Spring 2011

BitTorrent Traffic Detection with Deep Packet Inspection and Deep Flow Inspection

Raymond Wong
San Jose State University

Recommended Citation
Wong, Raymond, "BitTorrent Traffic Detection with Deep Packet Inspection and Deep Flow Inspection" (2011). *Master's Projects*. 250.
DOI: https://doi.org/10.31979/etd.6d8k-gr9q
https://scholarworks.sjsu.edu/etd_projects/250

BitTorrent Traffic Detection with Deep Packet Inspection and Deep Flow Inspection

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirements for the

Degree

Master of Science

By

Raymond Wong

Spring 2011

Raymond Wong has passed the defense for the project BitTorrent Traffic Detection with Deep Packet Inspection and Deep Flow Inspection.

Dr. Teng Moh	Date
Dr. Robert Chun	Date
Dr. Melody Moh	Date

NOTE: The advisor should send the final report to the graduate coordinator so that the student can be cleared for graduation

Table of Contents

Figures and Tables:

Abstract:

The peer-to-peer (P2P) technology has been well developed with the internet networking and BitTorrent (BT) is one of the very popular P2P sharing protocols widely used. BT network traffic detection has become very challenging in recent years due to smarter peer-to-peer applications. During my CS297 project, a new improved detection method based on Deep Packet Inspection (DPI) and Deep Flow Inspection (DFI) was proposed for detecting BT packets. The preliminary experiments show promising results in terms of detection rate. In my CS298 project, the proposed method is implemented in C and Matlab. In addition, the detection rate and performance of the proposed method was also compared with other existing methods. The major results have been submitted to a conference for a paper.

List of Acronyms

BT	BitTorrent
DFI	Deep Flow Inspection
DPI	Deep Packet Inspection
FTP	File Transfer Protocol
ISP	Internet service provider
MTU	Maximum Transmission Unit
P2P	Peer-to-Peers
SVM	Support Vector Machine
TCM	Time correlation metric
WiMax	Worldwide Interoperability for Microwave Access

Chapter 1:
Introduction

1.1 Why detection of P2P traffic is important?

The technology of internet networking has been evolving for the past few decades. It has expanded exponentially. Back in the old days, if we needed to get online, using analog phone line modem was the only option. Nowadays, we can easily get online by using broadband, 3G or even WiMax (Worldwide Interoperability for Microwave Access). Another technology that has been well developed with the internet networking is called peer-to-peer (P2P) network. BitTorrent (BT) is a very popular peer-to-peer file sharing protocol and is one of most accepted P2P protocols.

Detecting P2P traffic is important and it can be seen in different aspects from different people. For an enterprise network, the administrators may want to rate-limit the P2P traffic such that it can have enough bandwidth reserved for other critical/important applications. For local broadband internet service providers (ISP), they may want to limit the cost charged by the upstream ISP. Finally, for regular home users, most of them still have async internet connection service from their ISP. It means that the upstream rate and downstream rate are not equal and upstream rate is usually much lower than the downstream rate. If the upstream is congested, it will affect the overall internet experience.

1.2 Major existing techniques for detecting BT traffic

In general, there are four major methods to detect BT traffic: port-based technique, deep packet inspection (DPI) technique, deep flow inspection (DFI) technique, and combinations of above mentioned techniques. The port-based method is based on the TCP and/or UDP port and it assumes that BT clients use fixed port for data or messages transfer. This method is obsolete due to the fact that newer BT clients nowadays utilize user-defined port, random port, changed port or even camouflage port to avoid the port detection mechanism. DPI method is based on the packet signature and it looks at packet payload to detect BT packet. DFI method detects BT packets based on the TCP flow such as average packet size and total bytes transferred. Finally these techniques above can be used together to increase the detection rate.

In this project, a new improved BT packets detection method based on Deep Packet Inspection (DPI) follow by Deep Flow Inspection (DFI) is proposed. The proposed method is implemented in C and Matlab for simulation and verification propose. The experiments show promising results for both the detection rate as well as the execution time efficiency.

1.3 Outline of this report

This report is organized in following manner. Chapter 1 is an introduction. Chapter 2 is the literature survey. Chapter 3 is the summary of my CS297 research. In chapter 4, detailed explanations of the architecture and the implementation of the proposed method are discussed. The simulation results can be found in chapter 5. Finally, we will discuss the conclusion.

Chapter 2:
Literature Survey:

In this chapter, we will go through the detection methods found in the area of academia. A literature survey is important to this project as it shows how other people solve this problem. In general, there are three major techniques for detecting P2P traffic from other regular traffic.

The first method is based on the TCP and/or UDP port. The assumption of this method is that the P2P applications use fixed port to communicate between peer computers. One obvious advantage is that it is very easy to implement and almost no computation power is required. Moreover, using the traffic identification based on port not only can directly identify individual P2P applications (for example, eMule, Bit Torrent), but also can easily eliminate well-known non-P2P application (for example, FTP, E-Mail). The bottom line is that different applications would use different TCP/UDP ports for data communication. In order to avoid being detected by the detection software, most P2P application nowadays use new technology such as user-defined port, random port, changed port and camouflage port to avoid the port detection mechanism. As a result, this port based approach is obsolete.

Another detection method is called deep packet inspection (DPI) which is based on the payload of the packets. This detection mechanism is to inspect these particular characteristics to identify P2P traffic. P2P software's handshaking messages often follow certain patterns when communicating. In another word, they have some fixed characteristics in the application layer payload. One example would be the BitTorrent protocol. There is always a "BitTorrentprotocol" string appear in their handshaking packets. The advantage of this approach is high accuracy rate and implementation robustness; however, there are several disadvantages. First, since the payload needs to be examined, no encryption data can be supported. Secondly, this method leads to privacy issue because of the need to inspect the content of payload. Also, the signature of the P2P protocol may keep changing due to the continuous change of protocols. Finally, sometimes it is difficult to get characteristics out from open source Software. It is often more difficult for closed source software.

The third detection method is using the traffic flow to detect P2P traffic; it is called deep flow inspection (DFI). As the name implies, the analysis or the classification of P2P traffic is a flow-based, focusing on the connection level patterns of P2P applications. Thus, it does not require any payload analysis, unlike DPI. Because it doesn't require payload analysis, encrypted data packets can be easily supported. The down side of this approach is that there is an additional step of extracting the connection level pattern for the P2P traffics. And yet, there is no rule of thumb for which network feature should be used in this method.

2.1 Traditional methods approach

Liu et al. [10] propose a simple deep packet inspection (DPI) algorithm to detect BitTorrent traffic. Their algorithm is based on the handshaking message between the BitTorrent peers. According to the authors, in the BitTorrent header of the handshake messages has the following format. <a character (1 byte)><a string (19 byte)>

The first byte is a fixed character with value '19', and the string value is 'BitTorrent protocol'. Based on this common header, they use this feature as the signatures for identifying BitTorrent traffic.

Le and But [9] use a deep flow inspection (DFI) algorithm to classify traffic. They focused on the packet length statistics of traffic as the features for their classifier ~~for~~ detecting BitTorrent traffic. The four network features used in this paper are: minimum payload, ratio of small packets, ratio of large packets and small payload standard deviation. Three types of traffic traces were used to train and test their classifier. These include known BitTorrent traffic, known FTP traffic, and other traffic.

Erman et al. [6] propose semi-supervised learning algorithm to classify traffic. Their algorithm involves a two steps approach to training their classifier. The first step is clustering. In this step, the flows with the applications labeled will be partitioned by a unsupervised clustering algorithm. K-means algorithm was used in this step however it is not restricted it is the only algorithm can be used. As the authors pointed out that the key benefit of the unsupervised learning approach is the ability to identity hidden patterns. For example, new applications can be examining flows that form a new clustering.

The second step is to map clusters to applications and to determine the clusters label based on the flows label. This mapping is based on the estimation of the probabilities that the labeled flow samples within each of the clusters. It can be estimated by the maximum likelihood estimate, n_{jk}/n_k, where n_{jk} is the number of flows that were assigned to cluster k with label j, and nk is the total number of (labeled) flows that were assigned to cluster k. Figure 1 depicts the overview of the classification algorithm.

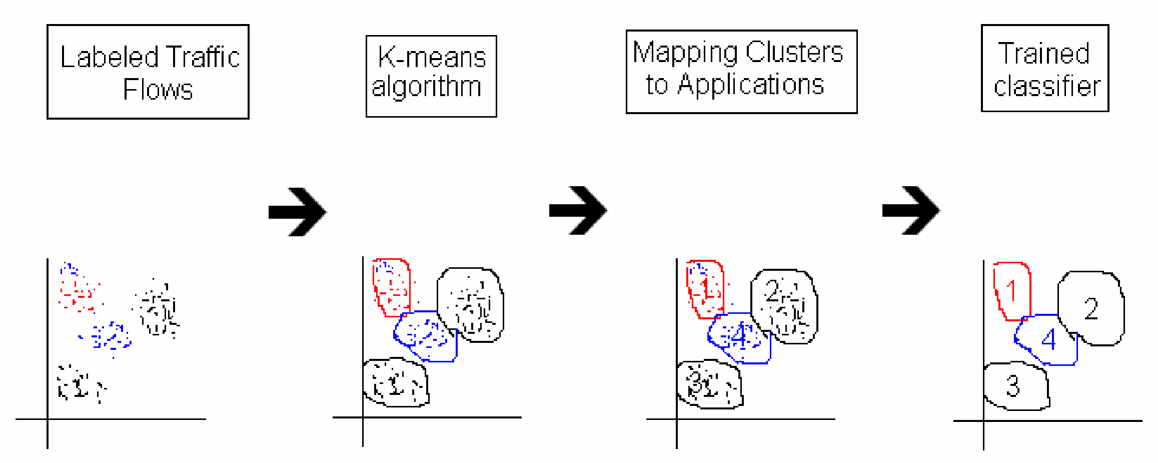


Figure 1: Overview of classification algorithm [6]

There are also some papers utilizes the popular classification techniques to detect P2P traffic. For example, Chen et al. [3] tried to use neural network as the tool to detect P2P traffic, where in the other paper [4], they used support vector machine (SVM). There is a paper [11] simply use packet length to detect P2P packets. Their claim is as follow. In P2P applications, there are many small size packets and large size (close to maximum transmission unit (MTU)) packets. The small size packets are often used to transfer messages between server and client such as synchronization and acknowledgement. The large size packets are usually used to transfer data (actual sharing content). Figure 2 shows the PDF of the packet length for some common applications.

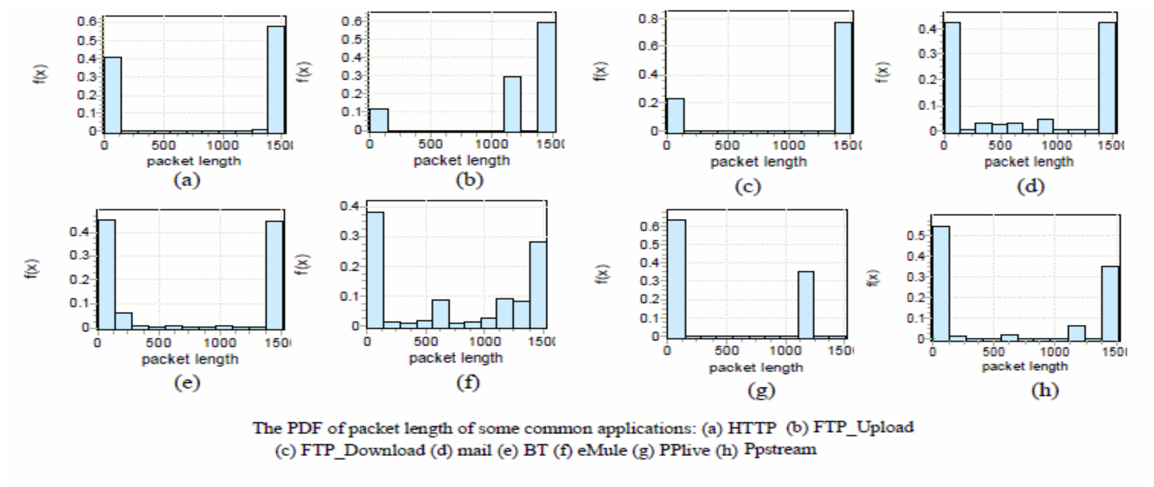


Figure 2: PDF of packet length for common applications [11]

2.2 Combined traditional methods approach

There are some other papers tried to combine more than one technique mentioned above to solve the P2P classification problem. The examples would be the papers written by Chen et al. [2] and Wang et al. [12]. Both papers claim that their combined approach is better than the tradition approaches. Chen et al [2] claim that using both DPI and DFI approach together can make both of the detection algorithm comprise each other, thus, the detection rate will increase. Another advantage of this approach is the parallelism (i.e. DPI and DFI algorithm can be executed in parallel). Instead of putting the DPI and DFI module execute in parallel, Wang et al. [12] execute them in a serial manner. Below are the major steps in their detection algorithm.

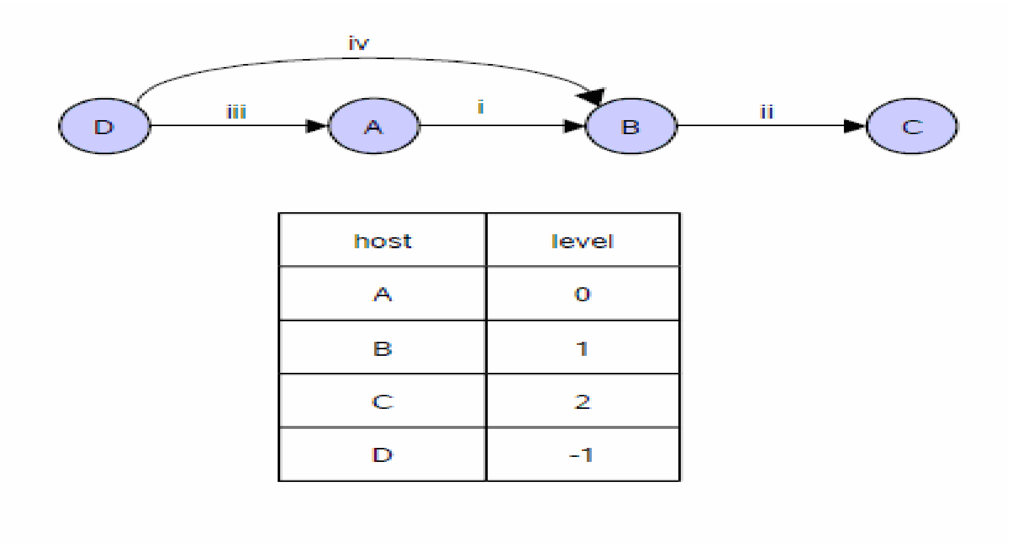
1) After obtaining a packet for examinations, first use traffic detection based on port to filter the common P2P traffic.
2) Then through a DFI module to match the characteristics of the data stream whether the traffic is P2P.
3) For those packets belonging to P2P flow, payload characteristic module (aka DPI module) match-up will be conducted to find out P2P traffic type.
4) Finally, two traffic detection groups will be formed. They are the “P2P known type traffic” and “P2P unknown type traffic”.

