Bell Laboratories

subject: **PF_IPOPTION: A Kernel Extension for IP Option Packet Processing**
Work Project No. MA30034004
File Case 20564

date: **June 15, 2000**

from: **Ping Pan**
Dept. 10009669
HO 4F-506A
+1 732 332 6744
pingpan@bell-labs.com
10009669-000615-02TM

Henning Schulzrinne
Columbia University
Dept. Computer Science
NY, NY 10027
+1 212 939 7042
hgs@cs.columbia.edu

Technical Memorandum

Existing UNIX kernels cannot easily deliver IP packets containing IP options to applications. To address this problem, we have defined and implemented a new protocol family called `PF_IPOPTION` for the FreeBSD kernel. We have verified the implementation with some of the commonly used IP options. Measurements in kernel and user space showed that BSD socket I/O is the performance bottleneck.

1 Introduction

IPv4 packets can carry optional information in IP header extensions [1] . IP options are inspected by all network routers. Some of the commonly used IP options include source routing, timestamp and router alert [2] . The router alert (RA) option asks transit routers to intercept and process packets that are not addressed to them directly. However, existing UNIX-based platforms do not offer a clean mechanism to capture IP option packets, as shown by investigating some of the possible approaches:

`PF_INET` **raw sockets:** The *socket()* call allows a user process to create a network I/O interface that delivers network packets to a user process. A socket is described by three properties, namely protocol family, type and protocol. For example, to intercept RSVP PATH messages, a user process uses family `PF_INET`, type `SOCK_RAW` and protocol `IPPROTO_RSVP`. However, these parameters cannot restrict the delivery of packets to those containing IP options. For example, RSVP packets are captured only because the IP protocol type is RSVP (46), not because they may contain router alert options. In the current BSD implementation, the IP input routine delivers all incoming RSVP messages to the raw socket bound to the `IP-PROTO_RSVP`, regardless of the IP option type or whether packet contains IP options.

Another problem with using this approach is that, in some versions of BSD [3, 4] , raw sockets cannot intercept TCP or UDP packets. Hence, there is no way for these systems to capture IP option packets carrying TCP or UDP transport layers.

BPF **[5] and *libpcap* [6] :** BPF and libpcap allow nodes to selectively capture packets from data links. However, the *libpcap* capture process is non-intrusive, meaning that a copy of the packet is delivered to the application, while another is forwarded normally. For example, in the case of Router Alert, the IP input routine in the kernel needs to capture and pass up the IP option packets to the user space, without forwarding it to the next hop. Otherwise, the end user will receive multiple copies of the same message, namely the original message and modified copies generated by the transit routers.

Another problem with *libpcap* is that all packets are captured at the link-layer. Thus, the user process needs to parse link-layer protocols before it can read IP option headers. The kernel has already parsed the link layer, resulting in needless duplication of work.

divert **[7] :** *Divert* sockets enable routers and end systems to intercept packets using filters in the kernel. In this approach, users create several filters on IP options, and open *divert* sockets to intercept the option packets. However, the filters are tightly coupled with the firewall table in the kernel. This causes two potential problems: first, we need to make sure that the IP option filters do not cause security risks. For instance, the IP option filters can only be applied after all firewall filters are completed. Second, the firewall table is designed to read and check multiple packet header fields. The filter lookup routine can be very CPU intensive. We argue that setting up firewall filters for IP options impacts performance without providing additional functionality.

In conclusion, we need a new and cleaner solution to process IP option packets when received by routers.

1.1 Design Alternatives

There are many ways to change the kernel to intercept option packets. First, we can define a new socket type that is dedicated to handling IP option packets. One example on such approach is the `SOCK_PACKET` socket type in Linux, which was designed to intercept packets base on the LLC ether-types[1] . However, in BSD, the socket type is mainly used to describe socket transport behavior, for example reliable or stream. Thus, we considered defining a new socket type as orthogonal to our goal.

We can also implement a new virtual interface, and attach it to the physical interface. For all packets going through this virtual interface, a routine is added to intercept IP option packets. This is the design principle behind BPF. However, IP message processing has little to do with physical

[1] Linux has declared this socket type as obsolete.

interfaces. It is simpler to process packets within the IP input routine than processing packets at interface input, as it avoids having to attach virtual interfaces to each physical interface.

