Формат BTF

Оригинал

Оглавление

1. Введение

BTF (BPF Type Format) - формат метаданных, представляющих отладочные сведения, связанные с программами и отображениями фильтров BPF. Изначально формат BTF служил для описания типов данных, а затем был расширен включением информации о функциях для определённых подпрограмм и сведений о строках в исходном коде.

Отладочная информация служит для вывода на печать, подписи функции и т. п. Подписи функций позволяют лучше представить символы ядра программ и функций bpf. Сведения о строках помогают генерировать транслированный байт-код и JIT[1]-код с аннотациями, а также журнальный файл проверяющего.

Спецификация BTF состоит из двух частей:

- API ядра;
- формат файлов ELF.

API ядра - это соглашение между ядром и пользовательским пространством. Ядро проверяет BTF до применения. Формат ELF является соглашением пользовательского пространства между файлом ELF и загрузчиком libbpf.

Тип и строки являются частью API ядра, описывающей отладочную информацию (в основном, типы), указываемую программой bpf.

2. Кодирование типов и строк BTF

В файле include/uapi/linux/btf.h приведено высокоуровневое определение кодирования типов и строк. Начало блока данных (blob) должно иметь вид

```
struct btf_header {
    __u16   magic;
    __u8    version;
```

[1]Just In Time - «на лету», код, создаваемый в процессе исполнения.

```
    __u8    flags;
    __u32   hdr_len;

    /* Все смещения указаны в байтах от конца этого заголовка */
    __u32   type_off;       /* Смещение раздела типов      */
    __u32   type_len;       /* Размер раздела типов        */
    __u32   str_off;        /* Смещение раздела строк      */
    __u32   str_len;        /* Размер раздела строк        */
};
```

Поле magic имеет значение 0xeB9F, которое по разному представляется в системах с прямым (big-endian, от старшего к младшему) и обратным (little-endian, от младшего к старшему) порядком и может применяться при проверке генерации BTF для разных типов целевых систем. Заголовок btf_header предусматривает возможность расширения с помощью hdr_len = sizeof(struct btf_header) при генерации блобов (blob).

2.1 Кодирование строк

Первой строкой в разделе строк должна быть пустая (null) строка, а остальная часть раздела представляет собой конкатенацию строк с null-символом завершения строки.

2.2 Кодирование типов

Идентификатор типа 0 зарезервирован для типа void. Раздел типов анализируется последовательно и идентификаторы назначаются каждому распознанному типу по порядку, начиная с 1. Поддерживаемые в настоящее время типы указаны ниже.

```
#define BTF_KIND_INT            1       /* Целое число      */
#define BTF_KIND_PTR            2       /* Указатель        */
#define BTF_KIND_ARRAY          3       /* Массив           */
#define BTF_KIND_STRUCT         4       /* Структура        */
#define BTF_KIND_UNION          5       /* Объединение      */
#define BTF_KIND_ENUM           6       /* Перечисляемые значения до 32 битов     */
#define BTF_KIND_FWD            7       /* Forward      */
#define BTF_KIND_TYPEDEF        8       /* Определение типа*/
#define BTF_KIND_VOLATILE       9       /* Переменная (Volatile)   */
#define BTF_KIND_CONST          10      /* Константа        */
#define BTF_KIND_RESTRICT       11      /* Ограничение      */
#define BTF_KIND_FUNC           12      /* Функция          */
#define BTF_KIND_FUNC_PROTO     13      /* Прототип функции*/
#define BTF_KIND_VAR            14      /* Переменная       */
#define BTF_KIND_DATASEC        15      /* Раздел           */
#define BTF_KIND_FLOAT          16      /* Число с плавающей запятой      */
#define BTF_KIND_DECL_TAG       17      /* Тег объявления  */
#define BTF_KIND_TYPE_TAG       18      /* Тег типа         */
#define BTF_KIND_ENUM64         19      /* Перечисляемые значения до 64 битов     */
```

Отметим, что раздел типов представляет отладочные сведения, а не просто типы, BTF_KIND_FUNC представляет не тип, а определённую подпрограмму.

