5-2006

A Prototype for In Situ Packet Filtering

William Watson Cline
College of William and Mary

A Prototype for In Situ Packet Filtering

William Cline

April 2006

Abstract

Traditional packet-filtering firewalls control network traffic based on pre-defined rules. These rules operate on packet envelope information, such as the IP or Ethernet headers. Some new firewall applications use "deep filtering," operating on packet payloads. This requires quick access to the full contents of network packets, as well as the ability to modify those contents while the packet is in transit. The Linux kernel includes tools or performing both "shallow" header-based filtering and deep filtering. However, the current deep filtering implementation is too slow for some applications.

We present a modified implementation of the Netfilter Project's IP_QUEUE module with the goal of higher performance. Our prototype yields a modest but substantial speed improvement. We discuss this prototype and present suggestions for further improvements.

Contents

List of Figures

List of Tables

1 Introduction

As computer networks have become more pervasive and their applications more varied, the need to secure them has grown commensurately. When a computer connects to a network (becomes a "network host"), its owner may wish to control how it communicates with other hosts. Firewalls are commonly-used tools for controlling access to a network host or group of network hosts.

A firewall is a software package running on a network host that filters incoming and outgoing communication. On an Internet Protocol (IP) network, messages are split into discrete, individually-delivered units called packets. Packets in transit may be processed by a firewall on the network host from which they originate, the host to which they are addressed, or by any host in between. When a packet reaches a firewall at any point along its journey, it may be allowed to pass or blocked ("dropped").

In a traditional rule-based packet filtering firewall, each packet is individually examined. The system administrator programs a set of rules, and the firewall consults these rules whenever a packet arrives on one of its network interfaces. Firewall rules are based on a packet's "envelope" information—the packet's source and destination addresses, its source or destination ports, its size, et cetera. Each rule defines a set of characteristics and an action to be taken on packets with matching envelope information.

For example, one rule might be to, "Block all incoming traffic destined for the HTTP server port," Another might, "Block all SMTP traffic coming from the network host at IP address 192.168.0.1."

1.1 Packet filtering firewalls in Linux: `netfilter`

The GNU/Linux family of operating systems includes a packet filtering framework known as `netfilter`. `Netfilter` comprises a packet-filtering firewall along with a number of extensions and utilities [8]. One of the most important parts of `netfilter` is `iptables`,

a command-line utility for viewing, adding, removing, and modifying firewall rules. These rules are applied by the kernel itself to evaluate network traffic. Like other rule-based firewall solutions, `iptables` allows the construction of sophisticated rule sets based on packets' source and destination addresses, source and destination ports, and other header information. Furthermore, the set of rules in effect can be dynamically modified, either by the system operator or by automated scripts. Thus, `iptables` can be used to respond to changing conditions and requirements without rebooting the host or otherwise interrupting the flow of network traffic.

`iptables` is highly versatile, as rules can be written for any network protocol supported by Linux. Furthermore, it is application-agnostic; the envelope information for traffic following a given protocol is all formatted similarly. `iptables` does not examine the actual packet data (the "payload"). There are times, however, when it would be useful to control the flow of packets based on their contents. In these cases, the agnosticism of `iptables` becomes a weakness.

1.2 Deep packet filtering in Linux: the ip_queue kernel module

There are a number of uses for filters that are aware of the contents of a packet, rather than merely where it came from and where it is going [2, 1, 3, 4, 5, 6, 7]. Many commercial vendors, including Microsoft, Cisco, and Symantec, have begun to develop deep filtering tools [9]. These "deep filtering" applications require greater functionality than that supplied by `iptables` and other traditional packet filters.

Some or all of the network traffic arriving at a host may be subjected to deep filtering. Choosing which packets to filter deeply remains the task of conventional rule sets set up and maintained with `iptables`. In addition to accepting or rejecting packets that meet specified header-based criteria, `netfilter` allows the kernel to defer certain packets—those that match the appropriate rule(s)—for further processing. Packets deferred in this manner

are "queued" until they can be undergo deep processing.

This extra processing is not performed within the operating system. The application-specific nature of deep filtering makes the inclusion of deep filtering code in the operating system kernel inappropriate. Deep filtering schemes are many and varied, and some are even specific to network traffic generated by particular software packages. Custom kernel code, in the form of one or more dynamically loaded modules, would have to be developed for every filtering application. This would be needlessly difficult from a software engineering perspective. Adding custom code to the kernel also poses security and stability issues, as an operating system is not protected against errors (security or otherwise) in its own kernel, as it is against userland code.

Because it is application-specific and requires dynamic change during a system's runtime, and because we wish to protect the operating system from errors in the filtering software, deep filtering ought to be carried out at the user level. Using the `netfilter` IP_QUEUE kernel module and the accompanying `libipq` C library, one can develop a userland packet filtering daemon that partners with the kernel to provide deep packet filtering.

The IP_QUEUE module interfaces with the kernel by introducing new functionality into `iptables`. Rather than accept or reject a packet matched by one of the rules in the kernel's firewall table, a rule may be cause a packet to be queued for further processing. The queued packet is sent to a daemon running on the system, processed, and then accepted or rejected. As required by some applications [2], the contents of the packet can also be modified by the daemon. The kernel then continues to either deliver or drop the packet in the usual fashion. Figure 1 describes the operation of the kernel/userland partnership.

The userland daemon itself is written using the `libipq` library, which provides a set of functions for receiving a packet from the kernel, examining its contents, and returning a "verdict" (accept or reject) to the kernel. Packets are received from the kernel by the processing daemon via a special socket connection using a special protocol called NETLINK_FIREWALL,

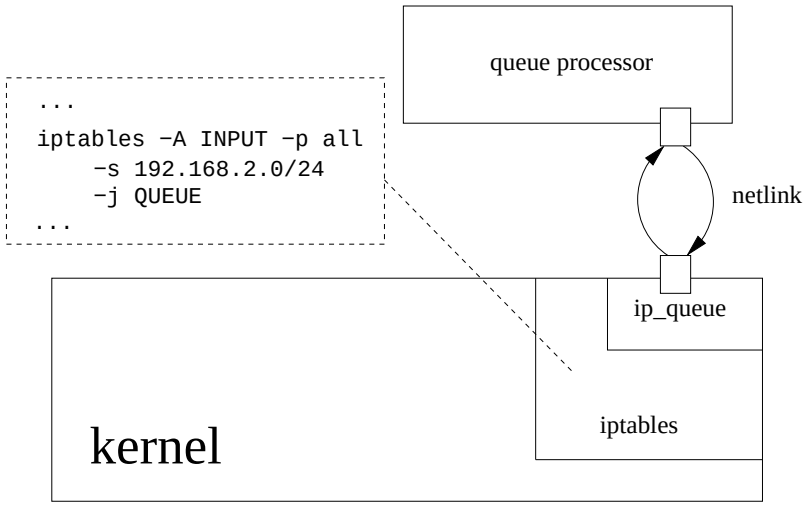

Figure 1: Overview of IP_QUEUE module operation

part of the PF_NETLINK protocol family.

