A Rule Based Interface to the Kernel for Selective Packet Relaying

K N Gopinath, Sumit Ganguly

March 23, 2000

1 Introduction

Network Firewalls have become an integral part of the security of the computer installation at any organization, whether commercial, academic or others. Broadly speaking, firewalls can be classified as follows [2]. *Packet filters* are firewalls that typically run at the network layer and use information in the IP and TCP headers to control traffic. Examples within public domain include Linux Packet filter (ipfw), BSD Packet filter etc. *Application level firewalls* typically are *proxy* servers which *relay* data at the application level. Examples are fwtk from Trusted Information Systems (TIS), Squid etc. *Circuit level proxies*, as typified by SOCKS [3], authenticate a given user, host, application triple, and, otherwise relay data at the session level.

Application level firewalls are capable of enforcing elaborate and fine-grain application specific security policies, such as content-based filtering of data, and can provide good audit facilities. A common disadvantage of both application level proxy and circuit level proxy is that they require application data to be copied back and forth between the proxy daemon and the kernel at the firewall machine. This in turn necessitates a context-switch between the kernel and the application daemon and vice-versa, virtually for every packet. Although such a design ensures that general security policies can be enforced, many site/application pairs may not have much use for inspecting every packet flowing through a connection. For such site/application pairs, the overhead of context switches for almost every packet due to the implementation of an application or circuit-level relay, increases the load on the firewall machine.

The concern for reducing potentially large number of context-switches at the firewall machine motivates that *mechanisms* be provided by the kernel that allows packets to be *selectively* sent to the application daemon as dictated by security policies. Such a design could significantly reduce the load on a firewall machine by reducing the overhead of context switches between the kernel and the application daemon at the firewall. A class of firewalls with such characteristics is emerging and is variously called as *Stateful*, *Dynamic* or *Smart* Packet Filters [6, 5]. A common characteristic of these firewalls appears to be the absence of a *relay* at the application-level or the circuit-level.

As a simple example, consider the `port` command transmitted as part of the standard FTP application. A stateful packet filter (SPF) "remembers" the port number argument of the `port` command. A subsequent incoming data connection is checked against the "remembered" value and is allowed only if the port numbers match.

Several commercially available firewalls such as Firewall-1 [6] are based on the general idea of stateful packet filters. However, the design and implementation details that are publicly available for these products are too meagre to allow comparative or critical evaluation. One of the decisions that needs to be made in the implementation of SPFs is the amount of application specific knowledge that needs to be built into the kernel. One possibility is that the kernel can be made aware of a significant portion of the application protocol. An example in this category is the IP masquerading module of Linux kernel version 2.0.34. It embeds the functionality of searching for and remembering the arguments of a `port` command of the FTP application inside the kernel. Another possibility would be the design of *mechanisms* which can be used by user space daemons to implement site-specific security policies but which make the kernel lightweight and independent of applications.

In this paper, we present a design and prototype implementation of kernel *mechanisms* that effectively allow *selective* packet *relaying* for TCP based applications. This is achieved by keeping traffic state/information both in the kernel and application space. These kernel mechanisms are designed to be lightweight, that is, the kernel is not required to be aware of the details of application level protocols. We further argue, by means of examples, that the mechanisms provided by the kernel are sufficiently general to allow a continuum of application level security policies to be enforced, ranging from simple

packet filtering to those which can be provided by relays. The implementation has been done by modifying the IP(v4) stack of Linux Kernel (version 2.0.34) running on an Intel x86 processor. Using the kernel mechanisms, we have built a prototype security daemon for FTP and TELNET applications which runs as a user process.

