

1. Introduction

Linux is a free Unix-like operating system, originally conceived by Linus Torvalds during his graduate studies at Helsinki University in 1991. Since its conception, the Linux project has received contributions from thousands of software developers around the world, most notably a very robust networking subsystem, and kernel-level implementations of the TCP/IP protocol suite.

Linux's low cost, robustness, and adherence to open networking standards have made it a favorite choice for such applications as network firewalls, IP Network Address Translation (NAT) and IP Masquerading, and even Virtual Private Networks (VPN's). While all of these applications solve different problems, they all require some common functionality from the Linux networking subsystem. Specifically, applications require that network packets be intercepted before entering the operating system's protocol stack. The packets may need to go through complex filtering (firewalls) and modification (NAT) before entering the protocol stack. The packets must be processed very quickly and efficiently, since in a typical networking configuration, packets will enter the protocol stack on the order of megabits per second (Mbps) or faster.

The current maintainer of the Linux kernel's firewall capabilities, Rusty Russell, realized that current independent efforts to do firewalling and NAT under Linux were facing similar problems related to complex requirements for packet filtering and modification. In 1998, he started a project called *netfilter* which is aiming to create a flexible framework for doing modification of packets (also referred to as packet mangling) outside of the Berkeley socket interface. Netfilter is currently under development in the 2.3.x series of experimental Linux kernels, and when finished will be

included in the 2.4.x series of stable kernels. This paper will discuss the architecture of netfilter and its relation to the Linux kernel's networking subsystem.

2. Ipchains

The current stable series of Linux kernels, version 2.2.x, includes *ipchains* which allows the the Linux kernel to performing packet filtering, a necessary feature of network firewalls. Ipchains was written by Paul "Rusty" Russell, an independent consultant and core Linux contributor, based in Australia. Ipchains consists of code which must be compiled directly into the Linux kernel, or compiled as a loadable kernel module. It also consists of a command-line utility (also called ipchains) which runs in user-space. An end-user would use the ipchains utility for configuring a firewall capabilities of a Linux system.

For example, if an end-user wished to deny ICMP packets from being accepted by a Linux server, he or she could do the following:

- check that IP firewalling is compiled into the kernel by verifying that the file: `/proc/net/ip_fwchains` exists.
- type: `ipchains -A input -p icmp -j DENY`

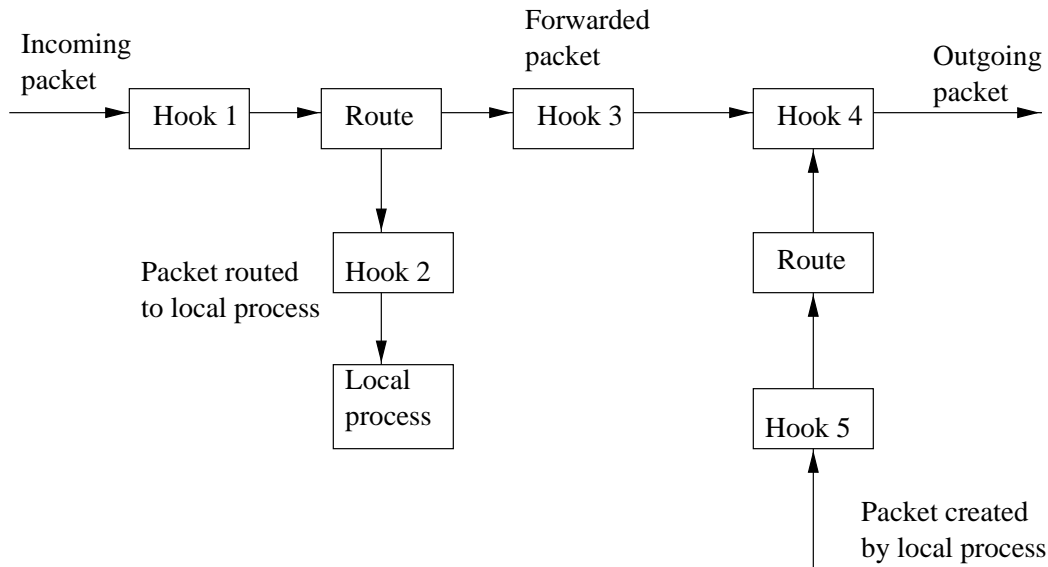
This firewall rule, also called a "chain" is stored in the Linux kernel. There are three types of chains: input, output, and forward. Input rules, like the one just illustrated, are executed when an incoming packet matches the rule. Output rules are executed when an outgoing packet matches the rule. Forward rules are executed when the packet is destined for another machine (ie. a router configuration). The rule illustrated above basically says, for all incoming packets which have the protocol type ICMP, deny them from entering the protocol stack.

