04:03:31 | 2003-Aug-20 04:33:15 | 1784.1 | 22285 |
| entry05 | 2003-Aug-20 04:39:10 | 2003-Aug-20 05:09:05 | 1794.9 | 21648 |
| entry06 | 2003-Aug-20 06:07:28 | 2003-Aug-20 06:35:06 | 1658.5 | 19384 |
| entry07 | 2003-Aug-20 09:42:17 | 2003-Aug-20 10:11:16 | 1739.2 | 55835 |
| entry08 | 2003-Aug-20 11:52:40 | 2003-Aug-20 12:20:26 | 1665.9 | 55494 |
| entry09 | 2003-Aug-20 13:45:37 | 2003-Aug-20 14:13:21 | 1664.5 | 66248 |
| entry10 | 2003-Aug-20 14:55:44 | 2003-Aug-20 15:22:37 | 1613.4 | 65036 |

Table 1: Broad statistics of each data set.

## 2 Data sets

In order to construct the sets of flows, the day trace was split into ten blocks of approximately 1680 seconds (28 minutes) each. In order to provide a wider sample of mixing across the day, the start of each sample was selected randomly (uniformly distributed over the whole day trace). Figure 1 illustrates heuristically our technique. It can be seen from Table 1 that there are a different number of flows in each data block, due to a variable density of traffic during each constant period. Since time statistics of flows are present in the analysis, we consider it to be important to keep a fixed time window when selecting flows.

Each data set represents a period of time taken from within the day. Table 1 provides a list of the data sets along with information about their duration and the number of flows present in each data set. While each of the data sets represent approximately the same period of time, the number of objects per data set fluctuates as a result of the variation in activity throughout the course of the day.

While the start times of the ten data sets were selected from a random uniform distribution, there is a clear bias toward the first 16 hours of the day. As noted in *Future work* the discriminator-based characterization is planned for all flows over the whole 24-hour period thereby removing this bias from the data.

# 3 Flow Definition

Each data set is represented as a text file, and consists of multiple lines. Each line represents an object (flow). In the Internet a flow may be defined as one or more packets traveling between two computer addresses using a particular protocol (e.g., TCP, UDP, ICMP)—and where appropriate a particular pair of ports (defined for each end of the flow). This tuple of information ($\text{host}_{\text{src}}, \text{host}_{\text{dest}}, \text{port}_{\text{src}}, \text{port}_{\text{dest}}$, protocol) is present in every packet.

Source and destination elements in the tuple will be reversed for packets traveling in the opposite direction. In this way a stream of packets may be either half- or full-duplex.

In order to simplify our definitions, we elected to concentrate upon the TCP protocol and TCP flows only, UDP data is planned in future work. TCP is a stateful protocol and its flows have a well-defined beginning and end, and subsequently do not lend themselves to the unclear start-end definitions that may plague a flow of data consisting of individual datagrams alone (such as UDP).

In the simplest case, a complete flow is well defined when both a complete flow setup and tear-down are observed. The complexity in any flow definition occurs when the setup is incomplete or the tear-down is abnormal. A set of rules is embodied into *netdude* [4] that implements the TCP state engine in a sufficiently robust fashion even in the face of packet loss (as might occur in a network monitoring system).

We use netdude to create a collection of complete TCP flows.

Complete TCP flows have a number of desirable properties. For example, using complete flows allows us to differentiate client from server, and this allows us to reliably identify the client and server ports. Additionally, complete TCP flows allow us to compute nearly all the discriminators provided in each data set.

# 4 Discriminators

This data set is intended to provide a wide variety of features to characterise flows. This includes simple statistics about packet length and inter-packet timings, and information derived from the transport protocol (TCP): such as SYN and ACK counts. This information is provided based on all packets (both directions) and on each direction individually (server → client and client → server).

Many packet statistics are derived directly by counting packets, and packet header-sizes. A significant number of features (such as estimates of round-trip time, size of TCP segments, and the total number of retransmissions) are derived from the TCP headers — we use *tcptrace*[5] for this information.

We describe each flow using three modes:

**idle** : no packets between client and server for greater-than or equal-to two seconds,

**interactive** : data packets moving in both directions, and

**bulk** : data packets in one direction and only acknowledgments in the other.

We provide features of the flow duration, the time a flow spends in the *bulk* mode, and the time spent by the flow in the *idle* mode.