1.2.1 Important data structures

To understand how IP_QUEUE works, we must first understand how network packets are stored in the kernel. Figure 2 shows the two primary data structures involved. `struct sk_buff` stores a single datagram fragment; the `data` field is a pointer to the packet payload. When a datagram is fragmented, it is stored in the kernel as a linked list of `sk_buff` structures. At every point in a packet's journey through the Linux kernel network stack, it is referenced by a pointer to its `sk_buff`.

That structure is ordinarily destroyed when the packet is sent on for delivery, dropped, or otherwise done with. So, when IP_QUEUE selects packets for deep filtering, it must make extra provisions for storing those packets. This necessitates an extra data structure to store references to queued packets while the kernel waits for the userland daemon to perform its processing task. The IP_QUEUE module maintains these references in a datastructure, named `ipq_queue_entry`.

```
struct sk_buff {
  ...
  struct sk_buff *next;
  unsigned char *data;
  ...
};
```

```
struct ipq_queue_entry {
  ...
  struct nf_info *info;
  struct sk_buff *skb;
};
```

Figure 2: Kernel data structures

1.2.2 Operation

Although IP_QUEUE provides all the functionality needed to perform deep packet filtering on GNU/Linux hosts, it is not currently suitable for all applications. Specifically, it is too slow for applications in which a high number of large packets must be quickly filtered. Under such conditions, packets cannot be processed quickly enough, leading to unacceptable slowdowns in service or triggering reliability protocol intervention (e.g., TCP retransmisssions) [2].

Figure 3 illustrates the operation of the queuing mechanism. When a packet arrives on a network interface, the kernel scans its `iptables` firewall rule set for the first rule that matches the packet. That rule may instruct the kernel to QUEUE the packet. Since the packet being queued is removed from the stream of network traffic, the IP_QUEUE module must keep it in a separate data structure called `ipq_queue_entry`.

1. The kernel then crafts a message to send to the queue processing daemon. This message contains the packet's header information along with the contents of the packet itself.

2. The message is sent to the daemon via the special socket connection.

3. The daemon reads the message, extracts and examines the packet information, and determines a verdict for the queued packet.

4. Finally, the daemon sends a socket message back to the kernel indicating its verdict.

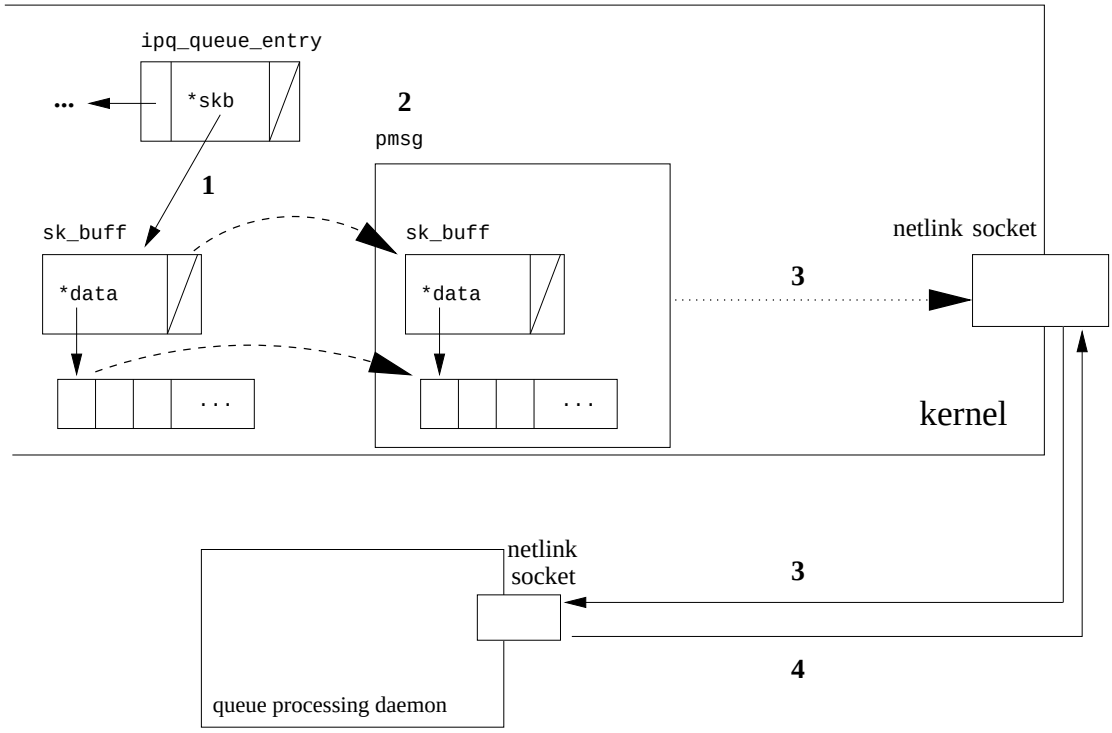

Figure 3: Operation of IP_QUEUE in base implementation

The kernel either accepts or rejects the packet based on the signal received on the socket connection.

Two things are worth noting here. First, the original packet has been removed from the stream of packets passing through the network host, but it remains stored in kernel memory in the IP_QUEUE module's list of queued packets. Second, the entire contents of the packet are copied into a new area of kernel memory before being pushed onto the NETLINK socket.

The latter operation is where the performance of the firewall is thought to suffer most [2]. This slowdown is enough to make IP_QUEUE's queueing scheme prohibitively expensive for some applications. In order to bring the power and utility of deep packet filtering to a wider variety of applications, the performance of IP_QUEUE must be improved. We describe our modification of the module in the next section.