Another idea is to define a new protocol type for IP option packets, similar to the one that has used by *divert*. On receive, we redirect packets containing IP options to a user socket. However, this approach requires significant changes in the IP protocol processing functions.

This leaves the addition of a new protocol family. The new family provides a new, simple and clean socket interface between the user process and the kernel.

Below, we describe the design, the kernel implementation and the performance of the new socket protocol family.

2 `PF_IPOPTION`: a new protocol family

We have defined and implemented a new protocol family, `PF_IPOPTION`, that allows users to intercept IP packet with options via the socket interface. In this section, we explain how users can program sockets with the new family.

To receive IP packets with option type, *type*, a user needs to open a raw socket[2].

```
socket(PF_IPOPTION, SO_RAW, type);
```

Users can modify the `PF_IPOPTION` sockets with the following options:

***IPOPT_RECVLOCAL*:** If this option is set, the kernel sends copies of local traffic to the process. (Local traffic consists of packets originating at the node or addressed to the node.) We allow users to "listen" for local packets but not to intercept them. If not set, local traffic will bypass the `PF_IPOPTION` sockets.

***IPOPT_RECVRA*:** If this option is set, the kernel captures all incoming IP packets with IP router alert options.

***IPOPT_RESVRSVP*:** If this option is set, the kernel captures incoming RSVP packets that have a router alert option in the IP header.

***IPOPT_RECVRTCP*:** If this option is set, the kernel captures incoming RTCP messages that have a router alert option in the IP header.

***IP_RECVIF*:** If this option is set, the kernel captures the interface index indicating the interface on which the IP packet was received. This option was originally developed by Bill Fenner for UDP in FreeBSD. Ingress interface information is important when processing signaling protocol packets.

The `IPOPTION` extension works only for packets received. To send packets containing IP options, programs can create raw sockets and set the *IP_HDRINCL* socket option.

[2]We use a raw socket because its handling is simple. To intercept IP option packets, it does not make sense to run *bind()* or *connection()*, which is required in stream and datagram sockets. But, using raw sockets restricts the use to processes running as root.

2.1 *ipodump*: an IP Option Display Program

We illustrate the usage of `PF_IPOPTION` by describing the operation of a new UNIX command, *ipodump*. *ipodump* is a utility that displays IP option headers of packets passing through a system. Its manual page is in Appendix A. The source code can be found at http://www.cs.columbia.edu/˜pingpan/software/ipodump.

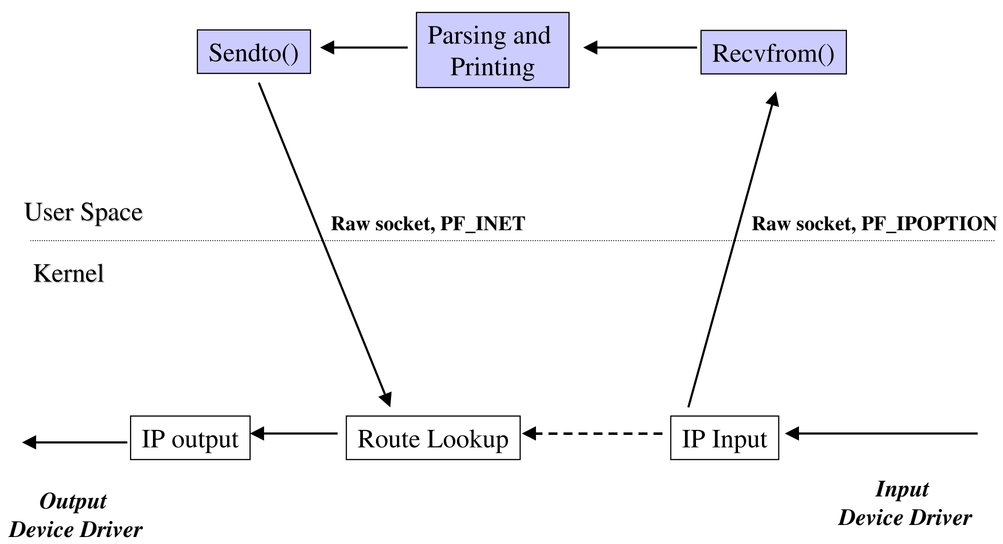

Figure 1: Packet forwarding path used by *ipodump*. Regular packets go directly from IP Input to route lookup (dotted line); intercepted packets are sent to user space, and reinjected back into the kernel after processing.