The effective bandwidth utilisation is a computation of entropy that provides an insight into the activity of a flow. Such an entropy measure may be computed over a number of different time-scales as a result the values provided are single points in a continuum.

Finally, from the inter-arrival time of packets (in both directions and each direction individually) we also provide a ranked list of the ten frequency components that contribute the most to the Fourier transform and the effective bandwidth utilisation [1]

[1]It may be argued that frequency components based upon *binning* would be more useful than a ranked list — we plan to add that in the future.

Table 2: Discriminators and Definitions

| Number | Short | Long |
|---|---|---|
| 1 | Server Port | Port Number at server; we can establish server and client ports as we limit ourselves to flows for which we see the initial connection set-up. |
| 2 | Client Port | Port Number at client |
| 3 | min_IAT | Minimum packet inter-arrival time for all packets of the flow (considering both directions). |
| 4 | q1_IAT | First quartile inter-arrival time |
| 5 | med_IAT | Median inter-arrival time |
| 6 | mean_IAT | Mean inter-arrival time |
| 7 | q3_IAT | Third quartile packet inter-arrival time |
| 8 | max_IAT | Maximum packet inter-arrival time |
| 9 | var_IAT | Variance in packet inter-arrival time |
| 10 | min_data_wire | Minimum of bytes in (Ethernet) packet, using the size of the packet *on the wire.* |
| 11 | q1_data_wire | First quartile of bytes in (Ethernet) packet |
| 12 | med_data_wire | Median of bytes in (Ethernet) packet |
| 13 | mean_data_wire | Mean of bytes in (Ethernet) packet |
| 14 | q3_data_wire | Third quartile of bytes in (Ethernet) packet |
| 15 | max_data_wire | Maximum of bytes in (Ethernet) packet |
| 16 | var_data_wire | Variance of bytes in (Ethernet) packet |
| 17 | min_data_ip | Minimum of total bytes in IP packet, using the size of payload declared by the IP packet |
| 18 | q1_data_ip | First quartile of total bytes in IP packet |
| 19 | med_data_ip | Median of total bytes in IP packet |
| 20 | mean_data_ip | Mean of total bytes in IP packet |
| 21 | q3_data_ip | Third quartile of total bytes in IP packet |
| 22 | max_data_ip | Maximum of total bytes in IP packet |
| 23 | var_data_ip | Variance of total bytes in IP packet |
| 24 | min_data_control | Minimum of control bytes in packet, size of the (IP/TCP) packet header |
| 25 | q1_data_control | First quartile of control bytes in packet |
| 26 | med_data_control | Median of control bytes in packet |
| 27 | mean_data_control | Mean of control bytes in packet |
| 28 | q3_data_control | Third quartile of control bytes in packet |
| 29 | max_data_control | Maximum of control bytes in packet |
| 30 | var_data_control | Variance of control bytes packet |
| 31 | total_packets_$a\ b$ | The total number of packets seen (client→server). |
| 32 | total_packets_$b\ a$ | ” (server→client) |

