Enhance Network Security with Dynamic Packet Filter

Heikki Julkunen[1] and C. Edward Chow
Department of Computer Science
University of Colorado at Colorado Springs
Colorado Springs, CO 80933-7150
Email: {jhjulkun, chow}@quandary.uccs.edu
Telephone: (719) 262-3110
FAX: (719) 262-3369
WWW: http://www.cs.uccs.edu/~chow

Abstract

This report presents the study, design and implementation of a firewall, in particular a major component of a firewall: the dynamic packet filter. A packet filter may be *static* or *dynamic*. A dynamic packet filter checks on the fly the outgoing IP packets from a computer and then allows incoming packets to get through the packet filter if the packets are from the same computer as the outgoing packets were sent to. There is currently no dynamic packet filters on the Linux operating system which has been chosen to be the development and test environment due to the source code availability. Some performance measurements have also been obtained to show that a safe system does not necessarily have to be very slow. This might otherwise be of some concern, as there is a trade-off between the security and the performance of the system.

1. Introduction

The need for network communication is large, and seems to be growing if judging the ever growing numbers of connected hosts to the Internet, the largest worldwide network. However this also means that the issue of security is becoming very important, as the number of hosts implies that the number of potential intrusions will increase as more people start using the net. However, some means of controlling access to computers are already present with special programs called ‘firewalls’, an old fire-fighting term for controlling damage when a fire has erupted.

When TCP/IP was designed, a very limited number of hosts were coupled, and nobody thought Internet would grow the way it has done. Thus Unix (which is the primary OS of Internet connected hosts) is designed to allow several users at the same time to make the cost per user as small as possible. All normal operations of a Unix computer can be done by remote, but this becomes today a security risk, as a user who might be on the other side of the world may be able to break in and steal information. When the security increases, the performance of the system may degrade. There is a trade-off between usability and security, as it will be more difficult to log into a computer from ‘outside' of the firewall.

1. Heikki Julkunen was a graduate student at University of Colorado at Colorado Springs when this work was done; he is now with NeuroData.

1.1 How is a break-in achieved

A break-in to a computer is done by finding a valid login/password combination to a computer. This can be done in several ways: exploiting bugs in existing code, cracking the password file, or IP spoofing[1].

Once a password file is found on a system, there are several programs that can be used to guess the login/password file. Studies indicate that users tend to choose passwords that are easily remembered like words or names. The password cracking programs take advantage of this as they use word lists when guessing the passwords. Studies indicate that 10-20% of all passwords are easily cracked within a few hours! [24].

The primary goal of the intruder is often to crack the 'root' password. The 'root' user of a Unix system is the system administrator, who has total control over the computers in the local network. However, the goal might just be to read an important directory. In that case it is sufficient to crack the password of a user who has access to the directory.

1.2 How will a Firewall help?

Firewall is slightly misleading, as a 'real' firewall will stop fire in either direction, whereas a computerized firewall will behave like one-way street. Traffic should thus be able to go though the firewall, but incoming traffic is thoroughly examined. As traffic though the firewall is monitored, illegal attempts to log into computers 'behind' or 'inside' the firewall will be much more difficult than without a firewall. To achieve maximum security, it is advisable that several firewalls are installed. This is sometimes hard to do as this would mean that the actual physical network has to be rewired. This is the case here at UCCS, if firewalls are to be installed for each department, large parts of the cables has to be rewired.

1.3 How a firewall works

Unfortunately almost everything that enhances the network security is called a firewall, which means that when two people are talking about firewalls they are not necessarily talking about the same thing. However, the two main classes of firewalls are the packet filter firewall and the proxying firewall [27].

1.3.1 Packet filter firewall

A packet filter is a special router or a program that examines the IP packets received and decides if packets are to be forwarded by checking a set of rules that specify what is to happen with the packet. The good thing about packet filters is that it is transparent to the user and the applications. All current software can be used without modifications, but the firewall manager must know a great deal about the installed software of the system in order to set up rules for the filter in order to keep the

1. IP spoofing is when a computer claims to be another one by using a trusted IP address of another computer, and thus gaining access to another computer. Discussed in Section 1.4

software running. The default rule is usually to reject everything that is not explicitly specified to be allowed to pass. The default rule might also be to allow everything to pass except the packets that are explicitly set to be denied, but in that case the firewall is easily bypassed. A *simple packet filter*, or PF for short, only checks the sender/receiver of the packet, and what type of protocol is used, but it does not usually check the TCP connections status flags, nor can it check UDP packets on the fly. The TCP flags are used to allow incoming calls **after** the host on the inside of the packet filter has started the conversation. This is hard to do with UDP since it is connectionless.

A packet filter that checks all outgoing TCP and UDP packets and only allows incoming packets to the same port is called a *dynamic packet filter*, hereafter called DPF.

1.3.2 Proxy firewalls

A proxying firewall is an intermediate server that both the internal and external host talk to. The user thinks he is using the external host directly, but everything is relayed through the proxy host.

The disadvantage with a proxying firewall is that all the existing software has to be modified in order to be able to access the outer side of the firewall. One of the more known proxy services is called SOCKS, and new software usually have support for SOCKS. One of the more popular programs that supports SOCKS is the web browser Netscape[31]. However, most software packages that come with most Unix distributions do not support SOCKS at this point.

The SOCKS manufacturer usually delivers a dynamically linked shared library which makes all the network programs aware of SOCKS. However it is not certain that the system uses shared libraries at all, and in that case all applications has to be recompiled with the new library. It is a major task to compile and then make sure that all hosts on the network has the correct version installed.

Figure 1.1 shows the server and client model of SOCKS, where the client is sandwiched between the transport and application layer of the TCP/IP protocol stack, and the proxy server is running as an application on the server. The actual server does not need any modifications.

1.3.3 Screening routers

A screening router is either a special piece of hardware that has two or more ethernet devices attached to it, or a computer with two or more ethernet devices. If the screening router is a hardware device, configuration software is included for maintaining the rules about what packets are allowed to be routed.

If the screening host is a computer, it has to run special software that can do the same thing as the hardware device can. This software is the packet filter, as described in Section 1.3.1.

The general name of this function is the screening router, but as both actually do packet filtering. We will mainly use the term packet filter.

The rules for setting up both the screening router and a PF generally don't have the same syntax, but they should be able to do the following:

- Accept a packet from an IP source to an IP destination. It should be possible to setup what ports are allowed. It might be useful to be able to set a range of accepted ports, or any port. It should also be possible to select what protocol is allowed.

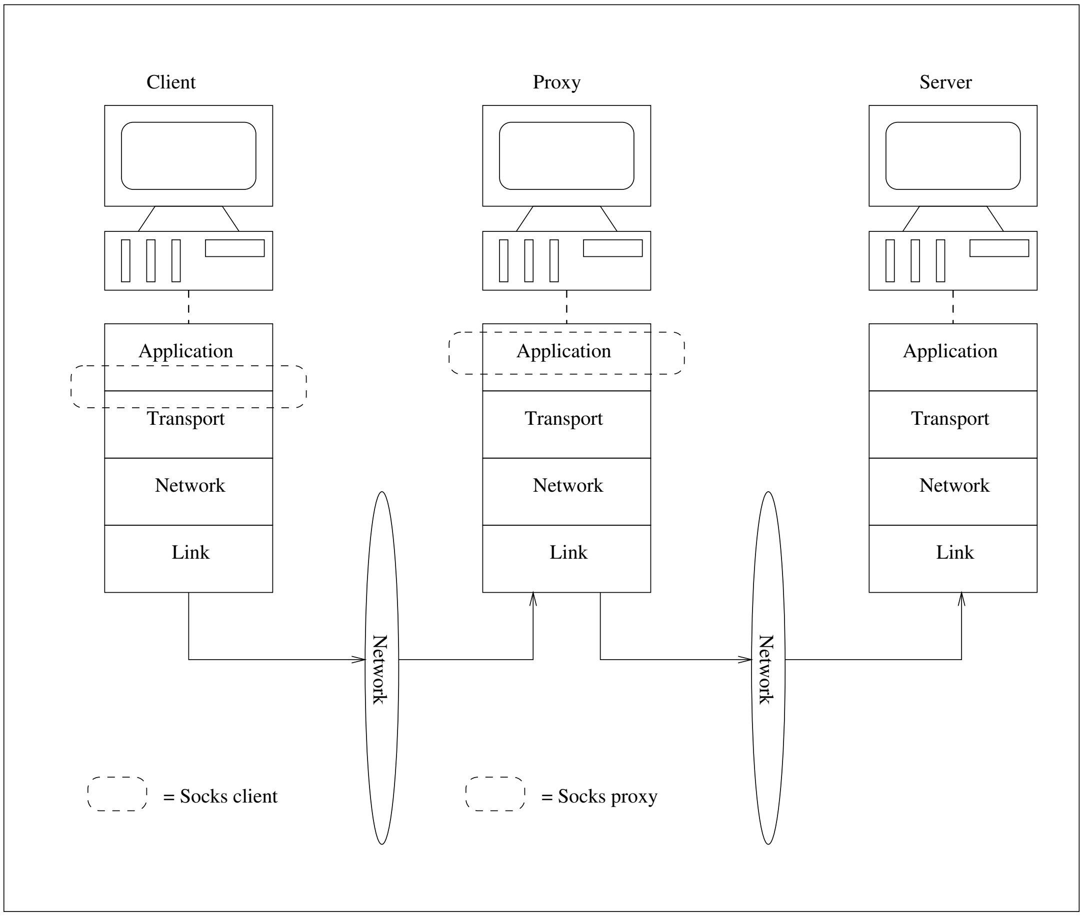

Figure 1.1: SOCKS at work

- Deny. The same as for Accept but deny the packet to pass.
- Default policy. When no rule applies, this sets the default policy.

It is also handy if it is possible to set the rule before or after another rule, as the order of the rules might determine if the packet is allowed to pass or not.

1.3.4 Router or Gateway?

There is a lot of confusion about if the term gateway or router should be used. Both do more or less the same job of forwarding packets from one segment of the ethernet to another one. A segment is a networking term that means that hosts are using the same physical media when communicating with each other. If thin or thick ethernet is used it is especially important that the network is segmented as packet collisions will otherwise occur when hosts are trying to send at the same time. Segments are often assigned different subnet numbers. A subnet is a logical arrangement of the IP addresses used. As UCCS uses a type B class IP address of 128.198.x.y, it makes it easy to use ‘x’ as a subnet and ‘y’ as a host on the subnet. Routers binds together the segments so that packets can traverse the entire network.

According to Stevens[5] the term gateway should be used when talking about an application gateway that connects two different protocol suites like TCP/IP and for instance IBM's SNA. The term router should be used when forwarding packets between segments. We will thus mainly be using the term router, but many of the Unix diagnostic tools will use the term gateway when they actually mean a router.

Other terms for routers are bridges or repeaters. A bridge or a repeater is a device that forwards packets between segments. The forwarding is usually done in hardware, which means that **all** packets are forwarded, even packets that should stay in the local segment (the packet was to another host on the same segment). This is also what happens here on UCCS, if a packet reaches the backbone, it is passed to all computers. The UCCS backbone has thus (unnecessarily) quite heavy traffic. One simple solution to the traffic trouble of the backbone is to install *hubs* with twisted pair ethernet. Unlike thin or thick ethernet, twisted pair can only connect at most two hosts at the same time. Instead of using the ‘bus' architecture of the thin or thick ethernet, the hosts are connected in a ‘star' architecture with a hub. By connecting hubs to other hubs, the traffic can traverse the network. The nice thing about hubs is that if traffic is between two of the hosts on the same hub, the hub realizes this, and the traffic never goes out to the backbone.

Figure 1.2 shows the same number of hosts on two different network architectures, the bus

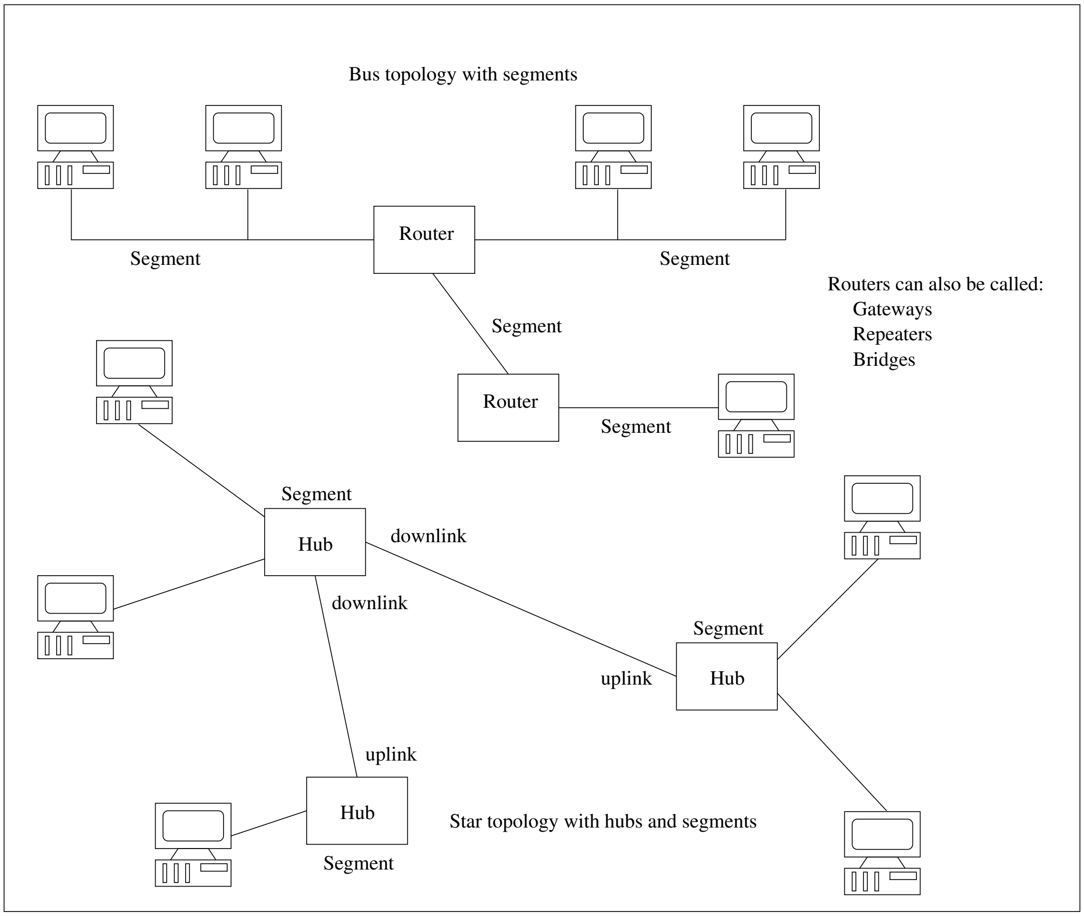

Figure 1.2: Network Topologies

architecture and the star architecture. In the example there is little to be gained by using the star architecture with hubs. However, if the network grows, the star topology will scale much better as a hub will not allow collisions on each segment. Each hub controls a number of hosts. When more hosts are needed than a single hub can handle, another hub is bought, and the new hub is connected to the old hub in another star. The link to the top central hub is called uplink or downlink (depending on which direction the traffic is supposed to go). If two hosts on the same hub are communicating, the hub will not forward the traffic to the up or down links. The hub thus acts as a router as well. When hubs are connected, the up and down links should be kept to a minimum, i.e., the topology should be as flat as possible. The single host on the hub in the leftmost bottom is a bad example of how hosts should be connected. The host should be connected directly to the uplink hub instead. Perhaps the idea was that the single host should soon be joined by other hosts.

When scaling up a bus architecture, collisions will eventually occur and network performance will go down. ‘Smart’ routers can be used on the bus architecture that will not forward packets blindly like repeaters or bridges usually do.