2.3 Other approaches

There is another interesting paper [5] which is based on the port analysis to create a graph. Typical peer-to-peer traffic detection methods focus on analyzing the host or packets, in this paper, the authors detect Peer-to-Peer traffics based on the port analysis. The following example depicts their algorithm. Figure 3 shows an example of how to construct a graph based on the paper's method. There are four nodes in this network.

Node A first makes a connection to node B. Thus we assigned level 0 to node A and assign level 1 to node B. After that, B makes a new connection to node C. As a result, we assigned level 2 to node C. Notice that once the level is assigned to a host, this level cannot change. Therefore, when node D makes another new connection to node A, we do not change A's level. Instead, we assign -1 to node D which is one level lower to node A. The last connection did not change any level from nodes because node D and B already have its own level. Once we have the graph constructed, we can use the rules below to determine if this port is considered used by P2P application. (P.S.: The authors didn't specify how to determine these thresholds)

1) The number of hosts that act both as servers and clients ("ClientServers") in the specific port exceeds the ClientServer threshold.
2) The network diameter is at least as great as 2,
3) The numbers of hosts that are present in the first and last level of the network exceed the Edge Level threshold.



host	level
A	0
B	1
C	2
D	-1

Figure 3: An example to construct a graph from nodes activities [5]

Another approach proposed by Zhang et al. [14] is to detect BT traffic based on recording and analyzing the peer-information which is obtained from the BitTorrent signaling traffic. In addition, the algorithm proposed by the authors can be run and distributed in different computers.

There is a paper proposed by Keralapura et al. [7] used a 2-stage detection mechanism to detect P2P traffic. The first stage is based on a newly proposed algorithm, Time correlation metric (TCM) algorithm, to detect P2P-like traffic. This algorithm is based on the traffic packets arrival time and it is a flow based approach. Since it doesn't require examining the payload of the packets, this algorithm can also work with encrypted traffic. The second stage is based on a payload signature extraction detection algorithm. With the stage 1 and stage 2, the P2P traffics can be accurately classified

Finally, in the paper written by Basher et al. [1], they didn't present any new algorithms to detect P2P traffics. Instead, they did a comparative analysis between web and peer-to-peer traffic. The data base that they used contains 1.12 billion IP packets totaling 639.4 Gigabytes. In addition, they also did a comparative study between two P2P protocols namely BitTorrent and Gnutella. The analyzed characteristics metrics are listed below in table I.

Table I: analyzed characteristics metrics by paper [1]

Characteristics	Comments
Flow size	The total bytes transferred during a TCP flow
Flow inter-arrival time	The time interval between two consecutive flow arrivals
Duration	The time between the start and the end of a TCP flow
Flow concurrency	The maximum number of TCP flows a single host uses concurrently to transfer content
Transfer volume	The total bytes transferred to and from a host during its activity period
Geography Distribution	The distribution of the shortest distance between individual hosts and authors' campus along the surface of the Earth

2.4 Performance Evaluation Matrix

Based on the literature survey that I have done, generally speaking there are two kinds of performance evaluation matrix used by the authors of the papers. The first type [12][13][10][9][6] simply uses the term "accuracy". It is defined as the number of correctly classified items divided by the total number of items. As the name implied, higher the accuracy represents the better the proposed algorithm.

Some other papers [4][3][11][5][7] used a rather formal definition of the statistical equations (equation 1 to 4) to evaluate the performance. The goal was to maximize (i.e. 1) the True Positive Rate (TPR) and True Negative Rate (TNR); at the same time to minimize (i.e. 0) the False Positive Rate (FPR) and False Negative Rate (FNR).

$$TPR(TruePositiveRate) = \frac{TP}{(TP + FN)} \quad \text{...Equation 1}$$

$$TNR(TrueNegativeRate) = \frac{TN}{(TN + FP)} \quad \text{..Equation 2}$$

$$FPR(FalsePositiveRate) = \frac{FP}{(FP + TN)} \quad \text{.......................................Equation 3}$$

$$FNR(FalseNegativeRate) = \frac{FN}{(TP + FN)} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{Equation 4}$$

where TP is the number of correctly classified objects for a given P2P class; TN is the number of correctly classified objects for Non-P2P class; FP is the number of objects falsely identified as P2P class and FN is the number of objects from P2P class that are falsely labeled as Non-P2P class.

Chapter 3:
Summary of CS297 Research

3.1 Proposed System

After the literature survey, a new improved BT detection method is proposed. The proposed BT detection method can be roughly divided into two parts, namely offline training module and online classification module. Packet flow can be either classified as BT class or non-BT class. The offline training module and the DFI portion of the online module system is inspired by the paper written by Erman et al. [6]. However, the classification accuracy is improved by using both DPI and DFI methods whereas only DFI was used in the paper proposed by Erman et al. [6]. The proposed system first uses DPI then DFI to classify packets which is different than the paper proposed by Wang et al. [12]. With the proper ordering of DPI and DFI, the simulation results show that the running execution time is improved. Figure 4 depicts our proposed system to classify BT packet flows.

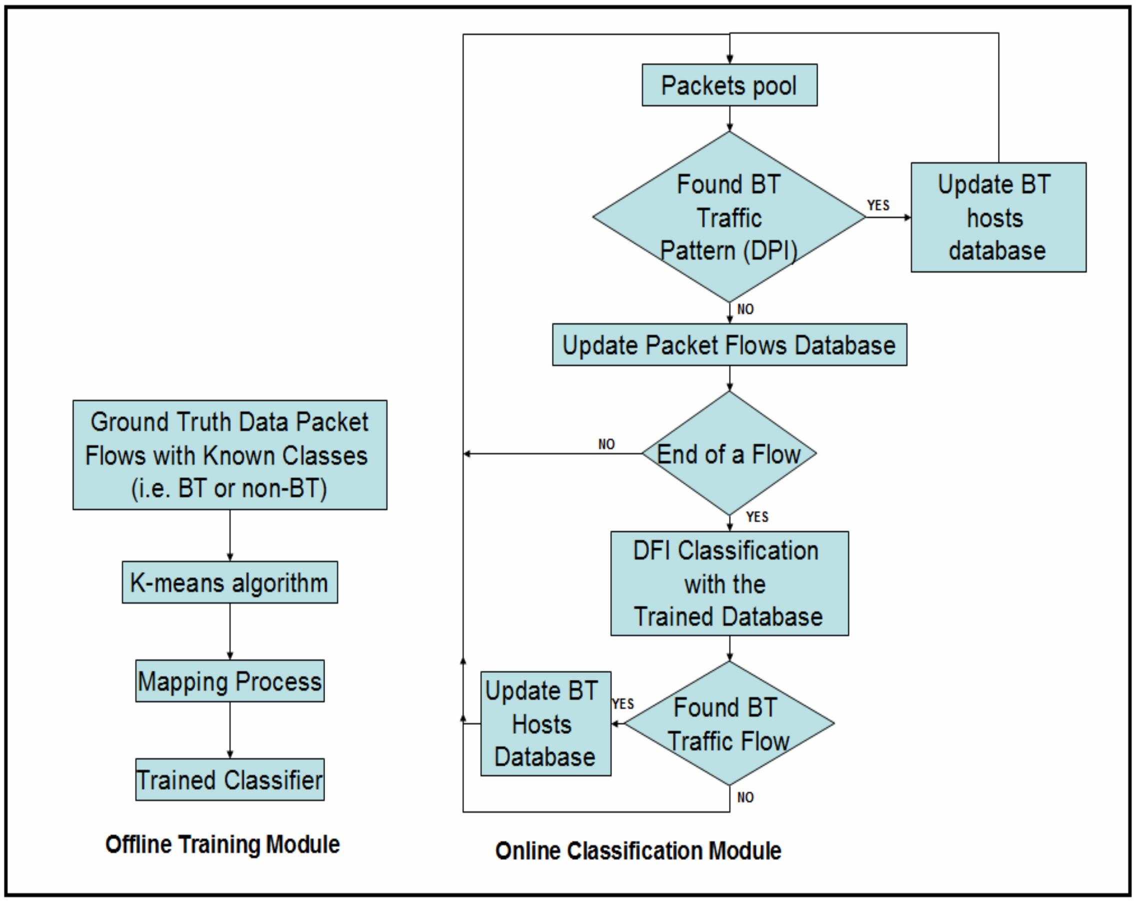

Packets pool
Found BT Traffic Pattern (DPI)
YES
Update BT hosts database
NO
Update Packet Flows Database
Ground Truth Data Packet Flows with Known Classes (i.e. BT or non-BT)
NO
End of a Flow
YES
K-means algorithm
DFI Classification with the Trained Database
Mapping Process
Update BT Hosts Database
YES
Found BT Traffic Flow
Trained Classifier
NO
Offline Training Module
Online Classification Module

Figure 4: Proposed system to classify BT packet flows

3.2 Preliminary results

In this section, preliminary results from my CS297 will be briefly discussed. There are two goals of performing these experiments. The first goal is to understand how many packets should be used to train a reliable classifier. The second goal is to make sure the packets captured software that I wrote is indeed working.

3.2.1 Ground truth generation

The ground truth is the packet flows with known classes. In order to train a classifier, there are two types of packet flows needed to capture, namely the BT and non-BT packet flows. To capture the BT packets, I manually force the BT client to use a single TCP port (i.e. 1200) for data transfer. Thus, all the BT traffic must go through this TCP port. Then, I start a sample torrent file and the BT client will automatically start downloading/uploading the contents. At the same time, I start my packet capturing program to obtain the packets. Similarly, to capture non-BT packets, I start my packet capturing program while we were creating non-BT network activities including HTTP, FTP and SSH. With the known class of the packets in the PCAP files, I could start training the classifier.

3.2.2 Study of DFI classifier accuracy

Figure 5 shows the classifier accuracy with increasing number of BT packet flows used to train the classifier. The classifier was first trained with a set of BT samples, and then it was tested against with some other BT packet flows to observe the accuracy. This experiment gives us some clues about the number of packet flows should be used in order train a reliable classifier for the DFI module. As expected, the more BT packets are used to train the classifier, the better the accuracy is. However, as the number of the BT packets increase, the classifier will be saturated at some point. After that, even more packets is provided, the accuracy does not increase significantly.

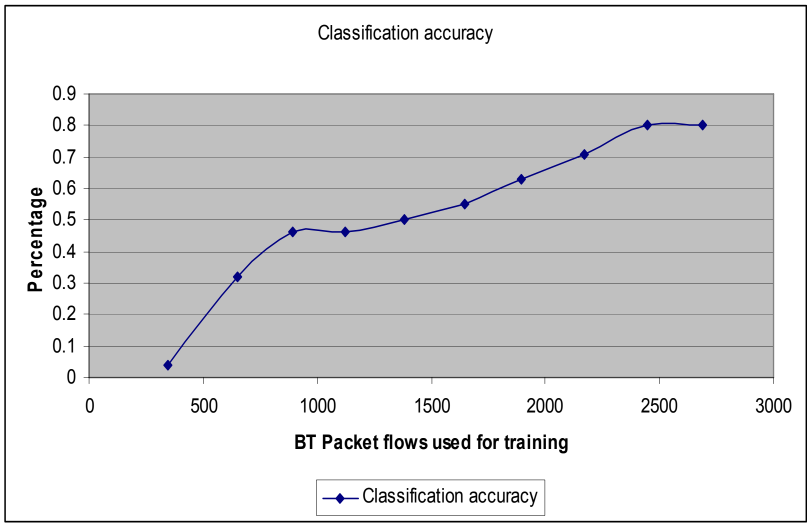


Figure 5: Classifier accuracy with different training samples

Chapter 4: CS298 Research

4.1 BT packets classification software

One of the deliverables of my CS298 project is the BT packets classification software. We will discuss the details in this chapter. The development environment of the software will be first discussed. After that, the discussion of two major components, namely offline training module and online classification module, will be followed.

4.1.1 Development environment

The BT packet classification software is built with Cygwin under Windows environment. Cygwin provides a UNIX like environment with the full GCC supported for software development and windows provides user friendly environment without worrying about the network card drivers. This combination provided a user friendly development environment and also was able to utilize open source libraries. Packets capture is made possible by WinPcap [16] library. WinPcap is an open source library that allow user to set his/her network interface card (NIC) to operate in "promiscuous" mode. Thus, all the packets going through the network will be captured. Figure 6 shows the system level of architecture of the software written in this project.

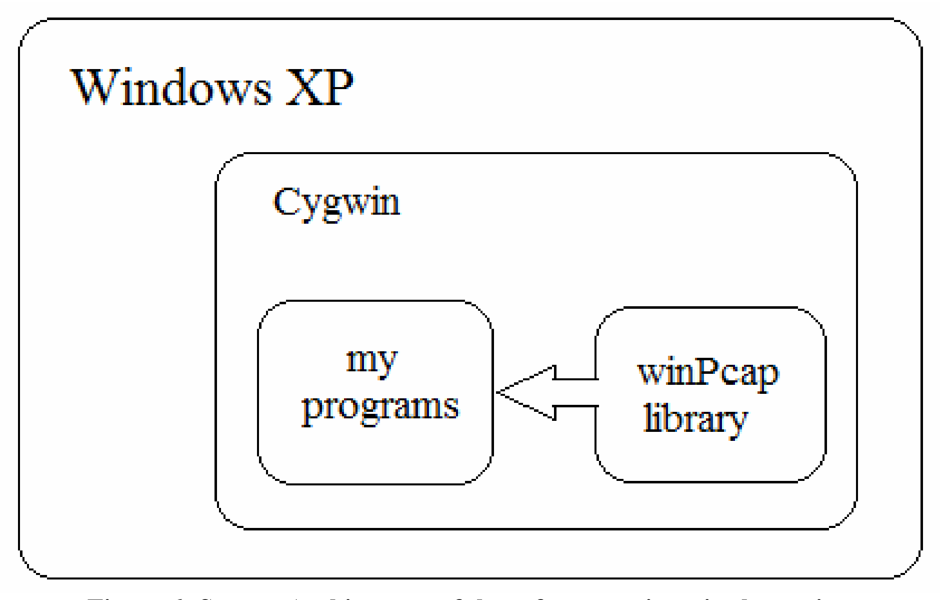


Figure 6: System Architecture of the software written in the project

4.2 Architecture and Implementation Details of the Proposed System

4.2.1 Offline training module

The objective of the offline training module (shown in figure 7) is to create a trained database for the DFI classification used in the online classification module. I divided the offline training module into three separate sub-programs for the ease of development. There are two C programs and one Matlab program. They are the packet capture program and the packet features extraction program. Both programs are linked with the winPcap library. As the names implied, the function of the packet capture program is to communicate with the network interface card (NIC) and to capture packets from the network. The captured packets will be stored in a file with PCAP format. The second C program is used to extract the network features from the packet file. There are 10 features to be extracted and they are shown in table I. Finally, the extracted features will be used to train the classifier. The classifier is based on the K-means clustering algorithm and it is inspired by paper written by Erman et al. [6]. The classifier program is written in Matlab. The advantage of using Matlab over C is because Matlab has a lot of build in numerical computation routines and the K-means cluster algorithm routine is also supported.