2 Description of modified implementation

When performing deep packet filtering with the IP_QUEUE module there are essentially four steps taken for every packet received by the firewall. First, the kernel must signal to the userland daemon that a new packet is ready for processing. Second, the daemon must somehow gain access to that packet's contents. Third, the daemon must signal its verdict back to the kernel—whether to accept the packet, reject the packet, et cetera. Fourth, the modified contents of the packet must replace the original contents of the queued packet.

The established implementation of IP_QUEUE accomplishes all four tasks using socket communication. As outlined above, a socket message both alerts the queue processing daemon to the presence of a new packet and contains that packet's data for examination. The verdict is returned to the kernel via another socket message sent back to the kernel; a modified copy of the packet is also included in that message, when needed.

Our implementation modifies the second and fourth tasks. We do not copy the contents

of the incoming packet into a new data structure to be sent for processing via the socket connection. Instead, we only supply the incoming packet's address in kernel memory; the prototype implementation described in this paper uses a pre-set portion of static kernel memory to hold this address for retrieval by the userland daemon. A more practical final solution would be to include the address as part of the Netlink socket message.[1] The user program then directly accesses the packet's contents in kernel memory. The packet payload can then be both read and modified in situ.

Direct access to kernel memory is provided by the `/dev/kmem` pseudo-device, available in both the 2.4 and 2.6 Linux kernel branches. `/dev/kmem` is a character special device that can be opened by a root-privileged user with the `open()` system call. Kernel memory can then be inspected and modified directly using ordinary file operations (such as the `read()` and `write()` system calls). The cursor, corresponding to the position within an ordinary file, here corresponds to an address in kernel address. By passing an address to the `lseek64()` system call, one can access any portion of kernel memory.

Recalling that the kernel retains the original packet in memory while waiting for the queue processing daemon to return a verdict, we can use file operations on `/dev/kmem` to access it directly, rather than waiting to receive a copy of it. Figure 4 demonstrates the new implementation.

1. Once again, a reference to the packet is saved.
2. A message containing only packet meta-data is prepared.
3. The message sent to userland via the `netlink` socket connection.
4. The queue processing daemon uses the `/dev/kmem` to read the packet contents in kernel memory.
5. Finally, the daemon returns a verdict message by the usual method.

[1] Thanks to Michael L. Weissberger for his assistance in developing a test of this method.

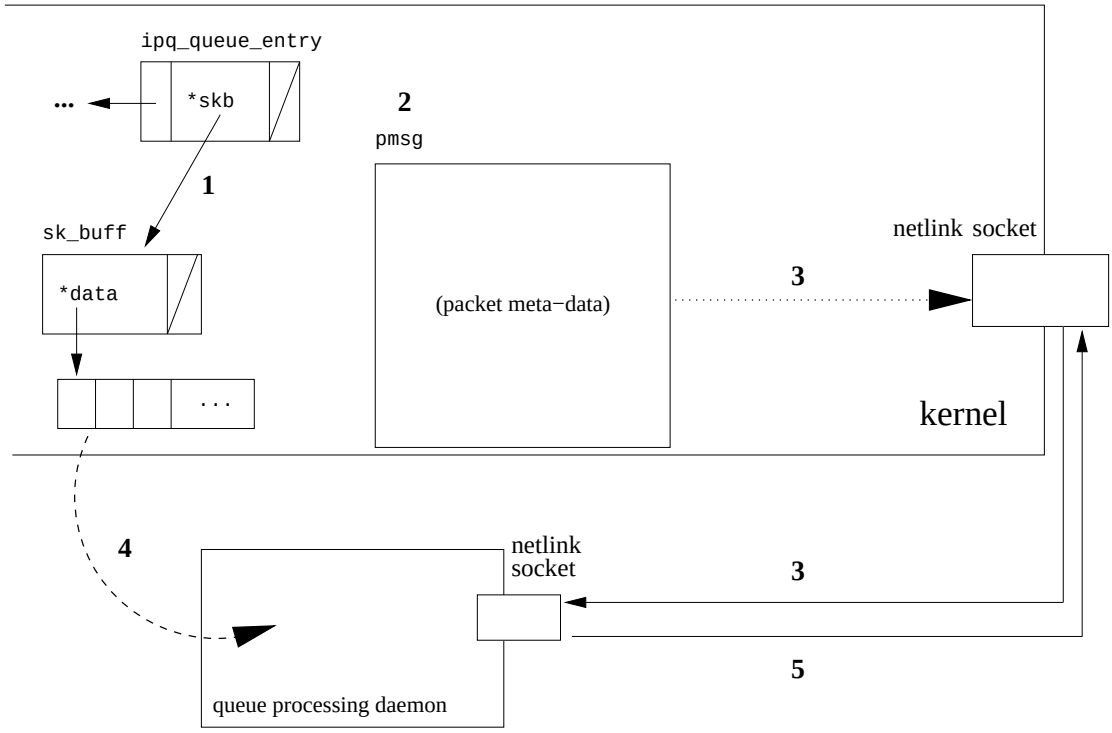

Figure 4: Operation of IP_QUEUE in new implementation

3 Discussion and evaluation

3.1 Compatibility

One advantage of this approach is that it carries few implications for existing queue processing daemon behavior. While any pre-existing queue processing daemon would need to be modified to use `/dev/kmem` (in addition to the changes in IP_QUEUE itself), the basic paradigm has not been changed; the modified daemon will still listen on a socket for a new packet, and the issuance of the verdict is unchanged. The daemon's basic structure of operation remains unchanged.

One risk of this implementation is its reliance on the availability of the `/dev/kmem` pseudo-device. Some see the existence of such a mechanism for directly accessing kernel memory as a security vulnerability that ought to be eliminated. For the time being however, it seems likely to remain part of Linux, since it is used by kernel developers for debugging. In any event, a user must have root privileges to access `/dev/kmem`; anyone user who already has root access can disrupt the system however she wishes, regardless of whether `/dev/kmem` is available or not.

3.2 Performance

Evaluating the performance of deep filtering schemes is challenging, because it depends greatly on the particular filtering application. As we explain below, the base (socket-based) implementation is sensitive to total datagram size. Our shared memory implemenation, by contrast, is sensitive to the size of the subset of packet data to be examined. Different filtering applications call for different portions of the filtered traffic to be examined. Applications that only require the daemon to examine a few bytes fare better under our particular shared memory implementation than applications calling for large portions of the datagram to be examined.