Continued on next page

| Number | Short | Long |
| --- | --- | --- |
| 33 | ack_pkts_sent_$a$ $b$ | The total number of ack packets seen (TCP segments seen with the ACK bit set) (client→server). |
| 34 | ack_pkts_sent_$b$ $a$ | ” (server→client) |
| 35 | pure_acks_sent_$a$ $b$ | The total number of ack packets seen that were not piggy-backed with data (just the TCP header and no TCP data payload) and did not have any of the SYN/FIN/RST flags set (client→server) |
| 36 | pure_acks_sent_$b$ $a$ | ” (server→client) |
| 37 | sack_pkts_sent_$a$ $b$ | The total number of ack packets seen carrying TCP SACK [6] blocks (client→server) |
| 38 | sack_pkts_sent_$b$ $a$ | ” (server→client) |
| 39 | dsack_pkts_sent_$a$ $b$ | The total number of sack packets seen that carried duplicate SACK (D-SACK) [7] blocks. (client→server) |
| 40 | dsack_pkts_sent_$b$ $a$ | ” (server→client) |
| 41 | max_sack_blks/ack_$a$ $b$ | The maximum number of sack blocks seen in any sack packet. (client→server) |
| 42 | max_sack_blks/ack_$b$ $a$ | ” (server→client) |
| 43 | unique_bytes_sent_$a$ $b$ | The number of unique bytes sent, i.e., the total bytes of data sent excluding retransmitted bytes and any bytes sent doing window probing. (client→server) |
| 44 | unique_bytes_sent_$b$ $a$ | ” (server→client) |
| 45 | actual_data_pkts_$a$ $b$ | The count of all the packets with at least a byte of TCP data payload. (client→server) |
| 46 | actual_data_pkts_$b$ $a$ | ” (server→client) |
| 47 | actual_data_bytes_$a$ $b$ | The total bytes of data seen. Note that this includes bytes from retransmissions / window probe packets if any. (client→server) |
| 48 | actual_data_bytes_$b$ $a$ | ” (server→client) |
| 49 | rexmt_data_pkts_$a$ $b$ | The count of all the packets found to be retransmissions. (client→server) |
| 50 | rexmt_data_pkts_$b$ $a$ | ” (server→client) |
| 51 | rexmt_data_bytes_$a$ $b$ | The total bytes of data found in the retransmitted packets. (client→server) |
| 52 | rexmt_data_bytes_$b$ $a$ | ” (server→client) |
| 53 | zwnd_probe_pkts_$a$ $b$ | The count of all the window probe packets seen. (Window probe packets are typically sent by a sender when the receiver last advertised a zero receive window, to see if the window has opened up now). (client→server) |
| 54 | zwnd_probe_pkts_$b$ $a$ | ” (server→client) |
| 55 | zwnd_probe_bytes_$a$ $b$ | The total bytes of data sent in the window probe packets. (client→server) |
| 56 | zwnd_probe_bytes_$b$ $a$ | ” (server→client) |
| 57 | outoforder_pkts_$a$ $b$ | The count of all the packets that were seen to arrive out of order. (client→server) |

Continued on next page

| Number | Short | Long |
|---|---|---|
| 58 | outoforder_pkts_*b a* | ” (server→client) |
| 59 | pushed_data_pkts_*a b* | The count of all the packets seen with the PUSH bit set in the TCP header. (client→server) |
| 60 | pushed_data_pkts_*b a* | ” (server→client) |
| 61 | SYN_pkts_sent_*a b* | The count of all the packets seen with the SYN bits set in the TCP header respectively (client→server) |
| 62 | FIN_pkts_sent_*a b* | The count of all the packets seen with the FIN bits set in the TCP header respectively (client→server) |
| 63 | SYN_pkts_sent_*b a* | The count of all the packets seen with the SYN bits set in the TCP header respectively (server→client) |
| 64 | FIN_pkts_sent_*b a* | The count of all the packets seen with the FIN bits set in the TCP header respectively (server→client) |
| 65 | req_1323_ws_*a b* | If the endpoint requested Window Scaling/Time Stamp options as specified in RFC 1323[8] a ‘Y’ is printed on the respective field. If the option was not requested, an ‘N’ is printed. For example, an “N/Y” in this field means that the window-scaling option was not specified, while the Time-stamp option was specified in the SYN segment. (client→server) |
| 66 | req_1323_ts_*a b* | . . . |
| 67 | req_1323_ws_*b a* | If the endpoint requested Window Scaling/Time Stamp options as specified in RFC 1323[8] a ‘Y’ is printed on the respective field. If the option was not requested, an ‘N’ is printed. For example, an “N/Y” in this field means that the window-scaling option was not specified, while the Time-stamp option was specified in the SYN segment. (client→server) |
| 68 | req_1323_ts_*b a* | . . . |
| 69 | adv_wind_scale_*a b* | The window scaling factor used. Again, this field is valid only if the connection was captured fully to include the SYN packets. Since the connection would use window scaling if and only if both sides requested window scaling [8], this field is reset to 0 (even if a window scale was requested in the SYN packet for this direction), if the SYN packet in the reverse direction did not carry the window scale option. (client→server) |
| 70 | adv_wind_scale_*b a* | ” (server→client) |
| 71 | req sack_*a b* | If the end-point sent a SACK permitted option in the SYN packet opening the connection, a ‘Y’ is printed; otherwise ‘N’ is printed. (client→server) |
| 72 | req sack_*b a* | ” (server→client) |
| 73 | sacks_sent_*a b* | The total number of ACK packets seen carrying SACK information. (client→server) |
| 74 | sacks_sent_*b a* | ” (server→client) |