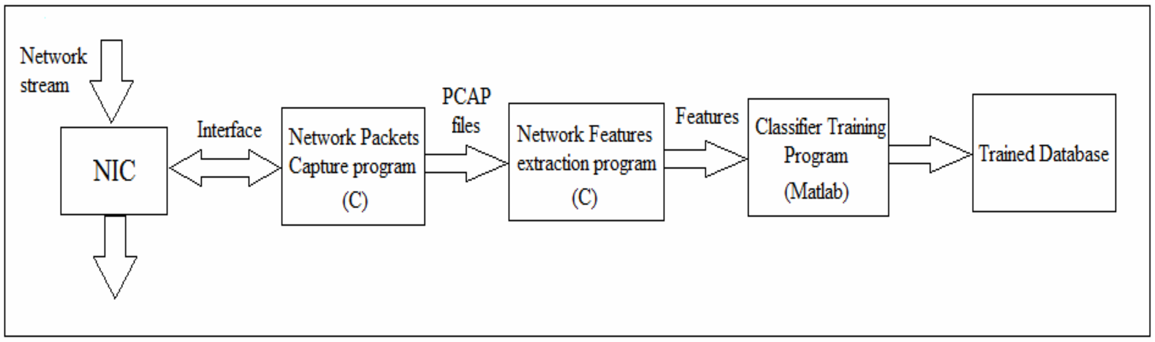

Network stream
NIC
Interface
Network Packets Capture program (C)
PCAP files
Network Features extraction program (C)
Features
Classifier Training Program (Matlab)
Trained Database

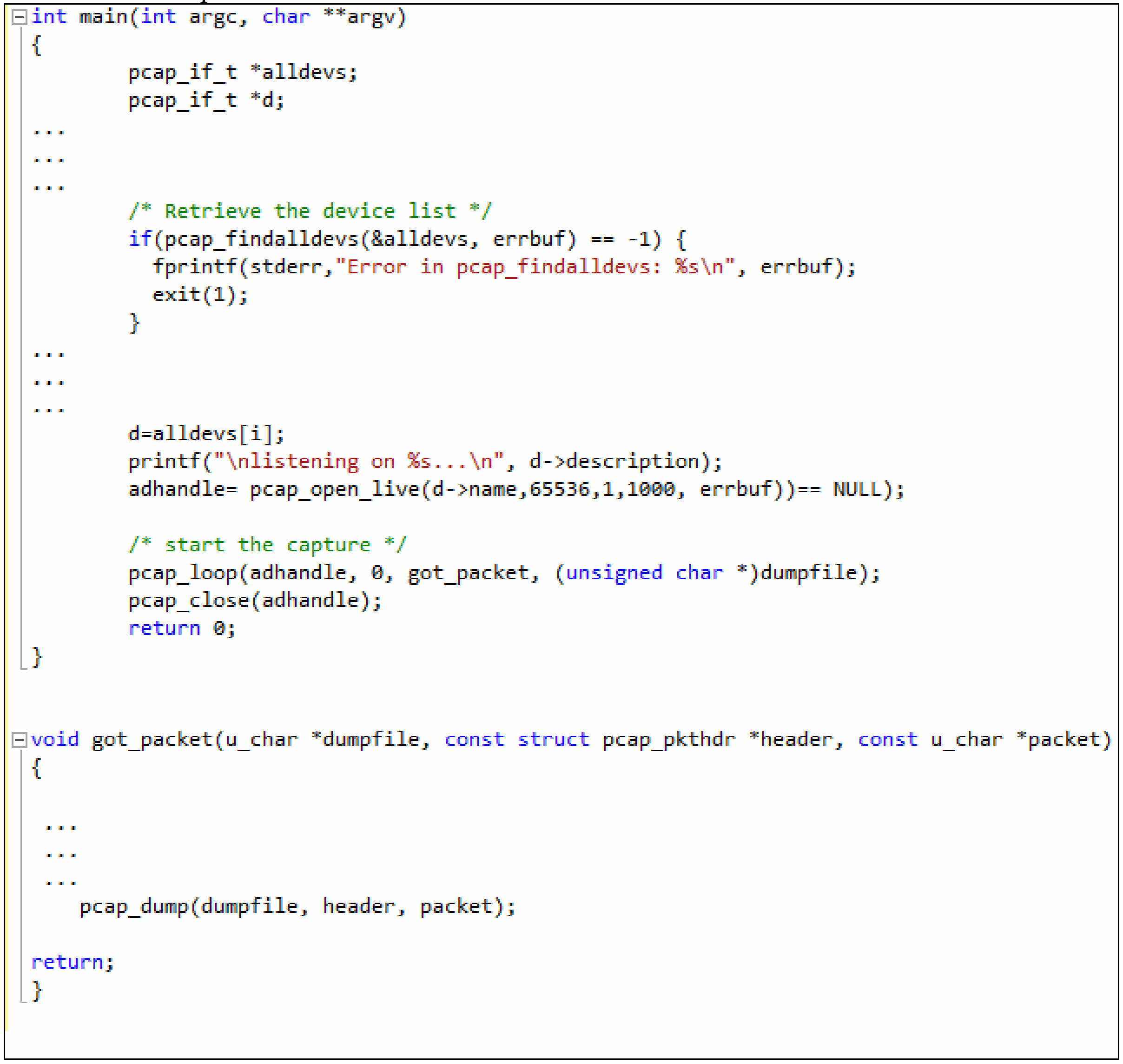
Figure 7: Offline Training Flow Chart

Table II: Network Features used in this project

No.	Features
1	Total number of packets
2	Average packet size
3	Total bytes
4	Total header (transport plus network layer) bytes
5	Number of flow initiator to flow responder packets
6	Total flow initiator to flow responder payload bytes
7	Total flow initiator to flow responder header bytes
8	Total flow initiator to flow responder header bytes
9	Number of flow responder to flow initiator packets
10	Total flow responder to flow initiator header bytes

4.2.1.1 Network packets capture program (C program)

Capturing packets with WinPcap API is quite easy. Figure 8 shows a sample code to capture packets and save them into a PCAP file. In order to capture packets, API function pcap_open_live is used. After opening the NIC device, the next step is to set up a call back function. A call back function (got_packet) will be fired when a packet is either received or sent. Inside the call back function, another API function pcap_dump is used to save the packets into a PCAP file.

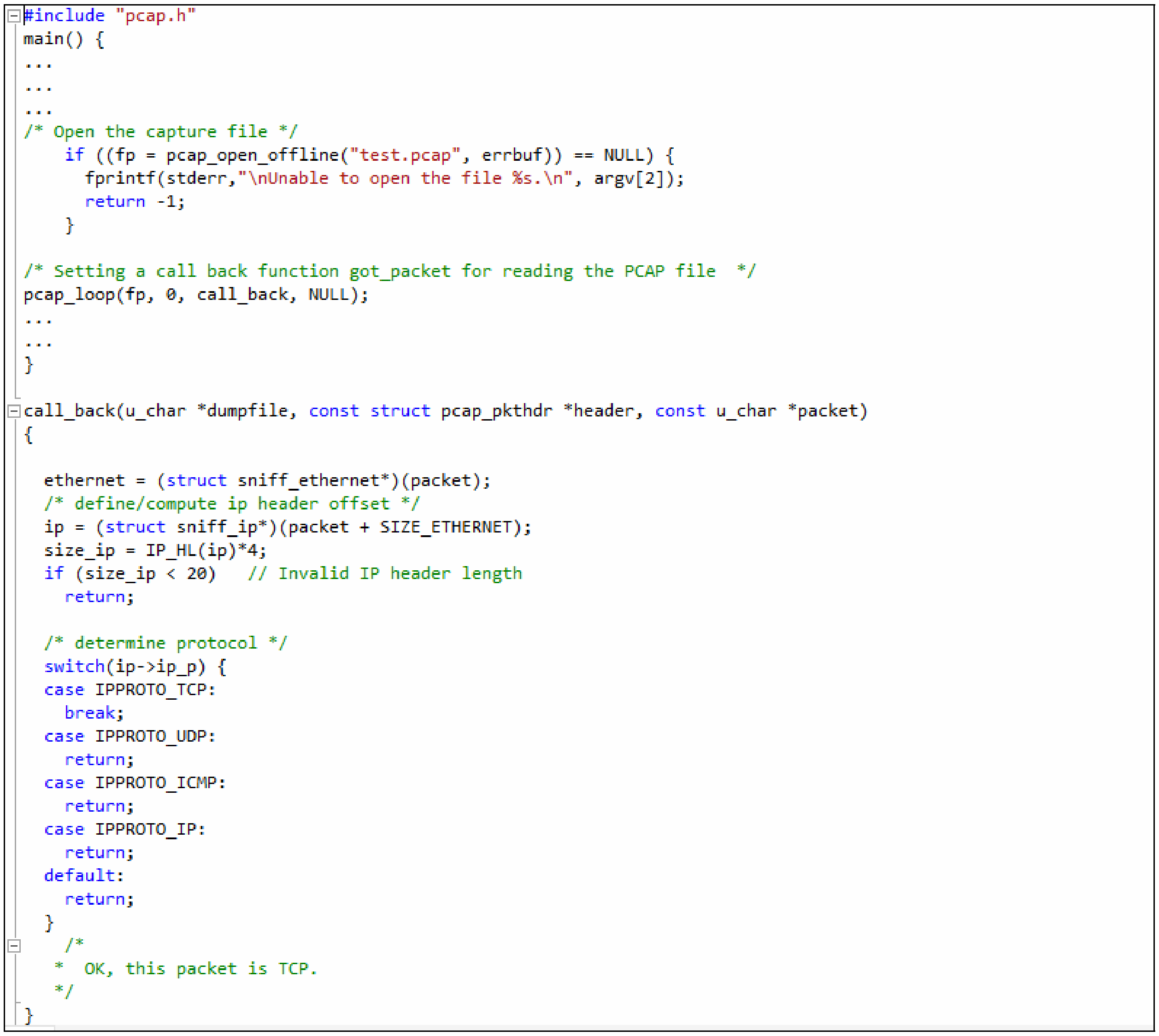


```
int main(int argc, char **argv)
{
        pcap_if_t *alldevs;
        pcap_if_t *d;
...
...
...
        /* Retrieve the device list */
        if(pcap_findalldevs(&alldevs, errbuf) == -1) {
          fprintf(stderr,"Error in pcap_findalldevs: %s\n", errbuf);
          exit(1);
        }
...
...
...
        d=alldevs[i];
        printf("\nlistening on %s...\n", d->description);
        adhandle= pcap_open_live(d->name,65536,1,1000, errbuf))== NULL);

        /* start the capture */
        pcap_loop(adhandle, 0, got_packet, (unsigned char *)dumpfile);
        pcap_close(adhandle);
        return 0;
}


void got_packet(u_char *dumpfile, const struct pcap_pkthdr *header, const u_char *packet)
{

 ...
 ...
 ...
    pcap_dump(dumpfile, header, packet);

return;
}
```

Figure 8: Packet capture sample code

4.2.1.2 Packet features extraction program (C program)

In order to read packets from a PCAP file, WinPcap API function (pcap_open_offline()) is used to open the PCAP file. This function provides similar interface to fopen() C function and it returns a handle if the command was successfully executed. Function pcap_loop(fp, 0, call_back, 0) is used next for setting a callback function while reading the PCAP file. The pcap_loop will call the function call_back() for every packet found in the PCAP file. In the callback function, the packet stream is stored in the location where the pointer variable (*packet) is pointing at. Since we are only interested in the TCP packets, all other non-TCP packets will be filtered. After the filtering, the networking features such as packet size and header size will be computed and stored in a network feature database. Figure 9 shows the sample coding of opening a PCAP file and setting a callback function for reading the PCAP file.



```
#include "pcap.h"
main() {
...
...
...
/* Open the capture file */
    if ((fp = pcap_open_offline("test.pcap", errbuf)) == NULL) {
      fprintf(stderr,"\nUnable to open the file %s.\n", argv[2]);
      return -1;
    }

/* Setting a call back function got_packet for reading the PCAP file  */
pcap_loop(fp, 0, call_back, NULL);
...
...
}

call_back(u_char *dumpfile, const struct pcap_pkthdr *header, const u_char *packet)
{

  ethernet = (struct sniff_ethernet*)(packet);
  /* define/compute ip header offset */
  ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
  size_ip = IP_HL(ip)*4;
  if (size_ip < 20)   // Invalid IP header length
    return;

  /* determine protocol */
  switch(ip->ip_p) {
  case IPPROTO_TCP:
    break;
  case IPPROTO_UDP:
    return;
  case IPPROTO_ICMP:
    return;
  case IPPROTO_IP:
    return;
  default:
    return;
  }
    /*
   *  OK, this packet is TCP.
   */
}
```

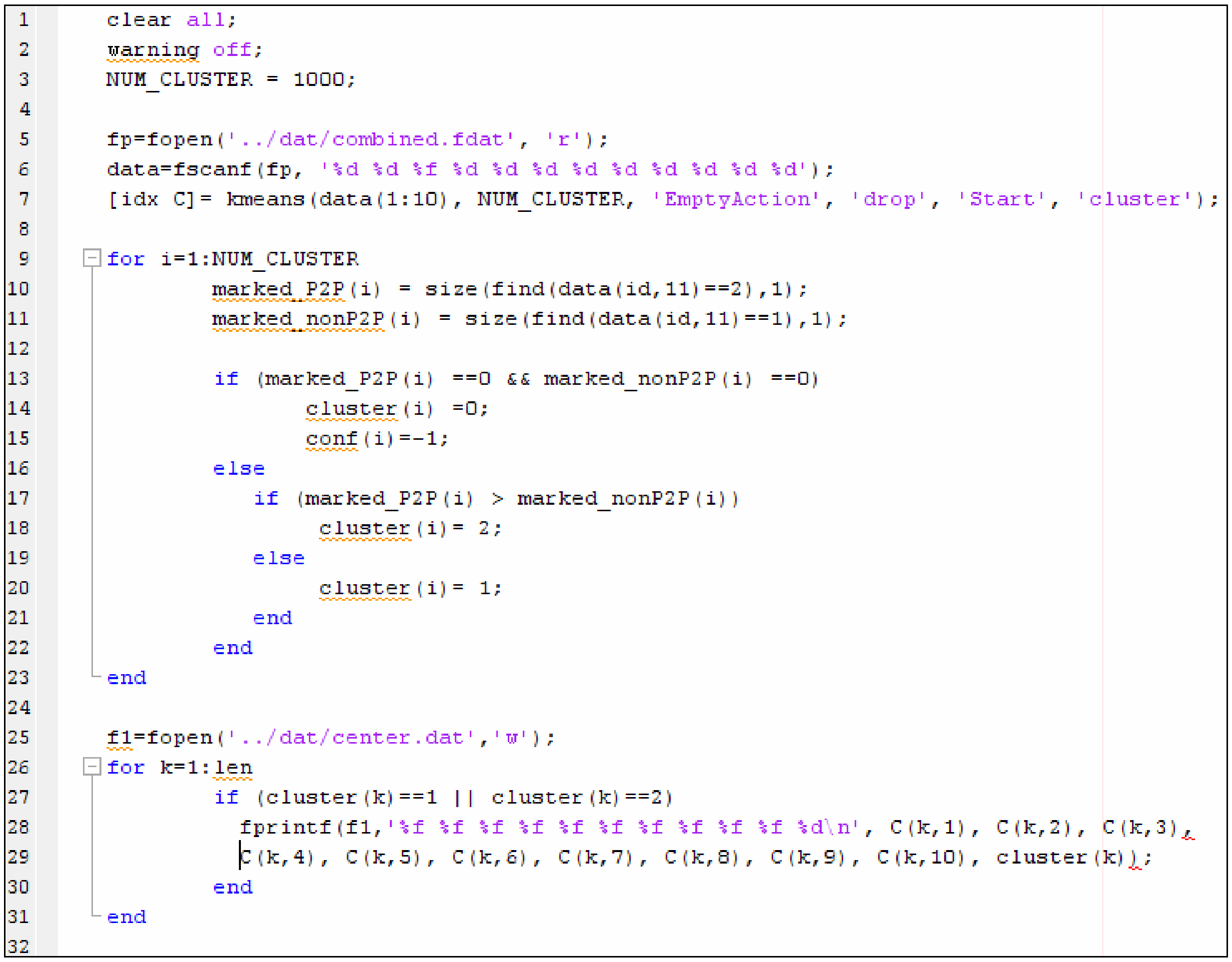
Figure 9: Sample coding for using Winpcap to read a PCAP file

4.2.1.3 Classifier Training program (Matlab)

As discussed previously, Matlab provides a set of build in numerical computation routines as well as the k-means algorithm support. Figure 10 shows the Matlab script to train the classifier. As we can see from line 6-7 (figure 10), the features are being read into a variable called "data". After that, Matlab command "kmeans" is applied to the data. Note that the second parameter indicates that we will cluster the data into 400 groups. The third parameter "'EmptyAction', 'drop'" of the kmeans command indicates that empty cluster will be dropped if a resulting cluster has no member associated.