Ipchains works quite well, but its maintainer, Rusty Russell, determined that it had some deep architectural problems, which prompted him to start the Netfilter project. The majority of Ipchains exists as kernel-level C and C++ code. There is currently no good interface for accessing Ipchains functionality from user-space, which prevents firewall applications from being written in other commonly used languages such as Perl, Tcl, or Java.

Ipchains and NAT (Network Address Translation) were developed as separate projects under Linux and currently interact in very complicated ways. Ipchains basically tries to accomplish some forms of packet filtering, masquerading, manipulation of the TOS (Type of Service) field in the IP header, and routing. However the current system is not extensible, so doing things like creating filter rules per user, or doing MAC address filtering is currently not possible. As the types of filtering becomes more complicated, Russell found that it might not be very efficient to store such complex dynamic rules. For these and other reasons, Rusty Russell decided that writing a general purpose kernel module for manipulating packets would be a good way to combine NAT and firewalling in one project.

3. Netfilter

Netfilter consists of a series of hooks to the protocol stack. The hooks are specific for each protocol. Currently Netfilter hooks for IPv4 and IPX exist, while hooks for IPv6 are currently under development. The following diagram, taken from the Netfilter-Hacking HOWTO document by Rusty Russell, illustrates the hooks for IPv4.



The hooks for each protocol are defined in protocol specific header files, ie. for IPv4 they are defined in the kernel header file:

`linux/netfilter_ipv4.h`. When a packet enters the Netfilter framework for an IPv4 system, it enters the Hook 1 (`NF_IP_PRE_ROUTING`). After passing through this hook, the packet is routed either to a local process, or forwarded on to another host. It can also be dropped altogether at this stage. If the packet is routed to a local process, it enters Hook 2 (`NF_IP_LOCAL_IN`), before being sent to the process. If the packet is forwarded to another host, the packet first enters Hook 3 (`NF_IP_FORWARD`), then Hook 4 (`NF_IP_POST_ROUTING`) before being sent back out on the wire. If a packet is created locally, it first enters Hook 5 (`NF_IP_LOCAL_OUT`), and is then routed. After being routed, it enters Hook 4, and is sent out on the wire.

Each hook can provide one of four responses. The responses are defined in the header file `linux/netfilter.h`. The responses are:

- NF_DROP (drop the packet and don't let it traverse any further)
- NF_ACCEPT (allow the packet to traverse the framework)
- NF_STOLEN (the hook has taken over the packet, so don't continue traversal)
- NF_QUEUE (queue the packet, possibly to be handle in userspace)

3.1 Iptables

Iptables is a descendant of Ipchains, but it is built on top of the Netfilter framework. Unlike Ipchains, Iptables does no rewriting of the packets; it just performs filtering. This significantly reduces the code size of Iptables, since it does not include any logic for NAT and masquerading, like Ipchains does. Iptables uses the hooks at NF_IP_LOCAL_IN, NF_IP_LOCAL_OUT, and NF_IP_LOCAL_FORWARD.

