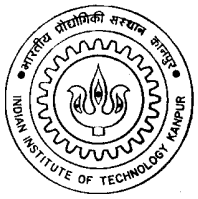
Design and Implementation of a Network Monitoring Tool

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Neeraj Kapoor



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

April, 2002

Certificate

This is to certify that the work contained in the thesis entitled "*Design and Implementation of a Network Monitoring Tool*", by *Neeraj Kapoor*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

April, 2002

(Dr. Dheeraj Sanghi)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

(Dr. Deepak Gupta)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

The extensive use of computers and networks for exchange of information has also had ramifications on the growth and spread of crime through their use. Law enforcement agencies need to keep up with the emerging trends in these areas for crime detection and prevention. Among the several needs of such agencies is the need to monitor, detect and analyze undesirable network traffic. However, the monitoring, detecting, and analysis of this traffic may be against the goal of maintaining privacy of individuals whose network communications are being monitored.

In this thesis, we discuss the design and implementation of the basic framework of a network monitoring tool – PickPacket – that can address the conflicting issues of network monitoring and privacy through its judicious use. PickPacket comprises of three components – a packet filter, post-processing applications, and a GUI for providing a detailed analysis of the collected data. The packet filter filters packets based on IP addresses, transport layer protocol port numbers, and application layer data content present in them. The implementation of application layer protocol filters for Telnet and SMTP and a text string filter is discussed in this report. We also describe the design and implementation of post-processing applications.

Acknowledgements

This thesis is dedicated to my dear Papa.

I take this opportunity to express my sincere gratitude toward my supervisors Dr. Dheeraj Sanghi and Dr. Deepak Gupta for their invaluable guidance throughout my thesis work. It would have never been possible for me to take this project to completion without their ideas and support. I consider myself extremely fortunate to have had a chance to work under their supervision. Learning how to design before implementing and how to write technical reports from them has been a life time experience in itself.

I also wish to thank whole heartily all the faculty members of the Department of Computer Science and Engineering, IIT Kanpur for improving my knowledge, my primary motive of coming to IITK. I also wish to thank the Head and other members of the Computer Center, IIT Kanpur for providing support for testing the application in the IIT Kanpur computer network. I also extend my thanks to the technical staff of the department for maintaining an excellent working facility.

I would like to thank Brajesh Ji, for being co-operative and for all the tea and samosas we had during lunch hours. Without his constant support and ideas this work would have been difficult for me. My sincere thanks to Atul Sir for his help and support. I would like to thank the whole of mtech2000 batch for the times, although small, I shared with them. Among my batchmates, I would especially thank Kedar for the inquisitiveness and liking we shared towards Linux, C and Vim.

I would like to thank my parents for taking me to this stage in life. It was the blessings of my Papa, Ma, Bhaiya and Behna which gave me courage to face the challenges and made my path easier.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the increase in use of computer networks for information exchange, regulation and control of the data transferred on these networks is required to secure the intellectual property of an organization. Thus, highly customizable network monitoring tools that capture data transmitted on the network are being designed. Some of these tools also analyze the collected data and provide valuable information to the user.

Network monitoring tools perform their task by sniffing packets from the network and filtering them on the basis of user specified rules. The tools that provide facility of specifying simple rules for filtering packets are called *packet filters*. Tools that filter packets based on complex rules and perform post-capture analysis of the collected data are termed as *network monitoring tools*. The following section describes packet sniffers. Later in this chapter, we focus on different packet filtering mechanisms and network monitoring tools. This leads to a motivation for developing and designing a new network monitoring tool - PickPacket.

1.1 Packet Sniffers

The basic goal of network monitoring is to read packets from the network and analyze its contents. At the lowest level, it requires that the network interface be able to read all the packets, irrespective of the destination of the packet. This can be

ensured by properly configuring the interface. This activity of monitoring packets on the network is known as *Packet Sniffing*. *Packet sniffers* are simple monitoring tools that can only dump the network traffic on the storage media.

1.2 Packet Filters

The amount of information that flows on a network is generally quite high with packets corresponding to different protocols and even a simple analysis of this data takes time. This time can be reduced considerably by allowing the user to specify some rules for capturing packets selectively. For example, the user should be allowed to specify rules that capture all the packets with a given IP address. This would reduce the amount of space required for saving the packets and also lessen the time required for analysis. *Packet filters* provide this facility of specifying such simple rules. These rules comprise of values corresponding to various fields present the protocol headers of a packet. If the the protocol headers of a packet contain these values then it is saved else it is dropped by the packet filter. We discuss below some existing packet filters.

The *CMU/Stanford Packet Filter* (CSPF) [9] was the first UNIX based kernel-resident, protocol independent packet demultiplexer developed. It provides individual user processes great flexibility in selecting which packets they will receive. It eventually evolved into the *Network Interface Tap* (NIT) [10] under SunOS 3, and later into *Berkeley Packet Filter* (BPF) [8]. Sun implemented NIT to capture packets and *etherfind* to print packet headers. These packet filters although being implemented inside the kernel, provide an architecture over which the user-level network monitoring tools can be built.

In 1993, a new packet filtering mechanism, the Berkeley Packet Filter (BPF) [8] was developed by Steve McCane and Van Jacobson. BPF essentially comprises of two components: a filter code and an interpreter which executes the filter code over the packet read from the network. The filter code uses a hypothetical machine consisting of an accumulator, an index register, a scratch memory store, and a program counter. This filter code is in an assembly like language and includes

operations like load, store, branch, return, some register transfer functions, etc. BPF offers substantial performance improvements over other packet filtering mechanisms due to the following two reasons:

1. There are two approaches for filtering packets: a boolean expression tree (used by CSPF) and a directed acyclic control flow graph or CFG (first used by NNstat [17] and then used by BPF). These two models are computationally equivalent. However, in implementation the tree model maps naturally into code for a stack machine while the CFG model naturally maps into code for a register machine. Since most machines are register based, the CFG approach leads to a more efficient implementation.

2. When a packet arrives at the network interface, the network interface driver saves it in its buffer and then copies it to the system protocol stack. But in the case of BPF, the driver after saving the packet in its buffer calls BPF which operates on the stored packet and decides whether it is to be accepted or not. No copy of the packet is made for this filtering process. This leads to a great performance advantage of BPF over other filtering mechanisms (e.g. NIT [10]) that make a copy of the packet before filtering it.

The BSD socket interface is a de-facto standard for writing network based applications. Thus the Linux operating system came up with the *Linux Socket Filter* (LSF) [16]. LSF is an in-kernel packet filter derived from BPF. It provides the user with a packet filtering facility on BSD sockets. Among other packet filters, *tcpdump* [6] is probably the most popular packet capturing tool in the UNIX community. It is based on BPF and has the packet capturing and filtering facilities implemented in a separate library, *pcap* [5]. There are a wide range of network monitoring tools that integrate the pcap library.

1.3 Network Monitoring Tools (NMT)

Many commercially and freely available network monitoring tools exists. *Ethereal* [4] is a network monitoring tool that allows the operator to examine data from a live

network or from a capture file on disk. The operator can browse the captured data, view the summary and detail information for each packet and the reconstructed stream of sessions. *Iris* [3] is a network traffic analyzer that reports statistics about the traffic and has components meant for session reconstruction.

Carnivore [15], a packet capturing system, has been developed by FBI. It is designed to collect information about the electronic communications to or from a specific user targeted in an investigation. Carnivore is quite different from other tools as it performs filtering of packets on the basis of a rich set of filtering criteria specific to the application layer protocols. This ensures a great degree of privacy to the other users of the network. Carnivore is also capable of monitoring dynamic-IP address based networks. Users can specify text string searches and other application layer protocol specific criteria for filtering.

1.4 PickPacket

PickPacket is a network monitoring tool similar to Carnivore. PickPacket can filter packets based on the IP addresses and transport layer protocol port numbers specified in its configuration file. Moreover, it can also filter packets based on rules specific to the application layer protocol data content present in these packets. For protocols like *SMTP*, the user can specify filtering of sessions on the basis of e-mail addresses and text strings present in the body of the e-mail. PickPacket can help in maintaining the privacy of other users in the network as it provides a rich set of criteria via which the user can control the amount of information saved by this tool while monitoring a network. The filtering criteria are specified by the user in a configuration file.

PickPacket comprises of three components. The first component captures and filters packets transmitted on the network. The second component processes the captured packets and retrieves meta-information from them. This processing involves session reconstruction of the connections present in the collected data. The meta information consists of IP addresses and port numbers of the communicating hosts and application layer protocol information (for example, the sender's and

receiver's e-mail addresses present in an e-mail). The third component reads this meta information and displays it to the user by means of a GUI. The GUI provides a detailed view of all the connections whose packets have matched the user specified criteria.

This thesis mainly focuses on the design and implementation of the basic framework of PickPacket and describes the application layer protocol filters for Telnet and Simple Mail Transfer Protocol (SMTP) sessions. Alongwith, the tool facilitates filtering of packets belonging to any TCP connection on the basis of user specified text strings. The design and implementation of the second component, i.e. the post-processing applications are also discussed in this report.

1.5 Organization of the Report

Chapter 2 discusses the overall design of packet filter. Chapter 3 deals with the implementation details related to the packet filter and various application layer protocol filters including the text string filter. Chapter 4 talks about the post-processing issues and implementation. Chapter 5 focuses on the performance issues. Chapter 6 concludes our work with the limitations and future directions.

Chapter 2

Design of Packet Filter

In this chapter, we describe the design and implementation of the packet filter component of PickPacket. The packet filter is responsible for capturing and filtering the packets transmitted on the network. The filtered packets are stored in a output file which can then be post-processed and the packets can be displayed by means of graphical user interface.

2.1 Packet Filtering : Design Goals

The packet filter should not only provide a rich set of filtering criteria to the user but should also provide mechanisms for protecting the privacy of other users of the network. Indiscriminate use of the filter may result in privacy violation of the users of the network. To maintain privacy, a careful specification of the filtering criteria is desired. Following features, if present in the tool, will help the user in specifying such filtering criteria.

1. The packet filter should allow capturing of packets based on the fields of the various protocol headers present in a packet. These fields may include IP addresses, protocol, port numbers, etc. This will help in using the tool as a simple packet sniffer with very basic filtering being performed on the sniffed packets. The filter should also support filtering based on the application layer data content present in a packet.

