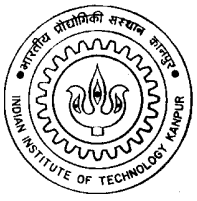
The Network Monitoring Tool - PickPacket: Filtering FTP and HTTP packets

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

Master of Technology

by

Brajesh Pande



to the

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

September, 2002

Certificate

This is to certify that the work contained in the thesis entitled "*The Network Monitoring Tool - PickPacket: Filtering FTP and HTTP packets*", by *Brajesh Pande*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

September, 2002

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

(Dr. Dheeraj Sanghi)
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.

PickPacket - a network monitoring tool - that can handle the conflicting issues of network monitoring and privacy through its judicious use – is discussed in Reference [23]. PickPacket has four components – *The PickPacket Configuration File Generator* for assisting the user in setting up the parameters for capturing packets, the *PickPacket Packet Filter* for capturing packets, the *PickPacket Post-Processor* for analyzing packets, and the *PickPacket Data Viewer* for showing the captured data to the user. This thesis discusses filtering FTP [38] and HTTP [17] packets in the PickPacket Packet Filter.

Acknowledgments

I take this opportunity to place on record my gratitude to my thesis supervisors Dr. Dheeraj Sanghi and Dr. Deepak Gupta. Their guidance and support saw the timely completion of this thesis. This thesis is for a project that is financially supported by the Ministry of Communications and Information Technology, New Delhi. The support of the Ministry of Communications and Information Technology for the project is duly acknowledged.

I also thank the other team members involved with the development of PickPacket - Neeraj, Sanjay, Prashant, Abhay, Nitin and Ankit for their cooperation and support. Abhay, Nitin and Ankit painstakingly performed several tests on PickPacket. The help extended by Sanjay and Prashant during the development of PickPacket will always remain in my memory. Apart from other benevelonces, Sanjay came up with a simple HTTP1.1 server client routine and Prashant added his wizardry with systems configuration and test setups. Diwaker pointed us to D4X that helped us in the tests. I fondly remember Neeraj who had asked me to send him the final shipment version of the code. Unfortunately, he is no longer with us.

I wish to thank Dr. Sanjeev Aggarwal on whose behest I undertook my MTech at IIT Kanpur. Without his encouragemnet this work would never have seen daylight. I also want to thank IIT Kanpur for allowing me to pursue my studies along with my work. I wish to thank everyone at the Computer Centre of IIT Kanpur for their support and help during my MTech. Dr. Raghavendra Tewari was always sympathetic and understanding and let me pour my studying blues on his shoulders.

I would like to mention a few persons/friends who have shaped my thinking in ways unknown to them. Manindra, Vijayan, Sumit and Atul - Thank You.

I thank my parents and all my Gurus for enhancing my knowledge in every possible way.

Finally, I thank my wife Sonu. It were the long hours stolen from the time due her that make the story of my MTech. Thank you from the bottom of my heart.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The use of computers has rapidly increased in the last few decades. Coupled with this has been the exponential growth of the Internet. Computers can now exchange large volumes of information. This has resulted in an ever increasing need for effective tools that can monitor the network.

Such monitoring tools help network administrators in evaluating and diagnosing performance problems with servers, the network wire, hubs and applications. Since machines cannot distinguish personalities and content, they can also be used for communication and exchange of information pertaining to unlawful activity. This is why law enforcing agencies have shown increased interest in network monitoring tools. It is felt that careful and judicious monitoring of data flowing across the net can help detect and prevent crime. Such monitoring tools, therefore, have an important role in intelligence gathering. Companies that want to safeguard their recent developments and research from falling into the hand of their competitors also resort to intelligence gathering. Thus there is a pressing need to monitor, detect and analyze undesirable network traffic.

However, the monitoring, detecting, and analysis of this traffic may be opposed to the goals of maintaining the privacy of individuals whose network communications are being monitored. This thesis describes PickPacket – a Network Monitoring Tool – that can address the conflicting issues of network monitoring and privacy through its judicious use. This tool was developed as a part of a research project sponsored

by the Ministry of Communications and Information Technology, New Delhi. The basic framework for this tool has also been discussed in Reference [23].

1.1 Sniffers

Network monitoring tools are also called sniffers. Network sniffers are named after a product called Sniffer Network Analyzer introduced in 1988 by Network General Corporation (now Network Associates Incorporated) who have also trademarked the word sniffer. However this word continues to be in popular use for lack of other convenient synonyms.

Several tools exist that can monitor network traffic. Usually such tools will put the network card of a computer into the promiscuous mode. This enables the computer to listen to the entire traffic on that section of the network. There can be an additional level of filtering of these packets based on the IP related header data present in the packet. Usually such filtering specifies simple criteria for the IP addresses and ports present in the packet. Filtered packets are written on to the disk. Post capture analysis is done on these packets to gather the required information from these packets.

However, this simplistic model of packet sniffing and filtering has its drawbacks. First, as only a minimal amount of filtering of packets received is carried out, the amount of data for post processing becomes enormous. Second, no filtering is done on the basis of the content of the packet payload. Third, as the entire data is dumped to the disk the privacy of innocent individuals who may be communicating during the time of monitoring the network may be violated. This motivates the design and implementation of PickPacket.

1.2 PickPacket

The purpose of PickPacket, like the simple filter discussed above is to monitor network traffic and to copy only selected packets for further analysis. However, the scope and complexity of criteria that can be specified for selecting packets is greatly

increased. The criteria for selecting packets can be specified at several layers of the protocol stack. Thus there can be criteria for the Network Layer – IP addresses, Transport Layer – Port numbers and Application Layer – Application dependent such as file names, email ids, URLs, text string searches etc. The filtering component of this tool does not inject any packets onto the network. Once the packets have been selected based on these criteria they are dumped to permanent storage.

A special provision has been made in the tool for two modes of capturing packets depending on the amount of granularity with which data has to be captured. These are the "PEN" mode and the "FULL" mode of operations. In the first mode it is only established that a packet corresponding to a particular criterion specified by the user was encountered and minimal information required for detailed investigation is captured. In the second mode the data of such a packet is also captured. Judiciously using these features can help protect the privacy of innocent users.

The packets dumped to the disk are analyzed in the off-line mode. Post dump analysis makes available to the investigator separate files for different connections. The tool provides a summary of all the connections and also provides an interface to view recorded traffic. This interface extensively uses existing software to render the captured data to the investigator. For instance, when rendering e-mail Outlook may be used through the interface provided. A GUI for generating the rules input to the filter is also provided.

1.3 Organization of the Report

This thesis focuses in detail on filtering data packets belonging to applications based on the File Transfer Protocol (FTP) [38] and the Hypertext Transfer Protocol (HTTP) [17]. Chapter 2 and Chapter 3 prepare the background that will help understand sniffers and PickPacket in general. Chapter 2 discuses sniffers in greater detail. Chapter 3 describes the high level design of PickPacket. Chapter 4 discusses the design and implementation details of filtering based on FTP and Chapter 5 discusses the same for HTTP. The rest of the thesis describes testing strategies. The final chapter concludes the thesis with suggestions for further work.

Chapter 2

Sniffers

Network sniffers are software applications often bundled with hardware devices and are used for eavesdropping on network traffic. Akin to a telephone wire-tap that allows a person to listen in on to other people's conversation, a sniffing program lets someone listen in on computer conversations. Network sniffers are named after a product called the Sniffer Network Analyzer introduced in 1988 by Network General Corporation (now Network Associates Incorporated). The word "sniffer" is a registered trademark of this company but is currently in popular use. Sniffers usually provide some form of protocol-level analysis that allows them to decode the data flowing across the network according to the needs of the user. Data flows in the network in packets and often this analysis is done on a packet by packet basis.

Sniffing programs have been traditionally used for helping in managing and administering networks. However, covertly, these programs are also used for breaking into computers. Recently, sniffers have also found use with law enforcement agencies for gathering intelligence and helping in crime prevention and detection. Typically such programs can be used for evaluating and diagnosing network related problems, debugging applications, rendering captured data, network intrusion detection and network traffic logging.

2.1 The Primary Mechanism of Sniffing

Any success in using sniffers can be attributed to the fact that machines on a local network share the same wire (transmission media). Since many machines share the same wire, each machine must have a unique identifier for the data to reach the correct destination through the shared wire. This unique identifier is called the MAC (Media Access Control) address of the machine.

When a machine on the network communicates with another, it packs the data that it wishes to send into a frame. This frame contains – other than the data and communication protocol headers – its own MAC address and the MAC address of the destination machine. Though other information is also put into the frame the focus of interest for the current discussion is the MAC address. If the destination machine happens to be on a wire other than the wire that this machine shares, the MAC address of the nearest router is set as the destination MAC address of the frame. The router on receiving the frame changes some of the frame data and the destination MAC address and forwards the data.

The Ethernet hardware (the standard network adapter) has a hardware chip that ignores all traffic not intended for that hardware. This is accomplished by ignoring all frames on the wire whose destination MAC addresses do not match the MAC address of the Ethernet hardware. Network sniffers turn off the filtering mechanism of the hardware chip on the network adapter and collect all frames irrespective of the destination MAC address. This is known as putting the network adapter into the "promiscuous mode".

2.2 Filtering Sniffed Data

The amount of information that flows across the network is quite high. A simple sniffer that just captures all the data flowing across the network and dumps it to the disk soon fills up the entire disk especially if placed on busy segments of the network. Analysis of this data for different protocols and connections takes considerable time and resources. Furthermore the entire data is usually not of interest to the user. Moreover, it would be desirable to gather data flowing across the network so that the

privacy of individuals who are accessing and dispensing data through the network is not compromised. It is therefore necessary to filter, on-line, the data gathered by the "promiscuous" network adapter.

Current day sniffers often come coupled with a filter that is provided filtering criteria for dumping packets to the disk. Rather than merely identifying packets based on low level characteristics such as packet source and destination, current sniffers can decode data from the various layers of the Open System Interconnection (OSI) network stack. Subsequent discussion focuses on the filtering mechanisms used in these sniffers.

The first level of filtering that can be applied on packets flowing across the network is based on the network parameters of that packet viz. the MAC addresses, IP addresses, protocols, and port numbers. Since the packet would first be available to the kernel before being handed over to the user application that is filtering the packets it is desirable to have in-kernel filtering of packets. With in-kernel filtering several packets would be rejected by the kernel and a context switch would not occur for each packet. This would speed up the filtering process. Currently in-kernel filtering is supported only for the basic network parameters and does not extend to the application level.

The second level of filtering is based on criteria specific to an application. For instance – email-ids for the Send Mail Transfer Protocol (SMTP) [25], user names for File Transfer Protocol (FTP) [38] and host names for Hypertext Transfer Protocol (HTTP) [17]. Since there is no support in the kernel for handling these parameters a user level application handles such filtering.

The third level of filtering is based on the content present in the application pay load. For instance it may be desired to search for the presense of a text string in a file transferred during a FTP session. Such filtering also needs to be handled by the user level application.

An interesting issue arises when in-kernel filtering is combined with user level filtering and the nature of application is such that the in-kernel filter has to dynamically change. In such cases the overhead for dynamically generating and using the in-kernel filter has to be considered. This is discussed in more detail in Chapter 3

and Chapter 4.