The remainder of the paper is organized as follows. Section 2 presents a rule-based interface to the kernel that can be used by application level security daemons to specify which packets should be inspected. Section 3 presents a detailed example of how an FTP security daemon might use the rule-based mechanism to enforce fine-grain access control policies. Section 4 presents a short description of how an IP packet is processed by the Linux kernel after it is received from the network interface. Section 5 presents an overview of our implementation. Sections 6 and 7 discuss the two main aspects of our kernel modification, namely, maintenance of connection state and communication between application process and the kernel respectively. Section 8 discusses in brief our prototype implementation of an application level security daemon for FTP and TELNET applications. We present our conclusions and discuss future work in Section 9.

2 Rule-Based Selective Filtering of Packets

In this section, we present the rule based interface to the kernel. In our scheme, the applications can specify rules for the packet filtering to the kernel. These rules have the following form.

`<src-ip,src-msk,src-port,dst-ip,dst-msk,dst-port,state,action>`

The fields `src-ip` through `dst-port` are familiar arguments used by standard packet filters as well - difference lies in last two arguments, `state` and `action`. The argument `state` is a user level abstraction of the possible states of a TCP connection. It can take one of the following three values

- *Connection initiation* (abbreviated as `i`), a connection is in this state when the first SYN packet is seen by the kernel.
- *Connection termination* (abbreviated as `t`), refers to the state entered when the first RST or FIN is seen by the kernel and remains in this state until the termination of the TCP connection is complete.

- *normal* (abbreviated as `n`), refers to the normal state of a TCP connection where the data is flowing through the connection. A normal state is entered when the three way handshake is complete and exited when an RST or FIN is seen.

The usefulness of this abstraction will be made clear later with the help of some examples.

The `action` argument of a rule can take one of the following values :

- Send and Wait for permission (SAW_perm) The kernel first sends the packet to the daemon and waits for a yes/no reply from the daemon. If it receives an affirmative reply, then the kernel forwards the packet to the appropriate network interface. Otherwise, it drops the packet and sends an icmp message to the originator of the packet.

- Send and Wait for packet (SAW_packet). The kernel first sends the packet to the daemon as in the case for SAW_perm. The daemon, in this case, in addition to replying with a yes/no, can also modify the packet and send the modified packet back to the kernel. If the daemon returns a yes, the kernel sends the returned packet to the network interface. Otherwise, it sends an icmp message to the source as before. Any application specicific masquerading is handled using this mechanism.

- Send and Do not Wait (SDW) The kernel sends a copy of the packet to the application daemon as well as forward the packet to the network. However, the kernel neither waits nor expects any response from the daemon.

- Forward Packet (F) The kernel forwards the packet to the network interface without informing the daemon.

- Drop Packet (D) The kernel drops the packet and sends an icmp message to the originating machine.

All rules apply to a simplex part of a duplex TCP connection. Thus, if the daemon intends to have identical rules applied to both incoming and outgoing packets of the same TCP connection, then, the rules have to be duplicated with the appropriate entities interchanged (e.g. source address exchanged with destination address).

Rules may be designed to implement a variety of security policies spanning a range between the generality provided by application level proxies, on one hand, to the specificity and efficiency of simple packet filters. To see this, assume that rules are written so that the kernel intercepts each packet and sends it to the application daemon and waits for the daemon to return the packet to the kernel (SAW_packet). This effectively mimicks the design of an application level proxy. On the other hand, rules may be written so that every packet is either forwarded or rejected without sending to the application level. This effectively implements the design of a packet filter. Also, more interestingly, rules may be written (depending on the application we want to proxy/site requirements) to *selectively* send packets to the application daemon and still realize rich security policies

3 An Example FTP Firewall

In this section, we show how a fine grain access control policy for the FTP application can be implemented using the rules described above. Suppose that a network administrator wishes to have the following access control policy for internal users of the FTP application.

- An FTP control connection may be authorized to go through based on a combination of userid and machine (or subnet).
- Depending on userid and/or machine id(subnet), certain operations such as get, put etc may be selectively enabled or disabled.
- The data flowing through the data channel need not be monitored.