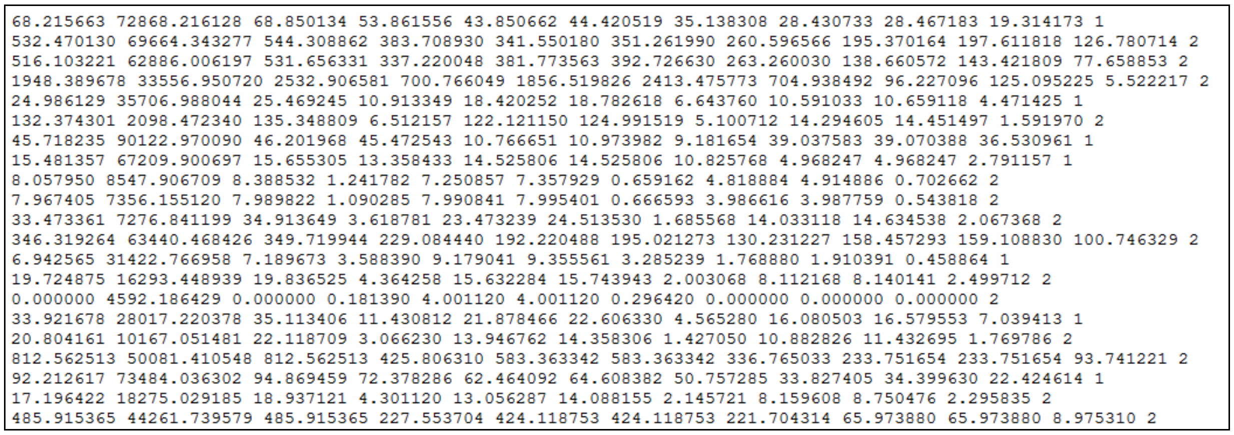
Finally, the "'start', 'cluster'" means that the initial clusters will be based on the kmeans cluster results of the random sampling of data.

With the clusters created, the next step is to determine if a given cluster is a BT cluster or a non-BT cluster. A simple majority rule is used to determine the class of the cluster. If number of BT packets is larger than the non-BT packets, that given cluster will be marked as BT cluster. On the other hand, if number of non-BT packets is larger than the BT packet, the cluster will be marked as non-BT cluster. The final output of this Matlab program is a trained database which contains the features centers. Figure 11 shows a portion of trained database file.



```
clear all;
warning off;
NUM_CLUSTER = 1000;

fp=fopen('../dat/combined.fdat', 'r');
data=fscanf(fp, '%d %d %f %d %d %d %d %d %d %d %d %d');
[idx C]= kmeans(data(1:10), NUM_CLUSTER, 'EmptyAction', 'drop', 'Start', 'cluster');

for i=1:NUM_CLUSTER
        marked_P2P(i)  = size(find(data(id,11)==2),1);
        marked_nonP2P(i)  = size(find(data(id,11)==1),1);

        if (marked_P2P(i)  ==0 && marked_nonP2P(i)  ==0)
               cluster(i)  =0;
               conf(i)=-1;
        else
            if (marked_P2P(i) > marked_nonP2P(i))
                 cluster(i)= 2;
            else
                 cluster(i)= 1;
            end
        end
end

f1=fopen('../dat/center.dat','w');
for k=1:len
        if (cluster(k)==1 || cluster(k)==2)
          fprintf(f1,'%f %f %f %f %f %f %f %f %f %f %d\n', C(k,1), C(k,2), C(k,3),
          C(k,4), C(k,5), C(k,6), C(k,7), C(k,8), C(k,9), C(k,10), cluster(k));
        end
end

```

Figure 10: Sample Matlab code for training the classifier with K-means algorithm



```
68.215663 72868.216128 68.850134 53.861556 43.850662 44.420519 35.138308 28.430733 28.467183 19.314173 1
532.470130 69664.343277 544.308862 383.708930 341.550180 351.261990 260.596566 195.370164 197.611818 126.780714 2
516.103221 62886.006197 531.656331 337.220048 381.773563 392.726630 263.260030 138.660572 143.421809 77.658853 2
1948.389678 33556.950720 2532.906581 700.766049 1856.519826 2413.475773 704.938492 96.227096 125.095225 5.522217 2
24.986129 35706.988044 25.469245 10.913349 18.420252 18.782618 6.643760 10.591033 10.659118 4.471425 1
132.374301 2098.472340 135.348809 6.512157 122.121150 124.991519 5.100712 14.294605 14.451497 1.591970 2
45.718235 90122.970090 46.201968 45.472543 10.766651 10.973982 9.181654 39.037583 39.070388 36.530961 1
15.481357 67209.900697 15.655305 13.358433 14.525806 14.525806 10.825768 4.968247 4.968247 2.791157 1
8.057950 8547.906709 8.388532 1.241782 7.250857 7.357929 0.659162 4.818884 4.914886 0.702662 2
7.967405 7356.155120 7.989822 1.090285 7.990841 7.995401 0.666593 3.986616 3.987759 0.543818 2
33.473361 7276.841199 34.913649 3.618781 23.473239 24.513530 1.685568 14.033118 14.634538 2.067368 2
346.319264 63440.468426 349.719944 229.084440 192.220488 195.021273 130.231227 158.457293 159.108830 100.746329 2
6.942565 31422.766958 7.189673 3.588390 9.179041 9.355561 3.285239 1.768880 1.910391 0.458864 1
19.724875 16293.448939 19.836525 4.364258 15.632284 15.743943 2.003068 8.112168 8.140141 2.499712 2
0.000000 4592.186429 0.000000 0.181390 4.001120 4.001120 0.296420 0.000000 0.000000 0.000000 2
33.921678 28017.220378 35.113406 11.430812 21.878466 22.606330 4.565280 16.080503 16.579553 7.039413 1
20.804161 10167.051481 22.118709 3.066230 13.946762 14.358306 1.427050 10.882826 11.432695 1.769786 2
812.562513 50081.410548 812.562513 425.806310 583.363342 583.363342 336.765033 233.751654 233.751654 93.741221 2
92.212617 73484.036302 94.869459 72.378286 62.464092 64.608382 50.757285 33.827405 34.399630 22.424614 1
17.196422 18275.029185 18.937121 4.301120 13.056287 14.088155 2.145721 8.159608 8.750476 2.295835 2
485.915365 44261.739579 485.915365 227.553704 424.118753 424.118753 221.704314 65.973880 65.973880 8.975310 2
```

Figure 11: Content inside the trained database

4.2.2 Online classification module

Figure 12 shows the online classification module top level block diagram. There are three input parameters required by this module, namely Packet capture file, trained database and the modes of operation. The online classification program supports four modes of operation. They are the DPI, DFI, DFI followed by DPI (DFI/DPI) and DPI followed by DFI (DPI/DFI). All four modes require users to provide a PCAP file as source of the packet flows. The DPI mode is based on the string (i.e. " BitTorrentprotocol") comparison to determine if the encounter packet is BT type. The DFI mode first extracts packet flows features from the PCAP file. After that, it uses the database from the offline training module to determine the packet flow type. The DFI/DPI combines both DFI and DPI techniques together to classify packets. It first uses the DFI method to determine the packet flow type. After that DPI method will be followed to examine if the packets contain the BT pattern.

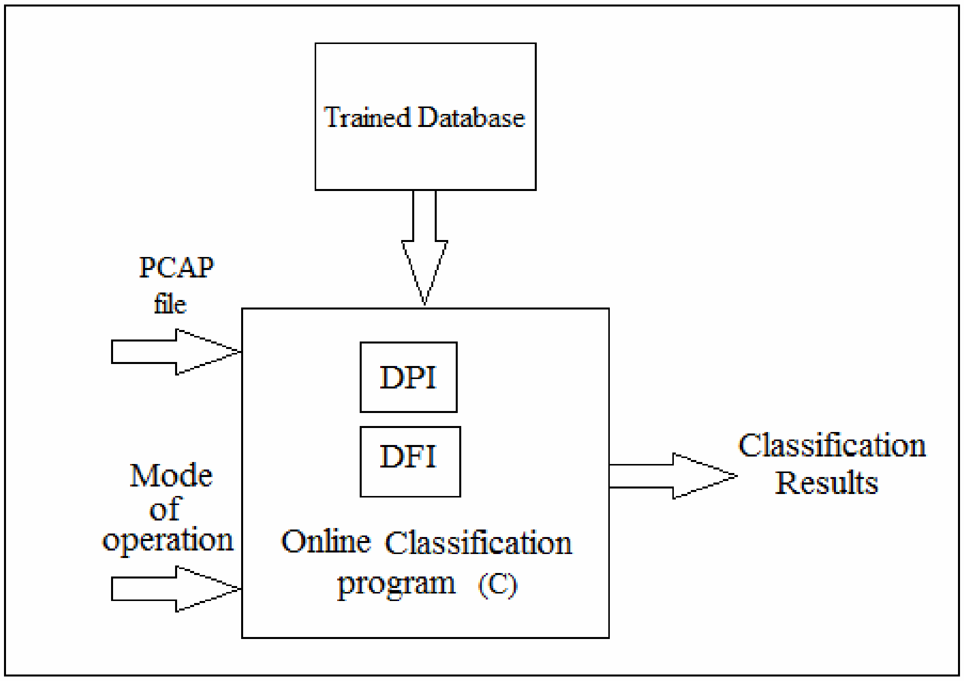
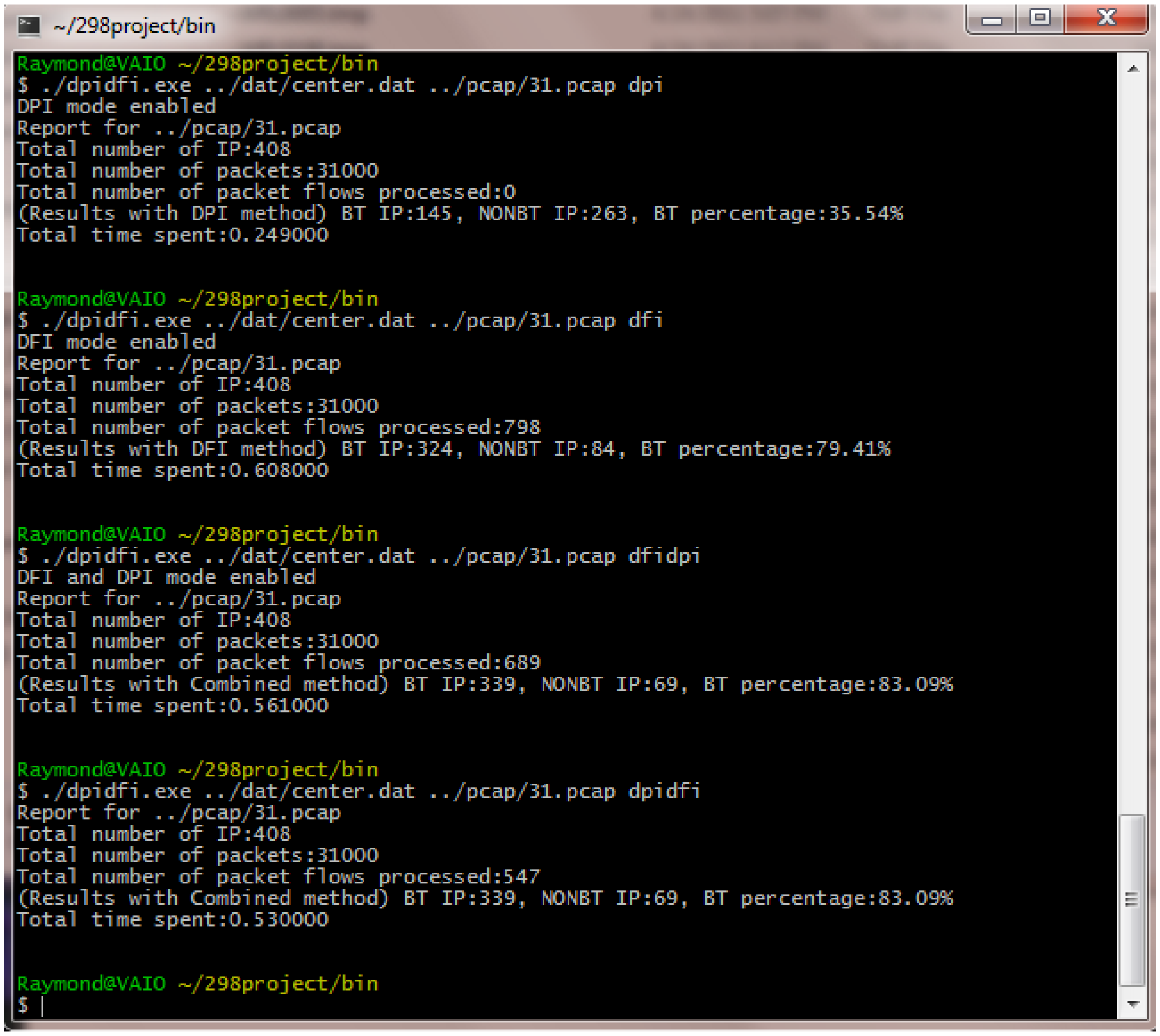


Figure 12: Online classification program block diagram

The DPI/DFI is the proposed method in this project. Like DFI/DPI approach, it utilizes more than one technique to classify packets. At first, an unknown class of packet flows will be inputted to the DPI module in which it can be determined if the packet is a BT packet. If it is a BT packet, database of BT hosts will be updated immediately. Otherwise, based on the packet information, the corresponding packet flow information will be updated (i.e. number of packet in flow, average packet size, etc) in the packet flow database. If that packet is at the end of a flow, we will update the flow information, and it will be applied to the DFI classification to determine if the given flow is from BT packets. The advantage of this approach will be discussed in the next section.

