Linux Bpf Documentation

The kernel development community

Jun 10, 2024

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 Cilium project also maintains a BPF and XDP Reference Guide that goes into great technical depth about the BPF Architecture.

CHAPTER

ONE

EBPF VERIFIER

The safety of the eBPF program is determined in two steps.

First step does DAG check to disallow loops and other CFG validation. In particular it will detect programs that have unreachable instructions. (though classic BPF checker allows them)

Second step starts from the first insn and descends all possible paths. It simulates execution of every insn and observes the state change of registers and stack.

At the start of the program the register R1 contains a pointer to context and has type PTR_TO_CTX. If verifier sees an insn that does R2=R1, then R2 has now type PTR_TO_CTX as well and can be used on the right hand side of expression. If R1=PTR_TO_CTX and insn is R2=R1+R1, then R2=SCALAR_VALUE, since addition of two valid pointers makes invalid pointer. (In 'secure' mode verifier will reject any type of pointer arithmetic to make sure that kernel addresses don't leak to unprivileged users)

If register was never written to, it's not readable:

```
bpf_mov R0 = R2
bpf_exit
```

will be rejected, since R2 is unreadable at the start of the program.

After kernel function call, R1-R5 are reset to unreadable and R0 has a return type of the function.

Since R6-R9 are callee saved, their state is preserved across the call.

```
bpf_mov R6 = 1
bpf_call foo
bpf_mov R0 = R6
bpf_exit
```

is a correct program. If there was R1 instead of R6, it would have been rejected.

load/store instructions are allowed only with registers of valid types, which are PTR_TO_CTX, PTR_TO_MAP, PTR_TO_STACK. They are bounds and alignment checked. For example:

```
bpf_mov R1 = 1
bpf_mov R2 = 2
bpf_xadd *(u32 *)(R1 + 3) += R2
bpf_exit
```

will be rejected, since R1 doesn't have a valid pointer type at the time of execution of instruction bpf_xadd.

At the start R1 type is PTR_TO_CTX (a pointer to generic `struct bpf_context`) A callback is used to customize verifier to restrict eBPF program access to only certain fields within ctx structure with specified size and alignment.

For example, the following insn:

```
bpf_ld R0 = *(u32 *)(R6 + 8)
```

intends to load a word from address R6 + 8 and store it into R0 If R6=PTR_TO_CTX, via is_valid_access() callback the verifier will know that offset 8 of size 4 bytes can be accessed for reading, otherwise the verifier will reject the program. If R6=PTR_TO_STACK, then access should be aligned and be within stack bounds, which are [-MAX_BPF_STACK, 0). In this example offset is 8, so it will fail verification, since it's out of bounds.

The verifier will allow eBPF program to read data from stack only after it wrote into it.

Classic BPF verifier does similar check with M[0-15] memory slots. For example:

```
bpf_ld R0 = *(u32 *)(R10 - 4)
bpf_exit
```

is invalid program. Though R10 is correct read-only register and has type PTR_TO_STACK and R10 - 4 is within stack bounds, there were no stores into that location.

Pointer register spill/fill is tracked as well, since four (R6-R9) callee saved registers may not be enough for some programs.

Allowed function calls are customized with bpf_verifier_ops->get_func_proto() The eBPF verifier will check that registers match argument constraints. After the call register R0 will be set to return type of the function.

Function calls is a main mechanism to extend functionality of eBPF programs. Socket filters may let programs to call one set of functions, whereas tracing filters may allow completely different set.

If a function made accessible to eBPF program, it needs to be thought through from safety point of view. The verifier will guarantee that the function is called with valid arguments.

seccomp vs socket filters have different security restrictions for classic BPF. Seccomp solves this by two stage verifier: classic BPF verifier is followed by seccomp verifier. In case of eBPF one configurable verifier is shared for all use cases.

See details of eBPF verifier in kernel/bpf/verifier.c

1.1 Register value tracking

In order to determine the safety of an eBPF program, the verifier must track the range of possible values in each register and also in each stack slot. This is done with `struct bpf_reg_state`, defined in include/linux/ bpf_verifier.h, which unifies tracking of scalar and pointer values. Each register state has a type, which is either NOT_INIT (the register has not been written to), SCALAR_VALUE (some value which is not usable as a pointer), or a pointer type. The types of pointers describe their base, as follows:

PTR_TO_CTX
Pointer to bpf_context.

CONST_PTR_TO_MAP
Pointer to struct bpf_map. "Const" because arithmetic on these pointers is forbidden.

PTR_TO_MAP_VALUE
Pointer to the value stored in a map element.

PTR_TO_MAP_VALUE_OR_NULL
Either a pointer to a map value, or NULL; map accesses (see *BPF maps*) return this type, which becomes a PTR_TO_MAP_VALUE when checked != NULL. Arithmetic on these pointers is forbidden.

PTR_TO_STACK
Frame pointer.

PTR_TO_PACKET
skb->data.

PTR_TO_PACKET_END
skb->data + headlen; arithmetic forbidden.

PTR_TO_SOCKET
Pointer to struct bpf_sock_ops, implicitly refcounted.

PTR_TO_SOCKET_OR_NULL
Either a pointer to a socket, or NULL; socket lookup returns this type, which becomes a PTR_TO_SOCKET when checked != NULL. PTR_TO_SOCKET is reference-counted, so programs must release the reference through the socket release function before the end of the program. Arithmetic on these pointers is forbidden.

However, a pointer may be offset from this base (as a result of pointer arithmetic), and this is tracked in two parts: the 'fixed offset' and 'variable offset'. The former is used when an exactly-known value (e.g. an immediate operand) is added to a pointer, while the latter is used for values which are not exactly known. The variable offset is also used in SCALAR_VALUEs, to track the range of possible values in the register.

The verifier's knowledge about the variable offset consists of:

- minimum and maximum values as unsigned
- minimum and maximum values as signed
- knowledge of the values of individual bits, in the form of a 'tnum': a u64 'mask' and a u64 'value'. 1s in the mask represent bits whose value is unknown; 1s in the value represent bits known to be 1. Bits known to be 0 have 0 in both mask and value; no bit should ever be 1 in both. For example, if a byte is read into a register from memory, the register's top 56 bits are known zero, while the low 8 are unknown - which is represented as the tnum (0x0; 0xff). If we then OR this with 0x40, we get (0x40; 0xbf), then if we add 1 we get (0x0; 0x1ff), because of potential carries.

Besides arithmetic, the register state can also be updated by conditional branches. For instance, if a SCALAR_VALUE is compared > 8, in the 'true' branch it will have a umin_value (unsigned minimum value) of 9, whereas in the 'false' branch it will have a umax_value of 8. A signed compare (with BPF_JSGT or BPF_JSGE) would instead update the signed minimum/maximum values. Information from the signed and unsigned bounds can be combined; for instance if a value is first tested < 8 and then tested s> 4, the verifier will conclude that the value is also > 4 and s< 8, since the bounds prevent crossing the sign boundary.

PTR_TO_PACKETs with a variable offset part have an 'id', which is common to all pointers sharing that same variable offset. This is important for packet range checks: after adding a variable to a packet pointer register A, if you then copy it to another register B and then add a constant 4 to A, both registers will share the same 'id' but the A will have a fixed offset of +4. Then if A is bounds-checked and found to be less than a PTR_TO_PACKET_END, the register B is now known to have a safe range of at least 4 bytes. See 'Direct packet access', below, for more on PTR_TO_PACKET ranges.

The 'id' field is also used on PTR_TO_MAP_VALUE_OR_NULL, common to all copies of the pointer returned from a map lookup. This means that when one copy is checked and found to be non-NULL, all copies can become PTR_TO_MAP_VALUEs. As well as range-checking, the tracked information is also used for enforcing alignment of pointer accesses. For instance, on most systems the packet pointer is 2 bytes after a 4-byte alignment. If a program adds 14 bytes to that to jump over the Ethernet header, then reads IHL and adds (IHL * 4), the resulting pointer will have a variable offset known to be 4n+2 for some n, so adding the 2 bytes (NET_IP_ALIGN) gives a 4-byte alignment and so word-sized accesses through that pointer are safe. The 'id' field is also used on PTR_TO_SOCKET and PTR_TO_SOCKET_OR_NULL, common to all copies of the pointer returned from a socket lookup. This has similar behaviour to the handling for PTR_TO_MAP_VALUE_OR_NULL->PTR_TO_MAP_VALUE, but it also handles reference tracking for the pointer. PTR_TO_SOCKET implicitly represents a reference to the corresponding `struct sock`. To ensure that the reference is not leaked, it is imperative to NULL-check the reference and in the non-NULL case, and pass the valid reference to the socket release function.

1.2 Direct packet access

In cls_bpf and act_bpf programs the verifier allows direct access to the packet data via skb->data and skb->data_end pointers. Ex:

```
1:  r4 = *(u32 *)(r1 +80)  /* load skb->data_end */
2:  r3 = *(u32 *)(r1 +76)  /* load skb->data */
3:  r5 = r3
4:  r5 += 14
5:  if r5 > r4 goto pc+16
R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt_end R5=pkt(id=0,off=14,r=14) R10=fp
6:  r0 = *(u16 *)(r3 +12) /* access 12 and 13 bytes of the packet */
```

this 2byte load from the packet is safe to do, since the program author did check `if (skb->data + 14 > skb->data_end) goto err` at insn #5 which means that in the fall-through case the register R3 (which points to skb->data) has at least 14 directly accessible bytes. The verifier marks it as R3=pkt(id=0,off=0,r=14). id=0 means that no additional variables were added to the register. off=0 means that no additional constants were added. r=14 is the range of safe access which means that bytes [R3, R3 + 14) are ok. Note that R5 is marked as R5=pkt(id=0,off=14,r=14). It also points to the packet data, but constant 14 was added to the register, so it now points to `skb->data + 14` and accessible range is [R5, R5 + 14 - 14) which is zero bytes.

More complex packet access may look like:

```
R0=inv1 R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt_end R5=pkt(id=0,off=14,r=14)␣
↪R10=fp
```

```
6:  r0 = *(u8 *)(r3 +7) /* load 7th byte from the packet */
7:  r4 = *(u8 *)(r3 +12)
8:  r4 *= 14
9:  r3 = *(u32 *)(r1 +76) /* load skb->data */
10:  r3 += r4
11:  r2 = r1
12:  r2 <<= 48
13:  r2 >>= 48
14:  r3 += r2
15:  r2 = r3
16:  r2 += 8
17:  r1 = *(u32 *)(r1 +80) /* load skb->data_end */
18:  if r2 > r1 goto pc+2
R0=inv(id=0,umax_value=255,var_off=(0x0; 0xff)) R1=pkt_end R2=pkt(id=2,off=8,
 ↪r=8) R3=pkt(id=2,off=0,r=8) R4=inv(id=0,umax_value=3570,var_off=(0x0;␣
 ↪0xfffe)) R5=pkt(id=0,off=14,r=14) R10=fp
19:  r1 = *(u8 *)(r3 +4)
```

The state of the register R3 is R3=pkt(id=2,off=0,r=8) id=2 means that two `r3 += rX` instructions were seen, so r3 points to some offset within a packet and since the program author did `if (r3 + 8 > r1) goto err` at insn #18, the safe range is [R3, R3 + 8). The verifier only allows 'add'/'sub' operations on packet registers. Any other operation will set the register state to 'SCALAR_VALUE' and it won't be available for direct packet access.

Operation `r3 += rX` may overflow and become less than original skb->data, therefore the verifier has to prevent that. So when it sees `r3 += rX` instruction and rX is more than 16-bit value, any subsequent bounds-check of r3 against skb->data_end will not give us 'range' information, so attempts to read through the pointer will give "invalid access to packet" error.

Ex. after insn `r4 = *(u8 *)(r3 +12)` (insn #7 above) the state of r4 is R4=inv(id=0,umax_value=255,var_off=(0x0; 0xff)) which means that upper 56 bits of the register are guaranteed to be zero, and nothing is known about the lower 8 bits. After insn `r4 *= 14` the state becomes R4=inv(id=0,umax_value=3570,var_off=(0x0; 0xfffe)), since multiplying an 8-bit value by constant 14 will keep upper 52 bits as zero, also the least significant bit will be zero as 14 is even. Similarly `r2 >>= 48` will make R2=inv(id=0,umax_value=65535,var_off=(0x0; 0xffff)), since the shift is not sign extending. This logic is implemented in adjust_reg_min_max_vals() function, which calls adjust_ptr_min_max_vals() for adding pointer to scalar (or vice versa) and adjust_scalar_min_max_vals() for operations on two scalars.

The end result is that bpf program author can access packet directly using normal C code as:

```
void *data = (void *)(long)skb->data;
void *data_end = (void *)(long)skb->data_end;
struct eth_hdr *eth = data;
struct iphdr *iph = data + sizeof(*eth);
struct udphdr *udp = data + sizeof(*eth) + sizeof(*iph);

if (data + sizeof(*eth) + sizeof(*iph) + sizeof(*udp) > data_end)
        return 0;
if (eth->h_proto != htons(ETH_P_IP))
        return 0;
```

```
if (iph->protocol != IPPROTO_UDP || iph->ihl != 5)
        return 0;
if (udp->dest == 53 || udp->source == 9)
        ...;
```

which makes such programs easier to write comparing to LD_ABS insn and significantly faster.

1.3 Pruning

The verifier does not actually walk all possible paths through the program. For each new branch to analyse, the verifier looks at all the states it's previously been in when at this instruction. If any of them contain the current state as a subset, the branch is 'pruned' - that is, the fact that the previous state was accepted implies the current state would be as well. For instance, if in the previous state, r1 held a packet-pointer, and in the current state, r1 holds a packet-pointer with a range as long or longer and at least as strict an alignment, then r1 is safe. Similarly, if r2 was NOT_INIT before then it can't have been used by any path from that point, so any value in r2 (including another NOT_INIT) is safe. The implementation is in the function regsafe(). Pruning considers not only the registers but also the stack (and any spilled registers it may hold). They must all be safe for the branch to be pruned. This is implemented in states_equal().

Some technical details about state pruning implementation could be found below.

1.3.1 Register liveness tracking

In order to make state pruning effective, liveness state is tracked for each register and stack slot. The basic idea is to track which registers and stack slots are actually used during subseqeuent execution of the program, until program exit is reached. Registers and stack slots that were never used could be removed from the cached state thus making more states equivalent to a cached state. This could be illustrated by the following program:

```
0: call bpf_get_prandom_u32()
1: r1 = 0
2: if r0 == 0 goto +1
3: r0 = 1
--- checkpoint ---
4: r0 = r1
5: exit
```

Suppose that a state cache entry is created at instruction #4 (such entries are also called "checkpoints" in the text below). The verifier could reach the instruction with one of two possible register states:

- r0 = 1, r1 = 0
- r0 = 0, r1 = 0

However, only the value of register `r1` is important to successfully finish verification. The goal of the liveness tracking algorithm is to spot this fact and figure out that both states are actually equivalent.

Data structures

Liveness is tracked using the following data structures:

```
enum bpf_reg_liveness {
      REG_LIVE_NONE = 0,
      REG_LIVE_READ32 = 0x1,
      REG_LIVE_READ64 = 0x2,
      REG_LIVE_READ = REG_LIVE_READ32 | REG_LIVE_READ64,
      REG_LIVE_WRITTEN = 0x4,
      REG_LIVE_DONE = 0x8,
};

struct bpf_reg_state {
      ...
      struct bpf_reg_state *parent;
      ...
      enum bpf_reg_liveness live;
      ...
};

struct bpf_stack_state {
      struct bpf_reg_state spilled_ptr;
      ...
};

struct bpf_func_state {
      struct bpf_reg_state regs[MAX_BPF_REG];
      ...
      struct bpf_stack_state *stack;
}

struct bpf_verifier_state {
      struct bpf_func_state *frame[MAX_CALL_FRAMES];
      struct bpf_verifier_state *parent;
      ...
}
```

- `REG_LIVE_NONE` is an initial value assigned to `->live` fields upon new verifier state creation;
- `REG_LIVE_WRITTEN` means that the value of the register (or stack slot) is defined by some instruction verified between this verifier state's parent and verifier state itself;
- `REG_LIVE_READ{32,64}` means that the value of the register (or stack slot) is read by a some child state of this verifier state;
- `REG_LIVE_DONE` is a marker used by `clean_verifier_state()` to avoid processing same verifier state multiple times and for some sanity checks;
- `->live` field values are formed by combining `enum bpf_reg_liveness` values using bitwise or.

Register parentage chains

In order to propagate information between parent and child states, a *register parentage chain* is established. Each register or stack slot is linked to a corresponding register or stack slot in its parent state via a `->parent` pointer. This link is established upon state creation in `is_state_visited()` and might be modified by `set_callee_state()` called from `__check_func_call()`.

The rules for correspondence between registers / stack slots are as follows:

- For the current stack frame, registers and stack slots of the new state are linked to the registers and stack slots of the parent state with the same indices.
- For the outer stack frames, only caller saved registers (r6-r9) and stack slots are linked to the registers and stack slots of the parent state with the same indices.
- When function call is processed a new `struct bpf_func_state` instance is allocated, it encapsulates a new set of registers and stack slots. For this new frame, parent links for r6-r9 and stack slots are set to nil, parent links for r1-r5 are set to match caller r1-r5 parent links.

This could be illustrated by the following diagram (arrows stand for `->parent` pointers):

```
    ...                     ; Frame #0, some instructions
--- checkpoint #0 ---
1 : r6 = 42                 ; Frame #0
--- checkpoint #1 ---
2 : call foo()              ; Frame #0
    ...                     ; Frame #1, instructions from foo()
--- checkpoint #2 ---
    ...                     ; Frame #1, instructions from foo()
--- checkpoint #3 ---
    exit                    ; Frame #1, return from foo()
3 : r1 = r6                 ; Frame #0  <- current state

           +-------------------------------+-------------------------------+
           |           Frame #0            |           Frame #1            |
Checkpoint +-------------------------------+-------------------------------+
#0         | r0 | r1-r5 | r6-r9 | fp-8 ... |
           +-------------------------------+
              ^    ^       ^       ^
              |    |       |       |
Checkpoint +-------------------------------+
#1         | r0 | r1-r5 | r6-r9 | fp-8 ... |
           +-------------------------------+
                    ^       ^       ^
                    |_______|_______|_______________
                            |       |               |
             nil  nil       |       |               |      nil     nil
              |    |        |       |               |       |       |
Checkpoint +-------------------------------+-------------------------------+
#2         | r0 | r1-r5 | r6-r9 | fp-8 ... | r0 | r1-r5 | r6-r9 | fp-8 ... |
           +-------------------------------+-------------------------------+
                           ^       ^               ^       ^       ^
```

```
             nil  nil      |       |               |       |       |
              |    |       |       |               |       |       |
Checkpoint +-------------------------------+-------------------------------+
#3         | r0 | r1-r5 | r6-r9 | fp-8 ... | r0 | r1-r5 | r6-r9 | fp-8 ... |
           +-------------------------------+-------------------------------+
                           ^       ^
             nil  nil      |       |
              |    |       |       |
Current    +-------------------------------+
state      | r0 | r1-r5 | r6-r9 | fp-8 ... |
           +-------------------------------+
                           \
                             r6 read mark is propagated via these links
                             all the way up to checkpoint #1.
                             The checkpoint #1 contains a write mark for r6
                             because of instruction (1), thus read propagation
                             does not reach checkpoint #0 (see section below).
```

Liveness marks tracking

For each processed instruction, the verifier tracks read and written registers and stack slots. The main idea of the algorithm is that read marks propagate back along the state parentage chain until they hit a write mark, which ‘screens off’ earlier states from the read. The information about reads is propagated by function `mark_reg_read()` which could be summarized as follows:

```
mark_reg_read(struct bpf_reg_state *state, ...):
    parent = state->parent
    while parent:
        if state->live & REG_LIVE_WRITTEN:
            break
        if parent->live & REG_LIVE_READ64:
            break
        parent->live |= REG_LIVE_READ64
        state = parent
        parent = state->parent
```

Notes:

- The read marks are applied to the **parent** state while write marks are applied to the **current** state. The write mark on a register or stack slot means that it is updated by some instruction in the straight-line code leading from the parent state to the current state.
- Details about REG_LIVE_READ32 are omitted.
- Function `propagate_liveness()` (see section *Read marks propagation for cache hits*) might override the first parent link. Please refer to the comments in the `propagate_liveness()` and `mark_reg_read()` source code for further details.

Because stack writes could have different sizes `REG_LIVE_WRITTEN` marks are applied conservatively: stack slots are marked as written only if write size corresponds to the size of the register, e.g. see function `save_register_state()`.

Consider the following example:

```
0: (*u64)(r10 - 8) = 0   ; define 8 bytes of fp-8
--- checkpoint #0 ---
1: (*u32)(r10 - 8) = 1   ; redefine lower 4 bytes
2: r1 = (*u32)(r10 - 8)  ; read lower 4 bytes defined at (1)
3: r2 = (*u32)(r10 - 4)  ; read upper 4 bytes defined at (0)
```

As stated above, the write at (1) does not count as `REG_LIVE_WRITTEN`. Should it be otherwise, the algorithm above wouldn't be able to propagate the read mark from (3) to checkpoint #0.

Once the `BPF_EXIT` instruction is reached `update_branch_counts()` is called to update the `->branches` counter for each verifier state in a chain of parent verifier states. When the `->branches` counter reaches zero the verifier state becomes a valid entry in a set of cached verifier states.

Each entry of the verifier states cache is post-processed by a function `clean_live_states()`. This function marks all registers and stack slots without `REG_LIVE_READ{32,64}` marks as `NOT_INIT` or `STACK_INVALID`. Registers/stack slots marked in this way are ignored in function `stacksafe()` called from `states_equal()` when a state cache entry is considered for equivalence with a current state.

Now it is possible to explain how the example from the beginning of the section works:

```
0: call bpf_get_prandom_u32()
1: r1 = 0
2: if r0 == 0 goto +1
3: r0 = 1
--- checkpoint[0] ---
4: r0 = r1
5: exit
```

- At instruction #2 branching point is reached and state `{ r0 == 0, r1 == 0, pc == 4 }` is pushed to states processing queue (pc stands for program counter).
- At instruction #4:
  - `checkpoint[0]` states cache entry is created: `{ r0 == 1, r1 == 0, pc == 4 }`;
  - `checkpoint[0].r0` is marked as written;
  - `checkpoint[0].r1` is marked as read;
- At instruction #5 exit is reached and `checkpoint[0]` can now be processed by `clean_live_states()`. After this processing `checkpoint[0].r0` has a read mark and all other registers and stack slots are marked as `NOT_INIT` or `STACK_INVALID`
- The state `{ r0 == 0, r1 == 0, pc == 4 }` is popped from the states queue and is compared against a cached state `{ r1 == 0, pc == 4 }`, the states are considered equivalent.

Read marks propagation for cache hits

Another point is the handling of read marks when a previously verified state is found in the states cache. Upon cache hit verifier must behave in the same way as if the current state was verified to the program exit. This means that all read marks, present on registers and stack slots of the cached state, must be propagated over the parentage chain of the current state. Example below shows why this is important. Function `propagate_liveness()` handles this case.

Consider the following state parentage chain (S is a starting state, A-E are derived states, -> arrows show which state is derived from which):

```
               r1 read
       <-------------                A[r1] == 0
                                     C[r1] == 0
  S ---> A ---> B ---> exit          E[r1] == 1
  |
  ` ---> C ---> D
  |
  ` ---> E      ^
                |___   suppose all these
        ^           states are at insn #Y
        |
  suppose all these
states are at insn #X
```

- Chain of states `S -> A -> B -> exit` is verified first.
- While `B -> exit` is verified, register `r1` is read and this read mark is propagated up to state `A`.
- When chain of states `C -> D` is verified the state `D` turns out to be equivalent to state `B`.
- The read mark for `r1` has to be propagated to state `C`, otherwise state `C` might get mistakenly marked as equivalent to state `E` even though values for register `r1` differ between `C` and `E`.

1.4 Understanding eBPF verifier messages

The following are few examples of invalid eBPF programs and verifier error messages as seen in the log:

Program with unreachable instructions:

```
static struct bpf_insn prog[] = {
BPF_EXIT_INSN(),
BPF_EXIT_INSN(),
};
```

Error:

```
unreachable insn 1
```

Program that reads uninitialized register:

```
BPF_MOV64_REG(BPF_REG_0, BPF_REG_2),
BPF_EXIT_INSN(),
```

Error:

```
0: (bf) r0 = r2
R2 !read_ok
```

Program that doesn’t initialize R0 before exiting:

```
BPF_MOV64_REG(BPF_REG_2, BPF_REG_1),
BPF_EXIT_INSN(),
```

Error:

```
0: (bf) r2 = r1
1: (95) exit
R0 !read_ok
```

Program that accesses stack out of bounds:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, 8, 0),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *)(r10 +8) = 0
invalid stack off=8 size=8
```

Program that doesn’t initialize stack before passing its address into function:

```
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_EXIT_INSN(),
```

Error:

```
0: (bf) r2 = r10
1: (07) r2 += -8
2: (b7) r1 = 0x0
3: (85) call 1
invalid indirect read from stack off -8+0 size 8
```

Program that uses invalid map_fd=0 while calling to map_lookup_elem() function:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 0x0
4: (85) call 1
fd 0 is not pointing to valid bpf_map
```

Program that doesn't check return value of map_lookup_elem() before accessing map element:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 0),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 0x0
4: (85) call 1
5: (7a) *(u64 *)(r0 +0) = 0
R0 invalid mem access 'map_value_or_null'
```

Program that correctly checks map_lookup_elem() returned value for NULL, but accesses the memory with incorrect alignment:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 1),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 4, 0),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 1
4: (85) call 1
5: (15) if r0 == 0x0 goto pc+1
 R0=map_ptr R10=fp
```

```
6: (7a) *(u64 *)(r0 +4) = 0
misaligned access off 4 size 8
```

Program that correctly checks map_lookup_elem() returned value for NULL and accesses memory with correct alignment in one side of 'if' branch, but fails to do so in the other side of 'if' branch:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 0),
BPF_EXIT_INSN(),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 1),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 1
4: (85) call 1
5: (15) if r0 == 0x0 goto pc+2
 R0=map_ptr R10=fp
6: (7a) *(u64 *)(r0 +0) = 0
7: (95) exit

from 5 to 8: R0=imm0 R10=fp
8: (7a) *(u64 *)(r0 +0) = 1
R0 invalid mem access 'imm'
```

Program that performs a socket lookup then sets the pointer to NULL without checking it:

```
BPF_MOV64_IMM(BPF_REG_2, 0),
BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_2, -8),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_MOV64_IMM(BPF_REG_3, 4),
BPF_MOV64_IMM(BPF_REG_4, 0),
BPF_MOV64_IMM(BPF_REG_5, 0),
BPF_EMIT_CALL(BPF_FUNC_sk_lookup_tcp),
BPF_MOV64_IMM(BPF_REG_0, 0),
BPF_EXIT_INSN(),
```

Error:

```
0: (b7) r2 = 0
1: (63) *(u32 *)(r10 -8) = r2
2: (bf) r2 = r10
```

```
3: (07) r2 += -8
4: (b7) r3 = 4
5: (b7) r4 = 0
6: (b7) r5 = 0
7: (85) call bpf_sk_lookup_tcp#65
8: (b7) r0 = 0
9: (95) exit
Unreleased reference id=1, alloc_insn=7
```

Program that performs a socket lookup but does not NULL-check the returned value:

```
BPF_MOV64_IMM(BPF_REG_2, 0),
BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_2, -8),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_MOV64_IMM(BPF_REG_3, 4),
BPF_MOV64_IMM(BPF_REG_4, 0),
BPF_MOV64_IMM(BPF_REG_5, 0),
BPF_EMIT_CALL(BPF_FUNC_sk_lookup_tcp),
BPF_EXIT_INSN(),
```

Error:

```
0: (b7) r2 = 0
1: (63) *(u32 *)(r10 -8) = r2
2: (bf) r2 = r10
3: (07) r2 += -8
4: (b7) r3 = 4
5: (b7) r4 = 0
6: (b7) r5 = 0
7: (85) call bpf_sk_lookup_tcp#65
8: (95) exit
Unreleased reference id=1, alloc_insn=7
```

CHAPTER

TWO

LIBBPF

If you are looking to develop BPF applications using the libbpf library, this directory contains important documentation that you should read.

To get started, it is recommended to begin with the *libbpf Overview* document, which provides a high-level understanding of the libbpf APIs and their usage. This will give you a solid foundation to start exploring and utilizing the various features of libbpf to develop your BPF applications.

2.1 libbpf Overview

libbpf is a C-based library containing a BPF loader that takes compiled BPF object files and prepares and loads them into the Linux kernel. libbpf takes the heavy lifting of loading, verifying, and attaching BPF programs to various kernel hooks, allowing BPF application developers to focus only on BPF program correctness and performance.

The following are the high-level features supported by libbpf:

- Provides high-level and low-level APIs for user space programs to interact with BPF programs. The low-level APIs wrap all the bpf system call functionality, which is useful when users need more fine-grained control over the interactions between user space and BPF programs.
- Provides overall support for the BPF object skeleton generated by bpftool. The skeleton file simplifies the process for the user space programs to access global variables and work with BPF programs.
- Provides BPF-side APIS, including BPF helper definitions, BPF maps support, and tracing helpers, allowing developers to simplify BPF code writing.
- Supports BPF CO-RE mechanism, enabling BPF developers to write portable BPF programs that can be compiled once and run across different kernel versions.

This document will delve into the above concepts in detail, providing a deeper understanding of the capabilities and advantages of libbpf and how it can help you develop BPF applications efficiently.

2.1.1 BPF App Lifecycle and libbpf APIs

A BPF application consists of one or more BPF programs (either cooperating or completely independent), BPF maps, and global variables. The global variables are shared between all BPF programs, which allows them to cooperate on a common set of data. libbpf provides APIs that user space programs can use to manipulate the BPF programs by triggering different phases of a BPF application lifecycle.

The following section provides a brief overview of each phase in the BPF life cycle:

- **Open phase**: In this phase, libbpf parses the BPF object file and discovers BPF maps, BPF programs, and global variables. After a BPF app is opened, user space apps can make additional adjustments (setting BPF program types, if necessary; pre-setting initial values for global variables, etc.) before all the entities are created and loaded.
- **Load phase**: In the load phase, libbpf creates BPF maps, resolves various relocations, and verifies and loads BPF programs into the kernel. At this point, libbpf validates all the parts of a BPF application and loads the BPF program into the kernel, but no BPF program has yet been executed. After the load phase, it's possible to set up the initial BPF map state without racing with the BPF program code execution.
- **Attachment phase**: In this phase, libbpf attaches BPF programs to various BPF hook points (e.g., tracepoints, kprobes, cgroup hooks, network packet processing pipeline, etc.). During this phase, BPF programs perform useful work such as processing packets, or updating BPF maps and global variables that can be read from user space.
- **Tear down phase**: In the tear down phase, libbpf detaches BPF programs and unloads them from the kernel. BPF maps are destroyed, and all the resources used by the BPF app are freed.

2.1.2 BPF Object Skeleton File

BPF skeleton is an alternative interface to libbpf APIs for working with BPF objects. Skeleton code abstract away generic libbpf APIs to significantly simplify code for manipulating BPF programs from user space. Skeleton code includes a bytecode representation of the BPF object file, simplifying the process of distributing your BPF code. With BPF bytecode embedded, there are no extra files to deploy along with your application binary.

You can generate the skeleton header file (`.skel.h`) for a specific object file by passing the BPF object to the bpftool. The generated BPF skeleton provides the following custom functions that correspond to the BPF lifecycle, each of them prefixed with the specific object name:

- `<name>__open()` – creates and opens BPF application (`<name>` stands for the specific bpf object name)
- `<name>__load()` – instantiates, loads,and verifies BPF application parts
- `<name>__attach()` – attaches all auto-attachable BPF programs (it's optional, you can have more control by using libbpf APIs directly)
- `<name>__destroy()` – detaches all BPF programs and frees up all used resources

Using the skeleton code is the recommended way to work with bpf programs. Keep in mind, BPF skeleton provides access to the underlying BPF object, so whatever was possible to do with generic libbpf APIs is still possible even when the BPF skeleton is used. It's an additive convenience feature, with no syscalls, and no cumbersome code.

Other Advantages of Using Skeleton File

- BPF skeleton provides an interface for user space programs to work with BPF global variables. The skeleton code memory maps global variables as a struct into user space. The struct interface allows user space programs to initialize BPF programs before the BPF load phase and fetch and update data from user space afterward.
- The `skel.h` file reflects the object file structure by listing out the available maps, programs, etc. BPF skeleton provides direct access to all the BPF maps and BPF programs as struct fields. This eliminates the need for string-based lookups with `bpf_object_find_map_by_name()` and `bpf_object_find_program_by_name()` APIs, reducing errors due to BPF source code and user-space code getting out of sync.
- The embedded bytecode representation of the object file ensures that the skeleton and the BPF object file are always in sync.

2.1.3 BPF Helpers

libbpf provides BPF-side APIs that BPF programs can use to interact with the system. The BPF helpers definition allows developers to use them in BPF code as any other plain C function. For example, there are helper functions to print debugging messages, get the time since the system was booted, interact with BPF maps, manipulate network packets, etc.

For a complete description of what the helpers do, the arguments they take, and the return value, see the bpf-helpers man page.

2.1.4 BPF CO-RE (Compile Once – Run Everywhere)

BPF programs work in the kernel space and have access to kernel memory and data structures. One limitation that BPF applications come across is the lack of portability across different kernel versions and configurations. BCC is one of the solutions for BPF portability. However, it comes with runtime overhead and a large binary size from embedding the compiler with the application.

libbpf steps up the BPF program portability by supporting the BPF CO-RE concept. BPF CO-RE brings together BTF type information, libbpf, and the compiler to produce a single executable binary that you can run on multiple kernel versions and configurations.

To make BPF programs portable libbpf relies on the BTF type information of the running kernel. Kernel also exposes this self-describing authoritative BTF information through `sysfs` at `/sys/kernel/btf/vmlinux`.

You can generate the BTF information for the running kernel with the following command:

```
$ bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

The command generates a `vmlinux.h` header file with all kernel types (*BTF types*) that the running kernel uses. Including `vmlinux.h` in your BPF program eliminates dependency on system-wide kernel headers.

libbpf enables portability of BPF programs by looking at the BPF program's recorded BTF type and relocation information and matching them to BTF information (vmlinux) provided by the running kernel. libbpf then resolves and matches all the types and fields, and updates necessary offsets and other relocatable data to ensure that BPF program's logic functions correctly for a

specific kernel on the host. BPF CO-RE concept thus eliminates overhead associated with BPF development and allows developers to write portable BPF applications without modifications and runtime source code compilation on the target machine.

The following code snippet shows how to read the parent field of a kernel `task_struct` using BPF CO-RE and libbf. The basic helper to read a field in a CO-RE relocatable manner is `bpf_core_read(dst, sz, src)`, which will read `sz` bytes from the field referenced by `src` into the memory pointed to by `dst`.

```
//...
struct task_struct *task = (void *)bpf_get_current_task();
struct task_struct *parent_task;
int err;

err = bpf_core_read(&parent_task, sizeof(void *), &task->parent);
if (err) {
  /* handle error */
}

/* parent_task contains the value of task->parent pointer */
```

In the code snippet, we first get a pointer to the current `task_struct` using `bpf_get_current_task()`. We then use `bpf_core_read()` to read the parent field of task struct into the `parent_task` variable. `bpf_core_read()` is just like `bpf_probe_read_kernel()` BPF helper, except it records information about the field that should be relocated on the target kernel. i.e, if the `parent` field gets shifted to a different offset within `struct task_struct` due to some new field added in front of it, libbpf will automatically adjust the actual offset to the proper value.

2.1.5 Getting Started with libbpf

Check out the libbpf-bootstrap repository with simple examples of using libbpf to build various BPF applications.

See also libbpf API documentation.

2.1.6 libbpf and Rust

If you are building BPF applications in Rust, it is recommended to use the Libbpf-rs library instead of bindgen bindings directly to libbpf. Libbpf-rs wraps libbpf functionality in Rust-idiomatic interfaces and provides libbpf-cargo plugin to handle BPF code compilation and skeleton generation. Using Libbpf-rs will make building user space part of the BPF application easier. Note that the BPF program themselves must still be written in plain C.

2.1.7 Additional Documentation

- Program types and ELF Sections
- API naming convention
- Building libbpf
- API documentation Convention

2.2 Program Types and ELF Sections

The table below lists the program types, their attach types where relevant and the ELF section names supported by libbpf for them. The ELF section names follow these rules:

- `type` is an exact match, e.g. `SEC("socket")`
- `type+` means it can be either exact `SEC("type")` or well-formed `SEC("type/extras")` with a '/' separator between `type` and `extras`.

When `extras` are specified, they provide details of how to auto-attach the BPF program. The format of `extras` depends on the program type, e.g. `SEC("tracepoint/<category>/<name>")` for tracepoints or `SEC("usdt/<path>:<provider>:<name>")` for USDT probes. The extras are described in more detail in the footnotes.