We specify four rules that implement the above security policy. In order to keep the rules short and simple, in this section we avoid mentioning the mask field. We illustrate the rules assuming that the user initiates the FTP application from an internal machine, with the hypothetical Internet address of 144.16.167.98. In general, this address could represent a subnet of the local area network which may be specified with the help of a mask.

Rule 1 says that connection initiation packets for the FTP control connection (port 21) originating from 144.16.167.98 be sent to the daemon and the kernel should wait for an affirmative reply from the daemon. The FTP

security daemon can use this rule to check if the user/machine has the authorization to use FTP or not. This can be done by using an authentication mechanism (say, using “ident” service) and by consulting a local database of access control lists. The notation – in the rules is used to denote “any” value.

Rule 1	src-ip	src-port	dest-ip	dest-port	state	action
	144.16.167.98	–	–	21	i	SAW_perm

Rule 2 says that all normal outbound packets on FTP control connection be sent to the daemon. The kernel, however, may forward the packet immediately.

Rule 2	src-ip	src-port	dest-ip	dest-port	state	action
	144.16.167.98	–	–	21	n	SDW

The rule above allows the FTP daemon to monitor all traffic passing through the FTP control channel. In particular, FTP commands such as PORT, RETR etc. are seen and if neccessary, remembered by the FTP firewall daemon. For the PORT command, the FTP daemon remembers the port number argument on the internal machine. Whenever a RETR command is seen, the FTP daemon can consult local database to see if the given user (whose identity and capabilities are determined during connection initiation) has RETR permissions. It should be noted that such application specific state is maintained by the FTP security daemon and *not the kernel.* Rule 2 does not disallow packet flow even if the user has issued a command he/she is not authorized. The security is ensured by the following rule.

Rule 3	src-ip	src-port	dest-ip	dest-port	state	action
	–	20	–	–	i	SAW_perm

The rule above says that any data channel initiation packet (port 20) be intercepted and sent to the FTP daemon. The kernel waits for an affirmative answer from the daemon. This rule together with Rule 2 provides a mechanism to disallow unauthorized traffic. Suppose that a user is not allowed to use RETR command of FTP. Then, the FTP daemon can remember that a RETR

is being attempted when it sees the RETR command on the control channel (Rule 2). The subsequent data channel connection packet is intercepted by Rule 3. The daemon can then send a negative response to the kernel which will cause the kernel to reject the connection. However, the FTP control connection will still be active and he can issue another FTP command. Thus, the command based authorization can be implemented.

Rule 4 is used to let the daemon know of the termination of FTP control channel so that the required cleanup action can be taken.

Rule 4	src-ip	src-port	dest-ip	dest-port	state	action
	144.16.167.98	–	–	21	t	SDW

The above set of rules assumed that the subnet has a valid IP address in the Internet. Suppose that this is not the case. The FTP security daemon may wish to masquerade its own IP address for the IP address argument of the `port` command of FTP. This is accomplished by replacing the Rule 2 by the Rule 2'.

Rule 2'	src-ip	src-port	dest-ip	dest-port	state	action
	144.16.167.98	–	–	21	n	SAW_packet

The SAW_packet value of the action field causes the kernel to send the packet to the daemon and then to wait for the packet to be delivered back to it by the daemon. The daemon masquerades the IP address of the `port` command (whenever applicable) and returns the packet to the kernel.

The above example illustrates a possible way of using the mechanisms provided by the rule based inspection to implement the desired security policies. An advantage of the scheme above has been to avoid sending packets on the data channel of FTP to the application level. Another advantage is that when a particular FTP data transfer request is rejected, the FTP control connection is not disturbed and can be used for subsequent requests.

The example also illustrates a design goal that we have implicitly tried to follow, namely, to keep the kernel simple and independent of application semantics.