2. The packet filter should allow specifying the amount of information that will be collected when a connection meets the filtering criteria. For example, the user may only require information whether any e-mail was sent or received on an e-mail addresses or he/she may also like to know the content of the transferred e-mail. The former mode of collecting information shall be referred as *Pen* mode and the latter as *Full* mode. PickPacket provides this facility by allowing specification of either of these modes in the configuration file. More details regarding the information collected in these modes with respect to the application layer protocol they are associated with are provided in the next chapter.

These expectations from the packet filter translate to design goals that the filter should meet. These goals are outlined below.

Flexibility: The packet filter should support a wide range of filtering criteria for different application layer protocols. These criteria should include important fields in the data transferred on connections belonging to these protocols, like username for *Telnet* [13] sessions, e-mail addresses for *SMTP* [7] connections etc. Support for filtering packets based on IP addresses, protocols and port numbers must also be provided.

Maintaining privacy of the users: Only those packets that meet the user specified criteria should be saved on the disk.

Ease of use: The filtering criteria for the packet filter should be easily configurable by a user with a basic knowledge of computer networks.

Fast Processing: The rate at which packets can be received from the network running the packet filter is determined by the bandwidth of the network. Also, the buffer space provided by the operating system for storing packets is limited. Thus, the filter should take decisions about a packet as soon as possible to prevent packet losses.

Robustness: As a result of providing a rich set of filtering criteria, the packet filter may consume a considerable amount of time in processing a packet.

Meanwhile, the kernel will read packets from the network and save them in its buffer. If sufficient space is not available in the buffer, packets may be dropped by the kernel and these dropped packets will not be available to the packet filter. The packet filter should handle such situations in a manner that the correctness of the data being collected is not sacrificed.

Passive Mode of Operation: The services provided by the network should not get affected due to monitoring. Any decrease in the performance of the network is also undesirable. For these reasons the packet filter should only passively monitor the network.

Extensibility: The design of the packet filter and the framework provided by it should be such that it easily supports addition of filtering modules for other transport layer and application layer protocols.

2.2 Design of the Packet Filter

The packet filter reads packets from the network and matches them against the user specified criteria. The matched packets are stored on the storage media for analysis. The overall architecture of the packet filter is shown in Figure 2.1.

The *configuration file* contains the filtering criteria and other configuration specified by the user. During *initialization* these criteria and configuration are passed to other modules of the system. The *filter generator* is responsible for generating the BPF filter code and passing it to the kernel for in-kernel filtering. The *demultiplexer* reads packets transmitted on the network and passes them to them to other modules based on the application layer protocol they belong to. The *connection manager* checks for out of sequence packets and informs this to the *application layer protocol filter*. The *application layer protocol filter* checks the data present in the packet and filters it based on the criteria. If the filtering criteria are fully met, the packets are saved on the disk by the *output file manager*.

When a packet is received, it is passed to the demultiplexer. The packet filter supports filtering of packets belonging to different application layer protocols

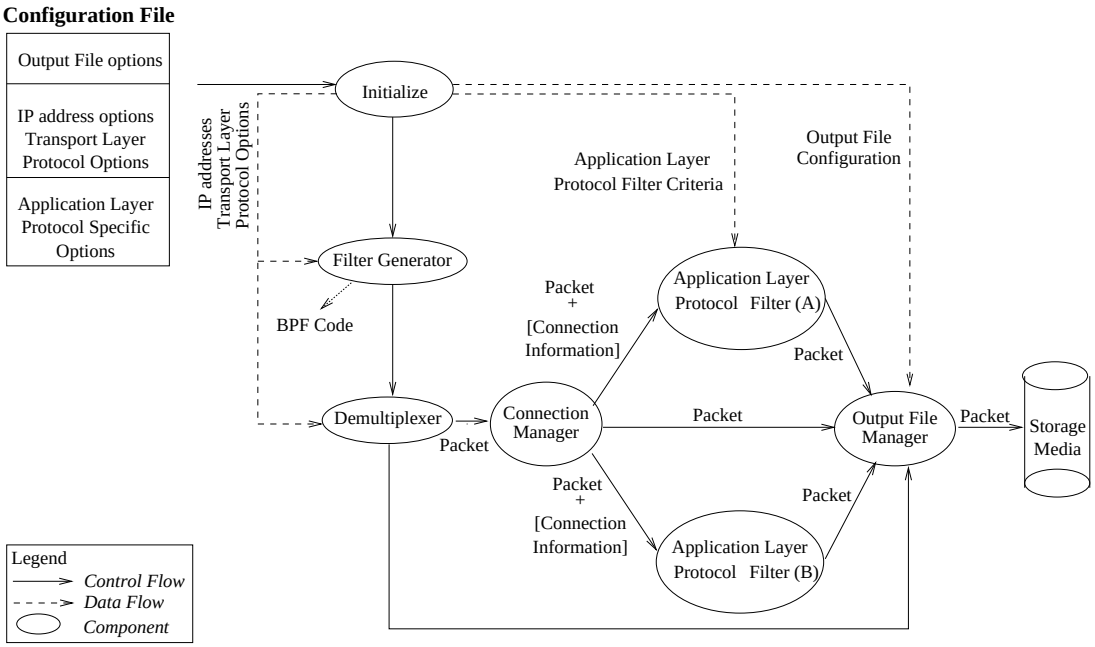


Figure 2.1: Basic Design of the Packet Filtering Phase

simultaneously. Based on the information present in the configuration file, the demultiplexer determines the application layer protocol to which this packet belongs and passes it to the connection manager corresponding to the transport layer protocol header present in the packet. There can be a connection manager for each transport layer protocol. A connection manager for a transport layer protocol maintains state information about the active connections that are using this protocol in the packet. After processing the packet, the connection manager passes the packet and possibly some state information to the application layer protocol filter module. If the packet or the connection meets the filtering criteria then, depending on the mode of operation, packets belonging to this connection are saved on disk by passing them to the output file manager.

The rest of this chapter provides a detail description of the configuration file and each of the modules involved and discusses their role in filtering the packets.

2.2.1 Configuration File

The configuration file is a text file which contains the rules specified by the user in a fixed format. A sample file is given in *Appendix A*. Criteria in the file can be broadly categorized into three sections -

1. The first section contains the configuration specific to the output files that are created by the packet filter for saving the filtered packets. The user can specify multiple output files with their maximum sizes. The packet filter can support multiple output file managers and the configuration for each of them can be specified in the file. Each of these output file managers may have their own formats for saving the data on the disk.

2. The second section contains rules for filtering packets based on source and destination IP addresses, transport layer protocol, and source and destination port numbers. These rules are referred as *basic criteria* in the rest of this report. Every rule in the basic criteria also specifies the application layer protocol to which packets matching a set of IP address and port number criteria belong. Based on this information, the packet filter passes these packets to the corresponding *application layer protocol filter*. The packet filter has support for specifying multiple such tuples for different application layer protocols in the same configuration file so as to simultaneously filter packets belonging to these protocols. The packet filter can also be asked to directly save the packets matching a set of basic criteria.

3. The third section comprises of multiple subsections, each of which contains rules corresponding to an application layer protocol. Based on these rules the application layer data content of the packets is analyzed.

2.2.2 Initialization

During initialization the packet filter reads the configuration file and initializes the various modules of the system. The first section of the configuration file, comprising of information about the output files, is read and passed to the corresponding output

file manager. The output file manager verifies whether the location specified for the output file has sufficient amount of free space and creates the specified output file.

The basic criteria are now read from the configuration file. Based on the application layer protocol name specified in these criteria, initialization functions of the corresponding application layer protocol modules are called. These functions allocate the memory required for maintaining state information about the connections of interest. Also, the initialization functions of the connection managers particular to the transport layer protocol specified in the basic criteria are called. These function allocate the memory required for maintaining connection information for connections that are of interest to the application layer protocol filter. After these initialization, the IP address range, transport layer protocol name and the port number range is passed to the demultiplexer. The demultiplexer saves this information by making an entry into its table and saving these values in it.

The third section of the configuration file has multiple subsections containing filtering rules specific to the application layer protocols. For each subsection, the routine of the registered application layer protocol filter that reads the configuration from the file is called. The details related to the configuration that can be specified for each of the application layer protocol filter is given in the next chapter.

2.2.3 Filter Generator

The current implementation of the packet filter utilizes in-kernel filtering capability of the Linux kernel provided by means of the *Linux Socket Filter* (LSF). This filtering is done based on the basic criteria specified in the configuration file. Only those packets which contain the IP address and the port number lying in the range of one of the basic criteria and having the transport layer protocol as specified in that basic criteria are passed to the packet filter. Thus, the packet filter does not read all the packets transmitted on the network. This reduces the total number of system calls made by the packet filter and the amount of data copied from the kernel's address space to the packet filter's address space during filtering.

LSF is a kernel resident packet filter. A user-level application can pass filtering criteria to LSF by means of an interface provided by Linux kernel. LSF follows

exactly the same filter code structure as the BSD Berkeley Packet Filter (BPF) [8]. The module of the packet filter that provides an interface to communicate with LSF is the *Filter Generator*. Using this interface other modules of the system can attach or remove BPF code from the kernel. As LSF filters packet based on the BPF code passed to it, we need a mechanism to generate this code. This support is provided by the *Pcap* [5] library. The *Pcap* library is a portable, system-independent library for capturing packets from the network. It has functions that generate BPF code from a BPF expression. *Filter generator* generates the BPF expression from the basic criteria and passes this expression to the Pcap library. The Pcap library returns the BPF code which is then passed to the kernel for in-kernel filtering of packets.

2.2.4 The Demultiplexer

Packets after being copied from the network are checked against the criteria based on the application layer data content present in them. For this the packet filter determines to which application layer protocol the packet belongs and passes it to the respective filtering module. In other words, packets are demultiplexed on the basis of the application layer protocol they belong to. The module performing this job is the *Demultiplexer*.

For demultiplexing packets the demultiplexer maintains a table of tuples containing the basic criteria specified in the configuration file. As stated earlier, every basic criteria contains a field specifying the name of the transport layer protocol. Thus, packets that match a basic criteria are passed to the connection manager routine for the corresponding transport layer protocol. If no processing of the data content is desired then the name of the application layer protocol can be specified as *"DUMP_PEN"* or *"DUMP_FULL"*. In either of these cases, packets matching this criteria are directly written to the disk by passing them to the output file manager. If the application layer protocol is specified as *"DUMP_PEN"* then the packet data till and including the transport layer protocol header is saved. If it is specified as *"DUMP_FULL"* then the entire packet is saved.