Program Type	Attach Type	ELF Section Name	Sleepable
BPF_PROG_TYPE_CGROU	BPF_CGROUP_DEVICE	cgroup/dev	
BPF_PROG_TYPE_CGROU		cgroup/skb	
	BPF_CGROUP_INET_EGR	cgroup_skb/egress	
	BPF_CGROUP_INET_ING	cgroup_skb/ingress	
BPF_PROG_TYPE_CGROU	BPF_CGROUP_GETSOCKO	cgroup/getsockopt	
	BPF_CGROUP_SETSOCKO	cgroup/setsockopt	
BPF_PROG_TYPE_CGROU	BPF_CGROUP_INET4_BI	cgroup/bind4	
	BPF_CGROUP_INET4_CO	cgroup/connect4	
	BPF_CGROUP_INET4_GE	cgroup/ getpeername4	
	BPF_CGROUP_INET4_GE	cgroup/ getsockname4	
	BPF_CGROUP_INET6_BI	cgroup/bind6	
	BPF_CGROUP_INET6_CO	cgroup/connect6	
	BPF_CGROUP_INET6_GE	cgroup/ getpeername6	

continues on next page

Table 1 – continued from previous page

Program Type	Attach Type	ELF Section Name	Sleepable
	BPF_CGROUP_INET6_GE	cgroup/ getsockname6	
	BPF_CGROUP_UDP4_REC	cgroup/recvmsg4	
	BPF_CGROUP_UDP4_SEN	cgroup/sendmsg4	
	BPF_CGROUP_UDP6_REC	cgroup/recvmsg6	
	BPF_CGROUP_UDP6_SEN	cgroup/sendmsg6	
BPF_PROG_TYPE_CGROU	BPF_CGROUP_INET4_PO	cgroup/post_bind4	
	BPF_CGROUP_INET6_PO	cgroup/post_bind6	
	BPF_CGROUP_INET_SOC	cgroup/sock_create	
		cgroup/sock	
	BPF_CGROUP_INET_SOC	cgroup/ sock_release	
BPF_PROG_TYPE_CGROU	BPF_CGROUP_SYSCTL	cgroup/sysctl	
BPF_PROG_TYPE_EXT		freplace+[1]	
BPF_PROG_TYPE_FLOW_	BPF_FLOW_DISSECTOR	flow_dissector	
BPF_PROG_TYPE_KPROBE		kprobe+[2]	
		kretprobe+[Page 25, 2]	
		ksyscall+[3]	
		kretsyscall+[Page 25, 3]	
		uprobe+[4]	
		uprobe.s+[Page 25, 4]	Yes
		uretprobe+[Page 25, 4]	
		uretprobe. s+[Page 25, 4]	Yes
		usdt+[5]	
	BPF_TRACE_KPROBE_MU	kprobe.multi+[6]	
		kretprobe. multi+[Page 25, 6]	
BPF_PROG_TYPE_LIRC_	BPF_LIRC_MODE2	lirc_mode2	
BPF_PROG_TYPE_LSM	BPF_LSM_CGROUP	lsm_cgroup+	
	BPF_LSM_MAC	lsm+[7]	
		lsm.s+[Page 25, 7]	Yes
BPF_PROG_TYPE_LWT_I		lwt_in	
BPF_PROG_TYPE_LWT_O		lwt_out	
BPF_PROG_TYPE_LWT_S		lwt_seg6local	
BPF_PROG_TYPE_LWT_X		lwt_xmit	
BPF_PROG_TYPE_PERF_		perf_event	
BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE		raw_tp.w+[8]	
		raw_tracepoint.w+	
BPF_PROG_TYPE_RAW_TRACEPOINT		raw_tp+[Page 25, 8]	
		raw_tracepoint+	
BPF_PROG_TYPE_SCHED		action	
BPF_PROG_TYPE_SCHED_CLS		classifier	
		tc	
BPF_PROG_TYPE_SK_LO	BPF_SK_LOOKUP	sk_lookup	
BPF_PROG_TYPE_SK_MS	BPF_SK_MSG_VERDICT	sk_msg	

continues on next page

Table 1 – continued from previous page

Program Type	Attach Type	ELF Section Name	Sleepable
BPF_PROG_TYPE_SK_RE	BPF_SK_REUSEPORT_SE	sk_reuseport/ migrate	
	BPF_SK_REUSEPORT_SE	sk_reuseport	
BPF_PROG_TYPE_SK_SK		sk_skb	
	BPF_SK_SKB_STREAM_P	sk_skb/ stream_parser	
	BPF_SK_SKB_STREAM_V	sk_skb/ stream_verdict	
BPF_PROG_TYPE_SOCKE		socket	
BPF_PROG_TYPE_SOCK_	BPF_CGROUP_SOCK_OPS	sockops	
BPF_PROG_TYPE_STRUC		struct_ops+	
BPF_PROG_TYPE_SYSCA		syscall	Yes
BPF_PROG_TYPE_TRACEPOINT		tp+[9]	
		tracepoint+[Page 25, 9]	
BPF_PROG_TYPE_TRACING	BPF_MODIFY_RETURN	fmod_ret+[1]	
		fmod_ret.s+[1]	Yes
	BPF_TRACE_FENTRY	fentry+[1]	
		fentry.s+[1]	Yes
	BPF_TRACE_FEXIT	fexit+[1]	
		fexit.s+[1]	Yes
	BPF_TRACE_ITER	iter+[10]	
		iter.s+[10]	Yes
	BPF_TRACE_RAW_TP	tp_btf+[1]	
BPF_PROG_TYPE_XDP	BPF_XDP_CPUMAP	xdp.frags/cpumap	
		xdp/cpumap	
	BPF_XDP_DEVMAP	xdp.frags/devmap	
		xdp/devmap	
	BPF_XDP	xdp.frags	
		xdp	

[1] The fentry attach format is fentry[.s]/<function>.
[2] The kprobe attach format is kprobe/<function>[+<offset>]. Valid characters for function are a-zA-Z0-9_. and offset must be a valid non-negative integer.
[3] The ksyscall attach format is ksyscall/<syscall>.
[4] The uprobe attach format is uprobe[.s]/<path>:<function>[+<offset>].
[5] The usdt attach format is usdt/<path>:<provider>:<name>.
[6] The kprobe.multi attach format is kprobe.multi/<pattern> where pattern supports * and ? wildcards. Valid characters for pattern are a-zA-Z0-9_.*?.
[7] The lsm attachment format is lsm[.s]/<hook>.
[8] The raw_tp attach format is raw_tracepoint[.w]/<tracepoint>.
[9] The tracepoint attach format is tracepoint/<category>/<name>.
[10] The iter attach format is iter[.s]/<struct-name>.

2.3 API naming convention

libbpf API provides access to a few logically separated groups of functions and types. Every group has its own naming convention described here. It's recommended to follow these conventions whenever a new function or type is added to keep libbpf API clean and consistent.

All types and functions provided by libbpf API should have one of the following prefixes: `bpf_`, `btf_`, `libbpf_`, `btf_dump_`, `ring_buffer_`, `perf_buffer_`.

2.3.1 System call wrappers

System call wrappers are simple wrappers for commands supported by sys_bpf system call. These wrappers should go to `bpf.h` header file and map one to one to corresponding commands.

For example `bpf_map_lookup_elem` wraps `BPF_MAP_LOOKUP_ELEM` command of sys_bpf, `bpf_prog_attach` wraps `BPF_PROG_ATTACH`, etc.

2.3.2 Objects

Another class of types and functions provided by libbpf API is "objects" and functions to work with them. Objects are high-level abstractions such as BPF program or BPF map. They're represented by corresponding structures such as `struct bpf_object`, `struct bpf_program`, `struct bpf_map`, etc.

Structures are forward declared and access to their fields should be provided via corresponding getters and setters rather than directly.

These objects are associated with corresponding parts of ELF object that contains compiled BPF programs.

For example `struct bpf_object` represents ELF object itself created from an ELF file or from a buffer, `struct bpf_program` represents a program in ELF object and `struct bpf_map` is a map.

Functions that work with an object have names built from object name, double underscore and part that describes function purpose.

For example `bpf_object__open` consists of the name of corresponding object, `bpf_object`, double underscore and `open` that defines the purpose of the function to open ELF file and create `bpf_object` from it.

All objects and corresponding functions other than BTF related should go to `libbpf.h`. BTF types and functions should go to `btf.h`.

2.3.3 Auxiliary functions

Auxiliary functions and types that don't fit well in any of categories described above should have `libbpf_` prefix, e.g. `libbpf_get_error` or `libbpf_prog_type_by_name`.

2.3.4 ABI

libbpf can be both linked statically or used as DSO. To avoid possible conflicts with other libraries an application is linked with, all non-static libbpf symbols should have one of the prefixes mentioned in API documentation above. See API naming convention to choose the right name for a new symbol.

2.3.5 Symbol visibility

libbpf follow the model when all global symbols have visibility "hidden" by default and to make a symbol visible it has to be explicitly attributed with `LIBBPF_API` macro. For example:

```
LIBBPF_API int bpf_prog_get_fd_by_id(__u32 id);
```

This prevents from accidentally exporting a symbol, that is not supposed to be a part of ABI what, in turn, improves both libbpf developer- and user-experiences.

2.3.6 ABI versioning

To make future ABI extensions possible libbpf ABI is versioned. Versioning is implemented by `libbpf.map` version script that is passed to linker.

Version name is `LIBBPF_` prefix + three-component numeric version, starting from `0.0.1`.

Every time ABI is being changed, e.g. because a new symbol is added or semantic of existing symbol is changed, ABI version should be bumped. This bump in ABI version is at most once per kernel development cycle.

For example, if current state of `libbpf.map` is:

```
LIBBPF_0.0.1 {
        global:
                bpf_func_a;
                bpf_func_b;
        local:
                \*;
};
```

, and a new symbol `bpf_func_c` is being introduced, then `libbpf.map` should be changed like this:

```
LIBBPF_0.0.1 {
        global:
                bpf_func_a;
                bpf_func_b;
        local:
```

```
                \*;
};
LIBBPF_0.0.2 {
        global:
                bpf_func_c;
} LIBBPF_0.0.1;
```

, where new version `LIBBPF_0.0.2` depends on the previous `LIBBPF_0.0.1`.

Format of version script and ways to handle ABI changes, including incompatible ones, described in details in [1].

2.3.7 Stand-alone build

Under https://github.com/libbpf/libbpf there is a (semi-)automated mirror of the mainline's version of libbpf for a stand-alone build.

However, all changes to libbpf's code base must be upstreamed through the mainline kernel tree.

2.4 API documentation convention

The libbpf API is documented via comments above definitions in header files. These comments can be rendered by doxygen and sphinx for well organized html output. This section describes the convention in which these comments should be formatted.

Here is an example from btf.h:

```
/**
 * @brief **btf__new()** creates a new instance of a BTF object from the raw
 * bytes of an ELF's BTF section
 * @param data raw bytes
 * @param size number of bytes passed in `data`
 * @return new BTF object instance which has to be eventually freed with
 * **btf__free()**
 *
 * On error, error-code-encoded-as-pointer is returned, not a NULL. To extract
 * error code from such a pointer `libbpf_get_error()` should be used. If
 * `libbpf_set_strict_mode(LIBBPF_STRICT_CLEAN_PTRS)` is enabled, NULL is
 * returned on error instead. In both cases thread-local `errno` variable is
 * always set to error code as well.
 */
```

The comment must start with a block comment of the form '/**'.

The documentation always starts with a @brief directive. This line is a short description about this API. It starts with the name of the API, denoted in bold like so: **api_name**. Please include an open and close parenthesis if this is a function. Follow with the short description of the API. A longer form description can be added below the last directive, at the bottom of the comment.

Parameters are denoted with the @param directive, there should be one for each parameter. If this is a function with a non-void return, use the @return directive to document it.

2.4.1 License

libbpf is dual-licensed under LGPL 2.1 and BSD 2-Clause.

2.4.2 Links

[1] **https://www.akkadia.org/drepper/dsohowto.pdf**
(Chapter 3. Maintaining APIs and ABIs).

2.5 Building libbpf

libelf and zlib are internal dependencies of libbpf and thus are required to link against and must be installed on the system for applications to work. pkg-config is used by default to find libelf, and the program called can be overridden with PKG_CONFIG.

If using pkg-config at build time is not desired, it can be disabled by setting NO_PKG_CONFIG=1 when calling make.

To build both static libbpf.a and shared libbpf.so:

```
$ cd src
$ make
```

To build only static libbpf.a library in directory build/ and install them together with libbpf headers in a staging directory root/:

```
$ cd src
$ mkdir build root
$ BUILD_STATIC_ONLY=y OBJDIR=build DESTDIR=root make install
```

To build both static libbpf.a and shared libbpf.so against a custom libelf dependency installed in /build/root/ and install them together with libbpf headers in a build directory /build/root/:

```
$ cd src
$ PKG_CONFIG_PATH=/build/root/lib64/pkgconfig DESTDIR=/build/root make
```

All general BPF questions, including kernel functionality, libbpf APIs and their application, should be sent to bpf@vger.kernel.org mailing list. You can subscribe to the mailing list search its archive. Please search the archive before asking new questions. It may be that this was already addressed or answered before.

CHAPTER

THREE

BPF STANDARDIZATION

This directory contains documents that are being iterated on as part of the BPF standardization effort with the IETF. See the IETF BPF Working Group page for the working group charter, documents, and more.

Contents

3.1 1 BPF Instruction Set Specification, v1.0

This document specifies version 1.0 of the BPF instruction set.

3.1.1 1.1 Documentation conventions

For brevity and consistency, this document refers to families of types using a shorthand syntax and refers to several expository, mnemonic functions when describing the semantics of instructions. The range of valid values for those types and the semantics of those functions are defined in the following subsections.

1.1.1 Types

This document refers to integer types with the notation *SN* to specify a type's signedness (*S*) and bit width (*N*), respectively.

Table 1: Meaning of signedness notation.

S	Meaning
u	unsigned
s	signed

Table 2: Meaning of bit-width notation.

N	Bit width
8	8 bits
16	16 bits
32	32 bits
64	64 bits
128	128 bits

For example, *u32* is a type whose valid values are all the 32-bit unsigned numbers and *s16* is a types whose valid values are all the 16-bit signed numbers.

1.1.2 Functions

- *htobe16*: Takes an unsigned 16-bit number in host-endian format and returns the equivalent number as an unsigned 16-bit number in big-endian format.
- *htobe32*: Takes an unsigned 32-bit number in host-endian format and returns the equivalent number as an unsigned 32-bit number in big-endian format.
- *htobe64*: Takes an unsigned 64-bit number in host-endian format and returns the equivalent number as an unsigned 64-bit number in big-endian format.
- *htole16*: Takes an unsigned 16-bit number in host-endian format and returns the equivalent number as an unsigned 16-bit number in little-endian format.

- *htole32*: Takes an unsigned 32-bit number in host-endian format and returns the equivalent number as an unsigned 32-bit number in little-endian format.
- *htole64*: Takes an unsigned 64-bit number in host-endian format and returns the equivalent number as an unsigned 64-bit number in little-endian format.
- *bswap16*: Takes an unsigned 16-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.
- *bswap32*: Takes an unsigned 32-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.
- *bswap64*: Takes an unsigned 64-bit number in either big- or little-endian format and returns the equivalent number with the same bit width but opposite endianness.

1.1.3 Definitions

Sign Extend
To *sign extend an* X *-bit number, A, to a* Y *-bit number, B ,* means to

1. Copy all X bits from *A* to the lower X bits of *B*.
2. Set the value of the remaining Y - X bits of *B* to the value of the most-significant bit of *A*.

Example

Sign extend an 8-bit number A to a 16-bit number B on a big-endian platform:

```
A:          10000110
B: 11111111 10000110
```

3.1.2 1.2 Instruction encoding

BPF has two instruction encodings:

- the basic instruction encoding, which uses 64 bits to encode an instruction
- the wide instruction encoding, which appends a second 64-bit immediate (i.e., constant) value after the basic instruction for a total of 128 bits.

The fields conforming an encoded basic instruction are stored in the following order:

```
opcode:8 src_reg:4 dst_reg:4 offset:16 imm:32 // In little-endian BPF.
opcode:8 dst_reg:4 src_reg:4 offset:16 imm:32 // In big-endian BPF.
```

imm
signed integer immediate value

offset
signed integer offset used with pointer arithmetic

src_reg
the source register number (0-10), except where otherwise specified (*64-bit immediate instructions* reuse this field for other purposes)

dst_reg
destination register number (0-10)

opcode
operation to perform

Note that the contents of multi-byte fields ('imm' and 'offset') are stored using big-endian byte ordering in big-endian BPF and little-endian byte ordering in little-endian BPF.

For example:

```
opcode                  offset imm          assembly
       src_reg dst_reg
07     0       1        00 00  44 33 22 11  r1 += 0x11223344 // little
       dst_reg src_reg
07     1       0        00 00  11 22 33 44  r1 += 0x11223344 // big
```

Note that most instructions do not use all of the fields. Unused fields shall be cleared to zero.

As discussed below in *64-bit immediate instructions*, a 64-bit immediate instruction uses a 64-bit immediate value that is constructed as follows. The 64 bits following the basic instruction contain a pseudo instruction using the same format but with opcode, dst_reg, src_reg, and offset all set to zero, and imm containing the high 32 bits of the immediate value.

This is depicted in the following figure:

```
      basic_instruction
.------------------------------.
|                              |
code:8 regs:8 offset:16 imm:32 unused:32 imm:32
                                |              |
                                '--------------'
                               pseudo instruction
```

Thus the 64-bit immediate value is constructed as follows:

imm64 = (next_imm << 32) | imm

where 'next_imm' refers to the imm value of the pseudo instruction following the basic instruction. The unused bytes in the pseudo instruction are reserved and shall be cleared to zero.

1.2.1 Instruction classes

The three LSB bits of the 'opcode' field store the instruction class:

class	value	description	reference
BPF_LD	0x00	non-standard load operations	*Load and store instructions*
BPF_LDX	0x01	load into register operations	*Load and store instructions*
BPF_ST	0x02	store from immediate operations	*Load and store instructions*
BPF_STX	0x03	store from register operations	*Load and store instructions*
BPF_ALU	0x04	32-bit arithmetic operations	*Arithmetic and jump instructions*
BPF_JMP	0x05	64-bit jump operations	*Arithmetic and jump instructions*
BPF_JMP32	0x06	32-bit jump operations	*Arithmetic and jump instructions*
BPF_ALU64	0x07	64-bit arithmetic operations	*Arithmetic and jump instructions*

3.1.3 1.3 Arithmetic and jump instructions

For arithmetic and jump instructions (BPF_ALU, BPF_ALU64, BPF_JMP and BPF_JMP32), the 8-bit 'opcode' field is divided into three parts:

4 bits (MSB)	1 bit	3 bits (LSB)
code	source	instruction class

code
the operation code, whose meaning varies by instruction class

source
the source operand location, which unless otherwise specified is one of:

source	value	description
BPF_K	0x00	use 32-bit 'imm' value as source operand
BPF_X	0x08	use 'src_reg' register value as source operand

instruction class
the instruction class (see *Instruction classes*)

1.3.1 Arithmetic instructions

BPF_ALU uses 32-bit wide operands while BPF_ALU64 uses 64-bit wide operands for otherwise identical operations. The 'code' field encodes the operation as below, where 'src' and 'dst' refer to the values of the source and destination registers, respectively.

code	value	offset	description
BPF_ADD	0x00	0	dst += src
BPF_SUB	0x10	0	dst -= src
BPF_MUL	0x20	0	dst *= src
BPF_DIV	0x30	0	dst = (src != 0) ? (dst / src) : 0
BPF_SDIV	0x30	1	dst = (src != 0) ? (dst s/ src) : 0
BPF_OR	0x40	0	dst |= src
BPF_AND	0x50	0	dst &= src
BPF_LSH	0x60	0	dst <<= (src & mask)
BPF_RSH	0x70	0	dst >>= (src & mask)
BPF_NEG	0x80	0	dst = -dst
BPF_MOD	0x90	0	dst = (src != 0) ? (dst % src) : dst
BPF_SMOD	0x90	1	dst = (src != 0) ? (dst s% src) : dst
BPF_XOR	0xa0	0	dst ^= src
BPF_MOV	0xb0	0	dst = src
BPF_MOVSX	0xb0	8/16/32	dst = (s8,s16,s32)src
BPF_ARSH	0xc0	0	*sign extending* dst >>= (src & mask)
BPF_END	0xd0	0	byte swap operations (see *Byte swap instructions* below)

Underflow and overflow are allowed during arithmetic operations, meaning the 64-bit or 32-bit value will wrap. If BPF program execution would result in division by zero, the destination

register is instead set to zero. If execution would result in modulo by zero, for BPF_ALU64 the value of the destination register is unchanged whereas for BPF_ALU the upper 32 bits of the destination register are zeroed.

BPF_ADD | BPF_X | BPF_ALU means:

```
dst = (u32) ((u32) dst + (u32) src)
```

where '(u32)' indicates that the upper 32 bits are zeroed.

BPF_ADD | BPF_X | BPF_ALU64 means:

```
dst = dst + src
```

BPF_XOR | BPF_K | BPF_ALU means:

```
dst = (u32) dst ^ (u32) imm32
```

BPF_XOR | BPF_K | BPF_ALU64 means:

```
dst = dst ^ imm32
```

Note that most instructions have instruction offset of 0. Only three instructions (BPF_SDIV, BPF_SMOD, BPF_MOVSX) have a non-zero offset.

The division and modulo operations support both unsigned and signed flavors.

For unsigned operations (BPF_DIV and BPF_MOD), for BPF_ALU, 'imm' is interpreted as a 32-bit unsigned value. For BPF_ALU64, 'imm' is first *sign extended* from 32 to 64 bits, and then interpreted as a 64-bit unsigned value.

For signed operations (BPF_SDIV and BPF_SMOD), for BPF_ALU, 'imm' is interpreted as a 32-bit signed value. For BPF_ALU64, 'imm' is first *sign extended* from 32 to 64 bits, and then interpreted as a 64-bit signed value.

The BPF_MOVSX instruction does a move operation with sign extension. BPF_ALU | BPF_MOVSX *sign extends* 8-bit and 16-bit operands into 32 bit operands, and zeroes the remaining upper 32 bits. BPF_ALU64 | BPF_MOVSX *sign extends* 8-bit, 16-bit, and 32-bit operands into 64 bit operands.

Shift operations use a mask of 0x3F (63) for 64-bit operations and 0x1F (31) for 32-bit operations.

1.3.2 Byte swap instructions

The byte swap instructions use instruction classes of BPF_ALU and BPF_ALU64 and a 4-bit 'code' field of BPF_END.

The byte swap instructions operate on the destination register only and do not use a separate source register or immediate value.

For BPF_ALU, the 1-bit source operand field in the opcode is used to select what byte order the operation converts from or to. For BPF_ALU64, the 1-bit source operand field in the opcode is reserved and must be set to 0.

class	source	value	description
BPF_ALU	BPF_TO_LE	0x00	convert between host byte order and little endian
BPF_ALU	BPF_TO_BE	0x08	convert between host byte order and big endian
BPF_ALU64	Reserved	0x00	do byte swap unconditionally

The ‘imm’ field encodes the width of the swap operations. The following widths are supported: 16, 32 and 64.

Examples:

`BPF_ALU | BPF_TO_LE | BPF_END` with imm = 16/32/64 means:

```
dst = htole16(dst)
dst = htole32(dst)
dst = htole64(dst)
```

`BPF_ALU | BPF_TO_BE | BPF_END` with imm = 16/32/64 means:

```
dst = htobe16(dst)
dst = htobe32(dst)
dst = htobe64(dst)
```

`BPF_ALU64 | BPF_TO_LE | BPF_END` with imm = 16/32/64 means:

```
dst = bswap16(dst)
dst = bswap32(dst)
dst = bswap64(dst)
```

1.3.3 Jump instructions

`BPF_JMP32` uses 32-bit wide operands while `BPF_JMP` uses 64-bit wide operands for otherwise identical operations. The ‘code’ field encodes the operation as below:

code	value	src	description	notes
BPF_JA	0x0	0x0	PC += offset	BPF_JMP class
BPF_JA	0x0	0x0	PC += imm	BPF_JMP32 class
BPF_JEQ	0x1	any	PC += offset if dst == src	
BPF_JGT	0x2	any	PC += offset if dst > src	unsigned
BPF_JGE	0x3	any	PC += offset if dst >= src	unsigned
BPF_JSET	0x4	any	PC += offset if dst & src	
BPF_JNE	0x5	any	PC += offset if dst != src	
BPF_JSGT	0x6	any	PC += offset if dst > src	signed
BPF_JSGE	0x7	any	PC += offset if dst >= src	signed
BPF_CALL	0x8	0x0	call helper function by address	see *Helper functions*
BPF_CALL	0x8	0x1	call PC += imm	see *Program-local functions*
BPF_CALL	0x8	0x2	call helper function by BTF ID	see *Helper functions*
BPF_EXIT	0x9	0x0	return	BPF_JMP only
BPF_JLT	0xa	any	PC += offset if dst < src	unsigned
BPF_JLE	0xb	any	PC += offset if dst <= src	unsigned
BPF_JSLT	0xc	any	PC += offset if dst < src	signed
BPF_JSLE	0xd	any	PC += offset if dst <= src	signed

The BPF program needs to store the return value into register R0 before doing a `BPF_EXIT`.

Example:

`BPF_JSGE | BPF_X | BPF_JMP32` (0x7e) means:

```
if (s32)dst s>= (s32)src goto +offset
```

where 's>=' indicates a signed '>=' comparison.

`BPF_JA | BPF_K | BPF_JMP32` (0x06) means:

```
gotol +imm
```

where 'imm' means the branch offset comes from insn 'imm' field.

Note that there are two flavors of `BPF_JA` instructions. The `BPF_JMP` class permits a 16-bit jump offset specified by the 'offset' field, whereas the `BPF_JMP32` class permits a 32-bit jump offset specified by the 'imm' field. A > 16-bit conditional jump may be converted to a < 16-bit conditional jump plus a 32-bit unconditional jump.

1.3.3.1 Helper functions

Helper functions are a concept whereby BPF programs can call into a set of function calls exposed by the underlying platform.

Historically, each helper function was identified by an address encoded in the imm field. The available helper functions may differ for each program type, but address values are unique across all program types.

Platforms that support the BPF Type Format (BTF) support identifying a helper function by a BTF ID encoded in the imm field, where the BTF ID identifies the helper name and type.

1.3.3.2 Program-local functions

Program-local functions are functions exposed by the same BPF program as the caller, and are referenced by offset from the call instruction, similar to `BPF_JA`. The offset is encoded in the imm field of the call instruction. A `BPF_EXIT` within the program-local function will return to the caller.

3.1.4 1.4 Load and store instructions

For load and store instructions (`BPF_LD`, `BPF_LDX`, `BPF_ST`, and `BPF_STX`), the 8-bit 'opcode' field is divided as:

3 bits (MSB)	2 bits	3 bits (LSB)
mode	size	instruction class

The mode modifier is one of:

mode modifier	value	description	reference
BPF_IMM	0x00	64-bit immediate instructions	*64-bit immediate instructions*
BPF_ABS	0x20	legacy BPF packet access (absolute)	*Legacy BPF Packet access instructions*
BPF_IND	0x40	legacy BPF packet access (indirect)	*Legacy BPF Packet access instructions*
BPF_MEM	0x60	regular load and store operations	*Regular load and store operations*
BPF_MEMSX	0x80	sign-extension load operations	*Sign-extension load operations*
BPF_ATOMIC	0xc0	atomic operations	*Atomic operations*

The size modifier is one of:

size modifier	value	description
BPF_W	0x00	word (4 bytes)
BPF_H	0x08	half word (2 bytes)
BPF_B	0x10	byte
BPF_DW	0x18	double word (8 bytes)

1.4.1 Regular load and store operations

The BPF_MEM mode modifier is used to encode regular load and store instructions that transfer data between a register and memory.

BPF_MEM | <size> | BPF_STX means:

```
*(size *) (dst + offset) = src
```

BPF_MEM | <size> | BPF_ST means:

```
*(size *) (dst + offset) = imm32
```

BPF_MEM | <size> | BPF_LDX means:

```
dst = *(unsigned size *) (src + offset)
```

Where size is one of: BPF_B, BPF_H, BPF_W, or BPF_DW and 'unsigned size' is one of u8, u16, u32 or u64.

1.4.2 Sign-extension load operations

The BPF_MEMSX mode modifier is used to encode *sign-extension* load instructions that transfer data between a register and memory.

BPF_MEMSX | <size> | BPF_LDX means:

```
dst = *(signed size *) (src + offset)
```

Where size is one of: BPF_B, BPF_H or BPF_W, and 'signed size' is one of s8, s16 or s32.

1.4.3 Atomic operations

Atomic operations are operations that operate on memory and can not be interrupted or corrupted by other access to the same memory region by other BPF programs or means outside of this specification.

All atomic operations supported by BPF are encoded as store operations that use the BPF_ATOMIC mode modifier as follows:

- BPF_ATOMIC | BPF_W | BPF_STX for 32-bit operations
- BPF_ATOMIC | BPF_DW | BPF_STX for 64-bit operations
- 8-bit and 16-bit wide atomic operations are not supported.

The 'imm' field is used to encode the actual atomic operation. Simple atomic operation use a subset of the values defined to encode arithmetic operations in the 'imm' field to encode the atomic operation:

imm	value	description
BPF_ADD	0x00	atomic add
BPF_OR	0x40	atomic or
BPF_AND	0x50	atomic and
BPF_XOR	0xa0	atomic xor

`BPF_ATOMIC | BPF_W | BPF_STX` with 'imm' = BPF_ADD means:

```
*(u32 *)(dst + offset) += src
```

`BPF_ATOMIC | BPF_DW | BPF_STX` with 'imm' = BPF ADD means:

```
*(u64 *)(dst + offset) += src
```

In addition to the simple atomic operations, there also is a modifier and two complex atomic operations:

imm	value	description
BPF_FETCH	0x01	modifier: return old value
BPF_XCHG	0xe0 | BPF_FETCH	atomic exchange
BPF_CMPXCHG	0xf0 | BPF_FETCH	atomic compare and exchange

The `BPF_FETCH` modifier is optional for simple atomic operations, and always set for the complex atomic operations. If the `BPF_FETCH` flag is set, then the operation also overwrites `src` with the value that was in memory before it was modified.

The `BPF_XCHG` operation atomically exchanges `src` with the value addressed by `dst + offset`.

The `BPF_CMPXCHG` operation atomically compares the value addressed by `dst + offset` with `R0`. If they match, the value addressed by `dst + offset` is replaced with `src`. In either case, the value that was at `dst + offset` before the operation is zero-extended and loaded back to `R0`.

1.4.4 64-bit immediate instructions

Instructions with the `BPF_IMM` 'mode' modifier use the wide instruction encoding defined in *Instruction encoding*, and use the 'src' field of the basic instruction to hold an opcode subtype.

The following table defines a set of `BPF_IMM | BPF_DW | BPF_LD` instructions with opcode subtypes in the 'src' field, using new terms such as "map" defined further below:

opcode construction	opcode	src	pseudocode	imm type	dst type
BPF_IMM | BPF_DW | BPF_LD	0x18	0x0	dst = imm64	integer	integer
BPF_IMM | BPF_DW | BPF_LD	0x18	0x1	dst = map_by_fd(imm)	map fd	map
BPF_IMM | BPF_DW | BPF_LD	0x18	0x2	dst = map_val(map_by_fd(imm)) + next_imm	map fd	data pointer
BPF_IMM | BPF_DW | BPF_LD	0x18	0x3	dst = var_addr(imm)	variable id	data pointer
BPF_IMM | BPF_DW | BPF_LD	0x18	0x4	dst = code_addr(imm)	integer	code pointer
BPF_IMM | BPF_DW | BPF_LD	0x18	0x5	dst = map_by_idx(imm)	map index	map
BPF_IMM | BPF_DW | BPF_LD	0x18	0x6	dst = map_val(map_by_idx(imm)) + next_imm	map index	data pointer

where

- map_by_fd(imm) means to convert a 32-bit file descriptor into an address of a map (see *Maps*)
- map_by_idx(imm) means to convert a 32-bit index into an address of a map
- map_val(map) gets the address of the first value in a given map
- var_addr(imm) gets the address of a platform variable (see *Platform Variables*) with a given id
- code_addr(imm) gets the address of the instruction at a specified relative offset in number of (64-bit) instructions
- the 'imm type' can be used by disassemblers for display
- the 'dst type' can be used for verification and JIT compilation purposes

1.4.4.1 Maps

Maps are shared memory regions accessible by BPF programs on some platforms. A map can have various semantics as defined in a separate document, and may or may not have a single contiguous memory region, but the 'map_val(map)' is currently only defined for maps that do have a single contiguous memory region.

Each map can have a file descriptor (fd) if supported by the platform, where 'map_by_fd(imm)' means to get the map with the specified file descriptor. Each BPF program can also be defined to use a set of maps associated with the program at load time, and 'map_by_idx(imm)' means to get the map with the given index in the set associated with the BPF program containing the instruction.

1.4.4.2 Platform Variables

Platform variables are memory regions, identified by integer ids, exposed by the runtime and accessible by BPF programs on some platforms. The 'var_addr(imm)' operation means to get the address of the memory region identified by the given id.

1.4.5 Legacy BPF Packet access instructions

BPF previously introduced special instructions for access to packet data that were carried over from classic BPF. However, these instructions are deprecated and should no longer be used.

Contents

3.2 1 BPF ABI Recommended Conventions and Guidelines v1.0

This is version 1.0 of an informational document containing recommended conventions and guidelines for producing portable BPF program binaries.

3.2.1 1.1 Registers and calling convention

BPF has 10 general purpose registers and a read-only frame pointer register, all of which are 64-bits wide.

The BPF calling convention is defined as:

- R0: return value from function calls, and exit value for BPF programs
- R1 - R5: arguments for function calls
- R6 - R9: callee saved registers that function calls will preserve
- R10: read-only frame pointer to access stack

R0 - R5 are scratch registers and BPF programs needs to spill/fill them if necessary across calls.

CHAPTER

FOUR

BPF TYPE FORMAT (BTF)

4.1 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*.

4.2 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       */
    __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.

4.2.1 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.

4.2.2 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 up to 32-bit values */
#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     */
#define BTF_KIND_ENUM64         19      /* Enumeration up to 64-bit values */
```

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, fwd, enum and enum64.
     */
    __u32 info;
    /* "size" is used by INT, ENUM, STRUCT, UNION and ENUM64.
     * "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 for unsigned, 1 for signed
- `info.kind`: BTF_KIND_ENUM
- `info.vlen`: number of enum values
- `size`: 1/2/4/8

`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

If the original enum value is signed and the size is less than 4, that value will be sign extended into 4 bytes. If the size is 8, the value will be truncated into 4 bytes.

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`: linkage information (BTF_FUNC_STATIC, BTF_FUNC_GLOBAL** or BTF_FUNC_EXTERN)
- `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).

Currently, only linkage values of BTF_FUNC_STATIC and BTF_FUNC_GLOBAL are supported in the kernel.

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.

2.2.18 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.

2.2.19 BTF_KIND_ENUM64

`struct btf_type` encoding requirement:

- `name_off`: 0 or offset to a valid C identifier
- `info.kind_flag`: 0 for unsigned, 1 for signed
- `info.kind`: BTF_KIND_ENUM64
- `info.vlen`: number of enum values
- `size`: 1/2/4/8

`btf_type` is followed by `info.vlen` number of `struct btf_enum64`.:

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

The `btf_enum64` encoding:

- `name_off`: offset to a valid C identifier
- `val_lo32`: lower 32-bit value for a 64-bit value
- `val_hi32`: high 32-bit value for a 64-bit value

If the original enum value is signed and the size is less than 8, that value will be sign extended into 8 bytes.

4.3 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:
    ......
    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.
```

4.3.1 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.

4.3.2 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.

4.3.3 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)
```

4.3.4 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.

4.3.5 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.

4.3.6 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.

4.3.7 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.

4.4 4. ELF File Format Interface

4.4.1 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*.

4.4.2 4.2 .BTF.ext section

The .BTF.ext section encodes func_info, line_info and CO-RE relocations 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;

    /* optional part of .BTF.ext header */
    __u32   core_relo_off;
    __u32   core_relo_len;
};
```

It is very similar to .BTF section. Instead of type/string section, it contains func_info, line_info and core_relo sub-sections. 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              /* __u32 value */
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              /* __u32 value */
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`).

The core_relo is organized as below.:

```
core_relo_rec_size              /* __u32 value */
btf_ext_info_sec for section #1 /* core_relo for section #1 */
btf_ext_info_sec for section #2 /* core_relo for section #2 */
```

`core_relo_rec_size` specifies the size of `bpf_core_relo` structure when .BTF.ext is generated. All `bpf_core_relo` structures within a single `btf_ext_info_sec` describe relocations applied to section named by `btf_ext_info_sec->sec_name_off`.

See *Documentation/bpf/llvm_reloc.rst* for more information on CO-RE relocations.

4.4.3 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.

4.5 5. Using BTF

4.5.1 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
      }
  }
]
```

4.5.2 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);
[...]
```

4.5.3 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
```

4.6 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
        .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
```

4.7 7. Testing

The kernel BPF selftest tools/testing/selftests/bpf/prog_tests/btf.c provides an extensive set of BTF-related tests.

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS (FAQ)

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

5.1 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.

5.1.1 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 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 zero-ing 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.

New BPF functionality is generally added through the use of kfuncs instead of new helpers. Kfuncs are not considered part of the stable API, and have their own lifecycle expectations as described in *3. kfunc lifecycle expectations*.

Q: Are tracepoints part of the stable ABI?

A: NO. Tracepoints are tied to internal implementation details hence they are subject to change and can break with newer kernels. BPF programs need to change accordingly when this happens.

Q: Are places where kprobes can attach part of the stable ABI?

A: NO. The places to which kprobes can attach are internal implementation details, which means that they are subject to change and can break with newer kernels. BPF programs need to change accordingly when this happens.

