APPLICATION LAYER AND IN-KERNEL PACKET FILTERING IN THE NETWORK MONITORINING TOOL - PICKPACKET

Brajesh Pande Dheeraj Sanghi Deepak Gupta
Department of Computer Science and Engineering, IIT Kanpur, India
brajesh@iitk.ac.in dheeraj@iitk.ac.in deepak@iitk.ac.in

Abstract: Corporate bodies as well as law enforcement agencies need to monitor, detect and analyze network traffic. However this may be against the goal of maintaining the privacy of individuals whose network communications are being monitored. Packet filters that address these conflicting issues are needed. Also, tools that do not filter packets based on application layer content require storing huge amounts of data and extracting necessary information by post processing. The storing of such information may be illegal and costly. However, extracting packets based on application layer content, on the fly, requires faster and more intelligent filters. In-kernel filters can speed up packet filtering but are restricted to filtering packets based on the network layer. Thus a combination of in-kernel filtering and user level filtering is required to handle filtering based on parameters from the application level.

A monitoring tool that combines user and kernel level filtering for capturing packets based on application layer content needs to be carefully designed especially when the application layer can use some network level based parameters that change dynamically. PickPacket is a network-monitoring tool that handles the conflicting issues of network monitoring and privacy through its judicious use. In this paper we first briefly introduce sniffers in general and press on the need for monitoring application layer level content. We then introduce some requirements for designing filters that can work in conjunction with legal frameworks to help monitor networks. We trace how these requirements further impact the design of a network-monitoring tool - PickPacket. In particular we maintain that speed is necessary for realizing the filtering component of such tool and how tradeoffs are introduced in the design for meeting this requirement. "The PickPacket Filter" combines in-kernel filtering with user level filtering for capturing packets based on application layer content. This paper focusses on the design of the "PickPacket Packet Filter" to explore relationships between in-kernel filtering and user level filtering.

1. Introduction:

The rapid increase in the use of the computers coupled with the exponential growth of the Internet has also had impact on its use for communication and exchange of information pertaining to unlawful activity. Effective tools that can monitor the network and can also keep up with the growing bandwidth speeds are required. Such monitoring tools help network administrators in evaluating and diagnosing performance problems with servers, the network wire, hubs and applications. 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 and protect intellectual property. Such monitoring tools, therefore, can have an important role in helping authorities gather information against terrorism, child pornography / exploitation, espionage, information warfare and fraud. 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 network traffic.

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 (ECPA 1986, RIPA 2000, TACT 1997). Nonetheless, the very same laws that assure privacy of communications also have mechanisms of warrants (RIPA 2000, TACT 1979) or exceptions (ECPA 1986) to allow monitoring communications for the sake of national interests and other lawful activities. In this paper we introduce such tools and try to lay down some requirements for them. We also

try to evolve these requirements on the basis of example network protocols and see how the design of such a tool is affected and realized in – PickPacket (Neeraj 2002, Pande 2002, Jain 2003, Prashant 2003, Pande 2005).

2. Sniffers:

Network monitoring tools are also called sniffers. The word sniffer is also a trademark of Network Associates Inc. Sniffing programs have traditionally been 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 for national security 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. However sniffers can be rendered useless through the use of encryption mechanisms. Switching from a "hubbed" to a switched network can thwart casual sniffers but one has to carefully safeguard against spoofing at various levels. Moreover the entire Internet cannot guarantee a switched network. Sniffers are passive listeners and hence it should not be possible to detect them though improperly configured sniffers give responses and can be detected.

Sniffers put the network card of a computer into the "promiscuous mode". This enables the computer to listen to the entire traffic on that subsection 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.

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.

2.1. Lawful Sniffing:

The very notion of sniffing network traffic is opposed to the idea of privacy. The question that begs an answer is how would one legally monitor undesirable traffic within the purview of these laws. First a judge has to be convinced that the act of sniffing is permissible based on some evidence that is supplied. Then having established a case for it the permission to partially or fully sniff a connection or a given set of connections may be given.