There can be a situation where an application might require modifying the BPF code while filtering packets. This can happen if the filter has detected a connection

being established from an IP address or on some port number that was not specified in the configuration file. An example of such a situation is *Passive FTP* [14]. As shown in Figure 2.2, in a passive file transfer, the FTP client contacts the server on the standard FTP command port and issues a passive (PASV) command. The FTP server replies with its own IP address and a port to which the client is supposed to connect for data connection. This port when sent from the server is a non-standard port and hence cannot be determined beforehand. For monitoring this connection the in-kernel filtering code i.e., the *BPF* code, needs to be modified.

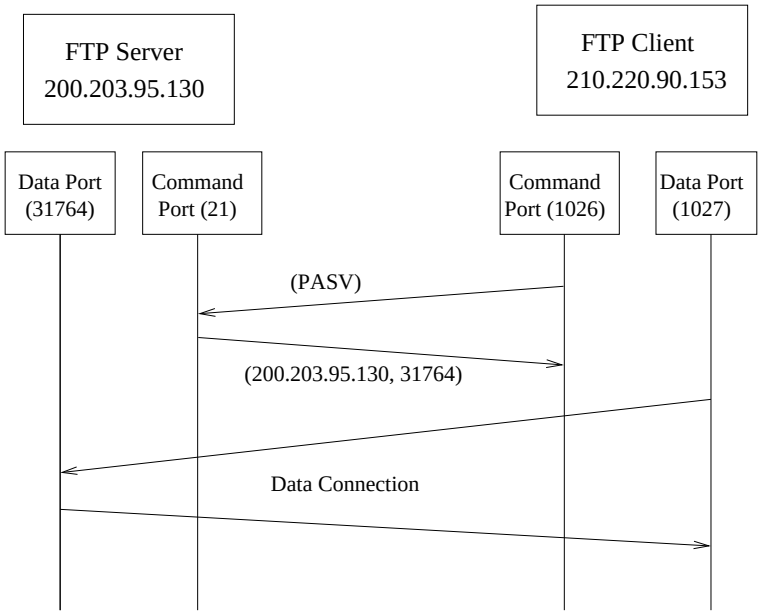


Figure 2.2: Passive File Transfers

For this purpose the demultiplexer maintains its table in two separate parts, a *static table* and *a dynamic table*. It provides a suitable interface for adding a new entry into these tables. Whenever an application layer protocol filter desires a modification in the *BPF* code, it adds the new tuple into the dynamic table of the demultiplexer and removes the existing BPF code from the kernel. Functions provided by the *Filter Generator* which generate the BPF code and pass it to the kernel for in-kernel filtering are then called by the application layer protocol filter.

In the interval between removing the current BPF code and attaching the new one, no in-kernel filtering is performed and all the packets that are read are buffered by the kernel. *Demultiplexer* checks every packet against the entries in its table. Packets which do not match any of these criteria are discarded by it.

2.2.5 Connection Manager

Connection manager maintains state specific to the transport layer protocol of the currently monitored connections. The packet filter can filter packets belonging to two transport layer protocols– *Transmission Control Protocol* (TCP) [12] and *User Datagram Protocol* (UDP) [11]. UDP is a connection-less protocol hence no connection information needs to be maintained for packets belonging to it. Thus a connection manager for UDP has not been implemented. Packets belonging to this protocol are filtered based on IP address and port number. State information for connections belonging to TCP needs to be maintained as it is a connection oriented protocol. We have implemented a connection manager for TCP and packets belonging to TCP are filtered based on the application layer data content present in them.

The application specific filtering reduces to text search in the application layer data content of the packets. In case of communications over connection oriented protocol, this text search should handle situations where the desired text is split across two or more packets before being transmitted on the network. As there may be losses or reordering of packets in the network, these filters should also check for packets that are received out of sequence while performing the search for split text. The component that does these checks is the *TCP Connection Manager*.

Out of sequence packets can be detected if any of the lower layer protocols or the application layer protocol maintain sequencing information about the data transferred over a connection. This information is derived from the *sequence number* field present in TCP header. TCP numbers each data byte in both the streams with unique sequence numbers. This sequence number is saved in the state information maintained by the connection manager.

The connection manager should maintain state information for only those connections that the application level protocol is interested in. For this reason, it provides functions by which an application layer protocol filter can alert it so as to maintain the sequence information for a connection. There can be instances when the application layer protocol filter may also maintain state information for the connections of interest. For example, while performing a text search, the search text may be split across two or more packets. In such cases, a partial match will occur in first packet and the application layer protocol filter should remember this information so as to check the next packet belonging to this connection.

The application layer protocol filter may maintain its information for connections of interest in a list. The connection manager also needs to maintain a list of state information for all the connections on which it has been alerted. This information is separately maintained for every connection, hence retrieving requires searching the list based on the communication tuple. Similar search needs to be performed by the application layer protocol filter on its list. We eliminate this duplication by merging these lists and maintaining a reference to the application layer protocol filter's information in the every connection information node present in the merged list. As shown in Figure 2.3, on receiving a packet the connection manager searches this list and retrieves its state information and the application layer state information maintained for packets on this connection. After processing the packet it passes on the packet and the application layer state information to the application layer protocol filter. Whenever the application layer protocol filter wants to maintain state information about a connection, it passes this information to the connection manager while alerting it.

Such an interface provides benefits like clear separation of the tasks, simplicity to the design, and extensibility for addition of other application layer protocol filters. Incorporating a filtering module for a new application layer protocol is easy if there is an existing application layer protocol filter having the same transport layer protocol as of the new one. We can also have a single connection manager for all the application layer protocols running on the same transport layer protocol.

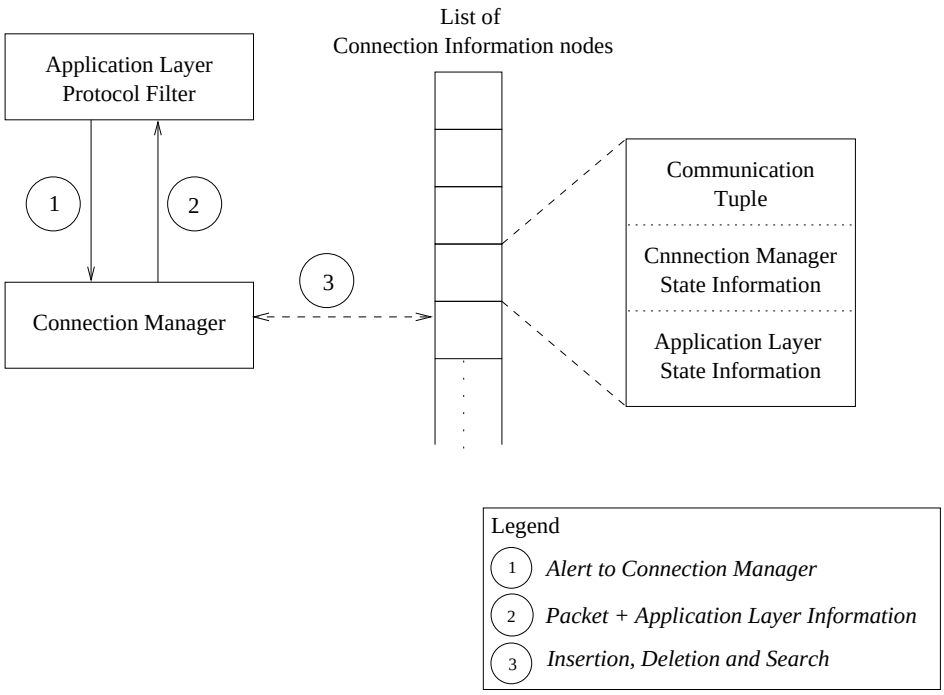


Figure 2.3: Interface between an Application Layer Protocol Filter and TCP Connection Manager

2.2.6 Application Layer Protocol Filter

The application layer protocol filter receives packets from the connection manager. It analyzes the contents of the packet checking it against the filtering criteria. If a packet or a connection satisfies the criteria then the filter passes this packet to the output file manager for saving it on the disk. Details related to the working of the various application layer protocol filters are explained in the next chapter.

2.2.7 Output File Manager

The configuration for the output file manager comprises of the full pathname of the output file and the maximum size of the file. On receiving a packet, the output file manager checks if the sum of the amount of data written to the currently open file and the size of the current packet exceeds the maximum size specified for this

file. If the maximum size is less than the sum then the current file is closed and the next file in the list is opened. The current packet is saved in this new file. But, if the maximum size is greater then the packet is saved in the current file itself. This feature is useful for downloading the filtered data in smaller increments and analyzing it without restarting the packet filter. The packet filter supports multiple output file managers and the configuration for each of these output file managers can be specified in the configuration file. We have implemented a *Default Output File Manager* that saves the packets on the storage media in the *Pcap* format as shown in Figure 2.4.

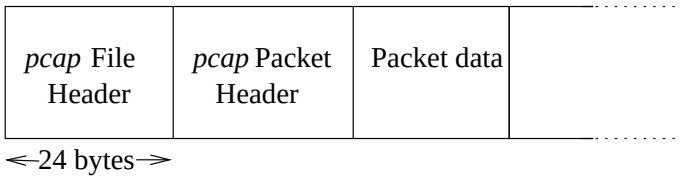


Figure 2.4: *pcap* file format

The output file begins with a 24 byte Pcap file header. This file header contains information related to the Pcap version and information about the network from which the data was read and saved in this file. After this header we have chunks of information corresponding to the packets which are saved in the file. Every chunk comprises of a packet header followed by the packet data. The packet header comprises of three fields - the actual length of the packet when it was read from the network, the length of the packet which was saved on the disk and the time at which this packet was read from the network.

Chapter 3

Implementation of Packet Filter

In this chapter we describe our implementation of the packet filter and the working of the application layer protocol filters. The packet filter has been implemented on Linux 2.4.3 kernel. Initially, we present an overview of the various modules involved and then discuss the working of the text string filter, and application layer protocol filters for Telnet and SMTP.

3.1 Overview

As shown in Figure 3.1, the packet filter comprises of four basic modules - *demultiplexer*, *TCP connection manager*, *application layer protocol filters* and *default output file manager*. Application layer protocols for which we have implemented filtering modules work over TCP and require identical functionality from the connection manager. Thus, we have implemented a single TCP connection manager. *SMTP Filter* and *Telnet Filter* are application layer protocol filters that filter e-mails and telnet sessions respectively. *Text String Filter* has also been implemented as an application layer protocol filter. This filter searches for user specified text strings in the application layer data content of packets.