Каждый тип включает указанные ниже общие данные.

```
struct btf_type {
    __u32 name_off;
    /* Набор информационных битов (info);
     * 0-15:  vlen (например, число элементов в структуре);
     * 16-23: не используются;
     * 24-28: тип (например, int, ptr, array и т. п.);
     * 29-30: не используются;
     * 31:    kind_flag, флаг типа, применяемый в настоящее время
     *        для STRUCT, UNION, FWD, ENUM и ENUM64.
     */
    __u32 info;
    /* Размер (size) используется INT, ENUM, STRUCT, UNION и ENUM64,
     * указывая размер описываемого типа.
     * Тип (type) используется PTR, TYPEDEF, VOLATILE, CONST, RESTRICT,
     * FUNC, FUNC_PROTO, DECL_TAG и TYPE_TAG, указывая type_id
     * другого типа.
     */
    union {
            __u32 size;
            __u32 type;
    };
};
```

Для некоторых типов за базовой структурой следуют специфические для этого типа данные. Поле name_off в структуре btf_type указывает смещение в таблице строк. Кодирование каждого типа рассматривается ниже.

2.2.1 BTF_KIND_INT

name_off - любое допустимое смещение;
info.kind_flag - 0;
info.kind - BTF_KIND_INT;
info.vlen - 0;
size - размер в байтах, заданный целым числом (int).

После btf_type размещается поле u32 с показанной ниже структурой битов

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

```
#define BTF_INT_BITS(VAL)       ((VAL)  & 0x000000ff)
```

BTF_INT_ENCODING представляет указанные ниже атрибуты

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

BTF_INT_ENCODING() предоставляет дополнительные сведения о назначении типа int - наличие знака (signedness), char или bool. Кодирование char и bool полезно в основном для форматирования вывода. Для типа int можно указать не более одного назначения (кодировки).

BTF_INT_BITS() указывает фактическое число битов этого типа целого числа (int). Например, 4-битовое поле (bitfield) кодирует BTF_INT_BITS() = 4. Значение btf_type.size * 8 должно быть не меньше BTF_INT_BITS() для типа. Максимальное значение BTF_INT_BITS() составляет 128.

BTF_INT_OFFSET() указывает битовое смещение для расчёта значения этого целого числа. Например, элемент битовой структуры имеет:

- смещение элемента btf от начала структуры - 100;
- элемент btf указывает тип int;
- тип int имеет BTF_INT_OFFSET() = 2 и BTF_INT_BITS() = 4.

Тогда в памяти структуры этот элемент будет занимать 4 бита, начиная с позиции 100 + 2 = 102.

Как вариант, для доступа к этому битовому полю можно использовать:

- смещение элемента btf от начала структуры - 102;
- элемент btf указывает тип int;
- тип int имеет BTF_INT_OFFSET() = 0 и BTF_INT_BITS() = 4.

Исходным назначением BTF_INT_OFFSET() является обеспечение гибкости кодирования битовых полей. В настоящее время llvm и pahole генерируют BTF_INT_OFFSET() = 0 для всех типов int.

2.2.2 BTF_KIND_PTR

name_off - 0;
info.kind_flag - 0;
info.kind - BTF_KIND_PTR;
info.vlen - 0;
type - указатель.

После btf_type нет дополнительных данных.

2.2.3 BTF_KIND_ARRAY

name_off - 0;
info.kind_flag - 0;
info.kind - BTF_KIND_ARRAY;
info.vlen - 0;
size/type - 0, не используется.

После btf_type следует структура btf_array

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

type - тип элемента;
index_type - тип индекса;
nelems - число элементов в массиве (возможно, 0).

Поле index_type может иметь обычный целочисленный тип (u8, u16, u32, u64, unsigned __int128). Изначально index_type следовал DWARF, где имеется index_type для типа array. В настоящее время index_type применяется в BTF лишь для проверки типа.