So, the independent variables for our tests were:

- The total size of the datagram
- The number of bytes in the diagram to be inspected—i.e., the number of "bytes of interest"
- The position of the bytes of interest within the datagram

Our test firewall machine contained a Pentium 4 processor running at 2.26 GHz. Performance of the different implementations was measured using the Pentium processor's built-in cycle counter, which counts the number of cycles elapsed since system boot time. By reading the value of this counter at different points in both the kernel and the queue processing daemon, we could find the time required for any portion of the IP_QUEUE module's operation.

Sample code can be found in the Appendix. All tests followed the same general procedure:

1. Set firewall rule (with `iptables`) to divert all incoming traffic destined for port 42 to IP_QUEUE.
2. Start queue processing daemon
3. Generate a number of packets from a remote network host.
4. Record the CPU cycle counter timing results produced by the kernel and/or the daemon (as appropriate) for each packet processed

3.2.1 Reading packets

For reading packets, we compared the times required to carry out the following steps:

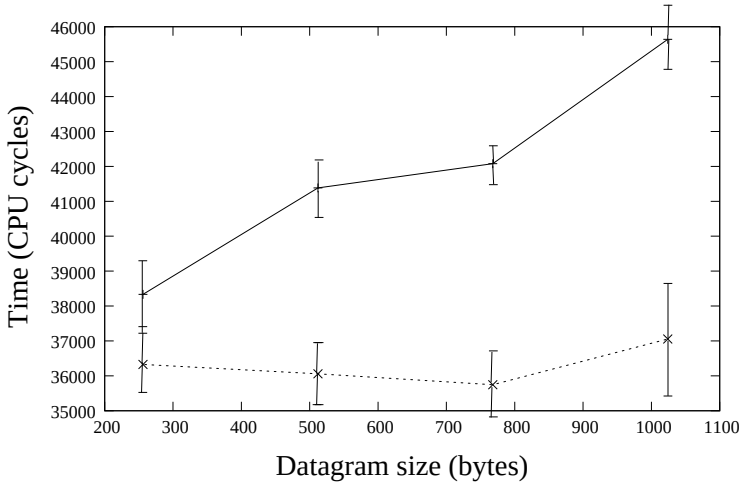

Figure 5: Read performance for varying datagram sizes

1. Prepare and send a Netlink message from the kernel to the user daemon indicating that a new packet has arrived. This message contains the entire packet contents in the original implementation; it contains only header information in our prototype.

2. Receive the message at the user daemon.

3. Read the contents of the packet payload, either in the Netlink message itself (base implementation) or directly in kernel memory (our implementation)

We tried varying both the size of the datagram to be examined while holding the bytes of interest constant (one byte of interest, located at the end of the datagram). The results are shown in Table 1 and Figure 5. We also tried holding the datagram size constant (at 1 kilobyte) and varying the location of the byte of interest. Those results are shown in Figure 2.

Compared with the base implementation, our shared memory implementation yields modest but substantial performance gains. As seen in Figure 5, the time complexity of the base implementation grows linearly with total datagram size. A larger datagram requires

Size (bytes)	Base implementation (CPU cycles)	New implementation (CPU cycles)
256	38333 ± 877.77	36325 ± 1006.7
512	41383 ± 740.92	36060 ± 1127.6
768	42077 ± 611.98	35747 ± 1184.5
1024	45639 ± 1171.9	37057 ± 1452.4

Table 1: Read performance for varying datagram sizes (95% confidence intervals)

Byte of interest	Time (base implementation)	Time (new implementation)
1st	43794 ± 997.04	36531 ± 1987.8
256th	44403 ± 1021.7	37017 ± 1114.9
512th	44743 ± 2560	36890 ± 1165.5
1024th	44019 ± 1133	36929 ± 1111.8

Table 2: Read performance for varying data positions (1KB datagram, 95% confidence intervals)

more data to be copied for transmission to the queue processing daemon. By contrast, our implementation's time complexity does not grow with datagram size (to a point—see below). When the datagram is unfragmented (that is, it resides in a single `sk_buff`) and the bytes of interest are contiguous, only a single file seek/read operation is required. The expected time complexity of a seek is constant as packet size grows (see Table 2).

However, the assumption that the packet will be unfragmented in kernel memory will not always hold. When a packet is fragmented (e.g., because it it is large), the daemon may have to traverse the linked list of `sk_buff`s to locate the bytes of interest. Every "hop" down the link list requires (1) a call to `lseek64()` to find the address of the `sk_buff`'s `next` field, (2) a `read()` operation to retrieve the address of the next fragment, and (3) an additional call to `lseek64()` to move the file cursor to that fragment. This operation can be performed quickly (3448 ± 81.97 CPU ticks, on average), but the number of "hops" required to process a particular packet will grow with packet fragmentation. So, given that larger packets are split among more `sk_buff` structures in the kernel, our implementation is actually linear in the size of the datagrams processed.

Base implementation	New implementation
40308 ± 1780.6	33229 ± 1930.9

Table 3: Packet modify-and-return performance comparison (1 kilobyte datagram)

This is not entirely discouraging, as we expect the bytes of interest to be a small contiguous subset of the total packet contents for some applications [2]. This will not always be the case, however, and the performance gains made by using shared kernel memory are partly lost to additional file input/output operations as the bytes of interest grow in number and spatial disparity. While we have demonstrated the potential performance gains of shared memory in our prototype, more effort will be required before it can offer an unequivocally faster solution under a wide variety of usage scenarios.

3.2.2 Writing packets

For writing packets, we measured the time required to carry out the following:

1. Change one byte of interest at the end of the packet payload—either the copy received via Netlink (in the base implementation) or directly in kernel memory (in our prototype)

2. Report an "accept" or "reject" verdict back to the kernel (in the base implementation, this step includes sending a modified copy of the packet back to the kernel along with the verdict)

In other words, we are measuring the time for a return trip for a packet from the queue processor back to the kernel, including the time required for the userland daemon to modify the packet's contents.