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 specific functions exposed as BPF helpers or kfuncs. The set of available functions 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: 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: Yes, through kfuncs and kptrs

The core BPF functionality such as program types, maps and helpers cannot be added to by modules. However, modules can expose functionality to BPF programs by exporting kfuncs (which may return pointers to module-internal data structures as kptrs).

Q: Directly calling kernel function is an ABI?

Q: Some kernel functions (e.g. tcp_slow_start) can be called by BPF programs. Do these kernel functions become an ABI?

A: NO.

The kernel function protos will change and the bpf programs will be rejected by the verifier. Also, for example, some of the bpf-callable kernel functions have already been used by other kernel tcp cc (congestion-control) implementations. If any of these kernel functions has changed, both the in-tree and out-of-tree kernel tcp cc implementations have to be changed. The same goes for the bpf programs and they have to be adjusted accordingly. See *3. kfunc lifecycle expectations* for details.

Q: Attaching to arbitrary kernel functions is an ABI?

Q: BPF programs can be attached to many kernel functions. Do these kernel functions become part of the ABI?

A: NO.

The kernel function prototypes will change, and BPF programs attaching to them will need to change. The BPF compile-once-run-everywhere (CO-RE) should be used in order to make it easier to adapt your BPF programs to different versions of the kernel.

Q: Marking a function with BTF_ID makes that function an ABI?

A: NO.

The BTF_ID macro does not cause a function to become part of the ABI any more than does the EXPORT_SYMBOL_GPL macro.

Q: What is the compatibility story for special BPF types in map values?

Q: Users are allowed to embed bpf_spin_lock, bpf_timer fields in their BPF map values (when using BTF support for BPF maps). This allows to use helpers for such objects on these fields inside map values. Users are also allowed to embed pointers to some kernel types (with __kptr_untrusted and __kptr BTF tags). Will the kernel preserve backwards compatibility for these features?

A: It depends. For bpf_spin_lock, bpf_timer: YES, for kptr and everything else: NO, but see below.

For struct types that have been added already, like bpf_spin_lock and bpf_timer, the kernel will preserve backwards compatibility, as they are part of UAPI.

For kptrs, they are also part of UAPI, but only with respect to the kptr mechanism. The types that you can use with a __kptr_untrusted and __kptr tagged pointer in your struct are NOT part of the UAPI contract. The supported types can and will change across kernel releases. However, operations like accessing kptr fields and bpf_kptr_xchg() helper will continue to be supported across kernel releases for the supported types.

For any other supported struct type, unless explicitly stated in this document and added to bpf.h UAPI header, such types can and will arbitrarily change their size, type, and alignment, or any other user visible API or ABI detail across kernel releases. The users must adapt their BPF programs to the new changes and update them to make sure their programs continue to work correctly.

NOTE: BPF subsystem specially reserves the ‘bpf_’ prefix for type names, in order to introduce more special fields in the future. Hence, user programs must avoid defining types with ‘bpf_’ prefix to not be broken in future releases. In other words, no backwards compatibility is guaranteed if one using a type in BTF with ‘bpf_’ prefix.

Q: What is the compatibility story for special BPF types in allocated objects?

Q: Same as above, but for allocated objects (i.e. objects allocated using bpf_obj_new for user defined types). Will the kernel preserve backwards compatibility for these features?

A: NO.

Unlike map value types, the API to work with allocated objects and any support for special fields inside them is exposed through kfuncs, and thus has the same lifecycle expectations as the kfuncs themselves. See *3. kfunc lifecycle expectations* for details.

5.2 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/submitting-patches.rst. This document only describes additional specifics related to BPF.

5.2.1 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.

5.2.2 Submitting patches

Q: How do I run BPF CI on my changes before sending them out for review?

A: BPF CI is GitHub based and hosted at https://github.com/kernel-patches/bpf. While GitHub also provides a CLI that can be used to accomplish the same results, here we focus on the UI based workflow.

The following steps lay out how to start a CI run for your patches:

- Create a fork of the aforementioned repository in your own account (one time action)
- Clone the fork locally, check out a new branch tracking either the bpf-next or bpf branch, and apply your to-be-tested patches on top of it
- Push the local branch to your fork and create a pull request against kernel-patches/bpf's bpf-next_base or bpf_base branch, respectively

Shortly after the pull request has been created, the CI workflow will run. Note that capacity is shared with patches submitted upstream being checked and so depending on utilization the run can take a while to finish.

Note furthermore that both base branches (bpf-next_base and bpf_base) will be updated as patches are pushed to the respective upstream branches they track. As such, your patch set will automatically (be attempted to) be rebased as well. This behavior can result in a CI run being aborted and restarted with the new base line.

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:

> https://patchwork.kernel.org/project/netdevbpf/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.kernel.org/project/netdevbpf/list/?delegate=121173

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 documentation on netdev subsystem at Documentation/process/maintainer-netdev.rst.

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 subsystem documentation at Documentation/process/maintainer-netdev.rst, 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 bpf-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 documentation for netdev subsystem at Documentation/process/maintainer-netdev.rst 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 docs at Documentation/process/maintainer-netdev.rst 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 selftests?

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.

5.2.3 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 the documentation on networking subsystem at Documentation/process/maintainer-netdev.rst.

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 docs at Documentation/process/maintainer-netdev.rst.

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.

5.2.4 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 kernel selftest documentation for details.

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 *BPF Type Format (BTF)* - 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

pahole starts to use libbpf definitions and APIs since v1.13 after the commit 21507cd3e97b ("pahole: add libbpf as submodule under lib/bpf"). It works well with the git repository because the libbpf submodule will use "git submodule update --init --recursive" to update.

Unfortunately, the default github release source code does not contain libbpf submodule source code and this will cause build issues, the tarball from https://git.kernel.org/pub/scm/devel/pahole/pahole.git/ is same with github, you can get the source tarball with corresponding libbpf submodule codes from

https://fedorapeople.org/~acme/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.

5.2.5 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 10.0.0
```

```
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

  Registered Targets:
    aarch64    - AArch64 (little endian)
    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: We recommend that developers who want the fastest incremental builds use the Ninja build system, you can find it in your system's package manager, usually the package is ninja or ninja-build.

You need ninja, 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
$ cd llvm-project/llvm/build
$ cmake .. -G "Ninja" -DLLVM_TARGETS_TO_BUILD="BPF;X86" \
           -DLLVM_ENABLE_PROJECTS="clang"    \
           -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.

Set `-DLLVM_TARGETS_TO_BUILD` equal to the target you wish to build, you will find a full list of targets within the llvm-project/llvm/lib/Target directory.

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

SIX

SYSCALL API

The primary info for the bpf syscall is available in the man-pages for bpf(2). For more information about the userspace API, see Documentation/userspace-api/ebpf/index.rst.

CHAPTER

SEVEN

HELPER FUNCTIONS

- bpf-helpers(7) maintains a list of helpers available to eBPF programs.

CHAPTER

EIGHT

BPF KERNEL FUNCTIONS (KFUNCS)

8.1 1. Introduction

BPF Kernel Functions or more commonly known as kfuncs are functions in the Linux kernel which are exposed for use by BPF programs. Unlike normal BPF helpers, kfuncs do not have a stable interface and can change from one kernel release to another. Hence, BPF programs need to be updated in response to changes in the kernel. See *3. kfunc lifecycle expectations* for more information.

8.2 2. Defining a kfunc

There are two ways to expose a kernel function to BPF programs, either make an existing function in the kernel visible, or add a new wrapper for BPF. In both cases, care must be taken that BPF program can only call such function in a valid context. To enforce this, visibility of a kfunc can be per program type.

If you are not creating a BPF wrapper for existing kernel function, skip ahead to *2.3 Using an existing kernel function*.

8.2.1 2.1 Creating a wrapper kfunc

When defining a wrapper kfunc, the wrapper function should have extern linkage. This prevents the compiler from optimizing away dead code, as this wrapper kfunc is not invoked anywhere in the kernel itself. It is not necessary to provide a prototype in a header for the wrapper kfunc.

An example is given below:

```
/* Disables missing prototype warnings */
__diag_push();
__diag_ignore_all("-Wmissing-prototypes",
                  "Global kfuncs as their definitions will be in BTF");

__bpf_kfunc struct task_struct *bpf_find_get_task_by_vpid(pid_t nr)
{
        return find_get_task_by_vpid(nr);
}

__diag_pop();
```

A wrapper kfunc is often needed when we need to annotate parameters of the kfunc. Otherwise one may directly make the kfunc visible to the BPF program by registering it with the BPF subsystem. See *2.3 Using an existing kernel function*.

8.2.2 2.2 Annotating kfunc parameters

Similar to BPF helpers, there is sometime need for additional context required by the verifier to make the usage of kernel functions safer and more useful. Hence, we can annotate a parameter by suffixing the name of the argument of the kfunc with a __tag, where tag may be one of the supported annotations.

8.2.3 2.2.1 __sz Annotation

This annotation is used to indicate a memory and size pair in the argument list. An example is given below:

```
__bpf_kfunc void bpf_memzero(void *mem, int mem__sz)
{
...
}
```

Here, the verifier will treat first argument as a PTR_TO_MEM, and second argument as its size. By default, without __sz annotation, the size of the type of the pointer is used. Without __sz annotation, a kfunc cannot accept a void pointer.

8.2.4 2.2.2 __k Annotation

This annotation is only understood for scalar arguments, where it indicates that the verifier must check the scalar argument to be a known constant, which does not indicate a size parameter, and the value of the constant is relevant to the safety of the program.

An example is given below:

```
__bpf_kfunc void *bpf_obj_new(u32 local_type_id__k, ...)
{
...
}
```

Here, bpf_obj_new uses local_type_id argument to find out the size of that type ID in program's BTF and return a sized pointer to it. Each type ID will have a distinct size, hence it is crucial to treat each such call as distinct when values don't match during verifier state pruning checks.

Hence, whenever a constant scalar argument is accepted by a kfunc which is not a size parameter, and the value of the constant matters for program safety, __k suffix should be used.

8.2.5 2.2.3 __uninit Annotation

This annotation is used to indicate that the argument will be treated as uninitialized.

An example is given below:

```
__bpf_kfunc int bpf_dynptr_from_skb(..., struct bpf_dynptr_kern *ptr__uninit)
{
...
}
```

Here, the dynptr will be treated as an uninitialized dynptr. Without this annotation, the verifier will reject the program if the dynptr passed in is not initialized.

8.2.6 2.2.4 __opt Annotation

This annotation is used to indicate that the buffer associated with an __sz or __szk argument may be null. If the function is passed a nullptr in place of the buffer, the verifier will not check that length is appropriate for the buffer. The kfunc is responsible for checking if this buffer is null before using it.

An example is given below:

```
__bpf_kfunc void *bpf_dynptr_slice(..., void *buffer__opt, u32 buffer__szk)
{
...
}
```

Here, the buffer may be null. If buffer is not null, it at least of size buffer_szk. Either way, the returned buffer is either NULL, or of size buffer_szk. Without this annotation, the verifier will reject the program if a null pointer is passed in with a nonzero size.

8.2.7 2.3 Using an existing kernel function

When an existing function in the kernel is fit for consumption by BPF programs, it can be directly registered with the BPF subsystem. However, care must still be taken to review the context in which it will be invoked by the BPF program and whether it is safe to do so.

8.2.8 2.4 Annotating kfuncs

In addition to kfuncs' arguments, verifier may need more information about the type of kfunc(s) being registered with the BPF subsystem. To do so, we define flags on a set of kfuncs as follows:

```
BTF_SET8_START(bpf_task_set)
BTF_ID_FLAGS(func, bpf_get_task_pid, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, bpf_put_pid, KF_RELEASE)
BTF_SET8_END(bpf_task_set)
```

This set encodes the BTF ID of each kfunc listed above, and encodes the flags along with it. Ofcourse, it is also allowed to specify no flags.

kfunc definitions should also always be annotated with the `__bpf_kfunc` macro. This prevents issues such as the compiler inlining the kfunc if it's a static kernel function, or the function being elided in an LTO build as it's not used in the rest of the kernel. Developers should not manually add annotations to their kfunc to prevent these issues. If an annotation is required to prevent such an issue with your kfunc, it is a bug and should be added to the definition of the macro so that other kfuncs are similarly protected. An example is given below:

```
__bpf_kfunc struct task_struct *bpf_get_task_pid(s32 pid)
{
...
}
```

8.2.9 2.4.1 KF_ACQUIRE flag

The KF_ACQUIRE flag is used to indicate that the kfunc returns a pointer to a refcounted object. The verifier will then ensure that the pointer to the object is eventually released using a release kfunc, or transferred to a map using a referenced kptr (by invoking bpf_kptr_xchg). If not, the verifier fails the loading of the BPF program until no lingering references remain in all possible explored states of the program.

8.2.10 2.4.2 KF_RET_NULL flag

The KF_RET_NULL flag is used to indicate that the pointer returned by the kfunc may be NULL. Hence, it forces the user to do a NULL check on the pointer returned from the kfunc before making use of it (dereferencing or passing to another helper). This flag is often used in pairing with KF_ACQUIRE flag, but both are orthogonal to each other.

8.2.11 2.4.3 KF_RELEASE flag

The KF_RELEASE flag is used to indicate that the kfunc releases the pointer passed in to it. There can be only one referenced pointer that can be passed in. All copies of the pointer being released are invalidated as a result of invoking kfunc with this flag. KF_RELEASE kfuncs automatically receive the protection afforded by the KF_TRUSTED_ARGS flag described below.

8.2.12 2.4.4 KF_TRUSTED_ARGS flag

The KF_TRUSTED_ARGS flag is used for kfuncs taking pointer arguments. It indicates that the all pointer arguments are valid, and that all pointers to BTF objects have been passed in their unmodified form (that is, at a zero offset, and without having been obtained from walking another pointer, with one exception described below).

There are two types of pointers to kernel objects which are considered "valid":

1. Pointers which are passed as tracepoint or struct_ops callback arguments.
2. Pointers which were returned from a KF_ACQUIRE kfunc.

Pointers to non-BTF objects (e.g. scalar pointers) may also be passed to KF_TRUSTED_ARGS kfuncs, and may have a non-zero offset.

The definition of "valid" pointers is subject to change at any time, and has absolutely no ABI stability guarantees.

As mentioned above, a nested pointer obtained from walking a trusted pointer is no longer trusted, with one exception. If a struct type has a field that is guaranteed to be valid (trusted or rcu, as in KF_RCU description below) as long as its parent pointer is valid, the following macros can be used to express that to the verifier:

- `BTF_TYPE_SAFE_TRUSTED`
- `BTF_TYPE_SAFE_RCU`
- `BTF_TYPE_SAFE_RCU_OR_NULL`

For example,

```
BTF_TYPE_SAFE_TRUSTED(struct socket) {
        struct sock *sk;
};
```

or

```
BTF_TYPE_SAFE_RCU(struct task_struct) {
        const cpumask_t *cpus_ptr;
        struct css_set __rcu *cgroups;
        struct task_struct __rcu *real_parent;
        struct task_struct *group_leader;
};
```

In other words, you must:

1. Wrap the valid pointer type in a `BTF_TYPE_SAFE_*` macro.
2. Specify the type and name of the valid nested field. This field must match the field in the original type definition exactly.

A new type declared by a `BTF_TYPE_SAFE_*` macro also needs to be emitted so that it appears in BTF. For example, `BTF_TYPE_SAFE_TRUSTED(struct socket)` is emitted in the `type_is_trusted()` function as follows:

```
BTF_TYPE_EMIT(BTF_TYPE_SAFE_TRUSTED(struct socket));
```

8.2.13 2.4.5 KF_SLEEPABLE flag

The KF_SLEEPABLE flag is used for kfuncs that may sleep. Such kfuncs can only be called by sleepable BPF programs (BPF_F_SLEEPABLE).

8.2.14 2.4.6 KF_DESTRUCTIVE flag

The KF_DESTRUCTIVE flag is used to indicate functions calling which is destructive to the system. For example such a call can result in system rebooting or panicking. Due to this additional restrictions apply to these calls. At the moment they only require CAP_SYS_BOOT capability, but more can be added later.

8.2.15 2.4.7 KF_RCU flag

The KF_RCU flag is a weaker version of KF_TRUSTED_ARGS. The kfuncs marked with KF_RCU expect either PTR_TRUSTED or MEM_RCU arguments. The verifier guarantees that the objects are valid and there is no use-after-free. The pointers are not NULL, but the object's refcount could have reached zero. The kfuncs need to consider doing refcnt != 0 check, especially when returning a KF_ACQUIRE pointer. Note as well that a KF_ACQUIRE kfunc that is KF_RCU should very likely also be KF_RET_NULL.

8.2.16 2.4.8 KF_DEPRECATED flag

The KF_DEPRECATED flag is used for kfuncs which are scheduled to be changed or removed in a subsequent kernel release. A kfunc that is marked with KF_DEPRECATED should also have any relevant information captured in its kernel doc. Such information typically includes the kfunc's expected remaining lifespan, a recommendation for new functionality that can replace it if any is available, and possibly a rationale for why it is being removed.

Note that while on some occasions, a KF_DEPRECATED kfunc may continue to be supported and have its KF_DEPRECATED flag removed, it is likely to be far more difficult to remove a KF_DEPRECATED flag after it's been added than it is to prevent it from being added in the first place. As described in *3. kfunc lifecycle expectations*, users that rely on specific kfuncs are encouraged to make their use-cases known as early as possible, and participate in upstream discussions regarding whether to keep, change, deprecate, or remove those kfuncs if and when such discussions occur.

8.2.17 2.5 Registering the kfuncs

Once the kfunc is prepared for use, the final step to making it visible is registering it with the BPF subsystem. Registration is done per BPF program type. An example is shown below:

```
BTF_SET8_START(bpf_task_set)
BTF_ID_FLAGS(func, bpf_get_task_pid, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, bpf_put_pid, KF_RELEASE)
BTF_SET8_END(bpf_task_set)

static const struct btf_kfunc_id_set bpf_task_kfunc_set = {
        .owner = THIS_MODULE,
        .set   = &bpf_task_set,
};

static int init_subsystem(void)
{
        return register_btf_kfunc_id_set(BPF_PROG_TYPE_TRACING, &bpf_task_
```

```
↪kfunc_set);
}
late_initcall(init_subsystem);
```

8.2.18 2.6 Specifying no-cast aliases with ___init

The verifier will always enforce that the BTF type of a pointer passed to a kfunc by a BPF program, matches the type of pointer specified in the kfunc definition. The verifier, does, however, allow types that are equivalent according to the C standard to be passed to the same kfunc arg, even if their BTF_IDs differ.

For example, for the following type definition:

```
struct bpf_cpumask {
        cpumask_t cpumask;
        refcount_t usage;
};
```

The verifier would allow a `struct bpf_cpumask *` to be passed to a kfunc taking a `cpumask_t *` (which is a typedef of `struct cpumask *`). For instance, both `struct cpumask *` and `struct bpf_cpmuask *` can be passed to *bpf_cpumask_test_cpu()*.

In some cases, this type-aliasing behavior is not desired. `struct nf_conn___init` is one such example:

```
struct nf_conn___init {
        struct nf_conn ct;
};
```

The C standard would consider these types to be equivalent, but it would not always be safe to pass either type to a trusted kfunc. `struct nf_conn___init` represents an allocated `struct nf_conn` object that has *not yet been initialized*, so it would therefore be unsafe to pass a `struct nf_conn___init *` to a kfunc that's expecting a fully initialized `struct nf_conn *` (e.g. `bpf_ct_change_timeout()`).

In order to accommodate such requirements, the verifier will enforce strict PTR_TO_BTF_ID type matching if two types have the exact same name, with one being suffixed with `___init`.

8.3 3. kfunc lifecycle expectations

kfuncs provide a kernel <-> kernel API, and thus are not bound by any of the strict stability restrictions associated with kernel <-> user UAPIs. This means they can be thought of as similar to EXPORT_SYMBOL_GPL, and can therefore be modified or removed by a maintainer of the subsystem they're defined in when it's deemed necessary.

Like any other change to the kernel, maintainers will not change or remove a kfunc without having a reasonable justification. Whether or not they'll choose to change a kfunc will ultimately depend on a variety of factors, such as how widely used the kfunc is, how long the kfunc has been in the kernel, whether an alternative kfunc exists, what the norm is in terms of stability for the subsystem in question, and of course what the technical cost is of continuing to support the kfunc.

There are several implications of this:

a) kfuncs that are widely used or have been in the kernel for a long time will be more difficult to justify being changed or removed by a maintainer. In other words, kfuncs that are known to have a lot of users and provide significant value provide stronger incentives for maintainers to invest the time and complexity in supporting them. It is therefore important for developers that are using kfuncs in their BPF programs to communicate and explain how and why those kfuncs are being used, and to participate in discussions regarding those kfuncs when they occur upstream.

b) Unlike regular kernel symbols marked with EXPORT_SYMBOL_GPL, BPF programs that call kfuncs are generally not part of the kernel tree. This means that refactoring cannot typically change callers in-place when a kfunc changes, as is done for e.g. an upstreamed driver being updated in place when a kernel symbol is changed.

   Unlike with regular kernel symbols, this is expected behavior for BPF symbols, and out-of-tree BPF programs that use kfuncs should be considered relevant to discussions and decisions around modifying and removing those kfuncs. The BPF community will take an active role in participating in upstream discussions when necessary to ensure that the perspectives of such users are taken into account.

c) A kfunc will never have any hard stability guarantees. BPF APIs cannot and will not ever hard-block a change in the kernel purely for stability reasons. That being said, kfuncs are features that are meant to solve problems and provide value to users. The decision of whether to change or remove a kfunc is a multivariate technical decision that is made on a case-by-case basis, and which is informed by data points such as those mentioned above. It is expected that a kfunc being removed or changed with no warning will not be a common occurrence or take place without sound justification, but it is a possibility that must be accepted if one is to use kfuncs.

8.3.1 3.1 kfunc deprecation

As described above, while sometimes a maintainer may find that a kfunc must be changed or removed immediately to accommodate some changes in their subsystem, usually kfuncs will be able to accommodate a longer and more measured deprecation process. For example, if a new kfunc comes along which provides superior functionality to an existing kfunc, the existing kfunc may be deprecated for some period of time to allow users to migrate their BPF programs to use the new one. Or, if a kfunc has no known users, a decision may be made to remove the kfunc (without providing an alternative API) after some deprecation period so as to provide users with a window to notify the kfunc maintainer if it turns out that the kfunc is actually being used.

It's expected that the common case will be that kfuncs will go through a deprecation period rather than being changed or removed without warning. As described in *2.4.8 KF_DEPRECATED flag*, the kfunc framework provides the KF_DEPRECATED flag to kfunc developers to signal to users that a kfunc has been deprecated. Once a kfunc has been marked with KF_DEPRECATED, the following procedure is followed for removal:

1. Any relevant information for deprecated kfuncs is documented in the kfunc's kernel docs. This documentation will typically include the kfunc's expected remaining lifespan, a recommendation for new functionality that can replace the usage of the deprecated function (or an explanation as to why no such replacement exists), etc.

2. The deprecated kfunc is kept in the kernel for some period of time after it was first marked as deprecated. This time period will be chosen on a case-by-case basis, and will typically

depend on how widespread the use of the kfunc is, how long it has been in the kernel, and how hard it is to move to alternatives. This deprecation time period is "best effort", and as described *above*, circumstances may sometimes dictate that the kfunc be removed before the full intended deprecation period has elapsed.

3. After the deprecation period the kfunc will be removed. At this point, BPF programs calling the kfunc will be rejected by the verifier.

8.4 4. Core kfuncs

The BPF subsystem provides a number of "core" kfuncs that are potentially applicable to a wide variety of different possible use cases and programs. Those kfuncs are documented here.

8.4.1 4.1 struct task_struct * kfuncs

There are a number of kfuncs that allow `struct task_struct *` objects to be used as kptrs:

__bpf_kfunc struct task_struct ***bpf_task_acquire**(struct task_struct *p)

Acquire a reference to a task. A task acquired by this kfunc which is not stored in a map as a kptr, must be released by calling *bpf_task_release()*.

Parameters

struct task_struct *p

The task on which a reference is being acquired.

__bpf_kfunc void **bpf_task_release**(struct task_struct *p)

Release the reference acquired on a task.

Parameters

struct task_struct *p

The task on which a reference is being released.

These kfuncs are useful when you want to acquire or release a reference to a `struct task_struct *` that was passed as e.g. a tracepoint arg, or a struct_ops callback arg. For example:

```
/**
 * A trivial example tracepoint program that shows how to
 * acquire and release a struct task_struct * pointer.
 */
SEC("tp_btf/task_newtask")
int BPF_PROG(task_acquire_release_example, struct task_struct *task, u64 clone_
↪flags)
{
        struct task_struct *acquired;

        acquired = bpf_task_acquire(task);
        if (acquired)
                /*
                 * In a typical program you'd do something like store
                 * the task in a map, and the map will automatically
```

```
                 * release it later. Here, we release it manually.
                 */
                bpf_task_release(acquired);
        return 0;
}
```

References acquired on `struct task_struct *` objects are RCU protected. Therefore, when in an RCU read region, you can obtain a pointer to a task embedded in a map value without having to acquire a reference:

```
#define private(name) SEC(".data." #name) __hidden __attribute__((aligned(8)))
private(TASK) static struct task_struct *global;

/**
 * A trivial example showing how to access a task stored
 * in a map using RCU.
 */
SEC("tp_btf/task_newtask")
int BPF_PROG(task_rcu_read_example, struct task_struct *task, u64 clone_flags)
{
        struct task_struct *local_copy;

        bpf_rcu_read_lock();
        local_copy = global;
        if (local_copy)
                /*
                 * We could also pass local_copy to kfuncs or helper functions␣
↪here,
                 * as we're guaranteed that local_copy will be valid until we␣
↪exit
                 * the RCU read region below.
                 */
                bpf_printk("Global task %s is valid", local_copy->comm);
        else
                bpf_printk("No global task found");
        bpf_rcu_read_unlock();

        /* At this point we can no longer reference local_copy. */

        return 0;
}
```

A BPF program can also look up a task from a pid. This can be useful if the caller doesn't have a trusted pointer to a `struct task_struct *` object that it can acquire a reference on with *bpf_task_acquire()*.

__bpf_kfunc struct task_struct ***bpf_task_from_pid**(s32 pid)

Find a struct task_struct from its pid by looking it up in the root pid namespace idr. If a task is returned, it must either be stored in a map, or released with *bpf_task_release()*.

Parameters

s32 pid
The pid of the task being looked up.

Here is an example of it being used:

```
SEC("tp_btf/task_newtask")
int BPF_PROG(task_get_pid_example, struct task_struct *task, u64 clone_flags)
{
        struct task_struct *lookup;

        lookup = bpf_task_from_pid(task->pid);
        if (!lookup)
                /* A task should always be found, as %task is a tracepoint arg.
 ↪ */
                return -ENOENT;

        if (lookup->pid != task->pid) {
                /* bpf_task_from_pid() looks up the task via its
                 * globally-unique pid from the init_pid_ns. Thus,
                 * the pid of the lookup task should always be the
                 * same as the input task.
                 */
                bpf_task_release(lookup);
                return -EINVAL;
        }

        /* bpf_task_from_pid() returns an acquired reference,
         * so it must be dropped before returning from the
         * tracepoint handler.
         */
        bpf_task_release(lookup);
        return 0;
}
```

8.4.2 4.2 struct cgroup * kfuncs

`struct cgroup *` objects also have acquire and release functions:

__bpf_kfunc struct cgroup ***bpf_cgroup_acquire**(struct cgroup *cgrp)
Acquire a reference to a cgroup. A cgroup acquired by this kfunc which is not stored in a map as a kptr, must be released by calling *bpf_cgroup_release()*.

Parameters

struct cgroup *cgrp
The cgroup on which a reference is being acquired.

__bpf_kfunc void **bpf_cgroup_release**(struct cgroup *cgrp)
Release the reference acquired on a cgroup. If this kfunc is invoked in an RCU read region, the cgroup is guaranteed to not be freed until the current grace period has ended, even if its refcount drops to 0.

Parameters

struct cgroup *cgrp
The cgroup on which a reference is being released.

These kfuncs are used in exactly the same manner as *bpf_task_acquire()* and *bpf_task_release()* respectively, so we won't provide examples for them.

Other kfuncs available for interacting with `struct cgroup *` objects are *bpf_cgroup_ancestor()* and *bpf_cgroup_from_id()*, allowing callers to access the ancestor of a cgroup and find a cgroup by its ID, respectively. Both return a cgroup kptr.

__bpf_kfunc struct cgroup ***bpf_cgroup_ancestor**(struct cgroup *cgrp, int level)
Perform a lookup on an entry in a cgroup's ancestor array. A cgroup returned by this kfunc which is not subsequently stored in a map, must be released by calling *bpf_cgroup_release()*.

Parameters

struct cgroup *cgrp
The cgroup for which we're performing a lookup.

int level
The level of ancestor to look up.

__bpf_kfunc struct cgroup ***bpf_cgroup_from_id**(u64 cgid)
Find a cgroup from its ID. A cgroup returned by this kfunc which is not subsequently stored in a map, must be released by calling *bpf_cgroup_release()*.

Parameters

u64 cgid
cgroup id.

Eventually, BPF should be updated to allow this to happen with a normal memory load in the program itself. This is currently not possible without more work in the verifier. *bpf_cgroup_ancestor()* can be used as follows:

```
/**
 * Simple tracepoint example that illustrates how a cgroup's
 * ancestor can be accessed using bpf_cgroup_ancestor().
 */
SEC("tp_btf/cgroup_mkdir")
int BPF_PROG(cgrp_ancestor_example, struct cgroup *cgrp, const char *path)
{
        struct cgroup *parent;

        /* The parent cgroup resides at the level before the current cgroup's␣
↪level. */
        parent = bpf_cgroup_ancestor(cgrp, cgrp->level - 1);
        if (!parent)
                return -ENOENT;

        bpf_printk("Parent id is %d", parent->self.id);
```

```
        /* Return the parent cgroup that was acquired above. */
        bpf_cgroup_release(parent);
        return 0;
}
```

8.4.3 4.3 struct cpumask * kfuncs

BPF provides a set of kfuncs that can be used to query, allocate, mutate, and destroy struct cpumask * objects. Please refer to *BPF cpumask kfuncs* for more details.

CHAPTER

NINE

BPF CPUMASK KFUNCS

9.1 1. Introduction

`struct cpumask` is a bitmap data structure in the kernel whose indices reflect the CPUs on the system. Commonly, cpumasks are used to track which CPUs a task is affinitized to, but they can also be used to e.g. track which cores are associated with a scheduling domain, which cores on a machine are idle, etc.

BPF provides programs with a set of *BPF Kernel Functions (kfuncs)* that can be used to allocate, mutate, query, and free cpumasks.

9.2 2. BPF cpumask objects

There are two different types of cpumasks that can be used by BPF programs.

9.2.1 2.1 `struct bpf_cpumask *`

`struct bpf_cpumask *` is a cpumask that is allocated by BPF, on behalf of a BPF program, and whose lifecycle is entirely controlled by BPF. These cpumasks are RCU-protected, can be mutated, can be used as kptrs, and can be safely cast to a `struct cpumask *`.

9.2.2 2.1.1 `struct bpf_cpumask *` lifecycle

A `struct bpf_cpumask *` is allocated, acquired, and released, using the following functions:

__bpf_kfunc struct bpf_cpumask ***bpf_cpumask_create**(void)

Create a mutable BPF cpumask.

Parameters

void

no arguments

Description

Allocates a cpumask that can be queried, mutated, acquired, and released by a BPF program. The cpumask returned by this function must either be embedded in a map as a kptr, or freed with *bpf_cpumask_release()*.

bpf_cpumask_create() allocates memory using the BPF memory allocator, and will not block. It may return NULL if no memory is available.

__bpf_kfunc struct bpf_cpumask ***bpf_cpumask_acquire**(struct bpf_cpumask *cpumask)

Acquire a reference to a BPF cpumask.

Parameters

struct bpf_cpumask *cpumask

The BPF cpumask being acquired. The cpumask must be a trusted pointer.

Description

Acquires a reference to a BPF cpumask. The cpumask returned by this function must either be embedded in a map as a kptr, or freed with *bpf_cpumask_release()*.

__bpf_kfunc void **bpf_cpumask_release**(struct bpf_cpumask *cpumask)

Release a previously acquired BPF cpumask.

Parameters

struct bpf_cpumask *cpumask

The cpumask being released.

Description

Releases a previously acquired reference to a BPF cpumask. When the final reference of the BPF cpumask has been released, it is subsequently freed in an RCU callback in the BPF memory allocator.