4 Path of an IP Packet within Linux Kernel

Linux implements a protocol family as a series of connected layers of software [1], analogous to the protocol layers themselves. When a network device receives a packet from the network it converts the received data into a socket buffer data structure, called `sk_buff`, and interrupts the CPU. The interrupt processing routine of the network device adds the received `sk_buff` structure into a backlog queue (discarding the packet if the queue is full). It then sets a flag that indicates to the scheduler that the *network bottom half handler* should be called. The network bottom half handler is the portion of the kernel that handles the checking, forwarding of the packet to higher layers or to the device etc.

When the network bottom half handler is run by the scheduler, the handler passes the packet to the appropriate higher level protocol (say, IP for IP packets). In the case of IP packets, the `ip_rcv()` function is invoked. This function decides whether the packet is destined for itself or not. If so, it passes the packet to the appropriate higher level protocol (say, TCP or UDP). If the packet is destined for some other machine, and this host is configured as a router (`ip_forwarding host`), it calls the function `ip_forward()` to further process the packet. This function checks the standard packet filter rules before deciding whether to call the underlying driver routine to send the packet. The network bottom half handler code in Linux is written to be non-reentrant.

5 Overview of Modifications to the Linux Kernel

The main modifications to the kernel were made in the bottom half handler part of the code. In the unmodified kernel, the function `ip_rcv()` calls the function `ip_forward()` to process packets which need to be routed to a different machine. In our modification, `ip_rcv()` calls `spf_ip_forward()` which implements the rule based selective packet filtering. Depending on the policy prescribed by the rules and the return value from the daemon (if applicable), the packet is either dropped or forwarded to the appropriate interface by calling `ip_forward()`.

The implementation of the rules mechanism can be conceptually divided

into two components, namely, (1) maintaining the state of a TCP connection and (2) performing the actions for every packet as prescribed by the rules. The key element in the latter component is a robust mechanism for communication between the kernel and the daemon. This is discussed in the section 7. Maintenance of the kernel state is discussed in section 6.

Rules are read into the kernel using an "ioctl" like mechanism. We have used the existing socket interface to set up the rules for the kernel. These rules are read in at boot time or may be *dynamically* inserted by the application. The rules are maintained as a linked list and packets are checked against the list to see if there is any match.

The rules facility (currently) uses the `ip_fw` packet filter implemenation within the Linux kernel, primarily to allow IP masquerading. The `ip_fw` implementation supports generalization of the IP address fields using the masks and supports IP masquerading. We assume that IP fragments are reassembled by enabling the "CONFIG_IP_ALWAYS_DEFRAG" option.

6 Connection Authorization and State Maintenance

In this section, we discuss the detals of how connection state is maintained and used in the routine `spf_ip_forward()`.

From the user's point of view, a connection can be in one of the three abstract states, namely, initiation, normal and termination. We implement these abstractions using a more detailed state diagram as shown in Figure 1 which is actually a simplification of the possible states a TCP connection can go through. We now discuss the transitions and the actions taken at each transition.

A state table is maintained by the routine `spf_ip_forward()` (basically to verify if a packet belongs to an *authenticated* connection). Every arriving packet is first checked to verify whether the packet belongs to an authorized TCP connection or not. A connection is authorized by processing the first SYN packet as follows. When a SYN packet arrives, a state table entry is created for the connection, identified by the four fields of the packet, namely, `src_ip, src_port, dest_ip, dest_port`. The state of the connection is set to T_SYN_RECVD (see figure 1). The rules are now checked to find out the

