Linux Bpf Documentation

The kernel development community

Jan 15, 2023

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 INSTRUCTION SET

1.1 Registers and calling convention

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

The eBPF calling convention is defined as:

- R0: return value from function calls, and exit value for eBPF 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 eBPF programs needs to spill/fill them if necessary across calls.

1.2 Instruction encoding

eBPF 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 value (imm64) after the basic instruction for a total of 128 bits.

The basic instruction encoding looks as follows:

32 bits (MSB)	16 bits	4 bits	4 bits	8 bits (LSB)
immediate	offset	source register	destination register	opcode

Note that most instructions do not use all of the fields. Unused fields 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
BPF_LD	0x00	non-standard load operations
BPF_LDX	0x01	load into register operations
BPF_ST	0x02	store from immediate operations
BPF_STX	0x03	store from register operations
BPF_ALU	0x04	32-bit arithmetic operations
BPF_JMP	0x05	64-bit jump operations
BPF_JMP32	0x06	32-bit jump operations
BPF_ALU64	0x07	64-bit arithmetic operations

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)
operation code	source	instruction class

The 4th bit encodes the source operand:

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

The four MSB bits store the operation code.

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:

code	value	description
BPF_ADD	0x00	dst += src
BPF_SUB	0x10	dst -= src
BPF_MUL	0x20	dst *= src
BPF_DIV	0x30	dst /= src
BPF_OR	0x40	dst |= src
BPF_AND	0x50	dst &= src
BPF_LSH	0x60	dst <<= src
BPF_RSH	0x70	dst >>= src
BPF_NEG	0x80	dst = ~src
BPF_MOD	0x90	dst %= src
BPF_XOR	0xa0	dst ^= src
BPF_MOV	0xb0	dst = src
BPF_ARSH	0xc0	sign extending shift right
BPF_END	0xd0	byte swap operations (see separate section below)

BPF_ADD | BPF_X | BPF_ALU means:

```
dst_reg = (u32) dst_reg + (u32) src_reg;
```

BPF_ADD | BPF_X | BPF_ALU64 means:

```
dst_reg = dst_reg + src_reg
```

BPF_XOR | BPF_K | BPF_ALU means:

```
src_reg = (u32) src_reg ^ (u32) imm32
```

BPF_XOR | BPF_K | BPF_ALU64 means:

```
src_reg = src_reg ^ imm32
```

1.3.2 Byte swap instructions

The byte swap instructions use an instruction class of `BFP_ALU` 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.

The 1-bit source operand field in the opcode is used to to select what byte order the operation convert from or to:

source	value	description
BPF_TO_LE	0x00	convert between host byte order and little endian
BPF_TO_BE	0x08	convert between host byte order and big endian

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 means:

```
dst_reg = htole16(dst_reg)
```

`BPF_ALU | BPF_TO_BE | BPF_END` with imm = 64 means:

```
dst_reg = htobe64(dst_reg)
```

`BPF_FROM_LE` and `BPF_FROM_BE` exist as aliases for `BPF_TO_LE` and `BPF_TO_BE` respectively.

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	description	notes
BPF_JA	0x00	PC += off	BPF_JMP only
BPF_JEQ	0x10	PC += off if dst == src	
BPF_JGT	0x20	PC += off if dst > src	unsigned
BPF_JGE	0x30	PC += off if dst >= src	unsigned
BPF_JSET	0x40	PC += off if dst & src	
BPF_JNE	0x50	PC += off if dst != src	
BPF_JSGT	0x60	PC += off if dst > src	signed
BPF_JSGE	0x70	PC += off if dst >= src	signed
BPF_CALL	0x80	function call	
BPF_EXIT	0x90	function / program return	BPF_JMP only
BPF_JLT	0xa0	PC += off if dst < src	unsigned
BPF_JLE	0xb0	PC += off if dst <= src	unsigned
BPF_JSLT	0xc0	PC += off if dst < src	signed
BPF_JSLE	0xd0	PC += off if dst <= src	signed

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

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 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)

The mode modifier is one of:

mode modifier	value	description
BPF_IMM	0x00	64-bit immediate instructions
BPF_ABS	0x20	legacy BPF packet access (absolute)
BPF_IND	0x40	legacy BPF packet access (indirect)
BPF_MEM	0x60	regular load and store operations
BPF_ATOMIC	0xc0	atomic operations