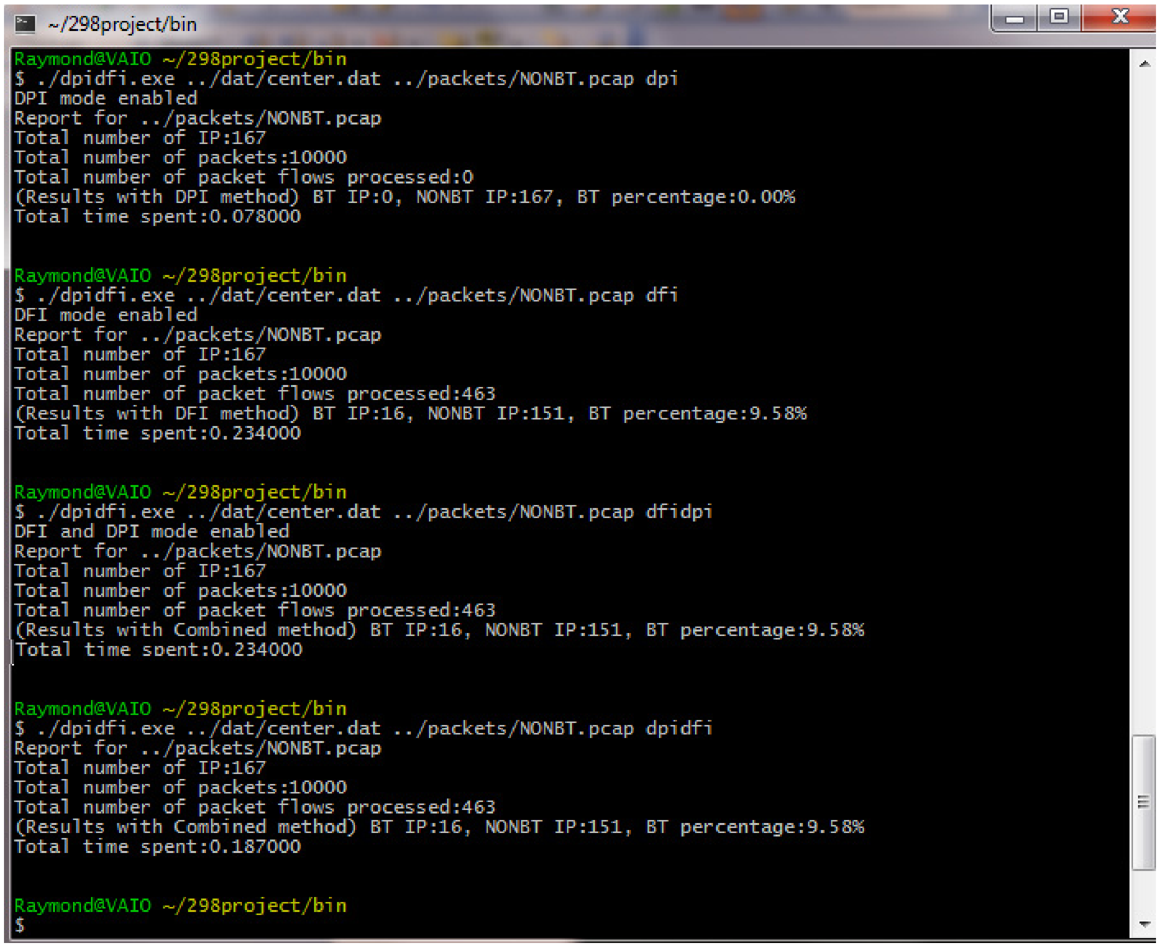
Figure 13 shows an example run of the online classification program for four different modes of operation with a sample BT PCAP file. As we can see from the sample run, there are 408 BT IP addresses in the file. The DPI method is able to detect 35.54% of BT IP; The DFI method is able to detect 79.41%. The more advanced methods are able to provide highest detection rate (i.e. Both DFI/DPI and DPI/DFI are able to detect 83.09%).



```
Raymond@VAIO ~/298project/bin
$ ./dpidfi.exe ../dat/center.dat ../pcap/31.pcap dpi
DPI mode enabled
Report for ../pcap/31.pcap
Total number of IP:408
Total number of packets:31000
Total number of packet flows processed:0
(Results with DPI method) BT IP:145, NONBT IP:263, BT percentage:35.54%
Total time spent:0.249000


Raymond@VAIO ~/298project/bin
$ ./dpidfi.exe ../dat/center.dat ../pcap/31.pcap dfi
DFI mode enabled
Report for ../pcap/31.pcap
Total number of IP:408
Total number of packets:31000
Total number of packet flows processed:798
(Results with DFI method) BT IP:324, NONBT IP:84, BT percentage:79.41%
Total time spent:0.608000


Raymond@VAIO ~/298project/bin
$ ./dpidfi.exe ../dat/center.dat ../pcap/31.pcap dfidpi
DFI and DPI mode enabled
Report for ../pcap/31.pcap
Total number of IP:408
Total number of packets:31000
Total number of packet flows processed:689
(Results with Combined method) BT IP:339, NONBT IP:69, BT percentage:83.09%
Total time spent:0.561000


Raymond@VAIO ~/298project/bin
$ ./dpidfi.exe ../dat/center.dat ../pcap/31.pcap dpidfi
Report for ../pcap/31.pcap
Total number of IP:408
Total number of packets:31000
Total number of packet flows processed:547
(Results with Combined method) BT IP:339, NONBT IP:69, BT percentage:83.09%
Total time spent:0.530000


Raymond@VAIO ~/298project/bin
$ |
```

Figure 13: Four different modes of classification with a sample BT PCAP file

By the same token, an example run of the online classification program for four different modes of operation with a sample Non-BT PCAP file was also shown. As we can see from Figure 14, there are 408 BT IP addresses in the file. The DPI method is able to detect 0% of BT IP; The DFI, DFI/DPI and DPI/DFI methods all have a false alarm rate of 9.58%.



```
~/298project/bin

Raymond@VAIO ~/298project/bin
$ ./dpidfi.exe ../dat/center.dat ../packets/NONBT.pcap dpi
DPI mode enabled
Report for ../packets/NONBT.pcap
Total number of IP:167
Total number of packets:10000
Total number of packet flows processed:0
(Results with DPI method) BT IP:0, NONBT IP:167, BT percentage:0.00%
Total time spent:0.078000

Raymond@VAIO ~/298project/bin
$ ./dpidfi.exe ../dat/center.dat ../packets/NONBT.pcap dfi
DFI mode enabled
Report for ../packets/NONBT.pcap
Total number of IP:167
Total number of packets:10000
Total number of packet flows processed:463
(Results with DFI method) BT IP:16, NONBT IP:151, BT percentage:9.58%
Total time spent:0.234000

Raymond@VAIO ~/298project/bin
$ ./dpidfi.exe ../dat/center.dat ../packets/NONBT.pcap dfidpi
DFI and DPI mode enabled
Report for ../packets/NONBT.pcap
Total number of IP:167
Total number of packets:10000
Total number of packet flows processed:463
(Results with Combined method) BT IP:16, NONBT IP:151, BT percentage:9.58%
Total time spent:0.234000

Raymond@VAIO ~/298project/bin
$ ./dpidfi.exe ../dat/center.dat ../packets/NONBT.pcap dpidfi
Report for ../packets/NONBT.pcap
Total number of IP:167
Total number of packets:10000
Total number of packet flows processed:463
(Results with Combined method) BT IP:16, NONBT IP:151, BT percentage:9.58%
Total time spent:0.187000

Raymond@VAIO ~/298project/bin
$
```

Figure 14: Four different modes of classification with a sample NON-BT PCAP file

Chapter 5: Experiment Results

5.1 Comparison with existing approaches

In this section we will compare the four common approaches to our proposed system. A comparison summary can be found in table III. These common approaches are based on DPI and/or DFI techniques with learning algorithms. The first one detects the handshaking messages pattern between BT peers. As discussed in section II, a string ("BitTorrentprotocol") can be found within the handshaking packets. This pattern can be used to determine whether a BT client is currently running in a network. This approach is simple and is able to provide acceptable accuracy results. However, the major drawback of this approach is that only non-encrypted packets are supported. In addition, it will also lead to privacy issue as examination of packets payload is needed.

The second major approach is based on the length statistics of the packet/flow length. Since this method bases upon the packet length, encrypted packets can also be supported. The philosophy behind this method is that BT and non-BT packets have different packet length distribution. Therefore, only thresholds are needed to determine if the packet is a BT packet. One of the major problems of this approach is that there are some non-BT packets also having the same length characteristics of BT packets. Also, there is no rule of thumb to select what length should be used. Hence, it is often difficult to determine the right threshold values.

The third approach is based on the learning algorithms such as K-means, SVM and neural network. It uses classification algorithms to train a database from the packet/flow characteristics. Examples of the features used including average packet size, total number of packet and inter packets arrival time. One problem with this approach is that prior training is required. Usually, it will take considerable amount of time to train a classifier. Thus, it is difficult to combine both offline training and online classification parts into an adaptive training system (i.e. using the classifier while still continue training the classifier)

The fourth common approach utilizes both DFI and DPI to identify BT packets. DFI is first used to classify whether packets are P2P; then DPI is followed to determine if the P2P packets are BT type. The major disadvantage of this approach is that it may be difficult to implement under a live network due to the fact that DFI is per flow based while DPI is per packet based. Thus, we will need to wait for the end of a packet flow in order to perform DFI classification while the BT pattern can only be seen in the handshaking packets which are only appearing at the beginning of a flow.

Finally, as mentioned in the previous section, the proposed approach also utilizes both DFI and DPI. Instead of performing DFI in the first step; the DPI is used first to determine if a packet is a BT packet, DFI is then followed if the BT flow pattern can not be found. The major advantage compare to the combined method mentioned above is that we don't need to wait for the whole flow to determine the packet type if the BT pattern is found in the DPI stage. In other words, we can quickly identify the packet flow is a BT flow without waiting for the whole flow to finish.

Table III: Comparison of major approaches for detecting BT packets

Methods	Schemes	Strengths	Limitations
BT header lookup method[10]	DPI	Simple	Not Working for encrypted packets
Packet/ Flow length statistics method [9][2]	DFI	Work even with encrypted packets	It is difficult to determine the thresholds
Learning algorithm [6]	DFI	Packet classification is based on a trained database	Prior training required
Combined method [12]	DFI and DPI	Higher accuracy compare to simple DFI or DPI approaches	It is difficult to determine the thresholds
Our proposed system	DFI and DPI	1) Higher accuracy compare to simple DFI or DPI approaches 2) Easier for implementation compared to combined method above 3) better running time efficiency	Prior training required

5.2 Network topology

The network setup that I used for performing the experiments is shown in figure 15. For the sake of the simplicity, there is only one PC behind a router. However, this minimal setup will not affect the experiment results as the main goal of this setup is to capture BT and non-BT packets. Inside the PC, BitComet 1.21 was installed as the BT client. A sample torrent file was downloaded for BT packets capturing purpose. As mentioned in section I, torrent file contains the information about the tracker server and the tracker server contains the peers' information about the shared files.

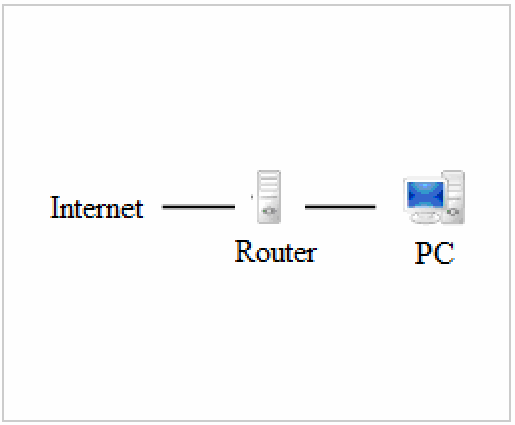


Figure 15: Network topology for experiments

5.3 Performance comparison (DPI vs DFI vs DFI/DPI vs DPI/DFI)
5.3.1 Classification accuracy

Four statistical tests were used to evaluate the classifiers in different modes of operation. They are the true positive rate (TPR), true negative rate (TNR), false positive rate (FPR) and false negative rate (FNR). As we discussed in section 2.4, this performance evaluation metric is very popular as it was used by many authors in their papers [4][3][11][5][7].

In order to test the classifier, the classifier was first trained with 8000 TCP packet flows in which more than 3500 of them are BT TCP packet flows. Table IV-XI show the classification results for DFI, DPI, DFI/DPI and DPI/DFI (proposed algorithm) methods for two simulation tests. These experiments were done with PCAP files with packets type known. The first test PCAP file contained packets with 408 BT IP addresses and 167 non-BT IP addresses and the second test PCAP file contained packets with 686 BT IP addresses and 454 non-BT IP addresses.

Based on the statistical results, the higher the TPR and the TNR, the better the classifier will be. There are a couple of observations from the experiments. The first observation is that the DPI method has 100% accuracy to detect HTTP, FTP and SSH as non-BT protocol. It is because the DPI method searches for the BT pattern string ("BitTorrentprotocol") explicitly inside the packets. Since the non-BT packets rarely have exact BT pattern string in the packets, 100% accuracy was expected. On the other hand, the DPI method is not always able to detect BT packets. It is because the BT pattern string only happens in the handshaking messages and it may not appear during the BT data transfer. Another observation is the methods utilizing both DFI and DPI yield better BT detection accuracy results than using DPI or DFI alone. As a matter of fact, DPI/DFI and DFI/DPI both provide the same accuracy in terms of classification. The reason is due to the fact that the same However, in terms of time efficiency DPI/DFI yield better results than DFI/DPI. We will discuss the performance details in the next section.

Table IV: Classification results with DPI (Test case 1)

DPI (Test Case 1)		
	TPR=31%	FPR=69%
	FNR=0%	TNR=100%

Table V: Classification results with DPI (Test case 2)

DPI (Test Case 2)		
	TPR=35%	FPR=65%
	FNR=0%	TNR=100%

Table VI: Classification results with DFI (Test case 1)

DFI (Test Case 1)		
	TPR=78%	FPR=22%
	FNR=14%	TNR=86%

Table VII: Classification results with DFI (Test case 2)

DFI (Test Case 2)		
	TPR=73%	FPR=27%
	FNR=12.5%	TNR=87.5%

Table VIII: Classification results with DFI/DPI (Test case 1)

DFI/DPI (Test Case 1)		
	TPR=87%	FPR=13%
	FNR=14%	TNR=86%

Table IX: Classification results with DFI/DPI (Test case 2)

DFI/DPI (Test Case 2)		
	TPR=85%	FPR=15%
	FNR=12.5%	TNR=87.5%

Table X: Classification results with DPI/DFI (Test case 1)

DPI/DFI (Test Case 1)		
	TPR=87%	FPR=13%
	FNR=14%	TNR=86%

Table XI: Classification results with DPI/DFI (Test case 2)

DPI/DFI (Test Case 2)		
	TPR=85%	FPR=15%
	FNR=12.5%	TNR=87.5%

5.3.2 Speed performance comparison

Figure 16 shows the packets classification time for various classification methods. Note that the DPI has the fastest classification time due to the classification is purely based on the string comparison (i.e. other methods are more computation intensive relatively). This method is fast; however, the accuracy would be low (compared to other methods) because the BT pattern string can only be found in the handshaking BT packets and we may not see this pattern often during large data transfer.

