# Linux Bpf Documentation

## The kernel development community

**Jul 14, 2020**

# CONTENTS

This directory contains documentation for the BPF (Berkeley Packet Filter) facility, with a focus on the extended BPF version (eBPF).

This kernel side documentation is still work in progress. The main textual documentation is (for historical reasons) described in Documentation/networking/filter.rst, which describe both classical and extended BPF instruction-set. The Cilium project also maintains a BPF and XDP Reference Guide that goes into great technical depth about the BPF Architecture.

The primary info for the bpf syscall is available in the man-pages for bpf(2).

CHAPTER

ONE

# BPF TYPE FORMAT (BTF)

## 1.1 BPF Type Format (BTF)

### 1.1.1 1. Introduction

BTF (BPF Type Format) is the metadata format which encodes the debug info related to BPF program/map. The name BTF was used initially to describe data types. The BTF was later extended to include function info for defined subroutines, and line info for source/line information.

The debug info is used for map pretty print, function signature, etc. The function signature enables better bpf program/function kernel symbol. The line info helps generate source annotated translated byte code, jited code and verifier log.

**The BTF specification contains two parts,**

- BTF kernel API
- BTF ELF file format

The kernel API is the contract between user space and kernel. The kernel verifies the BTF info before using it. The ELF file format is a user space contract between ELF file and libbpf loader.

The type and string sections are part of the BTF kernel API, describing the debug info (mostly types related) referenced by the bpf program. These two sections are discussed in details in 2. BTF Type and String Encoding.

### 1.1.2 2. BTF Type and String Encoding

The file `include/uapi/linux/btf.h` provides high-level definition of how types/strings are encoded.

The beginning of data blob must be:

```
struct btf_header {
    __u16   magic;
    __u8    version;
    __u8    flags;
    __u32   hdr_len;

    /* All offsets are in bytes relative to the end of this header */
    __u32   type_off;       /* offset of type section       */
```

(continues on next page)

(continued from previous page)

```
    __u32   type_len;       /* length of type section       */
    __u32   str_off;        /* offset of string section     */
    __u32   str_len;        /* length of string section     */
};
```

The magic is `0xeB9F`, which has different encoding for big and little endian systems, and can be used to test whether BTF is generated for big- or little-endian target. The `btf_header` is designed to be extensible with `hdr_len` equal to `sizeof(struct btf_header)` when a data blob is generated.

### 2.1 String Encoding

The first string in the string section must be a null string. The rest of string table is a concatenation of other null-terminated strings.

### 2.2 Type Encoding

The type id `0` is reserved for `void` type. The type section is parsed sequentially and type id is assigned to each recognized type starting from id `1`. Currently, the following types are supported:

```
#define BTF_KIND_INT            1       /* Integer      */
#define BTF_KIND_PTR            2       /* Pointer      */
#define BTF_KIND_ARRAY          3       /* Array        */
#define BTF_KIND_STRUCT         4       /* Struct       */
#define BTF_KIND_UNION          5       /* Union        */
#define BTF_KIND_ENUM           6       /* Enumeration  */
#define BTF_KIND_FWD            7       /* Forward      */
#define BTF_KIND_TYPEDEF        8       /* Typedef      */
#define BTF_KIND_VOLATILE       9       /* Volatile     */
#define BTF_KIND_CONST          10      /* Const        */
#define BTF_KIND_RESTRICT       11      /* Restrict     */
#define BTF_KIND_FUNC           12      /* Function     */
#define BTF_KIND_FUNC_PROTO     13      /* Function Proto       */
#define BTF_KIND_VAR            14      /* Variable     */
#define BTF_KIND_DATASEC        15      /* Section      */
```

Note that the type section encodes debug info, not just pure types. `BTF_KIND_FUNC` is not a type, and it represents a defined subprogram.

Each type contains the following common data:

```
struct btf_type {
    __u32 name_off;
    /* "info" bits arrangement
     * bits  0-15: vlen (e.g. # of struct's members)
     * bits 16-23: unused
     * bits 24-27: kind (e.g. int, ptr, array...etc)
     * bits 28-30: unused
     * bit     31: kind_flag, currently used by
     *             struct, union and fwd
     */
```

(continues on next page)

(continued from previous page)

```
    __u32 info;
    /* "size" is used by INT, ENUM, STRUCT and UNION.
     * "size" tells the size of the type it is describing.
     *
     * "type" is used by PTR, TYPEDEF, VOLATILE, CONST, RESTRICT,
     * FUNC and FUNC_PROTO.
     * "type" is a type_id referring to another type.
     */
    union {
            __u32 size;
            __u32 type;
    };
};
```

For certain kinds, the common data are followed by kind-specific data. The `name_off` in `struct btf_type` specifies the offset in the string table. The following sections detail encoding of each kind.

### 2.2.1 BTF_KIND_INT

**`struct btf_type` encoding requirement:**

- `name_off`: any valid offset
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_INT
- `info.vlen`: 0
- `size`: the size of the int type in bytes.

`btf_type` is followed by a `u32` with the following bits arrangement:

```
#define BTF_INT_ENCODING(VAL)   (((VAL) & 0x0f000000) >> 24)
#define BTF_INT_OFFSET(VAL)     (((VAL) & 0x00ff0000) >> 16)
#define BTF_INT_BITS(VAL)       ((VAL)  & 0x000000ff)
```

The `BTF_INT_ENCODING` has the following attributes:

```
#define BTF_INT_SIGNED  (1 << 0)
#define BTF_INT_CHAR    (1 << 1)
#define BTF_INT_BOOL    (1 << 2)
```

The `BTF_INT_ENCODING()` provides extra information: signedness, char, or bool, for the int type. The char and bool encoding are mostly useful for pretty print. At most one encoding can be specified for the int type.

The `BTF_INT_BITS()` specifies the number of actual bits held by this int type. For example, a 4-bit bitfield encodes `BTF_INT_BITS()` equals to 4. The `btf_type.size * 8` must be equal to or greater than `BTF_INT_BITS()` for the type. The maximum value of `BTF_INT_BITS()` is 128.

The `BTF_INT_OFFSET()` specifies the starting bit offset to calculate values for this int. For example, a bitfield struct member has:

- btf member bit offset 100 from the start of the structure,
- btf member pointing to an int type,
- the int type has `BTF_INT_OFFSET() = 2` and `BTF_INT_BITS() = 4`

Then in the struct memory layout, this member will occupy `4` bits starting from bits `100 + 2 = 102`.

Alternatively, the bitfield struct member can be the following to access the same bits as the above:

- btf member bit offset 102,
- btf member pointing to an int type,
- the int type has `BTF_INT_OFFSET() = 0` and `BTF_INT_BITS() = 4`

The original intention of `BTF_INT_OFFSET()` is to provide flexibility of bitfield encoding. Currently, both llvm and pahole generate `BTF_INT_OFFSET() = 0` for all int types.

### 2.2.2 BTF_KIND_PTR

**`struct btf_type` encoding requirement:**

- `name_off`: 0
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_PTR
- `info.vlen`: 0
- `type`: the pointee type of the pointer

No additional type data follow `btf_type`.

### 2.2.3 BTF_KIND_ARRAY

**`struct btf_type` encoding requirement:**

- `name_off`: 0
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_ARRAY
- `info.vlen`: 0
- `size/type`: 0, not used

`btf_type` is followed by one `struct btf_array`:

```
struct btf_array {
    __u32   type;
    __u32   index_type;
    __u32   nelems;
};
```

**The `struct btf_array` encoding:**

- `type`: the element type
- `index_type`: the index type
- `nelems`: the number of elements for this array (`0` is also allowed).

The `index_type` can be any regular int type (`u8`, `u16`, `u32`, `u64`, `unsigned __int128`). The original design of including `index_type` follows DWARF, which has an `index_type` for its array type. Currently in BTF, beyond type verification, the `index_type` is not used.

The `struct btf_array` allows chaining through element type to represent multi-dimensional arrays. For example, for `int a[5][6]`, the following type information illustrates the chaining:

- [1]: int
- [2]: array, `btf_array.type = [1]`, `btf_array.nelems = 6`
- [3]: array, `btf_array.type = [2]`, `btf_array.nelems = 5`

Currently, both pahole and llvm collapse multidimensional array into one-dimensional array, e.g., for `a[5][6]`, the `btf_array.nelems` is equal to `30`. This is because the original use case is map pretty print where the whole array is dumped out so one-dimensional array is enough. As more BTF usage is explored, pahole and llvm can be changed to generate proper chained representation for multidimensional arrays.

### 2.2.4 BTF_KIND_STRUCT

### 2.2.5 BTF_KIND_UNION

**`struct btf_type` encoding requirement:**

- `name_off`: 0 or offset to a valid C identifier
- `info.kind_flag`: 0 or 1
- `info.kind`: BTF_KIND_STRUCT or BTF_KIND_UNION
- `info.vlen`: the number of struct/union members
- `info.size`: the size of the struct/union in bytes

`btf_type` is followed by `info.vlen` number of `struct btf_member`.:

```
struct btf_member {
    __u32   name_off;
    __u32   type;
    __u32   offset;
};
```

**`struct btf_member` encoding:**

- `name_off`: offset to a valid C identifier
- `type`: the member type

- `offset`: <see below>

If the type info `kind_flag` is not set, the offset contains only bit offset of the member. Note that the base type of the bitfield can only be int or enum type. If the bitfield size is 32, the base type can be either int or enum type. If the bitfield size is not 32, the base type must be int, and int type `BTF_INT_BITS()` encodes the bitfield size.

If the `kind_flag` is set, the `btf_member.offset` contains both member bitfield size and bit offset. The bitfield size and bit offset are calculated as below.:

```
#define BTF_MEMBER_BITFIELD_SIZE(val)   ((val) >> 24)
#define BTF_MEMBER_BIT_OFFSET(val)      ((val) & 0xffffff)
```

In this case, if the base type is an int type, it must be a regular int type:

- `BTF_INT_OFFSET()` must be 0.
- `BTF_INT_BITS()` must be equal to `{1,2,4,8,16} * 8`.

The following kernel patch introduced `kind_flag` and explained why both modes exist:

https://github.com/torvalds/linux/commit/9d5f9f701b1891466fb3dbb1806ad97716f95cc3#diff-fa650a64fdd3968396883d2fe8215ff3

### 2.2.6 BTF_KIND_ENUM

**`struct btf_type` encoding requirement:**

- `name_off`: 0 or offset to a valid C identifier
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_ENUM
- `info.vlen`: number of enum values
- `size`: 4

`btf_type` is followed by `info.vlen` number of `struct btf_enum`.:

```
struct btf_enum {
    __u32   name_off;
    __s32   val;
};
```

**The `btf_enum` encoding:**

- `name_off`: offset to a valid C identifier
- `val`: any value

#### 2.2.7 BTF_KIND_FWD

**`struct btf_type` encoding requirement:**

- `name_off`: offset to a valid C identifier
- `info.kind_flag`: 0 for struct, 1 for union
- `info.kind`: BTF_KIND_FWD
- `info.vlen`: 0
- `type`: 0

No additional type data follow `btf_type`.

#### 2.2.8 BTF_KIND_TYPEDEF

**`struct btf_type` encoding requirement:**

- `name_off`: offset to a valid C identifier
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_TYPEDEF
- `info.vlen`: 0
- `type`: the type which can be referred by name at `name_off`

No additional type data follow `btf_type`.

#### 2.2.9 BTF_KIND_VOLATILE

**`struct btf_type` encoding requirement:**

- `name_off`: 0
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_VOLATILE
- `info.vlen`: 0
- `type`: the type with `volatile` qualifier

No additional type data follow `btf_type`.