Continued on next page

| Number | Short | Long |
|---|---|---|
| 75 | urgent_data_pkts_$a$ $b$ | The total number of packets with the URG bit turned on in the TCP header. (client→server) |
| 76 | urgent_data_pkts_$b$ $a$ | ” (server→client) |
| 77 | urgent_data_bytes_$a$ $b$ | The total bytes of urgent data sent. This field is calculated by summing the urgent pointer offset values found in packets having the URG bit set in the TCP header. (client→server) |
| 78 | urgent_data_bytes_$b$ $a$ | ” (server→client) |
| 79 | mss_requested_$a$ $b$ | The Maximum Segment Size (MSS) requested as a TCP option in the SYN packet opening the connection. (client→server) |
| 80 | mss_requested_$b$ $a$ | ” (server→client) |
| 81 | max_segm_size_$a$ $b$ | The maximum segment size observed during the lifetime of the connection. (client→server) |
| 82 | max_segm_size_$b$ $a$ | ” (server→client) |
| 83 | min_segm_size_$a$ $b$ | The minimum segment size observed during the lifetime of the connection. (client→server) |
| 84 | min_segm_size_$b$ $a$ | ” (server→client) |
| 85 | avg_segm_size_$a$ $b$ | The average segment size observed during the lifetime of the connection calculated as the value reported in the actual data bytes field divided by the actual data pkts reported. (client→server) |
| 86 | avg_segm_size_$b$ $a$ | ” (server→client) |
| 87 | max_win_adv_$a$ $b$ | The maximum window advertisement seen. If the connection is using window scaling (both sides negotiated window scaling during the opening of the connection), this is the maximum window-scaled advertisement seen in the connection. For a connection using window scaling, both the SYN segments opening the connection have to be captured in the dumpfile for this and the following window statistics to be accurate. (client→server) |
| 88 | max_win_adv_$b$ $a$ | ” (server→client) |
| 89 | min_win_adv_$a$ $b$ | The minimum window advertisement seen. This is the minimum window-scaled advertisement seen if both sides negotiated window scaling. (client→server) |
| 90 | min_win_adv_$b$ $a$ | ” (server→client) |
| 91 | zero_win_adv_$a$ $b$ | The number of times a zero receive window was advertised. (client→server) |
| 92 | zero_win_adv_$b$ $a$ | ” (server→client) |
| 93 | avg_win_adv_$a$ $b$ | The average window advertisement seen, calculated as the sum of all window advertisements divided by the total number of packets seen. If the connection endpoints negotiated window scaling, this average is calculated as the sum of all window-scaled advertisements divided by the number of window-scaled packets seen. Note that in the window-scaled case, the window advertisements in the SYN packets are excluded since the SYN packets themselves cannot have their window advertisements scaled, as per RFC 1323 [8]. (client→server) |

Continued on next page

| Number | Short | Long |
|---|---|---|
| 94 | avg_win_adv_*b a* | " (server→client) |
| 95 | initial_window-bytes_*a b* | The total number of bytes sent in the initial window i.e., the number of bytes seen in the initial flight of data before receiving the first ack packet from the other endpoint. Note that the ack packet from the other endpoint is the first ack acknowledging some data (the ACKs part of the 3-way handshake do not count), and any retransmitted packets in this stage are excluded. (client→server) |
| 96 | initial_window-bytes_*b a* | " (server→client) |
| 97 | initial_window-packets_*a b* | The total number of segments (packets) sent in the initial window as explained above. (client→server) |
| 98 | initial_window-packets_*b a* | " (server→client) |
| 99 | ttl_stream_length_*a b* | The Theoretical Stream Length. This is calculated as the difference between the sequence numbers of the SYN and FIN packets, giving the length of the data stream seen. Note that this calculation is aware of sequence space wrap-arounds, and is printed only if the connection was complete (both the SYN and FIN packets were seen). (client→server) |
| 100 | ttl_stream_length_*b a* | " (server→client) |
| 101 | missed_data_*a b* | The missed data, calculated as the difference between the ttl stream length and unique bytes sent. If the connection was not complete, this calculation is invalid and an "NA" (Not Available) is printed. (client→server) |
| 102 | missed_data_*b a* | " (server→client) |
| 103 | truncated_data_*a b* | The truncated data, calculated as the total bytes of data truncated during packet capture. For example, with tcpdump, the snaplen option can be set to 64 (with -s option) so that just the headers of the packet (assuming there are no options) are captured, truncating most of the packet data. In an Ethernet with maximum segment size of 1500 bytes, this would amount to truncated data of 1500 64 = 1436bytes for a packet. (client→server) |
| 104 | truncated_data_*b a* | " (server→client) |
| 105 | truncated_packets_*a b* | The total number of packets truncated as explained above. (client→server) |
| 106 | truncated_packets_*b a* | " (server→client) |
| 107 | data_xmit_time_*a b* | Total data transmit time, calculated as the difference between the times of capture of the first and last packets carrying non-zero TCP data payload. (client→server) |
| | | Continued on next page |