2.2.1 In-Kernel Filtering

In-kernel filtering as discussed above can filter packets based on network parameters present in the protocol headers of packets. The first among the chain of such filters was the CMU/Stanford Packet Filter [27] that evolved into Network Interface Tap(NIT) [33] under the SunOS 3 and later into BSD Packet Filter (BPF) [26]. BPF developed by Steve MacCane and Van Jacobson comprises of two components – the filter code and an interpretor for the code. The BPF interpretor assumes a pseudo machine with an accumulator, an index register, a scratch memory store and an implicit program counter. Simple functionality like Load, Store, Branch, Return etc. akin to assembly language is provided.

BPF [26] outperforms its successor CSPF [27] because firstly it filters packets based on a directed acyclic Control Flow Graph (CFG). CSPF [27] uses a boolean expression tree for the same. NNstat [43] was the first to use CFG for representing filtering expressions. Though the two models of computation – CFG and boolean expression tree – are equivalent the former is well suited for register based machines while the latter is suited for stack based machines. Moreover, the number of comparisons required by the former model for packet filtering can be shown to be less than the number of comparisons required by the latter. The NIT [33] model on the other hands copies packet that result in degradation of performance whereas BPF [26] does not copy packets. In kernel copying is done only in case of matches in BPF [26].

The Linux Socket Filter (LSF) [40] is derived from BPF [26] for machines using the Linux operating system.

2.3 Post-Processing Sniffed Data

Sniffers normally dump the packets that they capture directly to the disk. These packets usually require post capture processing to render them humanly readable.

Most sniffers provide various post-processing and rendering tools. Sniffers that provide statistics about the data captured with the sole purpose of helping network managers in diagnosing and evaluating performance problems with servers, the network wire, hubs and applications are usually called network monitoring tools. Traditionally such tools set up alerts on various events, show trends of network traffic over a time period and maintain some history information. Sometimes a monitoring tool is just a tool that can monitor any data flowing on the network.

2.4 Defending Against Sniffers

Several well known defenses exist for thwarting sniffing programs. Changing over from a "hubbed" to "switched" network is an effective method for guarding against casual sniffing. However, this method cannot be completely relied upon as switched networks can be compromised through spoofing of IP and MAC addresses, and spoofing of ARP packets. Moreover, the entire Internet can not be guaranteed to be switched. Other methods of defending against sniffing is encrypting the data flowing across the network. This method does not prevent sniffing. Rather, it makes decoding of captured data extremely difficult. SSL (Secure Sockets Layer) [18], PGP (Pretty Good Privacy) [2] and S/MIME(Secure Mime) [14], ssh (secure shell) [49], and Virtual Private Networks (VPNs) [28] are some of the techniques for encrypting data flowing across the network. Similarly, secure authentication mechanisms like Kerberos [32, 24] can prevent passwords from flowing across the network. Again these methods may not be available throughout the Internet.

2.5 Detecting Sniffers

It should be impossible to detect sniffers as they are passive listeners and do not inject anything into the network. However, sniffers configured on machines serving other functions can be detected. The basic idea behind most detection methods is to get an unexpected reply to say a ping, ARP, and source route packet. The time for a machine to respond to a ping after and before a net is loaded with

spurious traffic can also serve as a good detection method. Apart from that decoy machines can be set up to trap IPs sniffing passwords when the sniffed information is used. Sometimes Time-Domain Reflectometers can also be used. AntiSniff [1], CPM (Check Promiscuous Mode) [47], ifstatus [9] and sentinel [4] are some tools for detecting sniffers. Apocalypse Security [34] apart from having several sniffing and anti sniffing utilities also has an antiantisniffing utility.

2.6 Sniffers: Product Survey

Several commercially and freely available sniffers exist currently. Sniffers come in different flavors and capabilities for different Operating Systems. This section briefly discusses some of them.

Ethereal [15] is a UNIX-based program that also runs on Windows. It comes in both a read-only (protocol analyzer) version as well as a capture (sniffing) version. The read-only version is for decoding existing packet captures. WinDump [11] is a version of tcpdump for Windows that uses a libpcap-compatible library called WinCap. Network Associates Incorporated [31] have a range of sniffers including VOIP (Voice over IP) sniffers. Microsoft's WinNT Server comes with a built-in program called "Network Monitor". This can be added through the Networking control panel, by adding the service "Networking Monitor Tools Agent". Once installed, this tool can be run from the program menu under "Administrative Tools". BlackICE [5] is an intrusion detection system that can also log captured packets to disk in a format that can be read by other protocol analyzers. This may be more useful than a generic sniffing program when used in a security environment. EtherPeek NX [16] is a real time frame decoding and diagnostics tool and can be used both in the Windows and Macintosh environments. Triticom [46] have a suit of products that include application-level decoders and other monitoring software. Analyzer [10] is a public domain protocol analyzer with a toolkit for doing various kinds of analysis using the WinPcap library. The oldest utility in UNIX systems for sniffing packets is tcpdump [22] based on Berkely Packet Filters (BPF) [26]. An old utility called "snoop" is also used in Sun Solaris machines. It is much less capable than tcpdump,

but it is better at Sun-specific protocols like NFS/RPC. Snoop's tracefile has been specified in RFC 1761 [6]. It can be converted to tcpdump/libpcap [22, 48] format via many utilities, including 'tcptrace'. "Sniffit" [7] is a utility for analyzing application-layer data. The Trinux [45] Linux security toolkit bundles several utilities including sniffit, tcpdump and snort. SuperSniffer v1.3 [42] – to quote from the site is "an enhanced libpcap [48] based packet sniffer with many modifications like DES encryption of log file, traffic can be logged by regular expression pattern matching, POP and FTP connections are logged on one line, telnet negotiation garbage is discarded, duplicate connections are discarded, tcp packet reassembly, parallel tcp connection logging. Daemon mode where logs are dumped to specified port with authentication. Duplicate POP/FTP connections are not logged. Compiles under most operating systems, uses GNU autoconf". Klos Technologies [12] provide PacketView and Serial View on the DOS platform for sniffing packets on LAN (Local Area Networks) and PPP (Point-to-Point Protocol) connections respectively. The Gobbler and Beholder [19] is another DOS based tool for sniffing. The host site [19] has several security related tools. CMA 5000 [30] is a multi-layer network test platform. The nGenius [29] suit of tools is for non intrusive, real-time monitoring of the network and includes content analysis.

Carnivore [41, 20, 21] is a tool developed by the FBI. It can be thought of as a tool with the sole purpose of directed surveillance. This tool can capture packets based on a wide range of application-layer level based criteria. It functions through wire-taps across gateways and ISPs. Carnivore is also capable of monitoring dynamic IP address based networks. The capabilities of string searches in application-level content seem limited in this package. It can only capture email messages to and from a specific user's account and all network traffic to and from a specific user or IP address. It can also capture headers for various protocols.

PickPacket the focus of this thesis and also discussed in Reference [23] is a monitoring tool similar to Carnivore. This sniffer can filter packets across the levels of the OSI network stack for selected applications. Criteria for filtering can be specified for network layer and application layer for applications like FTP [38], HTTP [17], SMTP [25] etc. It also supports real-time searching for text string in application and

packet content. Unlike Carnivore, currently it does not have the ability of capturing packets by discovering IPs in a dynamic IP address based network. However, it is planned to extend PickPacket's capabilities to meet this requirement. Searching for content in MIME and Base64 encoded data is also proposed.

Chapter 3

PickPacket: Architecture and Design

This chapter discusses the design of PickPacket with special attention to the filtering in PickPacket. First the recommended architecture for PickPacket is discussed and its various components are identified. Discussion on these components is then undertaken with a view to elaborate on the design of the filtering mechanisms in PickPacket. Detailed design and implementation details are discussed in Reference [23].

3.1 The Architecture of PickPacket

PickPacket can be viewed as an aggregate of four components ideally deployed on four different machines. These components are – the *PickPacket Configuration File Generator* deployed on a Windows/Linux machine, the *PickPacket Filter* deployed on a Linux machine, the *PickPacket Post Processor* deployed on a Linux machine and the *PickPacket Data Viewer* GUI deployed on a Windows machine. An architectural view of PickPacket is shown in Figure 3.1 where these components are shown in rectangles. Initially, criteria are given to the PickPacket Filter through the PickPacket Configuration File Generator GUI. This generates a configuration file for the PickPacket Filter based on which the filter captures the packets. In the envisaged scenario of usage, the PickPacket Configuration File Generator would prepare a configuration file that would be transferred to the machine where the PickPacket

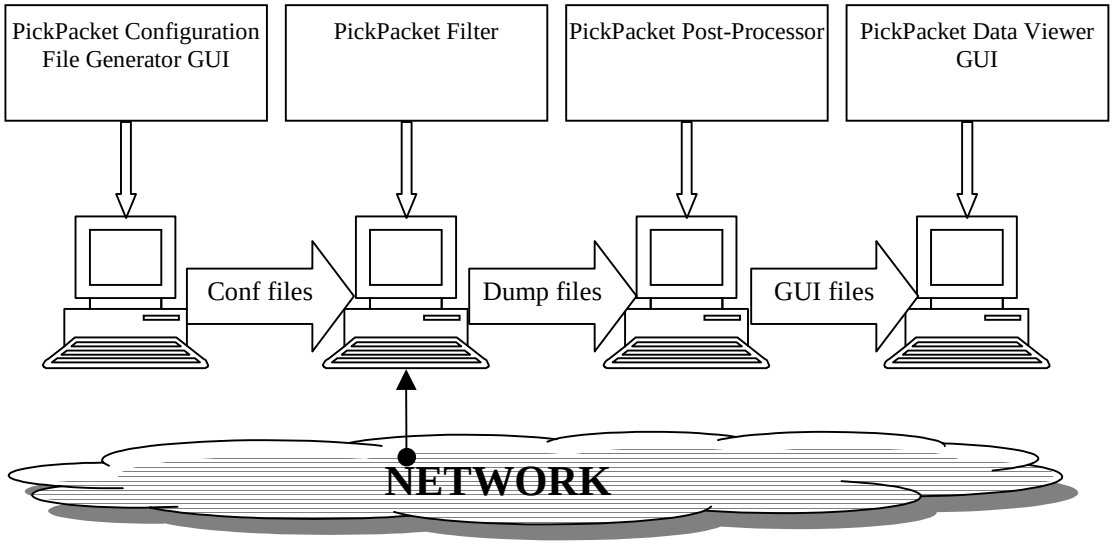


Figure 3.1: The Architecture of PickPacket

Filter would run. The PickPacket Filter captures packets according to the criteria specified in the configuration file and stores them to some storage device. Again it is advisable, though not necessary, that this device be some removable permanent storage. Then the removable permanent storage is taken offline for post processing and analysis. The PickPacket Post Processor would typically run on some machine other than the one on which the PickPacket Filter is running. The task of the Post Processor is to break the dumped data into separate connections and retrieve that information from the captured packets which is necessary for showing the captured data through a user friendly windows based GUI. After post processing and analysis a separate PickPacket Data Viewer GUI shows the results.