In the previous section, we showed that the classification accuracy of the DFI method follow by the DPI method (DFI/DPI) and the DPI method follow by the DFI method (DPI/DFI) is the same. However, there are some subtle differences. These subtle differences include resources usage and performance. Both the speed and memory usage appear to be faster and better in the DPI/DFI method. The reason is as follows. First notices that DFI is a flow based classification, it requires the whole packet flow is completed before any classification can be done. On the other hand, the DPI is a packet based classification; it only takes one packet in order to determine the packet class (i.e. BT or non-BT). Given that, assuming in a packet flow and one of its packets does contain the BT string identifier, if we perform the DPI/DFI method, we can quickly identify the packet flow is a BT flow without waiting for the whole flow. The advantage of this approach is that we can identify some of the packet flows class earlier. Once we determine the class of a packet flow, we do not need to keep track of the packet flow's information. Based on the experiments conducted, it appears that DPI/DFI method is about 15%-20% faster DFI/DPI method.

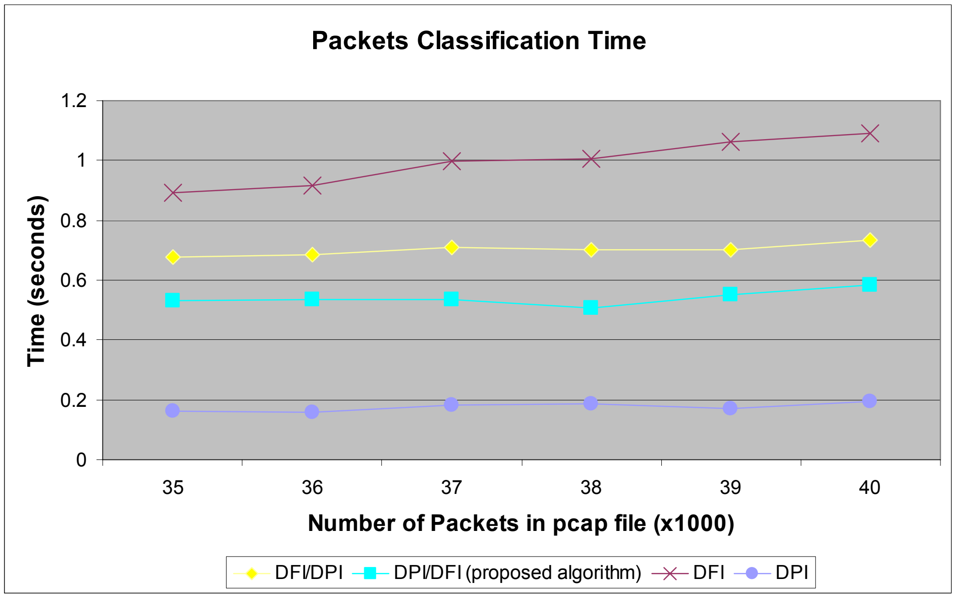


Figure 16: Packets Classification Execution Time experiment

Chapter 6: Conclusion

A paper with the project results from this project has submitted to a conference. The promising simulation results show that by combining multiple techniques such as DPI, DFI and learning algorithms, the detection rate of the BT packets can increase significantly. In addition, applying the correct order of these techniques can further increase the execution speed. In terms of the future work of this project, the ultimate goal would be applying the proposed algorithm into a live network situation.

Appendix A1: capture_bitTorrent.c

```
#include <stdlib.h>
#include "pcap.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "util.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);

static int maxpacket=100;
static int count = 1;                   /* packet counter */
static unsigned long long total_size_payload=0;

int main(int argc, char **argv)
{
        pcap_if_t *alldevs;
        pcap_if_t *d;
        int inum;
        int i=0;
        pcap_t *adhandle;
        char errbuf[PCAP_ERRBUF_SIZE];
        pcap_dumper_t *dumpfile;

        if (argc != 3) {
          printf("usage: %s maxpacket pcapname\n", argv[0]);
          return -1;
        }
        maxpacket = atoi(argv[1]);

        /* Retrieve the device list */
        if(pcap_findalldevs(&alldevs, errbuf) == -1) {
          fprintf(stderr,"Error in pcap_findalldevs: %s\n", errbuf);
          exit(1);
        }

        /* Print the list */
        for(d=alldevs; d; d=d->next) {
          printf("%d. %s", ++i, d->name);
          if (d->description)
            printf(" (%s)\n", d->description);
          else
            printf(" (No description available)\n");
        }

        if(i==0) {
          printf("\nNo interfaces found! Make sure WinPcap is installed.\n");
          return -1;
        }
        printf("Enter the interface number (1-%d):",i);
        scanf("%d", &inum);
        //        inum = 2;

        if(inum < 1 || inum > i) {
          printf("\nInterface number out of range.\n");
          /* Free the device list */
          pcap_freealldevs(alldevs);
          return -1;
        }

        printf("\nlistening on %s...\n", d->description);

      /* At this point, we don't need any more the device list. Free it */
        pcap_freealldevs(alldevs);

        /* start the capture */
        pcap_loop(adhandle, 0, got_packet, (unsigned char *)dumpfile);
        pcap_close(adhandle);
        return 0;
}
```

Appendix A1(Cont.): capture_bitTorrent.c

```
void got_packet(u_char *dumpfile, const struct pcap_pkthdr *header, const u_char *packet)
{

  /* declare pointers to packet headers */
  const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
  const struct sniff_ip *ip;              /* The IP header */
  const struct sniff_tcp *tcp;            /* The TCP header */

  int size_ip;
  int size_tcp;
  int size_payload;


  if (count-1==maxpacket) { /* done if we hit max packet */
    printf("Total number of packets captured:%d\n", count-1);
    printf("Total packet size captured:%llu\n", total_size_payload);
    exit(0);
  }

  /* define ethernet header */
  ethernet = (struct sniff_ethernet*)(packet);

  /* define/compute ip header offset */
  ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
  size_ip = IP_HL(ip)*4;
  if (size_ip < 20) /* Invalid IP header length */
    return;

  /* determine protocol */
  switch(ip->ip_p) {
  case IPPROTO_TCP:
    break;
  case IPPROTO_UDP:
  case IPPROTO_ICMP:
  case IPPROTO_IP:
  default:
    return;
  }

  /* OK, this packet is TCP.  */

  /* define/compute tcp header offset */
  tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
  size_tcp = TH_OFF(tcp)*4;
  if (size_tcp < 20) {
    return;
  }

  /* In order to capture BT packets only, I have setup the following
     1) My BT client uses TCP port 1200 only (set in BT client software)
     2) My BT client IP is 192.168.3.173
     The filter below is used to filter non-BT packets
   */
  //  if (((strcmp(inet_ntoa(ip->ip_src), "192.168.3.173") ==0) && (ntohs(tcp->th_sport) ==
1200)) ||
  //     ((strcmp(inet_ntoa(ip->ip_dst), "192.168.3.173") ==0) && (ntohs(tcp->th_dport) ==
1200))) {

    size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
    if (size_payload <= 0)
      return;

    count++;
    total_size_payload += size_payload;
    pcap_dump(dumpfile, header, packet);

return;
}
```

Appendix A2: genflowdata.c

```
#include "util.h"
#include <stdlib.h>
#include "pcap.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* Global variables */
flow* nptr;
static int count = 0;
static int pcount = 0;                   /* packet counter */

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);
void saveFlowData(char* path) ;

int main(int argc, char **argv)
{
  pcap_t *fp;
  char errbuf[PCAP_ERRBUF_SIZE];
  if(argc != 3) {
    printf("usage: %s pcap_filename flow_data_filename", argv[0]);
    return -1;
  }

  /* Open the capture file */
  if ((fp = pcap_open_offline(argv[1], errbuf)) == NULL) {
    fprintf(stderr,"\nUnable to open the file %s.\n", argv[1]);
    return -1;
  }

  /* allocate space for pkt stat */
  nptr = (flow*)malloc(MAXPACKETS*sizeof(flow));
  bzero(nptr, MAXPACKETS*sizeof(flow));

  /* read and dispatch packets until EOF is reached */
  pcap_loop(fp, 0, got_packet, NULL);

  qsort(nptr, count, sizeof(flow), (int(*)(const void*, const void*))flowCmpNumPkt);

  saveFlowData(argv[2]);

  pcap_close(fp);

  return 0;
}

void saveFlowData(char* path) {

  FILE *fp1, *fp2;
  char fp2name[100];
  sprintf(fp2name, "%s.flow", path);
  fp1 = fopen(path, "w");
  fp2 = fopen(fp2name, "w");

  if (0) { //Raymond
    int i,j;
    for (i=0;i<count;i++) {
      printf("Node :%d\n", i);
      printf("IP1:%s\n", nptr[i].ipAddr1);
      printf("Port1:%u\n", nptr[i].port1);
      printf("IP2:%s\n", nptr[i].ipAddr2);
      printf("Port2:%u\n", nptr[i].port2);
      printf("Total packets:%d\n", nptr[i].numTPkt);

      for (j=0;j<=nptr[i].infonum;j++) {
        printf("FLOW:%d, numPkt:%d, avgSize:%f, totalHeaderSize:%d, totalSize:%d\n",
               j, nptr[i].infoptr[j].numPkt, nptr[i].infoptr[j].avgSize,
nptr[i].infoptr[j].totalHeaderSize, nptr[i].infoptr[j].totalSize);
        printf("numSendPkt:%d, totalSendHeaderSize:%d, totalSendSize:%d\n",
               nptr[i].infoptr[j].numSendPkt, nptr[i].infoptr[j].totalSendHeaderSize,
nptr[i].infoptr[j].totalSendSize);
        printf("numReceivePkt:%d, totalReceiveHeaderSize:%d, totalReceiveSize:%d\n\n",
               nptr[i].infoptr[j].numReceivePkt, nptr[i].infoptr[j].totalReceiveHeaderSize,
nptr[i].infoptr[j].totalReceiveSize);
      }
    }
  } else {
    int i,j;
    for (i=0;i<count;i++) {
      for (j=0;j<=nptr[i].infonum;j++) {
        fprintf(fp1, "%d %d %d %d %f %d %d %d %d %d %d %d %d\n",
               i, nptr[i].numTPkt, j, nptr[i].infoptr[j].numPkt, nptr[i].infoptr[j].avgSize,
nptr[i].infoptr[j].totalHeaderSize,
               nptr[i].infoptr[j].totalSize, nptr[i].infoptr[j].numSendPkt,
nptr[i].infoptr[j].totalSendHeaderSize, nptr[i].infoptr[j].totalSendSize,
               nptr[i].infoptr[j].numReceivePkt, nptr[i].infoptr[j].totalReceiveHeaderSize,
nptr[i].infoptr[j].totalReceiveSize);
        fprintf(fp2, "%s:%u<->%s:%u\n", nptr[i].ipAddr1, nptr[i].port1, nptr[i].ipAddr2,
nptr[i].port2);
      }
    }
  }

  fclose(fp1);
  fclose(fp2);
}
void addFlow(int count, struct in_addr ip1, u_short port1, struct in_addr ip2, u_short port2,
unsigned headsize, unsigned pktsize, int send) {

  char str1[30];
  char str2[30];
  strcpy(str1, inet_ntoa(ip1));
  strcpy(str2, inet_ntoa(ip2));

  nptr[count].infoptr = (info*)malloc(999*sizeof(info));
  bzero(nptr[count].infoptr, 999*sizeof(info));

  strcpy(nptr[count].ipAddr1, str1);
  strcpy(nptr[count].ipAddr2, str2);
  nptr[count].port1 = ntohs(port1);
  nptr[count].port2 = ntohs(port2);

  nptr[count].numTPkt=1;

  nptr[count].infoptr[0].numPkt=1;
  nptr[count].infoptr[0].avgSize = pktsize;
  nptr[count].infoptr[0].totalSize=pktsize;

  if (send==1) {
    nptr[count].infoptr[0].numSendPkt=1;
    nptr[count].infoptr[0].totalSendHeaderSize=headsize;
    nptr[count].infoptr[0].totalSendSize=pktsize;
  } else {
    nptr[count].infoptr[0].numReceivePkt=1;
    nptr[count].infoptr[0].totalReceiveHeaderSize=headsize;
    nptr[count].infoptr[0].totalReceiveSize=pktsize;
  }
  qsort(nptr, count+1, sizeof(flow), (int(*)(const void*, const void*))flowCmp);
}
void
got_packet(u_char *dumpfile, const struct pcap_pkthdr *header, const u_char *packet)
{
  /* declare pointers to packet headers */
  const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
  const struct sniff_ip *ip;              /* The IP header */
  const struct sniff_tcp *tcp;            /* The TCP header */
  //  const u_char *payload;                    /* Packet payload */

  int size_ip;
  int size_tcp;
  int size_payload;

  ethernet = (struct sniff_ethernet*)(packet);

  /* define/compute ip header offset */
  ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
  size_ip = IP_HL(ip)*4;
  if (size_ip < 20)   // Invalid IP header length
    return;

  /* determine protocol */
  switch(ip->ip_p) {
   case IPPROTO_TCP:
    break;
  case IPPROTO_UDP:
    return;
  case IPPROTO_ICMP:
    return;
  case IPPROTO_IP:
     return;
   default:
    return;
  }
```

```
/*
   *  OK, this packet is TCP.
   */

  /* define/compute tcp header offset */
  tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
  size_tcp = TH_OFF(tcp)*4;
  if (size_tcp < 20)
    return;

  size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);

  flow n;
  strcpy(n.ipAddr1, inet_ntoa(ip->ip_src));   // sender
  strcpy(n.ipAddr2, inet_ntoa(ip->ip_dst));
  n.port1=ntohs(tcp->th_sport);
  n.port2=ntohs(tcp->th_dport);
  pcount++;

  flow* r = bsearch(&n, nptr, count, sizeof(flow), (int(*)(const void*, const void*))flowCmp);
  if (r) {
    if ((tcp->th_flags & TH_FIN) == TH_FIN)   // a flow ends
      r->infonum++;
    r->numTPkt++;
    r->infoptr[r->infonum].avgSize = ((r->infoptr[r->infonum].avgSize * r->infoptr[r-
>infonum].numPkt) + ntohs(ip->ip_len))/(r->infoptr[r->infonum].numPkt+1);
    r->infoptr[r->infonum].numPkt++;
    r->infoptr[r->infonum].totalHeaderSize += size_ip + size_tcp;
    r->infoptr[r->infonum].totalSize += ntohs(ip->ip_len);

    r->infoptr[r->infonum].numSendPkt++;
    r->infoptr[r->infonum].totalSendSize+=ntohs(ip->ip_len);
    r->infoptr[r->infonum].totalSendHeaderSize+= size_ip + size_tcp;
  } else {
    strcpy(n.ipAddr1, inet_ntoa(ip->ip_dst));
    strcpy(n.ipAddr2, inet_ntoa(ip->ip_src));
    n.port1=ntohs(tcp->th_dport);
    n.port2=ntohs(tcp->th_sport);
    r = bsearch(&n, nptr, count, sizeof(flow), (int(*)(const void*, const void*))flowCmp);
    if (r) {
      if ((tcp->th_flags & TH_FIN) == TH_FIN)
        r->infonum++;
      r->numTPkt++;
      r->infoptr[r->infonum].avgSize = ((r->infoptr[r->infonum].avgSize * r->infoptr[r-
>infonum].numPkt) + ntohs(ip->ip_len))/(r->infoptr[r->infonum].numPkt+1);
      r->infoptr[r->infonum].numPkt++;
      r->infoptr[r->infonum].totalHeaderSize += size_ip + size_tcp;
      r->infoptr[r->infonum].totalSize += ntohs(ip->ip_len);
      r->infoptr[r->infonum].numReceivePkt++;
      r->infoptr[r->infonum].totalReceiveSize+=ntohs(ip->ip_len);
      r->infoptr[r->infonum].totalReceiveHeaderSize+= size_ip + size_tcp;
    } else {
      addFlow(count, ip->ip_src ,tcp->th_sport ,ip->ip_dst, tcp->th_dport, size_ip + size_tcp,
ntohs(ip->ip_len), 1);
      count++;
    }
  }
return;
}
```