The results are given in Table 3. In addition to improving the speed with which packet data can be read by the queue processing daemon, using direct kernel memory access also

yields gains when modifying packets as they pass through the firewall host. In the original implementation, the queue processing daemon sends a modified payload back to the kernel along with its verdict. The kernel completely replaces the payload of the queued packet with that received from the processing daemon. These two steps of (1) sending a complete payload replacement via the Netlink socket and (2) copying the contents of into the kernel socket buffers can both be avoided using shared memory.

In our implementation, the queue processing daemon simply opens the `/dev/kmem` device with write ability, instead of read-only. Any byte of the payload can then be modified in situ. We tested the performance with 1 kilobyte datagrams and 1 byte of interest (see Table 3). Again, we expect this performance gain to shrink as the number and spatial disparity of the bytes of interest grow.

4 Conclusions

Deep packet filtering has many useful applications. The IP_QUEUE Linux kernel module provides tools to perform deep packet filtering on GNU/Linux operating systems, but its performance is not adequate for some applications. We have presented a faster prototypical implementation of deep packet inspection. By partly replacing IP_QUEUE's socket-based message passing scheme with one that uses shared kernel memory accessible by a privileged user packet processing daemon, we can speed up the operation of a deep packet filtering firewall.

We have demonstrated the value of shared memory as a practical method of user/kernel communication that is faster than using socket communication. Our prototypical implementation demonstrates some gains, but they are dependent upon the characteristics of the filtering application. There are almost certainly more performance gains to be made. Using `/dev/kmem` to access kernel memory spares the time required to copy packet contents between the kernel and the userland daemon, but it introduces new costs associated with

the file input/output operations.

Optimization of these file operations may be possible. Dedicating a contiguous segment of kernel memory for the storage of queued packets might reduce the amount of movement of the file position cursor, although this would require more extensive changes to the kernel than those presented here.

Casual experimentation suggested that using the `mmap()` system call to access kernel memory might be faster than calling file input/output operations. The costs of seeking within `/dev/kmem` are eliminated, and directly accessing kernel memory bytes as an array ought to be faster than using the `read()/write()` system calls. Unfortunately, the Linux kernel's internal functions that allow `mmap()`ing of the `/dev/kmem` pseudo-device to memory are in disrepair. Patching the kernel to fully enable this functionality is a promising avenue of future performance gains.

Finally, even more invasive changes to the IP_QUEUE module could be made that completely eliminate the netlink socket communication. For example, shared memory could be used for all aspects of userland/kernel communication, not merely the transmission of packet data; the userland queue processor could repeatedly poll a section of kernel memory for information about new packets (rather than reading that information in a socket message), access those packets using the methods presented here, then return a verdict using some other shared kernel memory data structure. That would require major changes to any queue processors already written, but the cost of rewriting them might be worthwhile for some applications. Further experimentation is needed.

References

[1] Young H. Cho and William H. Mangione-Smith. "Deep packet filtering with dedicated logic and read only memories." In Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE, 2004.

[2] James W. Deverick. "A Framework for Active Firewalls." `http://www.cs.wm.edu/~kearns/dissertations.d/jim\_abstract.html`. 2005.

[3] I. Dubrawsky. Firewall evolution - deep packet inspection. `http://online.securityfocus.com/infocus/1716`. 2003.

[4] Muralidaran Gangadharan and Kai Hwang. "Intranet security with micro-firewalls and mobile agents for proactive intrusion response". In Proceedings of the 2001 International Conference on Computer Networks and Mobile Computing, 2001.

[5] Paul A. Henry. "Protocol and application awareness: a new trend or an established tradition?" Information Systems Security, 12(6), January 2004.

[6] Kai Hwang and Muralidaran Gangadharan. "Micro-firewalls for dynamic network security with distributed intrusion detection." In Proceedings of the IEEE International Symposium on Network Computing and Applications, 2001.

[7] Sotiris Ioannidis, Angelos Keromytis, Steve Bellovin, and Johnathan Smith. "Implementing a distributed firewall". In ACM Computer and Communication Security, 2000.

[8] The Netfilter Project. `http://www.netfilter.org/`.

[9] Thomas Porter. "The Perils of Deep Packet Inspection". 11 January 2005. `http://www.securityfocus.com/infocus/1817`.

