NAME

bpf – Berkeley Packet Filter

SYNOPSIS

pseudo-device bpf

DESCRIPTION

The Berkeley Packet Filter provides a raw interface to data link layers in a protocol independent fashion. All packets on the network, even those destined for other hosts, are accessible through this mechanism.

The packet filter appears as a character special device, `/dev/bpf0`, `/dev/bpf1`, etc. After opening the device, the file descriptor must be bound to a specific network interface with the `BIOCSETIF` ioctl. A given interface can be shared be multiple listeners, and the filter underlying each descriptor will see an identical packet stream.

A separate device file is required for each minor device. If a file is in use, the open will fail and *errno* will be set to `EBUSY`.

Associated with each open instance of a **bpf** file is a user-settable packet filter. Whenever a packet is received by an interface, all file descriptors listening on that interface apply their filter. Each descriptor that accepts the packet receives its own copy.

Reads from these files return the next group of packets that have matched the filter. To improve performance, the buffer passed to read must be the same size as the buffers used internally by **bpf**. This size is returned by the `BIOCGBLEN` ioctl (see below), and can be set with `BIOCSBLEN`. Note that an individual packet larger than this size is necessarily truncated.

The packet filter will support any link level protocol that has fixed length headers. Currently, only Ethernet, SLIP, and PPP drivers have been modified to interact with **bpf**.

Since packet data is in network byte order, applications should use the `byteorder`(3) macros to extract multi-byte values.

A packet can be sent out on the network by writing to a **bpf** file descriptor. The writes are unbuffered, meaning only one packet can be processed per write. Currently, only writes to Ethernets and SLIP links are supported.

IOCTLS

The `ioctl`(2) command codes below are defined in ⟨`net/bpf.h`⟩. All commands require these includes:

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <net/bpf.h>
```

Additionally, `BIOCGETIF` and `BIOCSETIF` require ⟨`sys/socket.h`⟩ and ⟨`net/if.h`⟩.

In addition to `FIONREAD` and `SIOCGIFADDR`, the following commands may be applied to any open **bpf** file. The (third) argument to `ioctl`(2) should be a pointer to the type indicated.

`BIOCGBLEN` (`u_int`) Returns the required buffer length for reads on **bpf** files.

`BIOCSBLEN` (`u_int`) Sets the buffer length for reads on **bpf** files. The buffer must be set before the file is attached to an interface with `BIOCSETIF`. If the requested buffer size cannot be accommodated, the closest allowable size will be set and returned in the argument. A read call will result in `EIO` if it is passed a buffer that is not this size.

BIOCGDLT (u_int) Returns the type of the data link layer underlying the attached interface. EINVAL is returned if no interface has been specified. The device types, prefixed with "DLT_", are defined in ⟨net/bpf.h⟩.

BIOCPROMISC Forces the interface into promiscuous mode. All packets, not just those destined for the local host, are processed. Since more than one file can be listening on a given interface, a listener that opened its interface non-promiscuously may receive packets promiscuously. This problem can be remedied with an appropriate filter.

BIOCFLUSH Flushes the buffer of incoming packets, and resets the statistics that are returned by BIOCGSTATS.

BIOCGETIF (struct ifreq) Returns the name of the hardware interface that the file is listening on. The name is returned in the ifr_name field of the ifreq structure. All other fields are undefined.

BIOCSETIF (struct ifreq) Sets the hardware interface associate with the file. This command must be performed before any packets can be read. The device is indicated by name using the ifr_name field of the ifreq structure. Additionally, performs the actions of BIOCFLUSH.

BIOCSRTIMEOUT

BIOCGRTIMEOUT (struct timeval) Set or get the read timeout parameter. The argument specifies the length of time to wait before timing out on a read request. This parameter is initialized to zero by open(2), indicating no timeout.

BIOCGSTATS (struct bpf_stat) Returns the following structure of packet statistics:

```
struct bpf_stat {
        u_int bs_recv;     /* number of packets received */
        u_int bs_drop;     /* number of packets dropped */
};
```

The fields are:

bs_recv the number of packets received by the descriptor since opened or reset (including any buffered since the last read call); and

bs_drop the number of packets which were accepted by the filter but dropped by the kernel because of buffer overflows (i.e., the application's reads aren't keeping up with the packet traffic).

BIOCIMMEDIATE (u_int) Enable or disable "immediate mode", based on the truth value of the argument. When immediate mode is enabled, reads return immediately upon packet reception. Otherwise, a read will block until either the kernel buffer becomes full or a timeout occurs. This is useful for programs like rarpd(8) which must respond to messages in real time. The default for a new file is off.

BIOCSETF (struct bpf_program) Sets the filter program used by the kernel to discard uninteresting packets. An array of instructions and its length is passed in using the following structure:

```
struct bpf_program {
        int bf_len;
        struct bpf_insn *bf_insns;
};
```

The filter program is pointed to by the `bf_insns` field while its length in units of ‘`struct bpf_insn`’ is given by the `bf_len` field. Also, the actions of `BIOCFLUSH` are performed. See section **FILTER MACHINE** for an explanation of the filter language.

`BIOCVERSION` (`struct bpf_version`) Returns the major and minor version numbers of the filter language currently recognized by the kernel. Before installing a filter, applications must check that the current version is compatible with the running kernel. Version numbers are compatible if the major numbers match and the application minor is less than or equal to the kernel minor. The kernel version number is returned in the following structure:

```
struct bpf_version {
        u_short bv_major;
        u_short bv_minor;
};
```

The current version numbers are given by `BPF_MAJOR_VERSION` and `BPF_MINOR_VERSION` from ⟨`net/bpf.h`⟩. An incompatible filter may result in undefined behavior (most likely, an error returned by **ioctl**() or haphazard packet matching).

`BIOCSHDRCMPLT`

`BIOCGHDRCMPLT` (`u_int`) Set or get the status of the “header complete” flag. Set to zero if the link level source address should be filled in automatically by the interface output routine. Set to one if the link level source address will be written, as provided, to the wire. This flag is initialized to zero by default.

`BIOCSSEESENT`

`BIOCGSEESENT` (`u_int`) Set or get the flag determining whether locally generated packets on the interface should be returned by BPF. Set to zero to see only incoming packets on the interface. Set to one to see packets originating locally and remotely on the interface. This flag is initialized to one by default.

BPF HEADER

The following structure is prepended to each packet returned by `read`(2):

```
struct bpf_hdr {
        struct timeval bh_tstamp;     /* time stamp */
        u_long bh_caplen;             /* length of captured portion */
        u_long bh_datalen;            /* original length of packet */
        u_short bh_hdrlen;            /* length of bpf header (this struct
                                       plus alignment padding */
};
```

The fields, whose values are stored in host order, and are:

`bh_tstamp` The time at which the packet was processed by the packet filter.

`bh_caplen` The length of the captured portion of the packet. This is the minimum of the truncation amount specified by the filter and the length of the packet.

`bh_datalen` The length of the packet off the wire. This value is independent of the truncation amount specified by the filter.

bh_hdrlen The length of the **bpf** header, which may not be equal to **sizeof**(*struct bpf_hdr*).

The bh_hdrlen field exists to account for padding between the header and the link level protocol. The purpose here is to guarantee proper alignment of the packet data structures, which is required on alignment sensitive architectures and improves performance on many other architectures. The packet filter insures that the bpf_hdr and the network layer header will be word aligned. Suitable precautions must be taken when accessing the link layer protocol fields on alignment restricted machines. (This isn't a problem on an Ethernet, since the type field is a short falling on an even offset, and the addresses are probably accessed in a bytewise fashion).

Additionally, individual packets are padded so that each starts on a word boundary. This requires that an application has some knowledge of how to get from packet to packet. The macro BPF_WORDALIGN is defined in ⟨net/bpf.h⟩ to facilitate this process. It rounds up its argument to the nearest word aligned value (where a word is BPF_ALIGNMENT bytes wide).

For example, if 'p' points to the start of a packet, this expression will advance it to the next packet:

```
p = (char *)p + BPF_WORDALIGN(p->bh_hdrlen + p->bh_caplen)
```

For the alignment mechanisms to work properly, the buffer passed to read(2) must itself be word aligned. The malloc(3) function will always return an aligned buffer.

FILTER MACHINE

A filter program is an array of instructions, with all branches forwardly directed, terminated by a *return* instruction. Each instruction performs some action on the pseudo-machine state, which consists of an accumulator, index register, scratch memory store, and implicit program counter.

The following structure defines the instruction format:

```
struct bpf_insn {
        u_short code;
        u_char  jt;
        u_char  jf;
        u_long k;
};
```

The k field is used in different ways by different instructions, and the jt and jf fields are used as offsets by the branch instructions. The opcodes are encoded in a semi-hierarchical fashion. There are eight classes of instructions: BPF_LD, BPF_LDX, BPF_ST, BPF_STX, BPF_ALU, BPF_JMP, BPF_RET, and BPF_MISC. Various other mode and operator bits are or'd into the class to give the actual instructions. The classes and modes are defined in ⟨net/bpf.h⟩.

Below are the semantics for each defined **bpf** instruction. We use the convention that A is the accumulator, X is the index register, P[] packet data, and M[] scratch memory store. P[i:n] gives the data at byte offset "i" in the packet, interpreted as a word (n=4), unsigned halfword (n=2), or unsigned byte (n=1). M[i] gives the i'th word in the scratch memory store, which is only addressed in word units. The memory store is indexed from 0 to BPF_MEMWORDS - 1. k, jt, and jf are the corresponding fields in the instruction definition. "len" refers to the length of the packet.

BPF_LD These instructions copy a value into the accumulator. The type of the source operand is specified by an "addressing mode" and can be a constant (BPF_IMM), packet data at a fixed offset (BPF_ABS), packet data at a variable offset (BPF_IND), the packet length (BPF_LEN), or a word in the scratch memory store (BPF_MEM). For BPF_IND and BPF_ABS, the data size must be specified as a word (BPF_W), halfword (BPF_H), or byte (BPF_B). The semantics of all the recognized BPF_LD instructions follow.

```
BPF_LD+BPF_W+BPF_ABS   A <- P[k:4]
BPF_LD+BPF_H+BPF_ABS   A <- P[k:2]
BPF_LD+BPF_B+BPF_ABS   A <- P[k:1]
BPF_LD+BPF_W+BPF_IND   A <- P[X+k:4]
BPF_LD+BPF_H+BPF_IND   A <- P[X+k:2]
BPF_LD+BPF_B+BPF_IND   A <- P[X+k:1]
BPF_LD+BPF_W+BPF_LEN   A <- len
BPF_LD+BPF_IMM         A <- k
BPF_LD+BPF_MEM         A <- M[k]
```

`BPF_LDX` These instructions load a value into the index register. Note that the addressing modes are more restrictive than those of the accumulator loads, but they include `BPF_MSH`, a hack for efficiently loading the IP header length.

```
BPF_LDX+BPF_W+BPF_IMM  X <- k
BPF_LDX+BPF_W+BPF_MEM  X <- M[k]
BPF_LDX+BPF_W+BPF_LEN  X <- len
BPF_LDX+BPF_B+BPF_MSH  X <- 4*(P[k:1]&0xf)
```

`BPF_ST` This instruction stores the accumulator into the scratch memory. We do not need an addressing mode since there is only one possibility for the destination.

```
BPF_ST  M[k] <- A
```

`BPF_STX` This instruction stores the index register in the scratch memory store.

```
BPF_STX  M[k] <- X
```

`BPF_ALU` The alu instructions perform operations between the accumulator and index register or constant, and store the result back in the accumulator. For binary operations, a source mode is required (`BPF_K` or `BPF_X`).