#### 2.2.10 BTF_KIND_CONST

**`struct btf_type` encoding requirement:**

- `name_off`: 0
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_CONST
- `info.vlen`: 0

- `type`: the type with `const` qualifier

No additional type data follow `btf_type`.

### 2.2.11 BTF_KIND_RESTRICT

**`struct btf_type` encoding requirement:**

- `name_off`: 0
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_RESTRICT
- `info.vlen`: 0
- `type`: the type with `restrict` qualifier

No additional type data follow `btf_type`.

### 2.2.12 BTF_KIND_FUNC

**`struct btf_type` encoding requirement:**

- `name_off`: offset to a valid C identifier
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_FUNC
- `info.vlen`: 0
- `type`: a BTF_KIND_FUNC_PROTO type

No additional type data follow `btf_type`.

A BTF_KIND_FUNC defines not a type, but a subprogram (function) whose signature is defined by `type`. The subprogram is thus an instance of that type. The BTF_KIND_FUNC may in turn be referenced by a func_info in the 4.2 .BTF.ext section (ELF) or in the arguments to 3.3 BPF_PROG_LOAD (ABI).

### 2.2.13 BTF_KIND_FUNC_PROTO

**`struct btf_type` encoding requirement:**

- `name_off`: 0
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_FUNC_PROTO
- `info.vlen`: # of parameters
- `type`: the return type

`btf_type` is followed by `info.vlen` number of `struct btf_param`.:

```
struct btf_param {
    __u32   name_off;
    __u32   type;
};
```

If a BTF_KIND_FUNC_PROTO type is referred by a BTF_KIND_FUNC type, then `btf_param.name_off` must point to a valid C identifier except for the possible last argument representing the variable argument. The btf_param.type refers to parameter type.

If the function has variable arguments, the last parameter is encoded with `name_off = 0` and `type = 0`.

### 2.2.14 BTF_KIND_VAR

**`struct btf_type` encoding requirement:**

- `name_off`: offset to a valid C identifier
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_VAR
- `info.vlen`: 0
- `type`: the type of the variable

`btf_type` is followed by a single `struct btf_variable` with the following data:

```
struct btf_var {
    __u32   linkage;
};
```

**`struct btf_var` encoding:**

- **`linkage`: currently only static variable 0, or globally allocated** variable in ELF sections 1

Not all type of global variables are supported by LLVM at this point. The following is currently available:

- static variables with or without section attributes
- global variables with section attributes

The latter is for future extraction of map key/value type id’ s from a map definition.

#### 2.2.15 BTF_KIND_DATASEC

**`struct btf_type` encoding requirement:**

- **`name_off`: offset to a valid name associated with a variable or** one of .data/.bss/.rodata
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_DATASEC
- `info.vlen`: # of variables
- **`size`: total section size in bytes (0 at compilation time, patched** to actual size by BPF loaders such as libbpf)

`btf_type` is followed by `info.vlen` number of `struct btf_var_secinfo`.:

```
struct btf_var_secinfo {
    __u32   type;
    __u32   offset;
    __u32   size;
};
```

**`struct btf_var_secinfo` encoding:**

- `type`: the type of the BTF_KIND_VAR variable
- `offset`: the in-section offset of the variable
- `size`: the size of the variable in bytes

### 1.1.3 3. BTF Kernel API

**The following bpf syscall command involves BTF:**

- BPF_BTF_LOAD: load a blob of BTF data into kernel
- BPF_MAP_CREATE: map creation with btf key and value type info.
- BPF_PROG_LOAD: prog load with btf function and line info.
- BPF_BTF_GET_FD_BY_ID: get a btf fd
- BPF_OBJ_GET_INFO_BY_FD: btf, func_info, line_info and other btf related info are returned.

The workflow typically looks like:

```
Application:
    BPF_BTF_LOAD
        |
        v
    BPF_MAP_CREATE and BPF_PROG_LOAD
        |
        V
    ......

Introspection tool:
```

(continues on next page)

(continued from previous page)

```
    ......
    BPF_{PROG,MAP}_GET_NEXT_ID (get prog/map id's)
        |
        V
    BPF_{PROG,MAP}_GET_FD_BY_ID (get a prog/map fd)
        |
        V
    BPF_OBJ_GET_INFO_BY_FD (get bpf_prog_info/bpf_map_info with btf_id)
        |                                          |
        V                                          |
    BPF_BTF_GET_FD_BY_ID (get btf_fd)              |
        |                                          |
        V                                          |
    BPF_OBJ_GET_INFO_BY_FD (get btf)               |
        |                                          |
        V                                          V
    pretty print types, dump func signatures and line info, etc.
```

### 3.1 BPF_BTF_LOAD

Load a blob of BTF data into kernel. A blob of data, described in 2. BTF Type and String Encoding, can be directly loaded into the kernel. A `btf_fd` is returned to a userspace.

### 3.2 BPF_MAP_CREATE

A map can be created with `btf_fd` and specified key/value type id.:

```
__u32   btf_fd;         /* fd pointing to a BTF type data */
__u32   btf_key_type_id;        /* BTF type_id of the key */
__u32   btf_value_type_id;      /* BTF type_id of the value */
```

In libbpf, the map can be defined with extra annotation like below:

```
struct bpf_map_def SEC("maps") btf_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(struct ipv_counts),
    .max_entries = 4,
};
BPF_ANNOTATE_KV_PAIR(btf_map, int, struct ipv_counts);
```

Here, the parameters for macro BPF_ANNOTATE_KV_PAIR are map name, key and value types for the map. During ELF parsing, libbpf is able to extract key/value type_id's and assign them to BPF_MAP_CREATE attributes automatically.

### 3.3 BPF_PROG_LOAD

During prog_load, func_info and line_info can be passed to kernel with proper values for the following attributes:

```
__u32           insn_cnt;
__aligned_u64   insns;
......
__u32           prog_btf_fd;    /* fd pointing to BTF type data */
__u32           func_info_rec_size;     /* userspace bpf_func_info size */
__aligned_u64   func_info;      /* func info */
__u32           func_info_cnt;  /* number of bpf_func_info records */
__u32           line_info_rec_size;     /* userspace bpf_line_info size */
__aligned_u64   line_info;      /* line info */
__u32           line_info_cnt;  /* number of bpf_line_info records */
```

The func_info and line_info are an array of below, respectively.:

```
struct bpf_func_info {
    __u32   insn_off; /* [0, insn_cnt - 1] */
    __u32   type_id;  /* pointing to a BTF_KIND_FUNC type */
};
struct bpf_line_info {
    __u32   insn_off; /* [0, insn_cnt - 1] */
    __u32   file_name_off; /* offset to string table for the filename */
    __u32   line_off; /* offset to string table for the source line */
    __u32   line_col; /* line number and column number */
};
```

func_info_rec_size is the size of each func_info record, and line_info_rec_size is the size of each line_info record. Passing the record size to kernel make it possible to extend the record itself in the future.

**Below are requirements for func_info:**

- func_info[0].insn_off must be 0.
- the func_info insn_off is in strictly increasing order and matches bpf func boundaries.

**Below are requirements for line_info:**

- the first insn in each func must have a line_info record pointing to it.
- the line_info insn_off is in strictly increasing order.

For line_info, the line number and column number are defined as below:

```
#define BPF_LINE_INFO_LINE_NUM(line_col)        ((line_col) >> 10)
#define BPF_LINE_INFO_LINE_COL(line_col)        ((line_col) & 0x3ff)
```

#### 3.4 BPF_{PROG,MAP}_GET_NEXT_ID

In kernel, every loaded program, map or btf has a unique id. The id won't change during the lifetime of a program, map, or btf.

The bpf syscall command BPF_{PROG,MAP}_GET_NEXT_ID returns all id's, one for each command, to user space, for bpf program or maps, respectively, so an inspection tool can inspect all programs and maps.

#### 3.5 BPF_{PROG,MAP}_GET_FD_BY_ID

An introspection tool cannot use id to get details about program or maps. A file descriptor needs to be obtained first for reference-counting purpose.

#### 3.6 BPF_OBJ_GET_INFO_BY_FD

Once a program/map fd is acquired, an introspection tool can get the detailed information from kernel about this fd, some of which are BTF-related. For example, `bpf_map_info` returns `btf_id` and key/value type ids. `bpf_prog_info` returns `btf_id`, func_info, and line info for translated bpf byte codes, and jited_line_info.

#### 3.7 BPF_BTF_GET_FD_BY_ID

With `btf_id` obtained in `bpf_map_info` and `bpf_prog_info`, bpf syscall command BPF_BTF_GET_FD_BY_ID can retrieve a btf fd. Then, with command BPF_OBJ_GET_INFO_BY_FD, the btf blob, originally loaded into the kernel with BPF_BTF_LOAD, can be retrieved.

With the btf blob, `bpf_map_info`, and `bpf_prog_info`, an introspection tool has full btf knowledge and is able to pretty print map key/values, dump func signatures and line info, along with byte/jit codes.

### 1.1.4 4. ELF File Format Interface

#### 4.1 .BTF section

The .BTF section contains type and string data. The format of this section is same as the one describe in 2. BTF Type and String Encoding.

#### 4.2 .BTF.ext section

The .BTF.ext section encodes func_info and line_info which needs loader manipulation before loading into the kernel.

The specification for .BTF.ext section is defined at `tools/lib/bpf/btf.h` and `tools/lib/bpf/btf.c`.

The current header of .BTF.ext section:

```
struct btf_ext_header {
    __u16   magic;
    __u8    version;
    __u8    flags;
    __u32   hdr_len;

    /* All offsets are in bytes relative to the end of this header */
    __u32   func_info_off;
    __u32   func_info_len;
    __u32   line_info_off;
    __u32   line_info_len;
};
```

It is very similar to .BTF section. Instead of type/string section, it contains func_info and line_info section. See 3.3 BPF_PROG_LOAD for details about func_info and line_info record format.

The func_info is organized as below.:

```
func_info_rec_size
btf_ext_info_sec for section #1 /* func_info for section #1 */
btf_ext_info_sec for section #2 /* func_info for section #2 */
...
```

`func_info_rec_size` specifies the size of `bpf_func_info` structure when .BTF.ext is generated. `btf_ext_info_sec`, defined below, is a collection of func_info for each specific ELF section.:

```
struct btf_ext_info_sec {
   __u32   sec_name_off; /* offset to section name */
   __u32   num_info;
   /* Followed by num_info * record_size number of bytes */
   __u8    data[0];
};
```

Here, num_info must be greater than 0.

The line_info is organized as below.:

```
line_info_rec_size
btf_ext_info_sec for section #1 /* line_info for section #1 */
btf_ext_info_sec for section #2 /* line_info for section #2 */
...
```

`line_info_rec_size` specifies the size of `bpf_line_info` structure when .BTF.ext is generated.

The interpretation of `bpf_func_info->insn_off` and `bpf_line_info->insn_off` is different between kernel API and ELF API. For kernel API, the `insn_off` is the instruction offset in the unit of `struct bpf_insn`. For ELF API, the `insn_off` is the byte offset from the beginning of section (`btf_ext_info_sec->sec_name_off`).