A Selected code from original implementation

A.0.3 net/ipv4/netfilter/ip_queue.c

```
static struct sk_buff *
ipq_build_packet_message(struct ipq_queue_entry *entry, int *errp)
{
    unsigned char *old_tail;
    size_t size = 0;
    size_t data_len = 0;
    struct sk_buff *skb;
    struct ipq_packet_msg *pmsg;
    struct nlmsghdr *nlh;

    read_lock_bh(&queue_lock);

    switch (copy_mode) {
    case IPQ_COPY_META:
    case IPQ_COPY_NONE:
       size = NLMSG_SPACE(sizeof(*pmsg));
       data_len = 0;
       break;

    case IPQ_COPY_PACKET:
       if (entry->skb->ip_summed == CHECKSUM_HW &&
          (*errp = skb_checksum_help(entry->skb,
                                entry->info->outdev == NULL))) {
             read_unlock_bh(&queue_lock);
             return NULL;
       }
       if (copy_range == 0 || copy_range > entry->skb->len)
          data_len = entry->skb->len;
       else
          data_len = copy_range;
          size = NLMSG_SPACE(sizeof(*pmsg) + data_len);
          break;

    default:
        *errp = -EINVAL;
        read_unlock_bh(&queue_lock);
        return NULL;
    }
```

```
read_unlock_bh(&queue_lock);

skb = alloc_skb(size, GFP_ATOMIC);
if (!skb)
    goto nlmsg_failure;

old_tail= skb->tail;
nlh = NLMSG_PUT(skb, 0, 0, IPQM_PACKET, size - sizeof(*nlh));
pmsg = NLMSG_DATA(nlh);
memset(pmsg, 0, sizeof(*pmsg));

pmsg->packet_id       = (unsigned long )entry;
pmsg->data_len        = data_len;
pmsg->timestamp_sec   = entry->skb->tstamp.off_sec;
pmsg->timestamp_usec  = entry->skb->tstamp.off_usec;
pmsg->mark            = entry->skb->nfmark;
pmsg->hook            = entry->info->hook;
pmsg->hw_protocol     = entry->skb->protocol;

if (entry->info->indev)
    strcpy(pmsg->indev_name, entry->info->indev->name);
else
    pmsg->indev_name[0] = '\0';

if (entry->info->outdev)
    strcpy(pmsg->outdev_name, entry->info->outdev->name);
else
    pmsg->outdev_name[0] = '\0';

if (entry->info->indev && entry->skb->dev) {
    pmsg->hw_type = entry->skb->dev->type;
    if (entry->skb->dev->hard_header_parse)
        pmsg->hw_addrlen =
            entry->skb->dev->hard_header_parse(entry->skb,
                                pmsg->hw_addr);
}

if (data_len)
    if (skb_copy_bits(entry->skb, 0, pmsg->payload, data_len))
        BUG();

nlh->nlmsg_len = skb->tail - old_tail;
```

```
    return skb;

nlmsg_failure:
    if (skb)
        kfree_skb(skb);
    *errp = -EINVAL;
    printk(KERN_ERR "ip_queue: error creating packet message\n");
    return NULL;
}

static int
ipq_enqueue_packet(struct sk_buff *skb, struct nf_info *info,
   unsigned int queuenum, void *data)
{
    int status = -EINVAL;
    struct sk_buff *nskb;
    struct ipq_queue_entry *entry;

    if (copy_mode == IPQ_COPY_NONE)
        return -EAGAIN;

    entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
    if (entry == NULL) {
        printk(KERN_ERR "ip_queue: OOM in ipq_enqueue_packet()\n");
        return -ENOMEM;
    }

    entry->info = info;
    entry->skb = skb;

    nskb = ipq_build_packet_message(entry, &status);
    if (nskb == NULL)
        goto err_out_free;

    write_lock_bh(&queue_lock);

    if (!peer_pid)
        goto err_out_free_nskb;

    if (queue_total >= queue_maxlen) {
        queue_dropped++;
        status = -ENOSPC;
```

```
        if (net_ratelimit())
          printk (KERN_WARNING "ip_queue: full at %d entries, "
                  "dropping packets(s). Dropped: %d\n", queue_total,
                  queue_dropped);
        goto err_out_free_nskb;
    }

    /* netlink_unicast will either free the nskb or attach it to a socket */
    status = netlink_unicast(ipqnl, nskb, peer_pid, MSG_DONTWAIT);
    if (status < 0) {
        queue_user_dropped++;
        goto err_out_unlock;
    }

    __ipq_enqueue_entry(entry);

    write_unlock_bh(&queue_lock);
    return status;

err_out_free_nskb:
    kfree_skb(nskb);

err_out_unlock:
    write_unlock_bh(&queue_lock);

err_out_free:
    kfree(entry);
    return status;
}

static int
ipq_mangle_ipv4(ipq_verdict_msg_t *v, struct ipq_queue_entry *e)
{
    int diff;
    struct iphdr *user_iph = (struct iphdr *)v->payload;

    if (v->data_len < sizeof(*user_iph))
        return 0;
    diff = v->data_len - e->skb->len;
    if (diff < 0)
        skb_trim(e->skb, v->data_len);
    else if (diff > 0) {
```

```
        if (v->data_len > 0xFFFF)
            return -EINVAL;
        if (diff > skb_tailroom(e->skb)) {
            struct sk_buff *newskb;

            newskb = skb_copy_expand(e->skb,
                        skb_headroom(e->skb),
                        diff,
                        GFP_ATOMIC);
            if (newskb == NULL) {
                printk(KERN_WARNING "ip_queue: OOM "
      "in mangle, dropping packet\n");
                return -ENOMEM;
            }
            if (e->skb->sk)
                skb_set_owner_w(newskb, e->skb->sk);
            kfree_skb(e->skb);
            e->skb = newskb;
        }
        skb_put(e->skb, diff);
}
    if (!skb_make_writable(&e->skb, v->data_len))
        return -ENOMEM;
    memcpy(e->skb->data, v->payload, v->data_len);
    e->skb->ip_summed = CHECKSUM_NONE;

    return 0;
}
```

A.0.4 Queue processing daemon

```
/* Based on "The quick intro to libipq"
http://www.crhc.uiuc.edu/~grier/projects/libipq.html */

#include <linux/netfilter.h>
#include <netinet/in.h>
#include "libipq.h"
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <sys/time.h>

#define BUFSIZE 2048

static void die(struct ipq_handle *h)
{
ipq_perror("passer");
ipq_destroy_handle(h);
exit(1);
}

/* Read Timer Stamp Counter */
/* From http://www.cs.wm.edu/~kearns/001lab.d/rdtsc.html */
unsigned long long int rdtsc(void)
{
unsigned long long int x;
__asm__ volatile(".byte 0x0f,0x31" : "=A" (x));
return x;
}

int main(int argc, char **argv)
{
    int status;
    unsigned char buf[BUFSIZE];
    struct ipq_handle *h;

    struct sched_param sched_parameters;

    unsigned long long int start_time = 0;
    unsigned long long int curr_time = 0;
```

```
/* SET SCHEDULING PRIORITY */
sched_parameters.sched_priority = 99;

status = sched_setscheduler(0,SCHED_RR,&sched_parameters);
if (status != 0) {
    perror("Set scheduler error: ");
    exit(1);
} else {
    status = setpriority(PRIO_PROCESS,0,-20);
    if (status != 0)
        perror("setpriority: ");

    status = getpriority(PRIO_PROCESS,0);
    printf("Scheduling priority: %d\n", status);

    status = sched_getscheduler(0);
    printf("Scheduling status: %d\n", status);
}

/* PREPARE FOR NETLINK COMMUNICATION */
h = ipq_create_handle(0, PF_INET);
if (!h)
    die(h);

status = ipq_set_mode(h, IPQ_COPY_PACKET, BUFSIZE);
if (status < 0)
    die(h);

do{
    status = ipq_read(h, buf, BUFSIZE, 0);
    if (status < 0)
        die(h);

    switch (ipq_message_type(buf)) {
        case NLMSG_ERROR:
            fprintf(stderr, "Received error message %d\n",
            ipq_get_msgerr(buf));
            break;

        case IPQM_PACKET: {
            ipq_packet_msg_t *m = ipq_get_packet(buf);
```

```
                /* TIMING */
                start_time = *(unsigned long long int *) m->payload;
                curr_time  = rdtsc();

                // PRINT KERNEL MEMORY CONTENTS
                //printf("Kernel memory contents:\n");
                printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
                int j;
                start_time = rdtsc();
                for (j = 256; j < (int) m->data_len; j++) {
                    printf("%x ", m->payload[j]);
                    if (m->payload[j] == 'X') {
                        printf("Byte 1f found at index %d\n",j);
                        m->payload[j] = 'R';
                        break;
                    }
                }
                printf("Time diff: %llu\n", rdtsc() - start_time);

                printf("Issuing NF_ACCEPT verdict.\n\n");
                printf("Sending verdict at %llu\n",rdtsc());
                status = ipq_set_verdict(h, m->packet_id, NF_ACCEPT, m->data_len, buf
                if (status < 0) {
                    die(h);
                    break;
                }

                /* TIMING */
                break;
            }
            default:
                fprintf(stderr, "Unknown message type!\n");
                break;
        } // switch
    } while (1);

    ipq_destroy_handle(h);
    return 0;
} // main
```