Suppose there are some grounds for suspicion that an individual is using the network for unlawful communications. In almost all likelihood a judge may allow just enough sniffing to establish grounds for further investigation. We may be allowed to look only at IP addresses of packets that contain a particular "trigger" word. However, it may be mandated that this may only be done as and when the "trigger" word occurs. A sniffer in this case should at most dump IP addresses and port numbers to the disk. The judge could further allow more information to be dumped say of packets belonging to a specific protocol and that too only in the case when the "trigger" word occurs in the packets. Several such scenarios can be visualized. These can be generalized to make the case for a sniffing tool that dumps data to the disk at various levels of granularity and on different "trigger events". The monitoring of such events should be done in memory rather than in a post capture analysis.

From the discussions above it can be seen that such tools necessarily need to examine the content of packets at the application layer

content level. Such a tool in our opinion should search for the "triggers" in memory buffers. Creating such a tool is a challenging task and this has been attempted with PickPacket.

2.2. Some Sniffing Products:

Several commercially and freely available sniffers exist currently. Sniffers come in different flavours and capabilities for different Operating Systems. In this section we highlight just a few. A full survey would be beyond the scope of the present work. The sniffing faq (Graham 1998) is a good beginning for a product search and general information on sniffers. Ethereal (Combs) 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 (Degioanni 1998) is a version of tcpdump for Windows that uses a libpcap-compatible library called WinCap. Network Associates Incorporated (NAI) has a range of sniffers including voice over IP sniffers. BlackICE (ISS) 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 (EtherPeek) have several sniffing products for Windows and Macintosh environments. Triticom (Triticom) have a suit of products that include application-level decoders and other monitoring software. Analyzer (Degioanni 1998) 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 (Jacobson 1989). 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 trace file has been specified in RFC 1761 (Callaghan 1996). It can be converted to tcpdump/libpcap (Jacobson 1989) format via many utilities, including "tcptrace". The Trinux (Trinux) Linux security toolkit bundles several sniffing utilities. Klos Technologies (Klos) provide several sniffers on the DOS / Windows platform for sniffing packets on LAN (Local Area Networks) and PPP (Point-to-Point Protocol) connections.

Carnivore (Smith 2000) 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.

2.3. PickPacket:

PickPacket (Neeraj 2002, Pande 2002, Jain 2003, Prashant 2003, Pande 2005) 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 TELNET, SMTP, HTTP, and FTP etc. It also supports real-time searching for text string in application and packet content. The purpose of PickPacket, like the filters 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.

Like Carnivore, 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 further 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 especially because packets are kept in memory till all criteria specified by the user are fulfilled. The flexibility of selection of filtering parameters across the TCP/IP stack also helps in this. 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. A GUI for generating the input rules to the filter is also provided.

2.4. Filtering Packets - In-Kernel Filters:

The amount of information that flows on a network is generally quite high with packets corresponding to different protocols and even a simple analysis of this data takes time and space. This can be reduced considerably by allowing the user to specify some rules for capturing packets selectively. Packet filters provide this facility of specifying such simple rules. These rules comprise of values corresponding to various fields present in the protocol headers of a packet. If the protocol header of a packet contains these values then it is saved else the packet filter drops it.

The CMU/Stanford Packet Filter (CSPF) (Mogul 1987) was the first UNIX based kernel-resident, protocol independent packet demultiplexer. It provides user processes great flexibility in selecting packets that they receive. It eventually evolved into the Network Interface Tap (NIT) (Sun 1987) under SunOS 3, and later into Berkeley Packet Filter (BPF) (MacCanne 1993). Sun implemented NIT to capture packets and *etherfind* to print packet headers. These packet filters although being implemented inside the kernel, provide an architecture over which the user-level network monitoring tools can be built.

BPF essentially comprises of two components: a filter code and an interpreter that executes the filter code on the packet read from the network. The filter code uses a hypothetical machine consisting of an accumulator, an index register, a scratch memory store, and a program counter. This filter code is in an assembly like language and includes operations like load, store, branch, return, and some register transfer functions, etc.

BPF offers substantial performance improvements over other packet filtering mechanisms due to the following two reasons:

There are two approaches for filtering packets: a boolean expression tree (used by CSPF) and a directed acyclic control flow graph or CFG first used by NNstat (Braden 1988) and then used by BPF). These two models are computationally equivalent. However, in implementation the tree model maps naturally into code for a stack machine while the CFG model naturally maps into code for a register machine. Since most machines are register based, the CFG approach leads to a more efficient implementation.