### 1.1.5 5. Using BTF

#### 5.1 bpftool map pretty print

With BTF, the map key/value can be printed based on fields rather than simply raw bytes. This is especially valuable for large structure or if your data structure has bitfields. For example, for the following map,:

```
enum A { A1, A2, A3, A4, A5 };
typedef enum A ___A;
struct tmp_t {
     char a1:4;
     int  a2:4;
     int  :4;
     __u32 a3:4;
     int b;
     ___A b1:4;
     enum A b2:4;
};
struct bpf_map_def SEC("maps") tmpmap = {
     .type = BPF_MAP_TYPE_ARRAY,
     .key_size = sizeof(__u32),
     .value_size = sizeof(struct tmp_t),
     .max_entries = 1,
};
BPF_ANNOTATE_KV_PAIR(tmpmap, int, struct tmp_t);
```

bpftool is able to pretty print like below:

```
[{
      "key": 0,
      "value": {
          "a1": 0x2,
          "a2": 0x4,
          "a3": 0x6,
          "b": 7,
          "b1": 0x8,
          "b2": 0xa
      }
  }
]
```

#### 5.2 bpftool prog dump

The following is an example showing how func_info and line_info can help prog dump with better kernel symbol names, function prototypes and line information.:

```
$ bpftool prog dump jited pinned /sys/fs/bpf/test_btf_haskv
[...]
int test_long_fname_2(struct dummy_tracepoint_args * arg):
bpf_prog_44a040bf25481309_test_long_fname_2:
; static int test_long_fname_2(struct dummy_tracepoint_args *arg)
   0:   push   %rbp
   1:   mov    %rsp,%rbp
```

(continues on next page)

(continued from previous page)

```
   4:   sub    $0x30,%rsp
   b:   sub    $0x28,%rbp
   f:   mov    %rbx,0x0(%rbp)
  13:   mov    %r13,0x8(%rbp)
  17:   mov    %r14,0x10(%rbp)
  1b:   mov    %r15,0x18(%rbp)
  1f:   xor    %eax,%eax
  21:   mov    %rax,0x20(%rbp)
  25:   xor    %esi,%esi
; int key = 0;
  27:   mov    %esi,-0x4(%rbp)
; if (!arg->sock)
  2a:   mov    0x8(%rdi),%rdi
; if (!arg->sock)
  2e:   cmp    $0x0,%rdi
  32:   je     0x0000000000000070
  34:   mov    %rbp,%rsi
; counts = bpf_map_lookup_elem(&btf_map, &key);
[...]
```

### 5.3 Verifier Log

The following is an example of how line_info can help debugging verification failure.:

```
   /* The code at tools/testing/selftests/bpf/test_xdp_noinline.c
    * is modified as below.
    */
   data = (void *)(long)xdp->data;
   data_end = (void *)(long)xdp->data_end;
   /*
   if (data + 4 > data_end)
           return XDP_DROP;
   */
   *(u32 *)data = dst->dst;

$ bpftool prog load ./test_xdp_noinline.o /sys/fs/bpf/test_xdp_noinline␣
↪type xdp
    ; data = (void *)(long)xdp->data;
    224: (79) r2 = *(u64 *)(r10 -112)
    225: (61) r2 = *(u32 *)(r2 +0)
    ; *(u32 *)data = dst->dst;
    226: (63) *(u32 *)(r2 +0) = r1
    invalid access to packet, off=0 size=4, R2(id=0,off=0,r=0)
    R2 offset is outside of the packet
```

### 1.1.6 6. BTF Generation

You need latest pahole

https://git.kernel.org/pub/scm/devel/pahole/pahole.git/

or llvm (8.0 or later). The pahole acts as a dwarf2btf converter. It doesn't support .BTF.ext and btf BTF_KIND_FUNC type yet. For example,:

```
-bash-4.4$ cat t.c
struct t {
  int a:2;
  int b:3;
  int c:2;
} g;
-bash-4.4$ gcc -c -O2 -g t.c
-bash-4.4$ pahole -JV t.o
File t.o:
[1] STRUCT t kind_flag=1 size=4 vlen=3
        a type_id=2 bitfield_size=2 bits_offset=0
        b type_id=2 bitfield_size=3 bits_offset=2
        c type_id=2 bitfield_size=2 bits_offset=5
[2] INT int size=4 bit_offset=0 nr_bits=32 encoding=SIGNED
```

The llvm is able to generate .BTF and .BTF.ext directly with -g for bpf target only. The assembly code (-S) is able to show the BTF encoding in assembly format.:

```
-bash-4.4$ cat t2.c
typedef int __int32;
struct t2 {
  int a2;
  int (*f2)(char q1, __int32 q2, ...);
  int (*f3)();
} g2;
int main() { return 0; }
int test() { return 0; }
-bash-4.4$ clang -c -g -O2 -target bpf t2.c
-bash-4.4$ readelf -S t2.o
  ......
  [ 8] .BTF              PROGBITS         0000000000000000  00000247
       000000000000016e  0000000000000000           0     0     1
  [ 9] .BTF.ext          PROGBITS         0000000000000000  000003b5
       0000000000000060  0000000000000000           0     0     1
  [10] .rel.BTF.ext      REL              0000000000000000  000007e0
       0000000000000040  0000000000000010          16     9     8
  ......
-bash-4.4$ clang -S -g -O2 -target bpf t2.c
-bash-4.4$ cat t2.s
  ......
        .section        .BTF,"",@progbits
        .short  60319                   # 0xeb9f
        .byte   1
        .byte   0
        .long   24
        .long   0
        .long   220
        .long   220
```

(continues on next page)

(continued from previous page)

```
        .long   122
        .long   0                       # BTF_KIND_FUNC_PROTO(id = 1)
        .long   218103808               # 0xd000000
        .long   2
        .long   83                      # BTF_KIND_INT(id = 2)
        .long   16777216                # 0x1000000
        .long   4
        .long   16777248                # 0x1000020
  ......
        .byte   0                       # string offset=0
        .ascii  ".text"                 # string offset=1
        .byte   0
        .ascii  "/home/yhs/tmp-pahole/t2.c" # string offset=7
        .byte   0
        .ascii  "int main() { return 0; }" # string offset=33
        .byte   0
        .ascii  "int test() { return 0; }" # string offset=58
        .byte   0
        .ascii  "int"                   # string offset=83
  ......
        .section        .BTF.ext,"",@progbits
        .short  60319                   # 0xeb9f
        .byte   1
        .byte   0
        .long   24
        .long   0
        .long   28
        .long   28
        .long   44
        .long   8                       # FuncInfo
        .long   1                       # FuncInfo section string offset=1
        .long   2
        .long   .Lfunc_begin0
        .long   3
        .long   .Lfunc_begin1
        .long   5
        .long   16                      # LineInfo
        .long   1                       # LineInfo section string offset=1
        .long   2
        .long   .Ltmp0
        .long   7
        .long   33
        .long   7182                    # Line 7 Col 14
        .long   .Ltmp3
        .long   7
        .long   58
        .long   8206                    # Line 8 Col 14
```

### 1.1.7 7. Testing

Kernel bpf selftest test_btf.c provides extensive set of BTF-related tests.

CHAPTER

TWO

# FREQUENTLY ASKED QUESTIONS (FAQ)

Two sets of Questions and Answers (Q&A) are maintained.

## 2.1 BPF Design Q&A

BPF extensibility and applicability to networking, tracing, security in the linux kernel and several user space implementations of BPF virtual machine led to a number of misunderstanding on what BPF actually is. This short QA is an attempt to address that and outline a direction of where BPF is heading long term.

- Questions and Answers
  - Q: Is BPF a generic instruction set similar to x64 and arm64?
  - Q: Is BPF a generic virtual machine ?
  - BPF is generic instruction set with C calling convention.
    - Q: Why C calling convention was chosen?
    - Q: Can multiple return values be supported in the future?
    - Q: Can more than 5 function arguments be supported in the future?
  - Q: Can BPF programs access instruction pointer or return address?
  - Q: Can BPF programs access stack pointer ?
  - Q: Does C-calling convention diminishes possible use cases?
  - Q: Does it mean that 'innovative' extensions to BPF code are disallowed?
  - Q: Can loops be supported in a safe way?
  - Q: What are the verifier limits?
  - Instruction level questions
    - Q: LD_ABS and LD_IND instructions vs C code
    - Q: BPF instructions mapping not one-to-one to native CPU
    - Q: Why BPF_DIV instruction doesn' t map to x64 div?
    - Q: Why there is no BPF_SDIV for signed divide operation?

### 2.1.1 Questions and Answers

#### Q: Is BPF a generic instruction set similar to x64 and arm64?

A: NO.

#### Q: Is BPF a generic virtual machine ?

A: NO.

#### BPF is generic instruction set with C calling convention.

#### Q: Why C calling convention was chosen?

A: Because BPF programs are designed to run in the linux kernel which is written in C, hence BPF defines instruction set compatible with two most used architectures x64 and arm64 (and takes into consideration important quirks of other architectures) and defines calling convention that is compatible with C calling convention of the linux kernel on those architectures.

### Q: Can multiple return values be supported in the future?

A: NO. BPF allows only register R0 to be used as return value.

### Q: Can more than 5 function arguments be supported in the future?

A: NO. BPF calling convention only allows registers R1-R5 to be used as arguments. BPF is not a standalone instruction set. (unlike x64 ISA that allows msft, cdecl and other conventions)

### Q: Can BPF programs access instruction pointer or return address?

A: NO.

### Q: Can BPF programs access stack pointer ?

A: NO.

Only frame pointer (register R10) is accessible. From compiler point of view it's necessary to have stack pointer. For example, LLVM defines register R11 as stack pointer in its BPF backend, but it makes sure that generated code never uses it.

### Q: Does C-calling convention diminishes possible use cases?

A: YES.

BPF design forces addition of major functionality in the form of kernel helper functions and kernel objects like BPF maps with seamless interoperability between them. It lets kernel call into BPF programs and programs call kernel helpers with zero overhead, as all of them were native C code. That is particularly the case for JITed BPF programs that are indistinguishable from native kernel C code.

### Q: Does it mean that 'innovative' extensions to BPF code are disallowed?

A: Soft yes.

At least for now, until BPF core has support for bpf-to-bpf calls, indirect calls, loops, global variables, jump tables, read-only sections, and all other normal constructs that C code can produce.

### Q: Can loops be supported in a safe way?

A: It’s not clear yet.

BPF developers are trying to find a way to support bounded loops.

### Q: What are the verifier limits?

A: The only limit known to the user space is BPF_MAXINSNS (4096). It’s the maximum number of instructions that the unprivileged bpf program can have. The verifier has various internal limits. Like the maximum number of instructions that can be explored during program analysis. Currently, that limit is set to 1 million. Which essentially means that the largest program can consist of 1 million NOP instructions. There is a limit to the maximum number of subsequent branches, a limit to the number of nested bpf-to-bpf calls, a limit to the number of the verifier states per instruction, a limit to the number of maps used by the program. All these limits can be hit with a sufficiently complex program. There are also non-numerical limits that can cause the program to be rejected. The verifier used to recognize only pointer + constant expressions. Now it can recognize pointer + bounded_register. bpf_lookup_map_elem(key) had a requirement that ‘key’ must be a pointer to the stack. Now, ‘key’ can be a pointer to map value. The verifier is steadily getting ‘smarter’. The limits are being removed. The only way to know that the program is going to be accepted by the verifier is to try to load it. The bpf development process guarantees that the future kernel versions will accept all bpf programs that were accepted by the earlier versions.

## Instruction level questions

### Q: LD_ABS and LD_IND instructions vs C code

Q: How come LD_ABS and LD_IND instruction are present in BPF whereas C code cannot express them and has to use builtin intrinsics?

A: This is artifact of compatibility with classic BPF. Modern networking code in BPF performs better without them. See ‘direct packet access’.

### Q: BPF instructions mapping not one-to-one to native CPU

Q: It seems not all BPF instructions are one-to-one to native CPU. For example why BPF_JNE and other compare and jumps are not cpu-like?

A: This was necessary to avoid introducing flags into ISA which are impossible to make generic and efficient across CPU architectures.

### Q: Why BPF_DIV instruction doesn't map to x64 div?

A: Because if we picked one-to-one relationship to x64 it would have made it more complicated to support on arm64 and other archs. Also it needs div-by-zero runtime check.