Структура btf_array позволяет использовать цепочку типов элементов для создания многомерных массивов. Например, для int a[5][6] цепочку может представлять приведённая ниже информация о типах.

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

В настоящее время pahole и llvm сворачивают многомерные массивы в одномерные, например, для a[5][6] поле btf_array.nelems будет иметь значение 30. Это связано с тем, что первоначальным вариантом применения была печать отображения, где достаточно было выгрузки всего массива. По мере расширения применений BTF пакеты pahole и llvm могут быть изменены для генерации подобающего связанного представления многомерных массивов.

2.2.4 BTF_KIND_STRUCT u BTF_KIND_UNION

name_off - 0 или смещение действительного идентификатора C;
info.kind_flag - 0 или 1;
info.kind - BTF_KIND_STRUCT или BTF_KIND_UNION;
info.vlen - число элементов структуры или объединения;
info.size - размер структуры или объединения в байтах.

После btf_type следует info.vlen структур btf_member

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

name_off - смещение действительного идентификатора C;
type - тип элемента;
offset - см. ниже.

Если флаг info.kind_flag не установлен (0), поле offset содержит лишь смещение элемента. Отметим, что базовым типом для битового поля может быть только int или enum. Если битовое поле имеет размер 32, базой может быть int или enum, при других размерах - только int и целочисленный тип BTF_INT_BITS() указывает размер битового поля.

При установленном (1) флаге kind_flag поле btf_member.offset содержит размер и смещение битового поля

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

Если в этом случае базовым является тип int, это должно быть обычное целое число

BTF_INT_OFFSET() = 0.
BTF_INT_BITS() = {1, 2, 4, 8, 16} * 8.

Указанное ниже исправление (patch) в ядре добавляет kind_flag и разъясняет существование обоих режимов.

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

2.2.6 BTF_KIND_ENUM

name_off - 0 или смещение действительного идентификатора C;
info.kind_flag - 0 для беззнакового, 1 для числа со знаком;
info.kind - BTF_KIND_ENUM
info.vlen - количество перечисляемых значений;
size - 1/2/4/8.

После btf_type следует info.vlen структур btf_enum

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

name_off - смещение действительного идентификатора C;
val - любое значение.

Если исходное перечисляемое значение имеет знак и его размер меньше 4, значение будет расширено до 4 байтов (со знаком). Если размер равен 8, значение будет усечено до 4 байтов.

2.2.7 BTF_KIND_FWD

name_off - смещение действительного идентификатора C;
info.kind_flag - 0 для struct, 1 для union;
info.kind - BTF_KIND_FWD;
info.vlen - 0;
type - 0.

После btf_type нет дополнительных данных.

2.2.8 BTF_KIND_TYPEDEF

name_off - смещение действительного идентификатора C;
info.kind_flag - 0;
info.kind - BTF_KIND_TYPEDEF;
info.vlen - 0;
type - тип, который можно указать именем, заданным со смещением name_off.

После btf_type нет дополнительных данных.

2.2.9 BTF_KIND_VOLATILE

name_off - 0;
info.kind_flag - 0;
info.kind - BTF_KIND_VOLATILE;
info.vlen - 0;
type - тип с изменяемым классификатором.

После btf_type нет дополнительных данных.

2.2.10 BTF_KIND_CONST

name_off - 0;
info.kind_flag - 0;
info.kind - BTF_KIND_CONST
info.vlen - 0;
type - тип с постоянным классификатором.

После btf_type нет дополнительных данных.

2.2.11 BTF_KIND_RESTRICT

name_off - 0;
info.kind_flag - 0;
info.kind - BTF_KIND_RESTRICT;

info.vlen - 0;
type - тип с ограничительным классификатором.
После btf_type нет дополнительных данных.