| Number | Short | Long |
|---|---|---|
| 108 | data_xmit_time_*b a* | ” (server→client) |
| 109 | idletime_max_*a b* | Maximum idle time, calculated as the maximum time between consecutive packets seen in the direction. (client→server) |
| 110 | idletime_max_*b a* | ” (server→client) |
| 111 | throughput_*a b* | The average throughput calculated as the unique bytes sent divided by the elapsed time i.e., the value reported in the unique bytes sent field divided by the elapsed time (the time difference between the capture of the first and last packets in the direction). (client→server) |
| 112 | throughput_*b a* | ” (server→client) |
| 113 | RTT_samples_*a b* | The total number of Round-Trip Time (RTT) samples found. tcptrace is pretty smart about choosing only valid RTT samples. An RTT sample is found only if an ack packet is received from the other endpoint for a previously transmitted packet such that the acknowledgment value is 1 greater than the last sequence number of the packet. Further, it is required that the packet being acknowledged was not retransmitted, and that no packets that came before it in the sequence space were retransmitted after the packet was transmitted. Note : The former condition invalidates RTT samples due to the retransmission ambiguity problem, and the latter condition invalidates RTT samples since it could be the case that the ack packet could be cumulatively acknowledging the retransmitted packet, and not necessarily ack-ing the packet in question. (client→server) |
| 114 | RTT_samples_*b a* | ” (server→client) |
| 115 | RTT_min_*a b* | The minimum RTT sample seen. (client→server) |
| 116 | RTT_min_*b a* | ” (server→client) |
| 117 | RTT_max_*a b* | The maximum RTT sample seen. (client→server) |
| 118 | RTT_max_*b a* | ” (server→client) |
| 119 | RTT_avg_*a b* | The average value of RTT found, calculated straightforward-ly as the sum of all the RTT values found divided by the total number of RTT samples. (client→server) |
| 120 | RTT_avg_*b a* | ” (server→client) |
| 121 | RTT_stdv_*a b* | The standard deviation of the RTT samples. (client→server) |
| 122 | RTT_stdv_*b a* | ” (server→client) |
| 123 | RTT_from_3WHS_*a b* | The RTT value calculated from the TCP 3-Way Hand-Shake (connection opening) [9], assuming that the SYN packets of the connection were captured. (client→server) |
| | | Continued on next page |

| Number | Short | Long |
|---|---|---|
| 124 | RTT_from_3WHS_b a | ” (server→client) |
| 125 | RTT_full_sz_smpls_a b | The total number of full-size RTT samples, calculated from the RTT samples of full-size segments. Full-size segments are defined to be the segments of the largest size seen in the connection. (client→server) |
| 126 | RTT_full_sz_smpls_b a | ” (server→client) |
| 127 | RTT_full_sz_min_a b | The minimum full-size RTT sample. (client→server) |
| 128 | RTT_full_sz_min_b a | ” (server→client) |
| 129 | RTT_full_sz_max_a b | The maximum full-size RTT sample. (client→server) |
| 130 | RTT_full_sz_max_b a | ” (server→client) |
| 131 | RTT_full_sz_avg_a b | The average full-size RTT sample. (client→server) |
| 132 | RTT_full_sz_avg_b a | ” (server→client) |
| 133 | RTT_full_sz_stdev_a b | The standard deviation of full-size RTT samples. (client→server) |
| 134 | RTT_full_sz_stdev_b a | ” (server→client) |
| 135 | post-loss_acks_a b | The total number of ack packets received after losses were detected and a retransmission occurred. More precisely, a post-loss ack is found to occur when an ack packet acknowledges a packet sent (acknowledgment value in the ack pkt is 1 greater than the packet's last sequence number), and at least one packet occurring before the packet acknowledged, was retransmitted later. In other words, the ack packet is received after we observed a (perceived) loss event and are recovering from it. (client→server) |
| 136 | post-loss_acks_b a | ” (server→client) |
| 137 | segs_cum_acked_a b | The count of the number of segments that were cumulatively acknowledged and not directly acknowledged. (client→server) |
| 138 | segs_cum_acked_b a | ” (server→client) |
| 139 | duplicate_acks_a b | The total number of duplicate acknowledgments received. (client→server) |
| 140 | duplicate_acks_b a | ” (server→client) |
| 141 | triple_dupacks_a b | The total number of triple duplicate acknowledgments received (three duplicate acknowledgments acknowledging the same segment), a condition commonly used to trigger the fast-retransmit/fast-recovery phase of TCP. (client→server) |
| 142 | triple_dupacks_b a | ” (server→client) |
| 143 | max_#_retrans_a b | The maximum number of retransmissions seen for any segment during the lifetime of the connection. (client→server) |