```
BPF_ALU+BPF_ADD+BPF_K  A <- A + k
BPF_ALU+BPF_SUB+BPF_K  A <- A - k
BPF_ALU+BPF_MUL+BPF_K  A <- A * k
BPF_ALU+BPF_DIV+BPF_K  A <- A / k
BPF_ALU+BPF_AND+BPF_K  A <- A & k
BPF_ALU+BPF_OR+BPF_K   A <- A | k
BPF_ALU+BPF_LSH+BPF_K  A <- A << k
BPF_ALU+BPF_RSH+BPF_K  A <- A >> k
BPF_ALU+BPF_ADD+BPF_X  A <- A + X
BPF_ALU+BPF_SUB+BPF_X  A <- A - X
BPF_ALU+BPF_MUL+BPF_X  A <- A * X
BPF_ALU+BPF_DIV+BPF_X  A <- A / X
BPF_ALU+BPF_AND+BPF_X  A <- A & X
BPF_ALU+BPF_OR+BPF_X   A <- A | X
BPF_ALU+BPF_LSH+BPF_X  A <- A << X
BPF_ALU+BPF_RSH+BPF_X  A <- A >> X
BPF_ALU+BPF_NEG        A <- -A
```

`BPF_JMP` The jump instructions alter flow of control. Conditional jumps compare the accumulator against a constant (`BPF_K`) or the index register (`BPF_X`). If the result is true (or non-zero), the true branch is taken, otherwise the false branch is taken. Jump offsets are encoded in 8 bits so the longest jump is 256 instructions. However, the jump always (`BPF_JA`) opcode uses the 32 bit `k` field as the offset, allowing arbitrarily distant destinations. All conditionals use unsigned comparison conventions.

BPF_JMP+BPF_JA	pc += k
BPF_JMP+BPF_JGT+BPF_K	pc += (A > k) ? jt : jf
BPF_JMP+BPF_JGE+BPF_K	pc += (A >= k) ? jt : jf
BPF_JMP+BPF_JEQ+BPF_K	pc += (A == k) ? jt : jf
BPF_JMP+BPF_JSET+BPF_K	pc += (A & k) ? jt : jf
BPF_JMP+BPF_JGT+BPF_X	pc += (A > X) ? jt : jf
BPF_JMP+BPF_JGE+BPF_X	pc += (A >= X) ? jt : jf
BPF_JMP+BPF_JEQ+BPF_X	pc += (A == X) ? jt : jf
BPF_JMP+BPF_JSET+BPF_X	pc += (A & X) ? jt : jf

`BPF_RET` The return instructions terminate the filter program and specify the amount of packet to accept (i.e., they return the truncation amount). A return value of zero indicates that the packet should be ignored. The return value is either a constant (`BPF_K`) or the accumulator (`BPF_A`).

BPF_RET+BPF_A	accept A bytes
BPF_RET+BPF_K	accept k bytes

`BPF_MISC` The miscellaneous category was created for anything that doesn't fit into the above classes, and for any new instructions that might need to be added. Currently, these are the register transfer instructions that copy the index register to the accumulator or vice versa.

BPF_MISC+BPF_TAX	X <- A
BPF_MISC+BPF_TXA	A <- X

The **bpf** interface provides the following macros to facilitate array initializers: **BPF_STMT**(*opcode*, *operand*) and **BPF_JUMP**(*opcode*, *operand*, *true_offset*, *false_offset*).

EXAMPLES

The following filter is taken from the Reverse ARP Daemon. It accepts only Reverse ARP requests.

```
struct bpf_insn insns[] = {
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_REVARP, 0, 3),
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 20),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, REVARP_REQUEST, 0, 1),
        BPF_STMT(BPF_RET+BPF_K, sizeof(struct ether_arp) +
                 sizeof(struct ether_header)),
        BPF_STMT(BPF_RET+BPF_K, 0),
};
```

This filter accepts only IP packets between host 128.3.112.15 and 128.3.112.35.

```
struct bpf_insn insns[] = {
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 8),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 2),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 3, 4),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 0, 3),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 1),
        BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
        BPF_STMT(BPF_RET+BPF_K, 0),
};
```

Finally, this filter returns only TCP finger packets. We must parse the IP header to reach the TCP header. The BPF_JSET instruction checks that the IP fragment offset is 0 so we are sure that we have a TCP header.

```
struct bpf_insn insns[] = {
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 10),
        BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 23),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, IPPROTO_TCP, 0, 8),
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 20),
        BPF_JUMP(BPF_JMP+BPF_JSET+BPF_K, 0x1fff, 6, 0),
        BPF_STMT(BPF_LDX+BPF_B+BPF_MSH, 14),
        BPF_STMT(BPF_LD+BPF_H+BPF_IND, 14),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 79, 2, 0),
        BPF_STMT(BPF_LD+BPF_H+BPF_IND, 16),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 79, 0, 1),
        BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
        BPF_STMT(BPF_RET+BPF_K, 0),
};
```

SEE ALSO

tcpdump(1), ioctl(2), byteorder(3), ng_bpf(4)

McCanne, S. and Jacobson V., *An efficient, extensible, and portable network monitor*.

FILES

/dev/bpf**n**   the packet filter device

BUGS

The read buffer must be of a fixed size (returned by the BIOCGBLEN ioctl).

A file that does not request promiscuous mode may receive promiscuously received packets as a side effect of another file requesting this mode on the same hardware interface. This could be fixed in the kernel with additional processing overhead. However, we favor the model where all files must assume that the interface is promiscuous, and if so desired, must utilize a filter to reject foreign packets.

Data link protocols with variable length headers are not currently supported.

HISTORY

The Enet packet filter was created in 1980 by Mike Accetta and Rick Rashid at Carnegie-Mellon University. Jeffrey Mogul, at Stanford, ported the code to BSD and continued its development from 1983 on. Since then, it has evolved into the Ultrix Packet Filter at DEC, a STREAMS NIT module under SunOS 4.1, and BPF.

AUTHORS

Steven McCanne, of Lawrence Berkeley Laboratory, implemented BPF in Summer 1990. Much of the design is due to Van Jacobson.