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_reg + off) = src_reg
```

`BPF_MEM | <size> | BPF_ST` means:

```
*(size *) (dst_reg + off) = imm32
```

`BPF_MEM | <size> | BPF_LDX` means:

```
dst_reg = *(size *) (src_reg + off)
```

Where size is one of: `BPF_B`, `BPF_H`, `BPF_W`, or `BPF_DW`.

1.4.2 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 eBPF programs or means outside of this specification.

All atomic operations supported by eBPF 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_reg + off16) += src_reg
```

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

```
*(u64 *)(dst_reg + off16) += src_reg
```

`BPF_XADD` is a deprecated name for `BPF_ATOMIC | BPF_ADD`.

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_reg` with the value that was in memory before it was modified.

The `BPF_XCHG` operation atomically exchanges `src_reg` with the value addressed by `dst_reg + off`.

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

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`.

1.4.3 64-bit immediate instructions

Instructions with the `BPF_IMM` mode modifier use the wide instruction encoding for an extra imm64 value.

There is currently only one such instruction.

`BPF_LD | BPF_DW | BPF_IMM` means:

```
dst_reg = imm64
```

1.4.4 Legacy BPF Packet access instructions

eBPF has special 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 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 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 after a call to `BPF_ABS | BPF_LD` or `BPF_IND` | BPF_LD instructions.

These instructions have an implicit program exit condition as well. When an eBPF program is trying to access the data beyond the packet boundary, the program execution will be aborted.

`BPF_ABS | BPF_W | BPF_LD` means:

```
R0 = ntohl(*(u32 *) (((struct sk_buff *) R6)->data + imm32))
```

`BPF_IND | BPF_W | BPF_LD` means:

```
R0 = ntohl(*(u32 *) (((struct sk_buff *) R6)->data + src_reg + imm32))
```

CHAPTER

TWO

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

2.1 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 *eBPF 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 addes (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.

2.2 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.

2.3 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().

2.4 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

THREE

LIBBPF

3.1 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_`, `xsk_`, `btf_dump_`, `ring_buffer_`, `perf_buffer_`.

3.1.1 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.

3.1.2 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`.

3.1.3 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`.

3.1.4 AF_XDP functions

AF_XDP functions should have an `xsk_` prefix, e.g. `xsk_umem__get_data` or `xsk_umem__create`. The interface consists of both low-level ring access functions and high-level configuration functions. These can be mixed and matched. Note that these functions are not reentrant for performance reasons.

3.1.5 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.

3.1.6 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.

3.1.7 ABI versionning

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].

3.1.8 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.

3.2 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 formated.

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.

3.2.1 License

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

3.2.2 Links

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

3.3 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
```

This is documentation for libbpf, a userspace library for loading and interacting with bpf programs.

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 very well might be that this was already addressed or answered before.

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  */
#define BTF_KIND_FWD            7       /* Forward      */
#define BTF_KIND_TYPEDEF        8       /* Typedef      */
#define BTF_KIND_VOLATILE       9       /* Volatile     */
#define BTF_KIND_CONST          10      /* Const        */
#define BTF_KIND_RESTRICT       11      /* Restrict     */
#define BTF_KIND_FUNC           12      /* Function     */
#define BTF_KIND_FUNC_PROTO     13      /* Function Proto       */
#define BTF_KIND_VAR            14      /* Variable     */
#define BTF_KIND_DATASEC        15      /* Section      */
#define BTF_KIND_FLOAT          16      /* Floating point       */
#define BTF_KIND_DECL_TAG       17      /* Decl Tag     */
#define BTF_KIND_TYPE_TAG       18      /* Type Tag     */
```

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

Each type contains the following common data:

```
struct btf_type {
    __u32 name_off;
    /* "info" bits arrangement
     * bits  0-15: vlen (e.g. # of struct's members)
     * bits 16-23: unused
     * bits 24-28: kind (e.g. int, ptr, array...etc)
     * bits 29-30: unused
     * bit     31: kind_flag, currently used by
     *             struct, union and fwd
     */
    __u32 info;
    /* "size" is used by INT, ENUM, STRUCT and UNION.
     * "size" tells the size of the type it is describing.
     *
     * "type" is used by PTR, TYPEDEF, VOLATILE, CONST, RESTRICT,
     * FUNC, FUNC_PROTO, DECL_TAG and TYPE_TAG.
     * "type" is a type_id referring to another type.
     */
    union {
            __u32 size;
            __u32 type;
    };
};
```

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

2.2.1 BTF_KIND_INT

`struct btf_type` encoding requirement:

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

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

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

The `BTF_INT_ENCODING` has the following attributes:

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

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

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

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

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

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

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

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

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

2.2.2 BTF_KIND_PTR

struct btf_type encoding requirement:

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

No additional type data follow btf_type.