1.4 Firewall Architectures

Usually a firewall uses both packet filters and proxy servers to achieve maximum security. In Figures1.3, 1.4 and 1.5 some of the most common setups are shown. Figure 1.3 shows the dual homed host architecture, Figure 1.4 shows a screened host setup and finally Figure 1.5 shows the perimeter network setup.

The difference between the setups is the amount of extra hardware used, for instance screening routers. By adding screening routers more security is added, but more things have to be setup. For instance all the screening router rule sets have to be carefully examined so that normal and allowed traffic can pass the screening router, and everything else is stopped. This sounds easier than it is, as the rules for setting up a screening router are usually cryptic. Also, when new network software is installed on the intranet hosts, the rule sets have to be maintained[1]. The expenses for the extra hardware should be of no concern if enhanced security is the goal.

It may be more cost-effective to use a computer and run a PF program than buying a screening router. Running a PF program does not usually take that much computing power so even an older computer may be adequate to carry out the task. However, if a DPF is used, the computer may need additional memory as additional information is stored. How much memory will be used is hard to tell, as it depends on the amount of traffic and how fragmented the packets are. To send a never ending stream of fragmented packets without sending the first packet to a host running a PF program can actually make the host to consume all available memory.

This has to do with the way the IP packets are split (fragmented) by the IP layer when the data passed from the TCP or UDP layer is larger than the maximum size allowed on the device to be used. When a packet is fragmented, only the IP header is copied to the new subpackets, not the TCP or UDP header. As the port number is stored in the TCP or UDP header, a PF has to store all fragments until it gets the first packet which specifies the receiving port number. If the first packet never arrives, the memory eventually runs out. This method of ‘attack' is called denial of service.

1.4.1 Dual homed host

The configuration of the dual homed host in Figure 1.3 can supply excellent security, if the host used does not allow routing of packets from the inside to the outside\footnote {It is important to check this, some Unix versions have routing turned on by default, and packets that arrive on one of the network devices will be passed to other. To be able to use routing of packets in Linux, you must recompile the kernel.}. To be able to access the other side of the network, users must log onto the dual homed host and perform network access from this computer.

1. It is quite easy to get conflicts in the rule sets, so one can not just add a rule and hope for the best.

However, this firewall architecture is not transparent to the user, as he has to know the name of the dual homed host, and have a valid login/password on that computer to be able to access the outside world. Also, if the host is compromised, all intranet hosts are wide open for attacks.

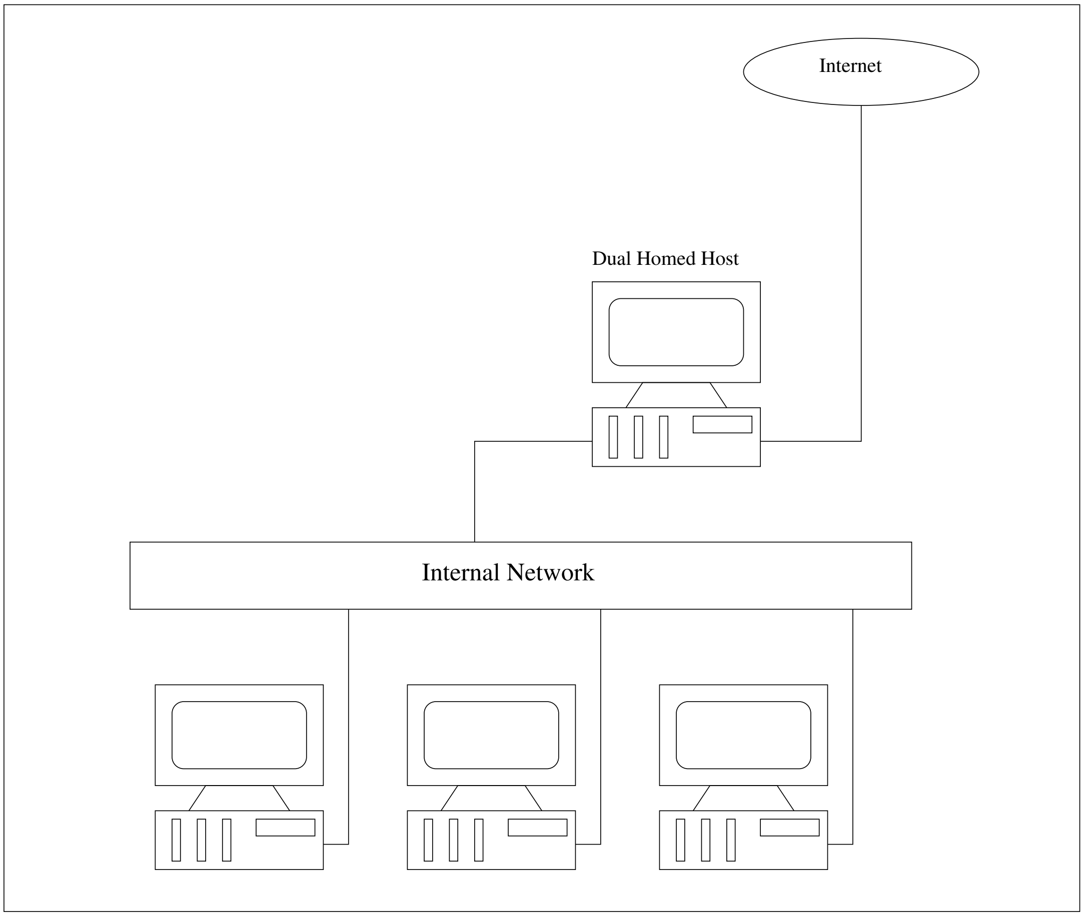

Figure 1.3: Dual Homed Host

1.4.2 Screened host

If the users on the intranet are to be able to use the internet in a more transparent way, the screened host configuration in Figure 1.4 may be used. The bastion[1] host provides the necessary proxying, see Section 1.3.2 for a brief explanation of what proxying is. The proxying is necessary as the screened router is setup to only allow packets from the bastion host to pass through it. The bastion host is a good place to put network services using protocols like SMTP, HTTP, FTP, etc. See Section 1.6.1 for a brief description of the network service protocols.

The weakness of this setup is that the bastion host resides on the intranet, and if the packet filter is misconfigured and the bastion host bypassed, the attacker have access to the intranet. A positive factor about proxying is that it keeps the names or the IP addresses of the hosts on the intranet away from the internet. This makes it harder for an attacker to select a host to attack.

1. A bastion host may also be called a stronghold.

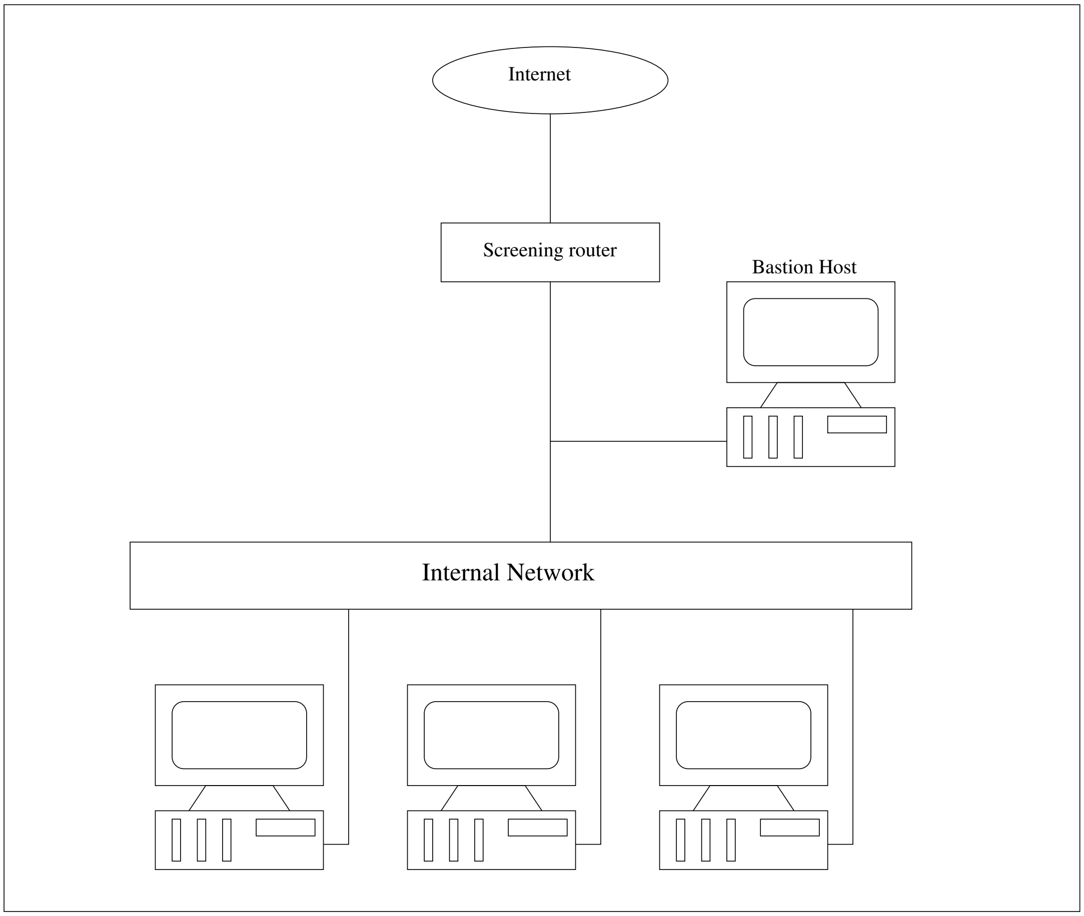

Figure 1.4: Screen Host Configuration.

This perimeter setup is shown in Figure1.5. The weakness of having the bastion host of the screened host setup on the intranet is corrected by setting up a perimeter network, also called ‘Demilitarized Zone’, where the only traffic allowed is to/from the bastion host and the screening routers. To verify that this setup is safe, a device[1] with packet sniffing capabilities is allowed to sniff the packets that pass by. If all packets are to/from the bastion host, everything is ok.

This setup is the safest of the firewall setups mentioned so far, since an intruder has to execute the following steps in order to get to the intranet:

- Pass the first packet filter to give access to the bastion host.
- Break into the bastion host.
- Pass the second packet filter to give access to the intranet.

1. Either a actual hardware device or a computer with a suitable program like tcpdump, see Section 2.7 for a description of tcpdump.

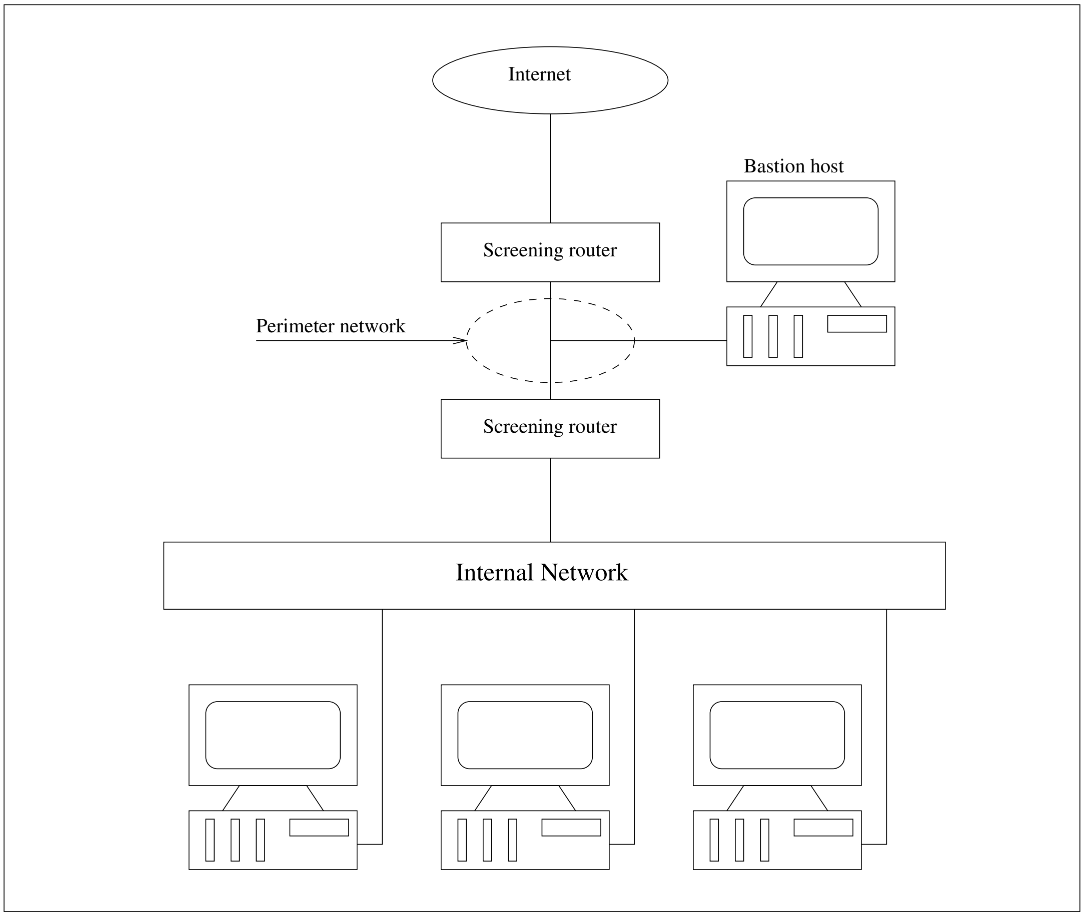

Figure 1.5: Perimeter Network.

1.5 IP spoofing

IP spoofing is when a host claims to be another host that is trusted by the host to which access is wanted by trying to use the trusted hosts IP address.

IP spoofing is however not that easy to do, as the offending computer has to be on the same segment as the real host in order to succeed. This is because packets are routed by masking the IP address with the network mask, and by the resulting IP address determine how packets are routed. The effect is that is it quite possible to use an already taken IP address, and to send packets to the host you want to gain access to, but when the host in turn replies the packets are routed not to the illegal host but to the official segment they **should** originate from. However, if this is the case an attacker can penetrate a host on the real segment first. Once this is done it is not to difficult to arrange IP spoofing. In order to succeed the real host must be silenced somehow as chaos occurs if two hosts share the same IP address. If all these steps succeeds it is possible to make a 'tunnel', i.e. another IP envelope is added to the original to be able to send it to the illegal host which simply removes the additional IP envelope and processes the packet in the usual manner. When sending data the same process is repeated in reverse order.

DPF

Another possibility is that the offender has successfully cracked into one of the major routers, and is able to successfully change the routing tables to make the packets arrive to the offending computer. In order to maintain the charade the offender must also disable the dynamic routing table update, as the route otherwise might be changed due to congestion.

An attempt at IP spoofing is discusses in detail in [39].

2. Design and Test environment

This section will discuss the design and testing environment used when creating the DPF. The first sections discusses the operating system used, what it offers and how it can be used to run a firewall. The next sections discusses where and how to add the enhanced DPF functions. After these sections the workings of the DPF will be explained, and how to properly test the developed code.

2.1 Operating System

The Linux operating system was chosen as a platform and ran it on three dedicated research computers as we then can run programs as ‘root’ (packet filter programs and similar often require this) and without disturbing the usual traffic on UCCS.

What is Linux? The following is a quote from the Linux webpage[29]:

“Linux is a freely-distributable implementation of UNIX for 80386, 80486 and Pentium machines. It supports a wide range of software, including X Windows, Emacs, TCP/IP networking (including SLIP/PPP/ISDN), the works. Linux was originally written by Linus Torvalds, in Helsinki, Finland. After over 2 years, Linux has become one of the most popular free unices available, and is continually being developed by Linus and teams of people all around the world. The linux kernel is covered by the Gnu Public License (GPL), and is usually bundled with various binaries that comprise a working unix operating system.”