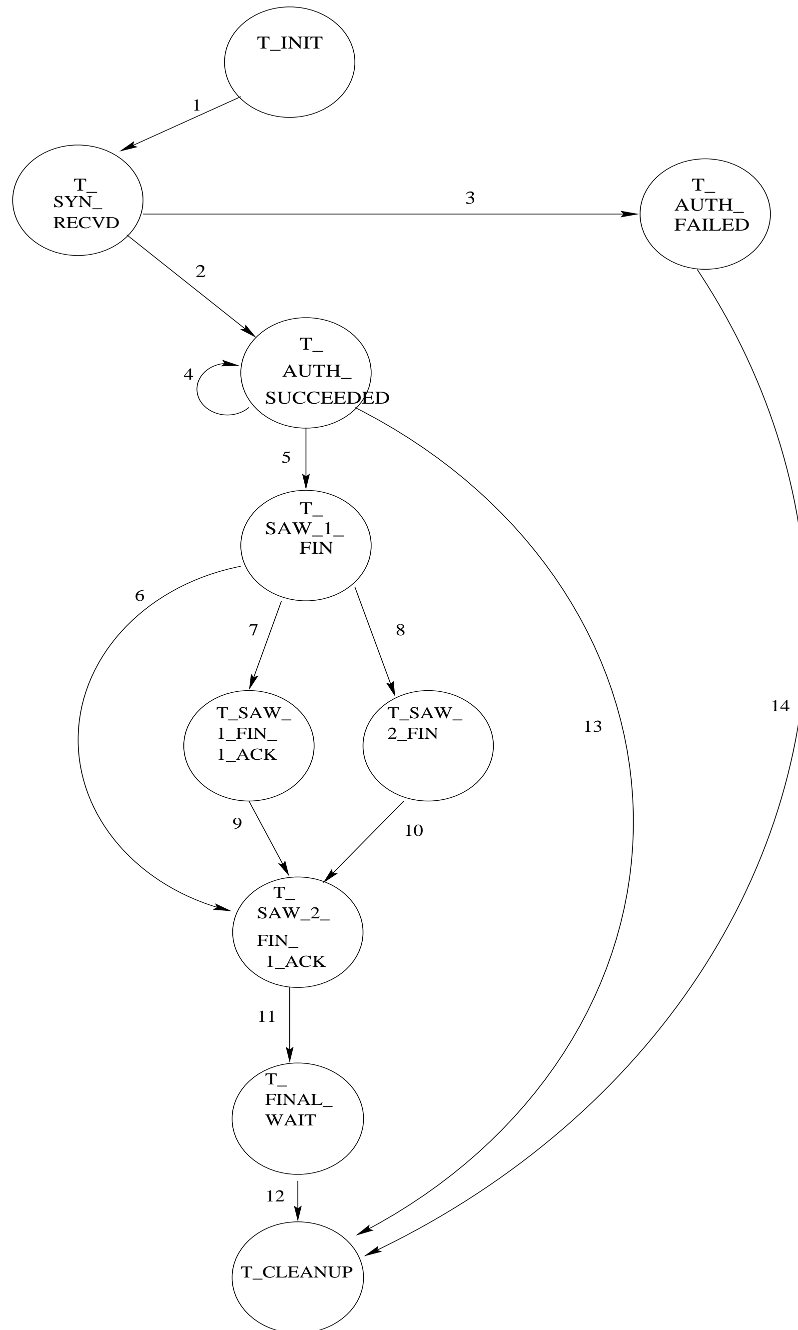

Figure 1: Connection Authentication State Diagram

policy prescribed for this connection during the connection intiation state. The resulting value could be any of the previously explained action values.

If the resulting value is either SDW or F, the state for the connection is set to T_AUTH_SUCCEEDED and the kernel function `ip_forward()` is called. If the action prescribed is `SAW_perm`, the packet is queued and the security daemon is called. The interface between the kernel and the security daemon will be explained shortly. When and if the daemon returns with an affirmative answer, the state for the connection is set to T_AUTH_SUCCEEDED and the kernel function `ip_forward` is called. If the value returned is D, or the daemon returns with a negative answer, the state of the connection is set to T_AUTH_FAILED state, then its state table entry is deleted.

Since TCP connections are duplex, once a connection is authorized, then an entry corresponding to opposite half of the duplex connection is created and its state is also set to T_AUTH_SUCCEEDED. A connection which is in the authorized state (i.e, T_AUTH_SUCCEEDED) remains in this state until either a FIN or RST packet is seen.

