BPF Type Format (BTF)

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 :ref:`BTF_Type_String`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\bpf\[linux-master][Documentation][bpf]btf.rst, line 26); *backlink***
>
> Unknown interpreted text role "ref".

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       */
    __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      */
```

```
#define BTF_KIND_FLOAT          16      /* Floating point       */
#define BTF_KIND_DECL_TAG       17      /* Decl Tag     */
#define BTF_KIND_TYPE_TAG       18      /* Type Tag     */
```

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 24-28: kind (e.g. int, ptr, array...etc)
     * bits 29-30: unused
     * bit     31: kind_flag, currently used by
     *             struct, union and fwd
     */
    __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, FUNC_PROTO, DECL_TAG and TYPE_TAG.
     * "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 multidimensional 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 :ref:`BTF_Ext_Section` (ELF) or in the arguments to :ref:`BPF_Prog_Load` (ABI).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\bpf\[linux-master][Documentation][bpf]btf.rst, line 372); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\bpf\[linux-master][Documentation][bpf]btf.rst, line 372); *backlink*

Unknown interpreted text role "ref".

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

2.2.16 BTF_KIND_FLOAT

`struct btf_type` encoding requirement:

- `name_off`: any valid offset
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_FLOAT
- `info.vlen`: 0
- `size`: the size of the float type in bytes: 2, 4, 8, 12 or 16.

No additional type data follow `btf_type`.

2.2.17 BTF_KIND_DECL_TAG

`struct btf_type` encoding requirement:

- `name_off`: offset to a non-empty string
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_DECL_TAG
- `info.vlen`: 0
- `type`: `struct`, `union`, `func`, `var` or `typedef`

`btf_type` is followed by `struct btf_decl_tag`.:

```
struct btf_decl_tag {
    __u32   component_idx;
```

```
};
```

The `name_off` encodes btf_decl_tag attribute string. The `type` should be `struct`, `union`, `func`, `var` or `typedef`. For `var` or `typedef` type, `btf_decl_tag.component_idx` must be `-1`. For the other three types, if the btf_decl_tag attribute is applied to the `struct`, `union` or `func` itself, `btf_decl_tag.component_idx` must be `-1`. Otherwise, the attribute is applied to a `struct`/`union` member or a `func` argument, and `btf_decl_tag.component_idx` should be a valid index (starting from 0) pointing to a member or an argument.

2.2.17 BTF_KIND_TYPE_TAG

`struct btf_type` encoding requirement:

- `name_off`: offset to a non-empty string
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_TYPE_TAG
- `info.vlen`: 0
- `type`: the type with `btf_type_tag` attribute

Currently, `BTF_KIND_TYPE_TAG` is only emitted for pointer types. It has the following btf type chain:

```
ptr -> [type_tag]*
    -> [const | volatile | restrict | typedef]*
    -> base_type
```

Basically, a pointer type points to zero or more type_tag, then zero or more const/volatile/restrict/typedef and finally the base type. The base type is one of int, ptr, array, struct, union, enum, func_proto and float types.

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:
    ......
    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 :ref:`BTF_Type_String`, can be directly loaded into the kernel. A `btf_fd` is returned to a userspace.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\bpf\[linux-master][Documentation][bpf]btf.rst, line 565); *backlink*

Unknown interpreted text role "ref".

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 {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, int);
    __type(value, struct ipv_counts);
    __uint(max_entries, 4);
} btf_map SEC(".maps");
```

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.

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 :ref:`BTF_Type_String`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\bpf\[linux-master][Documentation][bpf]btf.rst`, line 686); *backlink***
>
> Unknown interpreted text role "ref".

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 :ref:`BPF_Prog_Load` for details about func_info and line_info record format.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\bpf\[linux-master][Documentation][bpf]btf.rst`, line 715); *backlink***
>
> Unknown interpreted text role "ref".

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`).

4.2 .BTF_ids section

The .BTF_ids section encodes BTF ID values that are used within the kernel.

This section is created during the kernel compilation with the help of macros defined in `include/linux/btf_ids.h` header file. Kernel code can use them to create lists and sets (sorted lists) of BTF ID values.

The `BTF_ID_LIST` and `BTF_ID` macros define unsorted list of BTF ID values, with following syntax:

```
BTF_ID_LIST(list)
BTF_ID(type1, name1)
BTF_ID(type2, name2)
```

resulting in following layout in .BTF_ids section:

```
__BTF_ID__type1__name1__1:
.zero 4
__BTF_ID__type2__name2__2:
.zero 4
```

The `u32 list[];` variable is defined to access the list.

The `BTF_ID_UNUSED` macro defines 4 zero bytes. It's used when we want to define unused entry in BTF_ID_LIST, like:

```
BTF_ID_LIST(bpf_skb_output_btf_ids)
BTF_ID(struct, sk_buff)
BTF_ID_UNUSED
BTF_ID(struct, task_struct)
```

The `BTF_SET_START/END` macros pair defines sorted list of BTF ID values and their count, with following syntax:

```
BTF_SET_START(set)
BTF_ID(type1, name1)
BTF_ID(type2, name2)
BTF_SET_END(set)
```

resulting in following layout in .BTF_ids section:

```
__BTF_ID__set__set:
.zero 4
__BTF_ID__type1__name1__3:
.zero 4
__BTF_ID__type2__name2__4:
.zero 4
```

The `struct btf_id_set set;` variable is defined to access the list.

The `typeX` name can be one of following:

```
struct, union, typedef, func
```

and is used as a filter when resolving the BTF ID value.

All the BTF ID lists and sets are compiled in the .BTF_ids section and resolved during the linking phase of kernel build by `resolve_btfids` tool.

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 {
     __uint(type, BPF_MAP_TYPE_ARRAY);
     __type(key, int);
     __type(value, struct tmp_t);
     __uint(max_entries, 1);
} tmpmap SEC(".maps");
```

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
   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
```

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
        .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
```

7. Testing

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