### Q: Why there is no BPF_SDIV for signed divide operation?

A: Because it would be rarely used. llvm errors in such case and prints a suggestion to use unsigned divide instead.

### Q: Why BPF has implicit prologue and epilogue?

A: Because architectures like sparc have register windows and in general there are enough subtle differences between architectures, so naive store return address into stack won't work. Another reason is BPF has to be safe from division by zero (and legacy exception path of LD_ABS insn). Those instructions need to invoke epilogue and return implicitly.

### Q: Why BPF_JLT and BPF_JLE instructions were not introduced in the beginning?

A: Because classic BPF didn't have them and BPF authors felt that compiler workaround would be acceptable. Turned out that programs lose performance due to lack of these compare instructions and they were added. These two instructions is a perfect example what kind of new BPF instructions are acceptable and can be added in the future. These two already had equivalent instructions in native CPUs. New instructions that don't have one-to-one mapping to HW instructions will not be accepted.

### Q: BPF 32-bit subregister requirements

Q: BPF 32-bit subregisters have a requirement to zero upper 32-bits of BPF registers which makes BPF inefficient virtual machine for 32-bit CPU architectures and 32-bit HW accelerators. Can true 32-bit registers be added to BPF in the future?

A: NO.

But some optimizations on zero-ing the upper 32 bits for BPF registers are available, and can be leveraged to improve the performance of JITed BPF programs for 32-bit architectures.

Starting with version 7, LLVM is able to generate instructions that operate on 32-bit subregisters, provided the option -mattr=+alu32 is passed for compiling a program. Furthermore, the verifier can now mark the instructions for which zeroing the upper bits of the destination register is required, and insert an explicit zero-extension (zext) instruction (a mov32 variant). This means that for architectures without zext hardware support, the JIT back-ends do not need to clear the upper bits for subregisters written by alu32 instructions or narrow loads. Instead, the

back-ends simply need to support code generation for that mov32 variant, and to overwrite bpf_jit_needs_zext() to make it return "true" (in order to enable zext insertion in the verifier).

Note that it is possible for a JIT back-end to have partial hardware support for zext. In that case, if verifier zext insertion is enabled, it could lead to the insertion of unnecessary zext instructions. Such instructions could be removed by creating a simple peephole inside the JIT back-end: if one instruction has hardware support for zext and if the next instruction is an explicit zext, then the latter can be skipped when doing the code generation.

### Q: Does BPF have a stable ABI?

A: YES. BPF instructions, arguments to BPF programs, set of helper functions and their arguments, recognized return codes are all part of ABI. However there is one specific exception to tracing programs which are using helpers like bpf_probe_read() to walk kernel internal data structures and compile with kernel internal headers. Both of these kernel internals are subject to change and can break with newer kernels such that the program needs to be adapted accordingly.

### Q: How much stack space a BPF program uses?

A: Currently all program types are limited to 512 bytes of stack space, but the verifier computes the actual amount of stack used and both interpreter and most JITed code consume necessary amount.

### Q: Can BPF be offloaded to HW?

A: YES. BPF HW offload is supported by NFP driver.

### Q: Does classic BPF interpreter still exist?

A: NO. Classic BPF programs are converted into extend BPF instructions.

### Q: Can BPF call arbitrary kernel functions?

A: NO. BPF programs can only call a set of helper functions which is defined for every program type.

#### Q: Can BPF overwrite arbitrary kernel memory?

A: NO.

Tracing bpf programs can read arbitrary memory with bpf_probe_read() and bpf_probe_read_str() helpers. Networking programs cannot read arbitrary memory, since they don't have access to these helpers. Programs can never read or write arbitrary memory directly.

#### Q: Can BPF overwrite arbitrary user memory?

A: Sort-of.

Tracing BPF programs can overwrite the user memory of the current task with bpf_probe_write_user(). Every time such program is loaded the kernel will print warning message, so this helper is only useful for experiments and prototypes. Tracing BPF programs are root only.

#### Q: bpf_trace_printk() helper warning

Q: When bpf_trace_printk() helper is used the kernel prints nasty warning message. Why is that?

A: This is done to nudge program authors into better interfaces when programs need to pass data to user space. Like bpf_perf_event_output() can be used to efficiently stream data via perf ring buffer. BPF maps can be used for asynchronous data sharing between kernel and user space. bpf_trace_printk() should only be used for debugging.

#### Q: New functionality via kernel modules?

Q: Can BPF functionality such as new program or map types, new helpers, etc be added out of kernel module code?

A: NO.

## 2.2 HOWTO interact with BPF subsystem

This document provides information for the BPF subsystem about various workflows related to reporting bugs, submitting patches, and queueing patches for stable kernels.

For general information about submitting patches, please refer to Documentation/process/. This document only describes additional specifics related to BPF.

  - Q: To which mailing list do I need to submit my BPF patches?
  - Q: Where can I find patches currently under discussion for BPF subsystem?
  - Q: How do the changes make their way into Linux?
  - Q: How do I indicate which tree (bpf vs. bpf-next) my patch should be applied to?
  - Q: What does it mean when a patch gets applied to bpf or bpf-next tree?
  - Q: How long do I need to wait for feedback on my BPF patches?
  - Q: How often do you send pull requests to major kernel trees like net or net-next?
  - Q: Are patches applied to bpf-next when the merge window is open?
  - Q: Verifier changes and test cases
  - Q: samples/bpf preference vs selftests?
  - Q: When should I add code to the bpftool?
  - Q: When should I add code to iproute2's BPF loader?
  - Q: Do you accept patches as well for iproute2's BPF loader?
  - Q: What is the minimum requirement before I submit my BPF patches?
  - Q: Features changing BPF JIT and/or LLVM
- Stable submission
  - Q: I need a specific BPF commit in stable kernels. What should I do?
  - Q: Do you also backport to kernels not currently maintained as stable?
  - Q: The BPF patch I am about to submit needs to go to stable as well
  - Q: Queue stable patches
- Testing patches
  - Q: How to run BPF selftests
  - Q: Which BPF kernel selftests version should I run my kernel against?
- LLVM
  - Q: Where do I find LLVM with BPF support?
  - Q: Got it, so how do I build LLVM manually anyway?
  - Q: Reporting LLVM BPF issues
  - Q: New BPF instruction for kernel and LLVM
  - Q: clang flag for target bpf?

### 2.2.1 Reporting bugs

#### Q: How do I report bugs for BPF kernel code?

A: Since all BPF kernel development as well as bpftool and iproute2 BPF loader development happens through the bpf kernel mailing list, please report any found issues around BPF to the following mailing list:

bpf@vger.kernel.org

This may also include issues related to XDP, BPF tracing, etc.

Given netdev has a high volume of traffic, please also add the BPF maintainers to Cc (from kernel MAINTAINERS file):

- Alexei Starovoitov <ast@kernel.org>
- Daniel Borkmann <daniel@iogearbox.net>

In case a buggy commit has already been identified, make sure to keep the actual commit authors in Cc as well for the report. They can typically be identified through the kernel's git tree.

**Please do NOT report BPF issues to bugzilla.kernel.org since it is a guarantee that the reported issue will be overlooked.**

### 2.2.2 Submitting patches

#### Q: To which mailing list do I need to submit my BPF patches?

A: Please submit your BPF patches to the bpf kernel mailing list:

bpf@vger.kernel.org

In case your patch has changes in various different subsystems (e.g. networking, tracing, security, etc), make sure to Cc the related kernel mailing lists and maintainers from there as well, so they are able to review the changes and provide their Acked-by's to the patches.

#### Q: Where can I find patches currently under discussion for BPF subsystem?

A: All patches that are Cc'ed to netdev are queued for review under netdev patchwork project:

http://patchwork.ozlabs.org/project/netdev/list/

Those patches which target BPF, are assigned to a 'bpf' delegate for further processing from BPF maintainers. The current queue with patches under review can be found at:

https://patchwork.ozlabs.org/project/netdev/list/?delegate=77147

Once the patches have been reviewed by the BPF community as a whole and approved by the BPF maintainers, their status in patchwork will be changed to 'Accepted' and the submitter will be notified by mail. This means that the patches

look good from a BPF perspective and have been applied to one of the two BPF kernel trees.

In case feedback from the community requires a respin of the patches, their status in patchwork will be set to 'Changes Requested', and purged from the current review queue. Likewise for cases where patches would get rejected or are not applicable to the BPF trees (but assigned to the 'bpf' delegate).

### Q: How do the changes make their way into Linux?

A: There are two BPF kernel trees (git repositories). Once patches have been accepted by the BPF maintainers, they will be applied to one of the two BPF trees:

- https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/
- https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/

The bpf tree itself is for fixes only, whereas bpf-next for features, cleanups or other kind of improvements ("next-like" content). This is analogous to net and net-next trees for networking. Both bpf and bpf-next will only have a master branch in order to simplify against which branch patches should get rebased to.

Accumulated BPF patches in the bpf tree will regularly get pulled into the net kernel tree. Likewise, accumulated BPF patches accepted into the bpf-next tree will make their way into net-next tree. net and net-next are both run by David S. Miller. From there, they will go into the kernel mainline tree run by Linus Torvalds. To read up on the process of net and net-next being merged into the mainline tree, see the netdev-FAQ

Occasionally, to prevent merge conflicts, we might send pull requests to other trees (e.g. tracing) with a small subset of the patches, but net and net-next are always the main trees targeted for integration.

The pull requests will contain a high-level summary of the accumulated patches and can be searched on netdev kernel mailing list through the following subject lines (`yyyy-mm-dd` is the date of the pull request):

```
pull-request: bpf yyyy-mm-dd
pull-request: bpf-next yyyy-mm-dd
```

### Q: How do I indicate which tree (bpf vs. bpf-next) my patch should be applied to?

A: The process is the very same as described in the netdev-FAQ, so please read up on it. The subject line must indicate whether the patch is a fix or rather "next-like" content in order to let the maintainers know whether it is targeted at bpf or bpf-next.

For fixes eventually landing in bpf -> net tree, the subject must look like:

```
git format-patch --subject-prefix='PATCH bpf' start..finish
```

For features/improvements/etc that should eventually land in bpf-next -> net-next, the subject must look like:

```
git format-patch --subject-prefix='PATCH bpf-next' start..finish
```

If unsure whether the patch or patch series should go into bpf or net directly, or bpf-next or net-next directly, it is not a problem either if the subject line says net or net-next as target. It is eventually up to the maintainers to do the delegation of the patches.

If it is clear that patches should go into bpf or bpf-next tree, please make sure to rebase the patches against those trees in order to reduce potential conflicts.

In case the patch or patch series has to be reworked and sent out again in a second or later revision, it is also required to add a version number (v2, v3, ⋯) into the subject prefix:

```
git format-patch --subject-prefix='PATCH net-next v2' start..finish
```

When changes have been requested to the patch series, always send the whole patch series again with the feedback incorporated (never send individual diffs on top of the old series).

#### Q: What does it mean when a patch gets applied to bpf or bpf-next tree?

A: It means that the patch looks good for mainline inclusion from a BPF point of view.

Be aware that this is not a final verdict that the patch will automatically get accepted into net or net-next trees eventually:

On the bpf kernel mailing list reviews can come in at any point in time. If discussions around a patch conclude that they cannot get included as-is, we will either apply a follow-up fix or drop them from the trees entirely. Therefore, we also reserve to rebase the trees when deemed necessary. After all, the purpose of the tree is to:

i) accumulate and stage BPF patches for integration into trees like net and net-next, and

ii) run extensive BPF test suite and workloads on the patches before they make their way any further.

Once the BPF pull request was accepted by David S. Miller, then the patches end up in net or net-next tree, respectively, and make their way from there further into mainline. Again, see the netdev-FAQ for additional information e.g. on how often they are merged to mainline.

### Q: How long do I need to wait for feedback on my BPF patches?

A: We try to keep the latency low. The usual time to feedback will be around 2 or 3 business days. It may vary depending on the complexity of changes and current patch load.

### Q: How often do you send pull requests to major kernel trees like net or net-next?

A: Pull requests will be sent out rather often in order to not accumulate too many patches in bpf or bpf-next.

As a rule of thumb, expect pull requests for each tree regularly at the end of the week. In some cases pull requests could additionally come also in the middle of the week depending on the current patch load or urgency.

### Q: Are patches applied to bpf-next when the merge window is open?

A: For the time when the merge window is open, bpf-next will not be processed. This is roughly analogous to net-next patch processing, so feel free to read up on the netdev-FAQ about further details.

During those two weeks of merge window, we might ask you to resend your patch series once bpf-next is open again. Once Linus released a `v*-rc1` after the merge window, we continue processing of bpf-next.

For non-subscribers to kernel mailing lists, there is also a status page run by David S. Miller on net-next that provides guidance:

> http://vger.kernel.org/~davem/net-next.html

### Q: Verifier changes and test cases

Q: I made a BPF verifier change, do I need to add test cases for BPF kernel self-tests?

A: If the patch has changes to the behavior of the verifier, then yes, it is absolutely necessary to add test cases to the BPF kernel selftests suite. If they are not present and we think they are needed, then we might ask for them before accepting any changes.

In particular, test_verifier.c is tracking a high number of BPF test cases, including a lot of corner cases that LLVM BPF back end may generate out of the restricted C code. Thus, adding test cases is absolutely crucial to make sure future changes do not accidentally affect prior use-cases. Thus, treat those test cases as: verifier behavior that is not tracked in test_verifier.c could potentially be subject to change.

### Q: samples/bpf preference vs selftests?

Q: When should I add code to samples/bpf/ and when to BPF kernel selftests ?

A: In general, we prefer additions to BPF kernel selftests rather than samples/bpf/. The rationale is very simple: kernel selftests are regularly run by various bots to test for kernel regressions.

The more test cases we add to BPF selftests, the better the coverage and the less likely it is that those could accidentally break. It is not that BPF kernel selftests cannot demo how a specific feature can be used.

That said, samples/bpf/ may be a good place for people to get started, so it might be advisable that simple demos of features could go into samples/bpf/, but advanced functional and corner-case testing rather into kernel selftests.

If your sample looks like a test case, then go for BPF kernel selftests instead!

### Q: When should I add code to the bpftool?

A: The main purpose of bpftool (under tools/bpf/bpftool/) is to provide a central user space tool for debugging and introspection of BPF programs and maps that are active in the kernel. If UAPI changes related to BPF enable for dumping additional information of programs or maps, then bpftool should be extended as well to support dumping them.

### Q: When should I add code to iproute2's BPF loader?

A: For UAPI changes related to the XDP or tc layer (e.g. `cls_bpf`), the convention is that those control-path related changes are added to iproute2's BPF loader as well from user space side. This is not only useful to have UAPI changes properly designed to be usable, but also to make those changes available to a wider user base of major downstream distributions.

### Q: Do you accept patches as well for iproute2's BPF loader?

A: Patches for the iproute2's BPF loader have to be sent to:

> netdev@vger.kernel.org

While those patches are not processed by the BPF kernel maintainers, please keep them in Cc as well, so they can be reviewed.

The official git repository for iproute2 is run by Stephen Hemminger and can be found at:

> https://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git/

The patches need to have a subject prefix of '`[PATCH iproute2 master]`' or '`[PATCH iproute2 net-next]`'. '`master`' or '`net-next`' describes the target branch where the patch should be applied to. Meaning, if kernel changes went into the net-next kernel tree, then the related iproute2 changes need to go into the iproute2 net-next branch, otherwise they can be targeted at master branch. The iproute2 net-next

branch will get merged into the master branch after the current iproute2 version from master has been released.

Like BPF, the patches end up in patchwork under the netdev project and are delegated to 'shemminger' for further processing:

http://patchwork.ozlabs.org/project/netdev/list/?delegate=389

### Q: What is the minimum requirement before I submit my BPF patches?

A: When submitting patches, always take the time and properly test your patches prior to submission. Never rush them! If maintainers find that your patches have not been properly tested, it is a good way to get them grumpy. Testing patch submissions is a hard requirement!

Note, fixes that go to bpf tree must have a `Fixes:` tag included. The same applies to fixes that target bpf-next, where the affected commit is in net-next (or in some cases bpf-next). The `Fixes:` tag is crucial in order to identify follow-up commits and tremendously helps for people having to do backporting, so it is a must have!

We also don't accept patches with an empty commit message. Take your time and properly write up a high quality commit message, it is essential!

Think about it this way: other developers looking at your code a month from now need to understand why a certain change has been done that way, and whether there have been flaws in the analysis or assumptions that the original author did. Thus providing a proper rationale and describing the use-case for the changes is a must.

Patch submissions with >1 patch must have a cover letter which includes a high level description of the series. This high level summary will then be placed into the merge commit by the BPF maintainers such that it is also accessible from the git log for future reference.

### Q: Features changing BPF JIT and/or LLVM

Q: What do I need to consider when adding a new instruction or feature that would require BPF JIT and/or LLVM integration as well?

A: We try hard to keep all BPF JITs up to date such that the same user experience can be guaranteed when running BPF programs on different architectures without having the program punt to the less efficient interpreter in case the in-kernel BPF JIT is enabled.

If you are unable to implement or test the required JIT changes for certain architectures, please work together with the related BPF JIT developers in order to get the feature implemented in a timely manner. Please refer to the git log (`arch/*/net/`) to locate the necessary people for helping out.

Also always make sure to add BPF test cases (e.g. test_bpf.c and test_verifier.c) for new instructions, so that they can receive broad test coverage and help run-time testing the various BPF JITs.

In case of new BPF instructions, once the changes have been accepted into the Linux kernel, please implement support into LLVM's BPF back end. See LLVM section below for further information.

### 2.2.3 Stable submission

#### Q: I need a specific BPF commit in stable kernels. What should I do?

A: In case you need a specific fix in stable kernels, first check whether the commit has already been applied in the related `linux-*.y` branches:

https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/

If not the case, then drop an email to the BPF maintainers with the netdev kernel mailing list in Cc and ask for the fix to be queued up:

netdev@vger.kernel.org

The process in general is the same as on netdev itself, see also the netdev-FAQ.

#### Q: Do you also backport to kernels not currently maintained as stable?

A: No. If you need a specific BPF commit in kernels that are currently not maintained by the stable maintainers, then you are on your own.

The current stable and longterm stable kernels are all listed here:

https://www.kernel.org/

#### Q: The BPF patch I am about to submit needs to go to stable as well

What should I do?

A: The same rules apply as with netdev patch submissions in general, see the netdev-FAQ.

Never add "`Cc: stable@vger.kernel.org`" to the patch description, but ask the BPF maintainers to queue the patches instead. This can be done with a note, for example, under the `---` part of the patch which does not go into the git log. Alternatively, this can be done as a simple request by mail instead.

#### Q: Queue stable patches

Q: Where do I find currently queued BPF patches that will be submitted to stable?

A: Once patches that fix critical bugs got applied into the bpf tree, they are queued up for stable submission under:

http://patchwork.ozlabs.org/bundle/bpf/stable/?state=*

They will be on hold there at minimum until the related commit made its way into the mainline kernel tree.

After having been under broader exposure, the queued patches will be submitted by the BPF maintainers to the stable maintainers.

### 2.2.4 Testing patches

#### Q: How to run BPF selftests

A: After you have booted into the newly compiled kernel, navigate to the BPF selftests suite in order to test BPF functionality (current working directory points to the root of the cloned git tree):

```
$ cd tools/testing/selftests/bpf/
$ make
```

To run the verifier tests:

```
$ sudo ./test_verifier
```

The verifier tests print out all the current checks being performed. The summary at the end of running all tests will dump information of test successes and failures:

```
Summary: 418 PASSED, 0 FAILED
```

In order to run through all BPF selftests, the following command is needed:

```
$ sudo make run_tests
```

See the kernels selftest Documentation/dev-tools/kselftest.rst document for further documentation.

To maximize the number of tests passing, the .config of the kernel under test should match the config file fragment in tools/testing/selftests/bpf as closely as possible.

Finally to ensure support for latest BPF Type Format features - discussed in **`Documentation/bpf/btf.rst`_** - pahole version 1.16 is required for kernels built with CONFIG_DEBUG_INFO_BTF=y. pahole is delivered in the dwarves package or can be built from source at

https://github.com/acmel/dwarves

Some distros have pahole version 1.16 packaged already, e.g. Fedora, Gentoo.

#### Q: Which BPF kernel selftests version should I run my kernel against?

A: If you run a kernel `xyz`, then always run the BPF kernel selftests from that kernel `xyz` as well. Do not expect that the BPF selftest from the latest mainline tree will pass all the time.

In particular, test_bpf.c and test_verifier.c have a large number of test cases and are constantly updated with new BPF test sequences, or existing ones are adapted to verifier changes e.g. due to verifier becoming smarter and being able to better track certain things.

### 2.2.5 LLVM

#### Q: Where do I find LLVM with BPF support?

A: The BPF back end for LLVM is upstream in LLVM since version 3.7.1.

All major distributions these days ship LLVM with BPF back end enabled, so for the majority of use-cases it is not required to compile LLVM by hand anymore, just install the distribution provided package.

LLVM's static compiler lists the supported targets through `llc --version`, make sure BPF targets are listed. Example:

```
$ llc --version
LLVM (http://llvm.org/):
  LLVM version 6.0.0svn
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

  Registered Targets:
    bpf    - BPF (host endian)
    bpfeb  - BPF (big endian)
    bpfel  - BPF (little endian)
    x86    - 32-bit X86: Pentium-Pro and above
    x86-64 - 64-bit X86: EM64T and AMD64
```

For developers in order to utilize the latest features added to LLVM's BPF back end, it is advisable to run the latest LLVM releases. Support for new BPF kernel features such as additions to the BPF instruction set are often developed together.

All LLVM releases can be found at: http://releases.llvm.org/

#### Q: Got it, so how do I build LLVM manually anyway?

A: You need cmake and gcc-c++ as build requisites for LLVM. Once you have that set up, proceed with building the latest LLVM and clang version from the git repositories:

```
$ git clone https://github.com/llvm/llvm-project.git
$ mkdir -p llvm-project/llvm/build/install
$ cd llvm-project/llvm/build
$ cmake .. -G "Ninja" -DLLVM_TARGETS_TO_BUILD="BPF;X86" \
           -DLLVM_ENABLE_PROJECTS="clang"    \
           -DBUILD_SHARED_LIBS=OFF           \
           -DCMAKE_BUILD_TYPE=Release        \
           -DLLVM_BUILD_RUNTIME=OFF
$ ninja
```

The built binaries can then be found in the build/bin/ directory, where you can point the PATH variable to.

### Q: Reporting LLVM BPF issues

Q: Should I notify BPF kernel maintainers about issues in LLVM's BPF code generation back end or about LLVM generated code that the verifier refuses to accept?

A: Yes, please do!

LLVM's BPF back end is a key piece of the whole BPF infrastructure and it ties deeply into verification of programs from the kernel side. Therefore, any issues on either side need to be investigated and fixed whenever necessary.

Therefore, please make sure to bring them up at netdev kernel mailing list and Cc BPF maintainers for LLVM and kernel bits:

- Yonghong Song <yhs@fb.com>
- Alexei Starovoitov <ast@kernel.org>
- Daniel Borkmann <daniel@iogearbox.net>

LLVM also has an issue tracker where BPF related bugs can be found:

https://bugs.llvm.org/buglist.cgi?quicksearch=bpf

However, it is better to reach out through mailing lists with having maintainers in Cc.

### Q: New BPF instruction for kernel and LLVM

Q: I have added a new BPF instruction to the kernel, how can I integrate it into LLVM?

A: LLVM has a `-mcpu` selector for the BPF back end in order to allow the selection of BPF instruction set extensions. By default the `generic` processor target is used, which is the base instruction set (v1) of BPF.

LLVM has an option to select `-mcpu=probe` where it will probe the host kernel for supported BPF instruction set extensions and selects the optimal set automatically.

For cross-compilation, a specific version can be select manually as well

```
$ llc -march bpf -mcpu=help
Available CPUs for this target:

  generic - Select the generic processor.
  probe   - Select the probe processor.
  v1      - Select the v1 processor.
  v2      - Select the v2 processor.
[...]
```

Newly added BPF instructions to the Linux kernel need to follow the same scheme, bump the instruction set version and implement probing for the extensions such that `-mcpu=probe` users can benefit from the optimization transparently when upgrading their kernels.

If you are unable to implement support for the newly added BPF instruction please reach out to BPF developers for help.

By the way, the BPF kernel selftests run with `-mcpu=probe` for better test coverage.

### Q: clang flag for target bpf?

Q: In some cases clang flag `-target bpf` is used but in other cases the default clang target, which matches the underlying architecture, is used. What is the difference and when I should use which?

A: Although LLVM IR generation and optimization try to stay architecture independent, `-target <arch>` still has some impact on generated code:

- BPF program may recursively include header file(s) with file scope inline assembly codes. The default target can handle this well, while `bpf` target may fail if bpf backend assembler does not understand these assembly codes, which is true in most cases.
- When compiled without `-g`, additional elf sections, e.g., .eh_frame and .rela.eh_frame, may be present in the object file with default target, but not with `bpf` target.
- The default target may turn a C switch statement into a switch table lookup and jump operation. Since the switch table is placed in the global readonly section, the bpf program will fail to load. The bpf target does not support switch table optimization. The clang option `-fno-jump-tables` can be used to disable switch table generation.
- For clang `-target bpf`, it is guaranteed that pointer or long / unsigned long types will always have a width of 64 bit, no matter whether underlying clang binary or default target (or kernel) is 32 bit. However, when native clang target is used, then it will compile these types based on the underlying architecture's conventions, meaning in case of 32 bit architecture, pointer or long / unsigned long types e.g. in BPF context structure will have width of 32 bit while the BPF LLVM back end still operates in 64 bit. The native target is mostly needed in tracing for the case of walking `pt_regs` or other kernel structures where CPU's register width matters. Otherwise, `clang -target bpf` is generally recommended.

You should use default target when:

- Your program includes a header file, e.g., ptrace.h, which eventually pulls in some header files containing file scope host assembly codes.
- You can add `-fno-jump-tables` to work around the switch table issue.

Otherwise, you can use `bpf` target. Additionally, you must use bpf target when:

- Your program uses data structures with pointer or long / unsigned long types that interface with BPF helpers or context data structures. Access into these structures is verified by the BPF verifier and may result in verification failures if the native architecture is not aligned with the BPF architecture, e.g. 64-bit. An example of this is BPF_PROG_TYPE_SK_MSG require `-target bpf`

Happy BPF hacking!

CHAPTER

THREE

# PROGRAM TYPES

## 3.1 BPF_PROG_TYPE_CGROUP_SOCKOPT

`BPF_PROG_TYPE_CGROUP_SOCKOPT` program type can be attached to two cgroup hooks:

- `BPF_CGROUP_GETSOCKOPT` - called every time process executes `getsockopt` system call.
- `BPF_CGROUP_SETSOCKOPT` - called every time process executes `setsockopt` system call.

The context (`struct bpf_sockopt`) has associated socket (`sk`) and all input arguments: `level`, `optname`, `optval` and `optlen`.

### 3.1.1 BPF_CGROUP_SETSOCKOPT

`BPF_CGROUP_SETSOCKOPT` is triggered before the kernel handling of sockopt and it has writable context: it can modify the supplied arguments before passing them down to the kernel. This hook has access to the cgroup and socket local storage.

If BPF program sets `optlen` to -1, the control will be returned back to the userspace after all other BPF programs in the cgroup chain finish (i.e. kernel `setsockopt` handling will not be executed).

Note, that `optlen` can not be increased beyond the user-supplied value. It can only be decreased or set to -1. Any other value will trigger `EFAULT`.

#### Return Type

- `0` - reject the syscall, `EPERM` will be returned to the userspace.
- `1` - success, continue with next BPF program in the cgroup chain.

### 3.1.2 BPF_CGROUP_GETSOCKOPT

BPF_CGROUP_GETSOCKOPT is triggered after the kernel handing of sockopt. The BPF hook can observe optval, optlen and retval if it's interested in whatever kernel has returned. BPF hook can override the values above, adjust optlen and reset retval to 0. If optlen has been increased above initial getsockopt value (i.e. userspace buffer is too small), EFAULT is returned.

This hook has access to the cgroup and socket local storage.

Note, that the only acceptable value to set to retval is 0 and the original value that the kernel returned. Any other value will trigger EFAULT.

#### Return Type

- 0 - reject the syscall, EPERM will be returned to the userspace.
- 1 - success: copy optval and optlen to userspace, return retval from the syscall (note that this can be overwritten by the BPF program from the parent cgroup).

### 3.1.3 Cgroup Inheritance

Suppose, there is the following cgroup hierarchy where each cgroup has BPF_CGROUP_GETSOCKOPT attached at each level with BPF_F_ALLOW_MULTI flag:

```
A (root, parent)
 \
  B (child)
```

When the application calls getsockopt syscall from the cgroup B, the programs are executed from the bottom up: B, A. First program (B) sees the result of kernel's getsockopt. It can optionally adjust optval, optlen and reset retval to 0. After that control will be passed to the second (A) program which will see the same context as B including any potential modifications.

Same for BPF_CGROUP_SETSOCKOPT: if the program is attached to A and B, the trigger order is B, then A. If B does any changes to the input arguments (level, optname, optval, optlen), then the next program in the chain (A) will see those changes, not the original input setsockopt arguments. The potentially modified values will be then passed down to the kernel.

### 3.1.4 Large optval

When the optval is greater than the PAGE_SIZE, the BPF program can access only the first PAGE_SIZE of that data. So it has to options:

- Set optlen to zero, which indicates that the kernel should use the original buffer from the userspace. Any modifications done by the BPF program to the optval are ignored.
- Set optlen to the value less than PAGE_SIZE, which indicates that the kernel should use BPF's trimmed optval.

When the BPF program returns with the `optlen` greater than `PAGE_SIZE`, the userspace will receive `EFAULT` errno.

### 3.1.5 Example

See `tools/testing/selftests/bpf/progs/sockopt_sk.c` for an example of BPF program that handles socket options.

## 3.2 BPF_PROG_TYPE_CGROUP_SYSCTL

This document describes `BPF_PROG_TYPE_CGROUP_SYSCTL` program type that provides cgroup-bpf hook for sysctl.

The hook has to be attached to a cgroup and will be called every time a process inside that cgroup tries to read from or write to sysctl knob in proc.

### 3.2.1 1. Attach type

`BPF_CGROUP_SYSCTL` attach type has to be used to attach `BPF_PROG_TYPE_CGROUP_SYSCTL` program to a cgroup.

### 3.2.2 2. Context

`BPF_PROG_TYPE_CGROUP_SYSCTL` provides access to the following context from BPF program:

```
struct bpf_sysctl {
    __u32 write;
    __u32 file_pos;
};
```

- `write` indicates whether sysctl value is being read (`0`) or written (`1`). This field is read-only.
- `file_pos` indicates file position sysctl is being accessed at, read or written. This field is read-write. Writing to the field sets the starting position in sysctl proc file `read(2)` will be reading from or `write(2)` will be writing to. Writing zero to the field can be used e.g. to override whole sysctl value by `bpf_sysctl_set_new_value()` on `write(2)` even when it' s called by user space on `file_pos > 0`. Writing non-zero value to the field can be used to access part of sysctl value starting from specified `file_pos`. Not all sysctl support access with `file_pos != 0`, e.g. writes to numeric sysctl entries must always be at file position `0`. See also `kernel.sysctl_writes_strict` sysctl.

See linux/bpf.h for more details on how context field can be accessed.

### 3.2.3 3. Return code

`BPF_PROG_TYPE_CGROUP_SYSCTL` program must return one of the following return codes:

- `0` means “reject access to sysctl”;
- `1` means “proceed with access”.

If program returns `0` user space will get `-1` from `read(2)` or `write(2)` and `errno` will be set to `EPERM`.

### 3.2.4 4. Helpers

Since sysctl knob is represented by a name and a value, sysctl specific BPF helpers focus on providing access to these properties:

- `bpf_sysctl_get_name()` to get sysctl name as it is visible in `/proc/sys` into provided by BPF program buffer;
- `bpf_sysctl_get_current_value()` to get string value currently held by sysctl into provided by BPF program buffer. This helper is available on both `read(2)` from and `write(2)` to sysctl;
- `bpf_sysctl_get_new_value()` to get new string value currently being written to sysctl before actual write happens. This helper can be used only on `ctx->write == 1`;
- `bpf_sysctl_set_new_value()` to override new string value currently being written to sysctl before actual write happens. Sysctl value will be overridden starting from the current `ctx->file_pos`. If the whole value has to be overridden BPF program can set `file_pos` to zero before calling to the helper. This helper can be used only on `ctx->write == 1`. New string value set by the helper is treated and verified by kernel same way as an equivalent string passed by user space.

BPF program sees sysctl value same way as user space does in proc filesystem, i.e. as a string. Since many sysctl values represent an integer or a vector of integers, the following helpers can be used to get numeric value from the string:

- `bpf_strtol()` to convert initial part of the string to long integer similar to user space strtol(3);
- `bpf_strtoul()` to convert initial part of the string to unsigned long integer similar to user space strtoul(3);

See linux/bpf.h for more details on helpers described here.

### 3.2.5 5. Examples

See test_sysctl_prog.c for an example of BPF program in C that access sysctl name and value, parses string value to get vector of integers and uses the result to make decision whether to allow or deny access to sysctl.

### 3.2.6 6. Notes

`BPF_PROG_TYPE_CGROUP_SYSCTL` is intended to be used in **trusted** root environment, for example to monitor sysctl usage or catch unreasonable values an application, running as root in a separate cgroup, is trying to set.

Since task_dfl_cgroup(current) is called at sys_read / sys_write time it may return results different from that at sys_open time, i.e. process that opened sysctl file in proc filesystem may differ from process that is trying to read from / write to it and two such processes may run in different cgroups, what means `BPF_PROG_TYPE_CGROUP_SYSCTL` should not be used as a security mechanism to limit sysctl usage.

As with any cgroup-bpf program additional care should be taken if an application running as root in a cgroup should not be allowed to detach/replace BPF program attached by administrator.

## 3.3 BPF_PROG_TYPE_FLOW_DISSECTOR

### 3.3.1 Overview

Flow dissector is a routine that parses metadata out of the packets. It's used in the various places in the networking subsystem (RFS, flow hash, etc).

BPF flow dissector is an attempt to reimplement C-based flow dissector logic in BPF to gain all the benefits of BPF verifier (namely, limits on the number of instructions and tail calls).

### 3.3.2 API

BPF flow dissector programs operate on an `__sk_buff`. However, only the limited set of fields is allowed: `data`, `data_end` and `flow_keys`. `flow_keys` is `struct bpf_flow_keys` and contains flow dissector input and output arguments.

**The inputs are:**

- `nhoff` - initial offset of the networking header
- `thoff` - initial offset of the transport header, initialized to nhoff
- `n_proto` - L3 protocol type, parsed out of L2 header
- `flags` - optional flags

Flow dissector BPF program should fill out the rest of the `struct bpf_flow_keys` fields. Input arguments `nhoff/thoff/n_proto` should be also adjusted accordingly.

The return code of the BPF program is either BPF_OK to indicate successful dissection, or BPF_DROP to indicate parsing error.

### 3.3.3 __sk_buff->data

In the VLAN-less case, this is what the initial state of the BPF flow dissector looks like:

```
+------+------+------------+-----------+
| DMAC | SMAC | ETHER_TYPE | L3_HEADER |
+------+------+------------+-----------+
                           ^
                           |
                           +-- flow dissector starts here
```

```
skb->data + flow_keys->nhoff point to the first byte of L3_HEADER
flow_keys->thoff = nhoff
flow_keys->n_proto = ETHER_TYPE
```

In case of VLAN, flow dissector can be called with the two different states.

Pre-VLAN parsing:

```
+------+------+------+-----+-----------+-----------+
| DMAC | SMAC | TPID | TCI |ETHER_TYPE | L3_HEADER |
+------+------+------+-----+-----------+-----------+
                     ^
                     |
                     +-- flow dissector starts here
```

```
skb->data + flow_keys->nhoff point the to first byte of TCI
flow_keys->thoff = nhoff
flow_keys->n_proto = TPID
```

Please note that TPID can be 802.1AD and, hence, BPF program would have to parse VLAN information twice for double tagged packets.

Post-VLAN parsing:

```
+------+------+------+-----+-----------+-----------+
| DMAC | SMAC | TPID | TCI |ETHER_TYPE | L3_HEADER |
+------+------+------+-----+-----------+-----------+
                                        ^
                                        |
                                        +-- flow dissector starts here
```

```
skb->data + flow_keys->nhoff point the to first byte of L3_HEADER
flow_keys->thoff = nhoff
flow_keys->n_proto = ETHER_TYPE
```

In this case VLAN information has been processed before the flow dissector and BPF flow dissector is not required to handle it.

The takeaway here is as follows: BPF flow dissector program can be called with the optional VLAN header and should gracefully handle both cases: when single

or double VLAN is present and when it is not present. The same program can be called for both cases and would have to be written carefully to handle both cases.

### 3.3.4 Flags

`flow_keys->flags` might contain optional input flags that work as follows:

- `BPF_FLOW_DISSECTOR_F_PARSE_1ST_FRAG` - tells BPF flow dissector to continue parsing first fragment; the default expected behavior is that flow dissector returns as soon as it finds out that the packet is fragmented; used by `eth_get_headlen` to estimate length of all headers for GRO.
- `BPF_FLOW_DISSECTOR_F_STOP_AT_FLOW_LABEL` - tells BPF flow dissector to stop parsing as soon as it reaches IPv6 flow label; used by `___skb_get_hash` and `__skb_get_hash_symmetric` to get flow hash.
- `BPF_FLOW_DISSECTOR_F_STOP_AT_ENCAP` - tells BPF flow dissector to stop parsing as soon as it reaches encapsulated headers; used by routing infrastructure.

### 3.3.5 Reference Implementation

See `tools/testing/selftests/bpf/progs/bpf_flow.c` for the reference implementation and `tools/testing/selftests/bpf/flow_dissector_load.[hc]` for the loader. bpftool can be used to load BPF flow dissector program as well.

**The reference implementation is organized as follows:**

- `jmp_table` map that contains sub-programs for each supported L3 protocol
- `_dissect` routine - entry point; it does input `n_proto` parsing and does `bpf_tail_call` to the appropriate L3 handler

Since BPF at this point doesn't support looping (or any jumping back), jmp_table is used instead to handle multiple levels of encapsulation (and IPv6 options).

### 3.3.6 Current Limitations

BPF flow dissector doesn't support exporting all the metadata that in-kernel C-based implementation can export. Notable example is single VLAN (802.1Q) and double VLAN (802.1AD) tags. Please refer to the `struct bpf_flow_keys` for a set of information that's currently can be exported from the BPF context.

When BPF flow dissector is attached to the root network namespace (machine-wide policy), users can't override it in their child network namespaces.

## 3.4 LSM BPF Programs

These BPF programs allow runtime instrumentation of the LSM hooks by privileged users to implement system-wide MAC (Mandatory Access Control) and Audit policies using eBPF.

### 3.4.1 Structure

The example shows an eBPF program that can be attached to the `file_mprotect` LSM hook:

**`int file_mprotect(struct vm_area_struct *vma, unsigned long reqprot, unsigned lo`**

Other LSM hooks which can be instrumented can be found in `include/linux/lsm_hooks.h`.

eBPF programs that use BPF Type Format (BTF) do not need to include kernel headers for accessing information from the attached eBPF program' s context. They can simply declare the structures in the eBPF program and only specify the fields that need to be accessed.

```
struct mm_struct {
        unsigned long start_brk, brk, start_stack;
} __attribute__((preserve_access_index));

struct vm_area_struct {
        unsigned long start_brk, brk, start_stack;
        unsigned long vm_start, vm_end;
        struct mm_struct *vm_mm;
} __attribute__((preserve_access_index));
```

---

**Note:** The order of the fields is irrelevant.

---

This can be further simplified (if one has access to the BTF information at build time) by generating the `vmlinux.h` with:

```
# bpftool btf dump file <path-to-btf-vmlinux> format c > vmlinux.h
```

---

**Note:** `path-to-btf-vmlinux` can be `/sys/kernel/btf/vmlinux` if the build environment matches the environment the BPF programs are deployed in.

---

The `vmlinux.h` can then simply be included in the BPF programs without requiring the definition of the types.

The eBPF programs can be declared using the``BPF_PROG`` macros defined in tools/lib/bpf/bpf_tracing.h. In this example:

- `"lsm/file_mprotect"` indicates the LSM hook that the program must be attached to
- `mprotect_audit` is the name of the eBPF program

```
SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_audit, struct vm_area_struct *vma,
             unsigned long reqprot, unsigned long prot, int ret)
{
        /* ret is the return value from the previous BPF program
         * or 0 if it's the first hook.
         */
        if (ret != 0)
                return ret;

        int is_heap;

        is_heap = (vma->vm_start >= vma->vm_mm->start_brk &&
                   vma->vm_end <= vma->vm_mm->brk);

        /* Return an -EPERM or write information to the perf events buffer
         * for auditing
         */
        if (is_heap)
                return -EPERM;
}
```

The `__attribute__((preserve_access_index))` is a clang feature that allows the BPF verifier to update the offsets for the access at runtime using the BPF Type Format (BTF) information. Since the BPF verifier is aware of the types, it also validates all the accesses made to the various types in the eBPF program.

### 3.4.2 Loading

eBPF programs can be loaded with the `bpf(2)` syscall's `BPF_PROG_LOAD` operation:

```
struct bpf_object *obj;

obj = bpf_object__open("./my_prog.o");
bpf_object__load(obj);
```

This can be simplified by using a skeleton header generated by `bpftool`:

```
# bpftool gen skeleton my_prog.o > my_prog.skel.h
```

and the program can be loaded by including `my_prog.skel.h` and using the generated helper, `my_prog__open_and_load`.

### 3.4.3 Attachment to LSM Hooks

The LSM allows attachment of eBPF programs as LSM hooks using `bpf(2)` syscall's `BPF_RAW_TRACEPOINT_OPEN` operation or more simply by using the libbpf helper `bpf_program__attach_lsm`.

The program can be detached from the LSM hook by destroying the `link` link returned by `bpf_program__attach_lsm` using `bpf_link__destroy`.

One can also use the helpers generated in `my_prog.skel.h` i.e. `my_prog__attach` for attachment and `my_prog__destroy` for cleaning up.

### 3.4.4 Examples

An example eBPF program can be found in tools/testing/selftests/bpf/progs/lsm.c and the corresponding userspace code in tools/testing/selftests/bpf/prog_tests/test_lsm.c

CHAPTER

FOUR

# TESTING AND DEBUGGING BPF

## 4.1 BPF drgn tools

drgn scripts is a convenient and easy to use mechanism to retrieve arbitrary kernel data structures. drgn is not relying on kernel UAPI to read the data. Instead it's reading directly from `/proc/kcore` or vmcore and pretty prints the data based on DWARF debug information from vmlinux.

This document describes BPF related drgn tools.

See drgn/tools for all tools available at the moment and drgn/doc for more details on drgn itself.

### 4.1.1 bpf_inspect.py

#### Description

bpf_inspect.py is a tool intended to inspect BPF programs and maps. It can iterate over all programs and maps in the system and print basic information about these objects, including id, type and name.

The main use-case bpf_inspect.py covers is to show BPF programs of types `BPF_PROG_TYPE_EXT` and `BPF_PROG_TYPE_TRACING` attached to other BPF programs via `freplace/fentry/fexit` mechanisms, since there is no user-space API to get this information.

#### Getting started

List BPF programs (full names are obtained from BTF):

```
% sudo bpf_inspect.py prog
    27: BPF_PROG_TYPE_TRACEPOINT         tracepoint__tcp__tcp_send_reset
  4632: BPF_PROG_TYPE_CGROUP_SOCK_ADDR   tw_ipt_bind
 49464: BPF_PROG_TYPE_RAW_TRACEPOINT     raw_tracepoint__sched_process_exit
```

List BPF maps:

```
% sudo bpf_inspect.py map
  2577: BPF_MAP_TYPE_HASH                tw_ipt_vips
  4050: BPF_MAP_TYPE_STACK_TRACE         stack_traces
  4069: BPF_MAP_TYPE_PERCPU_ARRAY        ned_dctcp_cntr
```

Find BPF programs attached to BPF program `test_pkt_access`:

```
% sudo bpf_inspect.py p | grep test_pkt_access
   650: BPF_PROG_TYPE_SCHED_CLS          test_pkt_access
   654: BPF_PROG_TYPE_TRACING            test_main                                ␣
↪linked:[650->25: BPF_TRAMP_FEXIT test_pkt_access->test_pkt_access()]
   655: BPF_PROG_TYPE_TRACING            test_subprog1                            ␣
↪linked:[650->29: BPF_TRAMP_FEXIT test_pkt_access->test_pkt_access_
↪subprog1()]
   656: BPF_PROG_TYPE_TRACING            test_subprog2                            ␣
↪linked:[650->31: BPF_TRAMP_FEXIT test_pkt_access->test_pkt_access_
↪subprog2()]
   657: BPF_PROG_TYPE_TRACING            test_subprog3                            ␣
↪linked:[650->21: BPF_TRAMP_FEXIT test_pkt_access->test_pkt_access_
↪subprog3()]
   658: BPF_PROG_TYPE_EXT                new_get_skb_len                          ␣
↪linked:[650->16: BPF_TRAMP_REPLACE test_pkt_access->get_skb_len()]
   659: BPF_PROG_TYPE_EXT                new_get_skb_ifindex                      ␣
↪linked:[650->23: BPF_TRAMP_REPLACE test_pkt_access->get_skb_ifindex()]
   660: BPF_PROG_TYPE_EXT                new_get_constant                         ␣
↪linked:[650->19: BPF_TRAMP_REPLACE test_pkt_access->get_constant()]
```

It can be seen that there is a program `test_pkt_access`, id 650 and there are multiple other tracing and ext programs attached to functions in `test_pkt_access`.

For example the line:

```
658: BPF_PROG_TYPE_EXT                new_get_skb_len                          ␣
↪linked:[650->16: BPF_TRAMP_REPLACE test_pkt_access->get_skb_len()]
```

, means that BPF program id 658, type `BPF_PROG_TYPE_EXT`, name `new_get_skb_len` replaces (`BPF_TRAMP_REPLACE`) function `get_skb_len()` that has BTF id 16 in BPF program id 650, name `test_pkt_access`.

Getting help:

```
% sudo bpf_inspect.py
usage: bpf_inspect.py [-h] {prog,p,map,m} ...

drgn script to list BPF programs or maps and their properties
unavailable via kernel API.

See https://github.com/osandov/drgn/ for more details on drgn.

optional arguments:
  -h, --help      show this help message and exit

subcommands:
  {prog,p,map,m}
    prog (p)      list BPF programs
    map (m)       list BPF maps
```

### Customization

The script is intended to be customized by developers to print relevant information about BPF programs, maps and other objects.

For example, to print `struct bpf_prog_aux` for BPF program id 53077:

```
% git diff
diff --git a/tools/bpf_inspect.py b/tools/bpf_inspect.py
index 650e228..aea2357 100755
--- a/tools/bpf_inspect.py
+++ b/tools/bpf_inspect.py
@@ -112,7 +112,9 @@ def list_bpf_progs(args):
          if linked:
              linked = f" linked:[{linked}]"

-         print(f"{id_:>6}: {type_:32} {name:32} {linked}")
+         if id_ == 53077:
+             print(f"{id_:>6}: {type_:32} {name:32}")
+             print(f"{bpf_prog.aux}")


 def list_bpf_maps(args):
```

It produces the output:

```
% sudo bpf_inspect.py p
 53077: BPF_PROG_TYPE_XDP                tw_xdp_policer
*(struct bpf_prog_aux *)0xffff8893fad4b400 = {
        .refcnt = (atomic64_t){
                .counter = (long)58,
        },
        .used_map_cnt = (u32)1,
        .max_ctx_offset = (u32)8,
        .max_pkt_offset = (u32)15,
        .max_tp_access = (u32)0,
        .stack_depth = (u32)8,
        .id = (u32)53077,
        .func_cnt = (u32)0,
        .func_idx = (u32)0,
        .attach_btf_id = (u32)0,
        .linked_prog = (struct bpf_prog *)0x0,
        .verifier_zext = (bool)0,
        .offload_requested = (bool)0,
        .attach_btf_trace = (bool)0,
        .func_proto_unreliable = (bool)0,
        .trampoline_prog_type = (enum bpf_tramp_prog_type)BPF_TRAMP_FENTRY,
        .trampoline = (struct bpf_trampoline *)0x0,
        .tramp_hlist = (struct hlist_node){
                .next = (struct hlist_node *)0x0,
                .pprev = (struct hlist_node **)0x0,
        },
        .attach_func_proto = (const struct btf_type *)0x0,
        .attach_func_name = (const char *)0x0,
        .func = (struct bpf_prog **)0x0,
        .jit_data = (void *)0x0,
        .poke_tab = (struct bpf_jit_poke_descriptor *)0x0,
```

(continues on next page)

(continued from previous page)

```
        .size_poke_tab = (u32)0,
        .ksym_tnode = (struct latch_tree_node){
                .node = (struct rb_node [2]){
                        {
                                .__rb_parent_color = (unsigned␣
↪long)18446612956263126665,
                                .rb_right = (struct rb_node *)0x0,
                                .rb_left = (struct rb_node␣
↪*)0xffff88a0be3d0088,
                        },
                        {
                                .__rb_parent_color = (unsigned␣
↪long)18446612956263126689,
                                .rb_right = (struct rb_node *)0x0,
                                .rb_left = (struct rb_node␣
↪*)0xffff88a0be3d00a0,
                        },
                },
        },
        .ksym_lnode = (struct list_head){
                .next = (struct list_head *)0xffff88bf481830b8,
                .prev = (struct list_head *)0xffff888309f536b8,
        },
        .ops = (const struct bpf_prog_ops *)xdp_prog_ops+0x0 =␣
↪0xffffffff820fa350,
        .used_maps = (struct bpf_map **)0xffff889ff795de98,
        .prog = (struct bpf_prog *)0xffffc9000cf2d000,
        .user = (struct user_struct *)root_user+0x0 = 0xffffffff82444820,
        .load_time = (u64)2408348759285319,
        .cgroup_storage = (struct bpf_map *[2]){},
        .name = (char [16])"tw_xdp_policer",
        .security = (void *)0xffff889ff795d548,
        .offload = (struct bpf_prog_offload *)0x0,
        .btf = (struct btf *)0xffff8890ce6d0580,
        .func_info = (struct bpf_func_info *)0xffff889ff795d240,
        .func_info_aux = (struct bpf_func_info_aux *)0xffff889ff795de20,
        .linfo = (struct bpf_line_info *)0xffff888a707afc00,
        .jited_linfo = (void **)0xffff8893fad48600,
        .func_info_cnt = (u32)1,
        .nr_linfo = (u32)37,
        .linfo_idx = (u32)0,
        .num_exentries = (u32)0,
        .extable = (struct exception_table_entry *)0xffffffffa032d950,
        .stats = (struct bpf_prog_stats *)0x603fe3a1f6d0,
        .work = (struct work_struct){
                .data = (atomic_long_t){
                        .counter = (long)0,
                },
                .entry = (struct list_head){
                        .next = (struct list_head *)0x0,
                        .prev = (struct list_head *)0x0,
                },
                .func = (work_func_t)0x0,
        },
        .rcu = (struct callback_head){
                .next = (struct callback_head *)0x0,
```

(continues on next page)

(continued from previous page)

```
                .func = (void (*)(struct callback_head *))0x0,
        },
}
```

## 4.2 Testing BPF on s390

### 4.2.1 1. Introduction

IBM Z are mainframe computers, which are descendants of IBM System/360 from year 1964. They are supported by the Linux kernel under the name "s390". This document describes how to test BPF in an s390 QEMU guest.

### 4.2.2 2. One-time setup

The following is required to build and run the test suite:

- s390 GCC
- s390 development headers and libraries
- Clang with BPF support
- QEMU with s390 support
- Disk image with s390 rootfs

Debian supports installing compiler and libraries for s390 out of the box. Users of other distros may use debootstrap in order to set up a Debian chroot:

```
sudo debootstrap \
  --variant=minbase \
  --include=sudo \
  testing \
  ./s390-toolchain
sudo mount --rbind /dev ./s390-toolchain/dev
sudo mount --rbind /proc ./s390-toolchain/proc
sudo mount --rbind /sys ./s390-toolchain/sys
sudo chroot ./s390-toolchain
```

Once on Debian, the build prerequisites can be installed as follows:

```
sudo dpkg --add-architecture s390x
sudo apt-get update
sudo apt-get install \
  bc \
  bison \
  cmake \
  debootstrap \
  dwarves \
  flex \
  g++ \
  gcc \
  g++-s390x-linux-gnu \
```

(continues on next page)

(continued from previous page)

```
  gcc-s390x-linux-gnu \
  gdb-multiarch \
  git \
  make \
  python3 \
  qemu-system-misc \
  qemu-utils \
  rsync \
  libcap-dev:s390x \
  libelf-dev:s390x \
  libncurses-dev
```

Latest Clang targeting BPF can be installed as follows:

```
git clone https://github.com/llvm/llvm-project.git
ln -s ../../clang llvm-project/llvm/tools/
mkdir llvm-project-build
cd llvm-project-build
cmake \
  -DLLVM_TARGETS_TO_BUILD=BPF \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=/opt/clang-bpf \
  ../llvm-project/llvm
make
sudo make install
export PATH=/opt/clang-bpf/bin:$PATH
```

The disk image can be prepared using a loopback mount and debootstrap:

```
qemu-img create -f raw ./s390.img 1G
sudo losetup -f ./s390.img
sudo mkfs.ext4 /dev/loopX
mkdir ./s390.rootfs
sudo mount /dev/loopX ./s390.rootfs
sudo debootstrap \
  --foreign \
  --arch=s390x \
  --variant=minbase \
  --include=" \
    iproute2, \
    iputils-ping, \
    isc-dhcp-client, \
    kmod, \
    libcap2, \
    libelf1, \
    netcat, \
    procps" \
  testing \
  ./s390.rootfs
sudo umount ./s390.rootfs
sudo losetup -d /dev/loopX
```

#### 4.2.3 3. Compilation

In addition to the usual Kconfig options required to run the BPF test suite, it is also helpful to select:

```
CONFIG_NET_9P=y
CONFIG_9P_FS=y
CONFIG_NET_9P_VIRTIO=y
CONFIG_VIRTIO_PCI=y
```

as that would enable a very easy way to share files with the s390 virtual machine.

Compiling kernel, modules and testsuite, as well as preparing gdb scripts to simplify debugging, can be done using the following commands:

```
make ARCH=s390 CROSS_COMPILE=s390x-linux-gnu- menuconfig
make ARCH=s390 CROSS_COMPILE=s390x-linux-gnu- bzImage modules scripts_gdb
make ARCH=s390 CROSS_COMPILE=s390x-linux-gnu- \
  -C tools/testing/selftests \
  TARGETS=bpf \
  INSTALL_PATH=$PWD/tools/testing/selftests/kselftest_install \
  install
```

#### 4.2.4 4. Running the test suite

The virtual machine can be started as follows:

```
qemu-system-s390x \
  -cpu max,zpci=on \
  -smp 2 \
  -m 4G \
  -kernel linux/arch/s390/boot/compressed/vmlinux \
  -drive file=./s390.img,if=virtio,format=raw \
  -nographic \
  -append 'root=/dev/vda rw console=ttyS1' \
  -virtfs local,path=./linux,security_model=none,mount_tag=linux \
  -object rng-random,filename=/dev/urandom,id=rng0 \
  -device virtio-rng-ccw,rng=rng0 \
  -netdev user,id=net0 \
  -device virtio-net-ccw,netdev=net0
```

When using this on a real IBM Z, `-enable-kvm` may be added for better performance. When starting the virtual machine for the first time, disk image setup must be finalized using the following command:

```
/debootstrap/debootstrap --second-stage
```

Directory with the code built on the host as well as `/proc` and `/sys` need to be mounted as follows:

```
mkdir -p /linux
mount -t 9p linux /linux
mount -t proc proc /proc
mount -t sysfs sys /sys
```

After that, the test suite can be run using the following commands:

```
cd /linux/tools/testing/selftests/kselftest_install
./run_kselftest.sh
```

As usual, tests can be also run individually:

```
cd /linux/tools/testing/selftests/bpf
./test_verifier
```

### 4.2.5 5. Debugging

It is possible to debug the s390 kernel using QEMU GDB stub, which is activated by passing `-s` to QEMU.

It is preferable to turn KASLR off, so that gdb would know where to find the kernel image in memory, by building the kernel with:

```
RANDOMIZE_BASE=n
```

GDB can then be attached using the following command:

```
gdb-multiarch -ex 'target remote localhost:1234' ./vmlinux
```

### 4.2.6 6. Network

In case one needs to use the network in the virtual machine in order to e.g. install additional packages, it can be configured using:

```
dhclient eth0
```

### 4.2.7 7. Links

This document is a compilation of techniques, whose more comprehensive descriptions can be found by following these links:

- Debootstrap
- Multiarch
- Building LLVM
- Cross-compiling the kernel
- QEMU s390x Guest Support
- Plan 9 folder sharing over Virtio
- Using GDB with QEMU