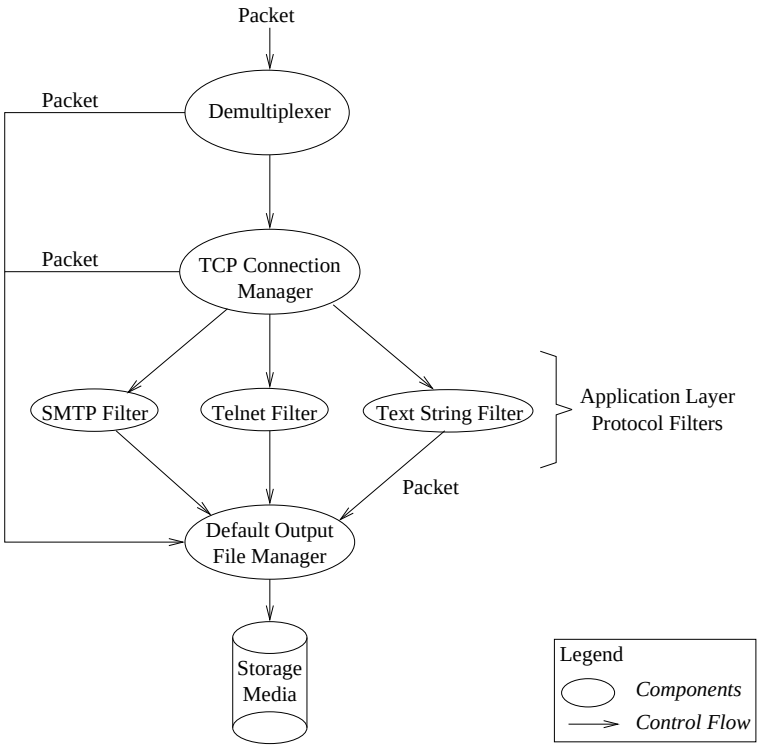


Figure 3.1: Modules of the Packet Filter

3.1.1 Demultiplexer

After reading the packet from the network, it is saved in a *My_ Packet* structure. The number of bytes present in the packet and the time at which it was read is also saved in this structure. This structure is passed to the demultiplexer. The demultiplexer maintains a table of tuples in a structure *dmplx_ table*. Every tuple comprises of the following fields - source and destination IP address range, source and destination port number range, transport layer protocol name, a reference to connection manager's function, and a reference to the memory allocated to the application layer protocol filter. This table is maintained in two parts, the *static_ table* and the *dynamic_ table*. The *static_ table* stores the basic criteria specified in the configuration file. The *dynamic_ table* stores those basic criteria which may be added by other modules while filtering.

On receiving a packet, the demultiplexer retrieves source and destination IP addresses and port numbers from the packet and searches for a matching tuple in its static and dynamic tables. A match occurs if source and destination IP addresses and port numbers present in the packet lie in the corresponding ranges present in a tuple with the transport layer protocol being the same. A reverse check is also made for capturing packets sent by either of the communicating hosts. On a match, the packet is either passed to the TCP connection manager or is passed to the default output file manager based on the application layer protocol name specified in the configuration file.

If the packet is passed to the TCP connection manager then the last field of the matched tuple, which is a reference to the memory allocated to the application layer protocol filter, is also passed. This memory consists of sequence information maintained by connection manager for the application layer protocol filter to which current packet belongs. The reason for maintaining this reference in demultiplexer table is that separate memory is being maintained for different application layer protocol filters whereas we have implemented a single connection manager. Thus, connection manager needs to know the location where it has stored connection information for the connection to which the current packet belongs.

The demultiplexer allows addition and removal of tuples from its dynamic table via the *dmplx_add_to_dynamic_table* and *dmplx_rem_from_dynamic_table* functions. These functions can be called by other modules when they require modifcations to the existing in-kernel filter code. After calling either of these functions the *filter_generator* and *attach_filter* functions should also be called for generating the new BPF code and passing it to the kernel.

The packet filter processes transport layer protocol headers of two protocols – Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). A connection manager for TCP has been implemented. Packets belonging to UDP are written on the disk by the demultiplexer in either *Pen* or *Full* mode.

3.1.2 TCP Connection Manager

The TCP connection manager checks for out of sequence packets and passes this information to the application layer protocol filters. For checking this it maintains the connection information shown in Figure 3.2 in memory. A reference to this memory is passed by the demultiplexer. There is a separate memory area for different application layer protocol filters. It consists of the following fields:

1. *filter_func*: A pointer to the application layer protocol filter function. We have implemented a single connection manager for various application layer protocol filters, The connection manager uses this pointer for passing the packet to the corresponding application layer filter function.

2. *write_packet*: Pointer to a function of the output file manager that writes the packets on to the disk.

3. *free_func*: Pointer to a function of the application layer protocol module that removes the state information maintained by the application layer protocol filter for a connection.

4. *last_flush_time*: A variable to store the time when TCP connection manager's flush function was last called.

5. *active_list*: Pointer to a hash table containing connection information for different connections.

6. *free_list*: Pointer to the start of the list of connection information nodes currently not in use.

The information maintained for a connection by the connection manager is retrieved when a packet belonging to this connection is received. There can be a situation when the packet filter does not know that a connection has terminated since the TCP FIN packet of that connection may be lost or dropped. Connection manager releases the information maintained for this connection based on a timer. When the timer expires, a flush function is called. This function searches the *active_list* for a connection information node which has not been used for a time equal

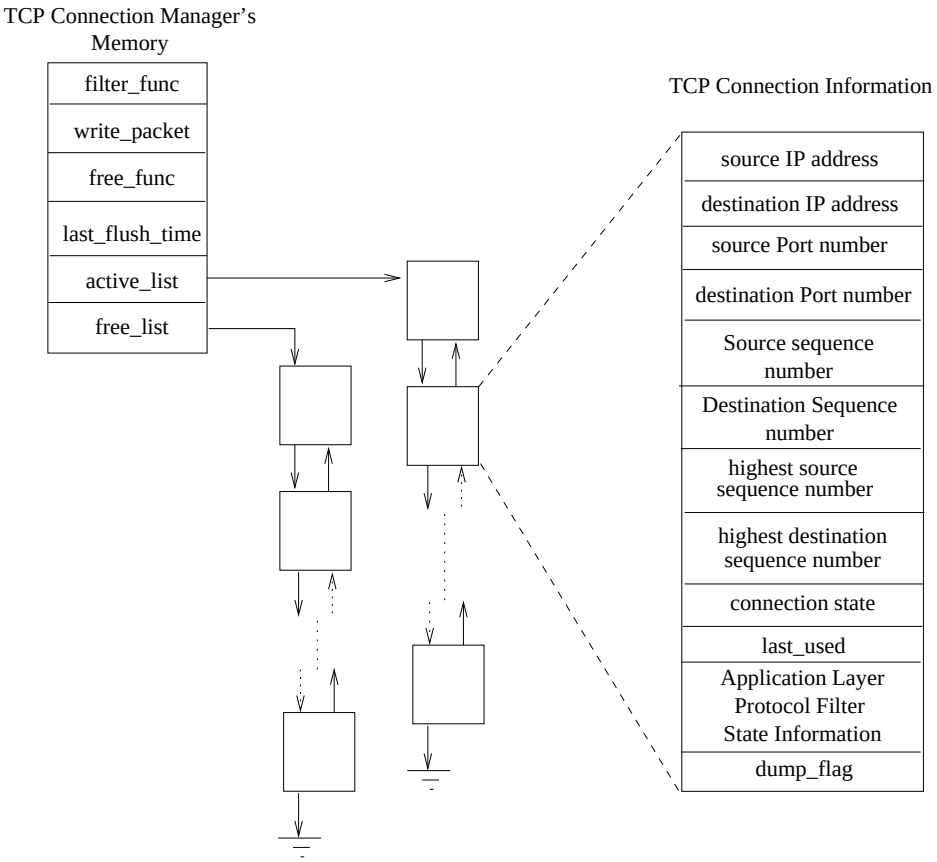


Figure 3.2: TCP Connection Manager Address Space

to or greater than the *TCP_KEEPALIVE_TIME* [12]. If such a node is found, it is removed from the *active_list* and placed in the *free_list*. For this reason, every connection information node has a *last_used* variable to remember the time when a packet belonging to this connection was last read by the packet filter. As shown in Figure 3.2, the connection information also contains the source and destination IP addresses and port numbers, the sequence numbers of the packets last received from the communicating hosts, the highest seen sequence numbers of the packets sent from them, and the connection state. A reference to the application layer protocol filter's state information is also present in this structure.

The connection manager can also pass the received packet to the default output file manager if the application layer protocol filter has requested it to do so. A *dump_flag* is maintained in the connection information for this purpose. The value of this flag can be set or unset by the application layer protocol filter. If the value of this flag is true then the connection manager passes this packet to the default output file manager by calling the function pointed by *write_packet*. The connection manager has *tcp_alert_current* and *tcp_alert_new* functions that are called when an application layer protocol filter determines that a connection needs to be monitored. Similarly, if the application layer protocol filter determines that the connection information for a connection should be released then it calls the *tcp_forget_current* or *tcp_forget_conn* functions provided by the connection manager.

3.2 Application Layer Protocol Filters

We have implemented two application layer protocol filters and a text string filter. For performing fast text searches in the application layer data content of a packet, a text search library has also been implemented. This library uses the *Boyer-Moore* [1] string-matching algorithm for searching the text. It has functions which can do case-sensitive and case-insensitive text searches based on this algorithm. The following sub-sections explain the working of the text string filter and the Telnet and SMTP filters.

3.2.1 Text String Filter

This filter searches for a user specified text string in the application layer data content of the packets. It only collects information of the connections in which one of the user specified text string is found. When the text string is split across two packets, both packets are captured by the filter. The working of this filter is shown in Figure 3.3.