Figure 1 shows the data path of intercepted option packets. *ipodump* consists of three parts, shown as shaded boxes, for intercepting packets, printing headers, and packet reinjection. The program can be configured from the command line to choose, for example, the IP option type and incoming interface. For example, when asked to intercept IP RSVP packets with router-alert options, *ipodump* executes roughly the following code, omitting details:

```
int sock, on=1;

sock = socket(PF_IPOPTION, SOCK_RAW, IPOPT_RA);
setsockopt(sock, IPPROTO_IP, IP_RECVRSVP, &on, sizeof(on));
...
```

The packet receiving and reinjection procedure is similar to other socket applications. When receiving a packet, the function invokes the *recvmsg* system call, which requires the user to provide the receiving control and data buffer pointers. The reason for using *recvmsg* is to be able to receive control messages, such as ingress interface data.

```
struct iovec iov;              /* provide data buffer info */
struct cmsghdr *cmsg;          /* pointer to control message */
char *packet = (char *)malloc(MAX_PKT_SIZE);
char *ctrl = (char *)malloc(MAXCTRLSIZE);
...
iov.iov_base = (char *)packet;
iov.iov_len = MAX_PKT_SIZE;
msg.msg_iov = &iov;
msg.msg_control = ctrl;
msg.msg_controllen = MAXCTRLSIZE;
...
len = recvmsg(sock, &msg, 0);
for (cmsg = CMSG_FIRSTHDR(&msg); cmsg != NULL;
  cmsg = CMSG_NXTHDR(&msg, cmsg)) {
      if (cmsg->cmsg_type == IP_RECVIF)
           if_index = CMSG_IFINDEX(cmsg);
      ...
}

rsvp_read(packet, len, if_index);  /* process RSVP message */
...
```

After parsing and printing, *ipodump* sends the packet back to the kernel using a raw socket. The socket option `IP_HDRINCL` allows the caller to supply a completed IP header so that the kernel does not have to construct an IP header. It is used here to reinject the previously received raw IP packets.

```
int tx_sock, on=1;

/* initialize the transmit socket */
tx_sock = socket(PF_INET, SOCK_RAW, 0);
setsockopt(tx_sock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
...
/* send out the packet */
sendto(tx_sock, packet, len, 0, ...);
...
```

3 Implementation

In this section, we describe how we implemented `PF_IPOPTION` in the BSD kernel. Figure 2 shows where the `PF_IPOPTION` extension is inserted into the IP input processing path.