3.2 The PickPacket Configuration File Generator

The PickPacket Configuration File Generator is a java based graphical user interface (GUI) that generates the configuration file that is input to the PickPacket Filter. This file is a text file with HTML like tags. A sample configuration file is given in

Appendix A. This file has four sections.

1. The first section contains specifications of the output files that are created by the PickPacket Filter for saving packets. It allows specification of multiple output files and their maximum sizes. A feature in the configuration file is the support for different output file managers. This feature would be useful if output has to be dumped in formats other than the default pcap [48] style format.

2. The second section contains criteria for filtering packets based on source and destination IP addresses, transport layer protocol, and source and destination port numbers. The application layer protocol that handles packets that match the specified criteria is also indicated. This information is required for demultiplexing packets to the correct application layer protocol filter.

3. The third section specifies the number of simultaneous connections that should be monitored for any application. This is used for space allocations.

4. The fourth section comprises of multiple subsections, each of which contains criteria corresponding to an application layer protocol. Based on these criteria the application layer data content of the packets is analyzed. Specifications for filters for SMTP [25], HTTP [17], and FTP [38] can also allow the user to specify the number of history packets to keep when content of such applications is being filtered for text strings.

3.3 PickPacket Packet Filter: Basic Design

The PickPacket Packet Filter reads packets from the network. It matches these packets against the criteria specified by the user. Packets that successfully match the specified criteria are stored on some storage media for further analysis. This section presents the design of the PickPacket Filter.

A typical filter can have several levels at which it filters packets:

1. Filtering based on network parameters (IP addresses, port numbers, etc.)

2. Filtering based on application layer protocol specific criteria (user names, email-ids, etc.)

3. Filtering based on content present in an application payload.

Usually the first level of filtering can be made very efficient through the use of in-kernel filters [26]. Since the content of application can be best deciphered by the

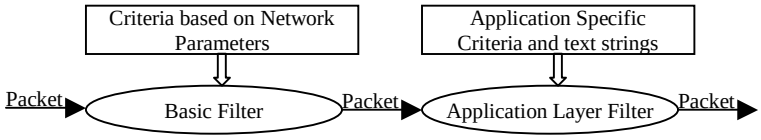


Figure 3.2: Filtering Levels

application itself, the second and third levels of filtering are combined. Figure 3.2 captures this notion of levels of filtering. In this figure the first level of filtering is named *Basic Filter* and the combined second and third level filtering has been named *Application Layer Filter*. The Basic Filter takes as input the packet and the network parameters based criteria and the Application Layer Filter takes as input the application specific criteria and search strings.

Since it would be convenient to have different filters for different application layer protocol based filters, the combined second and third level filtering can be split into several application specific filters – one for each application. If this model of filtering is chosen a *demultiplexer* is required between the first level filter and the application specific filters so that each application gets only relevant packets. This refinement is captured by Figure 3.3. The demultiplexer uses its own set of criteria for demultiplexing packets.

Finally, 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

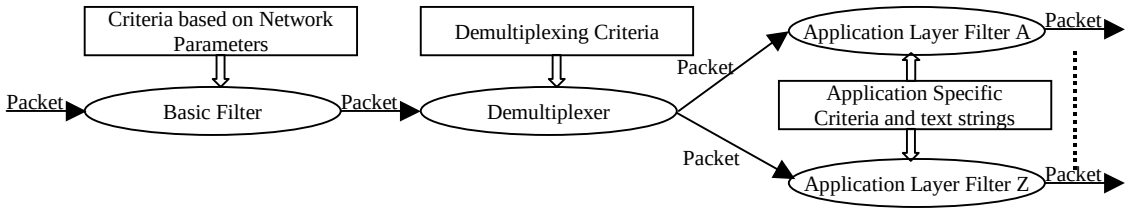


Figure 3.3: Demultiplexing Packets for Filtering

packets that are received out of sequence while performing the search for split text. Thus a component that does these checks is introduced as another refinement to the filter above. This component is called the *TCP Connection Manager*. This component is common to all application level filtering that allow searching for text strings in the application pay load. This level of refinement is captured in Figure 3.4. There are several considerations that go into designing the connection manager.

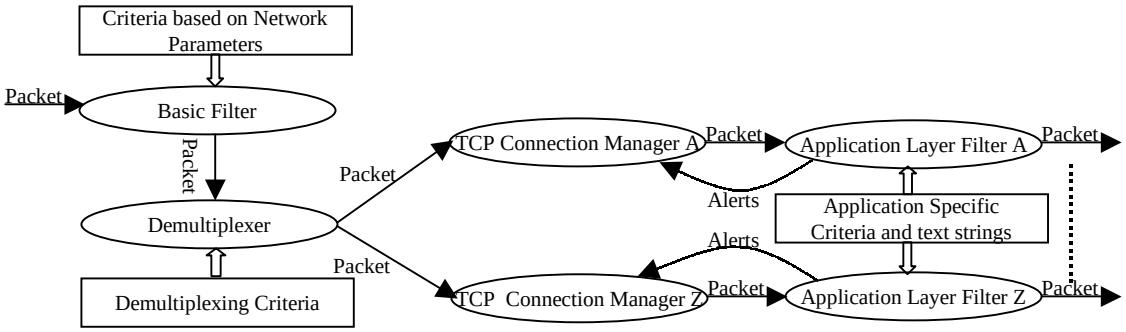


Figure 3.4: Some Components of a Filter

First the connection manager need not determine the sequencing of packets for all connections. Rather, it should determine sequencing for only those connections that an application layer filter is interested in. Communication between the application layer filter and the connection manager to indicate such interest is provided by means of *alerts*. A second consideration pertains to the level at which history data is remembered for an application. A cursory design would store remembered data at

the application layer level. Searching for this data is done based on the four tuples (source IP, destination IP, source port and the destination port). However this four tuple is also examined by the demultiplexer. States dependent on this four tuple are also maintained by the connection manager. Therefore it is best to pass the data that the application wishes to associate with a connection to the channel manager and subsequently to the demultiplexer. *Alerts* also incorporate this mechanism.

The discussions above lay the foundation for the basic design of the PickPacket Filter. Figure 3.5 shows the basic design of the PickPacket Filter. All the criteria

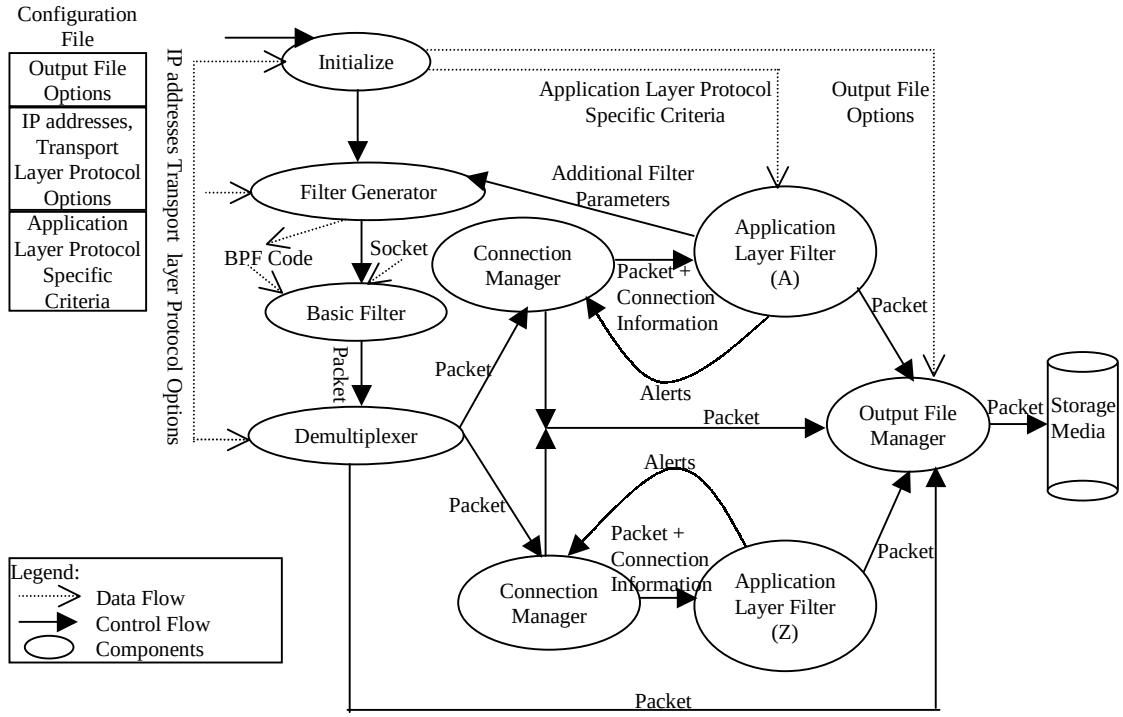


Figure 3.5: The Basic Design of the PickPacket Filter

input to various components are gathered into a *configuration file*. A component *Initialize* is added for initializations dependent on the configuration file. Another component the *Output File Manager* is added for dumping filtered packets to the disk. A *Filter Generator* is added for generating the in-kernel BPF [26] code. Hooks are provided for changing the BPF code generated. Functions that can generate the

filter code based on changed parameters can be called by applications such as FTP [38] during "PASSIVE" mode of file transfers. The reasons for having this ability in the PickPacket Filter is discussed at length in Chapter 4 .The *Demultiplexer* is provided the facility of calling the *Output File Manager* directly so that the filter can directly dump packets without resorting to application layer protocol based filtering, if necessary. The *Connection Manager* can also directly dump packets to the disk. This is required when all criteria have matched for a specific connection and the connection is still open. More details of these components can be found in Reference [23].

3.3.1 PickPacket Filter: Output File Formats

Conceptually, the *output file manager* can store files in any format. However, PickPacket stores output files in the *pcap* [48] file format. This file starts with a 24 byte pcap file header that contains information related to version of pcap and the network from which the file was captured. This is followed by zero or more chunks of data. Every chunk has a packet header followed by the packet data. The packet header has three fields – the length of the packet when it was read from the network, the length of the packet when it was saved and the time at which the packet was read from the network.

3.3.2 PickPacket Filter: Text String Search

The PickPacket Filter contains a text string search library. This library is extensively used by application layer filters in PickPacket. This library uses the *Boyer-Moore* [39] string-matching algorithm for searching text strings. This algorithm is used for both case sensitive and case insensitive search for text strings in packet data.

3.4 The PickPacket Post-Processor

The packet filter writes filtered packets to an output file that is analyzed offline to separate packets into their respective connections. The output file generated by the packet filter needs to be processed to analyze the captured data. This processing includes separating packets based on the transport layer protocol and the application layer protocol. As stated in Reference [23] the Post-Processor should meet the following objectives:

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.