As Linux includes the entire source code to the kernel, it is well suited for research and modifications in system related projects. However, there is currently no documentation about the kernel, and it is hard to find out what actually goes on in the kernel. See Section 5 for an investigation of the kernel source. There is no ‘standard, distribution of Linux, so there are various flavors of Linux around. Even though they usually share the same tools, the layout of the system is usually different.

The setup and maintenance is different depending on what type of Linux distribution is used. The distribution used when developing the main part of the DPF is Debian v1.1, which is a freely distributable version of Linux. Like the Linux kernel, Debian is maintained by a number of volunteers. More information about Debian can be found on the Debian webpage[30].

2.1.1 Linux firewalls

Before anything can be improved, one must examine the existing software and see what it can do. The Linux kernel supports PFs which can be used to implement a firewall. A general description of the rules used to maintain a screening router or a packet filter is in Section1.3.1.

DPF

The PFs provided by the kernel is actually four different filters which are maintained by the same rules, but the rules will be applied to packets at different places. This makes the syntax of the rules easier, once it is understood where the rules are applied. The Linux ‘man' pages often talk about ‘chains' which means the linked list for a PF.

The PFs provided are the following:

- Incoming PF. All incoming packets are checked against the rules in the incoming PF chain.
- Forwarding PF. All packets that need to be forwarded are checked against these rules. The packet must pass the incoming PF before it is checked against the rules in the forwarding PF chain.
- Outgoing PF. All outgoing packets are checked against these rules. A packet that is being forwarded has to first pass the forwarding rules, then the outgoing ones.
- Accounting PF. All packets (incoming, forwarding and outgoing) are checked against these rules, and for every rule there is an associated counter that gets increased for each packet matched against the rules. Mainly used for statistics.

In the worst case each packet has to pass three PFs, the incoming, forwarding and the outgoing one.

There is also a default setting for all PFs (except Accounting) that can be set to:

- Accept. If no matching rule is found, the packet is passed.
- Reject. If no matching rule is found, the packet is denied, and an ICMP packet is sent to the sender stating ‘host unreachable’.
- Deny. Like reject, but no ICMP packet is sent back.

2.1.2 Administrating the PF rules

To setup a rule, the *ipfwadm* program is used. For example, *ipfwadm -I -a reject -P all -S badhost -D 0/0* is used to append a rejecting rule to the incoming packet filter from host ‘badhost’ with the destination ‘everywhere’(0).

To clarify the example, the arguments of the ipfwadm are as follows:

- -I, set a rule for the Incoming PF. Similarly the -F specifies the forwarding PF, and -O specifies the outgoing one.
- -a reject, append a rejecting rule. It is possible to add rules either at the end of the PF chain or at the beginning. As soon as the PF finds a matching rule it stops looking, so the order can be critical. To insert a rule at the beginning, the switch -i is used. The rules inserted or appended may be: accept, reject or deny. To remove a rule the -d switch is used. When removing a rule, the first matching rule from the beginning of the chain is removed.
- -P all, the rule is used for all protocols. The protocols are TCP, UDP or ICMP.
- -S badhost, the source address is ‘badhost’ (using host name lookup) or an IP address in dotted decimal. It is possible to make the rule match a specific port or a range of ports, or up to 10 different ports (counting both source and destination ports) with the same rule. A port range is specified by lo:hi. If more than 10 different ports are to be specified, separate rules must be used.

- -D 0/0, the destination address is 0, and the network mask is 0. The network mask is the number of binary ones starting at the most significant bit. When the PF matches an IP address against a rule it takes the IP address, bitwise ANDs it with the network mask and then checks if the address of the packet and the rule is the same. The default network mask is 32 bits, and if the network mask would have been forgotten, it would have resulted in using a worthless rule. The reason is that the destination bitwise ANDed with 0xffffffff will not match zero. It is very unlikely that packets are sent to a host using an IP address of zero[1]!

By using the -c switch, the PF in question is queried if the packet specified by the other switches would have passed. Useful if a complex set of rules is to be tested. Like many other Unix tools, the argument following the switch is the value to use.

Before designing the DPF, the PFs are evaluated to see that they actually worked as described in the documentation. The setup used for this was the same as the setup used later when evaluating the DPF. For instance, one of the tests was to get Network File System (NFS) working by applying all the rules necessary by sniffing with the tcpdump program[2] to find out about which rules were needed.

2.2 Design prerequisites

After testing and confirming the function of the existing PF code of the kernel, the next task was how to add the wanted DPF functions. In order to understand where to add new functionality, one need to examine Linux kernel in depth. It is not an easy task as the kernel is large. Even though the source is relatively well documented, it is hard to understand how different parts of the kernel works together.

Traces are used to learn what happens when the kernel is running, how the system is initialized, and how packets are processed. As a rule of thumb, when forced to do changes of the kernel, the different functions are treated as black boxes and only new function calls to the DPF are added. There are three important issues need to be addressed: timer support, PF manipulation, and kernel debugging.

2.2.1 Timers

The DPF code has to maintain a timer which is decreased periodically. This was found with a ‘grep’ search for ‘timer’ and this was obviously supported by the kernel.

2.2.2 PF Manipulation

By tracing the execution of the ipfwadm program, we were able to figure out what was changed in the kernel when changing the PF chains. This was however not an easy task as the Linux designers likes to use pointers to functions. Also, as the kernel can not be debugged in real-time it is not easy to find out what is actually being executed.

1. An IP address of 0.0.0.0 does not exist, the meaning of this address is the default router.
2. See Section 2.7 for more info about the tools used.

2.2.3 Kernel Debugging

It is also hard to debug the DPF as stated it is impossible to run the kernel with a debugger. There must be some way of printing the internal status of the program while it is running. The Linux has a /proc directory which does not contain ordinary files, but special 'files' that invoke some predefined functions in the kernel. The DPF uses the file /proc/net/ip_dynamic to print out information about the internal status. In this case, a couple of the functions was checked that did something similar. Especially the IP masquerading subsystem has been used. IP masquerading is used to 'masquerade' the IP address of a host on the intranet to another address. Masquerading is mainly used when a host is connected to the internet with a modem using the Point to Point Protocol (PPP), or the Serial Line Internet Protocol (SLIP). The disadvantage of a point-to-point connection is that the IP address given is a host IP address, and no routing is possible as there are no subnet bits in the address. The masquerading code finds out that a host on the intranet wants to reach the internet, and exchanges the IP address and port of the intranet host with the IP address and an unused port of the host with the point-to-point connection.

Debugging is also possible by sending text-messages to the kernel log daemon (klogd) which in turn resends the messages to the ordinary system log daemon (syslogd). Syslogd can send the messages to either the console or file (or both). The printk() was heavily used. Using the DPF on a busy system created huge trace files (several megabytes) in only a couple of hours.

2.3 TCP/IP stack and DPF

After defining the requirements of the DPF and identifying the necessary software modules, the next step was to investigate where the DPF functionality could be added to the TCP/IP stack. Again, this was very time consuming as very little information of the Linux kernel could be found. Basically the approach was an ample usage of the grep program to search for what looked like interesting features. However, it helps that all the.c files of the TCP/IP stack is in the linux/net/ipv4 directory as it makes it easier to find the interesting files and functions by visual inspection of the code.

It turns out that the existing PF code is placed in the network layer of the TCP/IP stack. The main function is the ip_rcv() which handles all received packets. Following the branches of the code, ip_rcv() calls the ip_forward() function if packets are to be forwarded to another ethernet device.

Since the ip_rcv() function receives all IP packets, this seems to be a good place to add calls to the DPF about packets being received, and the ip_forward function seems to be a good place to add function calls about forwarded packets.

This only solves the problem when forwarding packets that are already traveling on the ethernet, it does not solve the situation where the packets comes from the host running the DPF. Thus calls were added to the DPF when packets are originated locally as well. Since the IP header may not be complete when upper layers call the network layer, the DPF call can not be added at a higher level in the ip code, but as close to the ethernet device transmit call as possible. The function calls to the DPF code is in ip_queue_xmit(). This worked fine most of the times, but sometimes the DPF code never got called and packets 'sneaked' out on the ethernet without being checked first. This implied that other paths existed to send out IP packets. After adding calls to the DPF in the ip_build_xmit()

function (twice due to branches), everything seemed to work. It is not clear why packets sometimes are intercepted at ip_queue_xmit() (the logical place in my opinion) and sometimes in ip_build_xmit(). It is possible that the DPF function calls may suffice with the calls in ip_build_xmit(), but as the overhead is not that great the code was not changed after it works.

The two DPF functions calls are the following:

int ip__dyn_check(struct sk_buff *skb, struct iphdr *iph, unsigned flags)

int ip_dyn_refresh(struct sk_buff *skb, struct iphdr *iph)

ip_dyn_check() is called when packets are sent or forwarded. The function will try to match the IP packet with the internal DPF entries. If no match is found it will create an entry. The flag argument tells the function where it called from. If the DPF was called from ip_queue_xmit()/ip_build_xmit(), the DPF will add rules to the incoming PF. If it was called from ip_forward the DPF adds rules to all three PFs, forwarding, outgoing, and incoming.

ip_dyn_refresh() is only called when packets are received, and will only refresh the timer if a match is found.

The sk_buff structure is the Linux counterpart of the BSD mbuf structure. The Linux version can store blocks of data up to 32KB-1 bytes or 64Kb-1 bytes as indicated by the following skbuff.h include file:

```
#ifdef CONFIG_SKB_LARGE
#define SK_WMEM_MAX    65535
#define SK_RMEM_MAX    65535
#else #define SK_WMEM_MAX    32767
#define SK_RMEM_MAX    32767
#endif
```

The other argument is a pointer of the IP packet being processed. Even though the sk_buff struct holds a pointer[1] to the data of the packet, the DPF was sometimes passed a NULL pointer from the ip_queue_xmit() code[2]. As all calling functions already kept track of a pointer to the IP header, we simply used that instead. The sk_buff struct holds however some other useful information, for instance, the total number of bytes to send. This is used when scanning for the PORT command in the data stream of a ftp packet, Section3.3.

Note the placement of the DPF in the TCP/IP stack of Figure 2.1. The PFs and the DPF is working between the link and network layer. Higher level functions of the network layer don't know that packets may be dumped.

2.4 Setup for Testing

To prove that the developed DPF works, a very straightforward configuration was setup to make testing easier. This means for instance that no proxying software is used, and thus no modifications of

1. Actually a union of several pointers.
2. The NULL pointer made the DPF to do illegal memory searches and the entire system went haywire.

the software on the hosts of the intranet. As the Linux kernel supports three types of PFs, a setup has been chosen that is suited for the kernel PFs, see Section 2.1.1.

The setup shown in Figure 2.1 is a mix between the *Dual Homed Host*, Figure 1.3 and the *Perimeter Network*, Figure 1.5. The difference from the dual homed host setup is that the DPF selectively passes packets between the networks. This makes the setup transparent to the user. The difference from the perimeter network setup is that all functions except proxying are done on only one computer. The test setup does not however make any sense for real secure applications as the computer that runs the DPF is on the intranet. Because of this, the setup looks like the *Screened host* setup as well! The setup is mainly chosen because of its straightforwardness and as UCCS only has one 100Mbps hub which is in use[1], and the fact that it is unnecessary to have a perimeter network in order to show that the DPF is working.

To evaluate the function of the DPF, the tcpdump program is used to sniff on both ethernet devices[2] of Gandalf.

2.5 The DPF in operation

Basically a DPF has to inspect all packets on the fly, and to see if something has to be done for the rules in the PF. This is done by an internal linked list which has the entry points hashed, see Section 2.5.1, to get good performance. As long as something is sent/received a timer is updated on the affected rule, and as long as the timer value is non-zero the rule is kept. When the timer reaches zero the rule is removed from the affected PF(s).

All packets from the intranet are checked in the existing PFs, but as stated in Section 2.4, no proxying is done. It is possible to use static rules for the packet filter, and in fact you must add a rule to allow packets from the hosts on the intranet to use the DPF as a router.

2.5.1 Hashing

Hashing is done to speed up the matching of packets and DPF entries. This is done by creating a hash key of the three-tuple of (Protocol, IPsource, IPsport).

Even though the hashed index calculation is simple, it can be confusing because of the problem with endian order. The network order is big endian, but the i386 architecture uses little endian.

In some cases the code assigns variables using a pointer from the data stream as this makes fast and efficient code. As long as a byte order conversion is done before using the value of the variable it works fine.

The standard network endian conversion functions are:

- *ntohl*. Converts a network long to a host long.

1. At the end of the thesis, another hub was bought, but by then much of the work was already done.
2. This is simply done by opening two windows and setting up two tcpdump programs to sniff on the different network devices.

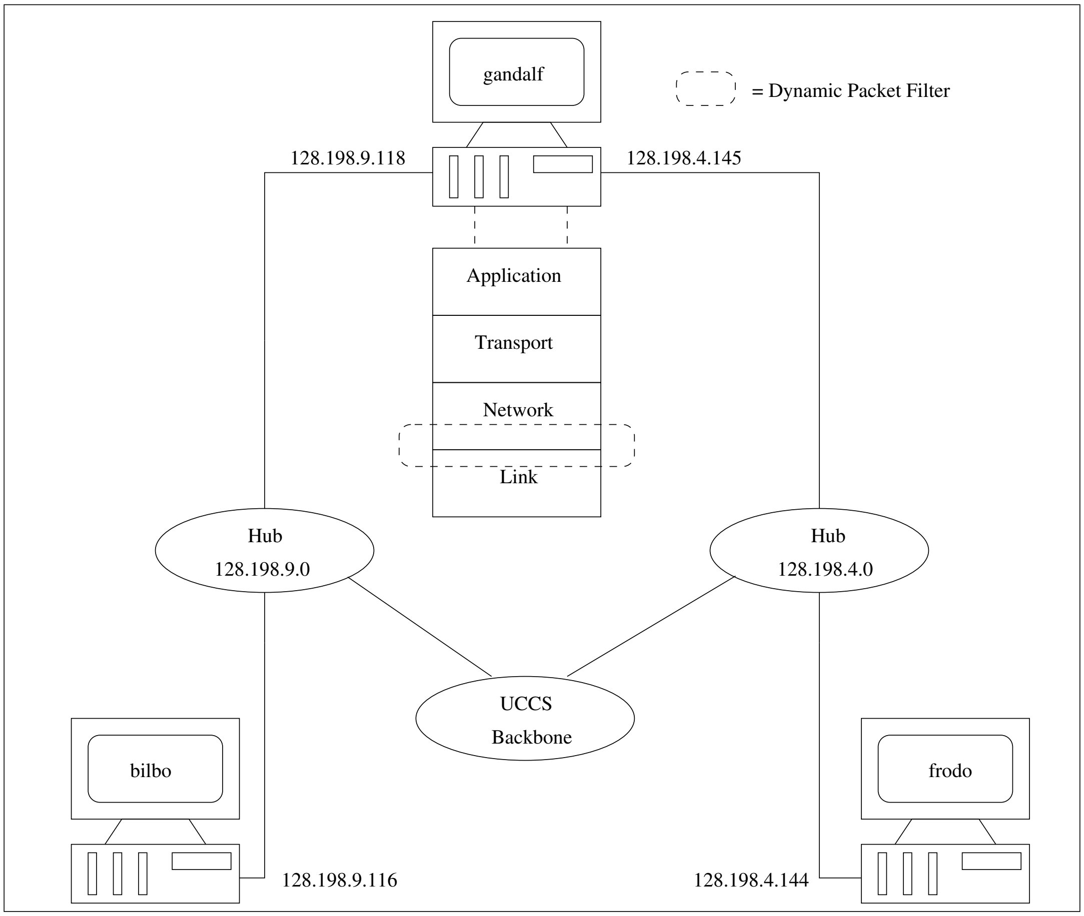

Figure 2.1: Setup used in the tests.

- *ntohs.* Converts a network short to a host short.
- *htonl.* Converts a host long to a network long
- *htons.* Converts a host short to a network short.