2.2.12 BTF_KIND_FUNC

name_off - смещение действительного идентификатора C;
info.kind_flag - 0;
info.kind - BTF_KIND_FUNC;
info.vlen - компоновочные сведения (BTF_FUNC_STATIC, BTF_FUNC_GLOBAL или BTF_FUNC_EXTERN);
type - тип BTF_KIND_FUNC_PROTO.
После btf_type нет дополнительных данных.

BTF_KIND_FUNC определяет не тип, а подпрограмму (функцию), подпись которой определяет тип. Подпрограмма является экземпляром этого типа. На BTF_KIND_FUNC может указывать func_info в 4.2 Раздел .BTF.ext (ELF) или аргументы 3.3 BPF_PROG_LOAD (ABI).

В настоящее время ядро поддерживает только привязки BTF_FUNC_STATIC и BTF_FUNC_GLOBAL.

2.2.13 BTF_KIND_FUNC_PROTO

name_off - 0;
info.kind_flag - 0;
info.kind - BTF_KIND_FUNC_PROTO;
info.vlen - число параметров;
type - тип возвращаемого значения.
После btf_type следует info.vlen структур btf_param

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

Если BTF_KIND_FUNC_PROTO указывается типом BTF_KIND_FUNC, поле btf_param.name_off должно указывать действительный идентификатор C, за исключением, возможно, последнего аргумента, представляющего переменную. Поле btf_param.type указывает тип параметра.

Если функция имеет переменное число аргументов, последний параметр кодируется с name_off = 0 и type = 0.

2.2.14 BTF_KIND_VAR

name_off - смещение действительного идентификатора C;
info.kind_flag - 0;
info.kind - BTF_KIND_VAR;
info.vlen - 0;
type - тип переменной.
После btf_type следует 1 структура btf_variable

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

В структуре btf_var элемент linkage может указывать статическую переменную (0) или глобальную переменную в разделах ELF (1)

В настоящее время LLVM поддерживает не все типы глобальных переменных, а лишь:

- статические переменные с атрибутами раздела или без них;
- глобальные переменные с атрибутами раздела.

Последний вариант предназначен для будущего извлечения идентификаторов типа «ключ-значение» из определения сопоставлений.

2.2.15 BTF_KIND_DATASEC

name_off - смещение действительного имени, связанного с переменной или одним из .data/.bss/.rodata;
info.kind_flag - 0;
info.kind - BTF_KIND_DATASEC;
info.vlen - число переменных;
size - общий размер раздела в байтах (0 во время компиляции, заменяется фактическим значением загрузчиками BPF, такими как libbpf).
После btf_type следует info.vlen структур btf_var_secinfo

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

type - тип переменной BTF_KIND_VAR;
offset - смещение переменной в разделе;
size - размер переменной в байтах.

2.2.16 BTF_KIND_FLOAT

name_off - любое действительное смещение;
info.kind_flag - 0;

info.kind - BTF_KIND_FLOAT;
info.vlen - 0;
size - размер типа float в байтах (2, 4, 8, 12 или 16).

После btf_type нет дополнительных данных.

2.2.17 BTF_KIND_DECL_TAG

name_off - смещение непустой строки;
info.kind_flag - 0;
info.kind - BTF_KIND_DECL_TAG;
info.vlen - 0;
type - struct, union, func, var или typedef.

После btf_type следует структура btf_decl_tag

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

Поле name_off указывает строку атрибута btf_decl_tag, которая может иметь тип struct, union, func, var или typedef. Для типа var или typedef должно устанавливаться btf_decl_tag.component_idx = -1. Для трёх оставшихся типов устанавливается -1, если атрибут btf_decl_tag применяется непосредственно к struct, union или func. В ином случае атрибут применяется к элементу struct или union или аргументу функции и в поле btf_decl_tag.component_idx следует указывать фактический индекс (начиная с 0) элемента или аргумента.

2.2.18 BTF_KIND_TYPE_TAG

name_off - смещение непустой строки;
info.kind_flag - 0;
info.kind - BTF_KIND_TYPE_TAG;
info.vlen - 0;
type - тип с атрибутом btf_type_tag.