Appendix A3: classifier.m

```
clear all; warning off;
NUM_CLUSTER = 400;  STARTF=4; FIELD=13;

a=fopen('./data/nonBT_1.dat', 'r');
s=fscanf(a, '%d %d %d %d %f %d %d %d %d %d %d %d %d');
n=size(s,1)/FIELD;
ss= reshape(s, FIELD, n)';
sss0=ss(:,STARTF:FIELD);
sss0=[sss0 ones(n,1)];     % label 1 = nonBT
fclose(a);

a=fopen('./data/nonBT_2.dat', 'r');
s=fscanf(a, '%d %d %d %d %f %d %d %d %d %d %d %d %d');
n=size(s,1)/FIELD;
ss= reshape(s, FIELD, n)';
sss1=ss(:,STARTF:FIELD);
sss1=[sss1 ones(n,1)];
fclose(a);

a=fopen('./data/BT_1.dat', 'r');
s=fscanf(a, '%d %d %d %d %f %d %d %d %d %d %d %d %d');
n=size(s,1)/FIELD;
ss= reshape(s, FIELD, n)';
sss2=ss(:,4:FIELD);
sss2=[sss2 2*ones(n,1)];    % label 2 = BT
fclose(a);

a=fopen('./data/BT_2.dat', 'r');
s=fscanf(a, '%d %d %d %d %f %d %d %d %d %d %d %d %d');
n=size(s,1)/FIELD;
ss= reshape(s, FIELD, n)';
sss3=ss(:,STARTF:FIELD);
sss3=[sss3 2*ones(n,1)];
fclose(a);

a=fopen('./data/ssh_pcap.dat', 'r');
s=fscanf(a, '%d %d %d %d %f %d %d %d %d %d %d %d %d');
n=size(s,1)/FIELD;
ss= reshape(s, FIELD, n)';
sss4=ss(:,STARTF:FIELD);
sss4=[sss4 1*ones(n,1)];
fclose(a);

a=fopen('./data/ssh_pcap_2.dat', 'r');
s=fscanf(a, '%d %d %d %d %f %d %d %d %d %d %d %d %d');
n=size(s,1)/FIELD;
ss= reshape(s, FIELD, n)';
sss5=ss(:,STARTF:FIELD);
sss5=[sss5 1*ones(n,1)];
fclose(a);
s_with_label = [sss0; sss1; sss2; sss3; sss4; sss5];

fprintf('There are %d BT flow\n', size(find(s_with_label(:,11)==2),1));
fprintf('There are %d nonBT flow\n', size(find(s_with_label(:,11)==1),1));
fprintf('Running K-means to train the classifier now (K=%d)...\n', NUM_CLUSTER);

s_without_label=s_with_label(:,1:10);
[idx C]= kmeans(s_without_label, NUM_CLUSTER, 'EmptyAction', 'drop');

for i=1:NUM_CLUSTER
        id=find(idx==i);
        marked_P2P = size(find(s_with_label(id,11)==2),1);
        marked_nonP2P = size(find(s_with_label(id,11)==1),1);

        if (marked_P2P ==0 && marked_nonP2P ==0)
               cluster(i) =0;
               conf(i)=-1;
        else
           if (marked_P2P > marked_nonP2P)
                cluster(i)= 2;
                conf(i)=marked_P2P/(marked_P2P+marked_nonP2P);
           else
                cluster(i)= 1;
                conf(i)=marked_nonP2P/(marked_P2P+marked_nonP2P);
           end
        end
end
P2Pclusters = size(find(cluster==2),2);
nonP2Pclusters = size(find(cluster==1),2);
fprintf('BT clusters number:%d, percentage:%f\n', P2Pclusters ,P2Pclusters/NUM_CLUSTER);
fprintf('NonBT clusters number:%d, percentage:%f\n', nonP2Pclusters,
nonP2Pclusters/NUM_CLUSTER);
fprintf('Empty clusters number:%d, percentage:%f\n', NUM_CLUSTER-P2Pclusters-nonP2Pclusters,
(NUM_CLUSTER-P2Pclusters-nonP2Pclusters)/NUM_CLUSTER);
fprintf('Done trainning...\n');
fprintf('Saving the classifier...\n');
save classifier
```

Appendix A4: dpi.c

```
#include <stdlib.h>
#include "pcap.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "util.h"

/* prototype of the packet handler */
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);

int main(int argc, char **argv)
{
  pcap_t *fp;
  char errbuf[PCAP_ERRBUF_SIZE];
  if(argc != 2) {
    printf("usage: %s filename", argv[0]);
    return -1;
  }

  /* Open the capture file */
  if ((fp = pcap_open_offline(argv[1], errbuf)) == NULL) {
    fprintf(stderr,"\nUnable to open the file %s.\n", argv[1]);
    return -1;
  }

  /* read and dispatch packets until EOF is reached */
  pcap_loop(fp, 0, got_packet, NULL);
  pcap_close(fp);
  return 0;
}

void got_packet(u_char *dumpfile, const struct pcap_pkthdr *header, const u_char *packet)
{
  /* declare pointers to packet headers */
  const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
  const struct sniff_ip *ip;              /* The IP header */
  const struct sniff_tcp *tcp;            /* The TCP header */
  const u_char *payload;                  /* Packet payload */

  static int counter=0;
  int size_ip;
  int size_tcp;
  int size_payload;

  /* define ethernet header */
  ethernet = (struct sniff_ethernet*)(packet);

  /* define/compute ip header offset */
  ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
  size_ip = IP_HL(ip)*4;

  if (size_ip < 20)   // Invalid IP header length
    return;

  if (ip->ip_p != IPPROTO_TCP)  // IPPROTO_TCP, IPPROTO_UDP, IPPROTO_ICMP, IPPROTO_IP
    return;

  /* define/compute tcp header offset */
  tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
  size_tcp = TH_OFF(tcp)*4;
  if (size_tcp < 20) // Invalid TCP header length
    return;

  size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
  if (size_payload <= 0) // Invalid payload length
    return;

  /* define/compute tcp payload (segment) offset */
   payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);

  if (strncmp((char*)payload, "\023BitTorrent protocolex", 22)==0) {
    counter++;
    printf("(%d) %s:%d<==>", counter, inet_ntoa(ip->ip_src), ntohs(tcp->th_sport));
    printf("%s:%d\n", inet_ntoa(ip->ip_dst), ntohs(tcp->th_dport));
    print_payload(payload, size_payload);
  }
  return;
}
```

Appendix A5: dfi.c

```
#include "util.h"
#include <stdlib.h>
#include "pcap.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <assert.h>
#include <math.h>

/* Global variables */
flow* nptr;
btip* ipptr;
static int ipcount=0;
static int count = 0;
static int pcount = 0;                    /* packet counter */
const u_char *payload = 0;                     /* Packet payload */
float C[400][10];
int CCLASS[400];
int NUMC=0;
double startcputime;
char *argv1;

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);
void reportData() ;
void readData(FILE* fp)
{
  int i=0;
  while(fscanf(fp, "%f %f %f %f %f %f %f %f %f %f %d\n",
               &C[i][0], &C[i][1], &C[i][2], &C[i][3], &C[i][4],
               &C[i][5], &C[i][6], &C[i][7], &C[i][8], &C[i][9],
               &CCLASS[i])!=EOF)
    {
      i++;
      assert(i<400);
    }
  NUMC = i-1;
}
int main(int argc, char **argv)
{
  pcap_t *fp;
  FILE *fp1;

  startcputime=sysGetCpuTime();

  char errbuf[PCAP_ERRBUF_SIZE];
  if(argc != 3) {
    printf("usage: %s pcap_filename cluster_center_data_base", argv[0]);
    return -1;
  }

  /* Open the capture file */
  if ((fp = pcap_open_offline(argv[1], errbuf)) == NULL) {
    fprintf(stderr,"\nUnable to open the file %s.\n", argv[1]);
    return -1;
  }
  argv1=argv[1];

  /* Open the center database */
  if ((fp1 = fopen(argv[2], "r")) == NULL) {
    fprintf(stderr,"\nUnable to open the file %s.\n", argv[2]);
    return -1;
  }
  readData(fp1);

  /* allocate space for pkt stat */
  nptr = (flow*)malloc(MAXPACKETS*sizeof(flow));
  bzero(nptr, MAXPACKETS*sizeof(flow));

  /* allocate space for btip stat */
  ipptr = (btip*)malloc(2*MAXPACKETS*sizeof(btip));
   bzero(ipptr, MAXPACKETS*sizeof(btip));

  /* read and dispatch packets until EOF is reached */
  pcap_loop(fp, 0, got_packet, NULL);
  qsort(nptr, count, sizeof(flow), (int(*)(const void*, const void*))flowCmpNumPkt);
  reportData();

  pcap_close(fp);

  return 0;
}
```

Appendix A5 (Cont.): dfi.c

```
void reportData() {
  int i;
  int btcount=0;
  int btdficount=0;
  double endcputime=sysGetCpuTime();
  for (i=0;i<ipcount;i++) {
    if (ipptr[i].bt_dfi == BT || ipptr[i].bt_dpi == BT) {
      btcount++;
    }
    if (ipptr[i].bt_dfi == BT) {
      btdficount++;
    }
  }
  printf("Report for %s\n", argv1);
  printf("Total number of IP:%d\n", ipcount);
  printf("(Results with DFI method) BT IP:%d, NONBT IP:%d\n", btdficount, ipcount-btdficount);
  printf("Total time spent:%f\n\n", endcputime - startcputime);
}


int dfi_class(float dp[10])
{
  float dist, mindist;
  int i, minindex;
  for (i=0;i<NUMC;i++) {
    dist = (C[i][0]-dp[0])*(C[i][0]-dp[0]) + (C[i][1]-dp[1])*(C[i][1]-dp[1]) + (C[i][2]-
dp[2])*(C[i][2]-dp[2]) +
      (C[i][3]-dp[3])*(C[i][3]-dp[3]) + (C[i][4]-dp[4])*(C[i][4]-dp[4]) + (C[i][5]-
dp[5])*(C[i][5]-dp[5]) +
      (C[i][6]-dp[6])*(C[i][6]-dp[6]) + (C[i][7]-dp[7])*(C[i][7]-dp[7]) + (C[i][8]-
dp[8])*(C[i][8]-dp[8]) +
      (C[i][9]-dp[9])*(C[i][9]-dp[9]) ;
    if (i==0) {
      mindist = dist;
      minindex = 0;
    }
    else if (mindist > dist) {
      mindist = dist;
      minindex = i;
    }
  }
  return CCLASS[minindex];  // BT=2, NONBT=1
}
void addIp(char *ipadd, int bt_dfi, int bt_dpi)
{
  btip n;
  strcpy(n.ipAddr, ipadd);
  btip* r = bsearch(&n, ipptr, ipcount, sizeof(btip), (int(*)(const void*, const
void*))ipCmp);
  if (r) {
    if (r->bt_dfi !=BT)
      r->bt_dfi = bt_dfi;
    if (r->bt_dpi !=BT)
      r->bt_dpi = bt_dpi;

  } else {
    strcpy(ipptr[ipcount].ipAddr, ipadd);
    ipptr[ipcount].bt_dfi = bt_dfi;
    ipptr[ipcount].bt_dpi = bt_dpi;
    qsort(ipptr, ipcount+1, sizeof(btip), (int(*)(const void*, const void*))ipCmp);

    ipcount++;
  }
}
void addFlow(int count, struct in_addr ip1, u_short port1, struct in_addr ip2, u_short port2,
unsigned headsize, unsigned pktsize, int send) {

  char str1[30];
  char str2[30];
  float dp[10];

  strcpy(str1, inet_ntoa(ip1));
  strcpy(str2, inet_ntoa(ip2));

  nptr[count].infoptr = (info*)malloc(999*sizeof(info));
  bzero(nptr[count].infoptr, 999*sizeof(info));

  strcpy(nptr[count].ipAddr1, str1);
  strcpy(nptr[count].ipAddr2, str2);
  nptr[count].port1 = ntohs(port1);
  nptr[count].port2 = ntohs(port2);

  nptr[count].numTPkt=1;
```

Appendix A5 (Cont.): dfi.c

```
    nptr[count].infoptr[0].numPkt=1;
    nptr[count].infoptr[0].avgSize = pktsize;
    nptr[count].infoptr[0].totalSize=pktsize;

    dp[0]=nptr[count].infoptr[0].numPkt;
    dp[1]=nptr[count].infoptr[0].avgSize;
    dp[2]=nptr[count].infoptr[0].totalHeaderSize;
    dp[3]=nptr[count].infoptr[0].totalSize;
    dp[4]=nptr[count].infoptr[0].numSendPkt;
    dp[5]=nptr[count].infoptr[0].totalSendHeaderSize;
    dp[6]=nptr[count].infoptr[0].totalSendSize;
    dp[7]=nptr[count].infoptr[0].numReceivePkt;
    dp[8]=nptr[count].infoptr[0].totalReceiveHeaderSize;
    dp[9]=nptr[count].infoptr[0].totalReceiveSize;
    nptr[count].infoptr[0].bt_dfi = dfi_class(dp);

    if (send==1) {
      nptr[count].infoptr[0].numSendPkt=1;
      nptr[count].infoptr[0].totalSendHeaderSize=headsize;
      nptr[count].infoptr[0].totalSendSize=pktsize;
    } else {
      nptr[count].infoptr[0].numReceivePkt=1;
      nptr[count].infoptr[0].totalReceiveHeaderSize=headsize;
      nptr[count].infoptr[0].totalReceiveSize=pktsize;
    }

    addIp(str1, nptr[count].infoptr[0].bt_dfi, nptr[count].infoptr[0].bt_dpi_flow);
    addIp(str2, nptr[count].infoptr[0].bt_dfi, nptr[count].infoptr[0].bt_dpi_flow);

    qsort(nptr, count+1, sizeof(flow), (int(*)(const void*, const void*))flowCmp);
  }

  void
  got_packet(u_char *dumpfile, const struct pcap_pkthdr *header, const u_char *packet)
  {
    float dp[10];
    /* declare pointers to packet headers */
    const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
    const struct sniff_ip *ip;              /* The IP header */
    const struct sniff_tcp *tcp;            /* The TCP header */

    int size_ip;
    int size_tcp;
    int size_payload;

    ethernet = (struct sniff_ethernet*)(packet);

    /* define/compute ip header offset */
    ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
    size_ip = IP_HL(ip)*4;
    if (size_ip < 20)   // Invalid IP header length
      return;

    /* determine protocol */
    switch(ip->ip_p) {
    case IPPROTO_TCP:
      break;
    case IPPROTO_UDP:
      return;
    case IPPROTO_ICMP:
      return;
    case IPPROTO_IP:
      return;
    default:
      return;
    }
    /*
     *  OK, this packet is TCP.
     */

    /* define/compute tcp header offset */
    tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
    size_tcp = TH_OFF(tcp)*4;
    if (size_tcp < 20)
      return;

    size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
    /* define/compute tcp payload (segment) offset */
    payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);

    flow n;
    strcpy(n.ipAddr1, inet_ntoa(ip->ip_src));   // sender
    strcpy(n.ipAddr2, inet_ntoa(ip->ip_dst));
    n.port1=ntohs(tcp->th_sport);
    n.port2=ntohs(tcp->th_dport);
    pcount++;
```