On receiving a packet, the text filter checks whether state information is being maintained for the connection to which this packet belongs. If no state information is maintained then it searches the data content of the current packet for the occurrence

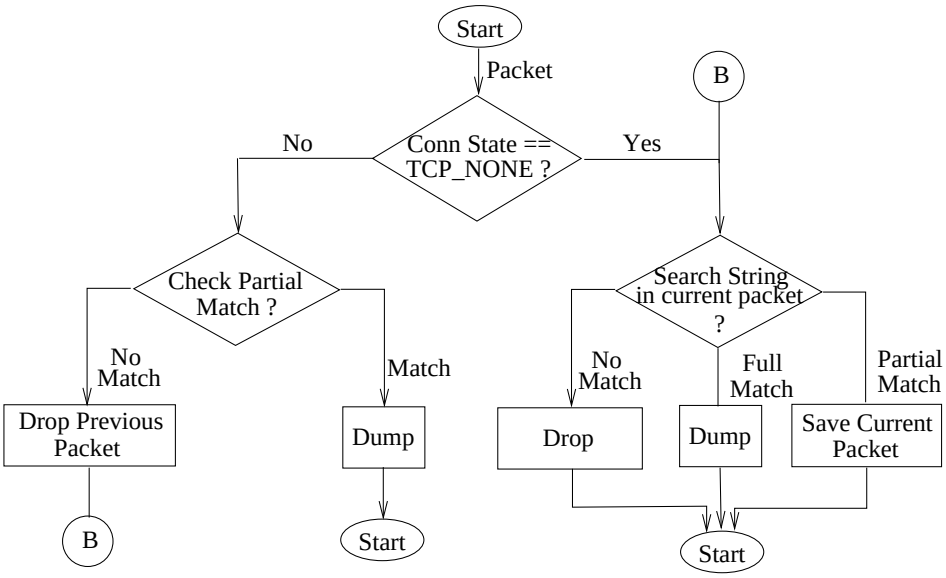


Figure 3.3: Stages in Text String Filter

of one of the user specified text string. If none of the text strings are present in the packet then this packet is dropped. If one of the text strings is present in this packet, a full match occurs. Now, based on the mode of operation, packets belonging to this connection are saved on the disk. If a partial match occurs, i.e. last bytes of the application layer data content are equal to a *proper prefix* of a user specified text string then the current packet is saved in the state information. The connection manager is alerted and this state information is passed to it.

If state information is being maintained and the current packet is a consecutive packet then the remaining portion of the partially matched string and the starting part of the data content present in this packet are compared. On a match, packets are saved on the disk based on the mode of operation. If the match does not occur then the packet saved in the state information is dropped and the current packet is searched for the occurrence of one of the specified text strings. If none of these strings are present in packet then the *tcp_forget_current* function is called to release the information being maintained for this connection. If one of the text strings is found in the current packet then based on the mode of operation packets belonging to this

connection are collected. If a partial match occurs then the current packet is saved but the connection manager is not alerted as it is already maintaining connection information for this connection.

Based on the mode of operation, the amount of information collected is as follows:

1. In *Pen* mode only packets which contain one of the specified text strings are collected.

2. In *Full* mode all packets starting from the one in which the text string occurred are collected by the filter.

3.2.2 SMTP Filter

SMTP filter filters e-mails transferred on the network. User can specify e-mail addresses and the text to be searched in e-mails. As shown in Figure 3.4, on receiving a packet containing the *MAIL FROM* command sent from the SMTP client the filter searches the user specified e-mail addresses in this packet. If a match occurs then this packet is saved in the in the state information maintained for this connection. Also, the match flag used for recording the user specified criteria satisfied by the connection is set to *MAIL_ID_MATCHED*. If the match does not occur then also the *MAIL FROM* packet is saved in the connection information as the following packets i.e., *RCPT TO* packet or the ones containing the mail envelope, may contain one of the the e-mail addresses specified by the user. In either of the above mentioned scenarios the connection state is modified to *MAIL_FROM_SENT* and the TCP connection manager is alerted by passing this state information to it. If the filter now receives a negative acknowledgment from the server i.e., the server has refused to send this mail due to an error, this state information is removed. If an e-mail address match occurs in the *MAIL FROM* packet and if this session has satisfied other criteria specified by the user (e.g. text string search in the body of the e-mail) then packets belonging to this e-mail are saved on the disk. Similar actions are taken when the filter receives a *RCPT TO* packet.

If the packet containing the *DATA* command is received then the packets following it and containing the envelope of the e-mail are searched for user specified e-mail

addresses. This search is only performed if neither the *MAIL FROM* packet nor the *RCPT TO* packets contained one of the user specified e-mail addresses. Checking the envelope of the e-mail guarantees that if the `Cc:` (Carbon Copy) field of the e-mail contains one of the user specified e-mail addresses then also it will be captured by the packet filter. Searching for the user specified e-mail addresses is based on a sub-string match. For example, an e-mail address `neeraj` will match the following e-mail addresses: `neeraj@iitk.ac.in`, `neerajk@iitk.ac.in`, and `kneeraj@iitk.ac.in`. The e-mail envelope and the body of the e-mail is searched for user specified text strings thus ensuring that if the text string occurs in the subject of the e-mail then also it will be detected.

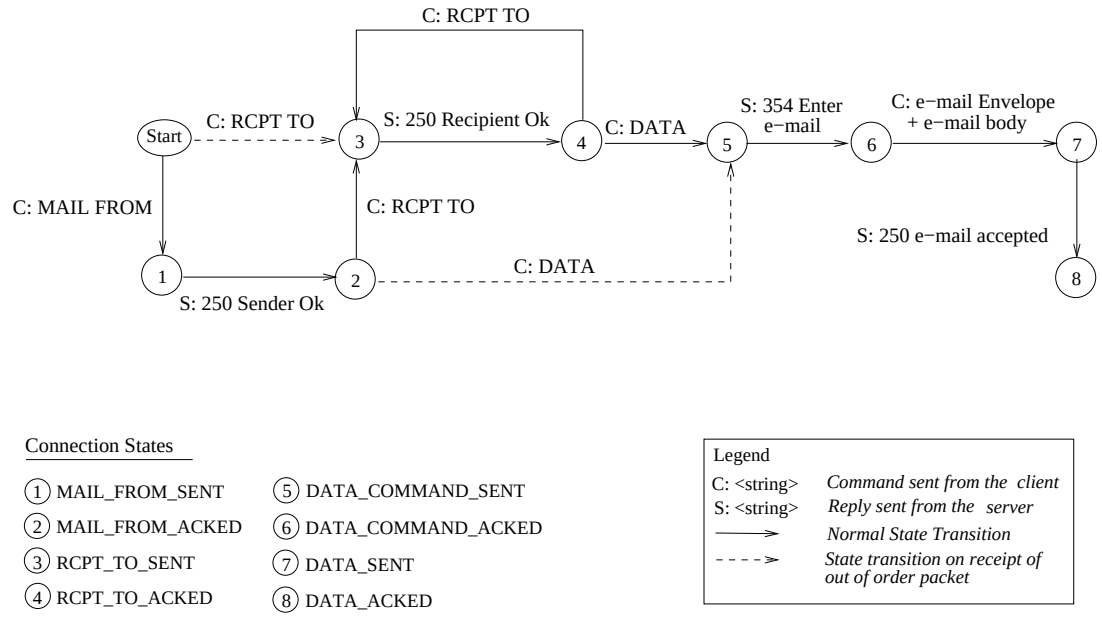


Figure 3.4: The SMTP Filter

Based on the packets received by the SMTP filter, the connection state of every SMTP connection is one of those shown in Figure 3.4. The amount of information recorded by the SMTP filter depends on the mode of operation specified in the configuration file.

1. If no text string is specified then the recorded information includes:

- In *Pen* mode of operation, packets containing the e-mail envelope (which contains the To/From/Subject and other header of the e-mail).
- In *Full* mode, all packets of the SMTP sessions.

2. If the user has specified one or more text strings then information about e-mails which contained one of the specified text strings is recorded. This information includes:

   - in *Pen* mode of operation, packets corresponding to the envelope of the e-mail.
   - in *Full* mode, the envelope of the e-mail and the packets starting from the one in which the text string occurred.

3.2.3 Telnet Filter

Telnet sessions are filtered based on the usernames that are used to establish these sessions. Two issues particular to the implementation of the Telnet filter are as follows:

1. Except for the Telnet commands [13], no standard format is prescribed for the messages exchanged between the Telnet server and client during the login phase. By messages we refer to the content that is displayed to the user during the login phase. This makes the checks for a login or a password prompt difficult.

2. On an average, the amount of data present in a packet transferred over a Telnet connection is small. For example, when a user is typing the username then every character is normally sent in a separate packet. This makes the checking for the username nontrivial.

The solution of the first issue was derived after analyzing the sessions of Telnet servers implemented on Linux, Windows 2000, IRIX, Solaris and HP-UX operating systems. The results obtained through the analysis are as follows:

1. The login prompt is present either in its entirety or as a trailing part of a packet. This helps in determining when the server is requesting a username from the client.

2. The password prompt is present in its entirety in a packet.

3. It a user does a successful login then generally a *"Last Login"* message is sent from the server. On an unsuccessful login, a *"Login incorrect"* message is sent from the server. These messages help in detecting a successful or an unsuccessful login. The Telnet filter collects information about a Telnet connection only if the user has successfully logged in.