Iptables includes a command-line user-space utility (iptables) which accepts commands in a syntax identical to the ipchains utility. For example, to deny ICMP packets, a user would type:

```
iptables -A input -p icmp -j DENY
```

The rules for iptables are stored in an array in memory. The array contains data structures of type `struct ipt_kern_entry` which is defined in `iptables.c` in the Netfilter source code. The data structure is defined as:

```
struct ipt_kern_entry
{
/* contains the specifications for the IP header to match*/
struct ipt_ip ip;

/* indicates the optional match functions, eg. tcp packet
matching. */
struct ipt_match *match;

/* contains data for the extra match functions */
union ipt_matchinfo matchinfo;

/* indicates the action to perform if the packet matches */
struct ipt_target *act;
};
```

```
    unsigned int cacheinfo;

    /* contains data for the function which runs if it matches*/
    union ipt_targinfo targinfo;
};
```

3.2 NAT

IP NAT (Network Address Translation) and IP Masquerading, which is also known as Network Address Port Translation, can be done using the Netfilter hooks. For packets which are not local, the `NF_IP_PRE_ROUTING` and `NF_IP_POST_ROUTING` hooks can be used. For packets which are local, the `NF_IP_LOCAL_IN` and `NF_IP_LOCAL_OUT` hooks can be used.

One important aspect of implementing NAT is to do proper *connection tracking*. For example, if a TCP connection is undergoing NAT, it is necessary to keep track of which parts of the header are being rewritten (src or dst) on outgoing packets, and which incoming packets correspond to this connection. A connection tracking module has been written on top of the Netfilter framework for the TCP, UDP, and ICMP protocols.

In the connection tracking module, an outgoing packet is converted to a "tuple" which represents a portion of the protocol header of interest. For TCP, these would be the source IP address, source port, destination IP address, and destination port. Each tuple has a portion which can be modified, and a portion which cannot be modified. For an outgoing TCP packet, the source address and port numbers can be modified, but the destination address and port numbers cannot be modified. Each outgoing packet has an inverse tuple for the incoming packet. For example, an

outgoing ICMP packet from 1.2.3.4 to 4.5.6.7 with ICMP ID 0x850 has an inverse ICMP packet from 4.5.6.7 to 1.2.3.4 with ICMP ID 0x850.

3.2.1 Ipnatcl

Ipnatcl is a command-line user-space utility, built on top of the Netfilter framework, which can be used to perform NAT. It replaces the ipmasqadm utility which was used for IP masquerading in Linux 2.2 kernels. Ipnatcl can be used to perform normal NAT (the modification of source IP addresses in packets), reverse NAT (the modification of destination IP addresses in packets).

In the following example, if you have a firewall with IP address 1.2.3.4, it is possible to redirect port 80 HTTP web requests from the firewall to an internal webserver running on port 8080 with IP address 4.5.6.7 with the following:

```
ipnatcl -I -d 1.2.3.4 -p tcp --dport 80
        -b dest -t 4.5.6.7 --to-port 8080
```

The `-b dest` flag specifies that the destination portion of incoming packets be rewritten.

Ipnatctl can also be used to perform simple load balancing. The following example will direct web traffic destined for webserver to webserver1 and webserver2, based on the least number of connections:

```
ipnatctl -I -d webserver -p tcp --dport 80 -b dest
                                                -t webserver1
ipnatctl -I -d webserver -p tcp --dport 80 -b dest
                                                -t webserver2
```

The two rules both match packets with a destination IP address webserver, and a TCP destination port of 80. The `-b dest` portion of the rule specifies that the destination IP address of the packet should be rewritten to

webserv1 or webserv2. Since both rules have identical matching conditions, the rule with the least number of current connections will be executed. While simplistic, this is an extremely useful way of doing load balancing for applications like web servers.

3.3 Using Netfilter in User-space

The modules for packet filtering and NAT can be extended by doing kernel-level C or C++ coding. However, Netfilter has an optional facility for manipulating packets in user-space, outside the kernel. Filtering packets in user-space would not be optimal for filtering large bandwidths, but for smaller bandwidths, and incremental testing and development, it is useful.

The first step for adding the capability to do user-space Netfiltering is to create an appropriate device with the Unix `mknod` command. For IPv4, the user would type:

```
mknod /dev/netfilter_ipv4 c 120 2
```

This creates the `/dev/netfilter_ipv4` device with the major number 120 (which is reserved for EXPERIMENTAL use), and a minor number of 2 (which is the protocol family for IPv4, `PF_INET` 2, defined in `linux/socket.h`).

If a user starts to read this device, he or she will be able to see all the packets running through the Netfilter framework, since the default behavior of the device is to register with all Netfilter hooks in the system. The user can be more selective about which hooks to read from by using the `ioctl()` system call. To do this, the user would first get a file descriptor to the netfilter device with the `open()` system call:

```
int fd = open("/dev/netfilter_ipv4", O_RDONLY);
```

Then, the user would populate a struct `nfdev_condition` which is defined in `netfilter_device.h`:

```
struct nfdev_condition
{
    /* Bitmask of hook numbers to match (1 << hooknum). */
    unsigned int hookmask;
    /* If non-zero, only catch packets with this mark. */
    unsigned int mark;
    /* If non-zero, only catch packets of this reason. */
    unsigned int reason;
};
```

The user would then use the following `ioctl()` calls to add or remove a condition (the `ioctl`'s are defined in `netfilter_device.h`):

```
ioctl(fd, NFDIOSADDCOND, &a_nfdev_condition); /* to add */
ioctl(fd, NFDIOSDELCOND, &a_nfdev_condition); /*to delete*/
```

When a user reads from `/dev/netfilter_ipv4`, he or she will receive each packet preceded by a struct `nfdev_head` data structure which is defined in `netfiler_dev.h`. The struct `nfdev_head` structure is:

```
struct nfdev_head
{
    /* Input interface name, could be empty */
    char iniface[IFNAMSIZ];
    /* Output interface name, could be empty */
    char outiface[IFNAMSIZ];
    /* Mark value. */
    unsigned long mark;
    /* Total packet length. */
    u_int32_t pktlen;
    /* Reason. */
    u_int32_t reason;
    /* The hook you came in on. */
    u_int32_t hook;
};
```

After reading the packet, the user can accept or reject the packet by populating a struct `nfdev_verdict` data structure, and executing an `NFDIOSVERDICT` `ioctl()` call. The contents of the `nfdev_verdict` data structure are:

```

struct nfdev_verdict
{
    /* NF_ACCEPT, NF_DROP, NF_QUEUE */
    unsigned int verdict;

    /* New packet length. */
    u_int32_t pktlen;

    /* Mark value. */
    unsigned long nfmark;

    /* Reason: explains mark value. */
    u_int32_t nfreason;
};

```

and the corresponding ioctl call would be:

```

ioctl(fd, NFDIOSVERDICT, &a_nfdev_verdict);

```

By using these ioctl() calls on the Netfilter device, it is possible to add Netfilter hooks and accept or reject packets entirely in user-space. This greatly adds to the packet-filtering capabilities of Linux.

4. Understanding the Linux networking system

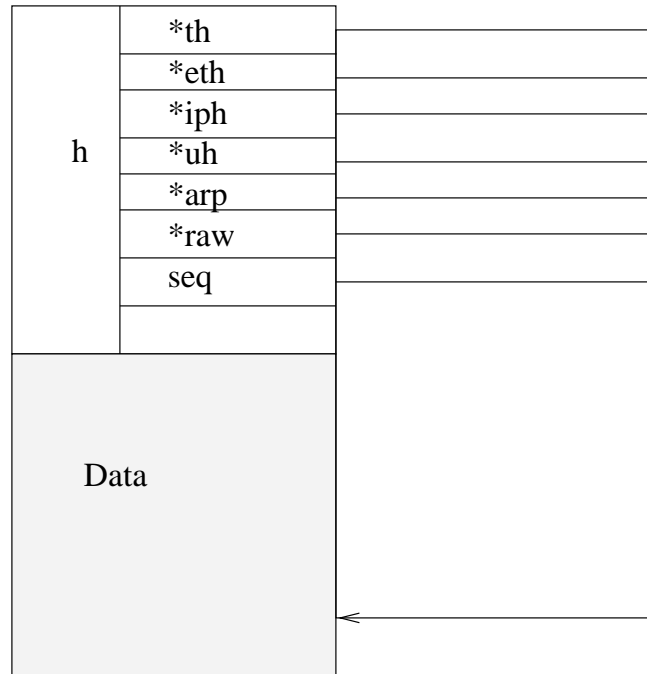
To have a better understanding of some of the design decisions and implementation details of Netfilter, it helps to have an understanding of the mechanisms and data structures which make up the Linux kernel's networking implementation.

4.1 sk_buff data structure for memory management

Once of the most important and most often used data structures in the Linux is the sk_buff data structure. This data structure is defined in `include/linux/skbuff.h`. The sk_buff is used to manage individual communication packets. sk_buffs are used to provide buffering and flow control to network protocols. An sk_buff is a control structure with a block of memory attached. The control structure contains pointers which point to

the header information in the attached data structure.

Linux uses doubly linked lists of `sk_buffs` as linear buffers. This is different from BSD derived Unixes which uses chains of small buffers (mbufs). Linear buffers use slightly more memory than mbufs, but are very fast for the most commonly used operations used on network buffers, ie. append to end, remove from start.



`sk_buff` contains pointers to header information in payload

The `sk_buff` structure is quite large, but some of its component data structures are detailed here:

```
struct sk_buff{
    /* pointers in doubly linked list */
    struct sk_buff *next, *previous;

    /* socket which owns this sk_buff */
    struct sock *sk;
```

```

    /* device which this sk_buff arrived from or will
       leave by */
    struct device *dev;

    /* Transport layer header */
    union
    {
        struct tcphdr    *th;
        struct udphdr    *uh;
        struct icmphdr   *icmph;
        struct igmp_hdr  *igmp;
        struct iphdr     *iph;
        struct spxhdr    *spx;
        unsigned char    *raw;
    } h;

    /* Network layer header */
    union
    {
        struct iphdr     *iph;
        struct ipv6hdr   *ipv6h;
        struct arphdr    *arp;
        struct ipxhdr    *ipx;
        unsigned char    *raw;
    } nh;

    /* Link layer header */
    union
    {
        struct ethhdr    *ethernet;
        unsigned char    *raw;
    } mac;

    .....
};

```

As can be seen in the definition of this data structure, the `sk_buff` contains pointers to header information at the link layer, network layer, and transport layer, so it plays a key role in the memory management of packets in the Linux kernel. Also, the `sk_buff` contains pointers to the next and previous `sk_buff` in a doubly linked list.

The Linux kernel includes utility functions for managing lists of `sk_buffs`. All of these operations are atomic, in order to prevent problems which occur when two threads of execution try to simultaneously perform

operations on the same block of memory. These operations include:

- `skb_dequeue()`, pull the first buffer from a list
- `skb_queue_head()`, place a buffer at the start of a list
- `skb_queue_tail()`, place a buffer at the end of a list
- `skb_unlink()`, remove a buffer from the list which contains it. The buffer is not freed, just removed from the list
- `skb_insert()`, place a buffer before a specific buffer in a list (often used for protocols like TCP, which need to re-order buffers to keep the packets in order)
- `skb_append()`, place a buffer after a specific buffer in a list (also used for TCP)
- `alloc_skb()`, create a new `sk_buff` and initialize it
- `kfree_skb()`, release a buffer, lower the memory count of the socket which owns this buffer

The following is a very simplified code example for managing a list of buffers. It is taken from [1],

```
void append_frame(char *buf, int len)
{
    /* allocate a sk_buff */
    struct sk_buff *skb = alloc_skb(len, GFP_ATOMIC);

    if(skb == NULL){
        /* Couldn't allocate a sk_buff, dropped a packet*/
        my_dropped++;
    }
    else{
        /* Reserve len bytes of data in the sk_buff */
        skb_put(skb, len);
        /* Copy the buffer into the sk_buff */
        memcpy(skb->data, data, len);
        /* Add the sk_buff to a linked list of sk_buffs */
        skb_append(&my_list, skb);
    }
}

void process_queue(void)
{
    struct sk_buff *skb;
    /* Pop the first sk_buff off of the linked list */
    while( (skb = skb_dequeue(&my_list)) != NULL){
        process_data(skb);
        /* Free the sk_buff after doing something with it */
        kfree_skb(skb, FREE_READ);
    }
}
```

```
}
```

The previous example illustrates how one would typically use the memory management functions for `sk_buffs`. `append_frame()` would typically be code inside a network device driver which is invoked during an interrupt when a network device receives a packet. This is illustrated in the `netif_rx()`, found in `net/core/dev.c` which receives a packet from a device driver and queues it to the upper protocol levels. Similarly, `process_queue()` is similar to `net_bh()`, found in the same file, which pops `sk_buffs` off a queue to be processed at higher levels.

4.2 How data travels the protocol stack in the Linux networking subsystem

In addition to understanding the fundamental memory data structure used in the Linux networking subsystem, it is also useful to know how packets travel through the protocol stack. This will give more insight into where Netfilter hooks are placed when doing packet filtering.

The Linux networking subsystem is based on the BSD socket interface, which allows file I/O operations such as `open()`, `read()`, `write()`, and `close()` to be performed on communication endpoints. For example, if a socket was opened between two hosts, the first host can write to the socket and the other host can read from the socket. For example, if a process calls:

```
write(socket, data, length);
```

This call will invoke the kernel function `sys_write()`, which is part of Linux's Virtual Files System. `sys_write()` insures that a write operation can be done on the file descriptor `socket`, and that the memory referenced

by data can be read. Since this file descriptor is for a socket, the `sock_write()` function is then invoked. `sock_write()` searches for the socket data structure associated with the file descriptor's inode. For a socket which belongs to the `AF_INET` family of sockets (most commonly used over standard TCP/IP networks), the `inet_write()` is then called with parameters which include the BSD socket data structure, and the data to be written. `inet_write()` in turn calls `inet_send()`, which extracts a pointer to the INET socket structure. This socket structure contains a pointer to the operation vector of the TCP implementation. `inet_send()` takes this pointer, and invokes its write operation, `tcp_write()`. `tcp_write()` takes parameters which include the pointer to the INET socket, the data, the length of the data, and other flags.

`tcp_write()` tests for certain error conditions, and if the tests pass, it allocates memory with the `wmalloc()` function. This memory contains an `sk_buff`, the header, and TCP segment. `tcp_write()` then calls `build_header()`, which first calls `ip_build_header()` to build the IP header, and then `tcp_build_header()`, to build the TCP header.

`tcp_write()` then copies data from the process to the address space of the TCP segment. If the data is larger than the maximum TCP segment size (MSS), the data is copied into multiple TCP segments. Alternatively, multiple small chunks of data can be combined into one TCP segment. Linux uses linear buffers, so this can be quite efficient.

`tcp_send_skb()` is then called to transfer the data contained in the `sk_buff`. At this stage, the checksum is calculated for the TCP segment, and other protocol specific information is added. `ip_queue_xmit()` is then called to put the packet in a wait queue of packets which are ready to be transferred. `ip_queue_xmit()` also calculates the IP header checksum.

The packet is then passed to `dev_queue_xmit()` which prepares the packet to be sent to the network device. A function which is pointed to by the pointer `hard_start_xmit` is then invoked, which passes the packet to the network device. The function which `hard_start_xmit` points to is different, depending on the type of network device being used. The network device then transmits the packet over a medium such as Ethernet, Token Ring, FDDI, ATM, etc.

If the receiving host at the other end is using Ethernet, when its Ethernet adapter receives the packet, it will trigger an interrupt which will call the `ei_interrupt()` function. This function will invoke `ei_receive()` if the packet was received without error. `ei_receive()` will then call the function pointed to by `block_input` (specific to a device driver), which sets up a new memory buffer and writes the packet to it.

`netif_rx()` is then called to place the packet on a list of all packets received by the system called the backlog list. The `ei_interrupt()`, `ei_receive()`, `block_input`, and `netif_rx()` functions are all called during the same interrupt.

The `net_bh()` function is then called, and it sets the `raw` pointer in the `sk_buff` structure to the beginning of the protocol packet, after the data link header. The packet type in the data link header will determine which protocol receive function will be invoked. For an IP packet, the `ip_rcv()` function will be called. `ip_rcv()` verifies the IP header checksum, and if IP options are set, the handling routines for these options are executed. If a packet is destined for another host, `ip_forward()` is invoked. If IP packets are fragmented, they are reassembled with `ip_defrag()`. The `raw` pointer in the union `h` is then set to the end of the IP header. Since the next protocol is TCP, the receive function for TCP, `tcp_rcv()` is then called.

The INET socket to which the packet belongs to will be determined by the `get_sock()` function, which examines the sender and destination addresses and port numbers.

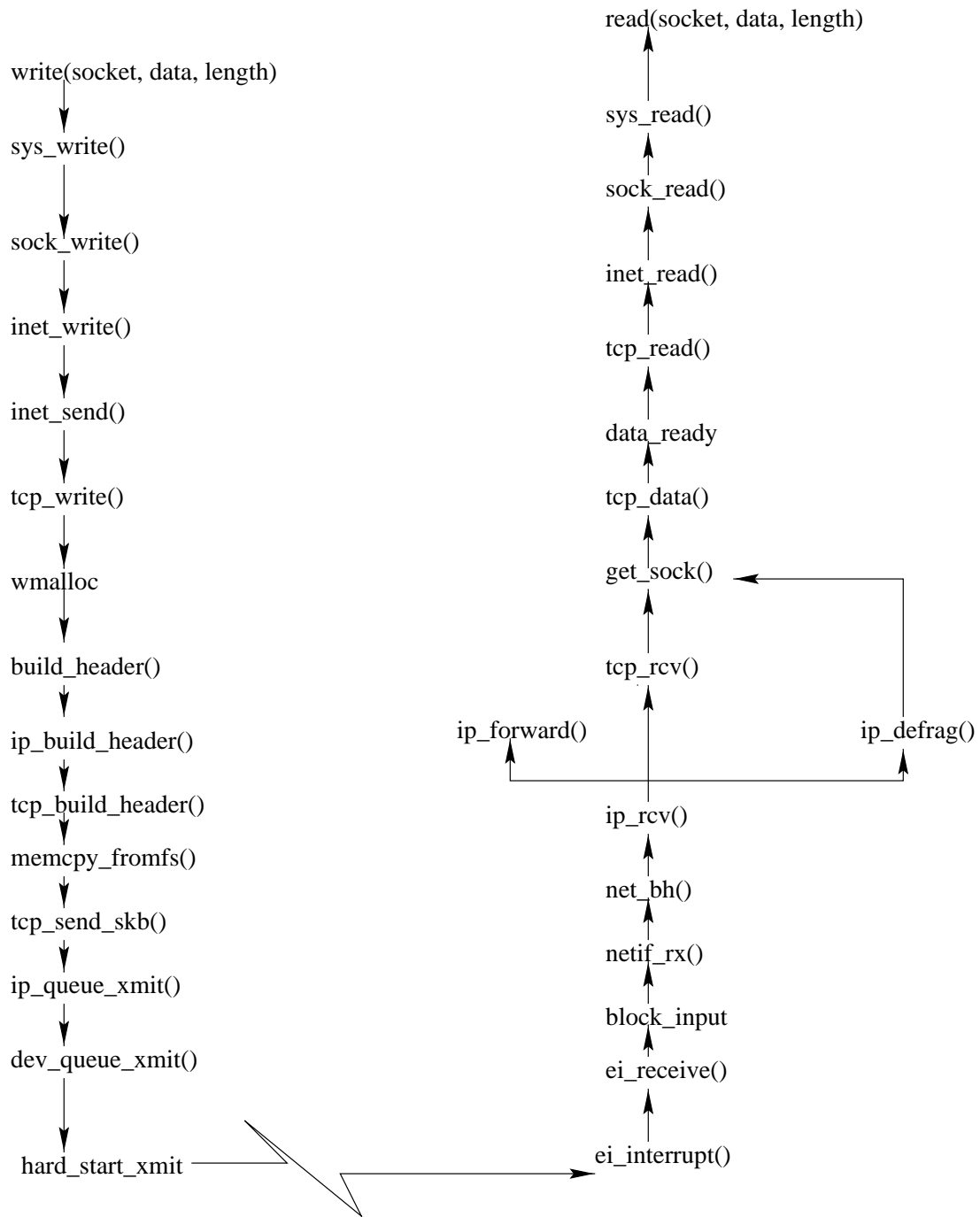
The data will be checked for integrity, and then `tcp_data()` will be called to add the packet to the list of `sk_buffs` received for the appropriate INET socket. If the data has been received in order and is ready to be processed, the appropriate ACK packets will be sent back after a delay, and the `data_ready` operation on the INET socket will be called. This function tells the socket that it has data which is ready to be processed. If the socket processes the data, it will pass through