Appendix A5 (Cont.): dfi.c

```
  flow* r = bsearch(&n, nptr, count, sizeof(flow), (int(*)(const void*, const void*))flowCmp);
  if (r) {
    r->numTPkt++;
    r->infoptr[r->infonum].avgSize = ((r->infoptr[r->infonum].avgSize * r->infoptr[r-
>infonum].numPkt) + ntohs(ip->ip_len))/(r->infoptr[r->infonum].numPkt+1);
    r->infoptr[r->infonum].numPkt++;
    r->infoptr[r->infonum].totalHeaderSize += size_ip + size_tcp;
    r->infoptr[r->infonum].totalSize += ntohs(ip->ip_len);
    r->infoptr[r->infonum].numSendPkt++;
    r->infoptr[r->infonum].totalSendSize+=ntohs(ip->ip_len);
    r->infoptr[r->infonum].totalSendHeaderSize+= size_ip + size_tcp;

    dp[0]=r->infoptr[r->infonum].numPkt;
    dp[1]=r->infoptr[r->infonum].avgSize;
    dp[2]=r->infoptr[r->infonum].totalHeaderSize;
    dp[3]=r->infoptr[r->infonum].totalSize;
    dp[4]=r->infoptr[r->infonum].numSendPkt;
    dp[5]=r->infoptr[r->infonum].totalSendHeaderSize;
    dp[6]=r->infoptr[r->infonum].totalSendSize;
    dp[7]=r->infoptr[r->infonum].numReceivePkt;
    dp[8]=r->infoptr[r->infonum].totalReceiveHeaderSize;
    dp[9]=r->infoptr[r->infonum].totalReceiveSize;
    r->infoptr[r->infonum].bt_dfi = dfi_class(dp);

    addIp(r->ipAddr1, r->infoptr[r->infonum].bt_dfi, r->infoptr[r->infonum].bt_dpi_packet);
    addIp(r->ipAddr2, r->infoptr[r->infonum].bt_dfi, r->infoptr[r->infonum].bt_dpi_packet);

    if ((tcp->th_flags & TH_FIN) == TH_FIN) {   // a flow ends
      r->infonum++;
    }
  } else {
    strcpy(n.ipAddr1, inet_ntoa(ip->ip_dst));
    strcpy(n.ipAddr2, inet_ntoa(ip->ip_src));
    n.port1=ntohs(tcp->th_dport);
    n.port2=ntohs(tcp->th_sport);
    r = bsearch(&n, nptr, count, sizeof(flow), (int(*)(const void*, const void*))flowCmp);
    if (r) {
      r->numTPkt++;
      r->infoptr[r->infonum].avgSize = ((r->infoptr[r->infonum].avgSize * r->infoptr[r-
>infonum].numPkt) + ntohs(ip->ip_len))/(r->infoptr[r->infonum].numPkt+1);
      r->infoptr[r->infonum].numPkt++;
      r->infoptr[r->infonum].totalHeaderSize += size_ip + size_tcp;
      r->infoptr[r->infonum].totalSize += ntohs(ip->ip_len);
      r->infoptr[r->infonum].numReceivePkt++;
      r->infoptr[r->infonum].totalReceiveSize+=ntohs(ip->ip_len);
      r->infoptr[r->infonum].totalReceiveHeaderSize+= size_ip + size_tcp;

      dp[0]=r->infoptr[r->infonum].numPkt;
      dp[1]=r->infoptr[r->infonum].avgSize;
      dp[2]=r->infoptr[r->infonum].totalHeaderSize;
      dp[3]=r->infoptr[r->infonum].totalSize;
      dp[4]=r->infoptr[r->infonum].numSendPkt;
      dp[5]=r->infoptr[r->infonum].totalSendHeaderSize;
      dp[6]=r->infoptr[r->infonum].totalSendSize;
      dp[7]=r->infoptr[r->infonum].numReceivePkt;
      dp[8]=r->infoptr[r->infonum].totalReceiveHeaderSize;
      dp[9]=r->infoptr[r->infonum].totalReceiveSize;
      r->infoptr[r->infonum].bt_dfi = dfi_class(dp);

      addIp(r->ipAddr1, r->infoptr[r->infonum].bt_dfi, r->infoptr[r->infonum].bt_dpi_packet);
      addIp(r->ipAddr2, r->infoptr[r->infonum].bt_dfi, r->infoptr[r->infonum].bt_dpi_packet);

      if ((tcp->th_flags & TH_FIN) == TH_FIN) { // a flow ends
        r->infonum++;
      }
    } else {  // a new flow
      addFlow(count, ip->ip_src ,tcp->th_sport ,ip->ip_dst, tcp->th_dport, size_ip + size_tcp,
ntohs(ip->ip_len), 1);
      count++;
    }
  }


return;
}
```

Appendix A6: util.c

```
#include "util.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <sys/times.h>
#include <limits.h>
#include "unistd.h"


int flowCmp(const void* a, const void* b){
  char tmp1[100], tmp2[100];
  flow* na = (flow*) a;
  flow* nb = (flow*) b;
  sprintf(tmp1, "%s.%d.%s.%d", na->ipAddr1, na->port1, na->ipAddr2, na->port2);
  sprintf(tmp2, "%s.%d.%s.%d", nb->ipAddr1, nb->port1, nb->ipAddr2, nb->port2);

  return strcmp(tmp1, tmp2);

}

int ipCmp(const void* a, const void* b){
  btip* na = (btip*) a;
  btip* nb = (btip*) b;
  return strcmp(na->ipAddr, nb->ipAddr);
}

int flowCmpNumPkt(const void* a, const void* b){
  flow* na = (flow*) a;
  flow* nb = (flow*) b;
  if (na->numTPkt < nb->numTPkt)
    return 1;
  if (na->numTPkt > nb->numTPkt)
    return -1;
  return 0;
}

Void print_hex_ascii_line(const u_char *payload, int len, int offset)
{

        int i;
        int gap;
        const u_char *ch;

        /* offset */
        printf("%05d   ", offset);

        /* hex */
        ch = payload;
        for(i = 0; i < len; i++) {
                printf("%02x ", *ch);
                ch++;
                /* print extra space after 8th byte for visual aid */
                if (i == 7)
                        printf(" ");
        }
        /* print space to handle line less than 8 bytes */
        if (len < 8)
                printf(" ");

        /* fill hex gap with spaces if not full line */
        if (len < 16) {
                gap = 16 - len;
                for (i = 0; i < gap; i++) {
                        printf("   ");
                }
        }
        printf("   ");

        /* ascii (if printable) */
        ch = payload;
        for(i = 0; i < len; i++) {
                if (isprint(*ch))
                        printf("%c", *ch);
                else
                        printf(".");
                ch++;
        }

        printf("\n");

return;
}
```

Appendix A6 (Cont.): util.c

```
void print_payload(const u_char *payload, int len)
{

        int len_rem = len;
        int line_width = 16;                    /* number of bytes per line */
        int line_len;
        int offset = 0;                                 /* zero-based offset counter */
        const u_char *ch = payload;

        if (len <= 0)
                return;

        /* data fits on one line */
        if (len <= line_width) {
                print_hex_ascii_line(ch, len, offset);
                return;
        }

        /* data spans multiple lines */
        for ( ;; ) {
                /* compute current line length */
                line_len = line_width % len_rem;
                /* print line */
                print_hex_ascii_line(ch, line_len, offset);
                /* compute total remaining */
                len_rem = len_rem - line_len;
                /* shift pointer to remaining bytes to print */
                ch = ch + line_len;
                /* add offset */
                offset = offset + line_width;
                /* check if we have line width chars or less */
                if (len_rem <= line_width) {
                        /* print last line and get out */
                        print_hex_ascii_line(ch, len_rem, offset);
                        break;
                }
        }

return;
}

double sysGetCpuTime()
{
  long curTime;
  struct tms tmsBuf;

  (void) times( &tmsBuf );
  curTime = tmsBuf.tms_utime + tmsBuf.tms_cutime;
  return( (double) curTime / sysconf(_SC_CLK_TCK) );
}
```

Appendix A7: util.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pcap.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

double sysGetCpuTime();

#define NONBT 1
#define BT 2
enum {DFI, DPI};

#define MAXPACKETS 10000

/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN  6

/* Ethernet header */
struct sniff_ethernet {
        u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
        u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
        u_short ether_type;                     /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip {
        u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* dont fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp {
        u_short th_sport;               /* source port */
        u_short th_dport;               /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgement number */
        u_char  th_offx2;               /* data offset, rsvd */
#define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;
        #define TH_FIN  0x01
        #define TH_SYN  0x02
        #define TH_RST  0x04
        #define TH_PUSH 0x08
        #define TH_ACK  0x10
        #define TH_URG  0x20
        #define TH_ECE  0x40
        #define TH_CWR  0x80
        #define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
        u_short th_win;                 /* window */
        u_short th_sum;                 /* checksum */
        u_short th_urp;                 /* urgent pointer */
};
```

Appendix A7 (Cont.): util.h

```
int ipCmp(const void* a, const void* b);

int flowCmp(const void* a, const void* b);
int flowCmpAvgSize(const void* a, const void* b);
int flowCmpNumPkt(const void* a, const void* b);
void print_payload(const u_char *payload, int len);
void print_hex_ascii_line(const u_char *payload, int len, int offset);

typedef struct info_s {
  unsigned numPkt;
  double avgSize;
  unsigned totalHeaderSize;
  unsigned totalSize;

  unsigned numSendPkt;
  unsigned totalSendHeaderSize;
  unsigned totalSendSize;

  unsigned numReceivePkt;
  unsigned totalReceiveHeaderSize;
  unsigned totalReceiveSize;

  int bt_dpi_packet;
  int bt_dpi_flow;
  int bt_dfi;
} info;

typedef struct flow_s {
  char ipAddr1[3*4+3+1];
  char ipAddr2[3*4+3+1];
  unsigned port1;
  unsigned port2;
  unsigned numTPkt;

  info *infoptr;
  int infonum;
} flow;

/*
typedef struct ip_s {
  char ipAddr[3*4+3+1];
  int  bt_dfi;
  int  bt_dpi;
  } ip; */

typedef struct BTIp_s {
  char ipAddr[3*4+3+1];
  int  bt_dfi;
  int  bt_dpi;
} btip;
```

References:

[1] N. Basher, A. Mahanti, A. Mahanti, C. Williamson and M. Arlitt, “A comparative analysis of web and peer-to-peer traffic,” Proceeding of the 17th international conference on World Wide Web, April 21-25, 2008, Beijing, China
, China

[2] H. Chen, Z. Hu, Z. Ye and W. Liu, “A New Model for P2P Traffic Identification Based on DPI and DFI,” Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on Digital Object Identifier: 10.1109/ICIECS.2009.5366295; Publication Year: 2009 , Page(s): 1 – 3

[3] H. Chen, Z. Hu, Z. Ye and W. Liu, “Research of P2P Traffic Identification Based on Neural Network,” Computer Network and Multimedia Technology, 2009. CNMT 2009. International Symposium on Digital Object Identifier: 10.1109/CNMT.2009.5374510; Publication Year: 2009 , Page(s): 1 – 4

[4] H. Chen, X. Zhou, F. You and C. Wang, “Study of Double-Characteristics-Based SVM Method for P2P Traffic Identification,” Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on Volume: 1. Digital Object Identifier: 10.1109/NSWCTC.2010.54 Publication Year: 2010 , Page(s): 202 – 205

[5] F. Constantinou and P. Mavrommatis, “Identifying Known and Unknown Peer-to-Peer Traffic,” Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications, p.93-102, July 24-26, 2006

[6] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson, “Offline/Realtime Traffic Classification Using Semi-Supervised Learning,” IFIP Performance, October 2007.

[7] R. Keralapura, A. Nucci and C. Chuah, “Self-Learning Peer-to-Peer Traffic Classifier,” Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on Digital Object Identifier: 10.1109/ICCCN.2009.5235313 Publication Year: 2009 , Page(s): 1 – 83

[8] A. Klemm, C. Lindemann, M. K. Vernon, and O. P. Waldhorst. Characterizing the query behavior in peer-to-peer file sharing systems. In IMC ’04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, pages 55–67. ACM Press, 2004

[9] T. Le and J. But, “Bittorrent traffic classification,” CAIA Technical report 091022A, 22 October 2009, http://caia.swin.edu.au/reports/091022A/CAIA-TR-091022A.pdf

[10] B. Liu, Zhitand Li and Zhanchun Li, “Measurements of BitTorrent System Based on Netfilter,” Computational Intelligence and Security, 2006 International Conference on Volume: 2 Digital Object Identifier: 10.1109/ICCIAS.2006.295304; Publication Year: 2006 , Page(s): 1470 – 1474

[11] F. Liu; Z. Li and J. Yu, “Applications Identification Based on the Statistics Analysis of Packet Length,” Information Engineering and Electronic Commerce, 2009. IEEC '09. International Symposium on Digital Object Identifier: 10.1109/IEEC.2009.38; Publication Year: 2009 , Page(s): 160 – 163

[12] C. Wang, T. Li and H. Chen, “P2P Traffic Identification Based on Double Layer Characteristics,” Information Technology and Computer Science, 2009. ITCS 2009. International Conference on Volume: 2, Digital Object Identifier: 10.1109/ITCS.2009.298, Publication Year: 2009 , Page(s): 593 – 596

[13] B. Xu, M. Chen, F. Lan and N. Wang, “P2P flows identification method based on listening port,” Broadband Network & Multimedia Technology, 2009. IC-BNMT '09. 2nd IEEE International Conference on Digital Object Identifier: 10.1109/ICBNMT.2009.5348496 Publication Year: 2009 , Page(s): 296 - 300

[14] R. Zhang, Y. Du and Y. Zhang, “A BT Traffic Identification Method Based on Peer-Cache,” Internet Computing for Science and Engineering (ICICSE), 2009 Fourth International Conference on Digital Object Identifier: 10.1109/ICICSE.2009.39 Publication Year: 2009, Page(s): 320 - 323

[15] D. Zhang, C. Zheng, H. Zhang and H. Yu, “Identification and Analysis of Skype Peer-to-Peer Traffic,” Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on Digital Object Identifier: 10.1109/ICIW.2010.36; Publication Year: 2010, Page(s): 200 - 206

[16]http://www.winpcap.org/