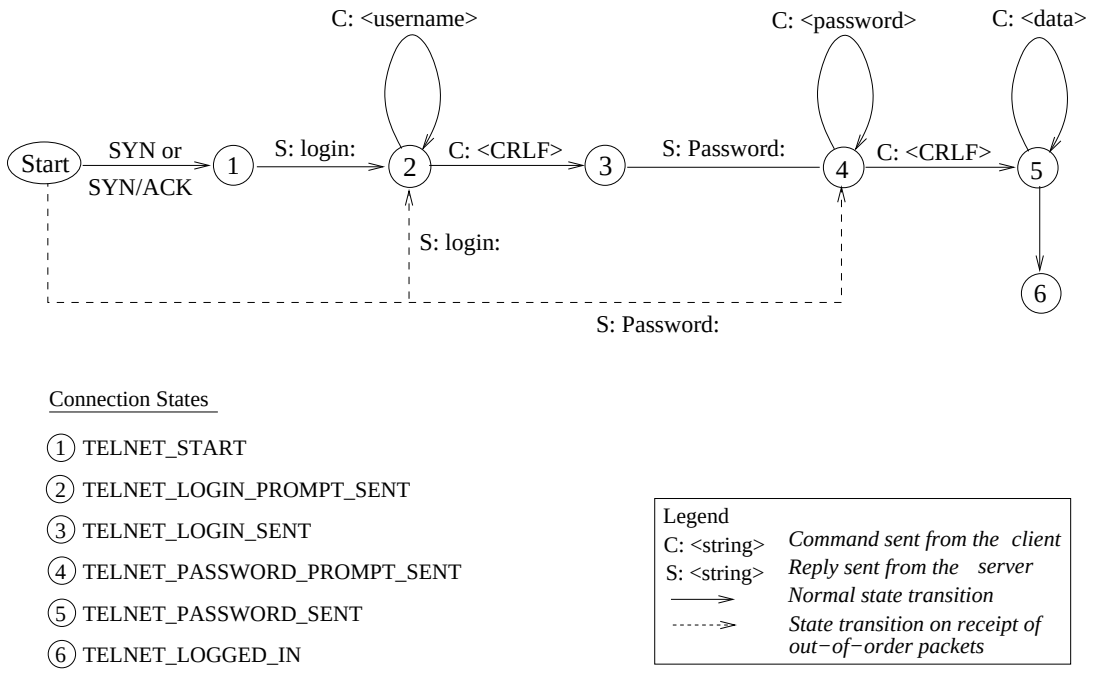


Figure 3.5: The Telnet Filter

A simple state diagram describing the working of the Telnet filter is shown in Figure 3.5. The monitoring of a Telnet session starts when the filter receives a TCP *SYN* or a TCP *SYN-ACK* packet from either of the communicating hosts. After this, the filter waits for a packet from the Telnet server containing the login prompt.

On receiving this packet, the data present in all the packets going to be sent from the client till the one containing carriage return (CR) and/or line feed (LF) is buffered in the username buffer. On receiving a CR or CRLF the usernames specified in the filtering criteria are searched in the string present in the username buffer. While searching a substring match is performed as in the case of e-mail addresses. If a match occurs then this session is further checked for a successful login. Till the time a successful login or an unsuccessful login is detected all the packets are saved in the state information maintained for this connection. If the login is successful then based on the mode of operation the amount of information collected by the filter is as follows:

1. If no username is specified then the saved information includes -

   - In *Pen* mode of operation, the packets containing the username of the Telnet connections and the time when the connection terminated.
   - In *Full* mode of operation, all the packets belonging to the Telnet sessions.

2. If one or more usernames are specified in the configuration file then sessions which are established only by these usernames are monitored. The recorded information in the two modes is same as mentioned above.

Chapter 4

Post-capture Analysis

The packet filter writes filtered packets in an output file that is analyzed offline to separate packets into their respective connections. In this chapter we initially discuss the objectives for post-capture analysis. Later, we focus on the design and implementation of the modules involved in this processing.

4.1 Objectives

The output file generated by the packet filter needs to be processed to analyze the captured data. This processing should include separating packets based on the transport layer protocol and the appplication layer protocol they belong to. If the transport layer protocol is a connectionless protocol then packets should be separated on the basis of the communication tuple. For packets belonging to a connection-oriented transport layer protocol, information about the total number of connections present in the output file, the data transferred in these connections, time at which the connections existed, and application layer protocol specific information should be made available. Thus, the objectives of post-capture analysis can be summarized as follows:

1. Packets present in the output file and belonging to a connection-oriented protocol should be separated into their respective connections. Packets belonging to a connectionless protocol should be separated based on the communication

tuple.

2. While post-processing the collected data, meta-information about the connections should be retrieved and saved in a human understandable format. This meta-information includes important fields present in the data content belonging to an application layer protocol. For example, e-mail addresses of SMTP connections, usernames of FTP connections etc.

The rest of this chapter describes our design and implementation details of the components involved.

4.2 Post-Processing Design

The components involved in post-processing of the collected data are shown in figure 4.1. Packets present in the output file generated by the packet filter are sorted by the *Sorter* module. The *Connection Breaker* module does session reconstruction for the connections present in the sorted output file and separates packets belonging to a connectionless protocol based on the communication tuple. The meta-information specific to the application layer protocols and present in the captured data is retrieved by the *Meta Information Gathering Module*.

The packets present in the output file may not be in the order they were transmitted on the network. This is because the packet filter may remember packets i.e., save them in memory, before writing them on the disk and while writing it may not guarantee that the packets are written in order they were transmitted on the network. Due to this reason the packets present in the output file are sorted by the sorter module and saved in the order they were read from the network. The criteria used for sorting these packets is the timestamp which is saved for every packet present in the output file. This timestamp is the time at which the packet was read from the network.

The connection breaker module reads the sorted output file and retrieves the connection information from the packets belonging to a connection oriented protocol and separates them into different files. Packets belonging to a connectionless

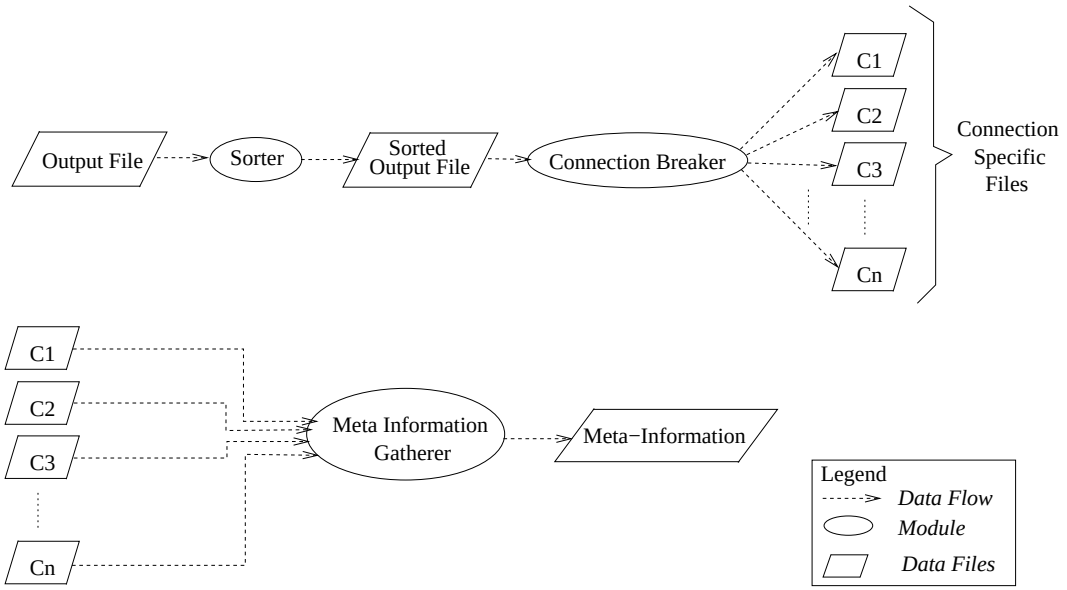


Figure 4.1: Post-Processing Design

protocol are separated based on the communication tuple. Now the meta information gathering module reads each of these connection specific files and retrieves the meta-information of every connection. This meta information serves as an input to the application providing the user-interface. The implementation level details of these components are as follows.

4.3 Implementation

4.3.1 The Sorter Module

The default output file manager saves packets in the output file in Pcap format. The Pcap packet header present in this format contains the timestamp at which the packet was read from the network. The sorter module sorts the packets present in the output file based on this timestamp as this guarantees that all the packets present in the output file are in the order they were read from the network. The size of the output file may be huge, hence we cannot use an *internal sorting* algorithm

for sorting packets present in the file. The *k-way merge sort* algorithm [2], which is an *external sorting* algorithm, is used for sorting packets present in the output file.

4.3.2 The Connection Breaking Module

PickPacket supports filtering of packets belonging to TCP and UDP transport layer protocols. Thus, packets present in the output file belong to either of these protocols. The connection breaking module reconstructs TCP sessions from the sorted output file and separates UDP packets on the basis of the communication tuple.

If the packet read from the file is a UDP packet then the connection breaker module retrieves information about the communication tuple present in this packet. Based on this information it checks whether a file corresponding to this tuple exists or not. If a file exists then the packet is appended to that file else a new file is created and the packet is saved in it.

The connection breaking module separates the TCP packets into multiple files based on the communication tuple. There can be a case that while the packet filter was monitoring the network, multiple connections were made on the same communication tuple. The connection breaking module further analyzes the packets present in these communication tuple specific files and separates these packets on the basis of the connections they belong to. It detects the start and end of a connection with the help of some rules and heuristics a summary of which is is given in the state diagram shown in Figure 4.2.

To detect the start and end of a connection, the connection breaking module maintains connection information of the communicating hosts. This information comprises of a packet list, connection state, expected sequence number and the last acknowledged sequence number of packets sent by the communicating hosts. While reading packets, the connection breaking module monitors four flag bits - `SYN, FIN, ACK and RST` - present in the TCP header [12]. If a `SYN` packet is seen then the state of the source of this packet is set to *SYN_SENT* and of the destination to *SYN_RCVD*. If `SYN` and `SYN-ACK` packets are not present in the file and a data packet has been read then the source and destination state in set to *ESTABLISHED* to account for the absence.