Continued on next page

| Number | Short | Long |
|---|---|---|
| 144 | max_#_retrans_$b$ $a$ | ” (server→client) |
| 145 | min_retr_time_$a$ $b$ | The minimum time seen between any two (re)transmissions of a segment amongst all the retransmissions seen. (client→server) |
| 146 | min_retr_time_$b$ $a$ | ” (server→client) |
| 147 | max_retr_time_$a$ $b$ | The maximum time seen between any two (re)transmissions of a segment. (client→server) |
| 148 | max_retr_time_$b$ $a$ | ” (server→client) |
| 149 | avg_retr_time_$a$ $b$ | The average time seen between any two (re)transmissions of a segment calculated from all the retransmissions. (client→server) |
| 150 | avg_retr_time_$b$ $a$ | ” (server→client) |
| 151 | sdv_retr_time_$a$ $b$ | The standard deviation of the retransmission-time samples obtained from all the retransmissions. (client→server) |
| 152 | sdv_retr_time_$b$ $a$ | ” (server→client) |
| 153 | min_data_wire_$a$ $b$ | Minimum number of bytes in (Ethernet) packet (client→server) |
| 154 | q1_data_wire_$a$ $b$ | First quartile of bytes in (Ethernet) packet |
| 155 | med_data_wire_$a$ $b$ | Median of bytes in (Ethernet) packet |
| 156 | mean_data_wire_$a$ $b$ | Mean of bytes in (Ethernet) packet |
| 157 | q3_data_wire_$a$ $b$ | Third quartile of bytes in (Ethernet) packet |
| 158 | max_data_wire_$a$ $b$ | Maximum of bytes in (Ethernet) packet |
| 159 | var_data_wire_$a$ $b$ | Variance of bytes in (Ethernet) packet |
| 160 | min_data_ip_$a$ $b$ | Minimum number of total bytes in IP packet |
| 161 | q1_data_ip_$a$ $b$ | First quartile of total bytes in IP packet |
| 162 | med_data_ip_$a$ $b$ | Median of total bytes in IP packet |
| 163 | mean_data_ip_$a$ $b$ | Mean of total bytes in IP packet |
| 164 | q3_data_ip_$a$ $b$ | Third quartile of total bytes in IP packet |
| 165 | max_data_ip_$a$ $b$ | Maximum of total bytes in IP packet |
| 166 | var_data_ip_$a$ $b$ | Variance of total bytes in IP packet |
| 167 | min_data_control_$a$ $b$ | Minimum of control bytes in packet |
| 168 | q1_data_control_$a$ $b$ | First quartile of control bytes in packet |
| 169 | med_data_control_$a$ $b$ | Median of control bytes in packet |
| 170 | mean_data_control_$a$ $b$ | Mean of control bytes in packet |
| 171 | q3_data_control_$a$ $b$ | Third quartile of control bytes in packet |
| 172 | max_data_control_$a$ $b$ | Maximum of control bytes in packet |
| 173 | var_data_control_$a$ $b$ | Variance of control bytes packet |
| 174 | min_data_wire_$b$ $a$ | Minimum number of bytes in (Ethernet) packet (server→client) |
| 175 | q1_data_wire_$b$ $a$ | First quartile of bytes in (Ethernet) packet |
| 176 | med_data_wire_$b$ $a$ | Median of bytes in (Ethernet) packet |
| 177 | mean_data_wire_$b$ $a$ | Mean of bytes in (Ethernet) packet |