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

The BSD socket interface is a de-facto standard for writing network-based applications. Thus the Linux operating system came up with the Linux Socket Filter (LSF) (Schulist). LSF is an

in-kernel packet filter derived from BPF. It provides the user with a packet filtering facility on BSD sockets. Among other packet filters, tcpdump is probably the most popular packet-capturing tool in the UNIX community. It is based on BPF and has the packet capturing and filtering facilities implemented in a separate library, pcap. There are wide ranges of network monitoring tools that integrate the pcap library.

3. Filtering Requirements:

In this section we discuss some requirements that were mandated for PickPacket so that it become a useful network-monitoring tool. Some of these requirements also have an impact upon the design of the filter. We believe that most filters should meet these requirements. The packet filter should not only provide a rich set of filtering criteria to the user but should also provide mechanisms for protecting the privacy of other users of the network. These expectations from the packet filter translate to following goals that the filter should meet.

Flexibility: The packet filter should allow capturing of packets based on the fields of the various protocol headers present in a packet. These fields may include IP addresses, protocol, port numbers, etc. The filter should also support filtering based on the application layer data content present in a packet. These criteria should include important fields in the data transferred on connections belonging to these protocols, like username for Telnet sessions, e-mail addresses for SMTP connections etc.

Granularity: The packet filter should allow specifying the amount of information that will be collected when a connection meets the filtering criteria. For example, the user may only require information whether any e-mail was sent or received on an e-mail addresses or may also like to know the content of the transferred e-mail. The former mode of collecting was referred as "Pen" mode and the latter as "Full" mode. PickPacket provides this facility by allowing specification of either of these modes.

Fast Processing / Robustness: The rate at which packets can be received from the network running the packet filter is determined by the bandwidth of the network. Buffer space that is provided by the operating system for storing packets is limited. Thus, the filter should take decisions about a packet as soon as possible to prevent packet losses. As a result of providing a rich set of filtering criteria, the packet the filter may consume a considerable amount of time in processing a packet. Meanwhile, the kernel will read packets from the network and save them in its buffer. If sufficient space is not available in the buffer, the kernel may drop packets and these dropped packets will not be available to the packet filter. The packet filter should handle such situations in a manner that the correctness of the data being collected is not sacrificed. This by far is one of the more important expectations of the PickPacket filter and is a source of trade offs in the design of PickPacket that we discuss later. Also it almost makes mandatory the use of in-kernel filters – a choice of design of "PickPacket Filter".

Maintaining privacy of the users: Only those packets that meet the user specified criteria should be saved on the disk. Tools that allow packet capturing across the TCP/IP stack should strictly follow this requirement. In many tools the search for specified criteria is not done in memory before dumping packets to the disk.

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

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

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

4. Design of the PickPacket Filter:

A typical filter can have several levels at which it filters packets. First it can filter packets based on network parameters (IP addresses, port numbers). Second level filtering can be on the basis of application layer protocol specific criteria and the third level of filtering can be based on the content of the application payload.

The first level of filtering can be made very efficient by using in-kernel filters. Since the content of application can be best deciphered by the application itself, the second and third levels of filtering are combined. Figure 1 captures this notion of levels of filtering.

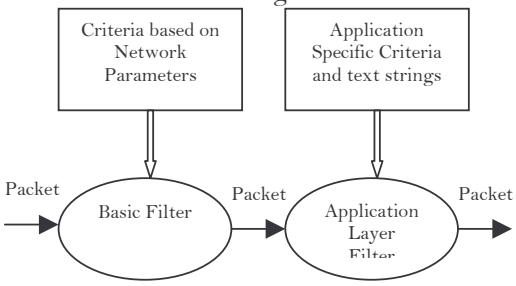

Figure 1. Filtering Levels

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.

A further refinement is captured in Figure 2. 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. The demultiplexer uses its specific set of criteria for routing packets.