B Selected code for new implementation

B.0.5 net/ipv4/netfilter/ip_queue.c

```
static struct sk_buff *
ipq_build_packet_message(struct ipq_queue_entry *entry, int *errp)
{
    unsigned char *old_tail;
    size_t size = 0;
    size_t data_len = 0;
    struct sk_buff *skb;
    struct ipq_packet_msg *pmsg;
    struct nlmsghdr *nlh;

    read_lock_bh(&queue_lock);

    switch (copy_mode) {
    case IPQ_COPY_META:
    case IPQ_COPY_NONE:
        size = NLMSG_SPACE(sizeof(*pmsg));
        data_len = 0;
        break;

    case IPQ_COPY_PACKET:
        if (entry->skb->ip_summed == CHECKSUM_HW &&
            (*errp = skb_checksum_help(entry->skb,
                                entry->info->outdev == NULL))) {
            read_unlock_bh(&queue_lock);
            return NULL;
        }
        if (copy_range == 0 || copy_range > entry->skb->len)
            data_len = entry->skb->len;
        else
            data_len = copy_range;

        size = NLMSG_SPACE(sizeof(*pmsg) + data_len);
        break;

    default:
        *errp = -EINVAL;
        read_unlock_bh(&queue_lock);
        return NULL;
    }
```

```
read_unlock_bh(&queue_lock);

skb = alloc_skb(size, GFP_ATOMIC);
if (!skb)
    goto nlmsg_failure;


old_tail= skb->tail;
nlh = NLMSG_PUT(skb, 0, 0, IPQM_PACKET, size - sizeof(*nlh));
pmsg = NLMSG_DATA(nlh);
memset(pmsg, 0, sizeof(*pmsg));

pmsg->packet_id       = (unsigned long )entry;
pmsg->data_len        = data_len;
pmsg->timestamp_sec   = entry->skb->tstamp.off_sec;
pmsg->timestamp_usec  = entry->skb->tstamp.off_usec;
pmsg->mark            = entry->skb->nfmark;
pmsg->hook            = entry->info->hook;
pmsg->hw_protocol     = entry->skb->protocol;

/* WWC */
//memcpy(&pmsg->mem_location,&entry->skb->data,4);

memcpy(pmsg->mem_location,&entry->skb->data,sizeof(int));
//pmsg->mem_location = (void *) entry->skb->data;

if (entry->info->indev)
    strcpy(pmsg->indev_name, entry->info->indev->name);
else
    pmsg->indev_name[0] = '\0';

if (entry->info->outdev)
    strcpy(pmsg->outdev_name, entry->info->outdev->name);
else
    pmsg->outdev_name[0] = '\0';

if (entry->info->indev && entry->skb->dev) {
    pmsg->hw_type = entry->skb->dev->type;
    if (entry->skb->dev->hard_header_parse)
        pmsg->hw_addrlen =
            entry->skb->dev->hard_header_parse(entry->skb,
```

```
                                    pmsg->hw_addr);
    }

    /* WWC */
    //WWC_PACKET_LOCATION = (int) entry->skb->data;
    WWC_PACKET_LOCATION = (int) entry->skb;

    //pmsg->mem_location = WWC_PACKET_LOCATION;
    //pmsg->mem_location = 0;
    printk(KERN_WARNING "WWC_PACKET_LOCATION: %x -> %x\n", (unsigned int) &WWC_PACKET

    unsigned long long int timestamp = rdtsc2();
    memcpy(entry->skb->data, &timestamp, sizeof(timestamp));
    //printk(KERN_WARNING "NL START: %llu\n",timestamp);

    if (data_len) {
        /* WWC */

        //memcpy(entry->skb->data + sizeof(timestamp), &entry->skb->data, sizeof(entr

        if (skb_copy_bits(entry->skb, 0, pmsg->payload, data_len))
             BUG();


    nlh->nlmsg_len = skb->tail - old_tail;
    return skb;

nlmsg_failure:
    if (skb)
        kfree_skb(skb);
    *errp = -EINVAL;
    printk(KERN_ERR "ip_queue: error creating packet message\n");
    return NULL;
}

static int
ipq_enqueue_packet(struct sk_buff *skb, struct nf_info *info,
   unsigned int queuenum, void *data)
{
    int status = -EINVAL;
    struct sk_buff *nskb;
    struct ipq_queue_entry *entry;
```

```
if (copy_mode == IPQ_COPY_NONE)
    return -EAGAIN;

entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
if (entry == NULL) {
    printk(KERN_ERR "ip_queue: OOM in ipq_enqueue_packet()\n");
    return -ENOMEM;
}

entry->info = info;
entry->skb = skb;

nskb = ipq_build_packet_message(entry, &status);

if (nskb == NULL)
    goto err_out_free;

write_lock_bh(&queue_lock);

if (!peer_pid)
    goto err_out_free_nskb;

if (queue_total >= queue_maxlen) {
    queue_dropped++;
    status = -ENOSPC;
    if (net_ratelimit())
              printk (KERN_WARNING "ip_queue: full at %d entries, "
              "dropping packets(s). Dropped: %d\n", queue_total,
              queue_dropped);
    goto err_out_free_nskb;
}

/* netlink_unicast will either free the nskb or attach it to a socket */
status = netlink_unicast(ipqnl, nskb, peer_pid, MSG_DONTWAIT);
if (status < 0) {
    queue_user_dropped++;
    goto err_out_unlock;
}

__ipq_enqueue_entry(entry);
```

```
    write_unlock_bh(&queue_lock);
    return status;

err_out_free_nskb:
    kfree_skb(nskb);

err_out_unlock:
    write_unlock_bh(&queue_lock);

err_out_free:
    kfree(entry);
    return status;
}

static int
ipq_mangle_ipv4(ipq_verdict_msg_t *v, struct ipq_queue_entry *e)
{
    int diff;
    struct iphdr *user_iph = (struct iphdr *)v->payload;

    if (v->data_len < sizeof(*user_iph))
        return 0;
    diff = v->data_len - e->skb->len;
    if (diff < 0)
        skb_trim(e->skb, v->data_len);
    else if (diff > 0) {
        if (v->data_len > 0xFFFF)
            return -EINVAL;
        if (diff > skb_tailroom(e->skb)) {
            struct sk_buff *newskb;

            newskb = skb_copy_expand(e->skb,
                                     skb_headroom(e->skb),
                                     diff,
                                     GFP_ATOMIC);
            if (newskb == NULL) {
                printk(KERN_WARNING "ip_queue: OOM "
                       "in mangle, dropping packet\n");
                return -ENOMEM;
            }
            if (e->skb->sk)
                skb_set_owner_w(newskb, e->skb->sk);
```

```
            kfree_skb(e->skb);
            e->skb = newskb;
        }
        skb_put(e->skb, diff);
    }
    if (!skb_make_writable(&e->skb, v->data_len))
        return -ENOMEM;
    memcpy(e->skb->data, v->payload, v->data_len);
    e->skb->ip_summed = CHECKSUM_NONE;

    return 0;
}
```