Continued on next page

| Number | Short | Long |
|---|---|---|
| 178 | q3_data_wire_$b\,a$ | Third quartile of bytes in (Ethernet) packet |
| 179 | max_data_wire_$b\,a$ | Maximum of bytes in (Ethernet) packet |
| 180 | var_data_wire_$b\,a$ | Variance of bytes in (Ethernet) packet |
| 181 | min_data_ip_$b\,a$ | Minimum number of total bytes in IP packet |
| 182 | q1_data_ip_$b\,a$ | First quartile of total bytes in IP packet |
| 183 | med_data_ip_$b\,a$ | Median of total bytes in IP packet |
| 184 | mean_data_ip_$b\,a$ | Mean of total bytes in IP packet |
| 185 | q3_data_ip_$b\,a$ | Third quartile of total bytes in IP packet |
| 186 | max_data_ip_$b\,a$ | Maximum of total bytes in IP packet |
| 187 | var_data_ip_$b\,a$ | Variance of total bytes in IP packet |
| 188 | min_data_control_$b\,a$ | Minimum of control bytes in packet |
| 189 | q1_data_control_$b\,a$ | First quartile of control bytes in packet |
| 190 | med_data_control_$b\,a$ | Median of control bytes in packet |
| 191 | mean_data_control_$b\,a$ | Mean of control bytes in packet |
| 192 | q3_data_control_$b\,a$ | Third quartile of control bytes in packet |
| 193 | max_data_control_$b\,a$ | Maximum of control bytes in packet |
| 194 | var_data_control_$b\,a$ | Variance of control bytes packet |
| 195 | min_IAT_$a\,b$ | Minimum of packet inter-arrival time (client→server) |
| 196 | q1_IAT_$a\,b$ | First quartile of packet inter-arrival time |
| 197 | med_IAT_$a\,b$ | Median of packet inter-arrival time |
| 198 | mean_IAT_$a\,b$ | Mean of packet inter-arrival time |
| 199 | q3_IAT_$a\,b$ | Third quartile of packet inter-arrival time |
| 200 | max_IAT_$a\,b$ | Maximum of packet inter-arrival time |
| 201 | var_IAT_$a\,b$ | Variance of packet inter-arrival time |
| 202 | min_IAT_$b\,a$ | Minimum of packet inter-arrival time (server→client) |
| 203 | q1_IAT_$b\,a$ | First quartile of packet inter-arrival time |
| 204 | med_IAT_$b\,a$ | Median of packet inter-arrival time |
| 205 | mean_IAT_$b\,a$ | Mean of packet inter-arrival time |
| 206 | q3_IAT_$b\,a$ | Third quartile of packet inter-arrival time |
| 207 | max_IAT_$b\,a$ | Maximum of packet inter-arrival time |
| 208 | var_IAT_$b\,a$ | Variance of packet inter-arrival time |
| 209 | Time_since_last_connection | Time since the last connection between these hosts |
| 210 | No._transitions_bulk/trans | The number of transitions between transaction mode and bulk transfer mode, where bulk transfer mode is defined as the time when there are more than three successive packets in the same direction without any packets carrying data in the other direction |
| 211 | Time_spent_in_bulk | Amount of time spent in bulk transfer mode |
| 212 | Duration | Connection duration |
| 213 | %_bulk | Percent of time spent in bulk transfer |
| 214 | Time_spent_idle | The time spent idle (where idle time is the accumulation of all periods of 2 seconds or greater when no packet was seen in either direction) |