For example:

```
struct cpumask_map_value {
        struct bpf_cpumask __kptr * cpumask;
};

struct array_map {
        __uint(type, BPF_MAP_TYPE_ARRAY);
        __type(key, int);
        __type(value, struct cpumask_map_value);
        __uint(max_entries, 65536);
} cpumask_map SEC(".maps");

static int cpumask_map_insert(struct bpf_cpumask *mask, u32 pid)
{
        struct cpumask_map_value local, *v;
        long status;
        struct bpf_cpumask *old;
        u32 key = pid;

        local.cpumask = NULL;
        status = bpf_map_update_elem(&cpumask_map, &key, &local, 0);
        if (status) {
                bpf_cpumask_release(mask);
                return status;
        }
```

```
        v = bpf_map_lookup_elem(&cpumask_map, &key);
        if (!v) {
                bpf_cpumask_release(mask);
                return -ENOENT;
        }

        old = bpf_kptr_xchg(&v->cpumask, mask);
        if (old)
                bpf_cpumask_release(old);

        return 0;
}

/**
 * A sample tracepoint showing how a task's cpumask can be queried and
 * recorded as a kptr.
 */
SEC("tp_btf/task_newtask")
int BPF_PROG(record_task_cpumask, struct task_struct *task, u64 clone_flags)
{
        struct bpf_cpumask *cpumask;
        int ret;

        cpumask = bpf_cpumask_create();
        if (!cpumask)
                return -ENOMEM;

        if (!bpf_cpumask_full(task->cpus_ptr))
                bpf_printk("task %s has CPU affinity", task->comm);

        bpf_cpumask_copy(cpumask, task->cpus_ptr);
        return cpumask_map_insert(cpumask, task->pid);
}
```

9.2.3 2.1.1 struct bpf_cpumask * as kptrs

As mentioned and illustrated above, these `struct bpf_cpumask *` objects can also be stored in a map and used as kptrs. If a `struct bpf_cpumask *` is in a map, the reference can be removed from the map with bpf_kptr_xchg(), or opportunistically acquired using RCU:

```
/* struct containing the struct bpf_cpumask kptr which is stored in the map. */
struct cpumasks_kfunc_map_value {
        struct bpf_cpumask __kptr * bpf_cpumask;
};

/* The map containing struct cpumasks_kfunc_map_value entries. */
struct {
```

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

/* ... */

/**
 * A simple example tracepoint program showing how a
 * struct bpf_cpumask * kptr that is stored in a map can
 * be passed to kfuncs using RCU protection.
 */
SEC("tp_btf/cgroup_mkdir")
int BPF_PROG(cgrp_ancestor_example, struct cgroup *cgrp, const char *path)
{
        struct bpf_cpumask *kptr;
        struct cpumasks_kfunc_map_value *v;
        u32 key = 0;

        /* Assume a bpf_cpumask * kptr was previously stored in the map. */
        v = bpf_map_lookup_elem(&cpumasks_kfunc_map, &key);
        if (!v)
                return -ENOENT;

        bpf_rcu_read_lock();
        /* Acquire a reference to the bpf_cpumask * kptr that's already stored␣
↪in the map. */
        kptr = v->cpumask;
        if (!kptr) {
                /* If no bpf_cpumask was present in the map, it's because
                 * we're racing with another CPU that removed it with
                 * bpf_kptr_xchg() between the bpf_map_lookup_elem()
                 * above, and our load of the pointer from the map.
                 */
                bpf_rcu_read_unlock();
                return -EBUSY;
        }

        bpf_cpumask_setall(kptr);
        bpf_rcu_read_unlock();

        return 0;
}
```

9.2.4 2.2 `struct cpumask`

`struct cpumask` is the object that actually contains the cpumask bitmap being queried, mutated, etc. A `struct bpf_cpumask` wraps a `struct cpumask`, which is why it's safe to cast it as such (note however that it is **not** safe to cast a `struct cpumask *` to a `struct bpf_cpumask *`, and the verifier will reject any program that tries to do so).

As we'll see below, any kfunc that mutates its cpumask argument will take a `struct bpf_cpumask *` as that argument. Any argument that simply queries the cpumask will instead take a `struct cpumask *`.

9.3 3. cpumask kfuncs

Above, we described the kfuncs that can be used to allocate, acquire, release, etc a `struct bpf_cpumask *`. This section of the document will describe the kfuncs for mutating and querying cpumasks.

9.3.1 3.1 Mutating cpumasks

Some cpumask kfuncs are "read-only" in that they don't mutate any of their arguments, whereas others mutate at least one argument (which means that the argument must be a `struct bpf_cpumask *`, as described above).

This section will describe all of the cpumask kfuncs which mutate at least one argument. *3.2 Querying cpumasks* below describes the read-only kfuncs.

9.3.2 3.1.1 Setting and clearing CPUs

bpf_cpumask_set_cpu() and *bpf_cpumask_clear_cpu()* can be used to set and clear a CPU in a `struct bpf_cpumask` respectively:

__bpf_kfunc void **bpf_cpumask_set_cpu**(u32 cpu, struct bpf_cpumask *cpumask)
Set a bit for a CPU in a BPF cpumask.

Parameters

u32 cpu
The CPU to be set in the cpumask.

struct bpf_cpumask *cpumask
The BPF cpumask in which a bit is being set.

__bpf_kfunc void **bpf_cpumask_clear_cpu**(u32 cpu, struct bpf_cpumask *cpumask)
Clear a bit for a CPU in a BPF cpumask.

Parameters

u32 cpu
The CPU to be cleared from the cpumask.

struct bpf_cpumask *cpumask
The BPF cpumask in which a bit is being cleared.

These kfuncs are pretty straightforward, and can be used, for example, as follows:

```
/**
 * A sample tracepoint showing how a cpumask can be queried.
 */
SEC("tp_btf/task_newtask")
int BPF_PROG(test_set_clear_cpu, struct task_struct *task, u64 clone_flags)
{
        struct bpf_cpumask *cpumask;

        cpumask = bpf_cpumask_create();
        if (!cpumask)
                return -ENOMEM;

        bpf_cpumask_set_cpu(0, cpumask);
        if (!bpf_cpumask_test_cpu(0, cast(cpumask)))
                /* Should never happen. */
                goto release_exit;

        bpf_cpumask_clear_cpu(0, cpumask);
        if (bpf_cpumask_test_cpu(0, cast(cpumask)))
                /* Should never happen. */
                goto release_exit;

        /* struct cpumask * pointers such as task->cpus_ptr can also be␣
↪queried. */
        if (bpf_cpumask_test_cpu(0, task->cpus_ptr))
                bpf_printk("task %s can use CPU %d", task->comm, 0);

release_exit:
        bpf_cpumask_release(cpumask);
        return 0;
}
```

bpf_cpumask_test_and_set_cpu() and *bpf_cpumask_test_and_clear_cpu()* are complementary kfuncs that allow callers to atomically test and set (or clear) CPUs:

__bpf_kfunc bool **bpf_cpumask_test_and_set_cpu**(u32 cpu, struct bpf_cpumask *cpumask)
Atomically test and set a CPU in a BPF cpumask.

Parameters

u32 cpu
The CPU being set and queried for.

struct bpf_cpumask *cpumask
The BPF cpumask being set and queried for containing a CPU.

Return

- true - **cpu** is set in the cpumask
- false - **cpu** was not set in the cpumask, or **cpu** is invalid.

__bpf_kfunc bool **bpf_cpumask_test_and_clear_cpu**(u32 cpu, struct bpf_cpumask *cpumask)
Atomically test and clear a CPU in a BPF cpumask.

Parameters

u32 cpu
The CPU being cleared and queried for.

struct bpf_cpumask *cpumask
The BPF cpumask being cleared and queried for containing a CPU.

Return

- true - **cpu** is set in the cpumask
- false - **cpu** was not set in the cpumask, or **cpu** is invalid.

We can also set and clear entire `struct bpf_cpumask *` objects in one operation using *bpf_cpumask_setall()* and *bpf_cpumask_clear()*:

__bpf_kfunc void **bpf_cpumask_setall**(struct bpf_cpumask *cpumask)
Set all of the bits in a BPF cpumask.

Parameters

struct bpf_cpumask *cpumask
The BPF cpumask having all of its bits set.

__bpf_kfunc void **bpf_cpumask_clear**(struct bpf_cpumask *cpumask)
Clear all of the bits in a BPF cpumask.

Parameters

struct bpf_cpumask *cpumask
The BPF cpumask being cleared.

9.3.3 3.1.2 Operations between cpumasks

In addition to setting and clearing individual CPUs in a single cpumask, callers can also perform bitwise operations between multiple cpumasks using *bpf_cpumask_and()*, *bpf_cpumask_or()*, and *bpf_cpumask_xor()*:

__bpf_kfunc bool **bpf_cpumask_and**(struct bpf_cpumask *dst, const struct cpumask *src1,
const struct cpumask *src2)
AND two cpumasks and store the result.

Parameters

struct bpf_cpumask *dst
The BPF cpumask where the result is being stored.

const struct cpumask *src1
The first input.

const struct cpumask *src2
The second input.

Return

- true - **dst** has at least one bit set following the operation
- false - **dst** is empty following the operation

Description

struct bpf_cpumask pointers may be safely passed to **src1** and **src2**.

__bpf_kfunc void **bpf_cpumask_or**(struct bpf_cpumask *dst, const struct cpumask *src1, const struct cpumask *src2)

OR two cpumasks and store the result.

Parameters

struct bpf_cpumask *dst
The BPF cpumask where the result is being stored.

const struct cpumask *src1
The first input.

const struct cpumask *src2
The second input.

Description

struct bpf_cpumask pointers may be safely passed to **src1** and **src2**.

__bpf_kfunc void **bpf_cpumask_xor**(struct bpf_cpumask *dst, const struct cpumask *src1, const struct cpumask *src2)

XOR two cpumasks and store the result.

Parameters

struct bpf_cpumask *dst
The BPF cpumask where the result is being stored.

const struct cpumask *src1
The first input.

const struct cpumask *src2
The second input.

Description

struct bpf_cpumask pointers may be safely passed to **src1** and **src2**.

The following is an example of how they may be used. Note that some of the kfuncs shown in this example will be covered in more detail below.

```
/**
 * A sample tracepoint showing how a cpumask can be mutated using
   bitwise operators (and queried).
 */
SEC("tp_btf/task_newtask")
int BPF_PROG(test_and_or_xor, struct task_struct *task, u64 clone_flags)
{
        struct bpf_cpumask *mask1, *mask2, *dst1, *dst2;

        mask1 = bpf_cpumask_create();
        if (!mask1)
```

```
                return -ENOMEM;

        mask2 = bpf_cpumask_create();
        if (!mask2) {
                bpf_cpumask_release(mask1);
                return -ENOMEM;
        }

        // ...Safely create the other two masks... */

        bpf_cpumask_set_cpu(0, mask1);
        bpf_cpumask_set_cpu(1, mask2);
        bpf_cpumask_and(dst1, (const struct cpumask *)mask1, (const struct␣
↪cpumask *)mask2);
        if (!bpf_cpumask_empty((const struct cpumask *)dst1))
                /* Should never happen. */
                goto release_exit;

        bpf_cpumask_or(dst1, (const struct cpumask *)mask1, (const struct␣
↪cpumask *)mask2);
        if (!bpf_cpumask_test_cpu(0, (const struct cpumask *)dst1))
                /* Should never happen. */
                goto release_exit;

        if (!bpf_cpumask_test_cpu(1, (const struct cpumask *)dst1))
                /* Should never happen. */
                goto release_exit;

        bpf_cpumask_xor(dst2, (const struct cpumask *)mask1, (const struct␣
↪cpumask *)mask2);
        if (!bpf_cpumask_equal((const struct cpumask *)dst1,
                               (const struct cpumask *)dst2))
                /* Should never happen. */
                goto release_exit;

 release_exit:
        bpf_cpumask_release(mask1);
        bpf_cpumask_release(mask2);
        bpf_cpumask_release(dst1);
        bpf_cpumask_release(dst2);
        return 0;
}
```

The contents of an entire cpumask may be copied to another using *bpf_cpumask_copy()*:

__bpf_kfunc void **bpf_cpumask_copy**(struct bpf_cpumask *dst, const struct cpumask *src)
    Copy the contents of a cpumask into a BPF cpumask.

Parameters

struct bpf_cpumask *dst
The BPF cpumask being copied into.

const struct cpumask *src
The cpumask being copied.

Description

A struct bpf_cpumask pointer may be safely passed to **src**.

9.3.4 3.2 Querying cpumasks

In addition to the above kfuncs, there is also a set of read-only kfuncs that can be used to query the contents of cpumasks.

__bpf_kfunc u32 **bpf_cpumask_first**(const struct *cpumask* *cpumask)
Get the index of the first nonzero bit in the cpumask.

Parameters

const struct cpumask *cpumask
The cpumask being queried.

Description

Find the index of the first nonzero bit of the cpumask. A struct bpf_cpumask pointer may be safely passed to this function.

__bpf_kfunc u32 **bpf_cpumask_first_zero**(const struct *cpumask* *cpumask)
Get the index of the first unset bit in the cpumask.

Parameters

const struct cpumask *cpumask
The cpumask being queried.

Description

Find the index of the first unset bit of the cpumask. A struct bpf_cpumask pointer may be safely passed to this function.

__bpf_kfunc u32 **bpf_cpumask_first_and**(const struct cpumask *src1, const struct cpumask *src2)
Return the index of the first nonzero bit from the AND of two cpumasks.

Parameters

const struct cpumask *src1
The first cpumask.

const struct cpumask *src2
The second cpumask.

Description

Find the index of the first nonzero bit of the AND of two cpumasks. struct bpf_cpumask pointers may be safely passed to **src1** and **src2**.

__bpf_kfunc bool **bpf_cpumask_test_cpu**(u32 cpu, const struct *cpumask* *cpumask)

Test whether a CPU is set in a cpumask.

Parameters

u32 cpu
The CPU being queried for.

const struct cpumask *cpumask
The cpumask being queried for containing a CPU.

Return

- true - **cpu** is set in the cpumask
- false - **cpu** was not set in the cpumask, or **cpu** is an invalid cpu.

__bpf_kfunc bool **bpf_cpumask_equal**(const struct cpumask *src1, const struct cpumask *src2)

Check two cpumasks for equality.

Parameters

const struct cpumask *src1
The first input.

const struct cpumask *src2
The second input.

Return

- true - **src1** and **src2** have the same bits set.
- false - **src1** and **src2** differ in at least one bit.

Description

struct bpf_cpumask pointers may be safely passed to **src1** and **src2**.

__bpf_kfunc bool **bpf_cpumask_intersects**(const struct cpumask *src1, const struct cpumask *src2)

Check two cpumasks for overlap.

Parameters

const struct cpumask *src1
The first input.

const struct cpumask *src2
The second input.

Return

- true - **src1** and **src2** have at least one of the same bits set.
- false - **src1** and **src2** don't have any of the same bits set.

Description

struct bpf_cpumask pointers may be safely passed to **src1** and **src2**.

__bpf_kfunc bool **bpf_cpumask_subset**(const struct cpumask *src1, const struct cpumask *src2)

Check if a cpumask is a subset of another.

Parameters

const struct cpumask *src1
The first cpumask being checked as a subset.

const struct cpumask *src2
The second cpumask being checked as a superset.

Return

- true - All of the bits of **src1** are set in **src2**.
- false - At least one bit in **src1** is not set in **src2**.

Description

struct bpf_cpumask pointers may be safely passed to **src1** and **src2**.

__bpf_kfunc bool **bpf_cpumask_empty**(const struct *cpumask* *cpumask)

Check if a cpumask is empty.

Parameters

const struct cpumask *cpumask
The cpumask being checked.

Return

- true - None of the bits in **cpumask** are set.
- false - At least one bit in **cpumask** is set.

Description

A struct bpf_cpumask pointer may be safely passed to **cpumask**.

__bpf_kfunc bool **bpf_cpumask_full**(const struct *cpumask* *cpumask)

Check if a cpumask has all bits set.

Parameters

const struct cpumask *cpumask
The cpumask being checked.

Return

- true - All of the bits in **cpumask** are set.
- false - At least one bit in **cpumask** is cleared.

Description

A struct bpf_cpumask pointer may be safely passed to **cpumask**.

__bpf_kfunc u32 **bpf_cpumask_any_distribute**(const struct *cpumask* *cpumask)

Return a random set CPU from a cpumask.

Parameters

const struct cpumask *cpumask
The cpumask being queried.

Return

- A random set bit within [0, num_cpus) if at least one bit is set.
- >= num_cpus if no bit is set.

Description

A struct bpf_cpumask pointer may be safely passed to **src**.

__bpf_kfunc u32 **bpf_cpumask_any_and_distribute**(const struct cpumask *src1, const struct cpumask *src2)

Return a random set CPU from the AND of two cpumasks.

Parameters

const struct cpumask *src1
The first cpumask.

const struct cpumask *src2
The second cpumask.

Return

- A random set bit within [0, num_cpus) from the AND of two cpumasks, if at least one bit is set.
- >= num_cpus if no bit is set.

Description

struct bpf_cpumask pointers may be safely passed to **src1** and **src2**.

Some example usages of these querying kfuncs were shown above. We will not replicate those examples here. Note, however, that all of the aforementioned kfuncs are tested in tools/testing/selftests/bpf/progs/cpumask_success.c, so please take a look there if you're looking for more examples of how they can be used.

9.4 4. Adding BPF cpumask kfuncs

The set of supported BPF cpumask kfuncs are not (yet) a 1-1 match with the cpumask operations in include/linux/cpumask.h. Any of those cpumask operations could easily be encapsulated in a new kfunc if and when required. If you'd like to support a new cpumask operation, please feel free to submit a patch. If you do add a new cpumask kfunc, please document it here, and add any relevant selftest testcases to the cpumask selftest suite.

CHAPTER

TEN

PROGRAM TYPES

10.1 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`.

10.1.1 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.

10.1.2 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).

10.1.3 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.

10.1.4 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 original kernel buffers without any modifications that the BPF program might have applied.

10.1.5 Example

Recommended way to handle BPF programs is as follows:

```
SEC("cgroup/getsockopt")
int getsockopt(struct bpf_sockopt *ctx)
{
        /* Custom socket option. */
        if (ctx->level == MY_SOL && ctx->optname == MY_OPTNAME) {
                ctx->retval = 0;
                optval[0] = ...;
```

```
                ctx->optlen = 1;
                return 1;
        }

        /* Modify kernel's socket option. */
        if (ctx->level == SOL_IP && ctx->optname == IP_FREEBIND) {
                ctx->retval = 0;
                optval[0] = ...;
                ctx->optlen = 1;
                return 1;
        }

        /* optval larger than PAGE_SIZE use kernel's buffer. */
        if (ctx->optlen > PAGE_SIZE)
                ctx->optlen = 0;

        return 1;
}

SEC("cgroup/setsockopt")
int setsockopt(struct bpf_sockopt *ctx)
{
        /* Custom socket option. */
        if (ctx->level == MY_SOL && ctx->optname == MY_OPTNAME) {
                /* do something */
                ctx->optlen = -1;
                return 1;
        }

        /* Modify kernel's socket option. */
        if (ctx->level == SOL_IP && ctx->optname == IP_FREEBIND) {
                optval[0] = ...;
                return 1;
        }

        /* optval larger than PAGE_SIZE use kernel's buffer. */
        if (ctx->optlen > PAGE_SIZE)
                ctx->optlen = 0;

        return 1;
}
```

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

10.2 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.

10.2.1 1. Attach type

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

10.2.2 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.

10.2.3 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`.

10.2.4 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.

10.2.5 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.

10.2.6 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.

10.3 BPF_PROG_TYPE_FLOW_DISSECTOR

10.3.1 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).

10.3.2 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.

10.3.3 __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.

10.3.4 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.

10.3.5 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).

10.3.6 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.

10.4 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.

10.4.1 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 long prot);

Other LSM hooks which can be instrumented can be found in `security/security.c`.

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.

10.4.2 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.

10.4.3 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.

10.4.4 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

10.5 BPF sk_lookup program

BPF sk_lookup program type (BPF_PROG_TYPE_SK_LOOKUP) introduces programmability into the socket lookup performed by the transport layer when a packet is to be delivered locally.

When invoked BPF sk_lookup program can select a socket that will receive the incoming packet by calling the bpf_sk_assign() BPF helper function.

Hooks for a common attach point (BPF_SK_LOOKUP) exist for both TCP and UDP.

10.5.1 Motivation

BPF sk_lookup program type was introduced to address setup scenarios where binding sockets to an address with `bind()` socket call is impractical, such as:

1. receiving connections on a range of IP addresses, e.g. 192.0.2.0/24, when binding to a wildcard address `INADRR_ANY` is not possible due to a port conflict,
2. receiving connections on all or a wide range of ports, i.e. an L7 proxy use case.

Such setups would require creating and `bind()`'ing one socket to each of the IP address/port in the range, leading to resource consumption and potential latency spikes during socket lookup.

10.5.2 Attachment

BPF sk_lookup program can be attached to a network namespace with `bpf(BPF_LINK_CREATE, ...)` syscall using the `BPF_SK_LOOKUP` attach type and a netns FD as attachment `target_fd`.

Multiple programs can be attached to one network namespace. Programs will be invoked in the same order as they were attached.

10.5.3 Hooks

The attached BPF sk_lookup programs run whenever the transport layer needs to find a listening (TCP) or an unconnected (UDP) socket for an incoming packet.

Incoming traffic to established (TCP) and connected (UDP) sockets is delivered as usual without triggering the BPF sk_lookup hook.

The attached BPF programs must return with either `SK_PASS` or `SK_DROP` verdict code. As for other BPF program types that are network filters, `SK_PASS` signifies that the socket lookup should continue on to regular hashtable-based lookup, while `SK_DROP` causes the transport layer to drop the packet.

A BPF sk_lookup program can also select a socket to receive the packet by calling `bpf_sk_assign()` BPF helper. Typically, the program looks up a socket in a map holding sockets, such as `SOCKMAP` or `SOCKHASH`, and passes a `struct bpf_sock *` to `bpf_sk_assign()` helper to record the selection. Selecting a socket only takes effect if the program has terminated with `SK_PASS` code.

When multiple programs are attached, the end result is determined from return codes of all the programs according to the following rules:

1. If any program returned `SK_PASS` and selected a valid socket, the socket is used as the result of the socket lookup.
2. If more than one program returned `SK_PASS` and selected a socket, the last selection takes effect.
3. If any program returned `SK_DROP`, and no program returned `SK_PASS` and selected a socket, socket lookup fails.
4. If all programs returned `SK_PASS` and none of them selected a socket, socket lookup continues on.

10.5.4 API

In its context, an instance of `struct bpf_sk_lookup`, BPF sk_lookup program receives information about the packet that triggered the socket lookup. Namely:

- IP version (`AF_INET` or `AF_INET6`),
- L4 protocol identifier (`IPPROTO_TCP` or `IPPROTO_UDP`),
- source and destination IP address,
- source and destination L4 port,
- the socket that has been selected with `bpf_sk_assign()`.

Refer to `struct bpf_sk_lookup` declaration in `linux/bpf.h` user API header, and bpf-helpers(7) man-page section for `bpf_sk_assign()` for details.

10.5.5 Example

See `tools/testing/selftests/bpf/prog_tests/sk_lookup.c` for the reference implementation.

For a list of all program types, see *Program Types and ELF Sections* in the *libbpf* documentation.

CHAPTER

ELEVEN

BPF MAPS

BPF 'maps' provide generic storage of different types for sharing data between kernel and user space. There are several storage types available, including hash, array, bloom filter and radix-tree. Several of the map types exist to support specific BPF helpers that perform actions based on the map contents. The maps are accessed from BPF programs via BPF helpers which are documented in the man-pages for bpf-helpers(7).

BPF maps are accessed from user space via the `bpf` syscall, which provides commands to create maps, lookup elements, update elements and delete elements. More details of the BPF syscall are available in ebpf-syscall and in the man-pages for bpf(2).

11.1 Map Types

11.1.1 BPF_MAP_TYPE_ARRAY and BPF_MAP_TYPE_PERCPU_ARRAY

Note:

- `BPF_MAP_TYPE_ARRAY` was introduced in kernel version 3.19
- `BPF_MAP_TYPE_PERCPU_ARRAY` was introduced in version 4.6

`BPF_MAP_TYPE_ARRAY` and `BPF_MAP_TYPE_PERCPU_ARRAY` provide generic array storage. The key type is an unsigned 32-bit integer (4 bytes) and the map is of constant size. The size of the array is defined in `max_entries` at creation time. All array elements are pre-allocated and zero initialized when created. `BPF_MAP_TYPE_PERCPU_ARRAY` uses a different memory region for each CPU whereas `BPF_MAP_TYPE_ARRAY` uses the same memory region. The value stored can be of any size, however, all array elements are aligned to 8 bytes.

Since kernel 5.5, memory mapping may be enabled for `BPF_MAP_TYPE_ARRAY` by setting the flag `BPF_F_MMAPABLE`. The map definition is page-aligned and starts on the first page. Sufficient page-sized and page-aligned blocks of memory are allocated to store all array values, starting on the second page, which in some cases will result in over-allocation of memory. The benefit of using this is increased performance and ease of use since userspace programs would not be required to use helper functions to access and mutate data.

Usage

Kernel BPF

bpf_map_lookup_elem()

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

Array elements can be retrieved using the `bpf_map_lookup_elem()` helper. This helper returns a pointer into the array element, so to avoid data races with userspace reading the value, the user must use primitives like `__sync_fetch_and_add()` when updating the value in-place.

bpf_map_update_elem()

```
long bpf_map_update_elem(struct bpf_map *map, const void *key, const void␣
↪*value, u64 flags)
```

Array elements can be updated using the `bpf_map_update_elem()` helper.

`bpf_map_update_elem()` returns 0 on success, or negative error in case of failure.

Since the array is of constant size, `bpf_map_delete_elem()` is not supported. To clear an array element, you may use `bpf_map_update_elem()` to insert a zero value to that index.

Per CPU Array

Values stored in `BPF_MAP_TYPE_ARRAY` can be accessed by multiple programs across different CPUs. To restrict storage to a single CPU, you may use a `BPF_MAP_TYPE_PERCPU_ARRAY`.

When using a `BPF_MAP_TYPE_PERCPU_ARRAY` the `bpf_map_update_elem()` and `bpf_map_lookup_elem()` helpers automatically access the slot for the current CPU.

bpf_map_lookup_percpu_elem()

```
void *bpf_map_lookup_percpu_elem(struct bpf_map *map, const void *key, u32 cpu)
```

The `bpf_map_lookup_percpu_elem()` helper can be used to lookup the array value for a specific CPU. Returns value on success , or `NULL` if no entry was found or `cpu` is invalid.

Concurrency

Since kernel version 5.1, the BPF infrastructure provides `struct bpf_spin_lock` to synchronize access.

Userspace

Access from userspace uses libbpf APIs with the same names as above, with the map identified by its `fd`.

Examples

Please see the `tools/testing/selftests/bpf` directory for functional examples. The code samples below demonstrate API usage.

Kernel BPF

This snippet shows how to declare an array in a BPF program.

```
struct {
        __uint(type, BPF_MAP_TYPE_ARRAY);
        __type(key, u32);
        __type(value, long);
        __uint(max_entries, 256);
} my_map SEC(".maps");
```

This example BPF program shows how to access an array element.

```
int bpf_prog(struct __sk_buff *skb)
{
        struct iphdr ip;
        int index;
        long *value;

        if (bpf_skb_load_bytes(skb, ETH_HLEN, &ip, sizeof(ip)) < 0)
                return 0;

        index = ip.protocol;
        value = bpf_map_lookup_elem(&my_map, &index);
        if (value)
                __sync_fetch_and_add(value, skb->len);

        return 0;
}
```

Userspace

BPF_MAP_TYPE_ARRAY

This snippet shows how to create an array, using `bpf_map_create_opts` to set flags.

```
#include <bpf/libbpf.h>
#include <bpf/bpf.h>

int create_array()
{
        int fd;
        LIBBPF_OPTS(bpf_map_create_opts, opts, .map_flags = BPF_F_MMAPABLE);

        fd = bpf_map_create(BPF_MAP_TYPE_ARRAY,
                            "example_array",       /* name */
                            sizeof(__u32),         /* key size */
                            sizeof(long),          /* value size */
                            256,                   /* max entries */
                            &opts);                /* create opts */
        return fd;
}
```

This snippet shows how to initialize the elements of an array.

```
int initialize_array(int fd)
{
        __u32 i;
        long value;
        int ret;

        for (i = 0; i < 256; i++) {
                value = i;
                ret = bpf_map_update_elem(fd, &i, &value, BPF_ANY);
                if (ret < 0)
                        return ret;
        }

        return ret;
}
```

This snippet shows how to retrieve an element value from an array.

```
int lookup(int fd)
{
        __u32 index = 42;
        long value;
        int ret;

        ret = bpf_map_lookup_elem(fd, &index, &value);
        if (ret < 0)
```

```
                return ret;

        /* use value here */
        assert(value == 42);

        return ret;
}
```

BPF_MAP_TYPE_PERCPU_ARRAY

This snippet shows how to initialize the elements of a per CPU array.

```
int initialize_array(int fd)
{
        int ncpus = libbpf_num_possible_cpus();
        long values[ncpus];
        __u32 i, j;
        int ret;

        for (i = 0; i < 256 ; i++) {
                for (j = 0; j < ncpus; j++)
                        values[j] = i;
                ret = bpf_map_update_elem(fd, &i, &values, BPF_ANY);
                if (ret < 0)
                        return ret;
        }

        return ret;
}
```

This snippet shows how to access the per CPU elements of an array value.

```
int lookup(int fd)
{
        int ncpus = libbpf_num_possible_cpus();
        __u32 index = 42, j;
        long values[ncpus];
        int ret;

        ret = bpf_map_lookup_elem(fd, &index, &values);
        if (ret < 0)
                return ret;

        for (j = 0; j < ncpus; j++) {
                /* Use per CPU value here */
                assert(values[j] == 42);
        }

        return ret;
}
```

Semantics

As shown in the example above, when accessing a BPF_MAP_TYPE_PERCPU_ARRAY in userspace, each value is an array with ncpus elements.

When calling bpf_map_update_elem() the flag BPF_NOEXIST can not be used for these maps.

11.1.2 BPF_MAP_TYPE_BLOOM_FILTER

Note:

- BPF_MAP_TYPE_BLOOM_FILTER was introduced in kernel version 5.16

BPF_MAP_TYPE_BLOOM_FILTER provides a BPF bloom filter map. Bloom filters are a space-efficient probabilistic data structure used to quickly test whether an element exists in a set. In a bloom filter, false positives are possible whereas false negatives are not.

The bloom filter map does not have keys, only values. When the bloom filter map is created, it must be created with a key_size of 0. The bloom filter map supports two operations:

- push: adding an element to the map
- peek: determining whether an element is present in the map

BPF programs must use bpf_map_push_elem to add an element to the bloom filter map and bpf_map_peek_elem to query the map. These operations are exposed to userspace applications using the existing bpf syscall in the following way:

- BPF_MAP_UPDATE_ELEM -> push
- BPF_MAP_LOOKUP_ELEM -> peek

The max_entries size that is specified at map creation time is used to approximate a reasonable bitmap size for the bloom filter, and is not otherwise strictly enforced. If the user wishes to insert more entries into the bloom filter than max_entries, this may lead to a higher false positive rate.

The number of hashes to use for the bloom filter is configurable using the lower 4 bits of map_extra in union bpf_attr at map creation time. If no number is specified, the default used will be 5 hash functions. In general, using more hashes decreases both the false positive rate and the speed of a lookup.

It is not possible to delete elements from a bloom filter map. A bloom filter map may be used as an inner map. The user is responsible for synchronising concurrent updates and lookups to ensure no false negative lookups occur.

Usage

Kernel BPF

bpf_map_push_elem()

```
long bpf_map_push_elem(struct bpf_map *map, const void *value, u64 flags)
```

A `value` can be added to a bloom filter using the `bpf_map_push_elem()` helper. The `flags` parameter must be set to `BPF_ANY` when adding an entry to the bloom filter. This helper returns `0` on success, or negative error in case of failure.

bpf_map_peek_elem()

```
long bpf_map_peek_elem(struct bpf_map *map, void *value)
```

The `bpf_map_peek_elem()` helper is used to determine whether `value` is present in the bloom filter map. This helper returns `0` if `value` is probably present in the map, or `-ENOENT` if `value` is definitely not present in the map.

Userspace

bpf_map_update_elem()

```
int bpf_map_update_elem (int fd, const void *key, const void *value, __u64␣
↪flags)
```

A userspace program can add a `value` to a bloom filter using libbpf's `bpf_map_update_elem` function. The `key` parameter must be set to `NULL` and `flags` must be set to `BPF_ANY`. Returns `0` on success, or negative error in case of failure.

bpf_map_lookup_elem()

```
int bpf_map_lookup_elem (int fd, const void *key, void *value)
```

A userspace program can determine the presence of `value` in a bloom filter using libbpf's `bpf_map_lookup_elem` function. The `key` parameter must be set to `NULL`. Returns `0` if `value` is probably present in the map, or `-ENOENT` if `value` is definitely not present in the map.

Examples

Kernel BPF

This snippet shows how to declare a bloom filter in a BPF program:

```
struct {
        __uint(type, BPF_MAP_TYPE_BLOOM_FILTER);
        __type(value, __u32);
        __uint(max_entries, 1000);
        __uint(map_extra, 3);
} bloom_filter SEC(".maps");
```

This snippet shows how to determine presence of a value in a bloom filter in a BPF program:

```
void *lookup(__u32 key)
{
        if (bpf_map_peek_elem(&bloom_filter, &key) == 0) {
                /* Verify not a false positive and fetch an associated
                 * value using a secondary lookup, e.g. in a hash table
                 */
                return bpf_map_lookup_elem(&hash_table, &key);
        }
        return 0;
}
```

Userspace

This snippet shows how to use libbpf to create a bloom filter map from userspace:

```
int create_bloom()
{
        LIBBPF_OPTS(bpf_map_create_opts, opts,
                    .map_extra = 3);            /* number of hashes */

        return bpf_map_create(BPF_MAP_TYPE_BLOOM_FILTER,
                              "ipv6_bloom",     /* name */
                              0,                /* key size, must be zero */
                              sizeof(ipv6_addr), /* value size */
                              10000,            /* max entries */
                              &opts);           /* create options */
}
```

This snippet shows how to add an element to a bloom filter from userspace:

```
int add_element(struct bpf_map *bloom_map, __u32 value)
{
        int bloom_fd = bpf_map__fd(bloom_map);
        return bpf_map_update_elem(bloom_fd, NULL, &value, BPF_ANY);
}
```

References

https://lwn.net/ml/bpf/20210831225005.2762202-1-joannekoong@fb.com/

11.1.3 BPF_MAP_TYPE_CGROUP_STORAGE

The BPF_MAP_TYPE_CGROUP_STORAGE map type represents a local fix-sized storage. It is only available with CONFIG_CGROUP_BPF, and to programs that attach to cgroups; the programs are made available by the same Kconfig. The storage is identified by the cgroup the program is attached to.

The map provide a local storage at the cgroup that the BPF program is attached to. It provides a faster and simpler access than the general purpose hash table, which performs a hash table lookups, and requires user to track live cgroups on their own.

This document describes the usage and semantics of the BPF_MAP_TYPE_CGROUP_STORAGE map type. Some of its behaviors was changed in Linux 5.9 and this document will describe the differences.

Usage

The map uses key of type of either `__u64 cgroup_inode_id` or `struct bpf_cgroup_storage_key`, declared in `linux/bpf.h`:

```
struct bpf_cgroup_storage_key {
        __u64 cgroup_inode_id;
        __u32 attach_type;
};
```

`cgroup_inode_id` is the inode id of the cgroup directory. `attach_type` is the program's attach type.

Linux 5.9 added support for type `__u64 cgroup_inode_id` as the key type. When this key type is used, then all attach types of the particular cgroup and map will share the same storage. Otherwise, if the type is `struct bpf_cgroup_storage_key`, then programs of different attach types be isolated and see different storages.

To access the storage in a program, use `bpf_get_local_storage`:

```
void *bpf_get_local_storage(void *map, u64 flags)
```

`flags` is reserved for future use and must be 0.

There is no implicit synchronization. Storages of BPF_MAP_TYPE_CGROUP_STORAGE can be accessed by multiple programs across different CPUs, and user should take care of synchronization by themselves. The bpf infrastructure provides `struct bpf_spin_lock` to synchronize the storage. See `tools/testing/selftests/bpf/progs/test_spin_lock.c`.

Examples

Usage with key type as `struct bpf_cgroup_storage_key`:

```
#include <bpf/bpf.h>

struct {
        __uint(type, BPF_MAP_TYPE_CGROUP_STORAGE);
        __type(key, struct bpf_cgroup_storage_key);
        __type(value, __u32);
} cgroup_storage SEC(".maps");

int program(struct __sk_buff *skb)
{
        __u32 *ptr = bpf_get_local_storage(&cgroup_storage, 0);
        __sync_fetch_and_add(ptr, 1);

        return 0;
}
```

Userspace accessing map declared above:

```
#include <linux/bpf.h>
#include <linux/libbpf.h>

__u32 map_lookup(struct bpf_map *map, __u64 cgrp, enum bpf_attach_type type)
{
        struct bpf_cgroup_storage_key = {
                .cgroup_inode_id = cgrp,
                .attach_type = type,
        };
        __u32 value;
        bpf_map_lookup_elem(bpf_map__fd(map), &key, &value);
        // error checking omitted
        return value;
}
```

Alternatively, using just `__u64 cgroup_inode_id` as key type:

```
#include <bpf/bpf.h>

struct {
        __uint(type, BPF_MAP_TYPE_CGROUP_STORAGE);
        __type(key, __u64);
        __type(value, __u32);
} cgroup_storage SEC(".maps");

int program(struct __sk_buff *skb)
{
        __u32 *ptr = bpf_get_local_storage(&cgroup_storage, 0);
        __sync_fetch_and_add(ptr, 1);
```

```
        return 0;
}
```

And userspace:

```
#include <linux/bpf.h>
#include <linux/libbpf.h>

__u32 map_lookup(struct bpf_map *map, __u64 cgrp, enum bpf_attach_type type)
{
        __u32 value;
        bpf_map_lookup_elem(bpf_map__fd(map), &cgrp, &value);
        // error checking omitted
        return value;
}
```

Semantics

BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE is a variant of this map type. This per-CPU variant will have different memory regions for each CPU for each storage. The non-per-CPU will have the same memory region for each storage.

Prior to Linux 5.9, the lifetime of a storage is precisely per-attachment, and for a single CGROUP_STORAGE map, there can be at most one program loaded that uses the map. A program may be attached to multiple cgroups or have multiple attach types, and each attach creates a fresh zeroed storage. The storage is freed upon detach.

There is a one-to-one association between the map of each type (per-CPU and non-per-CPU) and the BPF program during load verification time. As a result, each map can only be used by one BPF program and each BPF program can only use one storage map of each type. Because of map can only be used by one BPF program, sharing of this cgroup's storage with other BPF programs were impossible.

Since Linux 5.9, storage can be shared by multiple programs. When a program is attached to a cgroup, the kernel would create a new storage only if the map does not already contain an entry for the cgroup and attach type pair, or else the old storage is reused for the new attachment. If the map is attach type shared, then attach type is simply ignored during comparison. Storage is freed only when either the map or the cgroup attached to is being freed. Detaching will not directly free the storage, but it may cause the reference to the map to reach zero and indirectly freeing all storage in the map.

The map is not associated with any BPF program, thus making sharing possible. However, the BPF program can still only associate with one map of each type (per-CPU and non-per-CPU). A BPF program cannot use more than one BPF_MAP_TYPE_CGROUP_STORAGE or more than one BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE.

In all versions, userspace may use the attach parameters of cgroup and attach type pair in struct bpf_cgroup_storage_key as the key to the BPF map APIs to read or update the storage for a given attachment. For Linux 5.9 attach type shared storages, only the first value in the struct, cgroup inode id, is used during comparison, so userspace may just specify a __u64 directly.

The storage is bound at attach time. Even if the program is attached to parent and triggers in child, the storage still belongs to the parent.

Userspace cannot create a new entry in the map or delete an existing entry. Program test runs always use a temporary storage.

11.1.4 BPF_MAP_TYPE_CGRP_STORAGE

The BPF_MAP_TYPE_CGRP_STORAGE map type represents a local fix-sized storage for cgroups. It is only available with CONFIG_CGROUPS. The programs are made available by the same Kconfig. The data for a particular cgroup can be retrieved by looking up the map with that cgroup.

This document describes the usage and semantics of the BPF_MAP_TYPE_CGRP_STORAGE map type.

Usage

The map key must be sizeof(int) representing a cgroup fd. To access the storage in a program, use bpf_cgrp_storage_get:

```
void *bpf_cgrp_storage_get(struct bpf_map *map, struct cgroup *cgroup, void␣
↪*value, u64 flags)
```

flags could be 0 or BPF_LOCAL_STORAGE_GET_F_CREATE which indicates that a new local storage will be created if one does not exist.

The local storage can be removed with bpf_cgrp_storage_delete:

```
long bpf_cgrp_storage_delete(struct bpf_map *map, struct cgroup *cgroup)
```

The map is available to all program types.

Examples

A BPF program example with BPF_MAP_TYPE_CGRP_STORAGE:

```
#include <vmlinux.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

struct {
        __uint(type, BPF_MAP_TYPE_CGRP_STORAGE);
        __uint(map_flags, BPF_F_NO_PREALLOC);
        __type(key, int);
        __type(value, long);
} cgrp_storage SEC(".maps");

SEC("tp_btf/sys_enter")
int BPF_PROG(on_enter, struct pt_regs *regs, long id)
{
        struct task_struct *task = bpf_get_current_task_btf();
```

```
        long *ptr;

        ptr = bpf_cgrp_storage_get(&cgrp_storage, task->cgroups->dfl_cgrp, 0,
                                   BPF_LOCAL_STORAGE_GET_F_CREATE);
        if (ptr)
            __sync_fetch_and_add(ptr, 1);

        return 0;
}
```

Userspace accessing map declared above:

```
#include <linux/bpf.h>
#include <linux/libbpf.h>

__u32 map_lookup(struct bpf_map *map, int cgrp_fd)
{
        __u32 *value;
        value = bpf_map_lookup_elem(bpf_map__fd(map), &cgrp_fd);
        if (value)
            return *value;
        return 0;
}
```

Difference Between BPF_MAP_TYPE_CGRP_STORAGE and BPF_MAP_TYPE_CGROUP_STORAGE

The old cgroup storage map `BPF_MAP_TYPE_CGROUP_STORAGE` has been marked as deprecated (renamed to `BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED`). The new `BPF_MAP_TYPE_CGRP_STORAGE` map should be used instead. The following illusates the main difference between `BPF_MAP_TYPE_CGRP_STORAGE` and `BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED`.

(1). `BPF_MAP_TYPE_CGRP_STORAGE` can be used by all program types while `BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED` is available only to cgroup program types like BPF_CGROUP_INET_INGRESS or BPF_CGROUP_SOCK_OPS, etc.

(2). `BPF_MAP_TYPE_CGRP_STORAGE` supports local storage for more than one cgroup while `BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED` only supports one cgroup which is attached by a BPF program.

(3). `BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED` allocates local storage at attach time so `bpf_get_local_storage()` always returns non-NULL local storage. `BPF_MAP_TYPE_CGRP_STORAGE` allocates local storage at runtime so it is possible that `bpf_cgrp_storage_get()` may return null local storage. To avoid such null local storage issue, user space can do `bpf_map_update_elem()` to pre-allocate local storage before a BPF program is attached.

(4). `BPF_MAP_TYPE_CGRP_STORAGE` supports deleting local storage by a BPF program while `BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED` only deletes storage during prog detach time.

So overall, `BPF_MAP_TYPE_CGRP_STORAGE` supports all `BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED`

functionality and beyond. It is recommended to use BPF_MAP_TYPE_CGRP_STORAGE instead of BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED.

11.1.5 BPF_MAP_TYPE_CPUMAP

Note:

- BPF_MAP_TYPE_CPUMAP was introduced in kernel version 4.15

The 'cpumap' is primarily used as a backend map for XDP BPF helper call bpf_redirect_map() and XDP_REDIRECT action, like 'devmap'.

Unlike devmap which redirects XDP frames out to another NIC device, this map type redirects raw XDP frames to another CPU. The remote CPU will do SKB-allocation and call the normal network stack.

An example use-case for this map type is software based Receive Side Scaling (RSS).

The CPUMAP represents the CPUs in the system indexed as the map-key, and the map-value is the config setting (per CPUMAP entry). Each CPUMAP entry has a dedicated kernel thread bound to the given CPU to represent the remote CPU execution unit.

Starting from Linux kernel version 5.9 the CPUMAP can run a second XDP program on the remote CPU. This allows an XDP program to split its processing across multiple CPUs. For example, a scenario where the initial CPU (that sees/receives the packets) needs to do minimal packet processing and the remote CPU (to which the packet is directed) can afford to spend more cycles processing the frame. The initial CPU is where the XDP redirect program is executed. The remote CPU receives raw xdp_frame objects.

Usage

Kernel BPF

bpf_redirect_map()

```
long bpf_redirect_map(struct bpf_map *map, u32 key, u64 flags)
```

Redirect the packet to the endpoint referenced by map at index key. For BPF_MAP_TYPE_CPUMAP this map contains references to CPUs.

The lower two bits of flags are used as the return code if the map lookup fails. This is so that the return value can be one of the XDP program return codes up to XDP_TX, as chosen by the caller.

User space

Note: CPUMAP entries can only be updated/looked up/deleted from user space and not from an eBPF program. Trying to call these functions from a kernel eBPF program will result in the program failing to load and a verifier warning.

bpf_map_update_elem()

```
int bpf_map_update_elem(int fd, const void *key, const void *value, __u64␣
↪flags);
```

CPU entries can be added or updated using the `bpf_map_update_elem()` helper. This helper replaces existing elements atomically. The `value` parameter can be `struct bpf_cpumap_val`.