`tcp_read()`, `inet_read()`, `sock_read()`, and `sys_read()`, in a matter analogous to the equivalent `write()` function calls. Finally, the receiving

host can read the data with the function:

```
read(socket, data, length);
```

The traversal of a packet across the protocol stacks of two Linux hosts can be summarized with with following diagram:



5. Conclusions

The Netfilter project is an attempt to provide a robust framework for developing applications under Linux which require that network packets be examined and/or modified outside of the protocol stack. Netfilter provides hooks to various stages of a packet's traversal through the Linux, which allow an application to accept, deny, or modify a packet. Netfilter can be used in kernel space or in user space, unlike its predecessors, which could only run in kernel space.

In addition to Netfilter, this paper discussed various kernel data structures and procedures related to the processing of network packets under Linux, since knowledge of these areas is required to extend Netfilter.

REFERENCES

- [1] Cox, Alan. "Network Buffers and Memory Management," *Linux Journal*, Oct. 1996
(<http://www2.linuxjournal.com/lj-issues/issue30/1312.html>)
- [2] Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., and Verworner, D. *Linux Kernel Internals*, Reading, MA: Addison–Wesley, 1996
- [3] Russell, Rusty. *Netfilter Hacking HOWTO*,
<http://netfilter.kernelnotes.org>
- [4] Russell, Rusty. *iptables HOWTO*, <http://netfilter.kernelnotes.org>
- [5] Russell, Rusty. *ipnatctl HOWTO*, <http://netfilter.kernelnotes.org>
- [6] Russell, Rusty. *Linux ipchains HOWTO*,
<http://www.rustcorp.com/linux/ipchains/>
- [7] Linux kernel source code, versions 2.2.12 and 2.3.34, using the Linux LXR cross referencing tool at: <http://lxr.linux.no/source/>
- [8] McCanne, S., and Jacobson, V. "The BSD Packet Filter: A New Architecture for User–level Packet Capture," *Proc. Winter Usenix Conference*, USENIX, 1993