NAME

bpf – perform a command on an extended BPF map or program

SYNOPSIS

#include <linux/bpf.h>

int bpf(int *cmd*, **union bpf_attr ****attr*, **unsigned int** *size*);

DESCRIPTION

The **bpf**() system call performs a range of operations related to extended Berkeley Packet Filters. Extended BPF (or eBPF) is similar to the original ("classic") BPF (cBPF) used to filter network packets. For both cBPF and eBPF programs, the kernel statically analyzes the programs before loading them, in order to ensure that they cannot harm the running system.

eBPF extends cBPF in multiple ways, including the ability to call a fixed set of in-kernel helper functions (via the **BPF_CALL** opcode extension provided by eBPF) and access shared data structures such as eBPF maps.

Extended BPF Design/Architecture

eBPF maps are a generic data structure for storage of different data types. Data types are generally treated as binary blobs, so a user just specifies the size of the key and the size of the value at map-creation time. In other words, a key/value for a given map can have an arbitrary structure.

A user process can create multiple maps (with key/value-pairs being opaque bytes of data) and access them via file descriptors. Different eBPF programs can access the same maps in parallel. It's up to the user process and eBPF program to decide what they store inside maps.

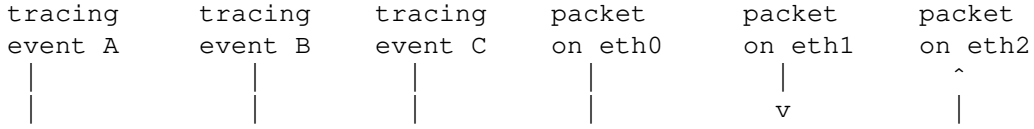
There's one special map type, called a program array. This type of map stores file descriptors referring to other eBPF programs. When a lookup in the map is performed, the program flow is redirected in-place to the beginning of another eBPF program and does not return back to the calling program. The level of nesting has a fixed limit of 32, so that infinite loops cannot be crafted. At run time, the program file descriptors stored in the map can be modified, so program functionality can be altered based on specific requirements. All programs referred to in a program-array map must have been previously loaded into the kernel via **bpf**(). If a map lookup fails, the current program continues its execution. See **BPF_MAP_TYPE_PROG_ARRAY** below for further details.

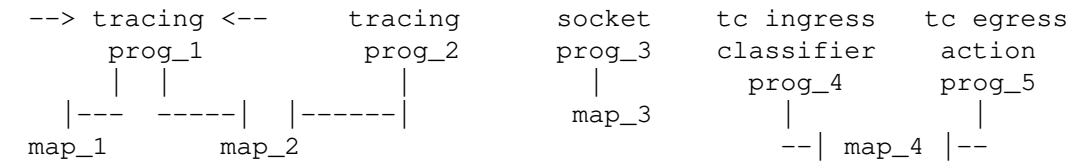
Generally, eBPF programs are loaded by the user process and automatically unloaded when the process exits. In some cases, for example, **tc-bpf**(8), the program will continue to stay alive inside the kernel even after the process that loaded the program exits. In that case, the tc subsystem holds a reference to the eBPF program after the file descriptor has been closed by the user-space program. Thus, whether a specific program continues to live inside the kernel depends on how it is further attached to a given kernel subsystem after it was loaded via **bpf**().

Each eBPF program is a set of instructions that is safe to run until its completion. An in-kernel verifier statically determines that the eBPF program terminates and is safe to execute. During verification, the kernel increments reference counts for each of the maps that the eBPF program uses, so that the attached maps can't be removed until the program is unloaded.

eBPF programs can be attached to different events. These events can be the arrival of network packets, tracing events, classification events by network queueing disciplines (for eBPF programs attached to a **tc**(8) classifier), and other types that may be added in the future. A new event triggers execution of the eBPF program, which may store information about the event in eBPF maps. Beyond storing data, eBPF programs may call a fixed set of in-kernel helper functions.

The same eBPF program can be attached to multiple events and different eBPF programs can access the same map:





Arguments

The operation to be performed by the **bpf**() system call is determined by the *cmd* argument. Each operation takes an accompanying argument, provided via *attr*, which is a pointer to a union of type *bpf_attr* (see below). The *size* argument is the size of the union pointed to by *attr*.

The value provided in *cmd* is one of the following:

BPF_MAP_CREATE
Create a map and return a file descriptor that refers to the map. The close-on-exec file descriptor flag (see **fcntl**(2)) is automatically enabled for the new file descriptor.

BPF_MAP_LOOKUP_ELEM
Look up an element by key in a specified map and return its value.

BPF_MAP_UPDATE_ELEM
Create or update an element (key/value pair) in a specified map.

BPF_MAP_DELETE_ELEM
Look up and delete an element by key in a specified map.

BPF_MAP_GET_NEXT_KEY
Look up an element by key in a specified map and return the key of the next element.

BPF_PROG_LOAD
Verify and load an eBPF program, returning a new file descriptor associated with the program. The close-on-exec file descriptor flag (see **fcntl**(2)) is automatically enabled for the new file descriptor.

The *bpf_attr* union consists of various anonymous structures that are used by different **bpf**() commands:

```
union bpf_attr {
    struct {    /* Used by BPF_MAP_CREATE */
        __u32         map_type;
        __u32         key_size;    /* size of key in bytes */
        __u32         value_size;  /* size of value in bytes */
        __u32         max_entries; /* maximum number of entries
                                      in a map */
    };

    struct {    /* Used by BPF_MAP_*_ELEM and BPF_MAP_GET_NEXT_KEY
                   commands */
        __u32         map_fd;
        __aligned_u64 key;
        union {
            __aligned_u64 value;
            __aligned_u64 next_key;
        };
        __u64         flags;
    };

    struct {    /* Used by BPF_PROG_LOAD */
        __u32         prog_type;
        __u32         insn_cnt;
```

```
            __aligned_u64 insns;      /* 'const struct bpf_insn *' */
            __aligned_u64 license;    /* 'const char *' */
            __u32         log_level;  /* verbosity level of verifier */
            __u32         log_size;   /* size of user buffer */
            __aligned_u64 log_buf;    /* user supplied 'char *'
                                         buffer */
            __u32         kern_version;
                                      /* checked when prog_type=kprobe
                                         (since Linux 4.1) */
        };
    } __attribute__((aligned(8)));
```

eBPF maps

Maps are a generic data structure for storage of different types of data. They allow sharing of data between eBPF kernel programs, and also between kernel and user-space applications.

Each map type has the following attributes:

* type
* maximum number of elements
* key size in bytes
* value size in bytes

The following wrapper functions demonstrate how various **bpf**() commands can be used to access the maps. The functions use the *cmd* argument to invoke different operations.

BPF_MAP_CREATE

The **BPF_MAP_CREATE** command creates a new map, returning a new file descriptor that refers to the map.

```
int
bpf_create_map(enum bpf_map_type map_type,
               unsigned int key_size,
               unsigned int value_size,
               unsigned int max_entries)
{
    union bpf_attr attr = {
        .map_type    = map_type,
        .key_size    = key_size,
        .value_size  = value_size,
        .max_entries = max_entries
    };

    return bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
}
```

The new map has the type specified by *map_type*, and attributes as specified in *key_size*, *value_size*, and *max_entries*. On success, this operation returns a file descriptor. On error, −1 is returned and *errno* is set to **EINVAL**, **EPERM**, or **ENOMEM**.

The *key_size* and *value_size* attributes will be used by the verifier during program loading to check that the program is calling **bpf_map_*_elem**() helper functions with a correctly initialized *key* and to check that the program doesn't access the map element *value* beyond the specified *value_size*. For example, when a map is created with a *key_size* of 8 and the eBPF program calls

```
bpf_map_lookup_elem(map_fd, fp - 4)
```

the program will be rejected, since the in-kernel helper function

```
bpf_map_lookup_elem(map_fd, void *key)
```

expects to read 8 bytes from the location pointed to by *key*, but the *fp - 4* (where *fp* is the top of the stack) starting address will cause out-of-bounds stack access.

Similarly, when a map is created with a *value_size* of 1 and the eBPF program contains

```
value = bpf_map_lookup_elem(...);
*(u32 *) value = 1;
```

the program will be rejected, since it accesses the *value* pointer beyond the specified 1 byte *value_size* limit.

Currently, the following values are supported for *map_type*:

```
enum bpf_map_type {
    BPF_MAP_TYPE_UNSPEC,  /* Reserve 0 as invalid map type */
    BPF_MAP_TYPE_HASH,
    BPF_MAP_TYPE_ARRAY,
    BPF_MAP_TYPE_PROG_ARRAY,
    BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    BPF_MAP_TYPE_PERCPU_HASH,
    BPF_MAP_TYPE_PERCPU_ARRAY,
    BPF_MAP_TYPE_STACK_TRACE,
    BPF_MAP_TYPE_CGROUP_ARRAY,
    BPF_MAP_TYPE_LRU_HASH,
    BPF_MAP_TYPE_LRU_PERCPU_HASH,
    BPF_MAP_TYPE_LPM_TRIE,
    BPF_MAP_TYPE_ARRAY_OF_MAPS,
    BPF_MAP_TYPE_HASH_OF_MAPS,
    BPF_MAP_TYPE_DEVMAP,
    BPF_MAP_TYPE_SOCKMAP,
    BPF_MAP_TYPE_CPUMAP,
};
```

map_type selects one of the available map implementations in the kernel. For all map types, eBPF programs access maps with the same **bpf_map_lookup_elem**() and **bpf_map_update_elem**() helper functions. Further details of the various map types are given below.

BPF_MAP_LOOKUP_ELEM

The **BPF_MAP_LOOKUP_ELEM** command looks up an element with a given *key* in the map referred to by the file descriptor *fd*.

```
int
bpf_lookup_elem(int fd, const void *key, void *value)
{
    union bpf_attr attr = {
        .map_fd = fd,
        .key    = ptr_to_u64(key),
        .value  = ptr_to_u64(value),
    };

    return bpf(BPF_MAP_LOOKUP_ELEM, &attr, sizeof(attr));
}
```

If an element is found, the operation returns zero and stores the element's value into *value*, which must point to a buffer of *value_size* bytes.

If no element is found, the operation returns −1 and sets *errno* to **ENOENT**.

BPF_MAP_UPDATE_ELEM

The **BPF_MAP_UPDATE_ELEM** command creates or updates an element with a given *key/value* in the map referred to by the file descriptor *fd*.

```
int
bpf_update_elem(int fd, const void *key, const void *value,
                uint64_t flags)
{
    union bpf_attr attr = {
        .map_fd = fd,
        .key    = ptr_to_u64(key),
        .value  = ptr_to_u64(value),
        .flags  = flags,
    };

    return bpf(BPF_MAP_UPDATE_ELEM, &attr, sizeof(attr));
}
```

The *flags* argument should be specified as one of the following:

BPF_ANY
Create a new element or update an existing element.

BPF_NOEXIST
Create a new element only if it did not exist.

BPF_EXIST
Update an existing element.

On success, the operation returns zero. On error, −1 is returned and *errno* is set to **EINVAL**, **EPERM**, **ENOMEM**, or **E2BIG**. **E2BIG** indicates that the number of elements in the map reached the *max_entries* limit specified at map creation time. **EEXIST** will be returned if *flags* specifies **BPF_NOEXIST** and the element with *key* already exists in the map. **ENOENT** will be returned if *flags* specifies **BPF_EXIST** and the element with *key* doesn't exist in the map.

BPF_MAP_DELETE_ELEM

The **BPF_MAP_DELETE_ELEM** command deletes the element whose key is *key* from the map referred to by the file descriptor *fd*.

```
int
bpf_delete_elem(int fd, const void *key)
{
    union bpf_attr attr = {
        .map_fd = fd,
        .key    = ptr_to_u64(key),
    };

    return bpf(BPF_MAP_DELETE_ELEM, &attr, sizeof(attr));
}
```

On success, zero is returned. If the element is not found, −1 is returned and *errno* is set to **ENOENT**.

BPF_MAP_GET_NEXT_KEY

The **BPF_MAP_GET_NEXT_KEY** command looks up an element by *key* in the map referred to by the file descriptor *fd* and sets the *next_key* pointer to the key of the next element.

```
int
bpf_get_next_key(int fd, const void *key, void *next_key)
{
    union bpf_attr attr = {
```

```
            .map_fd   = fd,
            .key      = ptr_to_u64(key),
            .next_key = ptr_to_u64(next_key),
        };

        return bpf(BPF_MAP_GET_NEXT_KEY, &attr, sizeof(attr));
    }
```

If *key* is found, the operation returns zero and sets the *next_key* pointer to the key of the next element. If *key* is not found, the operation returns zero and sets the *next_key* pointer to the key of the first element. If *key* is the last element, −1 is returned and *errno* is set to **ENOENT**. Other possible *errno* values are **ENOMEM**, **EFAULT**, **EPERM**, and **EINVAL**. This method can be used to iterate over all elements in the map.

close(map_fd)

Delete the map referred to by the file descriptor *map_fd*. When the user-space program that created a map exits, all maps will be deleted automatically (but see NOTES).

eBPF map types

The following map types are supported:

BPF_MAP_TYPE_HASH

Hash-table maps have the following characteristics:

* Maps are created and destroyed by user-space programs. Both user-space and eBPF programs can perform lookup, update, and delete operations.
* The kernel takes care of allocating and freeing key/value pairs.
* The **map_update_elem**() helper will fail to insert new element when the *max_entries* limit is reached. (This ensures that eBPF programs cannot exhaust memory.)
* **map_update_elem**() replaces existing elements atomically.

Hash-table maps are optimized for speed of lookup.

BPF_MAP_TYPE_ARRAY

Array maps have the following characteristics:

* Optimized for fastest possible lookup. In the future the verifier/JIT compiler may recognize lookup() operations that employ a constant key and optimize it into constant pointer. It is possible to optimize a non-constant key into direct pointer arithmetic as well, since pointers and *value_size* are constant for the life of the eBPF program. In other words, **array_map_lookup_elem**() may be 'inlined' by the verifier/JIT compiler while preserving concurrent access to this map from user space.
* All array elements pre-allocated and zero initialized at init time
* The key is an array index, and must be exactly four bytes.
* **map_delete_elem**() fails with the error **EINVAL**, since elements cannot be deleted.
* **map_update_elem**() replaces elements in a **nonatomic** fashion; for atomic updates, a hash-table map should be used instead. There is however one special case that can also be used with arrays: the atomic built-in **__sync_fetch_and_add()** can be used on 32 and 64 bit atomic counters. For example, it can be applied on the whole value itself if it represents a single counter, or in case of a structure containing multiple counters, it could be used on individual counters. This is quite often useful for aggregation and accounting of events.

Among the uses for array maps are the following:

* As "global" eBPF variables: an array of 1 element whose key is (index) 0 and where the value is a collection of 'global' variables which eBPF programs can use to keep state between events.

* Aggregation of tracing events into a fixed set of buckets.
* Accounting of networking events, for example, number of packets and packet sizes.

BPF_MAP_TYPE_PROG_ARRAY (since Linux 4.2)

A program array map is a special kind of array map whose map values contain only file descriptors referring to other eBPF programs. Thus, both the *key_size* and *value_size* must be exactly four bytes. This map is used in conjunction with the **bpf_tail_call**() helper.

This means that an eBPF program with a program array map attached to it can call from kernel side into

```
void bpf_tail_call(void *context, void *prog_map,
                   unsigned int index);
```

and therefore replace its own program flow with the one from the program at the given program array slot, if present. This can be regarded as kind of a jump table to a different eBPF program. The invoked program will then reuse the same stack. When a jump into the new program has been performed, it won't return to the old program anymore.

If no eBPF program is found at the given index of the program array (because the map slot doesn't contain a valid program file descriptor, the specified lookup index/key is out of bounds, or the limit of 32 nested calls has been exceed), execution continues with the current eBPF program. This can be used as a fall-through for default cases.

A program array map is useful, for example, in tracing or networking, to handle individual system calls or protocols in their own subprograms and use their identifiers as an individual map index. This approach may result in performance benefits, and also makes it possible to overcome the maximum instruction limit of a single eBPF program. In dynamic environments, a user-space daemon might atomically replace individual subprograms at run-time with newer versions to alter overall program behavior, for instance, if global policies change.

eBPF programs

The **BPF_PROG_LOAD** command is used to load an eBPF program into the kernel. The return value for this command is a new file descriptor associated with this eBPF program.

```
char bpf_log_buf[LOG_BUF_SIZE];

int
bpf_prog_load(enum bpf_prog_type type,
              const struct bpf_insn *insns, int insn_cnt,
              const char *license)
{
    union bpf_attr attr = {
        .prog_type = type,
        .insns     = ptr_to_u64(insns),
        .insn_cnt  = insn_cnt,
        .license   = ptr_to_u64(license),
        .log_buf   = ptr_to_u64(bpf_log_buf),
        .log_size  = LOG_BUF_SIZE,
        .log_level = 1,
    };

    return bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
}
```

prog_type is one of the available program types:

```
enum bpf_prog_type {
    BPF_PROG_TYPE_UNSPEC,        /* Reserve 0 as invalid
                                    program type */
```

```
        BPF_PROG_TYPE_SOCKET_FILTER,
        BPF_PROG_TYPE_KPROBE,
        BPF_PROG_TYPE_SCHED_CLS,
        BPF_PROG_TYPE_SCHED_ACT,
};
```

For further details of eBPF program types, see below.

The remaining fields of *bpf_attr* are set as follows:

* *insns* is an array of *struct bpf_insn* instructions.
* *insn_cnt* is the number of instructions in the program referred to by *insns*.
* *license* is a license string, which must be GPL compatible to call helper functions marked *gpl_only*. (The licensing rules are the same as for kernel modules, so that also dual licenses, such as "Dual BSD/GPL", may be used.)
* *log_buf* is a pointer to a caller-allocated buffer in which the in-kernel verifier can store the verification log. This log is a multi-line string that can be checked by the program author in order to understand how the verifier came to the conclusion that the eBPF program is unsafe. The format of the output can change at any time as the verifier evolves.
* *log_size* size of the buffer pointed to by *log_buf*. If the size of the buffer is not large enough to store all verifier messages, −1 is returned and *errno* is set to **ENOSPC**.
* *log_level* verbosity level of the verifier. A value of zero means that the verifier will not provide a log; in this case, *log_buf* must be a NULL pointer, and *log_size* must be zero.

Applying **close**(2) to the file descriptor returned by **BPF_PROG_LOAD** will unload the eBPF program (but see NOTES).

Maps are accessible from eBPF programs and are used to exchange data between eBPF programs and between eBPF programs and user-space programs. For example, eBPF programs can process various events (like kprobe, packets) and store their data into a map, and user-space programs can then fetch data from the map. Conversely, user-space programs can use a map as a configuration mechanism, populating the map with values checked by the eBPF program, which then modifies its behavior on the fly according to those values.

eBPF program types

The eBPF program type (*prog_type*) determines the subset of kernel helper functions that the program may call. The program type also determines the program input (context)—the format of *struct bpf_context* (which is the data blob passed into the eBPF program as the first argument).

For example, a tracing program does not have the exact same subset of helper functions as a socket filter program (though they may have some helpers in common). Similarly, the input (context) for a tracing program is a set of register values, while for a socket filter it is a network packet.

The set of functions available to eBPF programs of a given type may increase in the future.

The following program types are supported:

BPF_PROG_TYPE_SOCKET_FILTER (since Linux 3.19)

Currently, the set of functions for **BPF_PROG_TYPE_SOCKET_FILTER** is:

```
bpf_map_lookup_elem(map_fd, void *key)
                    /* look up key in a map_fd */
bpf_map_update_elem(map_fd, void *key, void *value)
                    /* update key/value */
bpf_map_delete_elem(map_fd, void *key)
                    /* delete key in a map_fd */
```

The *bpf_context* argument is a pointer to a *struct __sk_buff*.

BPF_PROG_TYPE_KPROBE (since Linux 4.1)
[To be documented]

BPF_PROG_TYPE_SCHED_CLS (since Linux 4.1)
[To be documented]

BPF_PROG_TYPE_SCHED_ACT (since Linux 4.1)
[To be documented]

Events

Once a program is loaded, it can be attached to an event. Various kernel subsystems have different ways to do so.

Since Linux 3.19, the following call will attach the program *prog_fd* to the socket *sockfd*, which was created by an earlier call to **socket**(2):

```
setsockopt(sockfd, SOL_SOCKET, SO_ATTACH_BPF,
           &prog_fd, sizeof(prog_fd));
```

Since Linux 4.1, the following call may be used to attach the eBPF program referred to by the file descriptor *prog_fd* to a perf event file descriptor, *event_fd*, that was created by a previous call to **perf_event_open**(2):

```
ioctl(event_fd, PERF_EVENT_IOC_SET_BPF, prog_fd);
```

EXAMPLES

```
/* bpf+sockets example:
 * 1. create array map of 256 elements
 * 2. load program that counts number of packets received
 *    r0 = skb->data[ETH_HLEN + offsetof(struct iphdr, protocol)]
 *    map[r0]++
 * 3. attach prog_fd to raw socket via setsockopt()
 * 4. print number of received TCP/UDP packets every second
 */
int
main(int argc, char **argv)
{
    int sock, map_fd, prog_fd, key;
    long long value = 0, tcp_cnt, udp_cnt;

    map_fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(key),
                            sizeof(value), 256);
    if (map_fd < 0) {
        printf("failed to create map '%s'\n", strerror(errno));
        /* likely not run as root */
        return 1;
    }

    struct bpf_insn prog[] = {
        BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),        /* r6 = r1 */
        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol)),
                                /* r0 = ip->proto */
        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4),
                                /* *(u32 *)(fp - 4) = r0 */
        BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),       /* r2 = fp */
        BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),      /* r2 = r2 - 4 */
        BPF_LD_MAP_FD(BPF_REG_1, map_fd),           /* r1 = map_fd */
        BPF_CALL_FUNC(BPF_FUNC_map_lookup_elem),
                                /* r0 = map_lookup(r1, r2) */
```

```
        BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
                                /* if (r0 == 0) goto pc+2 */
        BPF_MOV64_IMM(BPF_REG_1, 1),                /* r1 = 1 */
        BPF_XADD(BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0),
                                /* lock *(u64 *) r0 += r1 */
        BPF_MOV64_IMM(BPF_REG_0, 0),                /* r0 = 0 */
        BPF_EXIT_INSN(),                            /* return r0 */
    };

    prog_fd = bpf_prog_load(BPF_PROG_TYPE_SOCKET_FILTER, prog,
                            sizeof(prog) / sizeof(prog[0]), "GPL");

    sock = open_raw_sock("lo");

    assert(setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd,
                      sizeof(prog_fd)) == 0);

    for (;;) {
        key = IPPROTO_TCP;
        assert(bpf_lookup_elem(map_fd, &key, &tcp_cnt) == 0);
        key = IPPROTO_UDP;
        assert(bpf_lookup_elem(map_fd, &key, &udp_cnt) == 0);
        printf("TCP %lld UDP %lld packets\n", tcp_cnt, udp_cnt);
        sleep(1);
    }

    return 0;
}
```

Some complete working code can be found in the *samples/bpf* directory in the kernel source tree.

RETURN VALUE

For a successful call, the return value depends on the operation:

BPF_MAP_CREATE
: The new file descriptor associated with the eBPF map.

BPF_PROG_LOAD
: The new file descriptor associated with the eBPF program.

All other commands
: Zero.

On error, −1 is returned, and *errno* is set appropriately.

ERRORS

E2BIG The eBPF program is too large or a map reached the *max_entries* limit (maximum number of elements).

EACCES
: For **BPF_PROG_LOAD**, even though all program instructions are valid, the program has been rejected because it was deemed unsafe. This may be because it may have accessed a disallowed memory region or an uninitialized stack/register or because the function constraints don't match the actual types or because there was a misaligned memory access. In this case, it is recommended to call **bpf**() again with *log_level = 1* and examine *log_buf* for the specific reason provided by the verifier.

EBADF
: *fd* is not an open file descriptor.

EFAULT
One of the pointers (*key* or *value* or *log_buf* or *insns*) is outside the accessible address space.

EINVAL
The value specified in *cmd* is not recognized by this kernel.

EINVAL
For **BPF_MAP_CREATE**, either *map_type* or attributes are invalid.

EINVAL
For **BPF_MAP_*_ELEM** commands, some of the fields of *union bpf_attr* that are not used by this command are not set to zero.

EINVAL
For **BPF_PROG_LOAD**, indicates an attempt to load an invalid program. eBPF programs can be deemed invalid due to unrecognized instructions, the use of reserved fields, jumps out of range, infinite loops or calls of unknown functions.

ENOENT
For **BPF_MAP_LOOKUP_ELEM** or **BPF_MAP_DELETE_ELEM**, indicates that the element with the given *key* was not found.

ENOMEM
Cannot allocate sufficient memory.

EPERM
The call was made without sufficient privilege (without the **CAP_SYS_ADMIN** capability).

VERSIONS

The **bpf**() system call first appeared in Linux 3.18.

CONFORMING TO

The **bpf**() system call is Linux-specific.

NOTES

In the current implementation, all **bpf**() commands require the caller to have the **CAP_SYS_ADMIN** capability.

eBPF objects (maps and programs) can be shared between processes. For example, after **fork**(2), the child inherits file descriptors referring to the same eBPF objects. In addition, file descriptors referring to eBPF objects can be transferred over UNIX domain sockets. File descriptors referring to eBPF objects can be duplicated in the usual way, using **dup**(2) and similar calls. An eBPF object is deallocated only after all file descriptors referring to the object have been closed.

eBPF programs can be written in a restricted C that is compiled (using the **clang** compiler) into eBPF bytecode. Various features are omitted from this restricted C, such as loops, global variables, variadic functions, floating-point numbers, and passing structures as function arguments. Some examples can be found in the *samples/bpf/*_kern.c* files in the kernel source tree.

The kernel contains a just-in-time (JIT) compiler that translates eBPF bytecode into native machine code for better performance. In kernels before Linux 4.15, the JIT compiler is disabled by default, but its operation can be controlled by writing one of the following integer strings to the file */proc/sys/net/core/bpf_jit_enable*:

0 Disable JIT compilation (default).

1 Normal compilation.

2 Debugging mode. The generated opcodes are dumped in hexadecimal into the kernel log. These opcodes can then be disassembled using the program *tools/net/bpf_jit_disasm.c* provided in the kernel source tree.

Since Linux 4.15, the kernel may configured with the **CONFIG_BPF_JIT_ALWAYS_ON** option. In this case, the JIT compiler is always enabled, and the *bpf_jit_enable* is initialized to 1 and is immutable. (This kernel configuration option was provided as a mitigation for one of the Spectre attacks against the BPF

interpreter.)

The JIT compiler for eBPF is currently available for the following architectures:

* x86-64 (since Linux 3.18; cBPF since Linux 3.0);
* ARM32 (since Linux 3.18; cBPF since Linux 3.4);
* SPARC 32 (since Linux 3.18; cBPF since Linux 3.5);
* ARM-64 (since Linux 3.18);
* s390 (since Linux 4.1; cBPF since Linux 3.7);
* PowerPC 64 (since Linux 4.8; cBPF since Linux 3.1);
* SPARC 64 (since Linux 4.12);
* x86-32 (since Linux 4.18);
* MIPS 64 (since Linux 4.18; cBPF since Linux 3.16);
* riscv (since Linux 5.1).

SEE ALSO

seccomp(2), **bpf-helpers**(7), **socket**(7), **tc**(8), **tc-bpf**(8)

Both classic and extended BPF are explained in the kernel source file *Documentation/networking/filter.txt*.

COLOPHON

This page is part of release 5.05 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.