```
struct bpf_cpumap_val {
    __u32 qsize;  /* queue size to remote target CPU */
    union {
        int   fd; /* prog fd on map write */
        __u32 id; /* prog id on map read */
    } bpf_prog;
};
```

The flags argument can be one of the following:

- BPF_ANY: Create a new element or update an existing element.
- BPF_NOEXIST: Create a new element only if it did not exist.
- BPF_EXIST: Update an existing element.

bpf_map_lookup_elem()

```
int bpf_map_lookup_elem(int fd, const void *key, void *value);
```

CPU entries can be retrieved using the `bpf_map_lookup_elem()` helper.

bpf_map_delete_elem()

```
int bpf_map_delete_elem(int fd, const void *key);
```

CPU entries can be deleted using the `bpf_map_delete_elem()` helper. This helper will return 0 on success, or negative error in case of failure.

Examples

Kernel

The following code snippet shows how to declare a BPF_MAP_TYPE_CPUMAP called cpu_map and how to redirect packets to a remote CPU using a round robin scheme.

```
struct {
     __uint(type, BPF_MAP_TYPE_CPUMAP);
     __type(key, __u32);
     __type(value, struct bpf_cpumap_val);
     __uint(max_entries, 12);
 } cpu_map SEC(".maps");

 struct {
     __uint(type, BPF_MAP_TYPE_ARRAY);
     __type(key, __u32);
     __type(value, __u32);
     __uint(max_entries, 12);
 } cpus_available SEC(".maps");

 struct {
     __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
     __type(key, __u32);
     __type(value, __u32);
     __uint(max_entries, 1);
 } cpus_iterator SEC(".maps");

 SEC("xdp")
 int  xdp_redir_cpu_round_robin(struct xdp_md *ctx)
 {
     __u32 key = 0;
     __u32 cpu_dest = 0;
     __u32 *cpu_selected, *cpu_iterator;
     __u32 cpu_idx;

     cpu_iterator = bpf_map_lookup_elem(&cpus_iterator, &key);
     if (!cpu_iterator)
         return XDP_ABORTED;
     cpu_idx = *cpu_iterator;

     *cpu_iterator += 1;
     if (*cpu_iterator == bpf_num_possible_cpus())
         *cpu_iterator = 0;

     cpu_selected = bpf_map_lookup_elem(&cpus_available, &cpu_idx);
     if (!cpu_selected)
         return XDP_ABORTED;
     cpu_dest = *cpu_selected;

     if (cpu_dest >= bpf_num_possible_cpus())
```

```
        return XDP_ABORTED;

    return bpf_redirect_map(&cpu_map, cpu_dest, 0);
}
```

User space

The following code snippet shows how to dynamically set the max_entries for a CPUMAP to the max number of cpus available on the system.

```
int set_max_cpu_entries(struct bpf_map *cpu_map)
{
    if (bpf_map__set_max_entries(cpu_map, libbpf_num_possible_cpus()) < 0) {
        fprintf(stderr, "Failed to set max entries for cpu_map map: %s",
            strerror(errno));
        return -1;
    }
    return 0;
}
```

References

- https://developers.redhat.com/blog/2021/05/13/receive-side-scaling-rss-with-ebpf-and-cpumap#redirecting_into_a_cpumap

11.1.6 BPF_MAP_TYPE_DEVMAP and BPF_MAP_TYPE_DEVMAP_HASH

Note:

- `BPF_MAP_TYPE_DEVMAP` was introduced in kernel version 4.14
- `BPF_MAP_TYPE_DEVMAP_HASH` was introduced in kernel version 5.4

`BPF_MAP_TYPE_DEVMAP` and `BPF_MAP_TYPE_DEVMAP_HASH` are BPF maps primarily used as backend maps for the XDP BPF helper call `bpf_redirect_map()`. `BPF_MAP_TYPE_DEVMAP` is backed by an array that uses the key as the index to lookup a reference to a net device. While `BPF_MAP_TYPE_DEVMAP_HASH` is backed by a hash table that uses a key to lookup a reference to a net device. The user provides either `<key/ ifindex>` or `<key/ struct bpf_devmap_val>` pairs to update the maps with new net devices.

Note:

- The key to a hash map doesn't have to be an `ifindex`.
- While `BPF_MAP_TYPE_DEVMAP_HASH` allows for densely packing the net devices it comes at the cost of a hash of the key when performing a look up.

The setup and packet enqueue/send code is shared between the two types of devmap; only the lookup and insertion is different.

Usage

Kernel BPF

bpf_redirect_map()

```
long bpf_redirect_map(struct bpf_map *map, u32 key, u64 flags)
```

Redirect the packet to the endpoint referenced by `map` at index `key`. For `BPF_MAP_TYPE_DEVMAP` and `BPF_MAP_TYPE_DEVMAP_HASH` this map contains references to net devices (for forwarding packets through other ports).

The lower two bits of *flags* are used as the return code if the map lookup fails. This is so that the return value can be one of the XDP program return codes up to `XDP_TX`, as chosen by the caller. The higher bits of `flags` can be set to `BPF_F_BROADCAST` or `BPF_F_EXCLUDE_INGRESS` as defined below.

With `BPF_F_BROADCAST` the packet will be broadcast to all the interfaces in the map, with `BPF_F_EXCLUDE_INGRESS` the ingress interface will be excluded from the broadcast.

Note:

- The key is ignored if BPF_F_BROADCAST is set.
- The broadcast feature can also be used to implement multicast forwarding: simply create multiple DEVMAPs, each one corresponding to a single multicast group.

This helper will return `XDP_REDIRECT` on success, or the value of the two lower bits of the `flags` argument if the map lookup fails.

More information about redirection can be found *Redirect*

bpf_map_lookup_elem()

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

Net device entries can be retrieved using the `bpf_map_lookup_elem()` helper.

User space

Note: DEVMAP entries can only be updated/deleted from user space and not from an eBPF program. Trying to call these functions from a kernel eBPF program will result in the program failing to load and a verifier warning.

---

bpf_map_update_elem()

```
int bpf_map_update_elem(int fd, const void *key, const void *value, __u64␣
↪flags);
```

Net device entries can be added or updated using the `bpf_map_update_elem()` helper. This helper replaces existing elements atomically. The `value` parameter can be `struct bpf_devmap_val` or a simple `int ifindex` for backwards compatibility.

```
struct bpf_devmap_val {
    __u32 ifindex;   /* device index */
    union {
        int   fd;  /* prog fd on map write */
        __u32 id;  /* prog id on map read */
    } bpf_prog;
};
```

The `flags` argument can be one of the following:

- `BPF_ANY`: Create a new element or update an existing element.
- `BPF_NOEXIST`: Create a new element only if it did not exist.
- `BPF_EXIST`: Update an existing element.

DEVMAPs can associate a program with a device entry by adding a `bpf_prog.fd` to `struct bpf_devmap_val`. Programs are run after `XDP_REDIRECT` and have access to both Rx device and Tx device. The program associated with the `fd` must have type XDP with expected attach type `xdp_devmap`. When a program is associated with a device index, the program is run on an `XDP_REDIRECT` and before the buffer is added to the per-cpu queue. Examples of how to attach/use xdp_devmap progs can be found in the kernel selftests:

- `tools/testing/selftests/bpf/prog_tests/xdp_devmap_attach.c`
- `tools/testing/selftests/bpf/progs/test_xdp_with_devmap_helpers.c`

bpf_map_lookup_elem()

int **bpf_map_lookup_elem**(int fd, const void *key, void *value);

Net device entries can be retrieved using the `bpf_map_lookup_elem()` helper.

bpf_map_delete_elem()

int **bpf_map_delete_elem**(int fd, const void *key);

Net device entries can be deleted using the `bpf_map_delete_elem()` helper. This helper will return 0 on success, or negative error in case of failure.

Examples

Kernel BPF

The following code snippet shows how to declare a `BPF_MAP_TYPE_DEVMAP` called tx_port.

```
struct {
    __uint(type, BPF_MAP_TYPE_DEVMAP);
    __type(key, __u32);
    __type(value, __u32);
    __uint(max_entries, 256);
} tx_port SEC(".maps");
```

The following code snippet shows how to declare a `BPF_MAP_TYPE_DEVMAP_HASH` called forward_map.

```
struct {
    __uint(type, BPF_MAP_TYPE_DEVMAP_HASH);
    __type(key, __u32);
    __type(value, struct bpf_devmap_val);
    __uint(max_entries, 32);
} forward_map SEC(".maps");
```

Note: The value type in the DEVMAP above is a `struct bpf_devmap_val`

The following code snippet shows a simple xdp_redirect_map program. This program would work with a user space program that populates the devmap `forward_map` based on ingress ifindexes. The BPF program (below) is redirecting packets using the ingress `ifindex` as the key.

```
SEC("xdp")
int xdp_redirect_map_func(struct xdp_md *ctx)
{
    int index = ctx->ingress_ifindex;

    return bpf_redirect_map(&forward_map, index, 0);
}
```

The following code snippet shows a BPF program that is broadcasting packets to all the interfaces in the `tx_port` devmap.

```
SEC("xdp")
int xdp_redirect_map_func(struct xdp_md *ctx)
{
    return bpf_redirect_map(&tx_port, 0, BPF_F_BROADCAST | BPF_F_EXCLUDE_
↪INGRESS);
}
```

User space

The following code snippet shows how to update a devmap called `tx_port`.

```
int update_devmap(int ifindex, int redirect_ifindex)
{
    int ret;

    ret = bpf_map_update_elem(bpf_map__fd(tx_port), &ifindex, &redirect_
↪ifindex, 0);
    if (ret < 0) {
        fprintf(stderr, "Failed to update devmap_ value: %s\n",
            strerror(errno));
    }

    return ret;
}
```

The following code snippet shows how to update a hash_devmap called `forward_map`.

```
int update_devmap(int ifindex, int redirect_ifindex)
{
    struct bpf_devmap_val devmap_val = { .ifindex = redirect_ifindex };
    int ret;

    ret = bpf_map_update_elem(bpf_map__fd(forward_map), &ifindex, &devmap_val,␣
↪0);
    if (ret < 0) {
        fprintf(stderr, "Failed to update devmap_ value: %s\n",
            strerror(errno));
    }
```

```
    return ret;
}
```

References

- https://lwn.net/Articles/728146/
- https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=6f9d451ab1a33728adb72d7ff66a7b374d665176
- https://elixir.bootlin.com/linux/latest/source/net/core/filter.c#L4106

11.1.7 BPF_MAP_TYPE_HASH, with PERCPU and LRU Variants

Note:

- `BPF_MAP_TYPE_HASH` was introduced in kernel version 3.19
- `BPF_MAP_TYPE_PERCPU_HASH` was introduced in version 4.6
- Both `BPF_MAP_TYPE_LRU_HASH` and `BPF_MAP_TYPE_LRU_PERCPU_HASH` were introduced in version 4.10

`BPF_MAP_TYPE_HASH` and `BPF_MAP_TYPE_PERCPU_HASH` provide general purpose hash map storage. Both the key and the value can be structs, allowing for composite keys and values.

The kernel is responsible for allocating and freeing key/value pairs, up to the max_entries limit that you specify. Hash maps use pre-allocation of hash table elements by default. The `BPF_F_NO_PREALLOC` flag can be used to disable pre-allocation when it is too memory expensive.

`BPF_MAP_TYPE_PERCPU_HASH` provides a separate value slot per CPU. The per-cpu values are stored internally in an array.

The `BPF_MAP_TYPE_LRU_HASH` and `BPF_MAP_TYPE_LRU_PERCPU_HASH` variants add LRU semantics to their respective hash tables. An LRU hash will automatically evict the least recently used entries when the hash table reaches capacity. An LRU hash maintains an internal LRU list that is used to select elements for eviction. This internal LRU list is shared across CPUs but it is possible to request a per CPU LRU list with the `BPF_F_NO_COMMON_LRU` flag when calling `bpf_map_create`. The following table outlines the properties of LRU maps depending on the a map type and the flags used to create the map.

Flag	BPF_MAP_TYPE_LRU_HASH	BPF_MAP_TYPE_LRU_PERCPU_HASH
BPF_F_NO_COMMON_LRU	Per-CPU LRU, global map	Per-CPU LRU, per-cpu map
!BPF_F_NO_COMMON_LRU	Global LRU, global map	Global LRU, per-cpu map

Usage

Kernel BPF

bpf_map_update_elem()

```
long bpf_map_update_elem(struct bpf_map *map, const void *key, const void␣
↪*value, u64 flags)
```

Hash entries can be added or updated using the `bpf_map_update_elem()` helper. This helper replaces existing elements atomically. The `flags` parameter can be used to control the update behaviour:

- `BPF_ANY` will create a new element or update an existing element
- `BPF_NOEXIST` will create a new element only if one did not already exist
- `BPF_EXIST` will update an existing element

`bpf_map_update_elem()` returns 0 on success, or negative error in case of failure.

bpf_map_lookup_elem()

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

Hash entries can be retrieved using the `bpf_map_lookup_elem()` helper. This helper returns a pointer to the value associated with `key`, or `NULL` if no entry was found.

bpf_map_delete_elem()

```
long bpf_map_delete_elem(struct bpf_map *map, const void *key)
```

Hash entries can be deleted using the `bpf_map_delete_elem()` helper. This helper will return 0 on success, or negative error in case of failure.

Per CPU Hashes

For `BPF_MAP_TYPE_PERCPU_HASH` and `BPF_MAP_TYPE_LRU_PERCPU_HASH` the `bpf_map_update_elem()` and `bpf_map_lookup_elem()` helpers automatically access the hash slot for the current CPU.

bpf_map_lookup_percpu_elem()

```
void *bpf_map_lookup_percpu_elem(struct bpf_map *map, const void *key, u32 cpu)
```

The `bpf_map_lookup_percpu_elem()` helper can be used to lookup the value in the hash slot for a specific CPU. Returns value associated with `key` on `cpu` , or `NULL` if no entry was found or `cpu` is invalid.

Concurrency

Values stored in `BPF_MAP_TYPE_HASH` can be accessed concurrently by programs running on different CPUs. Since Kernel version 5.1, the BPF infrastructure provides `struct bpf_spin_lock` to synchronise access. See `tools/testing/selftests/bpf/progs/test_spin_lock.c`.

Userspace

bpf_map_get_next_key()

```
int bpf_map_get_next_key(int fd, const void *cur_key, void *next_key)
```

In userspace, it is possible to iterate through the keys of a hash using libbpf's `bpf_map_get_next_key()` function. The first key can be fetched by calling `bpf_map_get_next_key()` with `cur_key` set to `NULL`. Subsequent calls will fetch the next key that follows the current key. `bpf_map_get_next_key()` returns 0 on success, -ENOENT if cur_key is the last key in the hash, or negative error in case of failure.

Note that if `cur_key` gets deleted then `bpf_map_get_next_key()` will instead return the *first* key in the hash table which is undesirable. It is recommended to use batched lookup if there is going to be key deletion intermixed with `bpf_map_get_next_key()`.

Examples

Please see the `tools/testing/selftests/bpf` directory for functional examples. The code snippets below demonstrates API usage.

This example shows how to declare an LRU Hash with a struct key and a struct value.

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct key {
    __u32 srcip;
};

struct value {
    __u64 packets;
    __u64 bytes;
};
```

```
struct {
        __uint(type, BPF_MAP_TYPE_LRU_HASH);
        __uint(max_entries, 32);
        __type(key, struct key);
        __type(value, struct value);
} packet_stats SEC(".maps");
```

This example shows how to create or update hash values using atomic instructions:

```
static void update_stats(__u32 srcip, int bytes)
{
        struct key key = {
                .srcip = srcip,
        };
        struct value *value = bpf_map_lookup_elem(&packet_stats, &key);

        if (value) {
                __sync_fetch_and_add(&value->packets, 1);
                __sync_fetch_and_add(&value->bytes, bytes);
        } else {
                struct value newval = { 1, bytes };

                bpf_map_update_elem(&packet_stats, &key, &newval, BPF_NOEXIST);
        }
}
```

Userspace walking the map elements from the map declared above:

```
#include <bpf/libbpf.h>
#include <bpf/bpf.h>

static void walk_hash_elements(int map_fd)
{
        struct key *cur_key = NULL;
        struct key next_key;
        struct value value;
        int err;

        for (;;) {
                err = bpf_map_get_next_key(map_fd, cur_key, &next_key);
                if (err)
                        break;

                bpf_map_lookup_elem(map_fd, &next_key, &value);

                // Use key and value here

                cur_key = &next_key;
        }
}
```

Internals

This section of the document is targeted at Linux developers and describes aspects of the map implementations that are not considered stable ABI. The following details are subject to change in future versions of the kernel.

`BPF_MAP_TYPE_LRU_HASH` and variants

Updating elements in LRU maps may trigger eviction behaviour when the capacity of the map is reached. There are various steps that the update algorithm attempts in order to enforce the LRU property which have increasing impacts on other CPUs involved in the following operation attempts:

- Attempt to use CPU-local state to batch operations
- Attempt to fetch free nodes from global lists
- Attempt to pull any node from a global list and remove it from the hashmap
- Attempt to pull any node from any CPU's list and remove it from the hashmap

This algorithm is described visually in the following diagram. See the description in commit 3a08c2fd7634 ("bpf: LRU List") for a full explanation of the corresponding operations:

Map updates start from the oval in the top right "begin `bpf_map_update()`" and progress through the graph towards the bottom where the result may be either a successful update or a failure with various error codes. The key in the top right provides indicators for which locks may be involved in specific operations. This is intended as a visual hint for reasoning about how map contention may impact update operations, though the map type and flags may impact the actual contention on those locks, based on the logic described in the table above. For instance, if the map is created with type `BPF_MAP_TYPE_LRU_PERCPU_HASH` and flags `BPF_F_NO_COMMON_LRU` then all map properties would be per-cpu.

11.1.8 BPF_MAP_TYPE_LPM_TRIE

Note:

- `BPF_MAP_TYPE_LPM_TRIE` was introduced in kernel version 4.11

`BPF_MAP_TYPE_LPM_TRIE` provides a longest prefix match algorithm that can be used to match IP addresses to a stored set of prefixes. Internally, data is stored in an unbalanced trie of nodes that uses `prefixlen,data` pairs as its keys. The `data` is interpreted in network byte order, i.e. big endian, so `data[0]` stores the most significant byte.

LPM tries may be created with a maximum prefix length that is a multiple of 8, in the range from 8 to 2048. The key used for lookup and update operations is a `struct bpf_lpm_trie_key`, extended by `max_prefixlen/8` bytes.

- For IPv4 addresses the data length is 4 bytes
- For IPv6 addresses the data length is 16 bytes

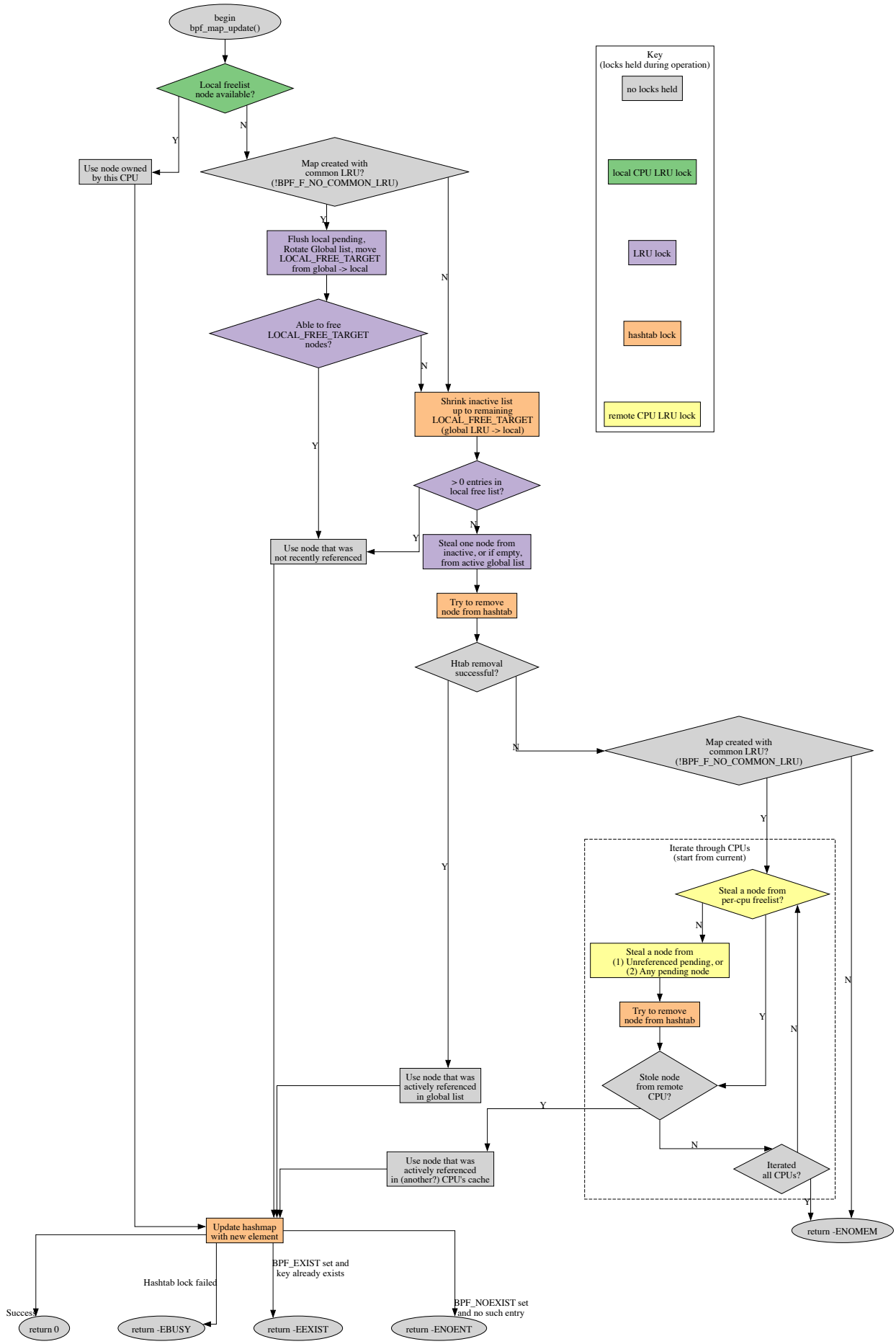


Fig. 1: LRU hash eviction during map update for `BPF_MAP_TYPE_LRU_HASH` and variants. See the dot file source for kernel function name code references.

The value type stored in the LPM trie can be any user defined type.

Note: When creating a map of type `BPF_MAP_TYPE_LPM_TRIE` you must set the `BPF_F_NO_PREALLOC` flag.

Usage

Kernel BPF

bpf_map_lookup_elem()

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

The longest prefix entry for a given data value can be found using the `bpf_map_lookup_elem()` helper. This helper returns a pointer to the value associated with the longest matching `key`, or `NULL` if no entry was found.

The `key` should have `prefixlen` set to `max_prefixlen` when performing longest prefix lookups. For example, when searching for the longest prefix match for an IPv4 address, `prefixlen` should be set to 32.

bpf_map_update_elem()

```
long bpf_map_update_elem(struct bpf_map *map, const void *key, const void␣
↪*value, u64 flags)
```

Prefix entries can be added or updated using the `bpf_map_update_elem()` helper. This helper replaces existing elements atomically.

`bpf_map_update_elem()` returns `0` on success, or negative error in case of failure.

> **Note:** The flags parameter must be one of BPF_ANY, BPF_NOEXIST or BPF_EXIST, but the value is ignored, giving BPF_ANY semantics.

bpf_map_delete_elem()

```
long bpf_map_delete_elem(struct bpf_map *map, const void *key)
```

Prefix entries can be deleted using the `bpf_map_delete_elem()` helper. This helper will return 0 on success, or negative error in case of failure.

Userspace

Access from userspace uses libbpf APIs with the same names as above, with the map identified by `fd`.

bpf_map_get_next_key()

```
int bpf_map_get_next_key (int fd, const void *cur_key, void *next_key)
```

A userspace program can iterate through the entries in an LPM trie using libbpf's `bpf_map_get_next_key()` function. The first key can be fetched by calling `bpf_map_get_next_key()` with `cur_key` set to `NULL`. Subsequent calls will fetch the next key that follows the current key. `bpf_map_get_next_key()` returns `0` on success, `-ENOENT` if `cur_key` is the last key in the trie, or negative error in case of failure.

`bpf_map_get_next_key()` will iterate through the LPM trie elements from leftmost leaf first. This means that iteration will return more specific keys before less specific ones.

Examples

Please see `tools/testing/selftests/bpf/test_lpm_map.c` for examples of LPM trie usage from userspace. The code snippets below demonstrate API usage.

Kernel BPF

The following BPF code snippet shows how to declare a new LPM trie for IPv4 address prefixes:

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct ipv4_lpm_key {
        __u32 prefixlen;
        __u32 data;
};

struct {
        __uint(type, BPF_MAP_TYPE_LPM_TRIE);
        __type(key, struct ipv4_lpm_key);
        __type(value, __u32);
        __uint(map_flags, BPF_F_NO_PREALLOC);
        __uint(max_entries, 255);
} ipv4_lpm_map SEC(".maps");
```

The following BPF code snippet shows how to lookup by IPv4 address:

```
void *lookup(__u32 ipaddr)
{
        struct ipv4_lpm_key key = {
                .prefixlen = 32,
```

```
                .data = ipaddr
        };

        return bpf_map_lookup_elem(&ipv4_lpm_map, &key);
}
```

Userspace

The following snippet shows how to insert an IPv4 prefix entry into an LPM trie:

```
int add_prefix_entry(int lpm_fd, __u32 addr, __u32 prefixlen, struct value␣
↪*value)
{
        struct ipv4_lpm_key ipv4_key = {
                .prefixlen = prefixlen,
                .data = addr
        };
        return bpf_map_update_elem(lpm_fd, &ipv4_key, value, BPF_ANY);
}
```

The following snippet shows a userspace program walking through the entries of an LPM trie:

```
#include <bpf/libbpf.h>
#include <bpf/bpf.h>

void iterate_lpm_trie(int map_fd)
{
        struct ipv4_lpm_key *cur_key = NULL;
        struct ipv4_lpm_key next_key;
        struct value value;
        int err;

        for (;;) {
                err = bpf_map_get_next_key(map_fd, cur_key, &next_key);
                if (err)
                        break;

                bpf_map_lookup_elem(map_fd, &next_key, &value);

                /* Use key and value here */

                cur_key = &next_key;
        }
}
```

11.1.9 BPF_MAP_TYPE_ARRAY_OF_MAPS and BPF_MAP_TYPE_HASH_OF_MAPS

Note:

- BPF_MAP_TYPE_ARRAY_OF_MAPS and BPF_MAP_TYPE_HASH_OF_MAPS were introduced in kernel version 4.12

BPF_MAP_TYPE_ARRAY_OF_MAPS and BPF_MAP_TYPE_HASH_OF_MAPS provide general purpose support for map in map storage. One level of nesting is supported, where an outer map contains instances of a single type of inner map, for example array_of_maps->sock_map.

When creating an outer map, an inner map instance is used to initialize the metadata that the outer map holds about its inner maps. This inner map has a separate lifetime from the outer map and can be deleted after the outer map has been created.

The outer map supports element lookup, update and delete from user space using the syscall API. A BPF program is only allowed to do element lookup in the outer map.

Note:

- Multi-level nesting is not supported.
- Any BPF map type can be used as an inner map, except for BPF_MAP_TYPE_PROG_ARRAY.
- A BPF program cannot update or delete outer map entries.

For BPF_MAP_TYPE_ARRAY_OF_MAPS the key is an unsigned 32-bit integer index into the array. The array is a fixed size with max_entries elements that are zero initialized when created.

For BPF_MAP_TYPE_HASH_OF_MAPS the key type can be chosen when defining the map. The kernel is responsible for allocating and freeing key/value pairs, up to the max_entries limit that you specify. Hash maps use pre-allocation of hash table elements by default. The BPF_F_NO_PREALLOC flag can be used to disable pre-allocation when it is too memory expensive.

Usage

Kernel BPF Helper

bpf_map_lookup_elem()

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

Inner maps can be retrieved using the bpf_map_lookup_elem() helper. This helper returns a pointer to the inner map, or NULL if no entry was found.

Examples

Kernel BPF Example

This snippet shows how to create and initialise an array of devmaps in a BPF program. Note that the outer array can only be modified from user space using the syscall API.