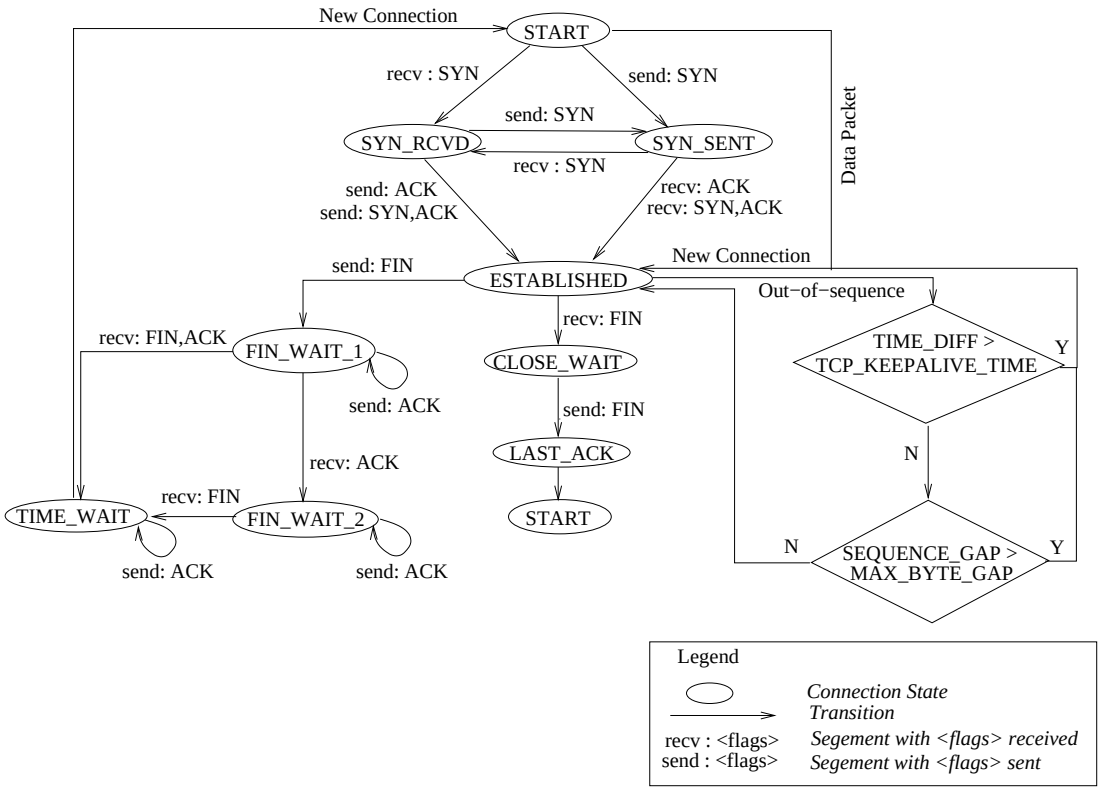


Figure 4.2: TCP/IP Session Reconstruction

If the connection state is *ESTABLISHED* and the packet read is a `FIN` packet, the connection breaking module sets the state of the source to *FIN_ WAIT_ 1* and of the destination to *CLOSE_ WAIT_ 1*. If a `FIN` packet is sent while being in *CLOSE_ WAIT_ 1* then the state of the source is changed to *LAST_ ACK*. Similarly, if an `ACK` packet is received while being in *FIN_ WAIT_ 1*, the state is modified to *FIN_ WAIT_ 2*. If a `FIN` is received now, the connection state is changed to *TIME_ WAIT*. In Figure 4.2, some minor transitions have been omitted for simplicity. For example, if a *SYN* packet is received while the connection is in *FIN_ WAIT_ 1* state then this is taken as start of a new connection.

Other than these rules, while examining the packets, heuristics are used to account for the missing packets. These heuristics are based on the fact that the initial sequence number (ISN) generated by TCP while establishing a communication is

a 32 bit pseudo-random number. It is quite likely that the ISNs of two separate connections will be wide apart in the sequence space. If the connection is in *ESTABLISHED* state and the data packet read from the file is having the sequence number not equal to the next expected sequence number then the time difference between the previous packet and the current packet is evaluated. If this difference is greater than the *TCP_KEEPALIVE_TIME* [12], the default value of which is 7200 seconds then this new packet is treated as belonging to a new connection. If the time difference is less than the *TCP_KEEPALIVE_TIME* then the difference in the sequence number of the current packet and the next expected sequence number is computed. If this difference is less than the *MAX_BYTES_GAP* then the current packet is taken as belonging to the same connection else it is taken as belonging to a new connection. The default value of the parameter MAX_BYTES_GAP is 20 KBytes but it can also be modified by the user.

4.3.3 The Meta Information Gathering Module

After separating the packets into multiple connection specific and communication tuple specific files, meta-information is retrieved from each of these files. For different application layer protocols, this information comprises of different fields. Processing is also done for FTP data connections. This includes determining which connection specific file corresponds to which FTP file transfer command. The reason is that while transferring a file on an FTP data connection, the name of the file is transferred on the FTP control connection. As these are separate connections, hence a correspondence between the FTP data transfer command containing the file name and the file containing the corresponding FTP data connection should be established. The meta information gathering module performs these tasks and generates files containing the meta-information. This meta-information and the communication specific files act as an input to the application providing the user-interface.

The meta information gathering module reads the files generated by the connection breaking module. For each of these files, it saves the application layer data content and the timestamp of every packet in a separate file. The meta-information

common to every connection and collected by the meta information gathering module includes: a connection identifier which is unique for every connection, the IP addresses and transport layer protocol port numbers, the name of the conversation file, and the time of the first and the last packets belonging to this connection. The conversation file contains the application layer data content of the packets sent by the communicating hosts. Based on the application layer protocol, to which this communication belongs, the information retrieved by the meta information gathering module also includes:

- for SMTP connections:

    1. The name of file containing the reconstructed e-mail as sent from the user. The content present in this e-mail depends on the amount of data saved by the packet filter. This file can be directly opened by *Microsoft Outlook Express*.
    2. The sender's e-mail address.
    3. The receiver's e-mail addresses.

- for FTP data connections:

    1. The FTP username and password (if present in the dump file) of the FTP control connection in which the FTP file transfer command was sent.
    2. The file name by which this file was referred in the FTP file transfer command.
    3. The type, structure and mode information about this data connection.
    4. The type of the file as determined from the Unix *file(1)* command.
    5. A connection identifier for the FTP control connection in which the file transfer command was sent.

Chapter 5

Performance Evaluation

In this chapter, we discuss the results of the experiments conducted to test the system in an actual network. The essential idea of these experiments was to determine the percentage of packet losses experienced by the packet filter while monitoring the network. The results indicate that the delay encountered due to the overhead of processing the sniffed packets does not lead to packet losses even at more than 10 Mbps traffic.

5.1 Determining packet losses

If the packet filter is slow in filtering packets, the kernel may start dropping packets when its internal buffers get filled up. As a result packets containing data meeting the user specified criteria may not be captured by the packet filter. The number of packets dropped may be determined if the kernel provides such statistics to the user after the network interface is set into promiscuous mode. Unfortunately such a facility is not provided by the Linux kernel. In order to get this information an alternate method was opted.

The packet filter cannot capture more packets than a sniffer which simple counts the packets seen without performing any filtering. If this sniffer can provide information about the number of packets it has read from the network then we can compare it with the number of packets read by the packet filter and hence determine if any

packets are lost by the packet filter. For this we had a setup where an instance of the packet filter was run on two machines. One of these instances recorded the number of packets it has read from the network and it did not perform any application level filtering. The other instance also recorded the number of packets but performed the application level filtering specified in its configuration file.

5.2 Experimental Setup

In these experiments two identical machines with Intel Pentium 801MHz CPU, 128 MB RAM and running Linux kernel version 2.4.3 were used. The experiments were carried out on a 100 Mbps Ethernet segment at the Indian Institute of Technology, Kanpur (IITK) Computer Center. The packet filter was run manually and simultaneously on the two machines. Identical IP address and port number criteria were specified in the input configuration files. This ensured that the in-kernel filtering code generated by the packet filter was identical on both machines. In one of the configuration files no application level filtering criteria were specified and the output file was specified as `/dev/null`. Thus, this instance of the packet filter read packets filtered by the kernel and wrote them to the NULL device. The other instance filtered packets based on the application layer protocol specific criteria present in its configuration file and wrote the packets in an output file located on the disk. For simplicity we will refer the former packet filter as the *test filter* and latter as the *real filter*. Except in the first case mentioned below, filtering was stopped by setting a timer which expired in 15 minutes.

5.2.1 Basic Filter

The configuration files for the *real* and *test* filters for this test are shown in *Appendix B*. In both of these configuration files the IP addresses and port numbers are specified as wildcards and no application level filtering is specified. After saving 500 MB of data both instances of the packet filter terminated.

The duration in which this amount of data was collected was 4 minutes and 59 seconds and the average data transfer rate during this time interval was 11.49 Mbps.

Filter	Total Packets received by the interface	Packets read by the packet filter	Packets saved
real filter	877909	845173	845173
test filter	878009	845176	845176

Table 5.1: Filtering by reading all the TCP packets

Table 5.1 shows the total number of packets received by the network interfaces, the number of packets read by the filters, and the number of packets saved by the filters. Both of these filters captured aprroximately an equal number of packets. The difference in the number of packets read by the kernel is due to the small difference in time at which filtering was started on the two machines.

5.2.2 Text Filter

The configuration files for the *real* and *test* filters for this test are shown in *Appendix B*. An IP address range and port number range were specified in the basic criteria present in the configuration file. The average data transfer rate during this test was 5.07 Mbps. Table 5.2 lists the filtering statistics of the two filters.

Filter	Total Packets received by the interface	Packets read by the packet filter	Packets saved
real filter	1065242	518864	28590
test filter	1064897	518786	518786

Table 5.2: Text Filtering by reading packets matching the IP address and port number criteria

5.2.3 SMTP Filter

The configuration files for the *real* and *test* filters for this test are shown in *Appendix B*. The IP addresses and port number specified in the configuration file are wildcards.

The average data transfer rate during this test was 7.41 Mbps. Table 5.3 lists the filtering statistics of the two filters.

Filter	Total Packets received by the interface	Packets read by the packet filter	Packets saved
real filter	1301000	5654	3240
test filter	1302976	5654	5654

Table 5.3: SMTP Filtering by reading all e-mail packets

5.2.4 Telnet Filter

The configuration files for the *real* and *test* filters for this test are shown in *Appendix B*. The IP addresses and port numbers specified in the configuration file are wild-cards. The average data transfer rate during this test was 9.51 Mbps. Table 5.4 shows the filtering statistics of the two filters.

Filter	Total Packets received by the interface	Packets read by the packet filter	Packets saved
real filter	1542746	48150	58
test filter	1542962	48151	48151

Table 5.4: Filtering of Telnet Sessions by reading all packets belonging to the Telnet protocol

5.3 Results

In all the tests discussed above, the slight difference in the number of packets read by the packet filters is due to the difference in time when the two instances of the packet filter started filtering. The results show that about an equal number of packets are

read by the two packet filters and thus we can conclude that no packets are dropped by the *real* filter. Thus the time required to filter packets based on the application layer data content does not force the kernel to drop packets even at high data rates.

Chapter 6

Conclusions

In this thesis we have presented an approach for capturing and filtering data transmitted on the network based on user specified criteria. The packets are not only filtered based on the values present in the various fields of the network and transport layer protocol headers, but also on the application layer data content present in them.

To compete with the speed at which packets are transmitted on the network, various optimizations have been performed in the design and implementation of the packet filter. The packet filter passes the BPF code responsible for filtering packets based on the IP addresses and transport layer protocol port numbers to the kernel. Based on this code the Linux Socket Filter filters the packets inside the kernel. These filtered packets are read by the packet filter and passed to the filtering module of the application layer protocol it belongs to. These modules inspect the application layer data content of the packet and filter it based on the fitering criteria specified by the user.