Three components – the *Sorter*, the *Connection Breaker*, and the *Meta Information Gatherer* are involved in the post-processing of captured packets. These are shown in Figure 3.6. 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. The Sorter, for this reason, sorts the packets present in the output file based on a time stamp value corresponding to the time the packets were read off 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. Internally

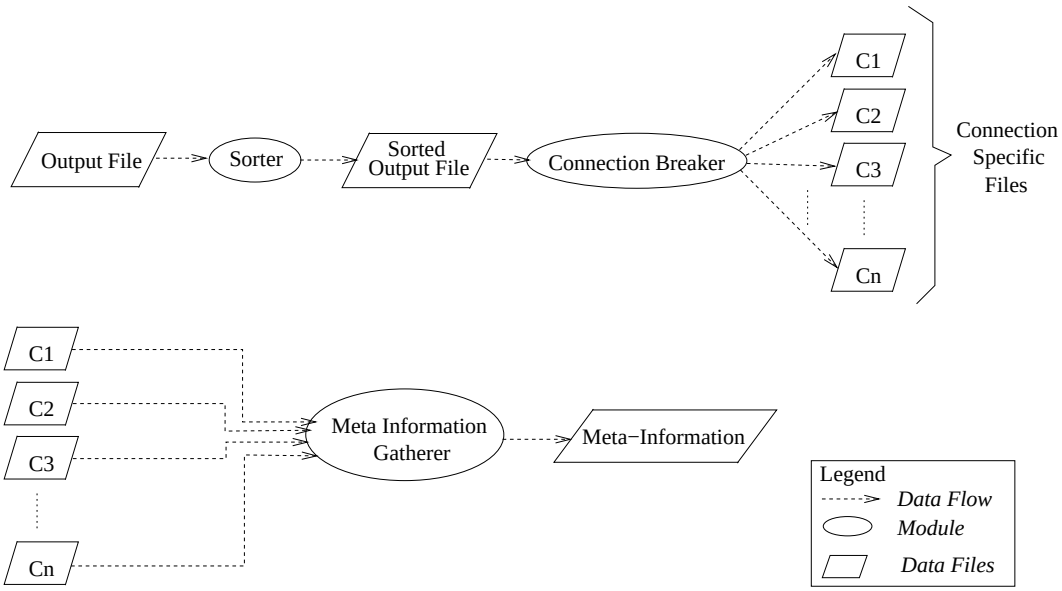


Figure 3.6: Post-Processing Design [23]

connection breaking is accomplished by a TCP [36] state machine based process. Packets belonging to a connectionless protocol like UDP [35] are separated based on the communication tuple. Now the meta information gathering module reads these connection specific files and retrieves the meta-information of every connection. Each application requires different meta-information and packets belonging to a particular application are processed by meta-information gathering modules for that application. This meta-information serves as an input to the application providing the user-interface. Further details are given in Reference [23].

3.5 The PickPacket Data Viewer

The PickPacket Data Viewer is used for rendering the post-processed information. This is a Visual Basic based GUI and runs on Windows. The choice of this platform was made for rapid prototyping and the rich API (Application Program Interface) library that is provided in Windows for rendering content belonging to an application. Initially the Data Viewer lists all connections by application type, source

and destination IP addresses and other such fields based on the meta-information that has been provided by the Post-Processor. These connections can be sorted and searched based on these fields. The Data Viewer also allows examining the details of a connection and can show the data for that connection through appropriate user agents commonly found in the Windows environment.

3.6 Final Remarks

PickPacket is a useful tool for gathering and rendering information flowing across the network. The design of PickPacket is modular, flexible, extensible, robust and efficient. Judicious use of the system can also help protecting the privacy of individuals and can dump only necessary data to the disk. Tools for Post-processing and subsequent rendering make the tool easy to use. The universality of the capture file formats offer the user a choice of using "rendering and post-processing tools" other than those provided by PickPacket.

The rest of this thesis focuses on two specific application layer filters of PickPacket – the filter based on the File Transfer Protocol [38] and the filter based on the Hypertext Transfer Protocol [17].

Chapter 4

Design and Implementation of the FTP Filter in PickPacket

This chapter discusses the design and implementation of the application layer filter in PickPacket that is based on the File Transfer Protocol (FTP) [38]. First the protocol itself is briefly described with special focus on those features of the protocol that directly impact the design of the PickPacket Filter. Then the design and implementation details of the application layer protocol filter are presented.

4.1 FTP Abstractions

Figure 4.1 shows the key abstractions of an FTP communication and their relationship to each other.These abstractions include the User Interface (UI), the Protocol Interpretor (PI), the FTP commands and replies, the Data Transfer Process (DTP), the files being transferred, the TCP based *command connection* and the TCP based *data connection*.

The User Interface offers a front end to the user. The Client Protocol Interpretor interprets the commands entered by the user and initiates a TCP based control connection to the server on the reserved FTP control port – 21. The port on the client side is chosen arbitrarily. Commands entered by the user are sent to the server over this connection. The Server Protocol Interpretor is responsible for interpreting

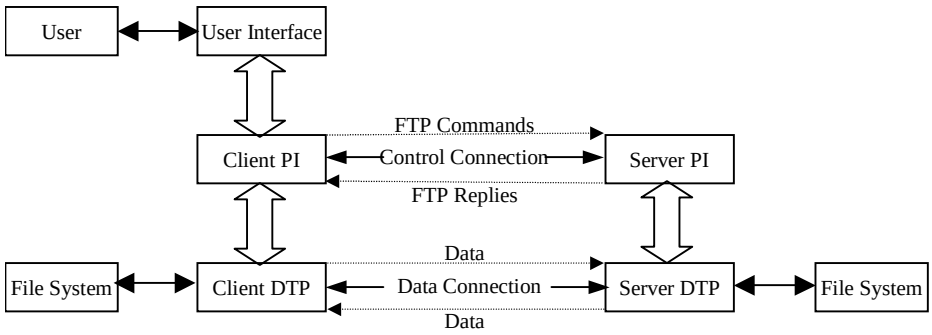


Figure 4.1: The Key Abstractions of FTP Communication

and executing the commands received. When data needs to be transferred between the server and the client a separate TCP based data connection is opened between the server and the client by the Data Transfer Process. The data connection may be initiated by the client or by the server depending on the sequence of commands issued. Sometimes the client can start a proxy connection between the server and some other machine. In such cases the data transfer occurs between the server and the other machine while control connections are open between the client and other machines. It is these mechanisms of data transfer that impact the design of the PickPacket Filter. There are several commands and replies that can be sent across the control connection. However, the focus of subsequent discussions will be the primary commands of the data transfer process and their impact upon the design of PickPacket.

4.2 FTP: File Transfer Methods

There are three methods of file transfer in FTP depending on the sequence of commands issued by a client after the control connection has been opened and user credentials have been established. In the first method the server initiates the data connection to a port designated by the client, in the second method the client initiates the data connection to the port designated by the server and in the third some machine other than the client initiates the data connection to a port designated

by the server. These methods are named normal, passive and proxy respectively. Figure 4.2 shows these methods of file transfer. Description of the methods is given

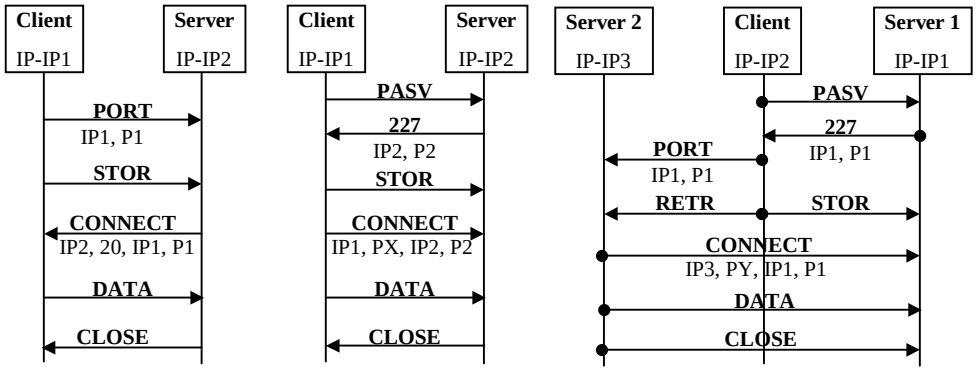


Figure 4.2: File Transfer Methods in FTP

in the subsections that follow. The "STOR" data retrieval command is discussed. The other important data retrieval command "RETR" is identical to "STOR" except that data is transferred from the server to the client across the data connection.

4.2.1 Normal Method of File Transfer

Conceptually, the steps for the normal method of file transfer are as follows:

1. The client issues a "PORT" command on the control connection. This command includes the IP of the client and the port that the client designates for the data transfer.

2. The server retrieves the port and the IP and indicates that the command is correct.

3. The client issues a "STOR" command that specifies a pathname for the file to be transferred.

4. On receiving the "STOR" command the server opens a data connection through the default data port on the server side (20) to the designated IP and port stored in step 2 and informs the client about the opening of the data connection on the control connection.

5. The client sends the data to be stored through the data connection.

6. The server stores the file to the pathname supplied by the "STOR" command and closes the data connection.

4.2.2 Passive Method of File Transfer

Conceptually, the steps for the passive method of file transfer are as follows:

1. The client issues a "PASV" command on the control connection.

2. The server reply contains the IP of the server and a port that the server designates for the data connection.

3. The client issues a "STOR" command that specifies a pathname for the file to be transferred and the server replies that the command is all right.

4. The client opens the data connection to the specified port on the server. The client can choose whatever port happens to be free.

5. The client sends the data to be stored through the data connection.

6. The server stores the file to the pathname supplied by the "STOR" command and closes the data connection.

4.2.3 Proxy Method of File Transfer

The proxy method of file transfer applies to three machines, the client (C), the first server ($S1$) and second server ($S2$). The proxy method of file transfer requires the following steps:

1. C opens a control connection with $S1$.

2. C opens a control connection with $S2$.

3. C sends the "PASV" command to $S1$.

4. $S1$ sends its IP and a port to client.

5. C sends a "PORT" command to $S2$ that has the IP and port supplied by $S1$. At this point of time $S1$ is listening on the port specified in the previous step and $S2$ is ready to connect to the IP and port specified.

6. C sends a "STOR" to $S1$ and a "RETR" to $S2$.

7. $S2$ connects to $S1$ on the IP and port retrieved and sends the required file.

4.3 Transfer Methods and PickPacket Filter Design

This section discusses the impact of the file transfer methods on the design of the PickPacket Filter.

Assume that a client ($C1$) with the IP address $IP1$ has to be monitored for file transfers to and from a server ($S1$) with the IP address $IP2$. The basic filter discussed in Chapter 3 is set up to monitor communications on any port from IP address $IP1$ to port 21 and port 20 and IP address $IP2$ and demultiplex packets corresponding to these tuples to the FTP filter. Basically the $IP-PORT-IP-PORT$ four tuple of $[IP1, *, IP2, 21]$ and the four tuple $[IP1, *, IP2, 20]$ would be monitored. Here "*" stands for any port or any IP address depending on context. This strategy works correctly for the normal method of file transfer. If the client sends some other port say PX in the PORT command, data will be transferred between the four tuple $[IP1, PX, IP2, 20]$. This is covered by the tuple $[IP1, *, IP2, 20]$.

However, in the passive method of file transfers, the server will reply to the "PASV" command by giving a port – say PY. Suppose that the client chooses the port PZ to establish the data connection. The corresponding four tuple for data communication would become $[IP1, PZ, IP2, PY]$. This is not covered by any of the tuples that are monitored and transferred data would be dropped by the basic filter and never reach the application layer FTP filter. Changing the monitored

tuples to a single tuple $[IP1, *, IP2, *]$ does not help as the set becomes too general and packets belonging to some other application are also demultiplexed to the FTP filter. In general it is advisable to keep the monitored set of tuples as restrictive as possible.

The only option left, is to add the tuple $[IP1, PX, IP2, PZ]$ to the tuples being monitored, as and when PX and PZ are discovered. Since PX can be known only when the client actually connects it is better to add the tuple [IP1, *, IP2, PZ] rather than the tuple $[IP1, PX, IP2, PZ]$ and demultiplex all connections matching this tuple to the FTP filter.

When the proxy method of file transfers is considered the tuple for data communication instead of being $[IP1, PX, IP2, PZ]$ would be $[IP3, PU, IP2, PZ]$. Since the issuing of the PASV command does not guarantee that the reply would not be used for the proxy method of file transfer, it is best to monitor the tuple $[*, *, IP2, PZ]$ and demultiplex packets belonging to this tuple to the FTP filter. This would handle both the passive and the proxy methods of file transfers.

The requirement that new tuples be added to the Basic Filter of PickPacket as and when such tuples are discovered has interesting implications. First, provisions should be made in the filter to add these tuples on the fly. Also, every time such a tuple is added the BPF filter code generated has to change. Provisions have to be made for generating the BPF code.

When BPF code is generated it is attached to a socket from which the packets are being read. If some other code is attached to the socket then it has to be removed. This enforces the following sequencing on the attachment, regeneration and detachment of the BPF code. First any BPF code that has been attached is detached from the filter. New parameters for generating the BPF code are inserted. Then, the BPF code is regenerated. This code is attached to the socket. Thus from the time the BPF code is detached to the time BPF code is reattached in-kernel filtering is disabled. The demultiplexer is responsible for discarding spurious packets collected during this period.

A consequence of this strategy is that an overhead has to be paid for regenerating the BPF code. This overhead typically boils down to about 10 to 15 milliseconds.

During this period packets have to be stored in the buffer attached to the filter. The size of this buffer has to be fine-tuned [44]. Even then, in the worst case, if every – say alternate – packet happens to be a "PASV" command then more time would be spent in generating the BPF code. In such scenarios packets would be dropped. The alternative to this is not to do any in-kernel filtering. This would result in a context switch for every packet and slow the overall performance of the filter and could again lead to dropping of packets. Currently in-kernel filtering has been chosen. This may lead to dropping of packets in pathological cases.

4.4 FTP Filter: Goals

The FTP Filter in PickPacket is designed to capture FTP packets flowing across a network segment according to the criteria specified by the user. Provisions have been made for specifying the criteria – user names, file names and text strings. The user can also specify the mode of operation "PEN" or "FULL". Initially a connection is examined for the match of the user name. Then the file transfer commands are checked for the match of a file name. Finally if both the previous criteria match, the text string specified is searched in the data connection.

Depending on the mode of operation – "PEN" or "FULL" – the amount of information dumped to the disk is different. In the "FULL" mode, packets of the control connection corresponding to the matched criteria are dumped to the disk and the data connection is also dumped to the disk. In the "PEN" mode of operations, only those packets of the control connection are dumped to the disk that match the user specified criteria. Further, in the "PEN" mode, the password of the user is replaced by "X" and the data connection is not dumped to the disk. The user can also specify the number of history data packets to store while searching for text strings.

4.5 FTP Filter: Command Sequences

Packets flowing across the control connection have been divided into several sequences for the purpose of filtering. Commands in FTP [38] are telnet [37] style

commands, and replies are numbers followed by descriptive text. A sequence is defined as a set of commands and their replies. Important sequences defined in the FTP filter are:

The Login Sequence consists of the command and replies that establish the credentials of a user for the FTP server. This sequence consists of the "USER", "PASS", and sometimes the "ACCT" commands and their replies. The end of a successful login sequence is indicated by the 230 reply.

The Type Sequence is used for defining the type of file being transferred ASCII or EBCDIC. A 200 reply marks a successful completion of the type sequence.

The Mode Sequence can be of the type stream, block or compressed. A 200 reply ends a mode sequence.

The Port Sequence always marks the beginning of a data transfer command. The sequence consists of the "PORT" and "STOR" or "RETR" or "STOU" command and their replies. A data connection is also established during this command. Finally when the file transfer is over, a 226 reply marks a successful file transfer.

The Passive Sequence always marks the beginning of a data transfer command. The sequence consists of the "PASV" and "STOR" or "RETR" or "STOU" command and their replies. A data connection is also established during this command. Finally when the file transfer is over, a 226 reply marks a successful file transfer. The 227 reply to the "PASV" command includes (h1,h2,h3,h4,p1,p2) where h1 to h4 are the bytes of the host IP address and p1 and p2 are the bytes of the port that the server will listen on for a connect from a client.

The Plain Data Transfer Sequence is very rarely used. It is identical to the "PORT" sequence except that the "PORT" command is not sent and default ports are used for transferring files.

The last four sequences listed above are data transfer sequences. The Mode and Type sequences define the parameters of file transfer. The parameters can be changed by the client of an FTP server.

4.6 FTP Filter: Design and Implementation

The design and implementation of the FTP Filter evolves around the command sequences identified in the previous section and the file transfer methods discussed. This section describes the design and implementation of the FTP Filter. The impact of the file transfer methods on the design of the PickPacket Filter was discussed in Section 4.3. This section is further divided into two subsections. The first subsection discusses the handling of control connections and the second subsection discusses the handling of data connections.

The FTP Filter maintains a structure that captures the state of a FTP connection. It allocates this structure for each connection and maintains a list of these structures. In subsequent discussions this list is referred to as "FTP_GSL". The structure that this list contains is referred to as "FTP_STR".

4.6.1 Handling Control Connections

The structure for a FTP connection, "FTP_STR", maintains another list that corresponds to the packets transferred on the control connection. The sequences identified in Section 4.5 occur in this list. Markers to this list that point to the start and end of sequences are maintained. Markers to the beginning and end of the sequence that the FTP Filter is currently processing are also maintained. Whenever a particular sequence completes successfully, the old sequence is removed from the list and the markers for that sequence are adjusted to point to the beginning and the end of the current sequence. On the completion of a data transfer sequence if all the criteria specified by the user have not matched, that data transfer sequence is removed from the list. Contents of the packets belonging to the control connection are examined on a packet by packet basis and the current sequence under progress is established. If a packet that starts a new sequence is received when the current sequence has not completed, the current sequence is removed from the list. Relevant packets are also checked for match of the user specified criteria. The structure "FTP_STR" also contains variables that record the match of criteria supplied by the user.

The exact command and replies are determined by parsing the command and replies flowing across the control connection. Parsers for decoding command and replies have been provided in the FTP Filter. A function extracts the ports and IP addresses from the "PORT" command and replies to the "PASV" command. Sequence sub states are maintained for checking the correctness of command sequences.

Whenever, a "PASV" or the "PORT" command are received by the FTP Filter the IP and the port information is extracted from these commands. The IP and the port information supplied by these commands should form the destination/source IP and the destination/source port for data connections. This information is added to the structure "FTP_STR" so that the list of these structures, "FTP_GSL", can be searched based on these entries. Moreover, on receiving a reply to the "PASV" command the basic filter of the PickPacket Filter is changed in a sequence of steps as outlined in Section 4.3. When a "PASV" command completes successfully the parameters of the BPF filer that include filtering based on the contents of the reply are changed. However, the BPF filter is not immediately recompiled. Rather, the BPF filter is recompiled when some new parameter is added to the BPF filter because of say another PASV command. This seems to be a reasonable optimization.

4.6.2 Handling Data Connections

When a packet is passed from the *Connection Manager* to the *Application Layer Filter* the data that the latter wants to be remembered by the former is also supplied. Section 3.2 discusses this mechanism in detail. Initially, a packet arriving across the data connection has no application level data associated with it. The list of "structures associated with a connection", "FTP_GSL", is searched for a structure with a data destination port and data destination IP that matches the source/destination port and IP address of the packet. If such a structure is found in the list this structure becomes the history data associated with the data connection. If no matching structure is found in the list it implies that this packet is not of interest. Further processing of the data packet is done on the basis of the contents of the structure, "FTP_STR", thus retrieved. "FTP_STR" maintains a list of history data packets. The size of this list is provided by the user. In case of matches

of text strings in the data packet the history data packets, the control connection packets, and subsequent packets on the data connection are dumped to the disk. If the data packet does not contain the text string it is added to the list of history data packets.

This completes the discussion on the design and implementation of the FTP Filter in PickPacket Filter. It is instructive to note that the design of the FTP Filter has a telling impact on the overall design of the PickPacet Filter. Studying major protocols that have to be implemented in filters that deal with application layer content can be a useful exercise.

Chapter 5

Design and Implementation of the HTTP Filter in PickPacet

This chapter discusses the design and implementation of the application layer filter in PickPacket that is based on the Hypertext Transfer Protocol (HTTP) [17]. First, the protocol is itself described with a focus on those features that are of interest for designing and implementing the filter. The major feature of the filter is an HTTP parser for parsing the packets. This is discussed in greater detail while the design and implementation of the HTTP filter are presented.

5.1 HTTP Simplified

HTTP is the Hypertext Transfer Protocol that is used to deliver virtually all files and other data – resources – on the World Wide Web. Usually HTTP takes place through TCP/IP sockets. The HTTP client comes equipped with a browser that sends requests to an HTTP server and elicits a response in return. HTTP servers by default listen on to port 80, though they can use any port.

5.1.1 HTTP Resources

HTTP transmits resources, not just files. A resource is some chunk of information that can be identified by a Uniform Resource Locater (URL) [3]. The most common

kind of a resource can be a file, but a resource may be a dynamically generated query result, the output of a CGI script etc. When some data that is interpreted by a server is attached to the URL it is called a Universal Resource Identifier (URI) [3]. This usage is more popular with technical manuals.

5.1.2 HTTP Transactions

HTTP transactions are named requests and responses. Requests are generated by an HTTP client and responses to requests are generated by an HTTP server. The format of the request and response messages are similar. Both kind of messages consist of

- An initial line (different for request and response)
- Zero or more header lines (vary across requests and responses)
- An empty line
- An optional message body

Initial lines and headers end with a Carriage Return followed by a Line Feed (CRLF). However, lines ending with plain line feeds are also acceptable.

The initial request line has three parts – a method name, the local path of the requested resource, and the version of HTTP being used. Each part is separated by a space. Method names and versions are HTTP/x.x in upper case. A typical request line is:

```
GET /path/to/file/index.html HTTP/1.1
```

There are several possible methods such as GET, PUT, POST etc.

The initial response line is also called the status line. This line also has three parts – the HTTP version, a response status code specifying the result of the request, and a reason phrase – separated by spaces. An example status line is:

```
HTTP/1.1 200 OK
```

Header lines provide information about the request or response, or about the object sent in the message body. The header lines are in the usual text header format, which is – one line per header, of the form "Header-Name: value", ending with CRLF. It's the same format used for email and news postings, defined in RFC 822 [8]. According to this RFC, header lines have the following characteristics:

- Header lines end in CRLF, LFs are also tolerated.
- The header name is not case-sensitive (though the value may be).
- Any number of spaces or tabs may be between the ":" and the value.
- Header lines beginning with space or tab are actually part of the previous header line, folded into multiple lines for easy reading.

Thus, the following two headers are equivalent:

```
Header1: value-A, value-B
HEADER1:    value-A,
            value-B
```

HTTP 1.0 defines 16 headers, though none are required. HTTP 1.1 defines 46 headers, and one (Host:) is required in requests.

An HTTP message may have a body of data sent after the header lines. In a response, this is where the requested resource is returned to the client (the most common use of the message body), or perhaps explanatory text if there's an error. In a request, this is where user-entered data or uploaded files are sent to the server.

If an HTTP message includes a body, there are header lines in the message that describe the body. In particular –

The Content-Type: header gives the MIME-type of the data in the body, such as text/html or image/gif.

The Content-Length: header gives the number of bytes in the body.

Transfer-Encoding: header gives the type of transfer encoding in HTTP/1.1 and is another method of specifying content lengths.

5.1.3 HTTP 1.1 and the HTTP Filter

HTTP 1.1 has recently been defined, to address new needs and overcome shortcomings of HTTP 1.0. Generally speaking, it is a superset of HTTP 1.0. Improvements include:

- Faster response, by allowing multiple transactions to take place over a single persistent connection.
- Faster response and great bandwidth savings, by adding cache support.
- Faster response for dynamically-generated pages, by supporting chunked encoding, which allows a response to be sent before its total length is known.
- Efficient use of IP addresses, by allowing multiple domains to be served from a single IP address.

Additional features of HTTP 1.1 that have been addressed by the HTTP Filter are – persistent connections, chunked transfer encoding and the "HOST:" header. Persistent connection also allows pipelining of requests. Clients can send requests to the server without waiting for a response. This directly impacts the HTTP Filter as a single packet can contain multiple requests. Chunked Transfer Encoding has a direct bearing on the HTTP Filter and is discussed in more detail in the following subsection.

5.1.4 Chunked Transfer Encoding

If a response has to be sent before its total length is known the simple chunked transfer-encoding can be used. This breaks the complete response into smaller chunks and sends them in series. Such a response can be identified as it contains the "Transfer-Encoding: chunked" header.

A chunked message body contains a series of chunks, followed by a line with "0" (zero), followed by optional footers (just like headers), and a blank line. Each chunk consists of two parts:

- A line with the size of the chunk data, in hex, possibly followed by a semicolon and extra parameters that can be ignored, and ending with CRLF.
- The data itself, followed by CRLF.

So a chunked response might look like the following:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

1a; ignore-stuff-here
abcdefghijklmnopqrstuvwxyz
10
1234567890abcdef
0
some-footer: some-value
another-footer: another-value
[blank line here]
```

The length of the text data is 42 bytes (1a + 10, in hex). Footers are treated like headers, as if they were at the top of the response. The chunks can contain any binary data, and may be much larger than the examples here. For comparison, the equivalent to the above response, without using chunked encoding is shown below:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Content-Length: 42
some-footer: some-value
another-footer: another-value

abcdefghijklmnopqrstuvwxyz1234567890abcdef
```

The HTTP Filter takes into account both of these method of data transfers.

This concludes an intuitive description of HTTP. It covers most of the features that have a direct bearing on the HTTP Filter. The rest of the chapter discusses the HTTP Filter.

5.2 HTTP Filter: Goals

The HTTP Filter captures HTTP packets flowing across a network segment according to the criteria specified by the user. Provisions have been made for specifying host names, paths, and text strings that will be monitored in a HTTP connection. The user is also allowed to specify ports other than the default port - 80 - on which HTTP servers may be running. Though the host name and the path name together specify the URL, IP addresses may also be specified instead of host names. This useful especially in capturing HTTP 1.0 communication which does not accept absolute URLs in the path and does not have the "Host:" field. The user can also specify the "PEN" or the "FULL" mode of capturing packets.

Once a host name and the path has matched in some packet of a HTTP connection the message body of the HTTP request and response as well as the URI are searched for a match of the specified text string. If all the criteria specified by the user match for the connection request packets are dumped to the disk in case of "PEN" mode of capturing packets. If the mode of capturing packets is "FULL" both request and response packets are dumped to the disk. The user can specify the number of history packets to store in case the criteria specified do not fully match.

5.3 HTTP Filter: Design and Implementation

The HTTP Filter has a structure that is allocated for each connection. This structure holds the information pertaining to that connection. Important members of this structure are the response and request structures. These structures have several parse states that are set by HTTP parsers. There is a parser for parsing request packets and another parser for parsing response packets. Figure 5.1 shows the flow chart for handling of a HTTP request packet in the HTTP Filter. The basic idea

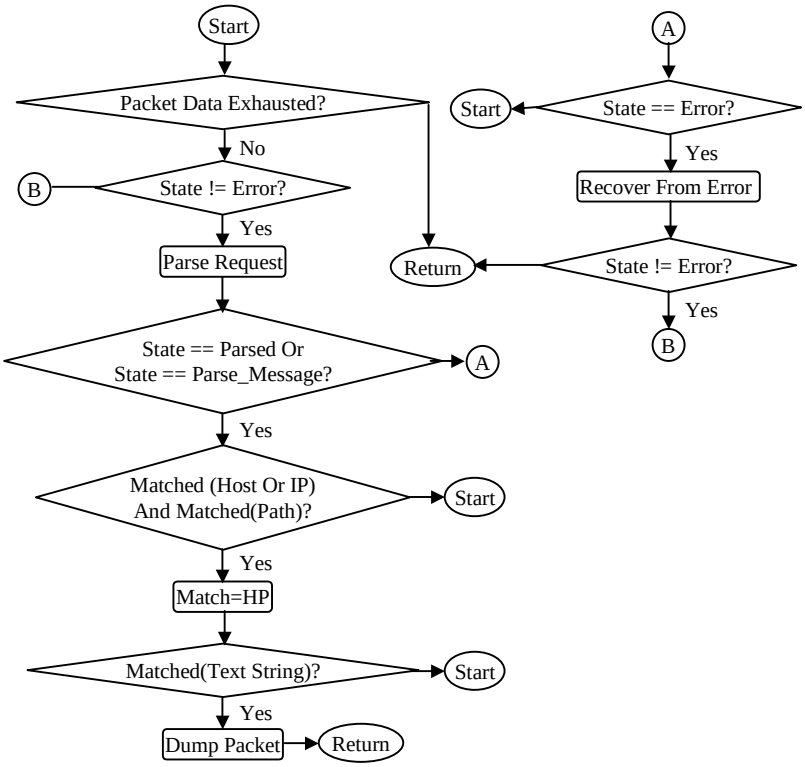


Figure 5.1: Handling a HTTP Request Packet

behind the flowchart is to parse the packet in a loop till packet data is exhausted. The parser for the request consumes the packet data and returns after setting states for the request structure discussed above. Data may be left in the packet after parsing because of pipelining or errors. Graceful error recovery mechanisms have been provided in the handling of the packets. After the parser returns further processing is necessary if parsing has either parsed an entire request or has retrieved partial content of the request. The parser may be able to retrieve partial content in cases where the message body of the request is split across packets. Under these conditions, the data retrieved from the packet by the parser is checked for match of user supplied criteria. If the criteria match the connection can be dumped otherwise, if the entire packet data has been exhausted, the packet can be put into a list of history packets. Requests are handled similarly except that checking is done only for

text strings and that too only if the state of match has already been set to indicate a host as well as a path match on a previous handling of some request. Figure 5.2 shows the handling of response packets by the HTTP Filter.

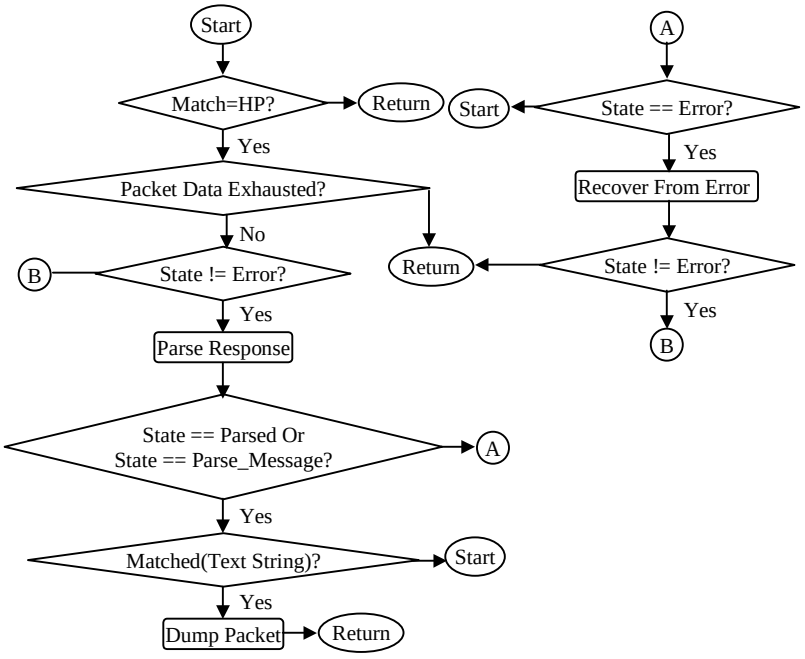


Figure 5.2: Handling a HTTP Response Packet

5.3.1 Parsing HTTP Packets

The parser of request and response packets forms the heart of the HTTP filter. The structure of HTTP transactions has already been discussed in Section 5.1.2. Two, major requirements have to be met while parsing HTTP packets. First, a packet can have more than one requests. Second - a request or response may be split arbitrarily across packets.

The parse defines parser states based on the structure of HTTP transactions. Thus for a request the parser can set the states - NONE, PARSE_REQ_LINE, PARSE_HEADER, PARSE_MESSAGE, PROCESSED, and ERROR. Similarly while parsing responses the parser can set the states - NONE, PARSE_RES_LINE,

PARSE_HEADER, PARSE_MESSAGE, PROCESSED, and ERROR. Corresponding to these states the parser can set several sub states that define the amount of parsing of a particular line by the parser. For instance, sub states like GETTING_METHOD, GOT_METHOD, GETTING_URI etc are defined. The parser also defines sub sub states for parsing the CRLF at the end of the lines of an HTTP request or response.

States are initialized once to kick off the parser. After that the parser examines each packet and sets appropriate states. Subsequent calls to the parser use the old state that has been set by the parser. In case an ERROR state is set the HTTP Filter tries to recover from this state by skipping to the first method or the first response in the packet depending on context. This strategy takes care of the second requirement placed on the parser. The first requirement is met by calling the parser in a loop till the entire data of the packet has been consumed by the parser. The parser, while in the state PARSE_MESSAGE also takes care of chunked encoding. Chunk data or the content data as may be specified by HTTP headers is suitably unchunked and copied to a buffer. String searches for user specified strings are carried out on this buffer.

This completes the discussion on the HTTP Filter in PickPacket. In this chapter, the design and implementation of the HTTP Filter based on the structure of HTTP transactions was presented. Goals met by the HTTP filter were also defined. The rest of the thesis presents the testing strategies for the PickPacket Filter and concludes this work.

Chapter 6

Performance Evaluation

The performance evaluation of the PickPacket Filter based on experiments conducted with the FTP and HTTP filters is described in this chapter. Performance of both the FTP filter and the HTTP filter of PickPacket is checked by specifying several filtering parameters for these applications in the configuration file and by generating heavy network traffic for that application while the filter is run.

The experiments for determining the performance of the application level filters are similar to experiments described in [23]. The application level filter cannot capture more packets than a sniffer which only counts the number of packets within some experimental error. If the number of packets captured by the application level filter after applying user specified criteria is the same as the number of packets captured by the simple sniffer then packets have not been dropped because of computations done by these filters. Two instances of the PickPacket Filter were run on two different machines for testing an application level filter. The first filter just counted the number of packets and the second filter also filtered these packets based on the specifications in the configuration files given in Appendix B.

Two identical machines with Intel Pentium 1.6 GHz CPU, 256 MB RAM and running Linux kernel version 2.4.18-3 were used on a 100 Mbps Ethernet segment. 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 the former packet filter is called the *base filter* and latter is referred to as the *real filter*. Filtering was stopped by setting a timer which expired in 4 minutes. However, the filtering was started manually.

6.1 Performance of the FTP Filter

Six scripts which downloaded a 50 MB file from 6 different servers in several FTP sessions were started on a client. In one session only a single data transfer command was issued and then the session was closed. Thus 6 FTP sessions were running in parallel. The client and the machines running the *real* and the *base* filter were on the same network segment. This resulted in a data transfer rate of 68 Mbps. The passive mode of file transfer was kept off so that no change of the kernel-level filter was required. This made the comparison with the base filter possible.

Filter	Total Packets received by the interface	Packets read by the packet filter	Packets saved
real filter	2004268	1999950	469544
base filter	1993757	1992260	1992260

Table 6.1: FTP Filtering by reading simulated traffic

Table 6.1 lists the filtering statistics of the two filters. The FTP filter handled this data rate for 50 username specifications, 50 file specifications and 50 text string search specifications in the configuration file. More number of parameters were not tried. The slight difference in the number of packets read by the two filters is due to the difference in time when the two filters started filtering the packets. Thus the time required by the FTP filter to filter data content does not force the kernel to drop packets for reasonable number of filtering parameters at high data transfer rates.

6.2 Performance of the HTTP Filter

A client on the same segment as the machines running the *real* and the *base* filter used the package "Downloader for X" [13] (D4X) to down load a 50 MB file from 6 servers. Several downloads of this file were initiated in parallel by D4X. At a time there were 8 connections on each of these servers. This resulted in a transfer rate of 63 Mbps.

Filter	Total Packets received by the interface	Packets read by the packet filter	Packets saved
real filter	1950553	1946923	358016
base filter	1942866	1940894	1940894

Table 6.2: HTTP Filtering by reading simulated traffic

Table 6.2 lists the filtering statistics of the two filters. The HTTP filter handled this data rate for 50 host specifications, 50 path specifications and 50 text string search specifications in the configuration file. The slight difference in the number of packets read by the two filters is due to the difference in time when the two filters started filtering the packets. Thus the time required by the HTTP filter to filter data content does not force the kernel to drop packets for reasonable number of filtering parameters at high data transfer rates.

6.3 Limitations of the FTP Filter

The FTP filter is expected to drop packets when "PASV" commands are closely spaced as discussed in Section 4.3. The experiments performed on the filter tried to generate this pathological case. This case can be generated by having small files transferred in the passive mode of file transfer using small buffer sizes attached to the socket. In the experiment a file of 980 bytes was transferred between two machines several times in the same control connection. File transfers were done through a simple script using several put commands. The configuration file for the

filter specified checking for one user name, one file name and one string. All these criteria matched every file transferred. The maximum number of put commands that could be handled by the FTP filter for a given buffer size before the filter started dropping packets was recorded. The filter was run on a machine different than the machines participating in the file transfers.

Buffer Size (bytes)	Maximum Puts (10 Mbps)	Maximum Puts (100 Mbps)
1024	1	1
1280	1	1
1536	2	2
1792	47	10
2048	300	11
2304	450	134
3072	-	265

Table 6.3: Maximum number of puts for a given buffer size in passive file transfer

Table 6.3 shows the maximum puts possible before the filter started dropping packets for machines connected through a 10 Mbps and a 100 Mbps hub respectively and transferring files through passive file transfers. The maximum transfer rate that could be achieved were 40 Kbps and 788 Kbps respectively because of the high overhead of establishing data connections for each put. This experiment shows that packets can be lost by the FTP filter under pathological circumstances. Having larger file sizes would only improve performance as the distance between consecutive puts would increase. Every file transferred in the normal mode of file transfer was captured by the FTP filter with a buffer size of 2048 bytes. The PickPacket Filter has been tuned to use a 1 MB buffer.

Chapter 7

Conclusions

This thesis discussed the filtering of packets flowing across the network by PickPacket with a special focus on filtering packets based on the FTP and HTTP application level protocols. PickPacket allows the filtering of packets on the basis of criteria specified by the user both at the network and the application level of the protocol stack.

PickPacket is a useful tool for gathering and rendering information flowing across the network. The design of PickPacket is modular, flexible, extensible, robust and efficient. Judicious use of the system can also help protect the privacy of individuals and can dump only necessary data to the disk. Tools for Post-processing and subsequent rendering make the tool easy to use. The universality of the capture file formats offer the user a choice of using "rendering and post-processing tools" other than those provided by PickPacket.

PickPacket is architecturally divided into four components the PickPacket Configuration File Generator, the PickPacket Filter, the PickPacket Post Processor, and the PickPacket Data Viewer. Each of these components were briefly discussed and the basic design of the PickPacket Filter was discussed. PickPacket uses in-kernel filtering to capture packets at the network level. The packets filtered by the in-kernel filter are passed to the application level filter for further processing.

Modules for filtering FTP and HTTP packets have been further discussed in this thesis. Users of PickPacket can specify names of users, file names and text search

strings for filtering packets belonging to FTP sessions. Host names, path names and text search strings can be specified for filtering packets belonging to HTTP sessions. Filtering packets belonging to FTP sessions impacts the design of the PickPacket Filter. The use of in-kernel filtering for capturing packets is retained as a design decision.

Several experiments were conducted to check the performance of the FTP and HTTP filters of PickPacket. These experiments show that these filters can successfully capture and filter packets on the basis of several criteria at high network loads. The limitations of the FTP filter under pathological cases of passive file transfers were also explored.

7.1 Further Work

PickPacket currently supports SMTP, FTP, and HTTP application level protocols. There is always scope for extending PickPacket to support other application level protocols. However, the protocols currently implemented do not support mime types for searching text strings. Useful work can be done to incorporate several of these mime types in various application level filters. Encrypting dumped packets and digital signatures can be added for making PickPacket more useful to law enforcement agencies. This can make packets captured admissible as evidence. The major limitation of PickPacket is that it currently does not support dynamic address allocation based networks. This would be required of PickPacket to make it useful in scenarios involving Internet Service Providers. PickPacket should be extended to include protocols like RADIUS and DHCP to achieve this.

Bibliography

[1] "Antisniff Site". http://www.L0pht.com/antisniff/.

[2] D. Atkins, W. Stallings, and P. ZimmerMan. "PGP Message Exchange Format". Technical report, 1996. http://www.ietf.org/rfc/rfc1991.txt.

[3] T. Berners-Lee, R. Fielding, U. C. Irvine, and L. Masinter. "Uniform Resource Identifiers (URI): Generic Syntax". Technical report, 1998. http://www.ietf.org/rfc/rfc2396.txt.

[4] bind. "Sentinel". http://www.packetfactory.net/Projects/sentinel.

[5] "Blackice". http://www.networkice.com and http://www.iss.net/netice.

[6] B. Callaghan and R. Gilligan. "Snoop Version 2 Packet Capture File Format". Technical report, 1996. http://www.faqs.org/rfcs/rfc1761.html.

[7] Brecht Claerhout. "Sniffit". http://reptile.rug.ac.be/ coder/sniffit/sniffit.html.

[8] David H. Crocker. "Standard for the Format of ARPA Internet Text Message". Technical report, 1982. http://www.ietf.org/rfc/rfc822.txt.

[9] Andrew Daviel. "ifstatus". ftp://andrew.triumf.ca/pub/security/ifstatus2.0.tar.gz.

[10] Loris Degioanni, Paolo Politano, Fluvio Risso, and Piero Viano. "Analyzer". http://netgroup-serv.polito.it/analyzer/.

[11] Loris Degioanni, Fulvio Risso, and Piero Viano. "Windump". http://netgroup-serv.polito.it/windump.

[12] "Klos". http://www.klos.com.

[13] "Downloader for X". http://www.krasu.ru/soft/chuchelo/ also available as RPM in Linux Distributions.

[14] S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, and L. Repka. "S/MIME Version 2 Message Specification ". Technical report, 1998. http://www.ietf.org/rfc/rfc2311.txt.

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

[16] "Etherpeek nx". http://www.wildpackets.com.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. "Hypertext Transfer Protocol". Technical report, 1997. http://www.ietf.org/rfc/rfc2068.txt.

[18] Alan O. Freier, Philip Karlton, and Paul C. Kocher. "The SSL Protocol". Technical report, 1996. http://wp.netscape.com/eng/ssl3/draft302.txt. SSL Available at http://www.openssl.org and many other sites.

[19] "Gobbler and Beholder". http://nmrc.org/files/msdos/gobbler.zip.

[20] Robert Graham. "carnivore faq". http://www.robertgraham.com/pubs/carnivore-faq.html.

[21] "How Carnivore Works". http://www.howstuffworks.com/carnivore.htm.

[22] 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 and www.tcpdump.org.

[23] Neeraj Kapoor. "Design and Implementation of a Network Monitoring Tool". Technical report, Department of Computer Science and Engineering, IIT Kanpur, Apr 2001. http://www.cse.iitk.ac.in/research/mtech2000/Y011111.html.

[24] "Kerberos Site". http://web.mit.edu/kerberos/www/.

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

[26] 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.

[27] Jeffrey C. Mogul, Richard 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.

[28] K. Muthukrishnan and A. Malis. "A Core MPLS IP VPN Architecture ". Technical report, 2000. http://www.ietf.org/rfc/rfc2917.txt.

[29] "ngenius". http://www.netscout.com.

[30] "CMA5000". http://www.nettest.com.

[31] "Network Associates Incorporated". http://www.sniffer.com.

[32] B. Clifford Neuman and Theodore Ts'o. "An Authentication Service for Computer Networks". In *IEEE Communications '94*, pages 33–38, 1994.

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

[34] Mike Perry. "Antiantisniffer". http://www.apocalypseonline.com/security/tools/tools.asp?exp_category=Sniffers.

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

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

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

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

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

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

[41] Stephen P. Smith, Henry Perrit Jr., 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.

[42] "Supersniffer v1.3". http://users.dhp.com/ ajax/projects/.

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

[44] Brian L. Tierney. "TCP Tuning Guide for Distributed Application on Wide Areas Networks". Technical report, Lawrence Berkeley National Laboratory, Feb 2001.

[45] "Trinux". http://www.trinux.org/.

[46] "LANdecoder32". http://www.triticom.com.

[47] Carnegie Mellon University. "Check Promiscuous Mode". ftp://coast.cs.purdue.edu/pub/tools/unix/sysutils/cpm/.

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

[49] Tatu Yloonen. "The SSH Secure Shell Remote Login Protocol". Technical report, 1996. http://www.free.lp.se/fish/rfc.txt. SSH available at http://www.openssh.org and many other sites.

Appendix A

A Sample Configuration File

```
#This is a sample configuration file
#Sections start and end with tags similar to HTML.
#Tags within sections can start and end subsections or can be tag-value pairs.
#All the tags that are recognized appear in this file.
#Empty lines are ignored.
#Lines beginning with a # are comments


# First Section specifies the sizes and names of the dump files
<Output_File_Manager_Settings>
        <Default_Output_File_manager_Settings>
#number of specified files
        Num_Of_Files=1
#the full file name relative/absolute will do
        File_Path=dump1.dump
#the file size in MB
        File_Size=12
        </Default_Output_File_manager_Settings>
</Output_File_Manager_Settings>
```

```
# The Second Section specifies the source and destination IP ranges
#       the source and destination ports, the protocol and the application
#       that should handle these IPs and ports
# The basic criteria here are for the Device and
# SrcIP1:SrcIP2:DestIP1:DestIP2:SrcP1:SrcP2:DestP1:DestP2:ProtoA:App
# Should be read as For the range of sorce IP from SrcIP1 to SrcIP2
#                   For associated ports from SrcP1 to SrcP2
#               and For the range of desitnation IP from DestIP1 to DestIP2
#                   For associated ports from DestP1 to DestP2
#               and FOR Protocol ProtoA
#               monitor connections according to Application App
# Protocols can be UDP or TCP
# Applications for TCP are
#     SMTP, FTP, HTTP, TELNET, TEXT, FULL_DUMP, PEN_DUMP
# Applications for UDP are
#     FULL_DUMP, PEN_DUMP
# No further specs are required for DUMP kind of applications.
<Basic_Criteria>
        DEVICE=eth0
        Num_Of_Criteria=8
        Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:25-25:TCP:SMTP
        Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:20-20:TCP:FTP
        Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:21-21:TCP:FTP
        Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:23-23:TCP:TELNET
        Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:80-80:TCP:HTTP
        Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:143-143:TCP:TEXT
        Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:1024-6535:TCP:FULL_DUMP
        Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:1024-65535:UDP:FULL_DUMP
</Basic_Criteria>
```

```
# The third section specifies the number  of connections to open simultaneously
# for some applications. Has tunable number of connections that should be monitored
# by some applications of interest SIMULTANEOUSLY
<NUM_CONNECTIONS>
        NUM_CONNECTIONS=3
        Num_SMTP_Connections=500
        Num_FTP_Connections=500
        Num_HTTP_Connections=500
</NUM_CONNECTIONS>

# The next sections describe in no particular order the application specific
#       input criteria.
#**************SMTP Specifications******
<SMTP_Configuration>
        <SMTP_Criteria>
                NUM_of_Criteria=1
                <Search_Email_ID>
                        Num_of_email_id=2
                        Case-Sensitive=yes
                        E-mail_ID=skjaincs@iitk.ac.in
                        E-mail_ID=brajesh@hotmail.com
                </Search_Email_ID>
                <Search_Text_Strings>
                        Num_of_Strings=0
                </Search_Text_Strings>
        </SMTP_Criteria>
        Num_of_Stored_Packets=750
        Mode_Of_Operation=full
</SMTP_Configuration>
#**********END SMTP Specifications******
```

```
##########FTP Specifications******
<FTP_Configuration>
        <FTP_Criteria>
                NUM_of_Criteria=1
                <Usernames>
                        Num_Of_Usernames=2
                        Case-Sensitive=no
                        Username=ankanand
                        Username=nmangal
                </Usernames>
                <Filenames>
                        Num_Of_Filenames=1
                        Case-Sensitive=no
                        Filename=test.txt
                </Filenames>
                <Search_Text_Strings>
                        Num_Of_Strings=1
                        Case-Sensitive=yes
                        String=book secret
                </Search_Text_Strings>
        </FTP_Criteria>
        Num_of_Stored_Packets=750
        Monitor_FTP_Data=yes
        Mode_of_Operation=full
</FTP_Configuration>
##########END FTP Specifications******
```

```
#*************HTTP Specifications******
<HTTP_Configuration>
        <HTTP_Criteria>
        NUM_of_Criteria=1
                <Host>
                        Num_Of_Hosts=1
                        Case-Sensitive=no
                        HOST=http://www.rediff.com
                </Host>
                <Path>
                        Num_Of_Paths=1
                        Case-Sensitive=yes
                        PATH=/cricket
                </Path>
                <Search_Text_Strings>
                        Num_of_Strings=1
                        Case-Sensitive=no
                        String=neutral venu
                </Search_Text_Strings>
        </HTTP_Criteria>
        <Port_List>
                Num_of_Ports=1
                HTTP_Server_Port=80
        </Port_List>
        Num_of_Stored_Packets=750
        Mode_Of_Operation=full
</HTTP_Configuration>
#*********END HTTP Specifications******
```

```
#*********TELNET Specifications******
<TELNET_Configuration>
        <Usernames>
                Num_of_Usernames=1
                Case-Sensitive=yes
                Username=ankanand
        </Usernames>
        Mode_Of_Operation=full
</TELNET_Configuration>
#*****END TELNET Specifications******
#*********TEXT SEARCH Specifications******
#These have to be added manually
<TEXT_Configuration>
        <Search_Text_Strings>
                Num_of_Strings=1
                Case-Sensitive=no
                String=timesofindia
        </Search_Text_Strings>
        Mode_Of_Operation=pen
</TEXT_Configuration>
#*****END TEXT SEARCH Specifications******
#**********End Application Specific Specifications****
```

Appendix B

Configuration Files und scripts used for Filter Testing

B.1 Files for testing FTP filter

B.1.1 Base Filter Configuration File

```
<Output_File_Manager_Settings>
    <Default_Output_File_manager_Settings>
    Num_Of_Files=1
    File_Path=/dev/null
    File_Size=4000
    </Default_Output_File_manager_Settings>
</Output_File_Manager_Settings>
<BASIC_CRITERIA>
    DEVICE=eth0
    Num_Of_Criteria=2
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:20-20:TCP:DUMP_FULL
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:21-21:TCP:DUMP_FULL
</BASIC_CRITERIA>
<NUM_CONNECTIONS>
    NUM_CONNECTIONS=1
```

```
    NUM_FTP_CONNECTIONS=1000
</NUM_CONNECTIONS>
```

B.1.2 Real Filter Configuration File

```
<Output_File_Manager_Settings>
    <Default_Output_File_manager_Settings>
    Num_Of_Files=1
    File_Path=/usr/dumpdata/demodump.dump
    File_Size=4000
    </Default_Output_File_manager_Settings>
</Output_File_Manager_Settings>
<BASIC_CRITERIA>
    DEVICE=eth0
    Num_Of_Criteria=2
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:20-20:TCP:FTP
    Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:21-21:TCP:FTP
</BASIC_CRITERIA>
<NUM_CONNECTIONS>
    NUM_CONNECTIONS=1
    NUM_FTP_CONNECTIONS=1000
</NUM_CONNECTIONS>
<FTP_Configuration>
    <FTP_Criteria>
    NUM_of_Criteria=1
        <Usernames>
            Num_Of_Usernames=50
            Case-Sensitive=no
            Usernames=somename
            Usernames=......          REPEATED 50 times
            Usernames=......
            Usernames=brajesh
```

```
        </Usernames>
        <Filenames>
            Num_Of_Filenames=50
            Case-Sensitive=no
            Filenames=abracadabra
            Filenames=..........    REPEATED 50 times
            Filenames=..........
            Filenames=xyz
        </Filenames>
        <Search_Text_Strings>
            Num_of_Strings=50
            Case-Sensitive=no
            String=arbit
            String=.....        REPEATED 50 times
            String=.....
            String=Test String
        </Search_Text_Strings>
    </FTP_Criteria>
    Num_of_Stored_Packets=100
    Mode_Of_Operation=full
</FTP_Configuration>
```

B.1.3 Sample FTP script on client

```
ftp -n serverA << !
user brajesh password mumble
passive
get xyz
bye
!
#These 6 lines were repeated 170 times
#In each script the user name was different
```

B.2 Files for testing HTTP filter

B.2.1 Base Filter Configuration File

```
<Output_File_Manager_Settings>
    <Default_Output_File_manager_Settings>
    Num_Of_Files=1
    File_Path=/dev/null
    File_Size=4000
    </Default_Output_File_manager_Settings>
</Output_File_Manager_Settings>
<BASIC_CRITERIA>
    DEVICE=eth0
    Num_Of_Criteria=2
    Criteria=172.31.19.1-172.31.19.7:0.0.0.0-0.0.0.0:1024-65535:80-80:TCP:DUMP_FULL
</BASIC_CRITERIA>
<NUM_CONNECTIONS>
    NUM_CONNECTIONS=1
    NUM_HTTP_CONNECTIONS=1000
</NUM_CONNECTIONS>
```

B.2.2 Real Filter Configuration File

```
<Output_File_Manager_Settings>
    <Default_Output_File_manager_Settings>
    Num_Of_Files=1
    File_Path=/usr/dumpdata/demodump.dump
    File_Size=4000
    </Default_Output_File_manager_Settings>
</Output_File_Manager_Settings>
<BASIC_CRITERIA>
    DEVICE=eth0
    Num_Of_Criteria=1
```

```
    Criteria=172.31.19.1-172.31.19.7:0.0.0.0-0.0.0.0:1024-65535:80-80:TCP:HTTP
</BASIC_CRITERIA>
<NUM_CONNECTIONS>
    NUM_CONNECTIONS=1
    NUM_HTTP_CONNECTIONS=1000
</NUM_CONNECTIONS>
<HTTP_Configuration>
    <HTTP_Criteria>
    NUM_of_Criteria=1
        <Host>
            Num_Of_Hosts=50
            Case-Sensitive=no
            HOST=google
            HOST=......          REPEATED 50 times
            HOST=......
            HOST=172.31
        </Host>
        <Path>
            Num_Of_Paths=50
            Case-Sensitive=no
            PATH=abracadabra
            PATH=...........     REPEATED 50 times
            PATH=...........
            PATH=test
        </Path>
        <Search_Text_Strings>
            Num_of_Strings=50
            Case-Sensitive=no
            String=arbit
            String=.....         REPEATED 50 times
            String=.....
```

```
            String=Test String
        </Search_Text_Strings>
    </HTTP_Criteria>
    <Port_List>
        Num_of_Ports=1
        HTTP_Server_Port=80
    </Port_List>
    Num_of_Stored_Packets=100
    Mode_Of_Operation=full
</HTTP_Configuration>
```