```
struct inner_map {
        __uint(type, BPF_MAP_TYPE_DEVMAP);
        __uint(max_entries, 10);
        __type(key, __u32);
        __type(value, __u32);
} inner_map1 SEC(".maps"), inner_map2 SEC(".maps");

struct {
        __uint(type, BPF_MAP_TYPE_ARRAY_OF_MAPS);
        __uint(max_entries, 2);
        __type(key, __u32);
        __array(values, struct inner_map);
} outer_map SEC(".maps") = {
        .values = { &inner_map1,
                    &inner_map2 }
};
```

See `progs/test_btf_map_in_map.c` in `tools/testing/selftests/bpf` for more examples of declarative initialisation of outer maps.

User Space

This snippet shows how to create an array based outer map:

```
int create_outer_array(int inner_fd) {
        LIBBPF_OPTS(bpf_map_create_opts, opts, .inner_map_fd = inner_fd);
        int fd;

        fd = bpf_map_create(BPF_MAP_TYPE_ARRAY_OF_MAPS,
                            "example_array",       /* name */
                            sizeof(__u32),         /* key size */
                            sizeof(__u32),         /* value size */
                            256,                   /* max entries */
                            &opts);                /* create opts */
        return fd;
}
```

This snippet shows how to add an inner map to an outer map:

```
int add_devmap(int outer_fd, int index, const char *name) {
        int fd;

        fd = bpf_map_create(BPF_MAP_TYPE_DEVMAP, name,
                            sizeof(__u32), sizeof(__u32), 256, NULL);
```

```
        if (fd < 0)
                return fd;

        return bpf_map_update_elem(outer_fd, &index, &fd, BPF_ANY);
}
```

References

- https://lore.kernel.org/netdev/20170322170035.923581-3-kafai@fb.com/
- https://lore.kernel.org/netdev/20170322170035.923581-4-kafai@fb.com/

11.1.10 BPF_MAP_TYPE_QUEUE and BPF_MAP_TYPE_STACK

Note:

- BPF_MAP_TYPE_QUEUE and BPF_MAP_TYPE_STACK were introduced in kernel version 4.20

BPF_MAP_TYPE_QUEUE provides FIFO storage and BPF_MAP_TYPE_STACK provides LIFO storage for BPF programs. These maps support peek, pop and push operations that are exposed to BPF programs through the respective helpers. These operations are exposed to userspace applications using the existing bpf syscall in the following way:

- BPF_MAP_LOOKUP_ELEM -> peek
- BPF_MAP_LOOKUP_AND_DELETE_ELEM -> pop
- BPF_MAP_UPDATE_ELEM -> push

BPF_MAP_TYPE_QUEUE and BPF_MAP_TYPE_STACK do not support BPF_F_NO_PREALLOC.

Usage

Kernel BPF

bpf_map_push_elem()

```
long bpf_map_push_elem(struct bpf_map *map, const void *value, u64 flags)
```

An element value can be added to a queue or stack using the bpf_map_push_elem helper. The flags parameter must be set to BPF_ANY or BPF_EXIST. If flags is set to BPF_EXIST then, when the queue or stack is full, the oldest element will be removed to make room for value to be added. Returns 0 on success, or negative error in case of failure.

bpf_map_peek_elem()

```
long bpf_map_peek_elem(struct bpf_map *map, void *value)
```

This helper fetches an element `value` from a queue or stack without removing it. Returns `0` on success, or negative error in case of failure.

bpf_map_pop_elem()

```
long bpf_map_pop_elem(struct bpf_map *map, void *value)
```

This helper removes an element into `value` from a queue or stack. Returns `0` on success, or negative error in case of failure.

Userspace

bpf_map_update_elem()

```
int bpf_map_update_elem (int fd, const void *key, const void *value, __u64␣
↪flags)
```

A userspace program can push `value` onto a queue or stack using libbpf's `bpf_map_update_elem` function. The `key` parameter must be set to `NULL` and `flags` must be set to `BPF_ANY` or `BPF_EXIST`, with the same semantics as the `bpf_map_push_elem` kernel helper. Returns `0` on success, or negative error in case of failure.

bpf_map_lookup_elem()

```
int bpf_map_lookup_elem (int fd, const void *key, void *value)
```

A userspace program can peek at the `value` at the head of a queue or stack using the libbpf `bpf_map_lookup_elem` function. The `key` parameter must be set to `NULL`. Returns `0` on success, or negative error in case of failure.

bpf_map_lookup_and_delete_elem()

```
int bpf_map_lookup_and_delete_elem (int fd, const void *key, void *value)
```

A userspace program can pop a `value` from the head of a queue or stack using the libbpf `bpf_map_lookup_and_delete_elem` function. The `key` parameter must be set to `NULL`. Returns `0` on success, or negative error in case of failure.

Examples

Kernel BPF

This snippet shows how to declare a queue in a BPF program:

```
struct {
        __uint(type, BPF_MAP_TYPE_QUEUE);
        __type(value, __u32);
        __uint(max_entries, 10);
} queue SEC(".maps");
```

Userspace

This snippet shows how to use libbpf's low-level API to create a queue from userspace:

```
int create_queue()
{
        return bpf_map_create(BPF_MAP_TYPE_QUEUE,
                              "sample_queue", /* name */
                              0,              /* key size, must be zero */
                              sizeof(__u32),  /* value size */
                              10,             /* max entries */
                              NULL);          /* create options */
}
```

References

https://lwn.net/ml/netdev/153986858555.9127.14517764371945179514.stgit@kernel/

11.1.11 BPF_MAP_TYPE_SK_STORAGE

Note:

- BPF_MAP_TYPE_SK_STORAGE was introduced in kernel version 5.2

BPF_MAP_TYPE_SK_STORAGE is used to provide socket-local storage for BPF programs. A map of type BPF_MAP_TYPE_SK_STORAGE declares the type of storage to be provided and acts as the handle for accessing the socket-local storage. The values for maps of type BPF_MAP_TYPE_SK_STORAGE are stored locally with each socket instead of with the map. The kernel is responsible for allocating storage for a socket when requested and for freeing the storage when either the map or the socket is deleted.

Note:

- The key type must be `int` and `max_entries` must be set to `0`.
- The BPF_F_NO_PREALLOC flag must be used when creating a map for socket-local storage.

Usage

Kernel BPF

bpf_sk_storage_get()

```
void *bpf_sk_storage_get(struct bpf_map *map, void *sk, void *value, u64 flags)
```

Socket-local storage for `map` can be retrieved from socket `sk` using the `bpf_sk_storage_get()` helper. If the `BPF_LOCAL_STORAGE_GET_F_CREATE` flag is used then `bpf_sk_storage_get()` will create the storage for `sk` if it does not already exist. `value` can be used together with `BPF_LOCAL_STORAGE_GET_F_CREATE` to initialize the storage value, otherwise it will be zero initialized. Returns a pointer to the storage on success, or `NULL` in case of failure.

Note:

- `sk` is a kernel `struct sock` pointer for LSM or tracing programs.
- `sk` is a `struct bpf_sock` pointer for other program types.

---

bpf_sk_storage_delete()

```
long bpf_sk_storage_delete(struct bpf_map *map, void *sk)
```

Socket-local storage for `map` can be deleted from socket `sk` using the `bpf_sk_storage_delete()` helper. Returns `0` on success, or negative error in case of failure.

User space

bpf_map_update_elem()

```
int bpf_map_update_elem(int map_fd, const void *key, const void *value, __u64␣
↪flags)
```

Socket-local storage for map `map_fd` can be added or updated locally to a socket using the `bpf_map_update_elem()` libbpf function. The socket is identified by a *socket* `fd` stored in the pointer `key`. The pointer `value` has the data to be added or updated to the socket `fd`. The type and size of `value` should be the same as the value type of the map definition.

The `flags` parameter can be used to control the update behaviour:

- `BPF_ANY` will create storage for *socket* `fd` or update existing storage.
- `BPF_NOEXIST` will create storage for *socket* `fd` only if it did not already exist, otherwise the call will fail with `-EEXIST`.

- `BPF_EXIST` will update existing storage for *socket* `fd` if it already exists, otherwise the call will fail with `-ENOENT`.

Returns `0` on success, or negative error in case of failure.

bpf_map_lookup_elem()

```
int bpf_map_lookup_elem(int map_fd, const void *key, void *value)
```

Socket-local storage for map `map_fd` can be retrieved from a socket using the `bpf_map_lookup_elem()` libbpf function. The storage is retrieved from the socket identified by a *socket* `fd` stored in the pointer `key`. Returns `0` on success, or negative error in case of failure.

bpf_map_delete_elem()

```
int bpf_map_delete_elem(int map_fd, const void *key)
```

Socket-local storage for map `map_fd` can be deleted from a socket using the `bpf_map_delete_elem()` libbpf function. The storage is deleted from the socket identified by a *socket* `fd` stored in the pointer `key`. Returns `0` on success, or negative error in case of failure.

Examples

Kernel BPF

This snippet shows how to declare socket-local storage in a BPF program:

```
struct {
        __uint(type, BPF_MAP_TYPE_SK_STORAGE);
        __uint(map_flags, BPF_F_NO_PREALLOC);
        __type(key, int);
        __type(value, struct my_storage);
} socket_storage SEC(".maps");
```

This snippet shows how to retrieve socket-local storage in a BPF program:

```
SEC("sockops")
int _sockops(struct bpf_sock_ops *ctx)
{
        struct my_storage *storage;
        struct bpf_sock *sk;

        sk = ctx->sk;
        if (!sk)
                return 1;

        storage = bpf_sk_storage_get(&socket_storage, sk, 0,
```

```
                                        BPF_LOCAL_STORAGE_GET_F_CREATE);
        if (!storage)
                return 1;

        /* Use 'storage' here */

        return 1;
}
```

Please see the `tools/testing/selftests/bpf` directory for functional examples.

References

https://lwn.net/ml/netdev/20190426171103.61892-1-kafai@fb.com/

11.1.12 BPF_MAP_TYPE_SOCKMAP and BPF_MAP_TYPE_SOCKHASH

Note:

- `BPF_MAP_TYPE_SOCKMAP` was introduced in kernel version 4.14
- `BPF_MAP_TYPE_SOCKHASH` was introduced in kernel version 4.18

`BPF_MAP_TYPE_SOCKMAP` and `BPF_MAP_TYPE_SOCKHASH` maps can be used to redirect skbs between sockets or to apply policy at the socket level based on the result of a BPF (verdict) program with the help of the BPF helpers `bpf_sk_redirect_map()`, `bpf_sk_redirect_hash()`, `bpf_msg_redirect_map()` and `bpf_msg_redirect_hash()`.

`BPF_MAP_TYPE_SOCKMAP` is backed by an array that uses an integer key as the index to look up a reference to a `struct sock`. The map values are socket descriptors. Similarly, `BPF_MAP_TYPE_SOCKHASH` is a hash backed BPF map that holds references to sockets via their socket descriptors.

Note: The value type is either __u32 or __u64; the latter (__u64) is to support returning socket cookies to userspace. Returning the `struct sock *` that the map holds to user-space is neither safe nor useful.

These maps may have BPF programs attached to them, specifically a parser program and a verdict program. The parser program determines how much data has been parsed and therefore how much data needs to be queued to come to a verdict. The verdict program is essentially the redirect program and can return a verdict of `__SK_DROP`, `__SK_PASS`, or `__SK_REDIRECT`.

When a socket is inserted into one of these maps, its socket callbacks are replaced and a `struct sk_psock` is attached to it. Additionally, this `sk_psock` inherits the programs that are attached to the map.

A sock object may be in multiple maps, but can only inherit a single parse or verdict program. If adding a sock object to a map would result in having multiple parser programs the update will return an EBUSY error.

The supported programs to attach to these maps are:

```
struct sk_psock_progs {
        struct bpf_prog *msg_parser;
        struct bpf_prog *stream_parser;
        struct bpf_prog *stream_verdict;
        struct bpf_prog *skb_verdict;
};
```

Note: Users are not allowed to attach `stream_verdict` and `skb_verdict` programs to the same map.

The attach types for the map programs are:

- `msg_parser` program - `BPF_SK_MSG_VERDICT`.
- `stream_parser` program - `BPF_SK_SKB_STREAM_PARSER`.
- `stream_verdict` program - `BPF_SK_SKB_STREAM_VERDICT`.
- `skb_verdict` program - `BPF_SK_SKB_VERDICT`.

There are additional helpers available to use with the parser and verdict programs: `bpf_msg_apply_bytes()` and `bpf_msg_cork_bytes()`. With `bpf_msg_apply_bytes()` BPF programs can tell the infrastructure how many bytes the given verdict should apply to. The helper `bpf_msg_cork_bytes()` handles a different case where a BPF program cannot reach a verdict on a msg until it receives more bytes AND the program doesn't want to forward the packet until it is known to be good.

Finally, the helpers `bpf_msg_pull_data()` and `bpf_msg_push_data()` are available to `BPF_PROG_TYPE_SK_MSG` BPF programs to pull in data and set the start and end pointers to given values or to add metadata to the `struct sk_msg_buff *msg`.

All these helpers will be described in more detail below.

Usage

Kernel BPF

bpf_msg_redirect_map()

```
long bpf_msg_redirect_map(struct sk_msg_buff *msg, struct bpf_map *map, u32␣
↪key, u64 flags)
```

This helper is used in programs implementing policies at the socket level. If the message `msg` is allowed to pass (i.e., if the verdict BPF program returns `SK_PASS`), redirect it to the socket referenced by `map` (of type `BPF_MAP_TYPE_SOCKMAP`) at index `key`. Both ingress and egress interfaces can be used for redirection. The `BPF_F_INGRESS` value in `flags` is used to select the ingress path otherwise the egress path is selected. This is the only flag supported for now.

Returns `SK_PASS` on success, or `SK_DROP` on error.

bpf_sk_redirect_map()

```
long bpf_sk_redirect_map(struct sk_buff *skb, struct bpf_map *map, u32 key u64␣
↪flags)
```

Redirect the packet to the socket referenced by `map` (of type `BPF_MAP_TYPE_SOCKMAP`) at index `key`. Both ingress and egress interfaces can be used for redirection. The `BPF_F_INGRESS` value in `flags` is used to select the ingress path otherwise the egress path is selected. This is the only flag supported for now.

Returns `SK_PASS` on success, or `SK_DROP` on error.

bpf_map_lookup_elem()

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

socket entries of type `struct sock *` can be retrieved using the `bpf_map_lookup_elem()` helper.

bpf_sock_map_update()

```
long bpf_sock_map_update(struct bpf_sock_ops *skops, struct bpf_map *map, void␣
↪*key, u64 flags)
```

Add an entry to, or update a `map` referencing sockets. The `skops` is used as a new value for the entry associated to `key`. The `flags` argument can be one of the following:

- `BPF_ANY`: Create a new element or update an existing element.
- `BPF_NOEXIST`: Create a new element only if it did not exist.
- `BPF_EXIST`: Update an existing element.

If the `map` has BPF programs (parser and verdict), those will be inherited by the socket being added. If the socket is already attached to BPF programs, this results in an error.

Returns 0 on success, or a negative error in case of failure.

bpf_sock_hash_update()

```
long bpf_sock_hash_update(struct bpf_sock_ops *skops, struct bpf_map *map,␣
↪void *key, u64 flags)
```

Add an entry to, or update a sockhash `map` referencing sockets. The `skops` is used as a new value for the entry associated to `key`.

The `flags` argument can be one of the following:

- `BPF_ANY`: Create a new element or update an existing element.
- `BPF_NOEXIST`: Create a new element only if it did not exist.

- BPF_EXIST: Update an existing element.

If the map has BPF programs (parser and verdict), those will be inherited by the socket being added. If the socket is already attached to BPF programs, this results in an error.

Returns 0 on success, or a negative error in case of failure.

bpf_msg_redirect_hash()

```
long bpf_msg_redirect_hash(struct sk_msg_buff *msg, struct bpf_map *map, void␣
↪*key, u64 flags)
```

This helper is used in programs implementing policies at the socket level. If the message msg is allowed to pass (i.e., if the verdict BPF program returns SK_PASS), redirect it to the socket referenced by map (of type BPF_MAP_TYPE_SOCKHASH) using hash key. Both ingress and egress interfaces can be used for redirection. The BPF_F_INGRESS value in flags is used to select the ingress path otherwise the egress path is selected. This is the only flag supported for now.

Returns SK_PASS on success, or SK_DROP on error.

bpf_sk_redirect_hash()

```
long bpf_sk_redirect_hash(struct sk_buff *skb, struct bpf_map *map, void *key,␣
↪u64 flags)
```

This helper is used in programs implementing policies at the skb socket level. If the sk_buff skb is allowed to pass (i.e., if the verdict BPF program returns SK_PASS), redirect it to the socket referenced by map (of type BPF_MAP_TYPE_SOCKHASH) using hash key. Both ingress and egress interfaces can be used for redirection. The BPF_F_INGRESS value in flags is used to select the ingress path otherwise the egress path is selected. This is the only flag supported for now.

Returns SK_PASS on success, or SK_DROP on error.

bpf_msg_apply_bytes()

```
long bpf_msg_apply_bytes(struct sk_msg_buff *msg, u32 bytes)
```

For socket policies, apply the verdict of the BPF program to the next (number of bytes) of message msg. For example, this helper can be used in the following cases:

- A single sendmsg() or sendfile() system call contains multiple logical messages that the BPF program is supposed to read and for which it should apply a verdict.
- A BPF program only cares to read the first bytes of a msg. If the message has a large payload, then setting up and calling the BPF program repeatedly for all bytes, even though the verdict is already known, would create unnecessary overhead.

Returns 0

bpf_msg_cork_bytes()

```
long bpf_msg_cork_bytes(struct sk_msg_buff *msg, u32 bytes)
```

For socket policies, prevent the execution of the verdict BPF program for message `msg` until the number of `bytes` have been accumulated.

This can be used when one needs a specific number of bytes before a verdict can be assigned, even if the data spans multiple `sendmsg()` or `sendfile()` calls.

Returns 0

bpf_msg_pull_data()

```
long bpf_msg_pull_data(struct sk_msg_buff *msg, u32 start, u32 end, u64 flags)
```

For socket policies, pull in non-linear data from user space for `msg` and set pointers `msg->data` and `msg->data_end` to `start` and `end` bytes offsets into `msg`, respectively.

If a program of type `BPF_PROG_TYPE_SK_MSG` is run on a `msg` it can only parse data that the (`data`, `data_end`) pointers have already consumed. For `sendmsg()` hooks this is likely the first scatterlist element. But for calls relying on MSG_SPLICE_PAGES (e.g., `sendfile()`) this will be the range (**0**, **0**) because the data is shared with user space and by default the objective is to avoid allowing user space to modify data while (or after) BPF verdict is being decided. This helper can be used to pull in data and to set the start and end pointers to given values. Data will be copied if necessary (i.e., if data was not linear and if start and end pointers do not point to the same chunk).

A call to this helper is susceptible to change the underlying packet buffer. Therefore, at load time, all checks on pointers previously done by the verifier are invalidated and must be performed again, if the helper is used in combination with direct packet access.

All values for `flags` are reserved for future usage, and must be left at zero.

Returns 0 on success, or a negative error in case of failure.

bpf_map_lookup_elem()

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

Look up a socket entry in the sockmap or sockhash map.

Returns the socket entry associated to `key`, or NULL if no entry was found.

bpf_map_update_elem()

```
long bpf_map_update_elem(struct bpf_map *map, const void *key, const void␣
↪*value, u64 flags)
```

Add or update a socket entry in a sockmap or sockhash.

The flags argument can be one of the following:

- BPF_ANY: Create a new element or update an existing element.
- BPF_NOEXIST: Create a new element only if it did not exist.
- BPF_EXIST: Update an existing element.

Returns 0 on success, or a negative error in case of failure.

bpf_map_delete_elem()

```
long bpf_map_delete_elem(struct bpf_map *map, const void *key)
```

Delete a socket entry from a sockmap or a sockhash.

Returns 0 on success, or a negative error in case of failure.

User space

bpf_map_update_elem()

```
int bpf_map_update_elem(int fd, const void *key, const void *value, __u64␣
↪flags)
```

Sockmap entries can be added or updated using the `bpf_map_update_elem()` function. The `key` parameter is the index value of the sockmap array. And the `value` parameter is the FD value of that socket.

Under the hood, the sockmap update function uses the socket FD value to retrieve the associated socket and its attached psock.

The flags argument can be one of the following:

- BPF_ANY: Create a new element or update an existing element.
- BPF_NOEXIST: Create a new element only if it did not exist.
- BPF_EXIST: Update an existing element.

bpf_map_lookup_elem()

```
int bpf_map_lookup_elem(int fd, const void *key, void *value)
```

Sockmap entries can be retrieved using the `bpf_map_lookup_elem()` function.

Note: The entry returned is a socket cookie rather than a socket itself.

bpf_map_delete_elem()

```
int bpf_map_delete_elem(int fd, const void *key)
```

Sockmap entries can be deleted using the `bpf_map_delete_elem()` function.

Returns 0 on success, or negative error in case of failure.

Examples

Kernel BPF

Several examples of the use of sockmap APIs can be found in:

- tools/testing/selftests/bpf/progs/test_sockmap_kern.h
- tools/testing/selftests/bpf/progs/sockmap_parse_prog.c
- tools/testing/selftests/bpf/progs/sockmap_verdict_prog.c
- tools/testing/selftests/bpf/progs/test_sockmap_listen.c
- tools/testing/selftests/bpf/progs/test_sockmap_update.c

The following code snippet shows how to declare a sockmap.

```
struct {
        __uint(type, BPF_MAP_TYPE_SOCKMAP);
        __uint(max_entries, 1);
        __type(key, __u32);
        __type(value, __u64);
} sock_map_rx SEC(".maps");
```

The following code snippet shows a sample parser program.

```
SEC("sk_skb/stream_parser")
int bpf_prog_parser(struct __sk_buff *skb)
{
        return skb->len;
}
```

The following code snippet shows a simple verdict program that interacts with a sockmap to redirect traffic to another socket based on the local port.

```
SEC("sk_skb/stream_verdict")
int bpf_prog_verdict(struct __sk_buff *skb)
{
        __u32 lport = skb->local_port;
        __u32 idx = 0;

        if (lport == 10000)
                return bpf_sk_redirect_map(skb, &sock_map_rx, idx, 0);

        return SK_PASS;
}
```

The following code snippet shows how to declare a sockhash map.

```
struct socket_key {
        __u32 src_ip;
        __u32 dst_ip;
        __u32 src_port;
        __u32 dst_port;
};

struct {
        __uint(type, BPF_MAP_TYPE_SOCKHASH);
        __uint(max_entries, 1);
        __type(key, struct socket_key);
        __type(value, __u64);
} sock_hash_rx SEC(".maps");
```

The following code snippet shows a simple verdict program that interacts with a sockhash to redirect traffic to another socket based on a hash of some of the skb parameters.

```
static inline
void extract_socket_key(struct __sk_buff *skb, struct socket_key *key)
{
        key->src_ip = skb->remote_ip4;
        key->dst_ip = skb->local_ip4;
        key->src_port = skb->remote_port >> 16;
        key->dst_port = (bpf_htonl(skb->local_port)) >> 16;
}

SEC("sk_skb/stream_verdict")
int bpf_prog_verdict(struct __sk_buff *skb)
{
        struct socket_key key;

        extract_socket_key(skb, &key);

        return bpf_sk_redirect_hash(skb, &sock_hash_rx, &key, 0);
}
```

User space

Several examples of the use of sockmap APIs can be found in:

- tools/testing/selftests/bpf/prog_tests/sockmap_basic.c
- tools/testing/selftests/bpf/test_sockmap.c
- tools/testing/selftests/bpf/test_maps.c

The following code sample shows how to create a sockmap, attach a parser and verdict program, as well as add a socket entry.

```
int create_sample_sockmap(int sock, int parse_prog_fd, int verdict_prog_fd)
{
        int index = 0;
        int map, err;

        map = bpf_map_create(BPF_MAP_TYPE_SOCKMAP, NULL, sizeof(int),␣
↪sizeof(int), 1, NULL);
        if (map < 0) {
                fprintf(stderr, "Failed to create sockmap: %s\n",␣
↪strerror(errno));
                return -1;
        }

        err = bpf_prog_attach(parse_prog_fd, map, BPF_SK_SKB_STREAM_PARSER, 0);
        if (err){
                fprintf(stderr, "Failed to attach_parser_prog_to_map: %s\n",␣
↪strerror(errno));
                goto out;
        }

        err = bpf_prog_attach(verdict_prog_fd, map, BPF_SK_SKB_STREAM_VERDICT,␣
↪0);
        if (err){
                fprintf(stderr, "Failed to attach_verdict_prog_to_map: %s\n",␣
↪strerror(errno));
                goto out;
        }

        err = bpf_map_update_elem(map, &index, &sock, BPF_NOEXIST);
        if (err) {
                fprintf(stderr, "Failed to update sockmap: %s\n",␣
↪strerror(errno));
                goto out;
        }

out:
        close(map);
        return err;
}
```

References

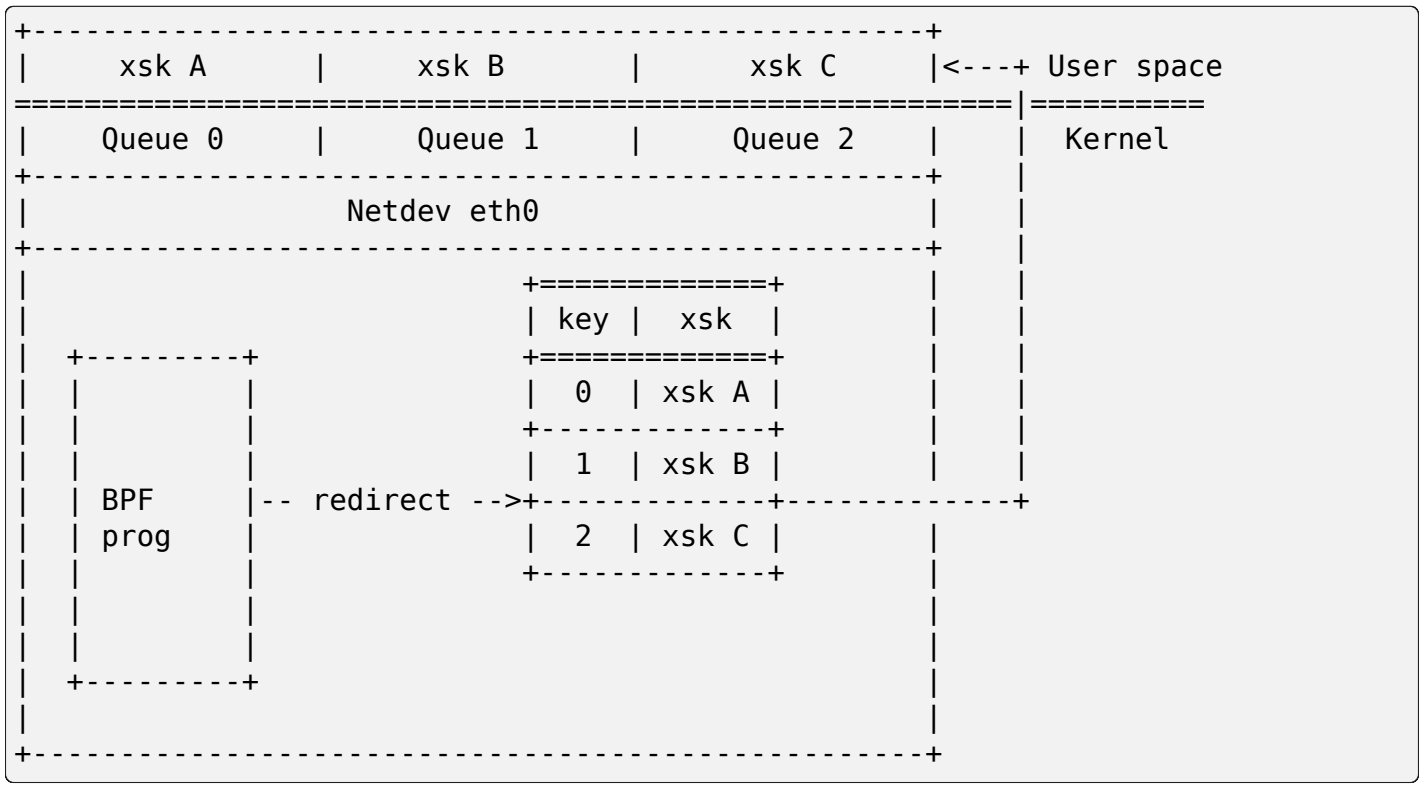
- https://github.com/jrfastab/linux-kernel-xdp/commit/c89fd73cb9d2d7f3c716c3e00836f07b1aeb2
- https://lwn.net/Articles/731133/
- http://vger.kernel.org/lpc_net2018_talks/ktls_bpf_paper.pdf
- https://lwn.net/Articles/748628/
- https://lore.kernel.org/bpf/20200218171023.844439-7-jakub@cloudflare.com/

11.1.13 BPF_MAP_TYPE_XSKMAP

Note:

- `BPF_MAP_TYPE_XSKMAP` was introduced in kernel version 4.18

The `BPF_MAP_TYPE_XSKMAP` is used as a backend map for XDP BPF helper call `bpf_redirect_map()` and `XDP_REDIRECT` action, like 'devmap' and 'cpumap'. This map type redirects raw XDP frames to AF_XDP sockets (XSKs), a new type of address family in the kernel that allows redirection of frames from a driver to user space without having to traverse the full network stack. An AF_XDP socket binds to a single netdev queue. A mapping of XSKs to queues is shown below:



Note: An AF_XDP socket that is bound to a certain <netdev/queue_id> will *only* accept XDP frames from that <netdev/queue_id>. If an XDP program tries to redirect from a <netdev/queue_id> other than what the socket is bound to, the frame will not be received on the

socket.

Typically an XSKMAP is created per netdev. This map contains an array of XSK File Descriptors (FDs). The number of array elements is typically set or adjusted using the `max_entries` map parameter. For AF_XDP `max_entries` is equal to the number of queues supported by the netdev.

Note: Both the map key and map value size must be 4 bytes.

Usage

Kernel BPF

bpf_redirect_map()

```
long bpf_redirect_map(struct bpf_map *map, u32 key, u64 flags)
```

Redirect the packet to the endpoint referenced by `map` at index `key`. For `BPF_MAP_TYPE_XSKMAP` this map contains references to XSK FDs for sockets attached to a netdev's queues.

Note: If the map is empty at an index, the packet is dropped. This means that it is necessary to have an XDP program loaded with at least one XSK in the XSKMAP to be able to get any traffic to user space through the socket.

bpf_map_lookup_elem()

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

XSK entry references of type `struct xdp_sock *` can be retrieved using the `bpf_map_lookup_elem()` helper.

User space

Note: XSK entries can only be updated/deleted from user space and not from a BPF program. Trying to call these functions from a kernel BPF program will result in the program failing to load and a verifier warning.

bpf_map_update_elem()

```
int bpf_map_update_elem(int fd, const void *key, const void *value, __u64␣
↪flags)
```

XSK entries can be added or updated using the `bpf_map_update_elem()` helper. The `key` parameter is equal to the queue_id of the queue the XSK is attaching to. And the `value` parameter is the FD value of that socket.

Under the hood, the XSKMAP update function uses the XSK FD value to retrieve the associated `struct xdp_sock` instance.

The flags argument can be one of the following:

- BPF_ANY: Create a new element or update an existing element.
- BPF_NOEXIST: Create a new element only if it did not exist.
- BPF_EXIST: Update an existing element.

bpf_map_lookup_elem()

```
int bpf_map_lookup_elem(int fd, const void *key, void *value)
```

Returns `struct xdp_sock *` or negative error in case of failure.

bpf_map_delete_elem()

```
int bpf_map_delete_elem(int fd, const void *key)
```

XSK entries can be deleted using the `bpf_map_delete_elem()` helper. This helper will return 0 on success, or negative error in case of failure.

Note: When libxdp deletes an XSK it also removes the associated socket entry from the XSKMAP.

Examples

Kernel

The following code snippet shows how to declare a `BPF_MAP_TYPE_XSKMAP` called `xsks_map` and how to redirect packets to an XSK.

```
struct {
        __uint(type, BPF_MAP_TYPE_XSKMAP);
        __type(key, __u32);
        __type(value, __u32);
        __uint(max_entries, 64);
```

```
} xsks_map SEC(".maps");


SEC("xdp")
int xsk_redir_prog(struct xdp_md *ctx)
{
        __u32 index = ctx->rx_queue_index;

        if (bpf_map_lookup_elem(&xsks_map, &index))
                return bpf_redirect_map(&xsks_map, index, 0);
        return XDP_PASS;
}
```

User space

The following code snippet shows how to update an XSKMAP with an XSK entry.

```
int update_xsks_map(struct bpf_map *xsks_map, int queue_id, int xsk_fd)
{
        int ret;

        ret = bpf_map_update_elem(bpf_map__fd(xsks_map), &queue_id, &xsk_fd,␣
↪0);
        if (ret < 0)
                fprintf(stderr, "Failed to update xsks_map: %s\n",␣
↪strerror(errno));

        return ret;
}
```

For an example on how create AF_XDP sockets, please see the AF_XDP-example and AF_XDP-forwarding programs in the bpf-examples directory in the libxdp repository. For a detailed explanation of the AF_XDP interface please see:

- libxdp-readme.
- AF_XDP kernel documentation.

Note: The most comprehensive resource for using XSKMAPs and AF_XDP is libxdp.

11.2 Usage Notes

int **bpf**(int command, union bpf_attr *attr, u32 size)

Use the `bpf()` system call to perform the operation specified by `command`. The operation takes parameters provided in `attr`. The `size` argument is the size of the `union bpf_attr` in `attr`.

BPF_MAP_CREATE

Create a map with the desired type and attributes in `attr`:

```
int fd;
union bpf_attr attr = {
        .map_type = BPF_MAP_TYPE_ARRAY;  /* mandatory */
        .key_size = sizeof(__u32);       /* mandatory */
        .value_size = sizeof(__u32);     /* mandatory */
        .max_entries = 256;              /* mandatory */
        .map_flags = BPF_F_MMAPABLE;
        .map_name = "example_array";
};

fd = bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
```

Returns a process-local file descriptor on success, or negative error in case of failure. The map can be deleted by calling `close(fd)`. Maps held by open file descriptors will be deleted automatically when a process exits.

Note: Valid characters for `map_name` are `A-Z`, `a-z`, `0-9`, `'_'` and `'.'`.

BPF_MAP_LOOKUP_ELEM

Lookup key in a given map using `attr->map_fd`, `attr->key`, `attr->value`. Returns zero and stores found elem into `attr->value` on success, or negative error on failure.

BPF_MAP_UPDATE_ELEM

Create or update key/value pair in a given map using `attr->map_fd`, `attr->key`, `attr->value`. Returns zero on success or negative error on failure.

BPF_MAP_DELETE_ELEM

Find and delete element by key in a given map using `attr->map_fd`, `attr->key`. Returns zero on success or negative error on failure.

CHAPTER

TWELVE

RUNNING BPF PROGRAMS FROM USERSPACE

This document describes the BPF_PROG_RUN facility for running BPF programs from userspace.

12.1 Overview

The BPF_PROG_RUN command can be used through the bpf() syscall to execute a BPF program in the kernel and return the results to userspace. This can be used to unit test BPF programs against user-supplied context objects, and as way to explicitly execute programs in the kernel for their side effects. The command was previously named BPF_PROG_TEST_RUN, and both constants continue to be defined in the UAPI header, aliased to the same value.

The BPF_PROG_RUN command can be used to execute BPF programs of the following types:

- BPF_PROG_TYPE_SOCKET_FILTER
- BPF_PROG_TYPE_SCHED_CLS
- BPF_PROG_TYPE_SCHED_ACT
- BPF_PROG_TYPE_XDP
- BPF_PROG_TYPE_SK_LOOKUP
- BPF_PROG_TYPE_CGROUP_SKB
- BPF_PROG_TYPE_LWT_IN
- BPF_PROG_TYPE_LWT_OUT
- BPF_PROG_TYPE_LWT_XMIT
- BPF_PROG_TYPE_LWT_SEG6LOCAL
- BPF_PROG_TYPE_FLOW_DISSECTOR
- BPF_PROG_TYPE_STRUCT_OPS
- BPF_PROG_TYPE_RAW_TRACEPOINT
- BPF_PROG_TYPE_SYSCALL

When using the BPF_PROG_RUN command, userspace supplies an input context object and (for program types operating on network packets) a buffer containing the packet data that the BPF program will operate on. The kernel will then execute the program and return the results to userspace. Note that programs will not have any side effects while being run in this mode; in particular, packets will not actually be redirected or dropped, the program return code will just be returned to userspace. A separate mode for live execution of XDP programs is provided, documented separately below.

12.2 Running XDP programs in "live frame mode"

The BPF_PROG_RUN command has a separate mode for running live XDP programs, which can be used to execute XDP programs in a way where packets will actually be processed by the kernel after the execution of the XDP program as if they arrived on a physical interface. This mode is activated by setting the BPF_F_TEST_XDP_LIVE_FRAMES flag when supplying an XDP program to BPF_PROG_RUN.

The live packet mode is optimised for high performance execution of the supplied XDP program many times (suitable for, e.g., running as a traffic generator), which means the semantics are not quite as straight-forward as the regular test run mode. Specifically:

- When executing an XDP program in live frame mode, the result of the execution will not be returned to userspace; instead, the kernel will perform the operation indicated by the program's return code (drop the packet, redirect it, etc). For this reason, setting the data_out or ctx_out attributes in the syscall parameters when running in this mode will be rejected. In addition, not all failures will be reported back to userspace directly; specifically, only fatal errors in setup or during execution (like memory allocation errors) will halt execution and return an error. If an error occurs in packet processing, like a failure to redirect to a given interface, execution will continue with the next repetition; these errors can be detected via the same trace points as for regular XDP programs.
- Userspace can supply an ifindex as part of the context object, just like in the regular (non-live) mode. The XDP program will be executed as though the packet arrived on this interface; i.e., the ingress_ifindex of the context object will point to that interface. Furthermore, if the XDP program returns XDP_PASS, the packet will be injected into the kernel networking stack as though it arrived on that ifindex, and if it returns XDP_TX, the packet will be transmitted *out* of that same interface. Do note, though, that because the program execution is not happening in driver context, an XDP_TX is actually turned into the same action as an XDP_REDIRECT to that same interface (i.e., it will only work if the driver has support for the ndo_xdp_xmit driver op).
- When running the program with multiple repetitions, the execution will happen in batches. The batch size defaults to 64 packets (which is same as the maximum NAPI receive batch size), but can be specified by userspace through the batch_size parameter, up to a maximum of 256 packets. For each batch, the kernel executes the XDP program repeatedly, each invocation getting a separate copy of the packet data. For each repetition, if the program drops the packet, the data page is immediately recycled (see below). Otherwise, the packet is buffered until the end of the batch, at which point all packets buffered this way during the batch are transmitted at once.
- When setting up the test run, the kernel will initialise a pool of memory pages of the same size as the batch size. Each memory page will be initialised with the initial packet data supplied by userspace at BPF_PROG_RUN invocation. When possible, the pages will be

recycled on future program invocations, to improve performance. Pages will generally be recycled a full batch at a time, except when a packet is dropped (by return code or because of, say, a redirection error), in which case that page will be recycled immediately. If a packet ends up being passed to the regular networking stack (because the XDP program returns `XDP_PASS`, or because it ends up being redirected to an interface that injects it into the stack), the page will be released and a new one will be allocated when the pool is empty.

When recycling, the page content is not rewritten; only the packet boundary pointers (`data`, `data_end` and `data_meta`) in the context object will be reset to the original values. This means that if a program rewrites the packet contents, it has to be prepared to see either the original content or the modified version on subsequent invocations.

CHAPTER

THIRTEEN

CLASSIC BPF VS EBPF

eBPF is designed to be JITed with one to one mapping, which can also open up the possibility for GCC/LLVM compilers to generate optimized eBPF code through an eBPF backend that performs almost as fast as natively compiled code.

Some core changes of the eBPF format from classic BPF:

- Number of registers increase from 2 to 10:

  The old format had two registers A and X, and a hidden frame pointer. The new layout extends this to be 10 internal registers and a read-only frame pointer. Since 64-bit CPUs are passing arguments to functions via registers the number of args from eBPF program to in-kernel function is restricted to 5 and one register is used to accept return value from an in-kernel function. Natively, x86_64 passes first 6 arguments in registers, aarch64/sparcv9/mips64 have 7 - 8 registers for arguments; x86_64 has 6 callee saved registers, and aarch64/sparcv9/mips64 have 11 or more callee saved registers.

  Thus, all eBPF registers map one to one to HW registers on x86_64, aarch64, etc, and eBPF calling convention maps directly to ABIs used by the kernel on 64-bit architectures.

  On 32-bit architectures JIT may map programs that use only 32-bit arithmetic and may let more complex programs to be interpreted.

  R0 - R5 are scratch registers and eBPF program needs spill/fill them if necessary across calls. Note that there is only one eBPF program (== one eBPF main routine) and it cannot call other eBPF functions, it can only call predefined in-kernel functions, though.

- Register width increases from 32-bit to 64-bit:

  Still, the semantics of the original 32-bit ALU operations are preserved via 32-bit subregisters. All eBPF registers are 64-bit with 32-bit lower subregisters that zero-extend into 64-bit if they are being written to. That behavior maps directly to x86_64 and arm64 subregister definition, but makes other JITs more difficult.

  32-bit architectures run 64-bit eBPF programs via interpreter. Their JITs may convert BPF programs that only use 32-bit subregisters into native instruction set and let the rest being interpreted.

  Operation is 64-bit, because on 64-bit architectures, pointers are also 64-bit wide, and we want to pass 64-bit values in/out of kernel functions, so 32-bit eBPF registers would otherwise require to define register-pair ABI, thus, there won't be able to use a direct eBPF register to HW register mapping and JIT would need to do combine/split/move operations for every register in and out of the function, which is complex, bug prone and slow. Another reason is the use of atomic 64-bit counters.

- Conditional jt/jf targets replaced with jt/fall-through:

While the original design has constructs such as `if (cond) jump_true; else jump_false;`, they are being replaced into alternative constructs like `if (cond) jump_true; /* else fall-through */`.

- Introduces bpf_call insn and register passing convention for zero overhead calls from/to other kernel functions:

  Before an in-kernel function call, the eBPF program needs to place function arguments into R1 to R5 registers to satisfy calling convention, then the interpreter will take them from registers and pass to in-kernel function. If R1 - R5 registers are mapped to CPU registers that are used for argument passing on given architecture, the JIT compiler doesn't need to emit extra moves. Function arguments will be in the correct registers and BPF_CALL instruction will be JITed as single 'call' HW instruction. This calling convention was picked to cover common call situations without performance penalty.

  After an in-kernel function call, R1 - R5 are reset to unreadable and R0 has a return value of the function. Since R6 - R9 are callee saved, their state is preserved across the call.

  For example, consider three C functions:

```
u64 f1() { return (*_f2)(1); }
u64 f2(u64 a) { return f3(a + 1, a); }
u64 f3(u64 a, u64 b) { return a - b; }
```

  GCC can compile f1, f3 into x86_64:

```
f1:
    movl $1, %edi
    movq _f2(%rip), %rax
    jmp  *%rax
f3:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

  Function f2 in eBPF may look like:

```
f2:
    bpf_mov R2, R1
    bpf_add R1, 1
    bpf_call f3
    bpf_exit
```

  If f2 is JITed and the pointer stored to `_f2`. The calls f1 -> f2 -> f3 and returns will be seamless. Without JIT, __bpf_prog_run() interpreter needs to be used to call into f2.