The following piece of code shows how the DPF finds out the IP address and port numbers by taking the values from the IP header without any byte order conversion. The long complicated pointer conversion makes a pointer of the first two bytes following the IP header. The pointer iph-$>$ihl returns the length of the IP header, but it has to be multiplied by four first.

```
struct ip_dyn *ip_dyn_get(struct iphdr *iph) {
    __u16 icmp_ports[] = {0, 0};
    __u16 *portptr = icmp_ports;
    int protocol;
    __u32 s_addr, d_addr;
    __u16 s_port, d_port;
    struct ip_dyn *ret;

    protocol = iph->protocol;
    if (protocol != IPPROTO_ICMP)
```

```
      portptr = (__u16 *)&(((char *)iph)[iph->ihl*4]);

   s_addr = iph->saddr;
   s_port = portptr[0];
   d_addr = iph->daddr;
   d_port = portptr[1];
   /* Try first to find the 5-tuple as it is */
   ret = ip_dyn_get_2(protocol, s_addr, s_port, d_addr, d_port);
   if (ret != NULL) {
      ret->flags &= ~IP_DYN_F_REVERSED;
   }
   /* Now try to find the reversed ditto */
   else {
      ret = ip_dyn_get_2(protocol, d_addr, d_port, s_addr, s_port);
      if (ret != NULL)
       ret->flags |= IP_DYN_F_REVERSED;
   }
   return(ret);
}
```

To get the hashed index, the ip_dyn_get_2() seen above will in turn call the following:

```
static __inline__ unsigned ip_dyn_hash_key(unsigned proto, __u32 addr, __u16
port) {
   return (proto^ntohl(addr)^ntohs(port))(IP_DYN_TAB_SIZE-1);
}
```

Example: The netperf server is used to benchmark the performance in Section 4. The netperf server is resident at port 12865 on Gandalf. 12865 is 0x3241 in hex. Gandalf's IP address is 128.198.9.118, which is 0x80c60976 in hex. The netperf uses UDP for the benchmark.

In order to calculate the hash value we first need to know the value of the UDP protocol, which is six. Tables 3 and 4 show what the IP and port number is interpreted as. As the IP address is taken

Table 1: IP long conversion functions

IP	Stored as	ntohl(IP)	Stored as	htonl(IP)	Stored as
7609C680	80c60976	80C60976	7609c680	80C60976	7609c680

Table 2: IP short conversion functions

IP	Stored as	ntohs(IP)	Stored as	htons(IP)	Stored as
4132	3241	3241	4132	3241	4132

from the IP header (which is in network order) the i386 architecture thinks it is handling 0x7609c680. The port number is also taken from the IP header, and thus the i386 architecture thinks it is handling 0x4132. By computing the hashed index value as in the hashing function, the following is done:

```
Hash = (proto XOR ntohl(IP) XOR ntohs(port)) AND 0xff.
```

The result is presented in Table 5.

Table 3: Hash Calculation Example

Variable	Value	Operation	Result
IP	0x7609c680	assign	0x7609c680
Port	0x4132	assign	0x4132
Proto	0x00000006	assign	0x00000006
SIZE	0xff	assign	0xff
Proto	0x00000006	xor	0x00000006
ntohl(IP)	0x80c60976	xor	0x80c60970
ntohs(Port)	0x00003241	xor	0x80c60b31
SIZE	0x000000ff	and	0x00000031
Hash		assign	0x00000031

2.5.2 Outgoing packets

Consider the case when the host running the DPF wants to send something to the outside world. See Figure 2.2. When a packet appears, the DPF checks to see that the protocol is of one of the three supported types: TCP, UDP and ICMP. ICMP does not use any ports, so if an ICMP packet is found, the ports are set to zero. If the protocol is of a supported type, the DPF tries to find a match in the internal linked list of DPF entries. The match is based on the five-tuple of the packet consisting of (Protocol, SourceIP, SourcePort, DestIP, DestPort).

If the five-tuple is not matched, an internal DPF entry is created. When this is done, the DPF will append a temporal rule to the incoming PF as well. The rule in the incoming PF chain will allow external hosts to send packets to the originator of the packet. The temporality of the rule is maintained by a timer. The timer in turn is maintained by the kernel which will do a callback to the time-out function supplied by the DPF. If the time-out function is called, it will remove the entry from the internal DPF list and from the incoming PF chain.

The next time a packet with the same five tuple is matched, the DPF will find the entry, and only reset the timer to a time-out value which can be defined. The time-out value is different for TCP, UDP and ICMP. See Section 2.6.

Before the DPF is called to check a packet, the packet has to pass the outgoing PF chain. This allows the administrator to disable certain protocols and possibly ‘suspicious' hosts.

Note that the source and destination addresses of the rule which is appended to the incoming PF, will be those of the packet, but with their order reversed.

Forwarding packets

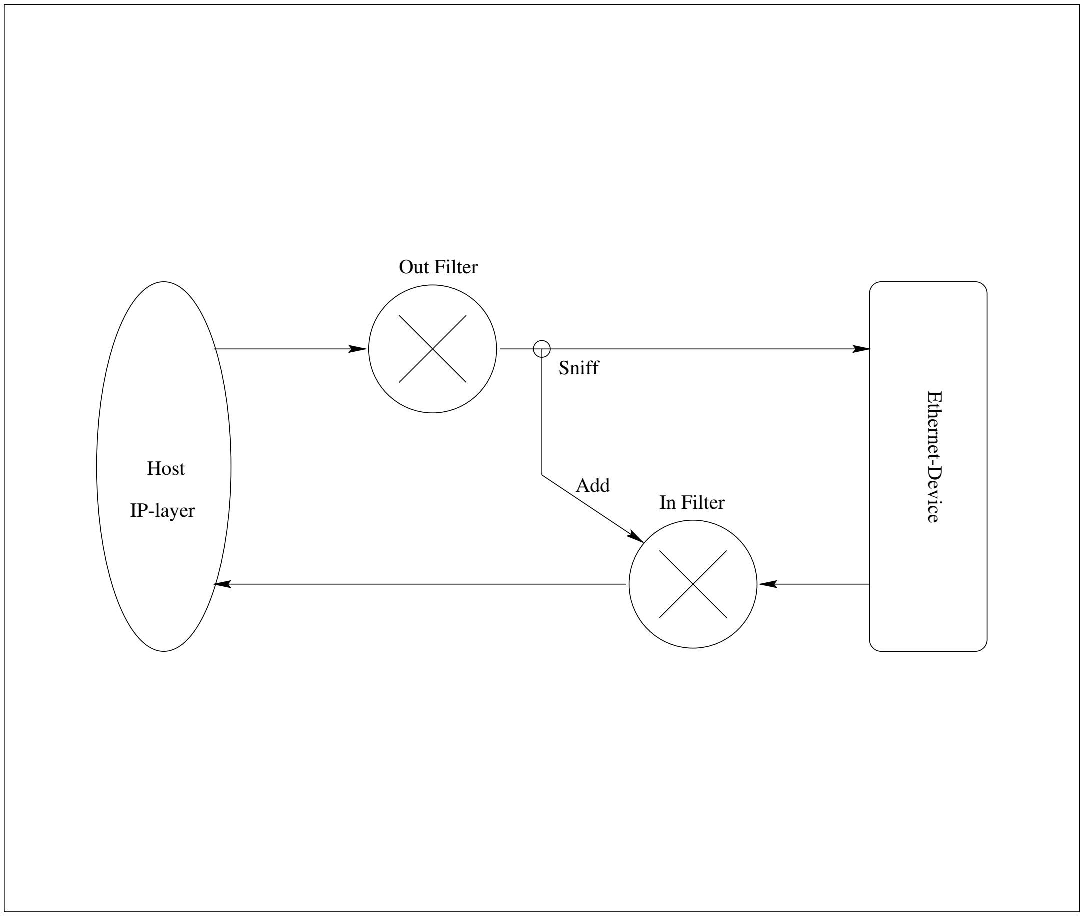

Figure 2.2: DPF processing of outgoing packets.

Figure 2.3 shows how the DPF handles packets that are to be forwarded to another ethernet-device. The DPF has a lot more to do when packets need to be forwarded. The packets have to pass the three existing PF chains in order to get out. The DPF thus has to add rules into the same PF chains, namely the Forwarding, Outgoing, and Incoming.

Again, the administrator may want to control what protocols and addresses are allowed by adding restricting rules in the beginning of the appropriate PF chain. Remember that the DPF always appends to the PF chains, and thus the existing rules take precedence.

As in Section 2.5.2 the corresponding timer is updated as long as packets are forwarded or received and the temporal rules will not be removed until the corresponding timer reaches zero.

2.6 DPF time-out values

The time-out values are by default set to the following number of ticks:

```
#define DYNAMIC_EXPIRE_TCP    15*60*HZ
#define DYNAMIC_EXPIRE_TCP_FIN  2*60*HZ
```

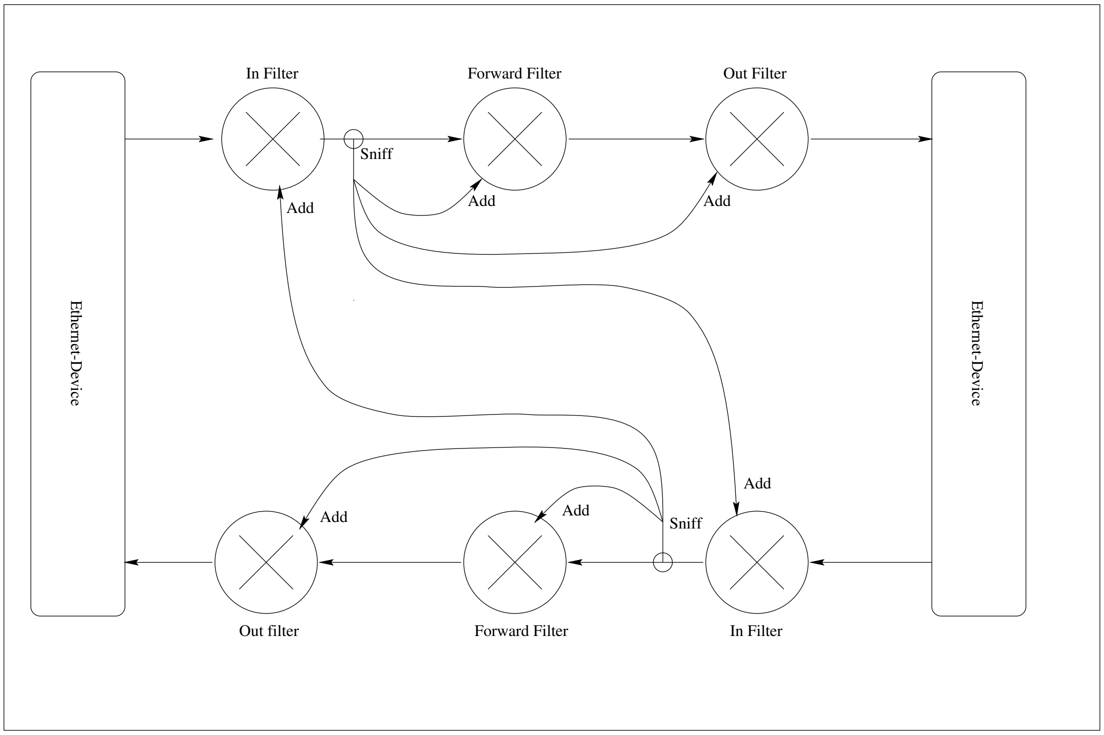

Figure 2.3: DPF processing of forwarded packets

```
#define DYNAMIC_EXPIRE_UDP       5*60*HZ
#define DYNAMIC_EXPIRE_ICMP      2*60*HZ
```

As Linux can be run on i386, Sparc and Alpha processors, the HZ definition is different from architecture to architecture, but it is defined to the number of ticks of a hardware interrupt per second. In case of the PC architecture (which uses the i386 processor architecture) the general purpose 8254 timer/counter chip is set to generate interrupts at 100Hz. The timer chip generates a hardware interrupt request (IRQ) every 10ms. The PC architecture had from the beginning only eight hardware interrupts, IRQ[0-7], but as more were needed another timer/counter chip was added. The other chip is cascade coupled to IRQ2, and thus an IRQ2 signals that some of the IRQ[8-15] lines are active. The IRQ lines have priority were the lowest number is the highest priority. The timer generates IRQ0.

The time-out for TCP is set to 15 minutes, except when both parts have closed the connection where the time-out is set to 2 minutes. UDP has 5 minutes before it times out, and ICMP has 2 minutes.

The time-out values are more or less taken out of thin air, except the TCP time-out value. The TCP time-out is determined by the TCP transaction specification which states that an endpoint should remain open in the state called TIME_WAIT for the time of 2*MSL, where MSL is a system constant. According to [5], MSL is supposed to be 2 minutes, but common values are 30 seconds, 1 minute or 2 minutes. Linux uses 1 minute.

Actually most values of the time-out would have worked if the two hosts have a reliable and fast connection, where the incoming packet arrives before the rule times out. If the rule times out, the TCP layer will assume that the packet was lost. If the DPF host was the one that was transmitting, the TCP layer will simply retransmit and a new rule will be added. If the remote host responds faster this time (or the packet is not lost this time), the data will eventually be transferred. If, on the other hand, the DPF host is receiving data, all hope is lost as it cannot receive anything since the rule has been removed from the incoming PF.

By the above reasoning it is thus important that not too short time-out values are chosen. This is why TCP has a longer time-out value than the other protocols, as it often is used for long distance ‘calls'. As many of the tests are done by typing commands to check the status of the PFs and DPF, a short time-out would also mean that a rule could have been placed on the DPF host, but when the PFs and DPF are queried the rule has already been removed.

The default time-out values can be changed by running a hacked[1] version of the *ipfwadm* program. The modified ipfwadm can also set additional debugging information which is needed when the performance of the DPF is measured.

2.7 Tools

To be able to test and to configure the setup, the set of software tools shown in Table 6 was used.

Table 4: Useful Tool

Tool	Where to find
ifconfig	netbase-package in Debian
ipfwadm	netbase-package in Debian
nc†	netcat-package in Debian
netperf	netperf web page[33]
netstat	netbase-package in Debian
ping	netbase-package in Debian
route	netbase-package in Debian
rpcinfo	netstd-package in Debian
tcpdump	tcpdump-package in Debian
traceroute	netstd-package in Debian
trafshow†	netdiag-package in Debian

The tools marked with † are unfortunately currently in the unstable[2] area of the Debian distribu-

1. Hacking is not considered to be a bad word in the Linux community, however cracking is.

tion. To use them, one must fetch and install them manually without the ease of the Debian package maintenance tool.

A brief explanation follows:

- *ifconfig*. Configure a network interface. Also shows the current settings of interface and statistics counters of the sent and received packets. The ifconfig is used for system maintenance and initial setup of a system. The following is a sample of its output:

```
gandalf# ifconfig
lo Link encap:Local Loopback
   inet addr:127.0.0.1  Bcast:127.255.255.255  Mask:255.0.0.0
   UP BROADCAST LOOPBACK RUNNING  MTU:3584  Metric:1
   RX packets:20 errors:0 dropped:0 overruns:0
   TX packets:20 errors:0 dropped:0 overuns:0

eth0Link encap:10Mbps Ethernet  HWaddr 08:00:09:CC:6B:3D
   inet addr:128.198.4.145  Bcast:128.198.4.255  Mask:255.255.255.0
   UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
   RX packets:258937 errors:37 dropped:0 overruns:0
   TX packets:405 errors:0 dropped:0 overruns:0
   Interrupt:11 Base address:0xf8e0 DMA chan:4

eth1Link encap:10Mbps Ethernet  HWaddr 08:00:09:7A:A3:90
   inet addr:128.198.9.118  Bcast:128.198.9.255  Mask:255.255.255.0
   UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
   RX packets:265828 errors:0 dropped:0 overruns:0
   TX packets:6872 errors:0 dropped:0 overruns:0
   Interrupt:9 Base address:0xfc00
```

- *ipfwadm*. Configure and display the rules associated with the Linux PFs. An example of using the ipfwadm program is shown in Section 2.1.2, and other examples are shown in Section 3.2.1. A modified version is used to configure some DPF specifics such as the time-out values, and additional features needed when evaluating the performance of the DPF, which is discussed in Section 4.
- *nc†*. Very useful tool for just sending and receiving messages without having to write special programs. Before the netperf benchmark program was found, nc was used to measure the performance of the DPF.
- *netperf*. Used to measure the performance of the PFs. Netperf can do the following tests:
  - Bulk data transfer measurements with both TCP and UDP.
  - Request-response measurements with the same protocols.

  See Section 4 for the performance measurements using the netperf program. More information can be found on the netperf web page [33].
- *netstat*. Shows various network statistics, for instance the kernel routing table. The routing table determines how packets are routed to other hosts. There are various examples of output from netstat in Section 3.1.

---

2. The Debian has a stable and an unstable distribution. Users of the unstable distribution are used as guinea pigs to find possible bugs.

- *ping*. Sends a mandatory ICMP ‘echo request’ message to a host to determine if it is alive. On some versions of ping, you have to add a verbose switch to get more useful information about how many packets were sent and how many replies it got, otherwise the ping program may only respond with either ‘host x is alive' or ‘host x does not respond’. Ping is mainly a diagnostics tool.

  Example of output:

```
gandalf# ping bilbo
PING bilbo.uccs.edu (128.198.9.116): 56 data bytes
64 bytes from 128.198.9.116: icmp_seq=0 ttl=64 time=0.5 ms
64 bytes from 128.198.9.116: icmp_seq=1 ttl=64 time=0.3 ms
64 bytes from 128.198.9.116: icmp_seq=2 ttl=64 time=0.3 ms
64 bytes from 128.198.9.116: icmp_seq=3 ttl=64 time=0.3 ms

--- bilbo.uccs.edu ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.3/0.3/0.5 ms
```

- *route* Configure the kernel routing table. The routing table determines the route packets take to reach their destinations. A specific entry to a host or network is used before any general rules to any default routers. The route command is most commonly run at boot to setup the default rules, but may be run at any time when changes are needed.

  Example of a network initialization script:

```
#!/bin/sh
ifconfig lo 127.0.0.1
route add -net 127.0.0.0
IPADDR=128.198.4.145
NETMASK=255.255.255.0
NETWORK=128.198.4.0
BROADCAST=128.198.4.255
GATEWAY=128.198.4.254
ifconfig eth0 ${IPADDR} netmask ${NETMASK} broadcast ${BROADCAST}
route add -net ${NETWORK}
route add default gw ${GATEWAY} metric 1
```

- *rpcinfo* Shows information about the RPC services available on a host. Discussed in Section 3.4 when NFS is used from a host behind the firewall.
- *tcpdump* Display the packets on the specified network device. Has a lot of options and a language to select what is to be displayed from the network device. Section 3.2.2 explains more about the output from the tcpdump program.
- *traceroute* Show what route the packets take to get to a particular host. The program works by increasing the TTL counter from 1 until a packet reaches the destination. As the routers between the originator and the destination will respond with a ‘host unreachable’ ICMP message if the TTL counter reaches zero, the program can show how the packets have been routed so far. The traceroute is used for diagnostic purposes.

  Example of output:

```
gandalf> traceroute sanluis
traceroute to sanluis.uccs.edu (128.198.2.62), 30 hops max, 40 byte packets
 1  hubeas.uccs.edu (128.198.7.254)  1.557 ms  1.339 ms  1.335 ms
 2  sanluis.uccs.edu (128.198.2.62)  1.714 ms  1.536 ms  1.591 ms
```

- *trafshow*† Shows the traffic of the selected network device, with the source and destination, the transport protocol and the total number of bytes sent so far.

  Example of output:

```
gandalf                                             0 days 00 hrs 09 min 17 sec
udp  ECEDHCP192.uccs.edu netbio uccs-brdcst.uccs.ed netbio        235
tcp  gandalf.uccs.edu    telnet annexp25.uccs.edu   1028        47231       8970
udp  zdomain.uccs.edu    netbio uccs-brdcst.uccs.ed netbio       8483
udp  hubeas.uccs.edu     route  255.255.255.255     route       10752
udp  ccnuccs.uccs.edu    domain gandalf.uccs.edu    1216          105         72
udp  ccnuccs.uccs.edu    domain gandalf.uccs.edu    1217          101         71
udp  ccnuccs.uccs.edu    domain gandalf.uccs.edu    1218          100         71
udp  ecepc33.uccs.edu    netbio uccs-brdcst.uccs.ed netbio        242
udp  mpc7.uccs.edu       ntp    gandalf.uccs.edu    ntp           684        684
udp  ecerez.uccs.edu     netbio uccs-brdcst.uccs.ed netbio         78
udp  128.198.9.115       netbio 128.198.9.255       netbio       1170
udp  ccnuccs.uccs.edu    domain gandalf.uccs.edu    1213          100         71
udp  ecerez.uccs.edu     netbio uccs-brdcst.uccs.ed netbio        241
udp  ccnuccs.uccs.edu    domain gandalf.uccs.edu    1214          156         72
udp  ccnuccs.uccs.edu    domain gandalf.uccs.edu    1215          156         72
udp  ECEDHCP164.uccs.edu netbio uccs-brdcst.uccs.ed netbio        964
udp  128.198.9.115       netbio 128.198.9.255       netbio        741
udp  ecepc40.uccs.edu    netbio uccs-brdcst.uccs.ed netbio        243
udp  128.198.4.83        netbio uccs-brdcst.uccs.ed netbio        240
udp  255.255.255.255     bootps 0.0.0.0             bootpc                  3280
udp  ecepc53.uccs.edu    netbio uccs-brdcst.uccs.ed netbio        251
udp  ecepc4.uccs.edu     netbio uccs-brdcst.uccs.ed netbio        250
971K total, 713K bad, 0K frag               54K tcp, 203K udp, 0K icmp, 0K unkn
```

3. Test of the DPF

In this Section the testing environment used for verifying the DPF functions will be discussed. Several network applications are tried, including Telnet, FTP and NFS. The Telnet sessions are first tried from the DPF host itself, then it is run from a host on the intranet. The other applications are only run from hosts on the intranet.

3.1 Physical setup

The testing environment is shown in Figure 2.1. Flagdnag and Odorf are on the same 10Mbps ethernet, 128.198.4.0. Gandalf and Bilbo are on the 100Mbps HP net, 128.198.9.0. One possible setback of the setup is that both hubs are connected to the UCCS backbone. See Section 1.3.4 for a discussion of hubs, segments, subnets and the UCCS backbone. Since the hubs are implicitly coupled, it means that if Bilbo and Frodo are communicating, their packets may travel the intended way through Gandalf, or through the UCCS backbone.

The routing tables on all computers have therefore been carefully setup to use the Gandalf as a router.

The routing table setup can be tested by making Gandalf to refuse any forwarding between its two ethernet cards. If the default PF policy is to accept, it is possible to do network activity between the two computers, and if the PF policy is set to reject/deny it is not. The test proves that packets can only travel the intended way, and not by the UCCS backbone.

The following IP addresses corresponds to these hosts:

- 128.198.9.116 = Bilbo on the 100Mbps ethernet.
- 128.198.9.118 = Gandalf on the 100Mbps ethernet.
- 128.198.4.144 = Odorf on the 10Mbps ethernet.
- 128.198.4.145 = Fladnag on the 10Mbps ethernet.

Note that Gandalf and Fladnag are actually the same host but on different subnets. The Odorf host is actually called Frodo, but like Gandalf the name is reversed when accessed through the 10Mbps ethernet device. Independent of which ethernet device they are accessed from, both Frodo and Gandalf state their real name on the command prompt.

3.1.1 Routing tables

The following routing table is the result of running the *netstat* program on Bilbo. The -rn switch sets the program to produce the routing table without trying to translate addresses to names.

```
bilbo# netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
128.198.4.144   128.198.9.118   255.255.255.255 UGH    1500 0          0 eth1
128.198.9.0     0.0.0.0         255.255.255.0   U      1500 0          0 eth1
127.0.0.0       0.0.0.0         255.0.0.0       U      3584 0          0 lo
0.0.0.0         128.198.9.254   0.0.0.0         UG     1500 0          0 eth1
```

Bilbo is on the 100Mbps ethernet. The first line is a specific rule to reach Odorf. The second line is a route to the .9 subnet. The third line is a rule to reach the loopback (lo) interface which is used for internal routing. A specific rule in the routing table is used before a generic one. This means that the rule in the first line of a route to Odorf via Gandalf will take precedence over using the default router of 128.198.9.254.

The other fields in the output is the Genmask which specifies the network mask used to find routing entries. The Flags field usually contains U, G or H. U means that the routing table entry is Up, and the entry may be used. G means that the routing entry uses a Gateway (router). Section 1.3.4 explains why the term router should be used instead of gateway. Finally the H means that the target is a Host. The MSS field shows the Maximum Segment Size of the device. Note the larger size of the loopback interface. A MSS of 1500 is suspicious, and we assume that it really means MTU instead. MSS, MTU and Window are discussed in Section 3.2.1. The Window field is the window size to use.

Zero means the default size, that is to let the TCP layer figure out what window size to use. The irtt field is the initial round trip time (rtt) estimation, which is used to avoid congestion of a network. A large rtt may fool the TCP layer to make unnecessary retransmissions, as the TCP layer thinks the packets are lost when they simply take too long time to arrive. Zero means the default, and thus the TCP layer calculates a rtt based on the timing of the packets sent/received.

The following routing table is used on Odorf:

```
frodo# netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
128.198.9.116   128.198.4.145   255.255.255.255 UGH    1500 0          0 eth0
128.198.4.0     0.0.0.0         255.255.255.0   U      1500 0          0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U      3584 0          0 lo
0.0.0.0         128.198.4.254   0.0.0.0         UG     1500 0          0 eth0
```

Odorf is on the 10Mbps ethernet. To reach Bilbo via Gandalf, a rule similar to the one used on Bilbo, has been added to override the default router.

The following routing table is used on Gandalf:

```
gandalf# netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
128.198.4.0     0.0.0.0         255.255.255.0   U      1500 0          0 eth0
128.198.9.0     0.0.0.0         255.255.255.0   U      1500 0          0 eth1
127.0.0.0       0.0.0.0         255.0.0.0       U      3584 0          0 lo
0.0.0.0         128.198.9.254   0.0.0.0         UG     1500 0          0
```

Gandalf is connected to both ethernets, and a host on the .4 subnet will be accessed through the eth0 device, and a host on the .9 subnet will be accessed through the eth1 device. If any routing has to be done, the kernel will figure out which device to use.

Any other destination will be reached through the default router of 128.198.9.254.

3.2 Test of Telnet

3.2.1 Telnet from DPF host to internet

This is a typical case of sending packets which is described in Section 2.5.2. The telnet protocol is commonly used to login to remote hosts and should thus be supported by a firewall. Telnet uses the TCP protocol, and no special considerations has to be done by the DPF.

The test of Telnet is done by setting all default policies to reject, and then allowing only outgoing connections from Gandalf.

This is done by running the following script called *setfirewall*.

```
#!/bin/sh
```

```
# Set default policy to reject on the PFs
ipfwadm -I -p reject
ipfwadm -O -p reject
ipfwadm -F -p reject

# Flush PF chains
ipfwadm -I -f
ipfwadm -O -f
ipfwadm -F -f

# Allow Gandalf to send
ipfwadm -O -a accept -S gandalf -D 0/0
ipfwadm -O -a accept -S fladnag -D 0/0

# Now list the contents of the PFs
ipfwadm -I -l
ipfwadm -F -l
ipfwadm -O -l
```

When running the script the last three lines generate the following:

```
gandalf# ./setfirewall
IP firewall input rules, default policy: reject
IP firewall forward rules, default policy: reject
IP firewall output rules, default policy: reject
type  prot source              destination          ports
acc   all  gandalf.uccs.edu    anywhere             n/a
acc   all  fladnag.uccs.edu    anywhere             n/a
```

This is the starting point of the test, nothing is allowed to get in, and only Gandalf is able to send on both ethernet devices.

Now do a telnet to odorf:

```
gandalf> telnet odorf
Trying 128.198.4.144...
Connected to odorf.uccs.edu.
Escape character is '^]'.
Debian Linux 1.1 Copyright (C) 1993-1996 Debian Association, Inc. and others

frodo login: jhjulkun
Password:
Last login: Mon Aug 26 11:09:00 from fladnag.uccs.edu Linux frodo 2.0.6 #1 Wed
Aug 14 06:28:10 MDT 1996 i586
Copyright (C) 1993-1996 Debian Association, Inc. and others

frodo>
```

Let the connection stay open and check the rules in the PF chains:

```
gandalf# ipfwadm -I -l
IP firewall input rules, default policy: reject
type  prot source              destination          ports
acc   tcp  odorf.uccs.edu      fladnag.uccs.edu     telnet -> 1042
```

```
gandalf# cat /proc/net/ip_dynamic
Pro Firewall Source IP:Port        Dest IP:Port          Expires Hash
TCP Outgoing 128.198.4.145:1042    128.198.4.144:23        87532  133
```

The ipfwadm program tells us that the incoming PF chain now accepts incoming packets from odorf. By checking the /proc/net/ip_dynamic 'file' which actually invokes a function in the DPF code, the status of the DPF is printed. The printout from the DPF code is printed in dotted decimal only, it is not converted to host names as the output from the ipfwadm program.

Note the expiration timer, it tells us that the rule will be in effect for 87532 'ticks'. The timer is decreased by 6000 ticks each minute, thus the rule will be in effect for 87532/6000≈15min. The timer is reset to 90000 every time a packet goes by. The difference of 90000 and 87532 is the time it takes to type the commands to list the contents of the incoming PF and list the /proc/net/ip_dynamic entries.

Now, lets close the telnet session:

```
frodo> exit
logout
Connection closed by foreign host.
gandalf>
```

When both sides have closed the connection, the timer is set to 2 minutes, which is 12000 ticks.

Let's confirm this:

```
gandalf# cat /proc/net/ip_dynamic
Pro Firewall Source IP:Port        Dest IP:Port          Expires Hash
TCP Outgoing 128.198.4.145:1042    128.198.4.144:23        11636  133
```

The rule is thus in effect for another two minutes after the session has been closed. The time-out is required by the TCP state diagram. This is to allow a final ACK to be resent in case the previous ACK was lost (the remote host will time-out and send a FIN message to remind the host about this). Unless the host is infected by some malicious users who checks all connections, it is unlikely that this might be a security hole, as TCP never allows the same endpoint (proto:IP:port) to be used within 2*MSL. See Section 2.6 for common values of MSL.

3.2.2 TCP dump of a telnet session

Even though telnet seems to work, a tcpdump listing of what happens can be used to verify the correct behavior of the DPF.

The tcpdump program has been tremendously useful for tracing what actually happens on the ethernet. To be able to run tcpdump one has to be root.

The tcpdump program uses a very condensed output to cram in as much information as possible in a limited space. The output lines has the following syntax:

```
timestamp source > dest Flags [other info]
```

The flags shown in Table 3.1 means the following:

- URG = Urgent data.
- ACK = The Acknowledge number is valid.

- PSH = The receiver should Push the data to the receiving application as quickly as possible. Not really used, usually set.
- SYN = Synchronize sequence numbers to initiate a connection.
- FIN = Finished sending data.

Table 5: Tcpdump flags

Tcpdump flag	Acronym	Meaning
A	ACK	Acknowledge
F	FIN	Finish
P	PSH	Push
R	RST	Reset
S	SYN	Synchronize
U	URG	Urgent
.		No flags set

The following lines is the start of a telnet session:

```
11:09:58.607781 fladnag.uccs.edu.1042 > odorf.uccs.edu.telnet: S 185302925:185302925(0) win 512 <mss 1460>
11:09:58.607781 odorf.uccs.edu.telnet > fladnag.uccs.edu.1042: S 2527100738:2527100738(0) ack 185302926 win 15360
<mss 1460>
```

The sequence starts by Gandalf sending a synchronization message to Odorf. Explanation:

- 185302925:185302925(0) means that the starting sequence is the first number, and the implied ending sequence is the second. (0) means that there are zero bytes of data.
- win 512 means that the window size of the receiver is 512 bytes. The window size is the maximum number of bytes which the sender is willing to accept currently. If a large window is offered, there may be more packets in transition and a higher bandwidth is achieved. In this case the window size is tiny, on Linux it is usually 16KB. The window size is different from OS to OS, for example the 4.2BSD has 2048 bytes, 4.3BSD has 4096 bytes. Solaris and AIX uses 8192 or 16384 bytes.
- <mss 1460> means that the Maximum Segment Size (MSS) is 1460 bytes. The ethernet frame (MTU) is 1500 bytes, and with an IP and TCP header of 20 bytes each, this leaves 1460 bytes for TCP payload.

Odorf returns its synchronization number. It also acknowledges the start of the next sequence number, 185302926, and sends its window size and MSS. From here on, the tcpdump program will only show the relative sequence number to make it easier to read, as is indicated by the ‘1’ in the next tcpdump line.

Usually another field is present, the Type Of Service field. The TOS field is usually not used, and it seems like Linux does not use it.

```
11:09:58.607781 fladnag.uccs.edu.1042 > odorf.uccs.edu.telnet: . ack 1 win 31744 (DF)
```

Gandalf now ACKs the synchronization number from Odorf. The data transfer can now begin. This handshake before any data transfer is called the **three way handshake**. Lets summarize the steps:

1. The client sends a TCP message with the SYN flag and its synchronization number.
2. The server responds with a SYN and its own synchronization number. The server also ACKs the synchronization number received from the client by increasing the number by one.
3. The client ACKs the synchronization number received from the server by increasing it.

The following data is part of the login sequence. The trailing (DF) means that the Don't Fragment flag is set, and the packet must not be fragmented on its way to the destination. If it is not possible for a router on the way to keep the packet unfragmented, it will send the ICMP 'host unreachable' message to the originating host. The host has then to step down to a smaller payload, and retransmit.

```
11:09:58.667781 fladnag.uccs.edu.1042 > odorf.uccs.edu.telnet: P 1:28(27) ack 1 win 31744 (DF)
11:09:58.687782 odorf.uccs.edu.telnet > fladnag.uccs.edu.1042: . ack 28 win 15360
```

The above two lines show that data is flowing from Fladnag to Odorf, and we will skip the rest of the trace until the connection is closed. The 'P 1:28(27)' may need some explanation. The P is the Push flag mentioned above. The 1:28(27) means that the sequence number starts at 1, and the expected ACK sequence number should be 28. There are 27 bytes of data.

```
11:10:36.747974 odorf.uccs.edu.telnet > fladnag.uccs.edu.1042: F 721:721(0) ack 141 win 15360
11:10:36.747974 fladnag.uccs.edu.1042 > odorf.uccs.edu.telnet: . ack 722 win 31744 (DF)
11:10:36.787974 fladnag.uccs.edu.1042 > odorf.uccs.edu.telnet: F 141:141(0) ack 722 win 31744
11:10:36.787974 odorf.uccs.edu.telnet > fladnag.uccs.edu.1042: . ack 142 win 15360
```

As it is Odorf that closes the connection, it will send the FIN flag. Gandalf sees the flag, returns an ACK and then sends a FIN itself. Both the client and the server must ACK the FIN message before the session is terminated.

3.2.3 Telnet from an intranet host behind DPF to Internet

The DPF has to work more if the telnet occurs from one of the hosts behind the DPF host. Section 2.5.3 explains what the DPF has to do.

The same setup is used as in Section 3.2.1, with the exception that the hosts on the intranet has to be added to the incoming PF chain. This is because the packets have to pass the incoming PF before anything else is considered.

Adding hosts on the intranet to the incoming PF is done by:

```
gandalf# ipfwadm -I -a accept -S 128.198.9.0/24 -D 0/0
gandalf# ipfwadm -I -l
IP firewall input rules, default policy: reject
type  prot source              destination         ports
acc   all  localnet/24         anywhere            n/a
```

DPF

Note the -S 128.198.9.0/24 which means that all IP addresses which match 128.198.9.0 ANDed with 24 bits of ones (starting at the most significant bit) are allowed to pass the incoming PF. Bilbo has an IP address of 0x80c60974 which ANDed with 0xffffff00 is 0x80c60900. This matches the rules IP address of 0x80c60900.

If a telnet session is opened from Bilbo to Odorf, it will have the following effect on the PFs.

```
gandalf# ipfwadm -I -l
IP firewall input rules, default policy: reject
type  prot source               destination          ports
acc   all  localnet/24          anywhere             n/a
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       1030 -> telnet

gandalf# ipfwadm -F -l
IP firewall forward rules, default policy: reject
type  prot source               destination          ports
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       1030 -> telnet

gandalf# ipfwadm -O -l
IP firewall output rules, default policy: reject
type  prot source               destination          ports
acc   all  gandalf.uccs.edu     anywhere             n/a
acc   all  fladnag.uccs.edu     anywhere             n/a
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       1030 -> telnet
```

The last line of each PF listing are the specific rules added by the DPF for a telnet session from Bilbo to Odorf. The PF can be set to accept a bidirectional rule, which means that either Bilbo:port → Odorf:port is accepted or Odorf:port → Bilbo:port. However, we have not found out how to set ipfwadm to show this.

Let's check the status of the DPF:

```
gandalf# cat /proc/net/ip_dynamic
Pro Firewall Source IP:Port        Dest IP:Port          Expires Hash
TCP  Forward 128.198.9.116:1030    128.198.4.144:23        82418  116
```

As it is indicated below the expiration time is set to the default time-out value when the telnet session is closed.

```
gandalf# cat /proc/net/ip_dynamic
Pro Firewall Source IP:Port        Dest IP:Port          Expires Hash
TCP  Forward 128.198.9.116:1030    128.198.4.144:23        11398  116
```

3.3 Test of FTP

FTP uses the TCP protocol. It opens a connection to port 21 on the remote server, where commands are sent and responses are received. If data is to be received from the server, a PORT command is sent to the server with the port number to receive data on the client. The client then listens to the port, and waits for the server to start sending data from port 20.

DPF

This will not work with the DPF! This is because the port numbers used are different than the ones used for the command connection, and worst of all it is the server that initiates the data connection. As the DPF never is notified, it will not add a rule to the PFs and thus the data packets from the server will be dumped. The FTP client will eventually time-out without receiving any data.

In order to use FTP there are two solutions:

- Scan the client-server command data for the PORT keyword, and setup rules in the affected PFs for the port specified.
- Newer versions of the FTP client/server also understands the PASV command, which makes the server listen, and the client to initiate the connection.

If the second solution is used, the DPF does not need any modifications, but if older versions of FTP clients are used, then something must be done. We have chosen to implement the first approach to make the DPF work with all FTP clients.

The DPF thus scans for the PORT command in the control-stream to the remote server and installs appropriate rules. Figure 3.1 shows the actual PORT command sent to the host Odorf corresponding to the tcpdump of Section 3.3.1. In this scenario the data port on Bilbo is 40001. The PORT command is sent as six bytes with the IP address bytes first and then the port number as two bytes. Bilbo's IP address is 128.198.9.144, and the port is 40001, which in two byte format is 156*256+65. The PORT command is thus: PORT 128,198,9,144,156,65.

3.3.1 FTP from intranet host behind DPF to internet

The following is the starting setup:

```
gandalf# ipfwadm -I -l
IP firewall input rules, default policy: reject
type  prot source               destination          ports
acc   all  localnet/24          anywhere             n/a

gandalf# ipfwadm -F -l
IP firewall forward rules, default policy: reject

gandalf# ipfwadm -O -l
IP firewall output rules, default policy: reject
type  prot source               destination          ports
acc   all  gandalf.uccs.edu     anywhere             n/a
acc   all  fladnag.uccs.edu     anywhere             n/a
```

Now, if a FTP session is started from Bilbo to Odorf, and after the user has typed DIR, the following rules are in effect:

```
gandalf# ipfwadm -I -l
IP firewall input rules, default policy: reject
type  prot source               destination          ports
acc   all  localnet/24          anywhere             n/a
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       1031 -> ftp
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       40001 -> ftp-data
```

```
gandalf# ipfwadm -F -l
IP firewall forward rules, default policy: reject
type  prot source               destination          ports
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       1031 -> ftp
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       40001 -> ftp-data
```

```
gandalf# ipfwadm -O -l
IP firewall output rules, default policy: reject
type  prot source               destination          ports
acc   all  gandalf.uccs.edu     anywhere             n/a
acc   all  fladnag.uccs.edu     anywhere             n/a
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       1031 -> ftp
acc   tcp  bilbo.uccs.edu       odorf.uccs.edu       40001 -> ftp-data
```

The status so far is visualized in Figure 3.1, where the client has sent the PORT command to the

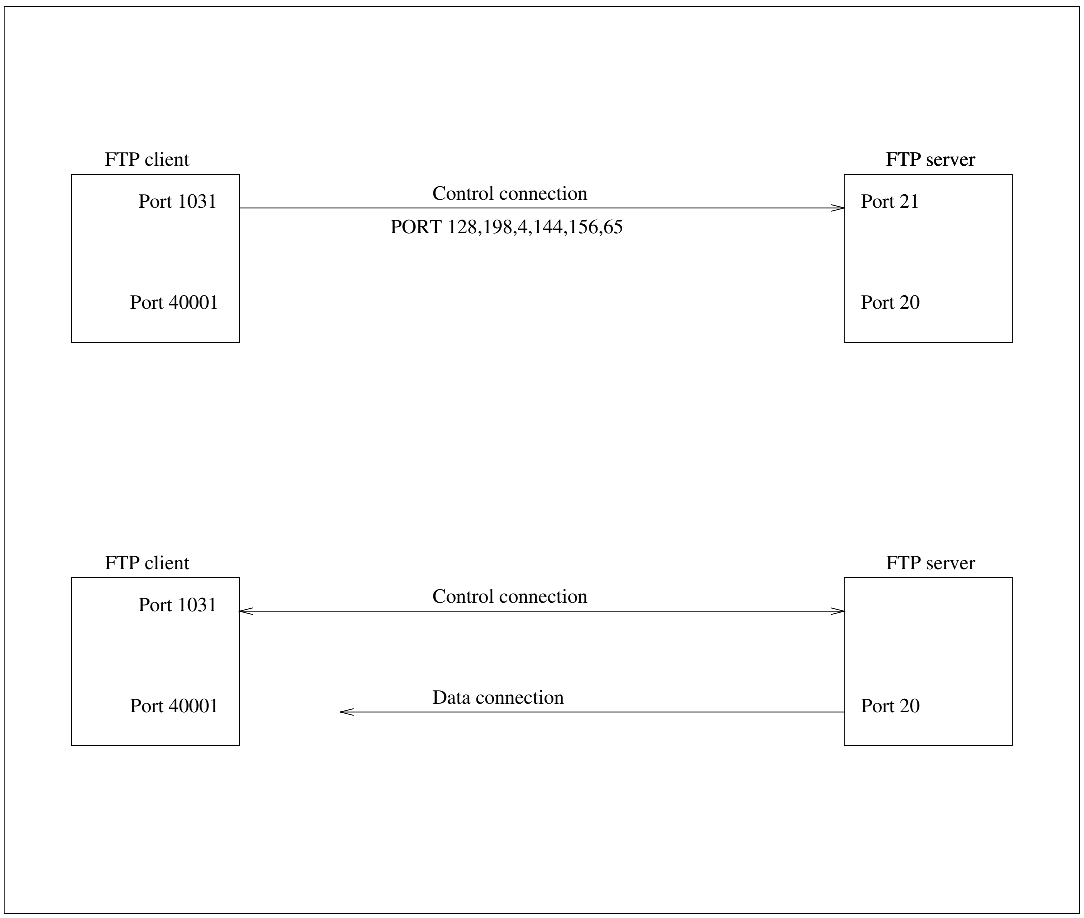

Figure 3.1: FTP port command

server, and the DPF has intercepted this message and allows the server to negotiate a connection to the client.

Let's check the status of the DPF:

```
gandalf# cat /proc/net/ip_dynamic
Pro Firewall Source IP:Port        Dest IP:Port          Expires Hash
TCP  Forward 128.198.9.116:40001   128.198.4.144:20         1399   51
TCP  Forward 128.198.9.116:1031    128.198.4.144:21        79398  117
```

As the data-connection is closed when the data is transferred, the expiration time is set to the default time for a closing TCP connection.

When the session is closed, the expiration time should have decreased:

```
gandalf# cat /proc/net/ip_dynamic
Pro Firewall Source IP:Port        Dest IP:Port          Expires Hash
TCP  Forward 128.198.9.116:1031    128.198.4.144:21        11217  117
```

3.3.2 TCP dump of FTP transfer

In the following trace the initial handshaking already is done, and the trace shows what happens after the user has sent the command DIR to the remote server. The actual PORT command cannot be seen as we have not enabled the ‘dump data' switch of the tcpdump which prints the data in the packets[1].

```
12:13:08.677104 odorf.uccs.edu.ftp-data > bilbo.uccs.edu.40001: S 3245975086:3245975086(0) win 512 <mss 1460> [tos 0x8]
12:13:08.677104 bilbo.uccs.edu.40001 > odorf.uccs.edu.ftp-data: S 256500895:256500895(0) ack 3245975087 win 15360 <mss 1460>
12:13:08.677104 odorf.uccs.edu.ftp-data > bilbo.uccs.edu.40001: . ack 1 win 15360 [tos 0x8]
12:13:08.677104 odorf.uccs.edu.ftp > bilbo.uccs.edu.1031: P 204:257(53) ack 70 win 15360 (DF) [tos 0x10]
12:13:08.697105 bilbo.uccs.edu.1031 > odorf.uccs.edu.ftp: . ack 257 win 15360 [tos 0x10]
12:13:08.727105 odorf.uccs.edu.ftp-data > bilbo.uccs.edu.40001: P 1:1025(1024) ack 1 win 15360 (DF) [tos 0x8]
12:13:08.727105 odorf.uccs.edu.ftp > bilbo.uccs.edu.1031: P 257:281(24) ack 70 win 15360 (DF) [tos 0x10]
12:13:08.747105 bilbo.uccs.edu.1031 > odorf.uccs.edu.ftp: . ack 281 win 15360 [tos 0x10]
12:13:08.747105 bilbo.uccs.edu.40001 > odorf.uccs.edu.ftp-data: . ack 1025 win 15360 [tos 0x8]
12:13:08.747105 odorf.uccs.edu.ftp-data > bilbo.uccs.edu.40001: P 1025:1265(240) ack 1 win 15360 (DF) [tos 0x8]
12:13:08.747105 odorf.uccs.edu.ftp-data > bilbo.uccs.edu.40001: F 1265:1265(0) ack 1 win 15360 [tos 0x8]
12:13:08.747105 bilbo.uccs.edu.40001 > odorf.uccs.edu.ftp-data: . ack 1266 win 15360 [tos 0x8]
12:13:08.757105 bilbo.uccs.edu.40001 > odorf.uccs.edu.ftp-data: F 1:1(0) ack 1266 win 15360 [tos 0x8]
12:13:08.757105 odorf.uccs.edu.ftp-data > bilbo.uccs.edu.40001: . ack 2 win 15360 [tos 0x8]
```

The trace shows the three-way-handshake between Odorf:ftp-data(20) and Bilbo:40001 which is the ephemeral port number chosen by the ftp client. If the PORT command would not have been intercepted by the DPF, the three-way-handshake would not have been successful as the incoming PF never would have allowed the SYN message to pass to Bilbo.

After the three-way-handshake the DIR data is sent to Bilbo, and the last lines show the closing sequence.

The new field in the tcpdump-dump output is the Type Of Service (TOS), whose bit-mapped values are shown in Table 8. By looking at the TOS bits above, one can see that the control

1. The tcpdump included in the Debian distribution seems to be compiled without this switch, as by enabling it, a huge security hole is made as passwords to remote logins can be displayed.

connection requests a TOS of 0x10, minimize delay. The data connection on the other hand requests a TOS of 0x08, maximize throughput.

Table 6: TOS bits

Bit mask	Meaning
0x02	Minimize-monetary cost
0x04	Maximize-reliability
0x08	Maximize-throughput
0x10	Minimize-delay

3.3.3 TCP dump of passive FTP session

Let's see what happens when a passive FTP session is used. The synchronization sequence was omitted and only the difference to the usual behavior was shown. The sequence shows that it is Bilbo (the client) that starts the three-way-handshake. This allows the DPF to append a rule in the incoming PF, and the data resulting from the DIR command is sent to Bilbo. The command port is 1034 on Bilbo, and the data port is 1035. Note that the data is not transferred from the usual data port of 20 from the server, this time an ephemeral port of 40000 is chosen on the server.

```
12:41:16.146458 bilbo.uccs.edu.1035 > odorf.uccs.edu.40000: S 764469348:764469348(0) win 512 <mss 1460>
12:41:16.146458 odorf.uccs.edu.40000 > bilbo.uccs.edu.1035: S 2298515571:2298515571(0) ack 764469349 win 15360 <mss 
1460>
12:41:16.146458 bilbo.uccs.edu.1035 > odorf.uccs.edu.40000: . ack 1 win 15360
12:41:16.146458 bilbo.uccs.edu.1034 > odorf.uccs.edu.ftp: P 43:49(6) ack 224 win 15360 (DF) [tos 0x10]
12:41:16.146458 odorf.uccs.edu.ftp > bilbo.uccs.edu.1034: P 224:277(53) ack 49 win 15360 (DF) [tos 0x10]
12:41:16.166458 bilbo.uccs.edu.1034 > odorf.uccs.edu.ftp: . ack 277 win 15360 [tos 0x10]
12:41:16.186458 odorf.uccs.edu.40000 > bilbo.uccs.edu.1035: P 1:1025(1024) ack 1 win 15360 (DF) [tos 0x10]
12:41:16.186458 odorf.uccs.edu.ftp > bilbo.uccs.edu.1034: P 277:301(24) ack 49 win 15360 (DF) [tos 0x10]
12:41:16.186458 odorf.uccs.edu.40000 > bilbo.uccs.edu.1035: P 1025:1265(240) ack 1 win 15360 (DF) [tos 0x10]
12:41:16.186458 odorf.uccs.edu.40000 > bilbo.uccs.edu.1035: F 1265:1265(0) ack 1 win 15360 [tos 0x10]
12:41:16.186458 bilbo.uccs.edu.1035 > odorf.uccs.edu.40000: . ack 1266 win 15360 [tos 0x8]
12:41:16.206458 bilbo.uccs.edu.1034 > odorf.uccs.edu.ftp: . ack 301 win 15360 [tos 0x10]
12:41:16.206458 bilbo.uccs.edu.1035 > odorf.uccs.edu.40000: F 1:1(0) ack 1266 win 15360 [tos 0x8]
```

4. Performance

This Section will discuss the performance of the DPF. To get some idea of how much overhead each additional layer of code adds, the performance has first been tested on a kernel without packet filters, then a kernel with packet filters, and finally a kernel with DPF (and thus PF as well).

Since the DPF uses the packet filters of the Linux kernel, the performance loss of the DPF compared to the PF is directly related to the added overhead of the DPF. The PF in turn also adds overhead which can be measured by comparing the numbers of the non packet filter kernel.

The results presented were achieved from using three dedicated research computers equipped with a Pentium 100MHz processor and 16MB memory. All computers have two ethernet cards, one 10Mbps and one 100Mbps, but only Gandalf uses both. Gandalf is the host that is benchmarked.

The tests were done using an unmodified 2.0.6 kernel without packet filtering, an unmodified kernel of the same version with packet filtering, and same kernel with the DPF code added.

The two hosts involved in the tests were idle during the testing, using the same setup used as when testing, see Figure 2.1. The hosts are connected to each other through a hub. The hubs were also more or less idle, as the hosts involved in the performance test had their own hub. The hosts should thus get all the bandwidth they want. Remember that a hub is ‘smart' enough to realize that if two of its hosts are communicating, the traffic will not reach the UCCS backbone. What is even more important is that the same applies in reverse when packets are flowing on the UCCS backbone. The hub keeps track of the hardware address of the ethernet devices, see Section 3.5, and the ‘uplink’ hub will not send packets to its ‘downlink’ hub if there is no need. However, the 10Mpbs hub used in this case has a thin ethernet connection as well as the usual twisted pair connectors. The hub is used to connect the twisted pair ethernet segment to the thin ethernet segment. This makes it susceptible for collisions on the thin ethernet, as it cannot allow any of its hosts to send while there is traffic on the thin ethernet. This is indicated by the jaggedness of some of the 10Mbps performance results.

The bulk stream performance tests are presented first, then the request-response tests. Each test is run first on an unmodified kernel, then a PF kernel and last on a kernel with added DPF functionality.

4.1 PF chain loop

After tracing the kernel, Section 5.2.8, for processing of incoming packets, one can see that a large amount of time can be spent in a loop which tries to figure out if packets are to be allowed to pass. This is done by the following code:

```
int call_in_firewall(int pf, struct device *dev, void *phdr, void *arg) {
    struct firewall_ops *fw=firewall_chain[pf];

    while(fw!=NULL) {
        int rc=fw->fw_input(fw,pf,dev,phdr,arg);
        if(rc!=FW_SKIP) return rc;
        fw=fw->next;
    }
    return firewall_policy[pf];
}
```

There is a lot of detail in the code, but it is not very difficult to figure out that the code traverses a linked list where the code tries to match a rule with a packet. If nothing is matched, the default policy is returned.

4.2 Testing procedure

From the previous section it should be clear that it should be possible to measure the performance of the PFs by appending/inserting rules into one of the PF chains. The rules added are ‘dummy’ rules, they do not contain any useful information as the IP addresses inserted are not in use on UCCS. The idea is to keep the PF looking for matches in the chain without finding any, and by doing so spending a lot of time.

The ‘dummy’ rules are inserted into the incoming PF chain as this will work to measure the performance of both a PF kernel and a kernel with DPF.

To measure the worst possible scenario it is important that both the PF and the DPF spend time looking for a match when packets arrive. When something is sent from the DPF host, a DPF entry is created in an internal linked list of the DPF code. To make the code efficient, the new entry is always added at the head of the list. Thus if a packet is received from the same five-tuple entry recently created, it will be matched very quickly as the DPF only searches until a match is found. This means that the DPF host cannot be used to send anything or the performance benchmark will not reflect the worst case.

When receiving packets on the other hand, the DPF will still try to match the packet, but it will never create a DPF entry.

The netperf program[33] was used to measure the performance of the DPF. Netperf is a client/server type of benchmark program for doing various network performance measurements. The server can be run either as a daemon, or it can be started by the *inetd* daemon super server if the file /etc/services has been modified to include the netperf server. The netperf webpage[33] tells all the details about how to setup netperf.

The netperf tests were then run from either Bilbo or Odorf to measure the performance of the Gandalf host.

The more packets are received, the more time will be spent in the loop code. This is why very small packets are sent, as they should stress the system the most.

The packet sizes chosen are the following: 32, 64, 128, 256 and 1472 bytes. 1472 bytes of payload is the maximum size an UDP packet can contain to remain unfragmented. If the loop is the major bottleneck, a large packet size should be less affected by the number of rules the PF/DPF have to search for.

The maximum unfragmented payload of UDP can be calculated by taking the MTU size of 1500 bytes, and then decrease it by the 20 bytes of the IP header, and the 8 bytes of the UDP header.

The netperf program runs several times to achieve a statistical confidence level. The confidence level is set to 99%, and the program has to run at least two times and at most 10 times to achieve the confidence level.

An example of a part of the output from the netperf program is shown below. In this particular test the packet size is 32 bytes. The socket buffer length is 32KB.

```
------------------------------------------------------
Sat Sep 14 12:27:10 MDT 1996 Testing with the following command line:
./netperf -l 10 -H gandalf -i 10,2 -I 99,10 -t UDP_STREAM -- -m 32 -s 32768 -S 32768
UDP UNIDIRECTIONAL SEND TEST to gandalf : +/-5.0% @ 99% conf.
Socket  Message  Elapsed      Messages
Size    Size     Time         Okay Errors   Throughput
bytes   bytes    secs            #      #   10^6bits/sec

32768      32   9.99       132917      0       3.41
32768           9.99       131478              3.37
```

The second line of output is the calculated measurement.

When doing some trial runs of the benchmarks, and when many rules had been inserted in the incoming PF chain, the system got so stressed that the X clock stopped updating for several seconds. Obviously the loop took so much time that the system did not have time to do non-system tasks. When this happened the netperf program did not get the confidence it expected, and a note like this was produced:

```
------------------------------------------------------
Sat Sep 14 12:57:37 MDT 1996 Testing with the following command line:
./netperf -l 10 -H gandalf -i 10,2 -I 99,10 -t UDP_STREAM -- -m 32 -s 32768 -S 32768
UDP UNIDIRECTIONAL SEND TEST to gandalf : +/-5.0% @ 99% conf.
!!! WARNING
!!! Desired confidence was not achieved within the specified iterations.
!!! This implies that there was variability in the test environment that
!!! must be investigated before going further.
!!! Confidence intervals: Throughput      : 17.9%
!!!                       Local CPU util  :  0.0%
!!!                       Remote CPU util :  0.0%

Socket  Message  Elapsed      Messages
Size    Size     Time         Okay Errors   Throughput
bytes   bytes    secs            #      #   10^6bits/sec
32768      32   9.99       132529      0       3.40
32768           9.99        95419              2.44
```

Sometimes the throughput becomes zero or close to zero when the system is stressed too much. This can be seen on some of the benchmarks of the DPF. When the numbers presented in the following benchmarks were measured, the hosts were not running any other tasks while running the tests.

4.3 Bulk transfer

Bulk transfer is to send as much data as possible and to measure how much was received. The tests are done using UDP. There is either something strange with Linux or with the netperf program, but it seems like it is not possible to send variable size packets using TCP. If tracing the packets with tcpdump, the packets were sent with a smaller size than the usual 1480 bytes (MTU - IPheader), but the receiving host would only respond with the RST flag set, and nothing was transferred.

DPF

When running the tests, a number of rules has to be inserted in the incoming packet filter chain. This is done by the following script:

```
#!/usr/bin/perl

$rules = 20;
$IP_start = 0xc0a80101;
$IP_hex =   0xc0a80100;

sub inc_IP {
   $IP_hex++;
   $IP_hex &= 0xffffffff;

   # Avoid router, broadcast and network address (fe,ff,00)
   while (($IP_hex & 0xff) >= 0xfe || ($IP_hex & 0xff) == 0) {
      $IP_hex++;
      $IP_hex &= 0xffffffff;
   }
   $IP_a = $IP_hex >> 24 & 0xff;
   $IP_b = $IP_hex >> 16 & 0xff;
   $IP_c = $IP_hex >> 8 & 0xff;
   $IP_d = $IP_hex & 0xff;
   $IP = $IP_a.".".$IP_b.".".$IP_c.".".$IP_d;
}

sub dec_IP {
   $IP_hex--;
   $IP_hex &= 0xffffffff;

   # Avoid router, broadcast and network address (fe,ff,00)
   while (($IP_hex & 0xff) >= 0xfe || ($IP_hex & 0xff) == 0) {
      $IP_hex--;
      $IP_hex &= 0xffffffff;
   }

   $IP_a = $IP_hex >> 24 & 0xff;
   $IP_b = $IP_hex >> 16 & 0xff;
   $IP_c = $IP_hex >> 8 & 0xff;
   $IP_d = $IP_hex & 0xff;
   $IP = $IP_a.".".$IP_b.".".$IP_c.".".$IP_d;
}

inc_IP;

while (<STDIN>) {
   if (/[aA]/) {
      for ($i=0; $i<$rules; $i++) {
         eval `ipfwadm -I -i accept -S $IP -D 0/0`;
         inc_IP;
      }
      print "Next IP = $IP\n";
   }
```

```
    if (/[dD]/) {
       for ($i=0; $i<$rules && $IP_hex>$IP_start; $i++) {
          dec_IP;
          eval `ipfwadm -I -d accept -S $IP -D 0/0`;
       }
       print "Next IP = $IP\n";
    }

    if (/[qQ]/) {
       while ($IP_hex>$IP_start) {
          dec_IP;
          eval `ipfwadm -I -d accept -S $IP -D 0/0`;
       }
       exit;
    }
}
```

The script will wait for the user to press A, D, or Q. When pressing A, the script will insert 20 rules into the packet filter chains, when pressing D it will remove 20 rules, and finally when pressing Q it will remove all rules it has added. The script is careful not to use an IP address that ends with 254, 255 or 0 as these mean router, broadcast and network address.

The test script above was run on Bilbo and Odorf to measure the network performance by sending packets to the netperf server. The netperf server counted how many packets were received, and returned, at the end of the test, the number to the netperf client.

Since the host Bilbo uses a 100Mbps ethernet, and Odorf uses a 10Mbps, the tests on both machines were run just out of curiosity to see how much performance improvement the 100Mbps ethernet has.

4.3.1 Bulk transfer comparison PF/DPF

Figure 4.1 shows all the bulk transfer measurements in one graph. The graphs show that the largest size packet (1472 bytes) is almost unaffected by the additional overhead when using the 100Mbps ethernet. Interestingly there is a drop when using the slower 10Mbps ethernet. There is no logical explanation why there should be a performance drop when using the 10Mbps ethernet, and not when using the 100Mbps. Possibly it might hardware related.

4.4 Request Response measurements

The request response (RR) benchmark works by sending a packet to the host under evaluation. The host under evaluation is expected to reply to the packet. When the host replies, a new request is sent and the host has to respond again. The number of successful requests and responses are counted and displayed at the end of the test. To be able to do some comparisons of the bulk transfer measurement and the requests response measurement, the same parameters were used. Thus the protocol used is UDP, the socket buffer size is 32KB, and the packet sizes are 32, 64, 128, 256 and 1472 bytes.

DPF

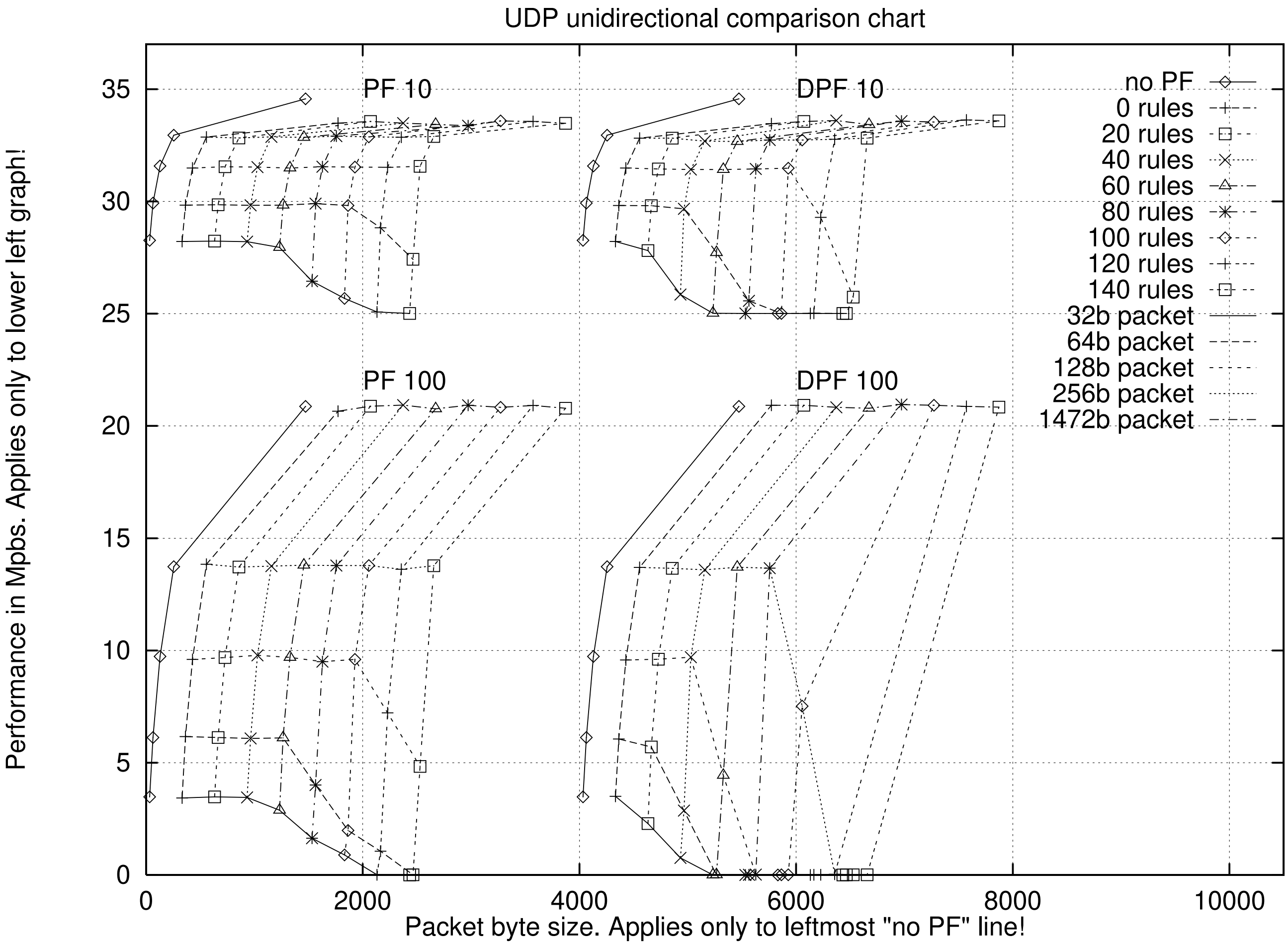

Figure 4.1: Bulk transfer comparison.

The rules were inserted in the incoming PF chain this time as well, but as the DPF host will be sending, the DPF is more or less bypassed.

The netperf program tries to get a statistical confidence level of 99%. However, the bulk transfer tests were run in the morning with little traffic on the UCCS backbone. When later running the RR benchmark, the 10Mbps measurements had much more variation (by looking at the jaggedness of the produced graphs) than before. Some of the values were thus rerun to make the graphs shown.

4.4.1 RR comparison PF/DPF

Figure 4.2 shows all the request response measurements in one graph. The drop in performance when using the DPF kernel is clearly seen. The drop is a result from the additional overhead of maintaining the internal DPF linked list. If comparing the trend of the PF and the DPF kernel, they do not differ significantly. This is as soon as the DPF host responds, the DPF entry will be inserted at the head of the DPF linked list, and the DPF search is bypassed as a incoming packet will find a match very quickly.

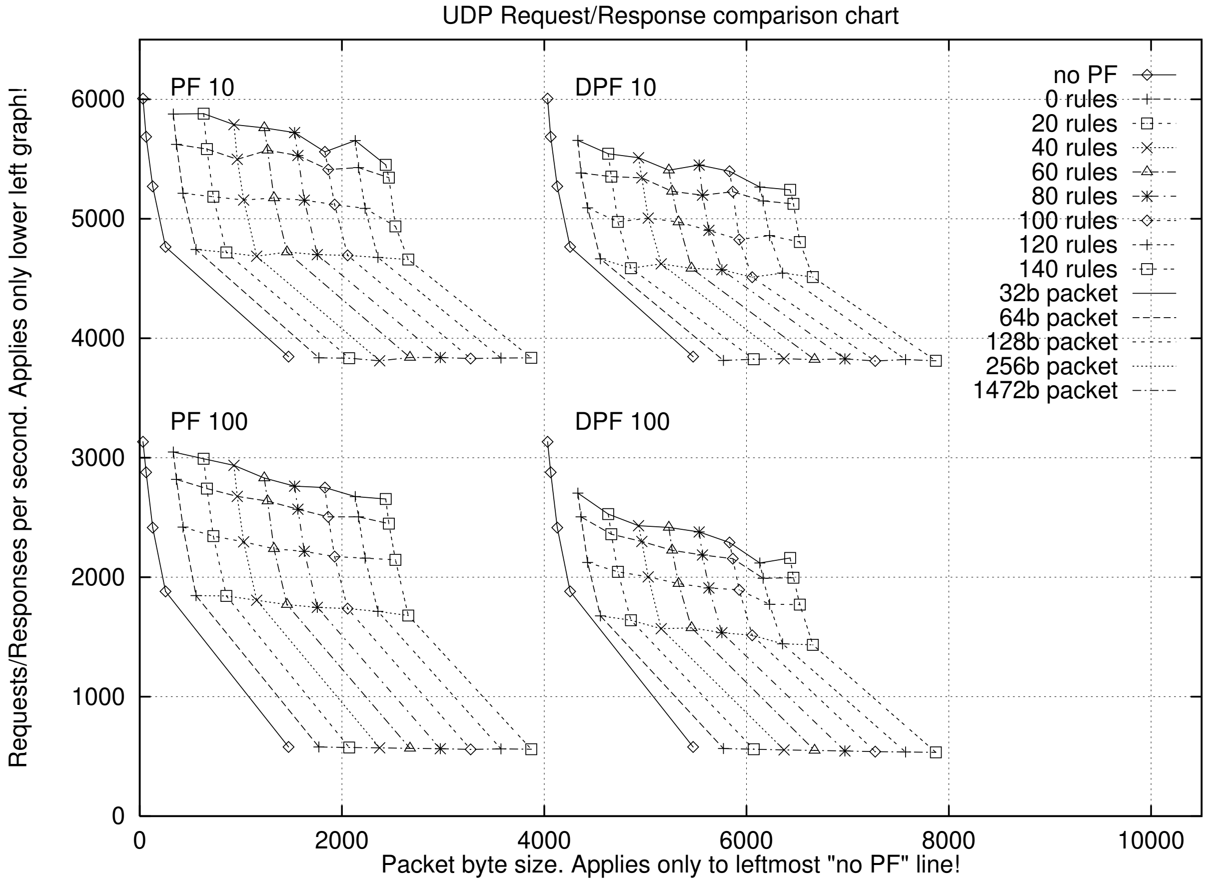

Figure 4.2: Request response comparison.

4.5 Maximum number of rules

An interesting question is how many rules can be used by a DPF system. The answer is that as long as there is free kernel memory to create the DPF entries and the PF entries, there is no limit. The next question is how much free kernel memory there is. When allocating memory for kernel applications a special function call called kmalloc() is used. The kmalloc() is similar to the C malloc() call, except that a GFP_ATOMIC can be specified which will allocate memory atomically, or fail if not enough memory is available. On a reasonably busy system there is usually about 20-60 rules in operation. Most are results of DNS lookups.

By studying a bug in the DPF code, we accidentally found how many rules can be appended as the code added rules for packets that were fragmented.

When a packet is fragmented by the network layer the TCP or UDP header is not copied to each fragment, only the IP header is. The DPF code was from the beginning not checking if the packet was fragmented or not. By reading what the DPF considered to be the port numbers, a lot of bogus rules were added, as the ports where not ports at all, it was raw data!

DPF

When the number of rules reached more than 2000 the entire system went down. The logs, which are eventually written to disk, were only stored in the memory cache, and nothing was thus written to disk. Because of this it is hard to know what made the system go down, but a reasonable guess is that too much memory was allocated which in turn made some other critical task to fail.

5. Conclusions and future improvements

This Section discusses some conclusions of using a DPF, and possible improvements or other avenues of research that may be of interest that have been discovered during the development of the DPF. The DPF can be used to enhance the network security of several hosts. The enhanced security is done in one single point, which means that the overall security of an entire intranet is enhanced in one step. The DPF is also in contrast of using proxying services, transparent for the users on the intranet, which means that no existing software has to be modified. The DPF is also very easy to maintain for the system administrator, with some luck he don't have to do anything at all. Even so, the system administrator should have a good knowledge about how the DPF works in order to use it in the most effective way.

If Linux is used for research in the future on UCCS, the overall network security must be maintained. In order to maintain and setup a Linux system, certain things has to be run as root. This might be considered to be a security risk. There is also the possibility of when changing the kernel, some bug is introduced and the network might be flooded with packets that should not be there. The easiest way would be to use a physically separated segment for the Linux research computers (or any other operating system as well). However, if using Linux that might not be a good idea as most of the currently available Linux documentation is found using the www, and thus some network access is almost a must. The best solution would be to connect the research computers on a separate segment like above, but allow the users of the research computers to access the UCCS backbone in a restricted manner. The restriction can be enforced by setting up another Linux computer running PF/DPF as a router between the research network and the UCCS backbone. The routing computer should be kept in a locked room and only be maintained by a professor or a trusted student. The root password of the computer used for routing should be a different one than the password used on the research computers. The above described setup would allow students to have root access to a powerful Unix platform without the risks.

5.1 Improvements

From the benchmarks of the DPF, the performance should be improved. Currently the DPF keeps track of both outgoing and incoming packets using the same list of DPF entries. That is perhaps not a good idea. By trying to find the entry of an incoming packet before an outgoing one, the current benchmarks could probably be improved as well.

The make the PF/DPF easier to maintain for the system administrator it would be nice if all rules could be more easily maintained in a graphical environment.

DPF

Documentation about how to setup and maintain a PF/DPF system would also be nice. The current documentation of the Linux PFs is not very good, and understanding of how everything is supposed to work has taken a lot of time.

5.2 Other research areas

The Linux kernel needs a good documentation before it becomes easier to improve parts of it. The netperf program is also supposed to be able to measure how busy the host under investigation is. This did not work. Setting up netperf to use variable packet sizes using TCP did not either work. UDP worked fine, and all the benchmarks were run using UDP.

References

TCP/IP related Books

[1][Sant] Michael Santifaller, *TCP/IP and ONC/NFS*, ISBN 0-201-42275-1.

[2][Huit] Christian Huitema, *Routing in the Internet*, ISBN 0-13-132192-7.

[3][Uyle] Uyless Black, *TCP/IP and related protocols*, ISBN 0-07-005560-2.

[4][Feit] Sidnie Feit, *TCP/IP architecture, protocols and implementation*, ISBN 0-07-020346-6.

[5][Stev94] W. Richard Stevens, *TCP/IP Illustrated Volume 1, The protocols*, ISBN 0-201-63346-9, 1994

[6][Wrig95] Gary R. Wright, W. Richard Stevens, *TCP/IP Illustrated Volume 2, The Implementation*, ISBN 0-201-63346-9, 1995

[7][Stev96] W. Richard Stevens, *TCP/IP Illustrated Volume 3, TCP for transactions, HTTP, NNTP and the Unix domain protocols*, ISBN 0-201-63495-3, 1996.

[8][Doub] Salim M. Douba, *Networking UNIX*, ISBN 0-672-30584-4

[9][Koch] Stephen G. Kochan, Patrick H.Wood, *Exploring the UNIX system*, ISBN 0-672-48447-1.

Unix system administration handbooks

[10][Hunt] Bruce H. Hunter, Karen Bradford Hunter, *Unix networks*, ISBN 0-13-089087-1.

[11][Gill] Daniel Gilly and staff of O'Reilly & Associates, *Unix in a nutshell*, ISBN 1-56592-001-5.

[12][Neme95] Evi Nemeth, Garth Snyder, Scott Seebass, Trent R. Hein, *Unix system administration handbook*, ISBN 0-13-151051.

[13][Wels95] Matt Welsh, Lar Kaufman, *Running Linux*, ISBN 1-5692-100-3, 1995.

[14][Kirc95] Olaf Kirch, *Linux network administrators guide*, ISBN 1-56592-087-2

[15][Reic] Kevin Reichland, David Burnette, *Unix communication and networking*, ISBN 1-55828-388-9.

[16][Dunp] Ed Dunphy, *The UNIX industry and Open Systems in transition*, ISBN 0-471-60608-1.

Unix programming

[17][Wall91] Larry Wall, Randal L. Schwartz, *Programming Perl*, ISBN 0-937175-64-1.

[18][Oram] Andrew Oram, Steve Talbott, *Managing projects with make*, ISBN 0-937175-90-0.

[19][Rago] Stephen A. Rago, *Unix System V Network programming*, ISBN 0-201-56318-5.

[20][Stev] W. Richard Stevens, *Advanced programming in the Unix environment*, ISBN 0-201-56318-5

Firewall and security related books

[21][Siya] Karanjit S. Siyan, Chris Hare, *Internet Firewalls and network security*, ISBN 1-56205-437-6.

[22][Madr] Thomas W. Madron, *Network security in the 90's*, ISBN 0-471-54777-8.

[23][Dutt] Ellen Dutton, *LAN security handbook*, ISBN 1-55851-387-6.

[24][Ches94] William R. Cheswick, Steven M Bellowin, *Firewalls and Internet Security*, ISBN 0-201-63357-4.

[25][Arno] N. Derek Arnold, *Unix Security, a practical tutorial*, ISBN 0-07-002560-6.

[26][Hend] Mike Hendry, *Practical Computer Network Security*, ISBN 0-89006-801-1

[27][Chap] D. Brent Chapman, Elizabeth D. Zwicky, *Building Internet firewalls*, ISBN 1-56592-124-0.

[28][Farr] Rik Farrow, *Unix System Security*, ISBN 0-201-57030-0.

Useful links

[29][Linu] The Linux information site, *Linux organization*, *http://www.linux.org*.

[30][Debi] The Debian Linux information site, *Debian organization*, http://www.debian.org.

[31][Sock] Ron Kuris, Steven Lass and Wei Lu, *Socks FAQ*, ftp://www.cdrom.com/pub/security/coast/doc/faq/faq_socks.

[32][Soci] David Koblas, *Socks information*, http://www.socks.nec.com.

[33][Netp] Hewlett Packard, *The Netperf program*, http://www.cup.hp.com/netperf/NetperfPage.html.

[34][John] Michael K. Johnson, *Linux Hacker's guide*, http://www.redhat.com/HyperNews/get/khg.htm.

[35][LinM] *Linux kernel mailing list*, http://www.uwsg.indiana.edu/hypermail/linux/kernel.

[36][Cert] *Cert advisory Spoofing*, ftp://info.cert.org/pub/cert_advisories/CA-95:01.IP.spoofing, January 95, CA-95:01

Articles

[37][SSC] SSC, Specialized Systems Consultants, *The Linux Journal*, http://www.ssc.com/lj/.

[38][Modu] Modul, J.C; Rashid, R.F; Accetta, M.J, *The packet filter: and efficient mechanism for user-level network code*, Operating Systems Review, vol.21, no.5, p39-51.

[39][Guha96] Guha, B; Muherjee, B, *Network security via reverse engineering of TCP code: vulnerability analysis and proposed solutions*, Proceedings IEEE INFOCOM '96. The conference on computer communications. vol2, pp603-610.