Modules for filtering e-mails (SMTP connections) and Telnet sessions have been implemented. Users can specify e-mail addresses and text strings to be searched in the e-mails transmitted on the network. Filtering of Telnet sessions is performed on the basis of the username used for establishing the session. A text string filter which filters connections based on the presence of a user specified text string in the application layer data content has also been implemented.

Several experiments were conducted for checking the performance of the packet filter. These experiments reveal that even at high network load the packet filter is able to successfully capture packets and filter them on the basis of the criterias specified by the user. Applications for post-capture analysis of the collected data were also implemented which reconstruct the TCP connections and retrieve the desired information from the captured data.

6.1 Future Work

We have implemented application layer protocol filter for two protocols. Filtering modules for other widely used application layer protocols like File Transfer Protocol (FTP) and Hyper-text Transfer Protocol (HTTP) should also be implemented. The main limitation of the current implementation is that only static IP addresses can be specified in the configuration file. Extending the application to dynamic IP address allocation based networks would be a challenging exercise.

Bibliography

[1] R. Boyer and J. Moore. "A fast string searching algorithm". In *Comm. ACM 20*, pages 762–772, 1977.

[2] Leiserson Corman and Rivest. *Introduction to Algorithms*. McGraw-Hill, 1992.

[3] eEye Digital Security. "Iris". The Network Traffic Analyzer (http://www.ethereal.com).

[4] Gerald Combs et al. "Ethereal". Available at http://www.ethereal.com.

[5] V. Jacobson, C. Leres, and S McCanne. *"pcap - Packet Capture Library"*, 2001. Unix man page.

[6] Van Jacobson, Craig Leres, and Steven McCanne. "tcpdump : A Network Monitoring and Packet Capturing Tool". Available via anonymous FTP from ftp://ftp.ee.lbl.gov.

[7] J. Klensin. "Simple Mail Transfer Protocol". Technical report, 2001. http://www.ietf.org/rfc/rfc2821.txt.

[8] Steve McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In *Proceedings of USENIX Winter Conference*, pages 259–269, San Diego, California, Jan 1993.

[9] Jeffrey C. Mogul, Richar F. Rashid, and Michael J. Accetta. "The Packet Filter: An Efficient Mechanism for User Level Network Code.". In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 2–4, Western Research Laboratory, Palo Alto, California, USA., 1987.

[10] Sun OS. *"Sun OS 4.1 Manual"*, 1987.

[11] J. Postel. "User Datagram Protocol". Technical report, 1980. http://www.ietf.org/rfc/rfc0768.txt.

[12] J. Postel. "Transmission Control Protocol". Technical report, Information Sciences Institute, 1981. http://www.ietf.org/rfc/rfc0793.txt.

[13] J. Postel and J. Reynolds. "Telnet Protocol Specification". Technical report, 1983. http://www.ietf.org/rfc/rfc0854.txt.

[14] J. Postel and J.K. Reynolds. "File Transfer Protocol". Technical report, 1985. http://www.ietf.org/rfc/rfc0959.txt.

[15] Stephen P.Smith, Jr. Henry Perrit, Harold Krent, Stephen Mencik, J. Allen Crider, Mengfen Shyong, and Larry L. Reynolds. "Independent Technical Review of the Carnivore System". Technical report, IIT Research Institute, Nov 2000. http://www.usdoj.gov/jmd/publications/carniv_entry.htm.

[16] Jay Schulist. "Linux Socket Filter". Details in the Linux kernel source tree file: Documentation/networking/filter.txt.

[17] Braden R. T. "A Pseudo-machine for Packet Monitoring and Statistics". In *Proceedings of SIGCOMM '88, ACM*, 1988.

Appendix A

Configuration File

```
# The first section contains the name of the output file and maximum size
<Output_File_Manager_Settings>
  <Default_Output_File_manager_Settings>

  Num_Of_Files=2
  File_Path=/tmp/output/output_1.dump
  File_Size=250
  File_Path=/tmp/output/output_2.dump
  File_Size=500
  </Default_Output_File_manager_Settings>
</Output_File_Manager_Settings>

# This is the second section containing the name of network interface and the IP
# address and port number criteria.
<Basic_Criterias>

# The name of the network device which is to be brought into promiscuous mode
DEVICE=eth0
```

```
Num_Of_Criterias=2
Criteria=172.31.18.1-172.31.18.255:130.0.1.1-130.0.10.90:0-0:25-25:TCP:SMTP
Criteria=200.230.90.1-200.230.90.10:0.0.0.0-0.0.0.0:0-0:0-0:UDP:DUMP_PEN
</Basic_Criterias>

# This is the third section of this file and contains the application layer
# protocol specific criteria. Here we have the filtering criteria for e-mails.
<SMTP_Configuration>

# This sub section contains the list of the e-mail addresses
<Search_Email_ID>

  Num_of_email_id=1
  Case-Sensitive=no
  E-mail_ID=joshua

</Search_Email_ID>

# This sub section contains the list of text strings
<Search_Text_Strings>

  Num_of_Strings=2
  Case-Sensitive=no
  String=ammunitions
  String=war

</Search_Text_Strings>

# The following line specifies the mode of operation.
Mode_Of_Operation=full
</SMTP_Configuration>
```

Appendix B

Configuration Files used in Tests

B.1 Basic Filter Configuration Files

B.1.1 Real Filter

```
<Output_File_Manager_Settings>
    <Default_Output_File_Manager_Settings>
        Num_Of_Files=1
        File_Path=/tmp/raw_Config_500mb.dump
        File_Size=500
    </Default_Output_File_Manager_Settings>
</Output_File_Manager_Settings>

<Basic_Criterias>
    DEVICE=eth0
    Num_Of_Criterias=1
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:0-0:0-0:TCP:DUMP_FULL
</Basic_Criterias>
```

B.1.2 Test Filter

```
<Output_File_Manager_Settings>
```

```
    <Default_Output_File_Manager_Settings>
        Num_Of_Files=1
        File_Path=/dev/null
        File_Size=500
    </Default_Output_File_Manager_Settings>
</Output_File_Manager_Settings>

<Basic_Criterias>
    DEVICE=eth0
    Num_Of_Criterias=1
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:0-0:0-0:TCP:DUMP_FULL
</Basic_Criterias>
```

B.2 Text Filter Configuration Files

B.2.1 Real Filter

```
<Output_File_Manager_Settings>
    <Default_Output_File_Manager_Settings>
        Num_Of_Files=1
        File_Path=/tmp/text_Criteria_25.dump
        File_Size=500
    </Default_Output_File_Manager_Settings>
</Output_File_Manager_Settings>

<Basic_Criterias>
    DEVICE=eth0
    Num_Of_Criterias=1
    Criteria=0.0.0.0-0.0.0.0:203.200.95.1-203.200.95.254:0-0:1-5000:TCP:TEXT
</Basic_Criterias>

<TEXT_Configuration>
```

```
    <Search_Text_Strings>
        Num_Of_Strings=3
        Case-Sensitive=no
        String=timesofindia
        String=iitk.ac.in
        String=google.com
    </Search_Text_Strings>
    Mode_Of_Operation=pen
</TEXT_Configuration>
```

B.2.2 Test Filter

```
<Output_File_Manager_Settings>
    <Default_Output_File_Manager_Settings>
        Num_Of_Files=1
        File_Path=/dev/null
        File_Size=0
    </Default_Output_File_Manager_Settings>
</Output_File_Manager_Settings>

<Basic_Criterias>
    DEVICE=eth0
    Num_Of_Criterias=1
    Criteria=0.0.0.0-0.0.0.0:203.200.95.1-203.200.95.254:0-0:1-5000:TCP:DUMP_FULL
</Basic_Criterias>
```

B.3 SMTP Filter Configuration Files

B.3.1 Real Filter

```
<Output_File_Manager_Settings>
```

```
    <Default_Output_File_Manager_Settings>
        Num_Of_Files=1
        File_Path=/tmp/smtp_Criteria_25.dump
        File_Size=500
    </Default_Output_File_Manager_Settings>
</Output_File_Manager_Settings>

<Basic_Criterias>
    DEVICE=eth0
    Num_Of_Criterias=1
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:0-0:25-25:TCP:SMTP
</Basic_Criterias>

<SMTP_Configuration>
    <Search_Email_ID>
        Num_Of_Email_Id=1
        Case-Sensitive=no
        E-mail_ID=iitk
    </Search_Email_ID>
    <Search_Text_Strings>
        Num_Of_Strings=0
    </Search_Text_Strings>
    Mode_Of_Operation=pen
</SMTP_Configuration>
```

B.3.2 Test Filter

```
<Output_File_Manager_Settings>
    <Default_Output_File_Manager_Settings>
        Num_Of_Files=1
        File_Path=/dev/null
        File_Size=0
```

```
    </Default_Output_File_Manager_Settings>
</Output_File_Manager_Settings>

<Basic_Criterias>
    DEVICE=eth0
    Num_Of_Criterias=1
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:0-0:25-25:TCP:DUMP_FULL
</Basic_Criterias>
```

B.4 Telnet Filter Configuration Files

B.4.1 Real Filter

```
<Output_File_Manager_Settings>
    <Default_Output_File_Manager_Settings>
        Num_Of_Files=1
        File_Path=/tmp/telnet.dump
        File_Size=100
    </Default_Output_File_Manager_Settings>
</Output_File_Manager_Settings>

<Basic_Criterias>
    DEVICE=eth0
    Num_Of_Criterias=1
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:0-0:23-23:TCP:TELNET
</Basic_Criterias>

<TELNET_Configuration>
    <Usernames>
        Num_Of_Usernames=1
        Case-Sensitive=no
        Username=neerajk
```

```
    </Usernames>
    Mode_Of_Operation=full
</TELNET_Configuration>
```

B.4.2 Test Filter

```
<Output_File_Manager_Settings>
    <Default_Output_File_Manager_Settings>
        Num_Of_Files=1
        File_Path=/dev/null
        File_Size=0
    </Default_Output_File_Manager_Settings>
</Output_File_Manager_Settings>

<Basic_Criterias>
    DEVICE=eth0
    Num_Of_Criterias=1
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:0-0:23-23:TCP:DUMP_FULL
</Basic_Criterias>
```