  For practical reasons all eBPF programs have only one argument 'ctx' which is already placed into R1 (e.g. on __bpf_prog_run() startup) and the programs can call kernel functions with up to 5 arguments. Calls with 6 or more arguments are currently not supported, but these restrictions can be lifted if necessary in the future.

  On 64-bit architectures all register map to HW registers one to one. For example, x86_64 JIT compiler can map them as ...

```
R0 - rax
R1 - rdi
R2 - rsi
R3 - rdx
R4 - rcx
R5 - r8
R6 - rbx
R7 - r13
R8 - r14
R9 - r15
R10 - rbp
```

... since x86_64 ABI mandates rdi, rsi, rdx, rcx, r8, r9 for argument passing and rbx, r12 - r15 are callee saved.

Then the following eBPF pseudo-program:

```
bpf_mov R6, R1 /* save ctx */
bpf_mov R2, 2
bpf_mov R3, 3
bpf_mov R4, 4
bpf_mov R5, 5
bpf_call foo
bpf_mov R7, R0 /* save foo() return value */
bpf_mov R1, R6 /* restore ctx for next call */
bpf_mov R2, 6
bpf_mov R3, 7
bpf_mov R4, 8
bpf_mov R5, 9
bpf_call bar
bpf_add R0, R7
bpf_exit
```

After JIT to x86_64 may look like:

```
push %rbp
mov %rsp,%rbp
sub $0x228,%rsp
mov %rbx,-0x228(%rbp)
mov %r13,-0x220(%rbp)
mov %rdi,%rbx
mov $0x2,%esi
mov $0x3,%edx
mov $0x4,%ecx
mov $0x5,%r8d
callq foo
mov %rax,%r13
mov %rbx,%rdi
mov $0x6,%esi
mov $0x7,%edx
mov $0x8,%ecx
mov $0x9,%r8d
```

```
callq bar
add %r13,%rax
mov -0x228(%rbp),%rbx
mov -0x220(%rbp),%r13
leaveq
retq
```

Which is in this example equivalent in C to:

```
u64 bpf_filter(u64 ctx)
{
    return foo(ctx, 2, 3, 4, 5) + bar(ctx, 6, 7, 8, 9);
}
```

In-kernel functions foo() and bar() with prototype: u64 (*)(u64 arg1, u64 arg2, u64 arg3, u64 arg4, u64 arg5); will receive arguments in proper registers and place their return value into `%rax` which is R0 in eBPF. Prologue and epilogue are emitted by JIT and are implicit in the interpreter. R0-R5 are scratch registers, so eBPF program needs to preserve them across the calls as defined by calling convention.

For example the following program is invalid:

```
bpf_mov R1, 1
bpf_call foo
bpf_mov R0, R1
bpf_exit
```

After the call the registers R1-R5 contain junk values and cannot be read. An in-kernel *eBPF verifier* is used to validate eBPF programs.

Also in the new design, eBPF is limited to 4096 insns, which means that any program will terminate quickly and will only call a fixed number of kernel functions. Original BPF and eBPF are two operand instructions, which helps to do one-to-one mapping between eBPF insn and x86 insn during JIT.

The input context pointer for invoking the interpreter function is generic, its content is defined by a specific use case. For seccomp register R1 points to seccomp_data, for converted BPF filters R1 points to a skb.

A program, that is translated internally consists of the following elements:

```
op:16, jt:8, jf:8, k:32    ==>    op:8, dst_reg:4, src_reg:4, off:16, imm:32
```

So far 87 eBPF instructions were implemented. 8-bit 'op' opcode field has room for new instructions. Some of them may use 16/24/32 byte encoding. New instructions must be multiple of 8 bytes to preserve backward compatibility.

eBPF is a general purpose RISC instruction set. Not every register and every instruction are used during translation from original BPF to eBPF. For example, socket filters are not using `exclusive add` instruction, but tracing filters may do to maintain counters of events, for example. Register R9 is not used by socket filters either, but more complex filters may be running out of registers and would have to resort to spill/fill to stack.

eBPF can be used as a generic assembler for last step performance optimizations, socket filters and seccomp are using it as assembler. Tracing filters may use it as assembler to generate code

from kernel. In kernel usage may not be bounded by security considerations, since generated eBPF code may be optimizing internal code path and not being exposed to the user space. Safety of eBPF can come from the *eBPF verifier*. In such use cases as described, it may be used as safe instruction set.

Just like the original BPF, eBPF runs within a controlled environment, is deterministic and the kernel can easily prove that. The safety of the program can be determined in two steps: first step does depth-first-search to disallow loops and other CFG validation; second step starts from the first insn and descends all possible paths. It simulates execution of every insn and observes the state change of registers and stack.

13.1 opcode encoding

eBPF is reusing most of the opcode encoding from classic to simplify conversion of classic BPF to eBPF.

For arithmetic and jump instructions the 8-bit 'code' field is divided into three parts:

```
+----------------+--------+--------------------+
|   4 bits       |  1 bit |   3 bits           |
| operation code | source | instruction class  |
+----------------+--------+--------------------+
(MSB)                                  (LSB)
```

Three LSB bits store instruction class which is one of:

Classic BPF classes	eBPF classes
BPF_LD 0x00	BPF_LD 0x00
BPF_LDX 0x01	BPF_LDX 0x01
BPF_ST 0x02	BPF_ST 0x02
BPF_STX 0x03	BPF_STX 0x03
BPF_ALU 0x04	BPF_ALU 0x04
BPF_JMP 0x05	BPF_JMP 0x05
BPF_RET 0x06	BPF_JMP32 0x06
BPF_MISC 0x07	BPF_ALU64 0x07

The 4th bit encodes the source operand ...

```
BPF_K     0x00
BPF_X     0x08
```

- in classic BPF, this means:

```
BPF_SRC(code) == BPF_X - use register X as source operand
BPF_SRC(code) == BPF_K - use 32-bit immediate as source operand
```

- in eBPF, this means:

```
BPF_SRC(code) == BPF_X - use 'src_reg' register as source operand
BPF_SRC(code) == BPF_K - use 32-bit immediate as source operand
```

... and four MSB bits store operation code.

If BPF_CLASS(code) == BPF_ALU or BPF_ALU64 [in eBPF], BPF_OP(code) is one of:

```
BPF_ADD   0x00
BPF_SUB   0x10
BPF_MUL   0x20
BPF_DIV   0x30
BPF_OR    0x40
BPF_AND   0x50
BPF_LSH   0x60
BPF_RSH   0x70
BPF_NEG   0x80
BPF_MOD   0x90
BPF_XOR   0xa0
BPF_MOV   0xb0  /* eBPF only: mov reg to reg */
BPF_ARSH  0xc0  /* eBPF only: sign extending shift right */
BPF_END   0xd0  /* eBPF only: endianness conversion */
```

If BPF_CLASS(code) == BPF_JMP or BPF_JMP32 [in eBPF], BPF_OP(code) is one of:

```
BPF_JA    0x00  /* BPF_JMP only */
BPF_JEQ   0x10
BPF_JGT   0x20
BPF_JGE   0x30
BPF_JSET  0x40
BPF_JNE   0x50  /* eBPF only: jump != */
BPF_JSGT  0x60  /* eBPF only: signed '>' */
BPF_JSGE  0x70  /* eBPF only: signed '>=' */
BPF_CALL  0x80  /* eBPF BPF_JMP only: function call */
BPF_EXIT  0x90  /* eBPF BPF_JMP only: function return */
BPF_JLT   0xa0  /* eBPF only: unsigned '<' */
BPF_JLE   0xb0  /* eBPF only: unsigned '<=' */
BPF_JSLT  0xc0  /* eBPF only: signed '<' */
BPF_JSLE  0xd0  /* eBPF only: signed '<=' */
```

So BPF_ADD | BPF_X | BPF_ALU means 32-bit addition in both classic BPF and eBPF. There are only two registers in classic BPF, so it means A += X. In eBPF it means dst_reg = (u32) dst_reg + (u32) src_reg; similarly, BPF_XOR | BPF_K | BPF_ALU means A ^= imm32 in classic BPF and analogous src_reg = (u32) src_reg ^ (u32) imm32 in eBPF.

Classic BPF is using BPF_MISC class to represent A = X and X = A moves. eBPF is using BPF_MOV | BPF_X | BPF_ALU code instead. Since there are no BPF_MISC operations in eBPF, the class 7 is used as BPF_ALU64 to mean exactly the same operations as BPF_ALU, but with 64-bit wide operands instead. So BPF_ADD | BPF_X | BPF_ALU64 means 64-bit addition, i.e.: dst_reg = dst_reg + src_reg

Classic BPF wastes the whole BPF_RET class to represent a single `ret` operation. Classic BPF_RET | BPF_K means copy imm32 into return register and perform function exit. eBPF is modeled to match CPU, so BPF_JMP | BPF_EXIT in eBPF means function exit only. The eBPF program needs to store return value into register R0 before doing a BPF_EXIT. Class 6 in eBPF is used as BPF_JMP32 to mean exactly the same operations as BPF_JMP, but with 32-bit wide operands for the comparisons instead.

For load and store instructions the 8-bit ‘code’ field is divided as:

```
+--------+--------+-------------------+
| 3 bits | 2 bits |   3 bits          |
|  mode  |  size  | instruction class |
+--------+--------+-------------------+
(MSB)                            (LSB)
```

Size modifier is one of ...

```
BPF_W   0x00    /* word */
BPF_H   0x08    /* half word */
BPF_B   0x10    /* byte */
BPF_DW  0x18    /* eBPF only, double word */
```

... which encodes size of load/store operation:

```
B  - 1 byte
H  - 2 byte
W  - 4 byte
DW - 8 byte (eBPF only)
```

Mode modifier is one of:

```
BPF_IMM     0x00  /* used for 32-bit mov in classic BPF and 64-bit in eBPF */
BPF_ABS     0x20
BPF_IND     0x40
BPF_MEM     0x60
BPF_LEN     0x80  /* classic BPF only, reserved in eBPF */
BPF_MSH     0xa0  /* classic BPF only, reserved in eBPF */
BPF_ATOMIC  0xc0  /* eBPF only, atomic operations */
```

CHAPTER

FOURTEEN

BPF ITERATORS

14.1 Motivation

There are a few existing ways to dump kernel data into user space. The most popular one is the `/proc` system. For example, `cat /proc/net/tcp6` dumps all tcp6 sockets in the system, and `cat /proc/net/netlink` dumps all netlink sockets in the system. However, their output format tends to be fixed, and if users want more information about these sockets, they have to patch the kernel, which often takes time to publish upstream and release. The same is true for popular tools like ss where any additional information needs a kernel patch.

To solve this problem, the drgn tool is often used to dig out the kernel data with no kernel change. However, the main drawback for drgn is performance, as it cannot do pointer tracing inside the kernel. In addition, drgn cannot validate a pointer value and may read invalid data if the pointer becomes invalid inside the kernel.

The BPF iterator solves the above problem by providing flexibility on what data (e.g., tasks, bpf_maps, etc.) to collect by calling BPF programs for each kernel data object.

14.2 How BPF Iterators Work

A BPF iterator is a type of BPF program that allows users to iterate over specific types of kernel objects. Unlike traditional BPF tracing programs that allow users to define callbacks that are invoked at particular points of execution in the kernel, BPF iterators allow users to define callbacks that should be executed for every entry in a variety of kernel data structures.

For example, users can define a BPF iterator that iterates over every task on the system and dumps the total amount of CPU runtime currently used by each of them. Another BPF task iterator may instead dump the cgroup information for each task. Such flexibility is the core value of BPF iterators.

A BPF program is always loaded into the kernel at the behest of a user space process. A user space process loads a BPF program by opening and initializing the program skeleton as required and then invoking a syscall to have the BPF program verified and loaded by the kernel.

In traditional tracing programs, a program is activated by having user space obtain a `bpf_link` to the program with `bpf_program__attach()`. Once activated, the program callback will be invoked whenever the tracepoint is triggered in the main kernel. For BPF iterator programs, a `bpf_link` to the program is obtained using `bpf_link_create()`, and the program callback is invoked by issuing system calls from user space.

Next, let us see how you can use the iterators to iterate on kernel objects and read data.

14.3 How to Use BPF iterators

BPF selftests are a great resource to illustrate how to use the iterators. In this section, we'll walk through a BPF selftest which shows how to load and use a BPF iterator program. To begin, we'll look at bpf_iter.c, which illustrates how to load and trigger BPF iterators on the user space side. Later, we'll look at a BPF program that runs in kernel space.

Loading a BPF iterator in the kernel from user space typically involves the following steps:

- The BPF program is loaded into the kernel through `libbpf`. Once the kernel has verified and loaded the program, it returns a file descriptor (fd) to user space.
- Obtain a `link_fd` to the BPF program by calling the `bpf_link_create()` specified with the BPF program file descriptor received from the kernel.
- Next, obtain a BPF iterator file descriptor (`bpf_iter_fd`) by calling the `bpf_iter_create()` specified with the `bpf_link` received from Step 2.
- Trigger the iteration by calling `read(bpf_iter_fd)` until no data is available.
- Close the iterator fd using `close(bpf_iter_fd)`.
- If needed to reread the data, get a new `bpf_iter_fd` and do the read again.

The following are a few examples of selftest BPF iterator programs:

- bpf_iter_tcp4.c
- bpf_iter_task_vma.c
- bpf_iter_task_file.c

Let us look at `bpf_iter_task_file.c`, which runs in kernel space:

Here is the definition of `bpf_iter__task_file` in vmlinux.h. Any struct name in `vmlinux.h` in the format `bpf_iter__<iter_name>` represents a BPF iterator. The suffix `<iter_name>` represents the type of iterator.

```
struct bpf_iter__task_file {
        union {
            struct bpf_iter_meta *meta;
        };
        union {
            struct task_struct *task;
        };
        u32 fd;
        union {
            struct file *file;
        };
};
```

In the above code, the field 'meta' contains the metadata, which is the same for all BPF iterator programs. The rest of the fields are specific to different iterators. For example, for task_file iterators, the kernel layer provides the 'task', 'fd' and 'file' field values. The 'task' and 'file' are reference counted, so they won't go away when the BPF program runs.

Here is a snippet from the `bpf_iter_task_file.c` file:

```
SEC("iter/task_file")
int dump_task_file(struct bpf_iter__task_file *ctx)
{
  struct seq_file *seq = ctx->meta->seq;
  struct task_struct *task = ctx->task;
  struct file *file = ctx->file;
  __u32 fd = ctx->fd;

  if (task == NULL || file == NULL)
    return 0;

  if (ctx->meta->seq_num == 0) {
    count = 0;
    BPF_SEQ_PRINTF(seq, "    tgid      gid       fd      file\n");
  }

  if (tgid == task->tgid && task->tgid != task->pid)
    count++;

  if (last_tgid != task->tgid) {
    last_tgid = task->tgid;
    unique_tgid_count++;
  }

  BPF_SEQ_PRINTF(seq, "%8d %8d %8d %lx\n", task->tgid, task->pid, fd,
         (long)file->f_op);
  return 0;
}
```

In the above example, the section name `SEC(iter/task_file)`, indicates that the program is a BPF iterator program to iterate all files from all tasks. The context of the program is `bpf_iter__task_file` struct.

The user space program invokes the BPF iterator program running in the kernel by issuing a `read()` syscall. Once invoked, the BPF program can export data to user space using a variety of BPF helper functions. You can use either `bpf_seq_printf()` (and BPF_SEQ_PRINTF helper macro) or `bpf_seq_write()` function based on whether you need formatted output or just binary data, respectively. For binary-encoded data, the user space applications can process the data from `bpf_seq_write()` as needed. For the formatted data, you can use `cat <path>` to print the results similar to `cat /proc/net/netlink` after pinning the BPF iterator to the bpffs mount. Later, use `rm -f <path>` to remove the pinned iterator.

For example, you can use the following command to create a BPF iterator from the `bpf_iter_ipv6_route.o` object file and pin it to the `/sys/fs/bpf/my_route` path:

```
$ bpftool iter pin ./bpf_iter_ipv6_route.o  /sys/fs/bpf/my_route
```

And then print out the results using the following command:

```
$ cat /sys/fs/bpf/my_route
```

14.4 Implement Kernel Support for BPF Iterator Program Types

To implement a BPF iterator in the kernel, the developer must make a one-time change to the following key data structure defined in the bpf.h file.

```
struct bpf_iter_reg {
          const char *target;
          bpf_iter_attach_target_t attach_target;
          bpf_iter_detach_target_t detach_target;
          bpf_iter_show_fdinfo_t show_fdinfo;
          bpf_iter_fill_link_info_t fill_link_info;
          bpf_iter_get_func_proto_t get_func_proto;
          u32 ctx_arg_info_size;
          u32 feature;
          struct bpf_ctx_arg_aux ctx_arg_info[BPF_ITER_CTX_ARG_MAX];
          const struct bpf_iter_seq_info *seq_info;
};
```

After filling the data structure fields, call `bpf_iter_reg_target()` to register the iterator to the main BPF iterator subsystem.

The following is the breakdown for each field in struct `bpf_iter_reg`.

Fields	Description
target	Specifies the name of the BPF iterator. For example: `bpf_map`, `bpf_map_elem`. The name should be different from other `bpf_iter` target names in the kernel.
attach_target and detach_target	Allows for target specific `link_create` action since some targets may need special processing. Called during the user space link_create stage.
show_fdinfo and fill_link_info	Called to fill target specific information when user tries to get link info associated with the iterator.
get_func_proto	Permits a BPF iterator to access BPF helpers specific to the iterator.
ctx_arg_info_size and ctx_arg_info	Specifies the verifier states for BPF program arguments associated with the bpf iterator.
feature	Specifies certain action requests in the kernel BPF iterator infrastructure. Currently, only BPF_ITER_RESCHED is supported. This means that the kernel function cond_resched() is called to avoid other kernel subsystem (e.g., rcu) misbehaving.
seq_info	Specifies the set of seq operations for the BPF iterator and helpers to initialize/free the private data for the corresponding `seq_file`.

Click here to see an implementation of the `task_vma` BPF iterator in the kernel.

14.5 Parameterizing BPF Task Iterators

By default, BPF iterators walk through all the objects of the specified types (processes, cgroups, maps, etc.) across the entire system to read relevant kernel data. But often, there are cases where we only care about a much smaller subset of iterable kernel objects, such as only iterating tasks within a specific process. Therefore, BPF iterator programs support filtering out objects from iteration by allowing user space to configure the iterator program when it is attached.

14.6 BPF Task Iterator Program

The following code is a BPF iterator program to print files and task information through the `seq_file` of the iterator. It is a standard BPF iterator program that visits every file of an iterator. We will use this BPF program in our example later.

```
#include <vmlinux.h>
#include <bpf/bpf_helpers.h>

char _license[] SEC("license") = "GPL";

SEC("iter/task_file")
int dump_task_file(struct bpf_iter__task_file *ctx)
{
      struct seq_file *seq = ctx->meta->seq;
      struct task_struct *task = ctx->task;
      struct file *file = ctx->file;
      __u32 fd = ctx->fd;
      if (task == NULL || file == NULL)
              return 0;
      if (ctx->meta->seq_num == 0) {
              BPF_SEQ_PRINTF(seq, "    tgid      pid       fd      file\n");
      }
      BPF_SEQ_PRINTF(seq, "%8d %8d %8d %lx\n", task->tgid, task->pid, fd,
                     (long)file->f_op);
      return 0;
}
```

14.7 Creating a File Iterator with Parameters

Now, let us look at how to create an iterator that includes only files of a process.

First, fill the `bpf_iter_attach_opts` struct as shown below:

```
LIBBPF_OPTS(bpf_iter_attach_opts, opts);
union bpf_iter_link_info linfo;
memset(&linfo, 0, sizeof(linfo));
linfo.task.pid = getpid();
opts.link_info = &linfo;
opts.link_info_len = sizeof(linfo);
```

`linfo.task.pid`, if it is non-zero, directs the kernel to create an iterator that only includes opened files for the process with the specified `pid`. In this example, we will only be iterating files for our process. If `linfo.task.pid` is zero, the iterator will visit every opened file of every process. Similarly, `linfo.task.tid` directs the kernel to create an iterator that visits opened files of a specific thread, not a process. In this example, `linfo.task.tid` is different from `linfo.task.pid` only if the thread has a separate file descriptor table. In most circumstances, all process threads share a single file descriptor table.

Now, in the userspace program, pass the pointer of struct to the `bpf_program__attach_iter()`.

```
link = bpf_program__attach_iter(prog, &opts); iter_fd =
bpf_iter_create(bpf_link__fd(link));
```

If both *tid* and *pid* are zero, an iterator created from this struct `bpf_iter_attach_opts` will include every opened file of every task in the system (in the namespace, actually.) It is the same as passing a NULL as the second argument to `bpf_program__attach_iter()`.

The whole program looks like the following code:

```
#include <stdio.h>
#include <unistd.h>
#include <bpf/bpf.h>
#include <bpf/libbpf.h>
#include "bpf_iter_task_ex.skel.h"

static int do_read_opts(struct bpf_program *prog, struct bpf_iter_attach_opts␣
↪*opts)
{
      struct bpf_link *link;
      char buf[16] = {};
      int iter_fd = -1, len;
      int ret = 0;

      link = bpf_program__attach_iter(prog, opts);
      if (!link) {
              fprintf(stderr, "bpf_program__attach_iter() fails\n");
              return -1;
      }
      iter_fd = bpf_iter_create(bpf_link__fd(link));
      if (iter_fd < 0) {
              fprintf(stderr, "bpf_iter_create() fails\n");
              ret = -1;
              goto free_link;
      }
      /* not check contents, but ensure read() ends without error */
      while ((len = read(iter_fd, buf, sizeof(buf) - 1)) > 0) {
              buf[len] = 0;
              printf("%s", buf);
      }
      printf("\n");
free_link:
      if (iter_fd >= 0)
              close(iter_fd);
```

```
      bpf_link__destroy(link);
      return 0;
}

static void test_task_file(void)
{
      LIBBPF_OPTS(bpf_iter_attach_opts, opts);
      struct bpf_iter_task_ex *skel;
      union bpf_iter_link_info linfo;
      skel = bpf_iter_task_ex__open_and_load();
      if (skel == NULL)
              return;
      memset(&linfo, 0, sizeof(linfo));
      linfo.task.pid = getpid();
      opts.link_info = &linfo;
      opts.link_info_len = sizeof(linfo);
      printf("PID %d\n", getpid());
      do_read_opts(skel->progs.dump_task_file, &opts);
      bpf_iter_task_ex__destroy(skel);
}

int main(int argc, const char * const * argv)
{
      test_task_file();
      return 0;
}
```

The following lines are the output of the program.

```
PID 1859

   tgid      pid       fd      file
   1859     1859        0 ffffffff82270aa0
   1859     1859        1 ffffffff82270aa0
   1859     1859        2 ffffffff82270aa0
   1859     1859        3 ffffffff82272980
   1859     1859        4 ffffffff8225e120
   1859     1859        5 ffffffff82255120
   1859     1859        6 ffffffff82254f00
   1859     1859        7 ffffffff82254d80
   1859     1859        8 ffffffff8225abe0
```

14.8 Without Parameters

Let us look at how a BPF iterator without parameters skips files of other processes in the system. In this case, the BPF program has to check the pid or the tid of tasks, or it will receive every opened file in the system (in the current *pid* namespace, actually). So, we usually add a global variable in the BPF program to pass a *pid* to the BPF program.

The BPF program would look like the following block.

```
......
int target_pid = 0;

SEC("iter/task_file")
int dump_task_file(struct bpf_iter__task_file *ctx)
{
      ......
      if (task->tgid != target_pid) /* Check task->pid instead to␣
↪check thread IDs */
              return 0;
      BPF_SEQ_PRINTF(seq, "%8d %8d %8d %lx\n", task->tgid, task->pid,␣
↪fd,
                     (long)file->f_op);
      return 0;
}
```

The user space program would look like the following block:

```
......
static void test_task_file(void)
{
      ......
      skel = bpf_iter_task_ex__open_and_load();
      if (skel == NULL)
              return;
      skel->bss->target_pid = getpid(); /* process ID.  For thread id,␣
↪use gettid() */
      memset(&linfo, 0, sizeof(linfo));
      linfo.task.pid = getpid();
      opts.link_info = &linfo;
      opts.link_info_len = sizeof(linfo);
      ......
}
```

`target_pid` is a global variable in the BPF program. The user space program should initialize the variable with a process ID to skip opened files of other processes in the BPF program. When you parametrize a BPF iterator, the iterator calls the BPF program fewer times which can save significant resources.

14.9 Parametrizing VMA Iterators

By default, a BPF VMA iterator includes every VMA in every process. However, you can still specify a process or a thread to include only its VMAs. Unlike files, a thread can not have a separate address space (since Linux 2.6.0-test6). Here, using *tid* makes no difference from using *pid*.

14.10 Parametrizing Task Iterators

A BPF task iterator with *pid* includes all tasks (threads) of a process. The BPF program receives these tasks one after another. You can specify a BPF task iterator with *tid* parameter to include only the tasks that match the given *tid*.

CHAPTER

FIFTEEN

BPF LICENSING

15.1 Background

- Classic BPF was BSD licensed

“BPF” was originally introduced as BSD Packet Filter in http://www.tcpdump.org/papers/bpf-usenix93.pdf. The corresponding instruction set and its implementation came from BSD with BSD license. That original instruction set is now known as “classic BPF”.

However an instruction set is a specification for machine-language interaction, similar to a programming language. It is not a code. Therefore, the application of a BSD license may be misleading in a certain context, as the instruction set may enjoy no copyright protection.

- eBPF (extended BPF) instruction set continues to be BSD

In 2014, the classic BPF instruction set was significantly extended. We typically refer to this instruction set as eBPF to disambiguate it from cBPF. The eBPF instruction set is still BSD licensed.

15.2 Implementations of eBPF

Using the eBPF instruction set requires implementing code in both kernel space and user space.

15.2.1 In Linux Kernel

The reference implementations of the eBPF interpreter and various just-in-time compilers are part of Linux and are GPLv2 licensed. The implementation of eBPF helper functions is also GPLv2 licensed. Interpreters, JITs, helpers, and verifiers are called eBPF runtime.

15.2.2 In User Space

There are also implementations of eBPF runtime (interpreter, JITs, helper functions) under Apache2 (https://github.com/iovisor/ubpf), MIT (https://github.com/qmonnet/rbpf), and BSD (https://github.com/DPDK/dpdk/blob/main/lib/librte_bpf).

15.2.3 In HW

The HW can choose to execute eBPF instruction natively and provide eBPF runtime in HW or via the use of implementing firmware with a proprietary license.

15.2.4 In other operating systems

Other kernels or user space implementations of eBPF instruction set and runtime can have proprietary licenses.

15.3 Using BPF programs in the Linux kernel

Linux Kernel (while being GPLv2) allows linking of proprietary kernel modules under these rules: Documentation/process/license-rules.rst

When a kernel module is loaded, the linux kernel checks which functions it intends to use. If any function is marked as “GPL only,” the corresponding module or program has to have GPL compatible license.

Loading BPF program into the Linux kernel is similar to loading a kernel module. BPF is loaded at run time and not statically linked to the Linux kernel. BPF program loading follows the same license checking rules as kernel modules. BPF programs can be proprietary if they don’t use “GPL only” BPF helper functions.

Further, some BPF program types - Linux Security Modules (LSM) and TCP Congestion Control (struct_ops), as of Aug 2021 - are required to be GPL compatible even if they don’t use “GPL only” helper functions directly. The registration step of LSM and TCP congestion control modules of the Linux kernel is done through EXPORT_SYMBOL_GPL kernel functions. In that sense LSM and struct_ops BPF programs are implicitly calling “GPL only” functions. The same restriction applies to BPF programs that call kernel functions directly via unstable interface also known as “kfunc”.

15.4 Packaging BPF programs with user space applications

Generally, proprietary-licensed applications and GPL licensed BPF programs written for the Linux kernel in the same package can co-exist because they are separate executable processes. This applies to both cBPF and eBPF programs.

CHAPTER

SIXTEEN

TESTING AND DEBUGGING BPF

16.1 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.

16.1.1 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,
        .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,
                .func = (void (*)(struct callback_head *))0x0,
        },
}
```

16.2 Testing BPF on s390

16.2.1 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.

16.2.2 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 \
  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
```

16.2.3 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
```

16.2.4 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
```

16.2.5 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
```

16.2.6 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
```

16.2.7 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

Contents

CHAPTER

SEVENTEEN

1 CLANG IMPLEMENTATION NOTES

This document provides more details specific to the Clang/LLVM implementation of the eBPF instruction set.

17.1 1.1 Versions

Clang defined "CPU" versions, where a CPU version of 3 corresponds to the current eBPF ISA.

Clang can select the eBPF ISA version using `-mcpu=v3` for example to select version 3.

17.2 1.2 Arithmetic instructions

For CPU versions prior to 3, Clang v7.0 and later can enable `BPF_ALU` support with `-Xclang -target-feature -Xclang +alu32`. In CPU version 3, support is automatically included.

17.3 1.3 Jump instructions

If `-O0` is used, Clang will generate the `BPF_CALL | BPF_X | BPF_JMP` (0x8d) instruction, which is not supported by the Linux kernel verifier.

17.4 1.4 Atomic operations

Clang can generate atomic instructions by default when `-mcpu=v3` is enabled. If a lower version for `-mcpu` is set, the only atomic instruction Clang can generate is `BPF_ADD` *without* `BPF_FETCH`. If you need to enable the atomics features, while keeping a lower `-mcpu` version, you can use `-Xclang -target-feature -Xclang +alu32`.

Contents

CHAPTER

EIGHTEEN

1 LINUX IMPLEMENTATION NOTES

This document provides more details specific to the Linux kernel implementation of the eBPF instruction set.

18.1 1.1 Byte swap instructions

`BPF_FROM_LE` and `BPF_FROM_BE` exist as aliases for `BPF_TO_LE` and `BPF_TO_BE` respectively.

18.2 1.2 Jump instructions

`BPF_CALL | BPF_X | BPF_JMP` (0x8d), where the helper function integer would be read from a specified register, is not currently supported by the verifier. Any programs with this instruction will fail to load until such support is added.

18.3 1.3 Maps

Linux only supports the 'map_val(map)' operation on array maps with a single element.

Linux uses an fd_array to store maps associated with a BPF program. Thus, map_by_idx(imm) uses the fd at that index in the array.

18.4 1.4 Variables

The following 64-bit immediate instruction specifies that a variable address, which corresponds to some integer stored in the 'imm' field, should be loaded:

opcode construction	opcode	src	pseudocode	imm type	dst type
BPF_IMM | BPF_DW | BPF_LD	0x18	0x3	dst = var_addr(imm)	variable id	data pointer

On Linux, this integer is a BTF ID.

18.5 1.5 Legacy BPF Packet access instructions

As mentioned in the ISA standard documentation, Linux has special eBPF instructions for access to packet data that have been carried over from classic BPF to retain the performance of legacy socket filters running in the eBPF interpreter.

The instructions come in two forms: `BPF_ABS | <size> | BPF_LD` and `BPF_IND | <size> | BPF_LD`.

These instructions are used to access packet data and can only be used when the program context is a pointer to a networking packet. `BPF_ABS` accesses packet data at an absolute offset specified by the immediate data and `BPF_IND` access packet data at an offset that includes the value of a register in addition to the immediate data.

These instructions have seven implicit operands:

- Register R6 is an implicit input that must contain a pointer to a struct sk_buff.
- Register R0 is an implicit output which contains the data fetched from the packet.
- Registers R1-R5 are scratch registers that are clobbered by the instruction.

These instructions have an implicit program exit condition as well. If an eBPF program attempts access data beyond the packet boundary, the program execution will be aborted.

`BPF_ABS | BPF_W | BPF_LD` (0x20) means:

```
R0 = ntohl(*(u32 *) ((struct sk_buff *) R6->data + imm))
```

where `ntohl()` converts a 32-bit value from network byte order to host byte order.

`BPF_IND | BPF_W | BPF_LD` (0x40) means:

```
R0 = ntohl(*(u32 *) ((struct sk_buff *) R6->data + src + imm))
```

CHAPTER

NINETEEN

OTHER

19.1 BPF ring buffer

This document describes BPF ring buffer design, API, and implementation details.

19.1.1 Motivation

There are two distinctive motivators for this work, which are not satisfied by existing perf buffer, which prompted creation of a new ring buffer implementation.

- more efficient memory utilization by sharing ring buffer across CPUs;
- preserving ordering of events that happen sequentially in time, even across multiple CPUs (e.g., fork/exec/exit events for a task).

These two problems are independent, but perf buffer fails to satisfy both. Both are a result of a choice to have per-CPU perf ring buffer. Both can be also solved by having an MPSC implementation of ring buffer. The ordering problem could technically be solved for perf buffer with some in-kernel counting, but given the first one requires an MPSC buffer, the same solution would solve the second problem automatically.

19.1.2 Semantics and APIs

Single ring buffer is presented to BPF programs as an instance of BPF map of type `BPF_MAP_TYPE_RINGBUF`. Two other alternatives considered, but ultimately rejected.

One way would be to, similar to `BPF_MAP_TYPE_PERF_EVENT_ARRAY`, make `BPF_MAP_TYPE_RINGBUF` could represent an array of ring buffers, but not enforce "same CPU only" rule. This would be more familiar interface compatible with existing perf buffer use in BPF, but would fail if application needed more advanced logic to lookup ring buffer by arbitrary key. `BPF_MAP_TYPE_HASH_OF_MAPS` addresses this with current approach. Additionally, given the performance of BPF ringbuf, many use cases would just opt into a simple single ring buffer shared among all CPUs, for which current approach would be an overkill.

Another approach could introduce a new concept, alongside BPF map, to represent generic "container" object, which doesn't necessarily have key/value interface with lookup/update/delete operations. This approach would add a lot of extra infrastructure that has to be built for observability and verifier support. It would also add another concept that BPF developers would have to familiarize themselves with, new syntax in libbpf, etc. But then would really provide no additional benefits over the approach of using a map. BPF_MAP_TYPE_RINGBUF doesn't support lookup/update/delete operations, but so doesn't few other map types (e.g., queue and stack; array doesn't support delete, etc).

The approach chosen has an advantage of re-using existing BPF map infrastructure (introspection APIs in kernel, libbpf support, etc), being familiar concept (no need to teach users a new type of object in BPF program), and utilizing existing tooling (bpftool). For common scenario of using a single ring buffer for all CPUs, it's as simple and straightforward, as would be with a dedicated "container" object. On the other hand, by being a map, it can be combined with ARRAY_OF_MAPS and HASH_OF_MAPS map-in-maps to implement a wide variety of topologies, from one ring buffer for each CPU (e.g., as a replacement for perf buffer use cases), to a complicated application hashing/sharding of ring buffers (e.g., having a small pool of ring buffers with hashed task's tgid being a look up key to preserve order, but reduce contention).

Key and value sizes are enforced to be zero. max_entries is used to specify the size of ring buffer and has to be a power of 2 value.

There are a bunch of similarities between perf buffer (BPF_MAP_TYPE_PERF_EVENT_ARRAY) and new BPF ring buffer semantics:

- variable-length records;
- if there is no more space left in ring buffer, reservation fails, no blocking;
- memory-mappable data area for user-space applications for ease of consumption and high performance;
- epoll notifications for new incoming data;
- but still the ability to do busy polling for new data to achieve the lowest latency, if necessary.

BPF ringbuf provides two sets of APIs to BPF programs:

- bpf_ringbuf_output() allows to *copy* data from one place to a ring buffer, similarly to bpf_perf_event_output();
- bpf_ringbuf_reserve()/bpf_ringbuf_commit()/bpf_ringbuf_discard() APIs split the whole process into two steps. First, a fixed amount of space is reserved. If successful, a pointer to a data inside ring buffer data area is returned, which BPF programs can use similarly to a data inside array/hash maps. Once ready, this piece of memory is either committed or discarded. Discard is similar to commit, but makes consumer ignore the record.

bpf_ringbuf_output() has disadvantage of incurring extra memory copy, because record has to be prepared in some other place first. But it allows to submit records of the length that's not known to verifier beforehand. It also closely matches bpf_perf_event_output(), so will simplify migration significantly.

bpf_ringbuf_reserve() avoids the extra copy of memory by providing a memory pointer directly to ring buffer memory. In a lot of cases records are larger than BPF stack space allows, so many programs have use extra per-CPU array as a temporary heap for preparing sample. bpf_ringbuf_reserve() avoid this needs completely. But in exchange, it only allows a known constant size of memory to be reserved, such that verifier can verify that BPF program can't access

memory outside its reserved record space. bpf_ringbuf_output(), while slightly slower due to extra memory copy, covers some use cases that are not suitable for `bpf_ringbuf_reserve()`.