For all packets that are not the intial SYN packet, the following principle is followed. *If an arriving packet is not a SYN packet, and the state table entry for that connection is either absent or is in T_AUTH_FAILED state, then the packet is dropped* (and an ICMP error message sent back). TCP packets that are part of the 3-way handshake protocol, but are not SYN packets, namely, SYN-ACK and ACK packets, are forwarded without checking against the rules only if the connection is in an authorized state. Otherwise such packets are dropped. Similarly, if a rule specifies some action to be taken upon termination of a connection, such a rule is triggered when the first RST or FIN packet is seen. Subsequent FIN/ACK packets are forwarded subject to the above principle.

The Table 1 summarizes each of the transitions.

7 Communication between Kernel and Application process

An important part of the implementation was to provide a robust means of communication between the kernel and an application level process/daemon

Transition	Receive	Action
1	Connection Initiation(SYN)packet	Create State Table entry
2	Authentication Success	Continue with the Connection
3	Authentication Failure	Reject the Connection
4	Packet of the Authenticated Connection	Forward
5	FIN packet	Forward and Remember
6	FIN/ACK	Forward and Remember
7	ACK for a FIN	Forward and Remember
8	FIN	Forward and Remember
9	ACK for FIN	Forward and Remember
10	ACK for FIN	Forward and Remember
11	ACK for FIN	Forward and Final Wait
12	Timeout	Final Cleanup
13	RST	Forward and Cleanup
14	Auth Failed for the Connection	Send an Error and Cleanup

Table 1: Transitions

implementing the security policies. In this section, we discuss this issue and then discuss how data packets are processed.

The rules based system requires the facility that packets be selectively sent to the application level and that kernel should wait for a response from an application process. This however implies that the kernel cannot simply use `msg_snd` and `msg_rcv` primitives for communication. If `msg_rcv` (generally blocking) is called by the kernel, the scheduler detects it as a call from an active instance of a bottom handler and will not schedule any process. This will cause the machine to "hang" Therefore, we made use of the netlink driver interface discussed below.

7.1 Netlink Driver

The netlink driver interface is a framework provided by the kernel and can be used for direct communication between the kernel and an application space process. This can be treated as a driver for a dummy device with a certain major and minor number from the application process point of view. Reads on that device by the process will block on the event that something is written to a kernel queue. The kernel writes to the queue and wakes up the sleeping

process. Similarly, writes from the device will trap into the kernel and execute the "callback handler" for the device inside the kernel.

7.2 Processing TCP data packets for authorized connections

A data packet arriving on an authorized TCP connection is first checked against the rules to determine the policy prescribed for the packet. If the packet is to be sent to the daemon and the kernel has to wait for a reply (`SAW_perm`, `SAW_packet`), then the packet is queued in a suspense table. If the policy is either `SAW_perm` or `SAW_packet` or `SDW`, the packet is written to a device managed by the netlink driver interface. If the policy is either to Drop (D) the packet or to Forward (F) the packet, then the appropriate action is taken without writing to the device.

An interesting part of the implementation was the communication of the daemon with the kernel, initiated by an action of the type `SAW_perm` or `SAW_packet` in a rule. The daemon writes to the device, which traps to the kernel and executes the callback handler for that device. This callback handler, written by us, (called `trustd_callback()`) checks the return value from the daemon, updates the state of the connection in case of SYN packets and/or forwards the packet.

