NAME

bpf - Berkeley Packet Filter

SYNOPSIS

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/bpf*. 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 by multiple listeners, and the filter underlying each descriptor will see an identical packet stream.

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.

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.

BUFFER MODES

bpf devices deliver packet data to the application via memory buffers provided by the application. The buffer mode is set using the BIOCSETBUFMODE ioctl, and read using the BIOCGETBUFMODE ioctl.

Buffered read mode

By default, **bpf** devices operate in the BPF_BUFMODE_BUFFER mode, in which packet data is copied explicitly from kernel to user memory using the read(2) system call. The user process will declare a fixed buffer size that will be used both for sizing internal buffers and for all read(2) operations on the file. This size is queried using the BIOCGBLEN ioctl, and is set using the BIOCSBLEN ioctl. Note that an individual packet larger than the buffer size is necessarily truncated.

Zero-copy buffer mode

bpf devices may also operate in the BPF_BUFMODE_ZEROCOPY mode, in which packet data is written directly into two user memory buffers by the kernel, avoiding both system call and copying overhead. Buffers are of fixed (and equal) size, page-aligned, and an even multiple of the page size. The maximum zero-copy buffer size is returned by the BIOCGETZMAX ioctl. Note that an individual packet larger than the buffer size is necessarily truncated.

The user process registers two memory buffers using the BIOCSETZBUF ioctl, which accepts a *struct bpf_zbuf* pointer as an argument:

```
struct bpf_zbuf {
        void *bz_bufa;
        void *bz_bufb;
        size_t bz_buflen;
};
```

bz_bufa is a pointer to the userspace address of the first buffer that will be filled, and *bz_bufb* is a pointer to the second buffer. **bpf** will then cycle between the two buffers as they fill and are acknowledged.

Each buffer begins with a fixed-length header to hold synchronization and data length information for the buffer:

```
struct bpf_zbuf_header {
        volatile u_int  bzh_kernel_gen;         /* Kernel generation number. */
        volatile u_int  bzh_kernel_len;         /* Length of data in the buffer. */
        volatile u_int  bzh_user_gen;  /* User generation number. */
        /* ...padding for future use... */
};
```

The header structure of each buffer, including all padding, should be zeroed before it is configured using BIOCSETZBUF. Remaining space in the buffer will be used by the kernel to store packet data, laid out in the same format as with buffered read mode.

The kernel and the user process follow a simple acknowledgement protocol via the buffer header to synchronize access to the buffer: when the header generation numbers, *bzh_kernel_gen* and *bzh_user_gen*, hold the same value, the kernel owns the buffer, and when they differ, userspace owns the buffer.

While the kernel owns the buffer, the contents are unstable and may change asynchronously; while the user process owns the buffer, its contents are stable and will not be changed until the buffer has been acknowledged.

Initializing the buffer headers to all 0’s before registering the buffer has the effect of assigning initial ownership of both buffers to the kernel. The kernel signals that a buffer has been assigned to userspace by modifying *bzh_kernel_gen*, and userspace acknowledges the buffer and returns it to the kernel by setting the value of *bzh_user_gen* to the value of *bzh_kernel_gen*.

In order to avoid caching and memory re-ordering effects, the user process must use atomic operations and memory barriers when checking for and acknowledging buffers:

```
#include <machine/atomic.h>

/*
 * Return ownership of a buffer to the kernel for reuse.
 */
static void
buffer_acknowledge(struct bpf_zbuf_header *bzh)
{

        atomic_store_rel_int(&bzh->bzh_user_gen, bzh->bzh_kernel_gen);
}

/*
 * Check whether a buffer has been assigned to userspace by the kernel.
 * Return true if userspace owns the buffer, and false otherwise.
 */
static int
buffer_check(struct bpf_zbuf_header *bzh)
{

        return (bzh->bzh_user_gen !=
            atomic_load_acq_int(&bzh->bzh_kernel_gen));
}
```

The user process may force the assignment of the next buffer, if any data is pending, to userspace using the BIOCROTZBUF ioctl. This allows the user process to retrieve data in a partially filled buffer before the buffer is full, such as following a timeout; the process must recheck for buffer ownership using the header generation numbers, as the buffer will not be assigned to userspace if no data was present.

As in the buffered read mode, kqueue(2), poll(2), and select(2) may be used to sleep awaiting the availability of a completed buffer. They will return a readable file descriptor when ownership of the next buffer is assigned to user space.

In the current implementation, the kernel may assign zero, one, or both buffers to the user process; however, an earlier implementation maintained the invariant that at most one buffer could be assigned to the user process at a time. In order to both ensure progress and high performance, user processes should acknowledge a completely processed buffer as quickly as possible, returning it for reuse, and not block

waiting on a second buffer while holding another buffer.

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 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 EINVAL 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*>.

BIOCGDLTLIST (struct bpf_dltlist) Returns an array of the available types of the data link layer underlying the attached interface:

```
struct bpf_dltlist {
        u_int bfl_len;
        u_int *bfl_list;
};
```

The available types are returned in the array pointed to by the *bfl_list* field while their length in u_int is supplied to the *bfl_len* field. ENOMEM is returned if there is not enough buffer space and EFAULT is returned if a bad address is encountered. The *bfl_len* field is modified on return to indicate the actual length in u_int of the array returned. If *bfl_list* is NULL, the *bfl_len* field is set

to indicate the required length of an array in u_int.

BIOCSDLT (u_int) Changes the type of the data link layer underlying the attached interface. EINVAL is returned if no interface has been specified or the specified type is not available for the interface.

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.

The interface remains in promiscuous mode until all files listening promiscuously are closed.

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

bs_dropthe number of packets which were accepted by the filter but dropped by the kernel because of buffer overflows (i.e., the application’s reads are not keeping up with the packet traffic).

BIOCIMMEDIATE (u_int) Enables or disables "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

BIOCSETFNR (struct bpf_program) Sets the read 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 {
        u_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. See section *FILTER MACHINE* for an explanation of the filter language. The only difference between BIOCSETF and BIOCSETFNR is BIOCSETF performs the actions of BIOCFLUSH while BIOCSETFNR does not.

BIOCSETWF (struct bpf_program) Sets the write filter program used by the kernel to control what type of packets can be written to the interface. See the BIOCSETF command for more information on the **bpf** filter program.

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).

BIOCGRSIG

BIOCSRSIG (u_int) Sets or gets the receive signal. This signal will be sent to the process or process group specified by FIOSETOWN. It defaults to SIGIO.

BIOCSHDRCMPLT

BIOCGHDRCMPLT (u_int) Sets or gets 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) These commands are obsolete but left for compatibility. Use BIOCSDIRECTION and BIOCGDIRECTION instead. Sets or gets 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.

BIOCSDIRECTION

BIOCGDIRECTION (u_int) Sets or gets the setting determining whether incoming, outgoing, or all packets on the interface should be returned by BPF. Set to BPF_D_IN to see only incoming packets on the interface. Set to BPF_D_INOUT to see packets originating locally and remotely on the interface. Set to BPF_D_OUT to see

only outgoing packets on the interface. This setting is initialized to BPF_D_INOUT by default.

BIOCSTSTAMP

BIOCGTSTAMP (u_int) Set or get format and resolution of the time stamps returned by BPF. Set to BPF_T_MICROTIME, BPF_T_MICROTIME_FAST, BPF_T_MICROTIME_MONOTONIC, or BPF_T_MICROTIME_MONOTONIC_FAST to get time stamps in 64-bit *struct timeval* format. Set to BPF_T_NANOTIME, BPF_T_NANOTIME_FAST, BPF_T_NANOTIME_MONOTONIC, or BPF_T_NANOTIME_MONOTONIC_FAST to get time stamps in 64-bit *struct timespec* format. Set to BPF_T_BINTIME, BPF_T_BINTIME_FAST, BPF_T_NANOTIME_MONOTONIC, or BPF_T_BINTIME_MONOTONIC_FAST to get time stamps in 64-bit *struct bintime* format. Set to BPF_T_NONE to ignore time stamp. All 64-bit time stamp formats are wrapped in *struct bpf_ts*. The BPF_T_MICROTIME_FAST, BPF_T_NANOTIME_FAST, BPF_T_BINTIME_FAST, BPF_T_MICROTIME_MONOTONIC_FAST, BPF_T_NANOTIME_MONOTONIC_FAST, and BPF_T_BINTIME_MONOTONIC_FAST are analogs of corresponding formats without _FAST suffix but do not perform a full time counter query, so their accuracy is one timer tick. The BPF_T_MICROTIME_MONOTONIC, BPF_T_NANOTIME_MONOTONIC, BPF_T_BINTIME_MONOTONIC, BPF_T_MICROTIME_MONOTONIC_FAST, BPF_T_NANOTIME_MONOTONIC_FAST, and BPF_T_BINTIME_MONOTONIC_FAST store the time elapsed since kernel boot. This setting is initialized to BPF_T_MICROTIME by default.

BIOCFEEDBACK (u_int) Set packet feedback mode. This allows injected packets to be fed back as input to the interface when output via the interface is successful. When BPF_D_INOUT direction is set, injected outgoing packet is not returned by BPF to avoid duplication. This flag is initialized to zero by default.

BIOCLOCK Set the locked flag on the **bpf** descriptor. This prevents the execution of ioctl commands which could change the underlying operating parameters of the device.

BIOCGETBUFMODE

BIOCSETBUFMODE (u_int) Get or set the current **bpf** buffering mode; possible values are BPF_BUFMODE_BUFFER, buffered read mode, and BPF_BUFMODE_ZBUF, zero-copy buffer mode.