В настоящее время BTF_KIND_TYPE_TAG создаётся лишь для указательных типов и имеет цепочку типов

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

Тип pointer фактически может указывать на несколько (включая 0) type_tag, затем (необязательно) на const, volatile, restrict, typedef и в конце на базовый тип, которым может служить int, ptr, array, struct, union, enum, func_proto или float.

2.2.19 BTF_KIND_ENUM64

name_off - 0 или смещение действительного идентификатора C;
info.kind_flag - 0 для беззнакового, 1 для числа со знаком;
info.kind - BTF_KIND_ENUM64;
info.vlen - число перечисляемых (enum) значений;
size - 1, 2, 4 или 8.

После btf_type следует info.vlen структур btf_enum64

```
struct btf_enum64 {
    __u32   name_off;
    __u32   val_lo32;
    __u32   val_hi32;
};
```

name_off - смещение действительного идентификатора C;
val_lo32 - 32 младших бита 64-битового значения;
val_hi32 - 32 старших бита 64-битового значения.

Если исходное значение enum имеет знак и его размер меньше 8, значение расширяется до 8 байтов (со знаком).

3. API ядра

Ниже указаны системные вызовы bpf, использующие BTF

- BPF_BTF_LOAD - загрузка блока данных (blob) BTF в ядро.
- BPF_MAP_CREATE - создание отображения с ключом btf и информацией о типе значения.
- BPF_PROG_LOAD - загрузка программы с функцией btf и информацией о строке.
- BPF_BTF_GET_FD_BY_ID - получение файлового дескриптора btf.
- BPF_OBJ_GET_INFO_BY_FD - возврат btf, func_info, line_info и других сведений, относящихся к btf.

Типичный рабочий процесс показан ниже.

Приложение

```
    BPF_BTF_LOAD
        |
        v
    BPF_MAP_CREATE и BPF_PROG_LOAD
        |
        V
    ......
```

```
    ......
    BPF_{PROG,MAP}_GET_NEXT_ID (get prog/map id)
        |
        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 с btf_id)
        |                                       |
        V                                       |
    BPF_BTF_GET_FD_BY_ID (get btf_fd)           |
        |                                       |
        V                                       |
    BPF_OBJ_GET_INFO_BY_FD (get btf)            |
        |                                       |
        V                                       V
    Типы для красивого вывода, дампов подписей функций, сведений о строках и т. п.
```

3.1 BPF_BTF_LOAD

Загружает blob данных BTF в ядро. Блок данных, описанный в разделе 2. Кодирование типов и строк BTF, можно напрямую загрузить в ядро. В пользовательское пространство возвращается файловый дескриптор btf_fd.

3.2 BPF_MAP_CREATE

Отображение может быть создано с btf_fd и заданной парой «ключ-значение» для идентификатора типа

```
__u32   btf_fd;              /* fd указывает тип данных BTF */
__u32   btf_key_type_id;     /* BTF type_id для ключа */
__u32   btf_value_type_id;   /* BTF type_id для значения */
```

В libbpf отображение может быть задано с дополнительной аннотацией, как показано ниже.

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

При синтаксическом анализе ELF libbpf может извлекать идентификаторы типа «ключ-значение» и автоматически назначить их атрибутам BPF_MAP_CREATE.

3.3 BPF_PROG_LOAD

При загрузке программы (prog_load) можно передать в ядро func_info и line_info с подобающими значениями указанных ниже атрибутов.

```
__u32           insn_cnt;
__aligned_u64   insns;
......
__u32           prog_btf_fd;         /* fd с указанием типа данных BTF */
__u32           func_info_rec_size;  /* Размер bpf_func_info в пользовательском пространстве */
__aligned_u64   func_info;           /* Сведения о функции */
__u32           func_info_cnt;       /* Число записей bpf_func_info */
__u32           line_info_rec_size;  /* Размер bpf_line_info в пользовательском пространстве */
__aligned_u64   line_info;           /* Сведения о строке */
__u32           line_info_cnt;       /* Число записей bpf_line_info */
```