Continued on next page

| Number | Short | Long |
|---|---|---|
| 215 | %_idle | Percent of time spent idle |
| 216 | Effective_Bandwidth | Effective Bandwidth based upon entropy [10] (both directions) |
| 217 | Effective_Bandwidth_*a b* | ” (client→server) |
| 218 | Effective_Bandwidth_*b a* | ” (server→client) |
| 219 | FFT_all | FFT of packet IAT (arctan of the top-ten frequencies ranked by the magnitude of their contribution) (all traffic) (Frequency #1) |
| 220 | FFT_all | ” (Frequency #2) |
| 221 | FFT_all | ” ... |
| 222 | FFT_all | ” ... |
| 223 | FFT_all | ” ... |
| 224 | FFT_all | ” ... |
| 225 | FFT_all | ” ... |
| 226 | FFT_all | ” ... |
| 227 | FFT_all | ” ... |
| 228 | FFT_all | ” (Frequency #10) |
| 229 | FFT_*a b* | FFT of packet IAT (arctan of the top-ten frequencies ranked by the magnitude of their contribution) (client→server) (Frequency #1) |
| 230 | FFT_*a b* | ” (Frequency #2) |
| 231 | FFT_*a b* | ” ... |
| 232 | FFT_*a b* | ” ... |
| 233 | FFT_*a b* | ” ... |
| 234 | FFT_*a b* | ” ... |
| 235 | FFT_*a b* | ” ... |
| 236 | FFT_*a b* | ” ... |
| 237 | FFT_*a b* | ” ... |
| 238 | FFT_*b a* | ” (Frequency #10) |
| 239 | FFT_*b a* | FFT of packet IAT (arctan of the top-ten frequencies ranked by the magnitude of their contribution) (server→client) (Frequency #1) |
| 240 | FFT_*b a* | ” (Frequency #2) |
| 241 | FFT_*b a* | ” ... |
| 242 | FFT_*b a* | ” ... |
| 243 | FFT_*b a* | ” ... |
| 244 | FFT_*b a* | ” ... |
| 245 | FFT_*b a* | ” ... |
| 246 | FFT_*b a* | ” ... |
| 247 | FFT_*b a* | ” ... |
| 248 | FFT_*b a* | ” (Frequency #10) |
| 249 | Classes | Application class, as assigned in [1] |

# 5 Conclusion

The pre-computed discriminator-data sets are available off-of

`http://www.dcs.qmul.ac.uk/research/nrl/` and `http://www.cl.cam.ac.uk/Research/SRG/netos/nprobe/data/papers/sigmetrics/index.html`. While we make every effort to ensure they are without flaw — and that these archives are maintained — they are provided on an as-is basis.

Additionally, scripts/code to allow the community to generate discriminators for their own data are available from the above web-sites.

### Future Work

It is the intention of the authors to provide sets of discriminators and input data for other traffic as it becomes available.

### Thanks

We thank Matt Roughan for some illuminating conversations pertaining to this work and we are indebted to a number of tool makers notably, everyone who made tcpdump/libpcap the backbone to our research it is today, Christian Kreibich for his *netdude* tools and Shawn Ostermann for the *tcptrace* toolkit.

## References

[1] A. Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications, 2005. http://www.cl.cam.ac.uk/ awm22/publication/moore2005toward.pdf.

[2] A. W. Moore. Discrete content-based classification — a data set. Technical report, Intel Research, Cambridge, 2005.

[3] Andrew Moore, James Hall, Christian Kreibich, Euan Harris, and Ian Pratt. Architecture of a Network Monitor. In *Passive & Active Measurement Workshop 2003 (PAM2003)*, April 2003.

[4] Sourceforge. netdude, 2005. http://netdude.sourceforge.net.

[5] Shawn Ostermann. tcptrace, 2003. http://www.tcptrace.org.

[6] M. Mathis and J. Mahdavi and S. Floyd and A. Romanow. TCP Selective Acknowledgement Options, October 1996. M.MATHIS, J.MAHDAVI, S.FLOYD, AND A.ROMANOW. TCP Selective Acknowledgement Options, October 1996.

[7] S. Floyd and J. Mahdavi and M. Mathis and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP, July 2000. S.FLOYD, J.MAHDAVI, M.MATHIS, AND M.PODOLSKY. An Extension to the Selective Acknowledgement (SACK) Option for TCP, July 2000.

[8] T. Berners-Lee and R. Fielding and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0, May 1996. T.BERNERS-LEE, R.FIELDING, AND H.FRYSTYK. Hypertext Transfer Protocol - HTTP/1.0, May 1996.

[9] J. Postel. Transmission Control Protocol, September 1981. INFORMATION SCIENCES INSTITUTE, U. O. S. C. Transmission Control Protocol, September 1981.

[10] N. G. Duffield, J. T. Lewis, N. O'Connell, R. Russell, and F. Toomey. Entropy of ATM traffic streams. *IEEE Journal on Selected Areas in Communications*, 13(6):981–990, August 1995.