The difference between commit and discard is very small. Discard just marks a record as discarded, and such records are supposed to be ignored by consumer code. Discard is useful for some advanced use-cases, such as ensuring all-or-nothing multi-record submission, or emulating temporary `malloc()/free()` within single BPF program invocation.

Each reserved record is tracked by verifier through existing reference-tracking logic, similar to socket ref-tracking. It is thus impossible to reserve a record, but forget to submit (or discard) it.

`bpf_ringbuf_query()` helper allows to query various properties of ring buffer. Currently 4 are supported:

- `BPF_RB_AVAIL_DATA` returns amount of unconsumed data in ring buffer;
- `BPF_RB_RING_SIZE` returns the size of ring buffer;
- `BPF_RB_CONS_POS/BPF_RB_PROD_POS` returns current logical position of consumer/producer, respectively.

Returned values are momentarily snapshots of ring buffer state and could be off by the time helper returns, so this should be used only for debugging/reporting reasons or for implementing various heuristics, that take into account highly-changeable nature of some of those characteristics.

One such heuristic might involve more fine-grained control over poll/epoll notifications about new data availability in ring buffer. Together with `BPF_RB_NO_WAKEUP/BPF_RB_FORCE_WAKEUP` flags for output/commit/discard helpers, it allows BPF program a high degree of control and, e.g., more efficient batched notifications. Default self-balancing strategy, though, should be adequate for most applications and will work reliable and efficiently already.

19.1.3 Design and Implementation

This reserve/commit schema allows a natural way for multiple producers, either on different CPUs or even on the same CPU/in the same BPF program, to reserve independent records and work with them without blocking other producers. This means that if BPF program was interrupted by another BPF program sharing the same ring buffer, they will both get a record reserved (provided there is enough space left) and can work with it and submit it independently. This applies to NMI context as well, except that due to using a spinlock during reservation, in NMI context, `bpf_ringbuf_reserve()` might fail to get a lock, in which case reservation will fail even if ring buffer is not full.

The ring buffer itself internally is implemented as a power-of-2 sized circular buffer, with two logical and ever-increasing counters (which might wrap around on 32-bit architectures, that's not a problem):

- consumer counter shows up to which logical position consumer consumed the data;
- producer counter denotes amount of data reserved by all producers.

Each time a record is reserved, producer that "owns" the record will successfully advance producer counter. At that point, data is still not yet ready to be consumed, though. Each record has 8 byte header, which contains the length of reserved record, as well as two extra bits: busy bit to denote that record is still being worked on, and discard bit, which might be set at commit time if record is discarded. In the latter case, consumer is supposed to skip the record and move on

to the next one. Record header also encodes record's relative offset from the beginning of ring buffer data area (in pages). This allows `bpf_ringbuf_commit()/bpf_ringbuf_discard()` to accept only the pointer to the record itself, without requiring also the pointer to ring buffer itself. Ring buffer memory location will be restored from record metadata header. This significantly simplifies verifier, as well as improving API usability.

Producer counter increments are serialized under spinlock, so there is a strict ordering between reservations. Commits, on the other hand, are completely lockless and independent. All records become available to consumer in the order of reservations, but only after all previous records where already committed. It is thus possible for slow producers to temporarily hold off submitted records, that were reserved later.

One interesting implementation bit, that significantly simplifies (and thus speeds up as well) implementation of both producers and consumers is how data area is mapped twice contiguously back-to-back in the virtual memory. This allows to not take any special measures for samples that have to wrap around at the end of the circular buffer data area, because the next page after the last data page would be first data page again, and thus the sample will still appear completely contiguous in virtual memory. See comment and a simple ASCII diagram showing this visually in `bpf_ringbuf_area_alloc()`.

Another feature that distinguishes BPF ringbuf from perf ring buffer is a self-pacing notifications of new data being availability. `bpf_ringbuf_commit()` implementation will send a notification of new record being available after commit only if consumer has already caught up right up to the record being committed. If not, consumer still has to catch up and thus will see new data anyways without needing an extra poll notification. Benchmarks (see tools/testing/selftests/bpf/benchs/bench_ringbufs.c) show that this allows to achieve a very high throughput without having to resort to tricks like "notify only every Nth sample", which are necessary with perf buffer. For extreme cases, when BPF program wants more manual control of notifications, commit/discard/output helpers accept `BPF_RB_NO_WAKEUP` and `BPF_RB_FORCE_WAKEUP` flags, which give full control over notifications of data availability, but require extra caution and diligence in using this API.

19.2 BPF LLVM Relocations

This document describes LLVM BPF backend relocation types.

19.2.1 Relocation Record

LLVM BPF backend records each relocation with the following 16-byte ELF structure:

```
typedef struct
{
  Elf64_Addr    r_offset;  // Offset from the beginning of section.
  Elf64_Xword   r_info;    // Relocation type and symbol index.
} Elf64_Rel;
```

For example, for the following code:

```
int g1 __attribute__((section("sec")));
int g2 __attribute__((section("sec")));
static volatile int l1 __attribute__((section("sec")));
```

```
static volatile int l2 __attribute__((section("sec")));
int test() {
  return g1 + g2 + l1 + l2;
}
```

Compiled with `clang --target=bpf -O2 -c test.c`, the following is the code with `llvm-objdump -dr test.o`:

```
 0:       18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0 ll
          0000000000000000:  R_BPF_64_64  g1
 2:       61 11 00 00 00 00 00 00 r1 = *(u32 *)(r1 + 0)
 3:       18 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r2 = 0 ll
          0000000000000018:  R_BPF_64_64  g2
 5:       61 20 00 00 00 00 00 00 r0 = *(u32 *)(r2 + 0)
 6:       0f 10 00 00 00 00 00 00 r0 += r1
 7:       18 01 00 00 08 00 00 00 00 00 00 00 00 00 00 00 r1 = 8 ll
          0000000000000038:  R_BPF_64_64  sec
 9:       61 11 00 00 00 00 00 00 r1 = *(u32 *)(r1 + 0)
10:       0f 10 00 00 00 00 00 00 r0 += r1
11:       18 01 00 00 0c 00 00 00 00 00 00 00 00 00 00 00 r1 = 12 ll
          0000000000000058:  R_BPF_64_64  sec
13:       61 11 00 00 00 00 00 00 r1 = *(u32 *)(r1 + 0)
14:       0f 10 00 00 00 00 00 00 r0 += r1
15:       95 00 00 00 00 00 00 00 exit
```

There are four relocations in the above for four `LD_imm64` instructions. The following `llvm-readelf -r test.o` shows the binary values of the four relocations:

```
Relocation section '.rel.text' at offset 0x190 contains 4 entries:
    Offset             Info             Type               Symbol's Value ␣
↪Symbol's Name
0000000000000000  0000000600000001 R_BPF_64_64            0000000000000000 g1
0000000000000018  0000000700000001 R_BPF_64_64            0000000000000004 g2
0000000000000038  0000000400000001 R_BPF_64_64            0000000000000000 sec
0000000000000058  0000000400000001 R_BPF_64_64            0000000000000000 sec
```

Each relocation is represented by `Offset` (8 bytes) and `Info` (8 bytes). For example, the first relocation corresponds to the first instruction (Offset 0x0) and the corresponding `Info` indicates the relocation type of `R_BPF_64_64` (type 1) and the entry in the symbol table (entry 6). The following is the symbol table with `llvm-readelf -s test.o`:

```
Symbol table '.symtab' contains 8 entries:
   Num:    Value          Size Type    Bind   Vis       Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT   ABS test.c
     2: 0000000000000008     4 OBJECT  LOCAL  DEFAULT     4 l1
     3: 000000000000000c     4 OBJECT  LOCAL  DEFAULT     4 l2
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT     4 sec
     5: 0000000000000000   128 FUNC    GLOBAL DEFAULT     2 test
     6: 0000000000000000     4 OBJECT  GLOBAL DEFAULT     4 g1
     7: 0000000000000004     4 OBJECT  GLOBAL DEFAULT     4 g2
```

The 6th entry is global variable g1 with value 0.

Similarly, the second relocation is at .text offset 0x18, instruction 3, has a type of R_BPF_64_64 and refers to entry 7 in the symbol table. The second relocation resolves to global variable g2 which has a symbol value 4. The symbol value represents the offset from the start of .data section where the initial value of the global variable g2 is stored.

The third and fourth relocations refer to static variables l1 and l2. From the .rel.text section above, it is not clear to which symbols they really refer as they both refer to symbol table entry 4, symbol sec, which has STT_SECTION type and represents a section. So for a static variable or function, the section offset is written to the original insn buffer, which is called A (addend). Looking at above insn 7 and 11, they have section offset 8 and 12. From symbol table, we can find that they correspond to entries 2 and 3 for l1 and l2.

In general, the A is 0 for global variables and functions, and is the section offset or some computation result based on section offset for static variables/functions. The non-section-offset case refers to function calls. See below for more details.

19.2.2 Different Relocation Types

Six relocation types are supported. The following is an overview and S represents the value of the symbol in the symbol table:

```
Enum  ELF Reloc Type     Description      BitSize  Offset        Calculation
0     R_BPF_NONE         None
1     R_BPF_64_64        ld_imm64 insn    32       r_offset + 4  S + A
2     R_BPF_64_ABS64     normal data      64       r_offset      S + A
3     R_BPF_64_ABS32     normal data      32       r_offset      S + A
4     R_BPF_64_NODYLD32  .BTF[.ext] data  32       r_offset      S + A
10    R_BPF_64_32        call insn        32       r_offset + 4  (S + A) / 8 -␣
↪1
```

For example, R_BPF_64_64 relocation type is used for ld_imm64 instruction. The actual to-be-relocated data (0 or section offset) is stored at r_offset + 4 and the read/write data bitsize is 32 (4 bytes). The relocation can be resolved with the symbol value plus implicit addend. Note that the BitSize is 32 which means the section offset must be less than or equal to UINT32_MAX and this is enforced by LLVM BPF backend.

In another case, R_BPF_64_ABS64 relocation type is used for normal 64-bit data. The actual to-be-relocated data is stored at r_offset and the read/write data bitsize is 64 (8 bytes). The relocation can be resolved with the symbol value plus implicit addend.

Both R_BPF_64_ABS32 and R_BPF_64_NODYLD32 types are for 32-bit data. But R_BPF_64_NODYLD32 specifically refers to relocations in .BTF and .BTF.ext sections. For cases like bcc where llvm ExecutionEngine RuntimeDyld is involved, R_BPF_64_NODYLD32 types of relocations should not be resolved to actual function/variable address. Otherwise, .BTF and .BTF.ext become unusable by bcc and kernel.

Type R_BPF_64_32 is used for call instruction. The call target section offset is stored at r_offset + 4 (32bit) and calculated as (S + A) / 8 - 1.

19.2.3 Examples

Types R_BPF_64_64 and R_BPF_64_32 are used to resolve ld_imm64 and call instructions. For example:

```
__attribute__((noinline)) __attribute__((section("sec1")))
int gfunc(int a, int b) {
  return a * b;
}
static __attribute__((noinline)) __attribute__((section("sec1")))
int lfunc(int a, int b) {
  return a + b;
}
int global __attribute__((section("sec2")));
int test(int a, int b) {
  return gfunc(a, b) +  lfunc(a, b) + global;
}
```

Compiled with `clang --target=bpf -O2 -c test.c`, we will have following code with *llvm-objdump -dr test.o*`:

```
Disassembly of section .text:

0000000000000000 <test>:
       0:       bf 26 00 00 00 00 00 00 r6 = r2
       1:       bf 17 00 00 00 00 00 00 r7 = r1
       2:       85 10 00 00 ff ff ff ff call -1
                0000000000000010:  R_BPF_64_32  gfunc
       3:       bf 08 00 00 00 00 00 00 r8 = r0
       4:       bf 71 00 00 00 00 00 00 r1 = r7
       5:       bf 62 00 00 00 00 00 00 r2 = r6
       6:       85 10 00 00 02 00 00 00 call 2
                0000000000000030:  R_BPF_64_32  sec1
       7:       0f 80 00 00 00 00 00 00 r0 += r8
       8:       18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0 ll
                0000000000000040:  R_BPF_64_64  global
      10:       61 11 00 00 00 00 00 00 r1 = *(u32 *)(r1 + 0)
      11:       0f 10 00 00 00 00 00 00 r0 += r1
      12:       95 00 00 00 00 00 00 00 exit

Disassembly of section sec1:

0000000000000000 <gfunc>:
       0:       bf 20 00 00 00 00 00 00 r0 = r2
       1:       2f 10 00 00 00 00 00 00 r0 *= r1
       2:       95 00 00 00 00 00 00 00 exit

0000000000000018 <lfunc>:
       3:       bf 20 00 00 00 00 00 00 r0 = r2
       4:       0f 10 00 00 00 00 00 00 r0 += r1
       5:       95 00 00 00 00 00 00 00 exit
```

The first relocation corresponds to `gfunc(a, b)` where `gfunc` has a value of 0, so the `call` instruction offset is `(0 + 0)/8 - 1 = -1`. The second relocation corresponds to `lfunc(a, b)` where `lfunc` has a section offset `0x18`, so the `call` instruction offset is `(0 + 0x18)/8 - 1 = 2`. The third relocation corresponds to ld_imm64 of `global`, which has a section offset `0`.

The following is an example to show how R_BPF_64_ABS64 could be generated:

```
int global() { return 0; }
struct t { void *g; } gbl = { global };
```

Compiled with `clang --target=bpf -O2 -g -c test.c`, we will see a relocation below in `.data` section with command `llvm-readelf -r test.o`:

```
Relocation section '.rel.data' at offset 0x458 contains 1 entries:
    Offset             Info             Type               Symbol's Value ␣
 ↪Symbol's Name
0000000000000000  0000000700000002 R_BPF_64_ABS64         0000000000000000␣
 ↪global
```

The relocation says the first 8-byte of `.data` section should be filled with address of `global` variable.

With `llvm-readelf` output, we can see that dwarf sections have a bunch of `R_BPF_64_ABS32` and `R_BPF_64_ABS64` relocations:

```
Relocation section '.rel.debug_info' at offset 0x468 contains 13 entries:
    Offset             Info             Type               Symbol's Value ␣
 ↪Symbol's Name
0000000000000006  0000000300000003 R_BPF_64_ABS32         0000000000000000 .
 ↪debug_abbrev
000000000000000c  0000000400000003 R_BPF_64_ABS32         0000000000000000 .
 ↪debug_str
0000000000000012  0000000400000003 R_BPF_64_ABS32         0000000000000000 .
 ↪debug_str
0000000000000016  0000000600000003 R_BPF_64_ABS32         0000000000000000 .
 ↪debug_line
000000000000001a  0000000400000003 R_BPF_64_ABS32         0000000000000000 .
 ↪debug_str
000000000000001e  0000000200000002 R_BPF_64_ABS64         0000000000000000 .
 ↪text
000000000000002b  0000000400000003 R_BPF_64_ABS32         0000000000000000 .
 ↪debug_str
0000000000000037  0000000800000002 R_BPF_64_ABS64         0000000000000000 gbl
0000000000000040  0000000400000003 R_BPF_64_ABS32         0000000000000000 .
 ↪debug_str
......
```

The .BTF/.BTF.ext sections has R_BPF_64_NODYLD32 relocations:

```
Relocation section '.rel.BTF' at offset 0x538 contains 1 entries:
    Offset             Info             Type               Symbol's Value ␣
 ↪Symbol's Name
0000000000000084  0000000800000004 R_BPF_64_NODYLD32      0000000000000000 gbl
```

```
Relocation section '.rel.BTF.ext' at offset 0x548 contains 2 entries:
    Offset             Info             Type               Symbol's Value ␣
↪Symbol's Name
000000000000002c  0000000200000004 R_BPF_64_NODYLD32      0000000000000000 .
↪text
0000000000000040  0000000200000004 R_BPF_64_NODYLD32      0000000000000000 .
↪text
```

19.3 CO-RE Relocations

From object file point of view CO-RE mechanism is implemented as a set of CO-RE specific relocation records. These relocation records are not related to ELF relocations and are encoded in .BTF.ext section. See *Documentation/bpf/btf.rst* for more information on .BTF.ext structure.

CO-RE relocations are applied to BPF instructions to update immediate or offset fields of the instruction at load time with information relevant for target kernel.

Field to patch is selected basing on the instruction class:

- For BPF_ALU, BPF_ALU64, BPF_LD *immediate* field is patched;
- For BPF_LDX, BPF_STX, BPF_ST *offset* field is patched;
- BPF_JMP, BPF_JMP32 instructions **should not** be patched.

19.3.1 Relocation kinds

There are several kinds of CO-RE relocations that could be split in three groups:

- Field-based - patch instruction with field related information, e.g. change offset field of the BPF_LDX instruction to reflect offset of a specific structure field in the target kernel.
- Type-based - patch instruction with type related information, e.g. change immediate field of the BPF_ALU move instruction to 0 or 1 to reflect if specific type is present in the target kernel.
- Enum-based - patch instruction with enum related information, e.g. change immediate field of the BPF_LD_IMM64 instruction to reflect value of a specific enum literal in the target kernel.

The complete list of relocation kinds is represented by the following enum:

```
enum bpf_core_relo_kind {
       BPF_CORE_FIELD_BYTE_OFFSET = 0,  /* field byte offset */
       BPF_CORE_FIELD_BYTE_SIZE   = 1,  /* field size in bytes */
       BPF_CORE_FIELD_EXISTS      = 2,  /* field existence in target kernel */
       BPF_CORE_FIELD_SIGNED      = 3,  /* field signedness (0 - unsigned, 1 -␣
↪signed) */
       BPF_CORE_FIELD_LSHIFT_U64  = 4,  /* bitfield-specific left bitshift */
       BPF_CORE_FIELD_RSHIFT_U64  = 5,  /* bitfield-specific right bitshift */
       BPF_CORE_TYPE_ID_LOCAL     = 6,  /* type ID in local BPF object */
       BPF_CORE_TYPE_ID_TARGET    = 7,  /* type ID in target kernel */
```

```
        BPF_CORE_TYPE_EXISTS       = 8,  /* type existence in target kernel */
        BPF_CORE_TYPE_SIZE         = 9,  /* type size in bytes */
        BPF_CORE_ENUMVAL_EXISTS    = 10, /* enum value existence in target␣
↪kernel */
        BPF_CORE_ENUMVAL_VALUE     = 11, /* enum value integer value */
        BPF_CORE_TYPE_MATCHES      = 12, /* type match in target kernel */
};
```

Notes:

- `BPF_CORE_FIELD_LSHIFT_U64` and `BPF_CORE_FIELD_RSHIFT_U64` are supposed to be used to read bitfield values using the following algorithm:

```
// To read bitfield ``f`` from ``struct s``
is_signed = relo(s->f, BPF_CORE_FIELD_SIGNED)
off = relo(s->f, BPF_CORE_FIELD_BYTE_OFFSET)
sz  = relo(s->f, BPF_CORE_FIELD_BYTE_SIZE)
l   = relo(s->f, BPF_CORE_FIELD_LSHIFT_U64)
r   = relo(s->f, BPF_CORE_FIELD_RSHIFT_U64)
// define ``v`` as signed or unsigned integer of size ``sz``
v = *({s|u}<sz> *)((void *)s + off)
v <<= l
v >>= r
```

- The `BPF_CORE_TYPE_MATCHES` queries matching relation, defined as follows:
  - for integers: types match if size and signedness match;
  - for arrays & pointers: target types are recursively matched;
  - for structs & unions:
    - local members need to exist in target with the same name;
    - for each member we recursively check match unless it is already behind a pointer, in which case we only check matching names and compatible kind;
  - for enums:
    - local variants have to have a match in target by symbolic name (but not numeric value);
    - size has to match (but enum may match enum64 and vice versa);
  - for function pointers:
    - number and position of arguments in local type has to match target;
    - for each argument and the return value we recursively check match.

19.3.2 CO-RE Relocation Record

Relocation record is encoded as the following structure:

```
struct bpf_core_relo {
       __u32 insn_off;
       __u32 type_id;
       __u32 access_str_off;
       enum bpf_core_relo_kind kind;
};
```

- `insn_off` - instruction offset (in bytes) within a code section associated with this relocation;
- `type_id` - BTF type ID of the "root" (containing) entity of a relocatable type or field;
- `access_str_off` - offset into corresponding .BTF string section. String interpretation depends on specific relocation kind:
  - for field-based relocations, string encodes an accessed field using a sequence of field and array indices, separated by colon (:). It's conceptually very close to LLVM's getelementptr instruction's arguments for identifying offset to a field. For example, consider the following C code:

    ```
    struct sample {
        int a;
        int b;
        struct { int c[10]; };
    } __attribute__((preserve_access_index));
    struct sample *s;
    ```

    * Access to `s[0].a` would be encoded as `0:0`:
      · `0`: first element of `s` (as if `s` is an array);
      · `0`: index of field `a` in `struct sample`.
    * Access to `s->a` would be encoded as `0:0` as well.
    * Access to `s->b` would be encoded as `0:1`:
      · `0`: first element of `s`;
      · `1`: index of field `b` in `struct sample`.
    * Access to `s[1].c[5]` would be encoded as `1:2:0:5`:
      · `1`: second element of `s`;
      · `2`: index of anonymous structure field in `struct sample`;
      · `0`: index of field `c` in anonymous structure;
      · `5`: access to array element #5.
  - for type-based relocations, string is expected to be just "0";
  - **for enum value-based relocations, string contains an index of enum** value within its enum type;
- `kind` - one of `enum bpf_core_relo_kind`.

19.3.3 CO-RE Relocation Examples

For the following C code:

```
struct foo {
  int a;
  int b;
  unsigned c:15;
} __attribute__((preserve_access_index));

enum bar { U, V };
```

With the following BTF definitions:

```
...
[2] STRUCT 'foo' size=8 vlen=2
       'a' type_id=3 bits_offset=0
       'b' type_id=3 bits_offset=32
       'c' type_id=4 bits_offset=64 bitfield_size=15
[3] INT 'int' size=4 bits_offset=0 nr_bits=32 encoding=SIGNED
[4] INT 'unsigned int' size=4 bits_offset=0 nr_bits=32 encoding=(none)
...
[16] ENUM 'bar' encoding=UNSIGNED size=4 vlen=2
       'U' val=0
       'V' val=1
```

Field offset relocations are generated automatically when `__attribute__((preserve_access_index))` is used, for example:

```
void alpha(struct foo *s, volatile unsigned long *g) {
  *g = s->a;
  s->a = 1;
}

00 <alpha>:
  0:  r3 = *(s32 *)(r1 + 0x0)
         00:  CO-RE <byte_off> [2] struct foo::a (0:0)
  1:  *(u64 *)(r2 + 0x0) = r3
  2:  *(u32 *)(r1 + 0x0) = 0x1
         10:  CO-RE <byte_off> [2] struct foo::a (0:0)
  3:  exit
```

All relocation kinds could be requested via built-in functions. E.g. field-based relocations:

```
void bravo(struct foo *s, volatile unsigned long *g) {
  *g = __builtin_preserve_field_info(s->b, 0 /* field byte offset */);
  *g = __builtin_preserve_field_info(s->b, 1 /* field byte size */);
  *g = __builtin_preserve_field_info(s->b, 2 /* field existence */);
  *g = __builtin_preserve_field_info(s->b, 3 /* field signedness */);
  *g = __builtin_preserve_field_info(s->c, 4 /* bitfield left shift */);
  *g = __builtin_preserve_field_info(s->c, 5 /* bitfield right shift */);
}
```

```
20 <bravo>:
   4:       r1 = 0x4
            20:  CO-RE <byte_off> [2] struct foo::b (0:1)
   5:       *(u64 *)(r2 + 0x0) = r1
   6:       r1 = 0x4
            30:  CO-RE <byte_sz> [2] struct foo::b (0:1)
   7:       *(u64 *)(r2 + 0x0) = r1
   8:       r1 = 0x1
            40:  CO-RE <field_exists> [2] struct foo::b (0:1)
   9:       *(u64 *)(r2 + 0x0) = r1
  10:       r1 = 0x1
            50:  CO-RE <signed> [2] struct foo::b (0:1)
  11:       *(u64 *)(r2 + 0x0) = r1
  12:       r1 = 0x31
            60:  CO-RE <lshift_u64> [2] struct foo::c (0:2)
  13:       *(u64 *)(r2 + 0x0) = r1
  14:       r1 = 0x31
            70:  CO-RE <rshift_u64> [2] struct foo::c (0:2)
  15:       *(u64 *)(r2 + 0x0) = r1
  16:       exit
```

Type-based relocations:

```
void charlie(struct foo *s, volatile unsigned long *g) {
  *g = __builtin_preserve_type_info(*s, 0 /* type existence */);
  *g = __builtin_preserve_type_info(*s, 1 /* type size */);
  *g = __builtin_preserve_type_info(*s, 2 /* type matches */);
  *g = __builtin_btf_type_id(*s, 0 /* type id in this object file */);
  *g = __builtin_btf_type_id(*s, 1 /* type id in target kernel */);
}

88 <charlie>:
  17:       r1 = 0x1
            88:  CO-RE <type_exists> [2] struct foo
  18:       *(u64 *)(r2 + 0x0) = r1
  19:       r1 = 0xc
            98:  CO-RE <type_size> [2] struct foo
  20:       *(u64 *)(r2 + 0x0) = r1
  21:       r1 = 0x1
            a8:  CO-RE <type_matches> [2] struct foo
  22:       *(u64 *)(r2 + 0x0) = r1
  23:       r1 = 0x2 ll
            b8:  CO-RE <local_type_id> [2] struct foo
  25:       *(u64 *)(r2 + 0x0) = r1
  26:       r1 = 0x2 ll
            d0:  CO-RE <target_type_id> [2] struct foo
  28:       *(u64 *)(r2 + 0x0) = r1
  29:       exit
```

Enum-based relocations:

```
void delta(struct foo *s, volatile unsigned long *g) {
  *g = __builtin_preserve_enum_value(*(enum bar *)U, 0 /* enum literal␣
↪existence */);
  *g = __builtin_preserve_enum_value(*(enum bar *)V, 1 /* enum literal value */
↪);
}

f0 <delta>:
  30:     r1 = 0x1 ll
          f0:  CO-RE <enumval_exists> [16] enum bar::U = 0
  32:     *(u64 *)(r2 + 0x0) = r1
  33:     r1 = 0x1 ll
          108:  CO-RE <enumval_value> [16] enum bar::V = 1
  35:     *(u64 *)(r2 + 0x0) = r1
  36:     exit
```

19.4 BPF Graph Data Structures

This document describes implementation details of new-style "graph" data structures (linked_list, rbtree), with particular focus on the verifier's implementation of semantics specific to those data structures.

Although no specific verifier code is referred to in this document, the document assumes that the reader has general knowledge of BPF verifier internals, BPF maps, and BPF program writing.

Note that the intent of this document is to describe the current state of these graph data structures. **No guarantees** of stability for either semantics or APIs are made or implied here.

19.4.1 Introduction

The BPF map API has historically been the main way to expose data structures of various types for use within BPF programs. Some data structures fit naturally with the map API (HASH, ARRAY), others less so. Consequently, programs interacting with the latter group of data structures can be hard to parse for kernel programmers without previous BPF experience.

Luckily, some restrictions which necessitated the use of BPF map semantics are no longer relevant. With the introduction of kfuncs, kptrs, and the any-context BPF allocator, it is now possible to implement BPF data structures whose API and semantics more closely match those exposed to the rest of the kernel.

Two such data structures - linked_list and rbtree - have many verification details in common.

Because both have “root”s (“head” for linked_list) and “node”s, the verifier code and this document refer to common functionality as “graph_api”, “graph_root”, “graph_node”, etc.

Unless otherwise stated, examples and semantics below apply to both graph data structures.

19.4.2 Unstable API

Data structures implemented using the BPF map API have historically used BPF helper functions - either standard map API helpers like `bpf_map_update_elem` or map-specific helpers. The new-style graph data structures instead use kfuncs to define their manipulation helpers. Because there are no stability guarantees for kfuncs, the API and semantics for these data structures can be evolved in a way that breaks backwards compatibility if necessary.

Root and node types for the new data structures are opaquely defined in the `uapi/linux/bpf.h` header.

19.4.3 Locking

The new-style data structures are intrusive and are defined similarly to their vanilla kernel counterparts:

```
struct node_data {
  long key;
  long data;
  struct bpf_rb_node node;
};

struct bpf_spin_lock glock;
struct bpf_rb_root groot __contains(node_data, node);
```

The “root” type for both linked_list and rbtree expects to be in a map_value which also contains a `bpf_spin_lock` - in the above example both global variables are placed in a single-value arraymap. The verifier considers this spin_lock to be associated with the `bpf_rb_root` by virtue of both being in the same map_value and will enforce that the correct lock is held when verifying BPF programs that manipulate the tree. Since this lock checking happens at verification time, there is no runtime penalty.

19.4.4 Non-owning references

Motivation

Consider the following BPF code:

```
struct node_data *n = bpf_obj_new(typeof(*n)); /* ACQUIRED */

bpf_spin_lock(&lock);

bpf_rbtree_add(&tree, n); /* PASSED */

bpf_spin_unlock(&lock);
```

From the verifier's perspective, the pointer `n` returned from `bpf_obj_new` has type `PTR_TO_BTF_ID | MEM_ALLOC`, with a `btf_id` of `struct node_data` and a nonzero `ref_obj_id`. Because it holds `n`, the program has ownership of the pointee's (object pointed to by `n`) lifetime. The BPF program must pass off ownership before exiting - either via `bpf_obj_drop`, which `free`'s the object, or by adding it to `tree` with `bpf_rbtree_add`.

(`ACQUIRED` and `PASSED` comments in the example denote statements where "ownership is acquired" and "ownership is passed", respectively)

What should the verifier do with `n` after ownership is passed off? If the object was `free`'d with `bpf_obj_drop` the answer is obvious: the verifier should reject programs which attempt to access `n` after `bpf_obj_drop` as the object is no longer valid. The underlying memory may have been reused for some other allocation, unmapped, etc.

When ownership is passed to `tree` via `bpf_rbtree_add` the answer is less obvious. The verifier could enforce the same semantics as for `bpf_obj_drop`, but that would result in programs with useful, common coding patterns being rejected, e.g.:

```
int x;
struct node_data *n = bpf_obj_new(typeof(*n)); /* ACQUIRED */

bpf_spin_lock(&lock);

bpf_rbtree_add(&tree, n); /* PASSED */
x = n->data;
n->data = 42;

bpf_spin_unlock(&lock);
```

Both the read from and write to `n->data` would be rejected. The verifier can do better, though, by taking advantage of two details:

- Graph data structure APIs can only be used when the `bpf_spin_lock` associated with the graph root is held
- Both graph data structures have pointer stability
  - Because graph nodes are allocated with `bpf_obj_new` and adding / removing from the root involves fiddling with the `bpf_{list,rb}_node` field of the node struct, a graph node will remain at the same address after either operation.

Because the associated `bpf_spin_lock` must be held by any program adding or removing, if we're in the critical section bounded by that lock, we know that no other program can add or remove until the end of the critical section. This combined with pointer stability means that, until the critical section ends, we can safely access the graph node through `n` even after it was used to pass ownership.

The verifier considers such a reference a *non-owning reference*. The ref returned by `bpf_obj_new` is accordingly considered an *owning reference*. Both terms currently only have meaning in the context of graph nodes and API.

Details

Let's enumerate the properties of both types of references.

owning reference

- This reference controls the lifetime of the pointee
- Ownership of pointee must be ‘released’ by passing it to some graph API kfunc, or via `bpf_obj_drop`, which `free`’s the pointee
  - If not released before program ends, verifier considers program invalid
- Access to the pointee’s memory will not page fault

non-owning reference

- This reference does not own the pointee
  - It cannot be used to add the graph node to a graph root, nor `free`’d via `bpf_obj_drop`
- No explicit control of lifetime, but can infer valid lifetime based on non-owning ref existence (see explanation below)
- Access to the pointee’s memory will not page fault

From verifier’s perspective non-owning references can only exist between spin_lock and spin_unlock. Why? After spin_unlock another program can do arbitrary operations on the data structure like removing and `free`-ing via bpf_obj_drop. A non-owning ref to some chunk of memory that was remove’d, `free`’d, and reused via bpf_obj_new would point to an entirely different thing. Or the memory could go away.

To prevent this logic violation all non-owning references are invalidated by the verifier after a critical section ends. This is necessary to ensure the “will not page fault” property of non-owning references. So if the verifier hasn’t invalidated a non-owning ref, accessing it will not page fault.

Currently `bpf_obj_drop` is not allowed in the critical section, so if there’s a valid non-owning ref, we must be in a critical section, and can conclude that the ref’s memory hasn’t been dropped-and- `free`’d or dropped-and-reused.

Any reference to a node that is in an rbtree _must_ be non-owning, since the tree has control of the pointee’s lifetime. Similarly, any ref to a node that isn’t in rbtree _must_ be owning. This results in a nice property: graph API add / remove implementations don’t need to check if a node has already been added (or already removed), as the ownership model allows the verifier to prevent such a state from being valid by simply checking types.

However, pointer aliasing poses an issue for the above “nice property”. Consider the following example:

```
struct node_data *n, *m, *o, *p;
n = bpf_obj_new(typeof(*n));     /* 1 */

bpf_spin_lock(&lock);

bpf_rbtree_add(&tree, n);        /* 2 */
m = bpf_rbtree_first(&tree);     /* 3 */

o = bpf_rbtree_remove(&tree, n); /* 4 */
p = bpf_rbtree_remove(&tree, m); /* 5 */

bpf_spin_unlock(&lock);
```

```
bpf_obj_drop(o);
bpf_obj_drop(p); /* 6 */
```

Assume the tree is empty before this program runs. If we track verifier state changes here using numbers in above comments:

1) n is an owning reference
2) n is a non-owning reference, it's been added to the tree
3) n and m are non-owning references, they both point to the same node
4) o is an owning reference, n and m non-owning, all point to same node
5) o and p are owning, n and m non-owning, all point to the same node
6) a double-free has occurred, since o and p point to same node and o was `free`'d in previous statement

States 4 and 5 violate our "nice property", as there are non-owning refs to a node which is not in an rbtree. Statement 5 will try to remove a node which has already been removed as a result of this violation. State 6 is a dangerous double-free.

At a minimum we should prevent state 6 from being possible. If we can't also prevent state 5 then we must abandon our "nice property" and check whether a node has already been removed at runtime.

We prevent both by generalizing the "invalidate non-owning references" behavior of `bpf_spin_unlock` and doing similar invalidation after `bpf_rbtree_remove`. The logic here being that any graph API kfunc which:

- takes an arbitrary node argument
- removes it from the data structure
- returns an owning reference to the removed node

May result in a state where some other non-owning reference points to the same node. So `remove`-type kfuncs must be considered a non-owning reference invalidation point as well.

CHAPTER

TWENTY

REDIRECT

20.1 XDP_REDIRECT

20.1.1 Supported maps

XDP_REDIRECT works with the following map types:

- `BPF_MAP_TYPE_DEVMAP`
- `BPF_MAP_TYPE_DEVMAP_HASH`
- `BPF_MAP_TYPE_CPUMAP`
- `BPF_MAP_TYPE_XSKMAP`

For more information on these maps, please see the specific map documentation.

20.1.2 Process

XDP_REDIRECT works by a three-step process, implemented in the functions below:

1. The bpf_redirect() and bpf_redirect_map() helpers will lookup the target of the redirect and store it (along with some other metadata) in a per-CPU struct bpf_redirect_info.
2. When the program returns the XDP_REDIRECT return code, the driver will call xdp_do_redirect() which will use the information in struct bpf_redirect_info to actually enqueue the frame into a map type-specific bulk queue structure.
3. Before exiting its NAPI poll loop, the driver will call xdp_do_flush(), which will flush all the different bulk queues, thus completing the redirect. Note that xdp_do_flush() must be called before napi_complete_done() in the driver, as the XDP_REDIRECT logic relies on being inside a single NAPI instance through to the xdp_do_flush() call for RCU protection of all in-kernel data structures.

Note: Not all drivers support transmitting frames after a redirect, and for those that do, not all of them support non-linear frames. Non-linear xdp bufs/frames are bufs/frames that contain more than one fragment.

20.1.3 Debugging packet drops

Silent packet drops for XDP_REDIRECT can be debugged using:

- bpf_trace
- perf_record

bpf_trace

The following bpftrace command can be used to capture and count all XDP tracepoints:

```
sudo bpftrace -e 'tracepoint:xdp:* { @cnt[probe] = count(); }'
Attaching 12 probes...
^C

@cnt[tracepoint:xdp:mem_connect]: 18
@cnt[tracepoint:xdp:mem_disconnect]: 18
@cnt[tracepoint:xdp:xdp_exception]: 19605
@cnt[tracepoint:xdp:xdp_devmap_xmit]: 1393604
@cnt[tracepoint:xdp:xdp_redirect]: 22292200
```

Note: The various xdp tracepoints can be found in `source/include/trace/events/xdp.h`

The following bpftrace command can be used to extract the `ERRNO` being returned as part of the err parameter:

```
sudo bpftrace -e \
'tracepoint:xdp:xdp_redirect*_err {@redir_errno[-args->err] = count();}
tracepoint:xdp:xdp_devmap_xmit {@devmap_errno[-args->err] = count();}'
```

perf record

The perf tool also supports recording tracepoints:

```
perf record -a -e xdp:xdp_redirect_err \
    -e xdp:xdp_redirect_map_err \
    -e xdp:xdp_exception \
    -e xdp:xdp_devmap_xmit
```

References

- https://github.com/xdp-project/xdp-tutorial/tree/master/tracing02-xdp-monitor

INDEX