BIOCSETZBUF (struct bpf_zbuf) Set the current zero-copy buffer locations; buffer locations may be set only once zero-copy buffer mode has been selected, and prior to attaching to an interface. Buffers must be of identical size, page-aligned, and an integer multiple of pages in size. The three fields *bz_bufa*, *bz_bufb*, and *bz_buflen* must be filled out. If buffers have already been set for this device, the ioctl will fail.

BIOCGETZMAX (size_t) Get the largest individual zero-copy buffer size allowed. As two buffers are used in zero-copy buffer mode, the limit (in practice) is twice the returned size. As zero-copy buffers consume kernel address space, conservative selection of buffer size is suggested, especially when there are multiple **bpf** descriptors in use on 32-bit systems.

BIOCROTZBUF Force ownership of the next buffer to be assigned to userspace, if any data present in the buffer. If no data is present, the buffer will remain owned by the kernel. This allows consumers of zero-copy buffering to implement timeouts and retrieve partially filled buffers. In order to handle the case where no data is present in the buffer and therefore ownership is not assigned, the user process must check *bzh_kernel_gen* against *bzh_user_gen*.

BIOCSETVLANPCP Set the VLAN PCP bits to the supplied value.

STANDARD IOCTLS

bpf now supports several standard ioctl(2)’s which allow the user to do async and/or non-blocking I/O to an open file descriptor.

FIONREAD (int) Returns the number of bytes that are immediately available for reading.

SIOCGIFADDR (struct ifreq) Returns the address associated with the interface.

FIONBIO (int) Sets or clears non-blocking I/O. If arg is non-zero, then doing a read(2) when no data is available will return -1 and *errno* will be set to EAGAIN. If arg is zero, non-blocking I/O is disabled. Note: setting this overrides the timeout set by BIOCSRTIMEOUT.

FIOASYNC (int) Enables or disables async I/O. When enabled (arg is non-zero), the process or

process group specified by FIOSETOWN will start receiving SIGIO ’s when packets arrive. Note that you must do an FIOSETOWN in order for this to take effect, as the system will not default this for you. The signal may be changed via BIOCSRSIG.

FIOSETOWN

FIOGETOWN (int) Sets or gets the process or process group (if negative) that should receive SIGIO when packets are available. The signal may be changed using BIOCSRSIG (see above).

BPF HEADER

One of the following structures is prepended to each packet returned by read(2) or via a zero-copy buffer:

```
struct bpf_xhdr {
        struct bpf_ts   bh_tstamp;     /* time stamp */
        uint32_t  bh_caplen;    /* length of captured portion */
        uint32_t  bh_datalen;   /* original length of packet */
        u_short         bh_hdrlen;     /* length of bpf header (this struct
                                          plus alignment padding) */
};

struct bpf_hdr {
        struct timeval  bh_tstamp;     /* time stamp */
        uint32_t  bh_caplen;    /* length of captured portion */
        uint32_t  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_xhdr*) or **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 ensures that the *bpf_xhdr*, *bpf_hdr* and the network layer header will be word aligned. Currently, *bpf_hdr* is used when the time stamp is set to BPF_T_MICROTIME, BPF_T_MICROTIME_FAST, BPF_T_MICROTIME_MONOTONIC, BPF_T_MICROTIME_MONOTONIC_FAST, or BPF_T_NONE for backward compatibility reasons. Otherwise, *bpf_xhdr* is used. However, *bpf_hdr* may be deprecated in the near future. Suitable precautions must be taken when accessing the link layer protocol fields on alignment restricted machines. (This is not 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;
        bpf_u_int32 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_MOD+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_XOR+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_MOD+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_XOR+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 does not 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*).

SYSCTL VARIABLES

A set of sysctl(8) variables controls the behaviour of the **bpf** subsystem

net.bpf.optimize_writers: 0

Various programs use BPF to send (but not receive) raw packets (cdpd, lldpd, dhcpd, dhcp relays, etc. are good examples of such programs). They do not need incoming packets to be send to them. Turning this option on makes new BPF users to be attached to write-only interface list until program explicitly specifies read filter via **pcap_set_filter**(). This removes any performance degradation for high-speed interfaces.

net.bpf.stats:

Binary interface for retrieving general statistics.

net.bpf.zerocopy_enable: 0

Permits zero-copy to be used with net BPF readers. Use with caution.

net.bpf.maxinsns: 512
Maximum number of instructions that BPF program can contain. Use tcpdump(1) **-d** option to determine approximate number of instruction for any filter.

net.bpf.maxbufsize: 524288
Maximum buffer size to allocate for packets buffer.

net.bpf.bufsize: 4096
Default buffer size to allocate for packets buffer.

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, ARPOP_REVREQUEST, 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), kqueue(2), poll(2), select(2), ng_bpf(4), bpf(9)

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

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.

Support for zero-copy buffers was added by Robert N. M. Watson under contract to Seccuris Inc.

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.

The SEESENT, DIRECTION, and FEEDBACK settings have been observed to work incorrectly on some interface types, including those with hardware loopback rather than software loopback, and point-to-point interfaces. They appear to function correctly on a broad range of Ethernet-style interfaces.