7.3 Avoiding Reentering Non-reentrant code

A subtle problem is possible in a straight forward implementation of our callback handler `trustd_callback`. The handler calls the function `ip_forward` which is a part of the net bottom half handler code. *But the net bottom half handler code is non-reentrant.* Here there is a possibility for a subtle race condition wherein just when we call `ip_forward()` from the callback handler, an interrupt might occur and after the interrupt the netbottom half handler is scheduled. *Thus we might have a condition in which the* `ip_forward()` *routine is reentered, which could lead to malfunction.*

We therefore need to make use of a synchronization mechanism in order to avoid above race conditions. The solution we have implemented for the above problem is to create a critical section out of the portion of non-reentrant code which could possibly have been re-entered. This portion of the code has

been enclosed inside the primitives `start_bh_atomic()` and `end_bh_atomic()`. If any part of the bottom-half handler is active, then these primitives will temporarily disable/enable the bottom half processing respectively. It also has the additional advantage that interrupts are not disabled during this time – only the bottom half processing is disabled . This is sufficient as this code might compete with a bottom half handler code and not directly with interrupt handler code.

8 Prototype Daemon

The protoype daemon currently implements functionalities like user authentication and maintaining state related to some applications - FTP and TELNET. For authentication of the internal users, it currently uses the "ident" protocol. It uses a simple database to store access control lists corresponding to userid, machineid and application level commands. The daemon, currently implements a prototype FTP proxy by remembering the FTP commands on the control channel and using this state and the database to decide whether to allow the subsequent data connection or not.

9 Conclusions and Future Work

In this paper, we have presented a rules based kernel interface which can be used by the application level process to specify *selective* packet *relaying*. The rules may be specified to implement site and application-specific security policies in an efficient manner. A prototype of the rules mechanism has been implemented by modifying portions of the Linux kernel (version 2.0.34). There are three advantages of this rules based mechanism. Firstly, the kernel remains lightweight and does not need to be aware of application level protocols. Secondly, the rules may be written so that every packet need not be sent to the application. This reduces the number of context switches between the kernel and the application-level processes. Finally, the rules mechanism is general in the following sense. Rules may be written to intercept every packet of a TCP connection, effectively simulating an application proxy. On the other hand, rules may be written to simulate packet filters in which no packet is sent to the application layer.

We are immediately preoccupied with consolidating the current implementation and making it more efficient. Several extensions to the current work are being explored. One possibility is to allow an application to specify patterns (regular expressions) and an action trigger, such that if a matching pattern is detected by the TCP layer of the kernel of the firewall machine, then it triggers a change in the way data flowing through the connection is monitored. Another possibility is to allow the matched pattern to be modified (in a simple manner). This facility could be used to specify IP masquerading as well as to allow masquerading within the data protion of TCP (eg: masquerading the IP address argument of the PORT command of the FTP application). This latter facility avoids a potential context-switch.

Another direction of work is the design and implementation of a kernel TCP level security module. All application specific security daemons communicate with this security module rather than with the IP layer as currently implemented. The security module would implement TCP Sliding Window Protocol on the packets sent to it by the rules mechanism. This design would further easily allow pattern matching on application data.

10 Acknowledgements

The authors are thankful to Profs. Deepak Gupta and Dheeraj Sanghi of the Department of Computer Science and Engineering at IIT Kanpur for valuable discussions and feedback. We are also thankful to Mr. Amitabh Roy of the ERNET project, Department of Electrical Engineering at IIT Kanpur for valuable guidance in the internals of Linux networking.

References

[1] David A.Rusling. Linux Kernel. URL http://www.metalab.unc.edu/mdw/LDP/tlk/tlk.html, 1996.

[2] William R. Cheswick and Steven R. Bellovin. *Firewalls and Internet Security.* "Addison-Wesley Publishing Company", 1994.

[3] M. Leech et al. Socks Protocol Version 5, Internet Draft, March 1996.

[4] Michael Beck et al. *Linux Kernel Internals.* "Addison-Wesley Publishing Company", 1996.

[5] Sun Microsystems Inc. Deploying Sunscreen EFS White Paper. URL http://www.sun/com, 1997.

[6] Checkpoint software Technologies Ltd. Checkpoint Firewall-1 White Paper. URL http://www.checkpoint.com, June 1997.