2.2.3 BTF_KIND_ARRAY

struct btf_type encoding requirement:

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

btf_type is followed by one struct btf_array:

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

The struct btf_array encoding:

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

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

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

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

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

2.2.4 BTF_KIND_STRUCT

2.2.5 BTF_KIND_UNION

struct btf_type encoding requirement:

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

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

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

struct btf_member encoding:

- `name_off`: offset to a valid C identifier
- `type`: the member type
- `offset`: <see below>

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

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

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

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

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

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

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

2.2.6 BTF_KIND_ENUM

`struct btf_type` encoding requirement:

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

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

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

The `btf_enum` encoding:

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

2.2.7 BTF_KIND_FWD

`struct btf_type` encoding requirement:

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

No additional type data follow `btf_type`.

2.2.8 BTF_KIND_TYPEDEF

`struct btf_type` encoding requirement:

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

No additional type data follow `btf_type`.

2.2.9 BTF_KIND_VOLATILE

`struct btf_type` encoding requirement:

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

No additional type data follow `btf_type`.

2.2.10 BTF_KIND_CONST

`struct btf_type` encoding requirement:

- `name_off`: 0
- `info.kind_flag`: 0
- `info.kind`: BTF_KIND_CONST
- `info.vlen`: 0
- `type`: the type with `const` qualifier

No additional type data follow `btf_type`.

2.2.11 BTF_KIND_RESTRICT

`struct btf_type` encoding requirement:

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

No additional type data follow `btf_type`.

2.2.12 BTF_KIND_FUNC

`struct btf_type` encoding requirement:

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

No additional type data follow `btf_type`.

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

2.2.13 BTF_KIND_FUNC_PROTO

`struct btf_type` encoding requirement:

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

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

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

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

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

2.2.14 BTF_KIND_VAR

`struct btf_type` encoding requirement:

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

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

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

`struct btf_var` encoding:

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

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

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

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

2.2.15 BTF_KIND_DATASEC

`struct btf_type` encoding requirement:

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

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

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

struct btf_var_secinfo encoding:

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

2.2.16 BTF_KIND_FLOAT

struct btf_type encoding requirement:

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

No additional type data follow btf_type.

2.2.17 BTF_KIND_DECL_TAG

struct btf_type encoding requirement:

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

btf_type is followed by struct btf_decl_tag.:

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

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

2.2.17 BTF_KIND_TYPE_TAG

`struct btf_type` encoding requirement:

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

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

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

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

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 and line_info which needs loader manipulation before loading into the kernel.

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

The current header of .BTF.ext section:

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

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

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

The func_info is organized as below.:

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

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

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

Here, num_info must be greater than 0.

The line_info is organized as below.:

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

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

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

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

Kernel bpf selftest *test_btf.c* provides 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 there is no BPF_SDIV for signed divide operation?

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

Q: Why BPF has implicit prologue and epilogue?

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

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

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

Q: BPF 32-bit subregister requirements

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

A: NO.

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

Starting with version 7, LLVM is able to generate instructions that operate on 32-bit subregisters, provided the option -mattr=+alu32 is passed for compiling a program. Furthermore, the verifier can now mark the instructions for which 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.

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: How much stack space a BPF program uses?

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

Q: Can BPF be offloaded to HW?

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

Q: Does classic BPF interpreter still exist?

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

Q: Can BPF call arbitrary kernel functions?

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

Q: Can BPF overwrite arbitrary kernel memory?

A: NO.

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

Q: Can BPF overwrite arbitrary user memory?

A: Sort-of.

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

Q: New functionality via kernel modules?

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

A: NO.

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.