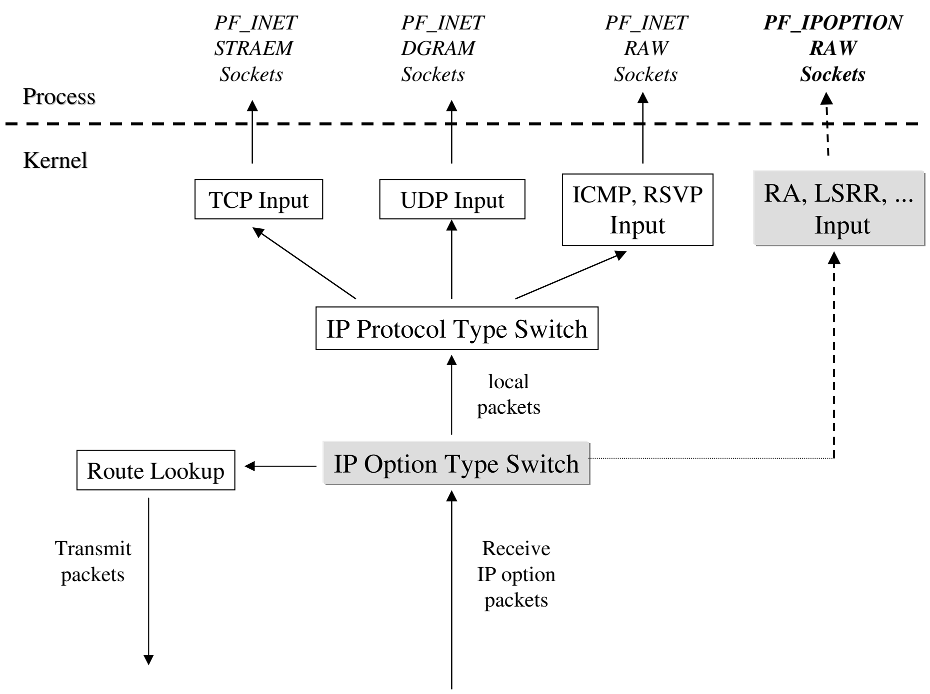

Figure 2: Relationship of IPOPTION processing to rest of kernel

One of our main objectives is to introduce as little processing overhead in the forwarding path as possible. To achieve this, we have used two bitmap variables: *ipopt_cfg* and *ipopt_pkt*. Users choose the type of IP option packets that they want to receive by calling the *setsockopt()* system function, which invokes *ipopt_ctloutput()* in the kernel. Based on the selected option type, *ipopt_ctloutput()* sets a corresponding bit in *ipopt_cfg*. Only one bit is set in the bitmap even if multiple sockets are opened to "listen" on the same option type. When receiving an IP options packet, the IP input routine records the options present in the packet and sets the corresponding bits in *ipopt_pkt*. Then, the IP option processing function can quickly determine whether or not to intercept the packet with one "and" operation (that is, *ipopt_cfg*&*ipopt_pkt*).

To further improve performance, we allocate one PCB (Protocol Control Block) chain per option type, instead of the traditional approach, where one PCB chain is responsible for keeping track of all the socket id's of a socket type or a socket family. Readers can refer to Stevens' UNIX TCP/IP book [3] that describe the operation of PCBs in detail.

After the kernel sends an intercepted packet to user space, it has to decide whether to drop or forward the original packet. If the system is operating as a router, the user process is responsible for processing and forwarding the intercepted packet. Therefore, the kernel needs to discard the

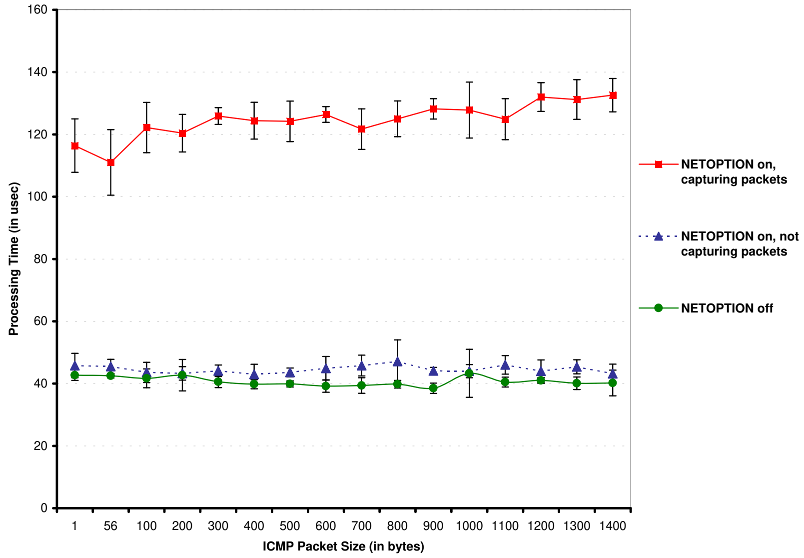

Figure 3: Kernel Packet processing time

original packet. But if the system is the packet's destination, there may be other sockets waiting for the arrival of the packet. In this case, the kernel cannot drop the packet and has to continue to process the packet. In our BSD implementation, we inserted IP option processing routine in two places: one is right before IP route lookup, the other just before IP protocol type switching. The first insertion is to catch routed packets, and the second insertion is to catch local packets. We always drop the routed packets after interception, and forward the local ones.