B.0.6 Queue processing daemon

```
/* Based on "The quick intro to libipq"
http://www.crhc.uiuc.edu/~grier/projects/libipq.html */

#define _LARGEFILE64_SOURCE
#include <linux/netfilter.h>
#include <netinet/in.h>
#include "libipq.h"
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <sys/types.h>

#define BUFSIZE 4096
#define KERNEL_ADDRESS_LOC 0xe11cde00

static void die(struct ipq_handle *h)
{
    ipq_perror("passer");
    ipq_destroy_handle(h);
    exit(1);
}

/* Read Timer Stamp Counter */
/* From http://www.cs.wm.edu/~kearns/001lab.d/rdtsc.html */
unsigned long long int rdtsc(void)
{
    unsigned long long int x;
    __asm__ volatile(".byte 0x0f,0x31" : "=A" (x));
    return x;
}

int main(int argc, char **argv)
{
    int status;
    unsigned char buf[BUFSIZE];
    unsigned char buf2[BUFSIZE];

    struct ipq_handle *h;
```

```
int fd;

char outbuf[BUFSIZE];
outbuf[0] = 'X';

unsigned long long int desired_offset = 0;
unsigned long long int actual_offset = 0;

struct sched_param sched_parameters;

//unsigned long long int start_time = 0;
unsigned long long int curr_time = 0;
unsigned long long int file_start = 0;
unsigned long long int file_end = 0;

int nextptr = 0;

/* SET SCHEDULING PRIORITY */
sched_parameters.sched_priority = 99;

status = sched_setscheduler(0,SCHED_RR,&sched_parameters);
if (status != 0) {
    perror("Set scheduler error: ");
    exit(1);
} else {
    status = setpriority(PRIO_PROCESS,0,-20);
    if (status != 0)
        perror("setpriority: ");

    status = getpriority(PRIO_PROCESS,0);
    printf("Scheduling priority: %d\n", status);

    status = sched_getscheduler(0);
    printf("Scheduling status: %d\n", status);
}

/* PREPARE FOR NETLINK COMMUNICATION */
h = ipq_create_handle(0, PF_INET);
if (!h)
    die(h);

status = ipq_set_mode(h, IPQ_COPY_META, BUFSIZE);
```

```
if (status < 0)
    die(h);

/* OPEN KERNEL MEMORY DEVICE */
fd = open("/dev/kmem",O_RDWR);

do {
    status = ipq_read(h, buf, BUFSIZE, 0);
    if (status < 0)
        die(h);

    switch (ipq_message_type(buf)) {
        case NLMSG_ERROR:
            fprintf(stderr, "Received error message %d\n",
            ipq_get_msgerr(buf));
            break;

        case IPQM_PACKET: {
            ipq_packet_msg_t *m = ipq_get_packet(buf);

            /* READ MEMORY ADDRESS (MLW) */
            file_start = rdtsc();
            lseek64(fd,(unsigned long long int) KERNEL_ADDRESS_LOC, SEEK_SET);
            read(fd,&desired_offset,sizeof(int));
            file_end = rdtsc();

            /* READ KERNEL MEMORY */
            file_start = rdtsc();
            lseek64(fd,(unsigned long int) desired_offset + 256,SEEK_SET);
            status = write(fd,outbuf,1);

            if (status < 0) {
                perror("read: ");
                //exit(1);
            }

            // PRINT KERNEL MEMORY CONTENTS
            printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
            printf("Kernel memory contents:\n");
            int j;
            for (j = 0; j < (int) 2048; j++) {
                printf("%x ", buf2[j]);
```

```
                    if (buf2[j] == 'X') {
                        printf("\nByte X found at index %d\n",j);
                        break;
                    }
                }
                printf("\n");

                /* PRINT TIMINIG RESULTS */
                //printf("NL_END (corrected): %llu\n",curr_time - (file_end - file_st
                /*
                printf("Time diff: %llu\n", curr_time - start_time);
                printf("File time: %llu\n",file_end - file_start);
                printf("Non-file time: %llu\n",(curr_time - start_time) - (file_end -
                */

                //printf("Issuing NF_ACCEPT verdict.\n\n");
                status = ipq_set_verdict(h, m->packet_id, NF_ACCEPT, 0, NULL);
if (status < 0) {
                    die(h);
                    break;
                }
                file_end = rdtsc();

                printf("Adjusting packet contents took %llu\n",file_end - file_start)

                break;
            }
            default:
                fprintf(stderr, "Unknown message type!\n");
                break;
        } // switch
    } while (1);

    ipq_destroy_handle(h);
    return 0;
} // main
```