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/. 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: 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 netdev-FAQ

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

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

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

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

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

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

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

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

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

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

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

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

```
git format-patch --subject-prefix='PATCH 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 netdev-FAQ for additional information e.g. on how often they are merged to mainline.

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

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

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

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

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

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

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

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

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

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

Q: Verifier changes and test cases

Q: I made a BPF verifier change, do I need to add test cases for BPF kernel 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 netdev-FAQ.

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

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

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

> https://www.kernel.org/

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

What should I do?

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

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

Q: Queue stable patches

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

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

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

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

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

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

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

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

https://github.com/acmel/dwarves

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

PROGRAM TYPES

8.1 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`.

8.1.1 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.

8.1.2 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).

8.1.3 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.

8.1.4 Large optval

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

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

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

8.1.5 Example

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

8.2 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.

8.2.1 1. Attach type

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

8.2.2 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.

8.2.3 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`.

8.2.4 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.

8.2.5 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.

8.2.6 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.

8.3 BPF_PROG_TYPE_FLOW_DISSECTOR

8.3.1 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).

8.3.2 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.

8.3.3 __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.

8.3.4 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.

8.3.5 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).

8.3.6 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.

8.4 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.

8.4.1 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 `include/linux/lsm_hooks.h`.

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

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

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

Note: The order of the fields is irrelevant.

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

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

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

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

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

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

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

        int is_heap;

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

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

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

8.4.2 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.

8.4.3 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.

8.4.4 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

8.5 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.

8.5.1 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.

8.5.2 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.

8.5.3 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.

8.5.4 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.

8.5.5 Example

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

CHAPTER

NINE

EBPF MAPS

‘maps’ is a generic storage of different types for sharing data between kernel and userspace.

The maps are accessed from user space via BPF syscall, which has commands:

- create a map with given type and attributes `map_fd = bpf(BPF_MAP_CREATE, union bpf_attr *attr, u32 size)` using attr->map_type, attr->key_size, attr->value_size, attr->max_entries returns process-local file descriptor or negative error
- lookup key in a given map `err = bpf(BPF_MAP_LOOKUP_ELEM, union bpf_attr *attr, u32 size)` using attr->map_fd, attr->key, attr->value returns zero and stores found elem into value or negative error
- create or update key/value pair in a given map `err = bpf(BPF_MAP_UPDATE_ELEM, union bpf_attr *attr, u32 size)` using attr->map_fd, attr->key, attr->value returns zero or negative error
- find and delete element by key in a given map `err = bpf(BPF_MAP_DELETE_ELEM, union bpf_attr *attr, u32 size)` using attr->map_fd, attr->key
- to delete map: close(fd) Exiting process will delete maps automatically

userspace programs use this syscall to create/access maps that eBPF programs are concurrently updating.

maps can have different types: hash, array, bloom filter, radix-tree, etc.

The map is defined by:

- type
- max number of elements
- key size in bytes
- value size in bytes

9.1 Map Types

9.1.1 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 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 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.

CHAPTER TEN

RUNNING BPF PROGRAMS FROM USERSPACE

This document describes the BPF_PROG_RUN facility for running BPF programs from userspace.

10.1 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.

10.2 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

ELEVEN

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.

11.1 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

TWELVE

BPF LICENSING

12.1 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.

12.2 Implementations of eBPF

Using the eBPF instruction set requires implementing code in both kernel space and user space.

12.2.1 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.

12.2.2 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).

12.2.3 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.

12.2.4 In other operating systems

Other kernels or user space implementations of eBPF instruction set and runtime can have proprietary licenses.

12.3 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".

12.4 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

THIRTEEN

TESTING AND DEBUGGING BPF

13.1 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.

13.1.1 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,
        },
}
```

13.2 Testing BPF on s390

13.2.1 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.

13.2.2 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
```

13.2.3 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
```

13.2.4 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
```

13.2.5 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
```

13.2.6 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
```

13.2.7 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

CHAPTER FOURTEEN

OTHER

14.1 BPF ring buffer

This document describes BPF ring buffer design, API, and implementation details.

14.1.1 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.

14.1.2 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 possition 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.

14.1.3 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 interruped 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.

14.2 BPF LLVM Relocations

This document describes LLVM BPF backend relocation types.

14.2.1 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 relations 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, for global variable `g2` which has a symbol value 4, the offset from the start of `.data` section.

The third and fourth relocations refers to static variables `l1` and `l2`. From `.rel.text` section above, it is not clear which symbols they really refers to as they both refers to symbol table entry 4, symbol `sec`, which has `STT_SECTION` type and represents a section. So for 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.

14.2.2 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`.

14.2.3 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
```

INDEX