Элементы func_info и line_info являются массивами, как показано ниже.

```
struct bpf_func_info {
    __u32   insn_off; /* [0, insn_cnt - 1] */
    __u32   type_id;  /* Указывает тип BTF_KIND_FUNC */
};
struct bpf_line_info {
    __u32   insn_off;         /* [0, insn_cnt - 1] */
    __u32   file_name_off;    /* Смещение имени файла в таблице строк */
    __u32   line_off;         /* Смещение строки исходного текста в таблице строк */
    __u32   line_col;         /* Номер строки и позиция в ней */
};
```

Поле func_info_rec_size указывает размер записи func_info, а line_info_rec_size - размер каждой записи line_info. Передача размера записи в ядро позволяет позднее извлечь саму запись.

Ниже указаны требования к func_info

- func_info[0].insn_off = 0;
- значения func_info.insn_off строго возрастают по порядку и соответствуют границам функции bpf.

Ниже указаны требования к line_info:

- первый элемент insn в каждой функции должен иметь запись line_info, указывающую на него;
- line_info.insn_off строго возрастают по порядку.

Для line_info номера строк и позиций определяются, как показано ниже

```
#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

В ядре каждая загруженная программа, сопоставление (map) или btf имеет уникальный идентификатор, который не меняется в течение срока действия программы, сопоставления или btf.

Команды системных вызовов bpf BPF_{PROG,MAP}_GET_NEXT_ID возвращают в пользовательское пространство все идентификаторы (по 1 на команду) программ или отображений bpf, чтобы инструменты могли проверить все программы и отображения.

3.5 BPF_{PROG,MAP}_GET_FD_BY_ID

Инструмент самоанализ не может применять идентификатор для получения подробных сведений о программах или отображениях. Сначала нужно получить дескриптор файла для подсчёта ссылок.

3.6 BPF_OBJ_GET_INFO_BY_FD

После получения дескриптора программы или отображения инструмент самоанализа может получить от ядра подробные сведения, часть которых относится к BTF. Например, bpf_map_info возвращает btf_id и «ключ-значение» для идентификаторов типа, а bpf_prog_info возвращает btf_id, func_info и сведения о строке для оттранслированного байт-кода bpf, а также jited_line_info.

3.7 BPF_BTF_GET_FD_BY_ID

С помощью btf_id, полученного в bpf_map_info или bpf_prog_info системный вызов bpf BPF_BTF_GET_FD_BY_ID может извлечь файловый дескриптор btf. Затем с помощью команды BPF_OBJ_GET_INFO_BY_FD можно получить btf blob, исходно загруженный в ядро с помощью BPF_BTF_LOAD. С btf blob, bpf_map_info и bpf_prog_info инструмент самоанализа получает полные сведения о btf и может обеспечить красивый вывод пар «ключ-значение», дампов подписей функций и сведений о строка вместе с кодами byte/jit.

4. Формат файлов ELF

4.1 Раздел .BTF

Раздел .BTF содержит сведения о типах и строках в формате, описанном в 2. Кодирование типов и строк BTF.

4.2 Раздел .BTF.ext

Раздел .BTF.ext кодирует func_info и line_info, которые загрузчик должен обрабатывать перед отправкой в ядро. Спецификация раздела .BTF.ext задана в файлах tools/lib/bpf/btf.h и tools/lib/bpf/btf.c.

Ниже показан текущий заголовок раздела .BTF.ext.

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

    /* Все смещения указаны в байтах от конца этого заголовк */
    __u32   func_info_off;
    __u32   func_info_len;
    __u32   line_info_off;
    __u32   line_info_len;
};
```

Это очень похоже на раздел .BTF, но вместо типов и строк этот раздел содержит записи func_info и line_info, формат которых описан в параграфе 3.3 BPF_PROG_LOAD.