Finally, application specific filtering reduces to text search in the application layer data content of the packets. In case of communications over

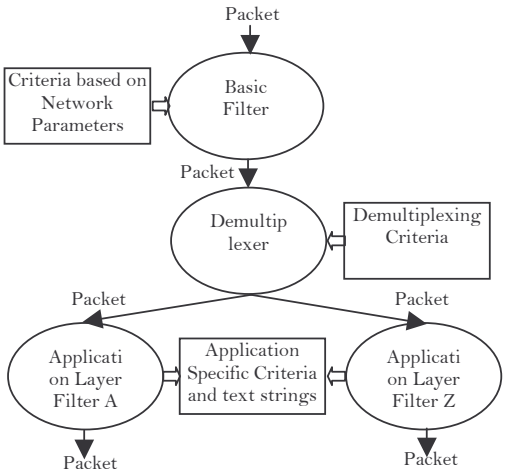

Figure 2. Demultiplexing Packets for Filtering

connection-oriented protocol, this text search should handle situations where the desired text is split across two or more packets before being transmitted on the network. As there may be losses or reordering of packets in the network, these filters should also check for packets that are received out of sequence while performing the search for split text. Thus a component that does these checks is introduced as another refinement to the filter above. This component is called the Connection Manager. Figure 3

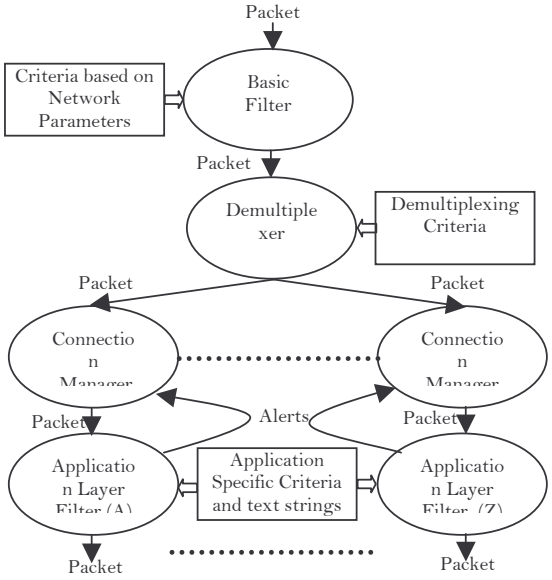

Fig 3. Some Components of A Packet Filter

captures this next level of refinement. This component is common to all application level filtering that allows searching for text strings in the application payload. However different connection managers can be defined when necessary for an application. Obviously, different connection managers can be written for handling UDP and TCP packets.

There are several considerations that go into designing the connection manager. 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 the demultiplexer also examines this four tuple. The connection manager also maintains states dependent on this four tuple. Therefore it is best to pass the data that the application wishes to associate with a connection to the connection manager and subsequently to the demultiplexer. Alerts incorporate this mechanism also.

The discussions above lay the foundation for the basic design of the PickPacket Filter. Figure 4 shows the basic design of the PickPacket Filter. All the criteria 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 inkernel BPF [11] 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 during "PASSIVE" mode of file transfers or by RADIUS application level filter after the IP from which a user is operating has been discovered. 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.

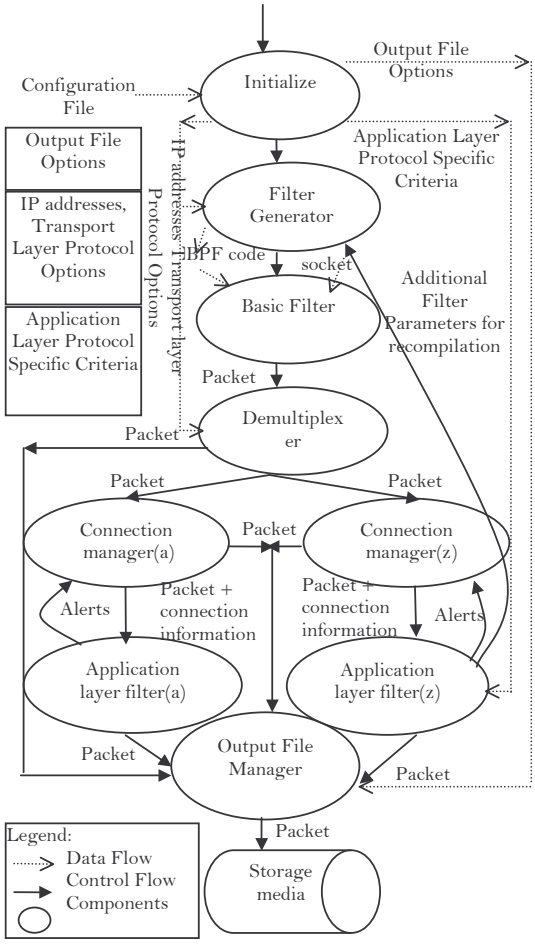

Figure 4. PickPacket Filter: The Basic Design

4.1. Output File Formats:

Conceptually, the output file manager can store files in any format. However, PickPacket stores output files in the pcap (Jacobson 2001) 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. Provisions have been made for digitally signing the output file on the fly.

4.2. Text String Search:

The PickPacket Filter contains a text string search library. Application layer filters in PickPacket extensively use this library. This library uses the Boyer-Moore (Boyer 1977) 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.

5. Evaluation of the Design:

In this section we evaluate the design of PickPacket outlined above to trace its trade-offs especially when in-kernel filtering and application level filtering is combined and we have an application layer protocol that forces us to change the Basic Filter at runtime.

Let us consider the FTP protocol and a case where the "PASSIVE" file transfers are being used. In this transfer the server transmits a port number to the client that is used by the client for subsequent file transfer operations. So an additional port – the one sent by the server has to be included in further filtering by the "Basic Filter" and the "Demultiplexer".

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 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 transfers. 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. Similar considerations with the "PROXY" method of file transfer lead us to conclude that it is best to monitor the tuple ⌈*; *; IP2; PZ⌉ and demultiplex packets belonging to this tuple to the FTP filter. In this case a third party IP address is additionally introduced that we can capture wit the first * in the four tuple. 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 depends on the processing power of the hardware and the size of the BPF code but can be assumed to be about 5 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 (Brian 2001). 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 down the overall performance of the filter and could again lead to dropping of packets. We believe that in-kernel is necessary for meeting the speed and robustness requirements of the filter. This may lead to dropping of packets only in pathological cases.

Similar considerations can apply to the RADIUS protocol filtering where first packets have to be routed to the RADIUS filter through a UDP connection manager and IP addresses for applications have to be discovered. These addresses have to be then recompiled into the filter.

5.1. Testing PickPacket:

Currently TELNET, FTP, SMTP, HTTP and RADIUS application level filters that have the ability of searching for text strings as well as an application level filter that only does text string searches on any packets passed to it have been incorporated in PickPacket. Each application level filter is capable of searching for parameters relevant to the application also. Each module and component of PickPacket was thoroughly tested to remove bugs through a black box testing strategy. Test cases were generated for all possible input sequences and the output was verified for its correctness. Performance of application level filters was 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 was running. The application level filter cannot see more packets than a sniffer that only counts the number of packets. If the number of packets examined by the application level filter after applying user specified criteria is the same as the number of packets dumped 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 specified application level criteria. These tests were found to be satisfactory for various filters. The RADIUS filter was also tested by using "radclient" - a utility provided by the Cistron Radius Server (CISTRON) that can generate different scenarios of the RADIUS protocol in action. Another black box model of testing for the application was done by simulating an ISP like environment and testing the filter with heavy load. Under the test conditions the various applications could handle all loads generated on a 100 MBPS line.

6. Conclusions:

Corporate bodies as well as law enforcing agencies need packet filters. Filters that meet conflicting needs of monitoring as well as maintaining privacy of individuals have to be carefully designed. Special requirements have to be met by such filters. The PickPacket Filter is a filter that can meet these requirements.

PickPacket is able to meet the flexibility and extensibility criteria enunciated earlier as we can easily specify any criteria across the network model layers and also add and develop new filters for different applications and types of connections. The speed and robustness criteria are met by the appropriate use of in-kernel filters and granularity is provided for by the "PEN" and "FULL" mode of operations. The ease of operations is maintained by providing GUI front ends for specifying the input file and for rendering the captured data. Digital signatures help increasing the veracity of captured data.

The privacy of network users however is not guaranteed to the extent that the filter can be set up maliciously. Otherwise, all due care has been taken that the filter dump only packets that it has been configured for and nothing else. Privacy of users is however guaranteed better than other such tools. Tools that we surveyed either dump data to the disk and then search it for "trigger events" or do not examine application layer content. Application layer content is monitored entirely in memory buffers in PickPacket. As a consequence of this PickPacket may not gather all the conversation depending on the history of conversation recorded. However the at the very least, part of the conversations that contains the "trigger event" is dumped and privacy of the conversation is better protected.

There is a slight overhead in using in-kernel filters especially when they have to be recompiled. This may cause dropping of packets in pathological cases where frequent recompilation of these filters is needed. This overhead can be justified on the basis of the speedup that such filters can provide.

Currently we do not know of a filter that can address all the concerns raised in this paper to the extent that has been done by PickPacket and we consider it a useful tool for network monitoring.

7. References:

Boyer R., Moore J. "A Fast String Searching Algorithm". In Comm. ACM 20, pages 762--772, 1977.

Braden R. T. "A Pseudo-machine for Packet Monitoring and Statistics". In Proc. of SIGCOMM '88, ACM, 1988.

Brian L. T., "TCP Tuning Guide for Distributed Application on Wide Area Networks", Technical Report, Lawrence Berkley National Laboratory, Feb 2001.

Callaghan B. and Gilligan R. "Snoop Version 2 Packet Capture File Format". Technical report, 1996.

CISTRON,"Cistron-Radius-Server"
ftp:///ftp.radius.cistron.nl /pub/radius".

Combs Gerald et. al., "Ethereal", http://www.ethereal.com.

Degioanni L., Risso F., Viano P.. and Bruno J. "WinDump", 1998, http://www.winpcap.org/

ECPA, "Electronic Communications Privacy Act". United States Code, Title 18, Part I, Chapter 119, Sec. 2150, 1986.

EtherPeek, http://www.wildpackets.com

Graham R., "Sniffing (network, wiretap, sniffer) FAQ, 1998.
http://www.robertgraham.com/pubs/sniffing-faq.html
http://linuxsecurity.net/resource_files/intrusion_detection/sniffing-faq.html

ISS, Internet Security Systems, "BlackIce", http://www.networkice.com

Jain S. K., "Implementation of RADIUS Support in PickPacket". Masters Thesis Dept. of Computer Science and Engg., IIT Kanpur, May 2003.

Jacobson V., Leres C., and McCanne S. "tcpdump: A Network Monitoring and Capturing Tool", 1989 . Unix man page.

Kapoor N., "Design and Implementation of a Network Monitoring Tool". Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur, Apr 2002.

Klos, Klos Technologies,http://www.klos.com/

McCanne S., Jacobson V. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In Proc. of USENIX Winter Conf., pages 259--269, Jan 1993.

Mogul J C, Rashid R F, Accetta M J. "The Packet Filter: An Efficient Mechanism for User Level Network Code". In Proc. of the 11th ACM Symposium on Operating Systems Principles. 1987

NAI., "Network Associates Incorporated", http://www. sniffer.com.

Pande B., "The Network Monitoring Tool - Pickpacket: FilteringFTP and HTTP Packets". Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur, Sept 2002.

Pande B., Sanghi D., Gupta D., Jain S. K., "The Network Monitoring Tool – PickPacket". In Proc. Int. Conf. On Information Technology and Applications, Sydney, Australia, July 2005.

Prashant A. "Pickpacket: Design and Implementation of the HTTP Postprocessor and MIME Parse Decoder". Technical report, Dept. of Computer Science and Engg., IIT Kanpur, Apr 2003.

RIPA, Regulation of Investigatory Powers Act 2000, 2000 Chapter 23. "http://www.opsi.gov.uk/acts/acts2000/20000023.htm"

Schulist A. "Linux Socket Filter". "http://www.ibiblio.org/peanut/Kernel-2.6.6/networking/filter.txt".

Smith S. P., Perrit H. Jr. et al. "Independent Technical Review of the Carnivore System". Technical report, IIT Re- search Institute, Nov 2000.

Sun OS. "Sun OS 4.1 Manual", 1987.

TACT, "Telecommunications (Interception) Act 1979". Commonwealth of Australia Consolidated Acts, 1979.

TACT,"Telecommunications-Act-1997". Commonwealth of Australia Consolidated Acts, 1997.

Triticom, http://www.triticom.com/

Trinux, http://www.trinux.com