4 Performance

We evaluated the performance of our implementation on an Intel Celeron 500 MHz PC running FreeBSD 3.4. This PC was connected to two other PC's through 10 Mb/s Ethernet interfaces; we confirmed that it could route traffic between the interfaces at media speed. We used the *ping -R* command to generate record route IP option packets.

We modified the *ipodump* command such that it reinjects IP option packets back into the network immediately after intercepting them. We also added two timing checks in the kernel, one at the beginning of the IP input, the other at the end of IP output routine. Using this timing information, we could measure the overall network processing delay caused by using `PF_IPOPTION` sockets.

The results of our measurements are shown in Figure 3. The average packet processing time for a kernel without `PF_IPOPTION` implementation was 40.74 μs. When using `PF_IPOPTION`, the

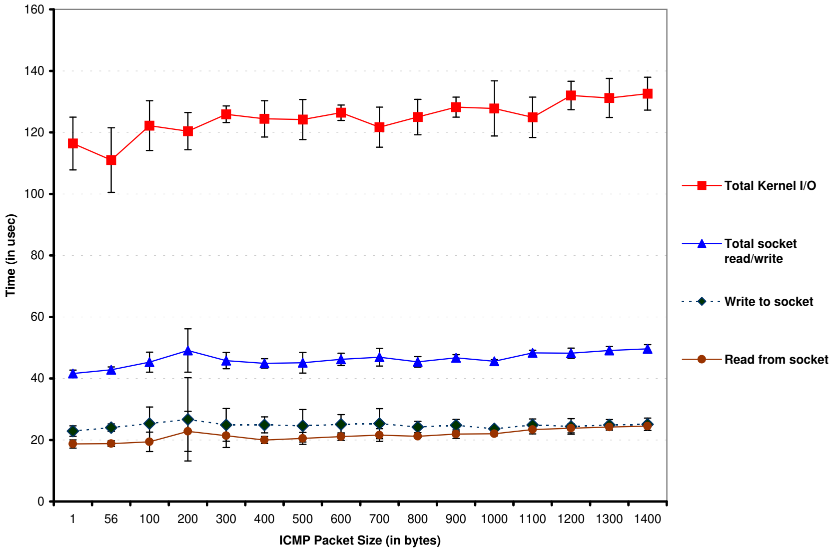

Figure 4: Comparing the kernel and user processing overhead

time for processing option packets, but not intercepting them, was 44.59 μsec on average. Thus, the extra checks for IP option packets taking place in the IP forwarding loop caused delay to increase by 8.6%. However, an intercepted packet spent 124.64 μs traversing kernel and user space, for a 67.3% increase in delay. In this mode, the system could only forward at most 8,023 IP option packets per second.

We observed that the processing delay did not depend significantly on the packet size. Out of the 16 sets of data we have collected, the delay for processing 1-byte ICMP packets with IP record route option was 116.4 μsec, compared with 132.6 μs for 1400-byte ICMP packets.

To understand the causes of this performance degradation, we added several timers in a modified version of the *ipodump* command to measure the time needed to read a packet from a raw `PF_IPOPTION` socket and the time needed to write a packet to a raw socket[3] The results are shown in Figure 4.

We observed that reading and writing did not take much time, approximately 20-25 μs. The total time that a packet spent in the user space was 46.28 μs on average. In other words, out of 124.64 μs total processing delay, the BSD socket interface contributed as much as 78.36 μs or 62.87%.

Kernel performance is affected by CPU clock speed and caching architecture. We ran the same tests on a 600 MHz Pentium PC. As shown in Figure 5, the total processing delay is reduced to 60.28 μsec (or by 51.64%) with processor that has a 20% higher clock speed.

[3] More precisely, we measured the read time as the time from after returning from *select()* to after executing *recvmsg()*.. The write duration was measured as the time from *recvmsg()* to *sendto()*. Refer to Section 2.1 for the actual code.

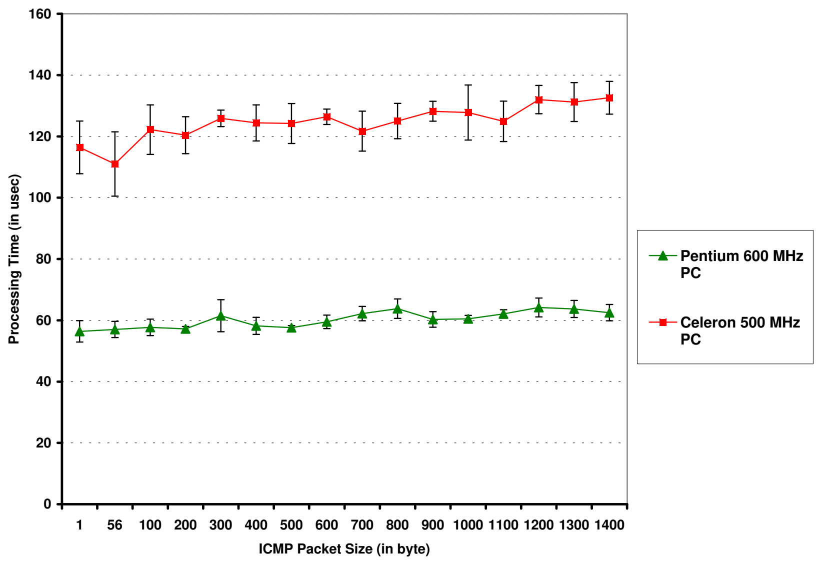

Figure 5: Processing overhead for different CPU's

5 Discussion

We have defined and developed a new socket protocol family `PF_IPOPTION` on FreeBSD. The implementation does not depend on other socket operations, and thus integrating the change into the kernel was fairly simple.

We have measured the packet processing delays both in kernel and in user space. We discovered that the socket I/O seems to be the performance bottleneck, with the delay being almost unaffected by the packet size.

The performance numbers raise some interesting issues. The `PF_IPOPTION` can be used to support RSVP at routers. The routers in a reasonably sized network may need to process large number of RSVP refresh messages. Given the BSD's kernel performance constraint, it would be to costly and difficult to support these refresh messages. To achieve high performance, one solution is to implement signaling protocols (such as RSVP) in the kernel, with users defining socket operations to control protocol operation. A different solution is to develop the protocol in the user space, but to "bundle" the refresh messages into a large message before sending them to the neighbors. This "scatter-gather I/O" mechanism can avoid the per-packet delay in the socket I/O. An example of such mechanism can be found in [8] .

I/O performance can be enhanced by streamlining the socket implementation, although we are unable to suggest any concrete steps. Our comparison between two different Pentium processors showed that cache size, more than raw clock speed, affects the overall protocol performance.

6 Code availability

The source and object code for the `IPOPTION` extension and *ipodump* have been available since April 2000 at http://www.cs.columbia.edu/˜pingpan/software.

We have tested the kernel extension on FreeBSD 3.x and 4.x machines to capture RTCP and RSVP messages, using a modified version of the *rtptools* to generate and record RTCP messages with IP router alert option. The latest *rtptools* source code incorporating these change can be found at http://www.cs.columbia.edu/˜hgs/rtptools.

7 Acknowledgments

Bernard Suter, Rohit Dube and Ram Ramjee helped by discussing implementation options.

References

[1] R. T. Braden, "Requirements for internet hosts - communication layers," Request for Comments 1122, Internet Engineering Task Force, Oct. 1989.

[2] D. Katz, "IP router alert option," Request for Comments 2113, Internet Engineering Task Force, Feb. 1997.

[3] W. R. Stevens, *TCP/IP illustrated: the implementation*, vol. 2. Reading, Massachusetts: Addison-Wesley, 1994.

[4] W. R. Stevens, *UNIX Network Programming: Networking APIs - Sockets and XTI*, vol. 1. Upper Saddle River: Prentice Hall, 1998.

[5] S. McCanne and V. Jacobson, "A BSD packet filter: A new architecture for user-level packet capture," in *Proc. of Usenix Winter Conference*, (San Diego, California), pp. 259–269, Usenix, Jan. 1993.

[6] V. Jacobson and S. McCanne, "libpcap: Packet capture library." ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[7] A. Cobbs, "The divert manual page." http://www.freebsd.org/.

[8] L. Berger, D. Gan, G. Swallow, and P. Pan, "RSVP refresh reduction extensions," Internet Draft, Internet Engineering Task Force, July 1999. Work in progress.

A ipodump Manual Page

```
NAME
       ipodump - dump IP option packet

SYNOPSIS
       ipodump [ -ix ] [ -c count ] [ -s snaplen ] [ -T expres-
       sion ]

DESCRIPTION
       Ipodump prints out the IP option headers of packets  rout-
       ed through a system.

       In  default  operation,  ipodump captures and displays all
       passing IP options packets. Ipodump can also print out the
       option packets that match the expression. After a captured
       packet is printed, ipodump puts it back onto the wire. The
       packet  can then be forwarded to its original destination.

       To run this command, the user  needs  to  have  superuser
       privileges.   The system needs to have the IPOPTION kernel
       extension. The kernel extension source code  is  available
       at

       http://www.cs.columbia.edu/~pingpan/software/netipopt.


OPTIONS
       -i     Set  the IP_RECVIF option on the socket being used.
              Ipodump displays the receiving interface index  for
              all captured packets.


       -x     Print   each  received  packet  (starting  from  IP
              header) in hex.  The smaller of the  entire  packet
              or snaplen bytes will be printed.


       -c count
              Stop  after  receiving  count IP option packets. If
              this option is not specified, ipodump will  operate
              until interrupted.

       -s snaplen
```

Specify the number of data bytes to display. The default is 128.

-T expression

Set the kernel to intercept the packets specified in "expression. Currently known filters are rr (route record), ts (timestamp), sec (security), lsrr (loose source route), ssrr (strict source route), ra (router alert), rsvp (RSVP in router-alert) and rtcp (RTCP in router-alert).

AVAILABILITY

The ipodump command works in FreeBSD 3.3, 3.4 and 4.0.

BUGS

The ipodump command takes packets from the kernel. After parsing and printing, it puts packets back into the kernel. This impacts packet forwarding performance.

AUTHORS

The ipodump command and IPOPTION kernel extension were written by Ping Pan <pingpan@cs.columbia.edu> while at Bell Labs. The idea was cooked up by Ping and Henning Schulzrinne <hgs@cs.columbia.edu>.

Bell Laboratories

Document Cover Sheet for Technical Memorandum

Title: PF_IPOPTION: A Kernel Extension for IP Option Packet Processing

Authors	Electronic Address	Location	Phone	Company (if other than Lucent–BL)
Ping Pan	pingpan@bell-labs.com	HO 4F-506A	+1 732 332 6744	
Henning Schulzrinne	hgs@cs.columbia.edu	NY, NY 10027	+1 212 939 7042	Columbia University

Document No.	Filing Case No.	Work Project No.
10009669-000615-02TM	20564	MA30034004

Keywords:
Kernel, Socket

MERCURY Announcement Bulletin Sections
CMM– Communications CMP– Computing

Abstract

Existing UNIX kernels cannot easily deliver IP packets containing IP options to applications. To address this problem, we have defined and implemented a new protocol family called `PF_IPOPTION` for the FreeBSD kernel. We have verified the implementation with some of the commonly used IP options. Measurements in kernel and user space showed that BSD socket I/O is the performance bottleneck.

Pages of Text 1 Other Pages 14 Total 15
No. Figs. 5 No. Tables 0 No. Refs. 8

Mailing Label

Complete Copy	**Cover Sheet Only**
Directors in 112, 113, 118	Members of 112, 113, 118

Future Lucent Distribution Release to any Lucent employee (excluding contract employees)

Author Signatures

Ping Pan   Henning Schulzrinne

Organizational Approvals

Tom La Porta   Krishan K. Sabnani

For Use by Recipient of Cover Sheet:

Computing network users may order copies via the *library -1* command;
for information, type *man library* after the UNIX® system prompt.

Otherwise:
Enter your Company ID number. ____________
Return this sheet to any ITDS location.

Internal Technical Document Service

( ) AK 2H-28	( ) IH 7M-103	( ) DR 2F-19	( ) NW-ITDS
( ) ALC 1B-102	( ) MV 3L-19	( ) INH 1C-114	( ) PR 5-2120
( ) CB 3O-2011	( ) WH 3E-204	( ) IW 2Z-156	
( ) HO 4F-112		( ) MT 3B-117	