Организация func_info показана ниже.

```
func_info_rec_size
btf_ext_info_sec /* func_info для раздела 1 */
btf_ext_info_sec /* func_info для раздела 2 */
...
```

Поле func_info_rec_size задаёт размер структуры bpf_func_info при генерации .BTF.ext, а показанная ниже структура btf_ext_info_sec является набором func_info для каждого конкретного раздела ELF.

```
struct btf_ext_info_sec {
   __u32   sec_name_off; /* Смещение имени раздела */
   __u32   num_info;
   /* Далее следует num_info * record_size байтов */
   __u8    data[0];
};
```

Поле num_info должно быть больше 0. Организация line_info показана ниже.

```
line_info_rec_size
btf_ext_info_sec /* line_info для раздела 1 */
btf_ext_info_sec /* line_info для раздела 2 */
...
```

Поле line_info_rec_size указывает размер структуры bpf_line_info при генерации .BTF.ext.

Интерпретация bpf_func_info->insn_off и bpf_line_info->insn_off различается в API ядра и ELF API. Для API ядра insn_off указывает смещение инструкции в структурах bpf_insn. Для ELF API поле insn_off указывает смещение в байтах от начала раздела (btf_ext_info_sec->sec_name_off).

4.2 Раздел .BTF_ids

Раздел .BTF_ids кодирует значения BTF ID, которые используются в ядре. Раздел создаётся при компиляции ядра с помощью макросов, заданных в заголовочном файле include/linux/btf_ids.h. Затем код ядра может использоваться для создания списков и наборов (сортированных списков) значений BTF ID.

Макросы BTF_ID_LIST и BTF_ID определяют несортированный список значений BTF ID, как показано ниже.

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

Это даёт показанную ниже схему раздела .BTF_ids.

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

Определена переменная u32 list[] для доступа к списку.

Макрос BTF_ID_UNUSED определяет 4 нулевых байта, которые служат для задания в списке BTF_ID_LIST неиспользуемой записи, как показано ниже.

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

Макросы BTF_SET_START и BTF_SET_END определяют сортированный список значений BTF ID и их числа, как показано ниже.

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

Это даёт показанную ниже схему раздела .BTF_ids.

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

Определена структура btf_id_set set для доступа к этому списку.

В качестве имени typeX может использоваться struct, union, typedef, func и это служит фильтром при извлечении значения BTF ID.

Все списки и наборы BTF ID компилируются в раздел .BTF_ids и распознаются на этапе компоновки при сборке ядра с помощью инструмента resolve_btfids.

5. Применение BTF

5.1 Красивый вывод отображений с помощью bpftool

С помощью BTF отображения !ключ-значение! Можно выводить по полям, а не в форме необработанных байтов. Это особенно полезно для больших структур и структур с битовыми полями, например, для приведённого сопоставления.

```
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");
```

Команда bpftool позволяет вывести его в форме

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

5.2 Дамп программы с помощью bpftool

Ниже показано, как func_info и line_info может помочь при выводе дампа программы с именами символов ядра, прототипами функций и сведениями о строках.

```
$ 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 Журнал проверки

Ниже показано, как line_info может помочь при отладке.

```
   /* Код tools/testing/selftests/bpf/test_xdp_noinline.c изменён,
    * как показано ниже.
    */
   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. Генерация BTF

Сначала нужно установить pahole или llvm (8.0 или выше). Программа pahole служит конвертером dwarf в btf и пока не поддерживает .BTF.ext и тип BTF_KIND_FUNC. Например,

```
-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
```

Файл 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
```

Пакет llvm может генерировать .BTF и .BTF.ext напрямую с опцией -g лишь для цели bpf. Опция ассемблера (-S) позволяет представить кодирование BTF в формате ассемблера.

```
-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. Тестирование

Программа самотестирования BPF в ядре (tools/testing/selftests/bpf/prog_tests/btf.c) обеспечивает широкий набор связанных с BTF тестов.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru