[译] Linux Socket Filtering (LSF, aka BPF)（KernelDoc，2021）

Published at 2021-08-27 | Last Update 2022-05-01

译者序

本文翻译自 2021 年 Linux `5.10` 内核文档： Linux Socket Filtering aka Berkeley Packet Filter (BPF)， 文档源码见 Documentation/networking/filter.rst。

Linux Socket Filtering (LSF) 是最初将 BSD 系统上的**数据包过滤技术** BPF（伯克利包过滤器）移植到 Linux 系统时使用的名称，但后来大家还是更多称呼其为 BPF（ aka ： as known as）。本文介绍了 Linux BPF 的一些 **底层设计和实现**（包括 cBPF 和 eBPF），可作为 Cilium：BPF 和 XDP 参考指南（2021） 的很好补充，这两篇可能是目前除了内核源码之外，学习 BPF 的最全/最好参考。 本文适合有一定 BPF 经验的开发者阅读，不适合初学者。

由于内核文档更新不是非常及时，文中部分内容已经与 5.10 代码对不上，因此（少量）过时内容在翻译时略去了。另外，为文中的大部分 BPF 汇编 / x86_64 汇编加了注释，并插入了一些 5.10 代码片段或链接，方便更深入理解。

由于译者水平有限，本文不免存在遗漏或错误之处。如有疑问，请查阅原文。

以下是译文。

> SPDX-License-Identifier: GPL-2.0

---

cBPF 相关内容

---

1 cBPF 引言

Linux Socket Filtering (LSF) 从 Berkeley Packet Filter（BPF）**衍生而来**。 虽然 BSD 和 Linux Kernel filtering 有一些重要不同，但**在 Linux 语境中提到 BPF 或 LSF 时**， 我们指的是 Linux 内核中的同一套**过滤机制**。

1.1 LSF (cBPF) 与 BSD BPF

BPF 允许用户空间程序**向任意 socket attach 过滤器（filter）， 对流经 socket 的数据进行控制**（放行或拒绝）。 LSF 完全遵循了 BSD BPF 的过滤器代码结构（filter code structure），因此实现过滤器时， BSD bpf.4 manpage 是很好的参考文档。

但 **Linux BPF 要比 BSD BPF 简单很多**：

- 用户无需关心设备（devices）之类的东西；
- 只需要创建自己的过滤器代码（filter code），通过 `SO_ATTACH_FILTER` 选项将其发送到内核；
- 接下来只要这段代码能通过内核校验，用户就能立即在 socket 上开始过滤数据了。

1.2 `ATTACH / DETACH / LOCK` 操作

- `SO_ATTACH_FILTER` 用于将 filter attach 到 socket。
- `SO_DETACH_FILTER` 用于从 socket 中 detach 过滤器。

  但这种情况可能比较少，因为关闭一个 socket 时，attach 在上面的所有 filters 会被**自动删除**。另一个不太常见的场景是：向一个已经有 filter 的 socket 再 attach 一个 filter： 内核负责将老的移除，替换成新的 —— 只要新的过滤器通过了校验，否则还是老的在工作。

- `SO_LOCK_FILTER` 选项支持将 attach 到 socket 上的 **filter 锁定**。一旦锁定之后，这个过滤器就**不能被删除或修改**了。这样就能保证下列操作之后：

  i. 进程创建 socket
  ii. attach filter
  iii. 锁定 filter
  iv. **drop privileges**

  这个 filter 就会一直运行在该 socket 上，直到后者被关闭。

1.3 LSF/BPF 使用场景

BPF 模块的**最大用户**可能就是 `libpcap` 。例如，对于高层过滤命令 `tcpdump -i em1 port 22` ，

- **libpcap 编译器**能将其编译生成一个 cBPF 程序，然后通过前面介绍的 `SO_ATTACH_FILTER` 就能加载到内核；
- 加 `-ddd` 参数，可以 **dump 这条命令对应的字节码**： `tcpdump -i em1 port 22 -ddd` 。

虽然我们这里讨论的都是 socket，但 **Linux 中 BPF 还可用于很多其他场景**。例如

- netfilter 中的 `xt_bpf`
- 内核 qdisc 层的 `cls_bpf`
- seccomp-bpf (SECure COMPuting)

- 其他很多地方，包括 team driver、PTP。

1.4 cBPF 经典论文

最初的 BPF 论文：

> Steven McCanne and Van Jacobson. 1993. **The BSD packet filter: a new architecture for user-level packet capture**. In Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93). USENIX Association, Berkeley, CA, USA, 2-2. [http://www.tcpdump.org/papers/bpf-usenix93.pdf]

2 cBPF 数据结构

2.1 `struct sock_filter`

要开发 cBPF 应用，用户空间程序需要 include `<linux/filter.h>`，其中定义了下面的结构体：

```
struct sock_filter {	/* Filter block */
	__u16	code;   /* Actual filter code */
	__u8	jt;	/* Jump true */
	__u8	jf;	/* Jump false */
	__u32	k;      /* Generic multiuse field */
};
```

这个结构体包含 `code` `jt` `jf` `k` 四个字段。 `jt` 和 `jf` 是 jump offset， `k` 是一个 `code` 可以使用的通用字段。

2.2 `struct sock_fprog`

要实现 socket filtering，需要通过 `setsockopt(2)` 将一个 `struct sock_fprog` 指针

传递给内核（后面有例子）。 这个结构体的定义：

```
struct sock_fprog {                     /* Required for SO_ATTACH_FILTER. */
    unsigned short           len;  /* Number of filter blocks */
    struct sock_filter __user *filter;
};
```

3 cBPF 示例： `libpcap` 过滤 socket 流量

3.1 `setsockopt()` 将字节码 attach 到 socket

两个结构体 `struct sock_filter` 和 `struct sock_fprog` 在前一节介绍过了：

```
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <linux/if_ether.h>
/* ... */

/* From the example above: tcpdump -i em1 port 22 -dd */
struct sock_filter code[] = {
    { 0x28,  0,  0, 0x0000000c },
    { 0x15,  0,  8, 0x000086dd },
    { 0x30,  0,  0, 0x00000014 },
    { 0x15,  2,  0, 0x00000084 },
    { 0x15,  1,  0, 0x00000006 },
    { 0x15,  0, 17, 0x00000011 },
    { 0x28,  0,  0, 0x00000036 },
    { 0x15, 14,  0, 0x00000016 },
    { 0x28,  0,  0, 0x00000038 },
    { 0x15, 12, 13, 0x00000016 },
    { 0x15,  0, 12, 0x00000800 },
    { 0x30,  0,  0, 0x00000017 },
    { 0x15,  2,  0, 0x00000084 },
    { 0x15,  1,  0, 0x00000006 },
    { 0x15,  0,  8, 0x00000011 },
    { 0x28,  0,  0, 0x00000014 },
```

```
        { 0x45,  6,  0, 0x00001fff },
        { 0xb1,  0,  0, 0x0000000e },
        { 0x48,  0,  0, 0x0000000e },
        { 0x15,  2,  0, 0x00000016 },
        { 0x48,  0,  0, 0x00000010 },
        { 0x15,  0,  1, 0x00000016 },
        { 0x06,  0,  0, 0x0000ffff },
        { 0x06,  0,  0, 0x00000000 },
};

struct sock_fprog bpf = {
        .len = ARRAY_SIZE(code),
        .filter = code,
};

sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
if (sock < 0)
        /* ... bail out ... */

ret = setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf, sizeof(bpf));
if (ret < 0)
        /* ... bail out ... */

/* ... */
close(sock);
```

以上代码将一个 filter attach 到了一个 `PF_PACKET` 类型的 socket，功能是 **放行所有 IPv4/IPv6 22 端口的包，其他包一律丢弃**。

这里只展示了 attach 代码；detach 时， `setsockopt(2)` 除了 `SO_DETACH_FILTER` 不需要其他参数； `SO_LOCK_FILTER` 可用于防止 filter 被 detach，需要带一个整形参数 0 或 1。

注意 socket filters 并不是只能用于 PF_PACKET 类型的 socket，也可以用于其他 socket 家族。

3.2 `setsockopt()` attach/detach/lock 时的参数

总结前面用到的几次系统调用：

- `setsockopt(sockfd, SOL_SOCKET, SO_ATTACH_FILTER, &val, sizeof(val));`
- `setsockopt(sockfd, SOL_SOCKET, SO_DETACH_FILTER, &val, sizeof(val));`
- `setsockopt(sockfd, SOL_SOCKET, SO_LOCK_FILTER, &val, sizeof(val));`

3.3 libpcap 适用和不适用的场景

lipcap **高层语法**封装了上面代码中看到的那些底层操作， 功能已经**覆盖了大部分 socket filtering 的场景**， 因此如果想开发流量过滤应用，开发者应该首选基于 libpcap。

除非遇到以下情况，否则不要纯手工编写过滤器：

1. 开发环境比较特殊，无法使用或链接 libpcap；
2. 使用的 filter 需要用到 libpcap 编译器还没有支持的 Linux extensions；
3. 开发的 filter 比较复杂，libpcap 编译器无法很好地支持该 filter；
4. 需要对特定的 filter 代码做优化，而不想使用 libpcap 编译器生成的代码；

libpcap 不适用的场景举例：

1. **xt_bpf 和 cls_bpf 用户**可能需要更加复杂的 filter 代码， libpcap 无法很好地表达。
2. BPF JIT 开发者可能希望**手写测试用例**，因此也需要直接编写或修改 BPF 代码。

4 cBPF 引擎和指令集

4.1 `bpf_asm`： 最小 BPF 汇编器（assembler）

内核 `tools/bpf/` 目录下有个小工具 `bpf_asm` ，能用它来**编写 low-level filters**， 例如前面提到的一些 libpcap 不适用的场景。

BPF 语法类似汇编，在 bpf_asm 已经中实现了，接下来还会用这种汇编解释其他一些程序 （而不是直接使用难懂的 opcodes，二者的原理是一样的）。这种汇编语法非常接近

Steven McCanne's and Van Jacobson's BPF paper 中的建模。

4.2 cBPF 架构

cBPF 架构由如下几个基本部分组成：

```
=======         ====================================================
Element         Description
=======         ====================================================
A               32 bit wide accumulator (32bit 位宽的累加器)
X               32 bit wide X register  (32bit 位宽 X 寄存器)
M[]             16 x 32 bit wide misc registers aka "scratch memory stor
                (16x32bit 数组，数组索引 0~15，可存放任意内容)
=======         ====================================================
```

BPF 程序经过 bpf_asm 处理之后**变成一个 `struct sock_filter` 类型的数组**（这个结构体前面介绍过），因此数组中的每个元素都是**以如下格式编码**的：

```
op:16, jt:8, jf:8, k:32
```

- `op`： 16bit opcode，其中包括了特定的指令；
- `jt`： jump if true
- `jf`： jump if false
- `k`： 多功能字段，**存放的什么内容，根据 op 类型来解释**。

4.3 bpf_asm 实现的指令集

指令集包括 load、store、branch、alu、return 等指令，bpf_asm 语言中**实现了这些指令**。下面的表格列出了 bpf_asm 中具体包括的指令，对应的 opcode 定义在 `linux/filter.h`：

```
===========     ===================   =====================
指令              寻址模式                解释
```

```
===========      ====================  ====================
ld               1, 2, 3, 4, 12        Load word into A
ldi              4                     Load word into A
ldh              1, 2                  Load half-word into A
ldb              1, 2                  Load byte into A
ldx              3, 4, 5, 12           Load word into X
ldxi             4                     Load word into X
ldxb             5                     Load byte into X

st               3                     Store A into M[]
stx              3                     Store X into M[]

jmp              6                     Jump to label
ja               6                     Jump to label
jeq              7, 8, 9, 10           Jump on A == <x>
jneq             9, 10                 Jump on A != <x>
jne              9, 10                 Jump on A != <x>
jlt              9, 10                 Jump on A <  <x>
jle              9, 10                 Jump on A <= <x>
jgt              7, 8, 9, 10           Jump on A >  <x>
jge              7, 8, 9, 10           Jump on A >= <x>
jset             7, 8, 9, 10           Jump on A &  <x>

add              0, 4                  A + <x>
sub              0, 4                  A - <x>
mul              0, 4                  A * <x>
div              0, 4                  A / <x>
mod              0, 4                  A % <x>
neg                                    !A
and              0, 4                  A & <x>
or               0, 4                  A | <x>
xor              0, 4                  A ^ <x>
lsh              0, 4                  A << <x>
rsh              0, 4                  A >> <x>

tax                                    Copy A into X
txa                                    Copy X into A

ret              4, 11                 Return
===========      ====================  ====================
```

其中第二列是寻找模式，定义见下面。

4.4 12 种指令寻址模式

寻址模式的定义如下：

```
==============  ====================  ====================================
寻址模式          语法                    解释
==============  ====================  ====================================
 0              x/%x                  Register X
 1              [k]                   BHW at byte offset k in the packet
 2              [x + k]               BHW at the offset X + k in the pack
 3              M[k]                  Word at offset k in M[]
 4              #k                    Literal value stored in k
 5              4*([k]&0xf)           Lower nibble * 4 at byte offset k i
 6              L                     Jump label L
 7              #k,Lt,Lf              Jump to Lt if true, otherwise jump
 8              x/%x,Lt,Lf            Jump to Lt if true, otherwise jump
 9              #k,Lt                 Jump to Lt if predicate is true
10              x/%x,Lt               Jump to Lt if predicate is true
11              a/%a                  Accumulator A
12              extension             BPF extension
==============  ====================  ====================================
```

注意最后一种： `BPF extensions` ， 这是 Linux 对 BPF 的扩展，下一节详细介绍。

4.5 Linux BPF extensions（Linux BPF 扩展）

除了常规的一些 load 指令，Linux 内核还有一些 BPF extensions，它们用一个 **负 offset 加上一个特殊的 extension offset 来 “overloading” k 字段**， 然后将这个**结果加载到寄存器 A 中**：

```
==================================  ====================================
Extension                           描述（实际对应的结构体字段或值）
==================================  ====================================
len                                 skb->len
proto                               skb->protocol
```

```
type                                    skb->pkt_type
poff                                    Payload start offset
ifidx                                   skb->dev->ifindex
nla                                     Netlink attribute of type X with of
nlan                                    Nested Netlink attribute of type X
mark                                    skb->mark
queue                                   skb->queue_mapping
hatype                                  skb->dev->type
rxhash                                  skb->hash
cpu                                     raw_smp_processor_id()
vlan_tci                                skb_vlan_tag_get(skb)
vlan_avail                              skb_vlan_tag_present(skb)
vlan_tpid                               skb->vlan_proto
rand                                    prandom_u32()
==================================      ==================================
```

这些扩展也可以加上 # 前缀。

> 以上提到的负 offset 和具体 extension 的 offset，定义见 include/uapi/linux/filter.h：
>
> ```
> /* RATIONALE. Negative offsets are invalid in BPF.
>    We use them to reference ancillary data.
>    Unlike introduction new instructions, it does not break
>    existing compilers/optimizers.
>  */
> #define SKF_AD_OFF    (-0x1000)
> #define SKF_AD_PROTOCOL 0
> #define SKF_AD_PKTTYPE     4
> #define SKF_AD_IFINDEX     8
> #define SKF_AD_NLATTR    12
> #define SKF_AD_NLATTR_NEST    16
> #define SKF_AD_MARK     20
> #define SKF_AD_QUEUE    24
> #define SKF_AD_HATYPE    28
> #define SKF_AD_RXHASH    32
> #define SKF_AD_CPU    36
> #define SKF_AD_ALU_XOR_X    40
> #define SKF_AD_VLAN_TAG    44
> #define SKF_AD_VLAN_TAG_PRESENT 48
> ```

```
#define SKF_AD_PAY_OFFSET    52
#define SKF_AD_RANDOM    56
#define SKF_AD_VLAN_TPID    60
#define SKF_AD_MAX    64

#define SKF_NET_OFF    (-0x100000)
#define SKF_LL_OFF    (-0x200000)

#define BPF_NET_OFF    SKF_NET_OFF
#define BPF_LL_OFF    SKF_LL_OFF
```

在 kernel/bpf/core.c 等地方使用：

```
/* No hurry in this branch
 *
 * Exported for the bpf jit load helper.
 */
void *bpf_internal_load_pointer_neg_helper(const struct sk_buff *skb,
{
    u8 *ptr = NULL;

    if (k >= SKF_NET_OFF)
        ptr = skb_network_header(skb) + k - SKF_NET_OFF;
    else if (k >= SKF_LL_OFF)
        ptr = skb_mac_header(skb) + k - SKF_LL_OFF;

    if (ptr >= skb->head && ptr + size <= skb_tail_pointer(skb))
        return ptr;

    return NULL;
}
```

Cilium：BPF 和 XDP 参考指南（2021） 中对此亦有提及。

译注。

4.6 cBPF 汇编示例若干（附代码解读）

过滤 **ARP 包**：

```
ldh [12]            ; 将 skb 第 12,13 两个字节（h 表示 half word，两个字节，即 sl
jne #0x806, drop    ; 如果寄存器 A 中的值不等于 0x0806（ARP 协议），则跳转到 drop
ret #-1             ; （能执行到这一行，说明是 ARP 包），返回 -1
drop: ret #0        ; 返回 0
```

过滤 **IPv4 TCP 包**：

```
ldh [12]            ; 将 skb 第 12,13 两个字节（h 表示 half word，两个字节，即 sl
jne #0x800, drop    ; 如果寄存器 A 中的值不等于 0x0800（IPv4 协议），则跳转到 drop
ldb [23]            ; 将 skb 第 23 字节（b 表示 byte，一个字节，即 ipv4_hdr->prc
jneq #6, drop       ; 如果寄存器 A 中的值不等于 6（TCP 协议），则跳转到 drop
ret #-1             ; （能执行到这一行，说明是 TCP 包），返回 -1
drop: ret #0        ; 返回 0
```

过滤 **VLAN ID 等于 10 的包**：

```
ld vlan_tci         ; 根据前面介绍的 BPF extensions，这会转换成 skb_vlan_tag_ge
jneq #10, drop      ; 如果寄存器 A 中的值不等于 10，则跳转到 drop
ret #-1             ; （能执行到这一行说明 VLAN ID 等于 10），返回 -1
drop: ret #0        ; 返回 0
```

对 ICMP 包随机采集，采样频率 1/4：

```
ldh [12]            ; 将 skb 第 12,13 两个字节（h 表示 half word，两个字节，即 sl
jne #0x800, drop    ; 如果寄存器 A 中的值不等于 0x0800（IPv4 协议），则跳转到 drop
ldb [23]            ; 将 skb 第 23 字节（b 表示 byte，一个字节，即 ipv4_hdr->prc
jneq #1, drop       ; 如果寄存器 A 中的值不等于 1（ICMP 协议），则跳转到 drop
ld rand             ; 获取一个 u32 类型的随机数，存入寄存器 A
mod #4              ; 将寄存器 A 中的值原地对 4 取模（结果仍然存入 A）
jneq #1, drop       ; 如果 A 中的值（即取模的结果）不等于 1，跳转到 drop
ret #-1             ; （能执行到这里说明对 4 取模等于 1），返回 -1
drop: ret #0        ; 返回 0
```

SECCOMP filter example：

```
ld [4]                  /* offsetof(struct seccomp_data, arch) */
jne #0xc000003e, bad    /* AUDIT_ARCH_X86_64 */
ld [0]                  /* offsetof(struct seccomp_data, nr) */
jeq #15, good           /* __NR_rt_sigreturn */
jeq #231, good          /* __NR_exit_group */
jeq #60, good           /* __NR_exit */
jeq #0, good            /* __NR_read */
jeq #1, good            /* __NR_write */
jeq #5, good            /* __NR_fstat */
jeq #9, good            /* __NR_mmap */
jeq #14, good           /* __NR_rt_sigprocmask */
jeq #13, good           /* __NR_rt_sigaction */
jeq #35, good           /* __NR_nanosleep */
bad: ret #0             /* SECCOMP_RET_KILL_THREAD */
good: ret #0x7fff0000   /* SECCOMP_RET_ALLOW */
```

4.7 用 bps_asm 编译成字节码

以上代码片段都可以放到文件中（下面用 foo 表示），然后用 bpf_asm 来生成 opcodes，后者是可以被 xt_bpf 和 cls_bpf 理解的格式，能直接加载。以上面的 ARP 代码为例：

```
$ ./bpf_asm foo
4,40 0 0 12,21 0 1 2054,6 0 0 4294967295,6 0 0 0,
```

也可以输出成更容易复制粘贴的与 C 类似的格式：

```
$ ./bpf_asm -c foo
{ 0x28,  0,  0, 0x0000000c },
{ 0x15,  0,  1, 0x00000806 },
{ 0x06,  0,  0, 0xffffffff },
{ 0x06,  0,  0, 0000000000 },
```

4.8 调试

xt_bpf 和 cls_bpf 场景中可能会用到非常复杂的 BPF 过滤器，不像上面的代码一眼就能看懂。 因此在将这些复杂程序（过滤器）直接 attach 到真实系统之前，最好先在线下测试一遍。

`bpf_dbg` 就是用于这一目的的小工具，位于内核源码 `tools/bpf/` 中。它可以测试 BPF filters，输入是 pcap 文件，支持单步运行、打印 BPF 虚拟机的寄存器状态等等。

```
# 使用默认 stdin/stdout
$ ./bpf_dbg

# 指定输入输出
$ ./bpf_dbg test_in.txt test_out.txt
```

此外，还支持：

- 在 `~/.bpf_dbg_init` 配置 libreadline；
- 命令历史保存到文件 `~/.bpf_dbg_history` 中；
- 命令行补全。

`load` 命令

加载 **bpf_asm 标准输出文件**，或 **tcpdump -ddd 输出文件** （例如 `tcpdump -iem1 -ddd port 22 | tr '\n' ','` 的输出）：

```
> load bpf 6,40 0 0 12,21 0 3 2048,48 0 0 23,21 0 1 1,6 0 0 65535,6 0 0 0
```

注意：对于 JIT debugging（后面介绍），以上命令会**创建一个临时 socket**， 然后将 BPF 代码加载到内核。因此这对 JIT 开发者也有帮助。

加载**标准 tcpdump pcap 文件**：

```
> load pcap foo.pcap
```

run 命令

对 pcap 内的前 n 个包执行过滤器：

```
> run [<n>]
bpf passes:1 fails:9
```

打印的是命中和未命中过滤规则的包数。

disassemble 命令

反汇编：

```
> disassemble
l0:    ldh [12]
l1:    jeq #0x800, l2, l5
l2:    ldb [23]
l3:    jeq #0x1, l4, l5
l4:    ret #0xffff
l5:    ret #0
```

dump 命令

以 C 风格打印 BPF 代码：

```
$ dump
/* { op, jt, jf, k }, */
{ 0x28,  0,  0, 0x0000000c },
{ 0x15,  0,  3, 0x00000800 },
{ 0x30,  0,  0, 0x00000017 },
{ 0x15,  0,  1, 0x00000001 },
{ 0x06,  0,  0, 0x0000ffff },
{ 0x06,  0,  0, 0000000000 },
```

breakpoint 命令

```
> breakpoint 0
breakpoint at: l0:    ldh [12]

> breakpoint 1
breakpoint at: l1:    jeq #0x800, l2, l5
```

run 命令

在特定指令设置断点之后，执行 run：

```
> run

-- register dump --
pc:       [0]                       <-- program counter
code:     [40] jt[0] jf[0] k[12]    <-- plain BPF code of current instruc
curr:     l0:    ldh [12]           <-- disassembly of current instructio
A:        [00000000][0]             <-- content of A (hex, decimal)
X:        [00000000][0]             <-- content of X (hex, decimal)
M[0,15]:  [00000000][0]             <-- folded content of M (hex, decimal
-- packet dump --                   <-- Current packet from pcap (hex)
len: 42
    0: 00 19 cb 55 55 a4 00 14 a4 43 78 69 08 06 00 01
16: 08 00 06 04 00 01 00 14 a4 43 78 69 0a 3b 01 26
32: 00 00 00 00 00 00 0a 3b 01 01
(breakpoint)

> breakpoint # 打印断点
breakpoints: 0 1
```

step 命令

从当前 pc offset 开始，单步执行：

```
> step [-<n>, +<n>]
```

每执行一步，就会像 run 输出一样 dump 寄存器状态。

注意这里可以单步前进，也可以单步后退。

`select` 命令

选择从第 n 个包开始执行：

```
> select <n> # 接下来执行 run 或 step 命令
```

注意与 Wireshark 一样，n 是从 1 开始的。

`quit` 命令

```
> quit
```

退出 bpf_dbg。

5 cBPF JIT 编译器

Linux 内核内置了一个 BPF JIT 编译器，支持 x86_64、SPARC、PowerPC、ARM、ARM64、MIPS、 RISC-V 和 s390，编译内核时需要打开 **CONFIG_BPF_JIT**。

5.1 内核配置项： `bpf_jit_enable`

启用 JIT 编译器：

```
$ echo 1 > /proc/sys/net/core/bpf_jit_enable
```

如果想每次编译过滤器时，都将生成的 opcode 镜像都**打印到内核日志中**，可以配置：

```
$ echo 2 > /proc/sys/net/core/bpf_jit_enable
```

这对 JIT 开发者和审计来说比较有用。下面是 dmesg 的输出：

```
[ 3389.935842] flen=6 proglen=70 pass=3 image=ffffffffa0069c8f
[ 3389.935847] JIT code: 00000000: 55 48 89 e5 48 83 ec 60 48 89 5d f8 44
[ 3389.935849] JIT code: 00000010: 44 2b 4f 6c 4c 8b 87 d8 00 00 00 be 0c
[ 3389.935850] JIT code: 00000020: e8 1d 94 ff e0 3d 00 08 00 00 75 16 be
[ 3389.935851] JIT code: 00000030: 00 e8 28 94 ff e0 83 f8 01 75 07 b8 ff
[ 3389.935852] JIT code: 00000040: eb 02 31 c0 c9 c3
```

如果在编译时设置了 **CONFIG_BPF_JIT_ALWAYS_ON**， `bpf_jit_enable` 就会 **永久性设为 1**，再设置成其他值时会报错 —— 包括将其设为 2 时，因为 **并不推荐将最终的 JIT 镜像打印到内核日志**，通常推荐开发者通过 bpftool `tools/bpf/bpftool/` 来查看镜像内容。

5.2 工具： `bpf_jit_disasm`

在内核 `tools/bpf/` 目录下还有一个 `bpf_jit_disasm` 工具，用于生成**反汇编**（disassembly），

```
$ ./bpf_jit_disasm
70 bytes emitted from JIT compiler (pass:3, flen:6)
ffffffffa0069c8f + <x>:
0:    push   %rbp                ; 这些已经是 eBPF 指令而非 cBPF 指令，后面
1:    mov    %rsp,%rbp
4:    sub    $0x60,%rsp
8:    mov    %rbx,-0x8(%rbp)
c:    mov    0x68(%rdi),%r9d
10:   sub    0x6c(%rdi),%r9d
14:   mov    0xd8(%rdi),%r8
1b:   mov    $0xc,%esi
20:   callq  0xffffffffe0ff9442
25:   cmp    $0x800,%eax
2a:   jne    0x0000000000000042
2c:   mov    $0x17,%esi
31:   callq  0xffffffffe0ff945e
36:   cmp    $0x1,%eax
39:   jne    0x0000000000000042
```

```
3b:    mov    $0xffff,%eax
40:    jmp    0x0000000000000044
42:    xor    %eax,%eax
44:    leaveq
45:    retq
```

`-o` 参数可以**对照打印字节码和相应的汇编指令**，对 JIT 开发者非常有用：

```
# ./bpf_jit_disasm -o
70 bytes emitted from JIT compiler (pass:3, flen:6)
ffffffffa0069c8f + <x>:
0:    push   %rbp
      55
1:    mov    %rsp,%rbp
      48 89 e5
4:    sub    $0x60,%rsp
      48 83 ec 60
8:    mov    %rbx,-0x8(%rbp)
      48 89 5d f8
c:    mov    0x68(%rdi),%r9d
      44 8b 4f 68
10:   sub    0x6c(%rdi),%r9d
      44 2b 4f 6c
14:   mov    0xd8(%rdi),%r8
      4c 8b 87 d8 00 00 00
1b:   mov    $0xc,%esi
      be 0c 00 00 00
20:   callq  0xffffffffe0ff9442
      e8 1d 94 ff e0
25:   cmp    $0x800,%eax
      3d 00 08 00 00
2a:   jne    0x0000000000000042
      75 16
2c:   mov    $0x17,%esi
      be 17 00 00 00
31:   callq  0xffffffffe0ff945e
      e8 28 94 ff e0
36:   cmp    $0x1,%eax
      83 f8 01
39:   jne    0x0000000000000042
      75 07
```

```
3b:   mov    $0xffff,%eax
      b8 ff ff 00 00
40:   jmp    0x0000000000000044
      eb 02
42:   xor    %eax,%eax
      31 c0
44:   leaveq
      c9
45:   retq
      c3
```

5.3 JIT 开发者工具箱

对 JIT 开发者来说，我们已经介绍的这几个工具：

- bpf_jit_disasm
- bpf_asm
- bpf_dbg

都是非常有用的。

---

eBPF 相关内容

---

6 BPF kernel internals（eBPF）

在内核内部，解释器（the kernel interpreter）使用的是与 cBPF 类似、但属于**另一种指令集的格式**。 这种指令集格式的参考处理器原生指令集建模，因此**更接近底层处理器架构**， 性能更好（后面会详细介绍）。

这种**新的指令集称为“eBPF”**，也叫“internal BPF”。

> 注意：eBPF 这个名字源自 **[e]xtended BPF**（直译为“扩展的 BPF”）， 它与 **BPF extensions**（直译为“BPF 扩展”，见前面章节）并不是一个概念！
>
> - eBPF 是一种**指令集架构（ISA）**，
> - BPF extensions 是早年 **cBPF** 中对 `BPF_LD | BPF_{B,H,W} | BPF_ABS` 几个指令**进行 overloading 的技术**。

6.1 eBPF 设计考虑

- 力求 **JIT 编译时，能将 eBPF 指令一一映射到原生指令**， 这种设计也给 GCC/LLVM 等编译器打开了方便之门，因为它们可以通过各自的 **eBPF 后端生成优化的、与原生编译之后的代码几乎一样快**的 eBPF 代码。
- 最初设计时，目标是能用**“受限的 C 语言”**来编写程序，然后 通过可选的 GCC/LLVM 后端编译成 eBPF，因此它能以两步最小的性能开销，即时映射成 现代 64 位 CPU 指令，即 **C -> eBPF -> native code**。

6.2 cBPF->eBPF 自动转换

下列 cBPF 的经典使用场景中：

- seccomp BPF
- classic socket filters
- cls_bpf traffic classifier
- team driver’s classifier for its load-balancing mode
- netfilter’s xt_bpf extension
- PTP dissector/classifier

- and much more.

cBPF 已经在内核中被**透明地转换成了 eBPF，然后在 eBPF 解释器中执行**。

在 in-kernel handlers 中，可以使用下面的函数：

- `bpf_prog_create()` 创建过滤器；
- `bpf_prog_destroy()` 销毁过滤器。

`BPF_PROG_RUN(filter, ctx)` 执行过滤代码，它或者 透明地**触发 eBPF 解释器执行**，或者**执行 JIT 编译之后的代码**。

- `filter` 是 `bpf_prog_create()` 的返回值，类型是 `struct bpf_prog *` 类型；
- `ctx` 是程序上下文（例如 skb 指针）。

在转换成新指令之前，会通过 `bpf_check_classic()` 检查 cBPF 程序是否有问题。

当前，在大部分 32 位架构上，都是用 cBPF 格式做 JIT 编译； 而在 x86-64, aarch64, s390x, powerpc64, sparc64, arm32, riscv64, riscv32 架构上，直接从 eBPF 指令集执行 JIT 编译。

6.3 eBPF 相比 cBPF 的核心变化

6.3.1 寄存器数量从 2 个增加到 10 个

- 老格式（cBPF）只有两个寄存器 A 和 X，以及一个隐藏的栈指针（frame pointer）。
- 新格式（eBPF）扩展到了 10 个内部寄存器，以及一个只读的栈指针。

传参寄存器数量

由于 64 位 CPU 都是**通过寄存器传递函数参数**的，因此从 eBPF 程序 传给内核函数（in-kernel function）的**参数数量限制到 5 个**，另有 一个寄存器用来接收内核函数的返回值，

考虑到**具体的处理器架构**，

- x86_64 有 6 寄存器用于传参，6 个被调用方（callee）负责保存的寄存器；
- aarch64/sparcv9/mips64 有 7-8 个传参寄存器，11 个被调用方（callee）负责保存的寄存器。

eBPF 调用约定

因此，**eBPF 调用约定**（calling convention）定义如下：

- R0 - return value from in-kernel function, and exit value for eBPF program
- R1 - R5 - arguments from eBPF program to in-kernel function
- R6 - R9 - callee saved registers that in-kernel function will preserve
- R10 - read-only frame pointer to access stack

这样的设计，使**所有的 eBPF 寄存器都能一一映射到** x86_64、aarch64 等架构上的**硬件寄存器**，eBPF 调用约定也直接映射到 64 位的内核 ABI。

在 32 位架构上，JIT 只能编译那些只使用了 32bit 算术操作的程序，其他更复杂的程序，交给解释器来执行。

6.3.2 寄存器位宽从 32bit 扩展到 64bit

原来的 32bit ALU 操作仍然是支持的，通过 32bit 子寄存器执行。 所有 eBPF 寄存器都是 64bit 的，如果对 32bit 子寄存器有写入操作，它会被 zero-extend 成 64bit。 这种行为能直接映射到 x86_64 和 arm64 子寄存器的定义，但对其他处理器架构来说，JIT 会更加困难。

32-bit 的处理器架构上，通过解释器执行 64-bit 的 eBPF 程序。这种平台上的 JIT 编译器 只能编译那些只使用了 32bit 子寄存器的程序，其他不能编译的程序，通过解释器执行。

eBPF 操作都是 64 位的，原因：

1. 64 位处理器架构上指针也是 64 位的，我们希望与内核函数交互时，输入输出的都

是 64 位值。如果使用 32 位 eBPF 寄存器，就需要定义 register-pair ABI，导致无法 直接将 eBPF 寄存器映射到物理寄存器，JIT 就必须为与函数调用相关的每个寄存器承 担 拼装/拆分/移动 等等操作，不仅复杂，而且很容易产生 bug，效率也很低。

2. 另一个原因是 eBPF 使用了 64 位的原子计数器（atomic 64-bit counters）。

6.3.3 条件跳转： `jt/fall-through` 取代 `jt/jf`

cBPF 的设计中有条件判断：

```
if (cond)
    jump_true;
else
    jump_false;
```

现在被下面的结构取代了：

```
if (cond)
    jump_true;

/* else fall-through */
```

6.3.4 引入 `bpf_call` 指令和寄存器传参约定，实现零（额外）开销内核函数调用

引入的寄存器传参约定，能实现 零开销内核函数调用（zero overhead calls from/to other kernel functions）。

在调用内核函数之前，eBPF 程序需要按照调用约定，将参数依次放到 R1-R5 寄存器；然后解释器会从这些寄存器读取参数，传递给内核函数。

原理：JIT 实现零（额外）开销内核函数调用

如果 R1-R5 能一一映射到处理器上的寄存器，JIT 编译器就**无需 emit 任何额外的指令**：

- 函数**参数已经在（硬件处理器）期望的位置**；
- 只需要将 eBPF `BPF_CALL` **JIT 编译成一条处理器原生的 `call` 指令**就行了。
- 这种**无性能损失的调用约定**设计，足以覆盖大部分场景。

函数调用结束后，R1-R5 会被重置为不可读状态（unreadable），函数返回值存放在 R0。 R6-R9 是被调用方（callee）保存的，因此函数调用结束后里面的值是可读的。

示例解析（一）：eBPF/C 函数混合调用，JIT 生成的 x86_64 指令

考虑下面的三个 C 函数：

```
u64 f1() { return (*_f2)(1); }
u64 f2(u64 a) { return f3(a + 1, a); }
u64 f3(u64 a, u64 b) { return a - b; }
```

GCC 能将 f1 和 f3 编译成 x86_64：

```
f1:
movl $1, %edi          ; 将常量 1 加载到 edi 寄存器
movq _f2(%rip), %rax   ; 将 _f2 地址加载到 rax 寄存器
jmp  *%rax             ; 跳转到 rax 寄存器中指定的地址（即函数 _f2 的起始地址）

f3:
movq %rdi, %rax        ; 将寄存器 rdi 中的值加载到寄存器 rax
subq %rsi, %rax        ; 将寄存器 rax 中的值减去寄存器 rsi 中的值（即 a-b）
ret                    ; 返回
```

f2 的 eBPF 代码可能如下：

```
f2:
bpf_mov R2, R1          ; 即 R2 = a
bpf_add R1, 1           ; 即 R1 = a + 1
bpf_call f3             ; 调用 f3，传递给 f3 的两个参数分别在 R1 和 R2 中
bpf_exit                ; 退出
```

- 如果 **f2 是 JIT 编译的**，函数地址保存为变量 `_f2`，那调用链 `f1 -> f2 -> f3` 及返回就都是连续的。
- 如果**没有 JIT**，就需要**调用解释器 `__bpf_prog_run()`** 来调用执行 f2。

出于一些实际考虑，

1. **所有 eBPF 程序都只有一个参数 `ctx`，放在 R1 寄存器中**，例如 `__bpf_prog_run(ctx)`。
2. 函数调用最多支持传递 5 个参数，但如果将来有需要，这个限制也可以进一步放宽。

eBPF 寄存器到 x86_64 硬件寄存器一一映射关系

在 64 位架构上，所有寄存器都能一一映射到硬件寄存器。例如，由于 **x86_64 ABI 硬性规定**了

1. **rdi**、**rsi**、**rdx**、**rcx**、**r8**、 r9 寄存器作为参数传递；
2. **rbx**，以及 **r12 - r15** 由被调用方（callee）保存；

因此 **x86_64 编译会做如下映射**：

```
R0 -> rax

R1 -> rdi  ; 传参，调用方（caller）保存
R2 -> rsi  ; 传参，调用方（caller）保存
R3 -> rdx  ; 传参，调用方（caller）保存
R4 -> rcx  ; 传参，调用方（caller）保存
R5 -> r8   ; 传参，调用方（caller）保存

R6 -> rbx  ; 被调用方（callee）保存
R7 -> r13  ; 被调用方（callee）保存
R8 -> r14  ; 被调用方（callee）保存
R9 -> r15  ; 被调用方（callee）保存
R10 -> rbp ; 被调用方（callee）保存
...
```

示例解析（二）：C 调 eBPF 代码编译成 x86_64 汇编后的样子

根据上面的映射关系，下面的 BPF 程序：

```
// BPF 指令格式：
// <指令>  <目的寄存器>  <源寄存器/常量>

bpf_mov  R6, R1 ; 将 ctx 保存到 R6
bpf_mov  R2, 2  ; 将常量 2（即将调用的函数 foo() 的参数）加载到 R2 寄存器
bpf_mov  R3, 3
bpf_mov  R4, 4
bpf_mov  R5, 5
bpf_call foo    ; 调用 foo 函数
bpf_mov  R7, R0 ; 将 foo() 的返回值（在 R0 中）保存到 R7 中
bpf_mov  R1, R6 ; 从 R6 中恢复 ctx 状态，保存到 R1；这样下次执行调用函数调用时就可
bpf_mov  R2, 6  ; 将常量 2（即将调用的函数 bar() 的参数）加载到 R2 寄存器
bpf_mov  R3, 7
bpf_mov  R4, 8
bpf_mov  R5, 9
bpf_call bar    ; 调用 bar() 函数
bpf_add  R0, R7 ; 将 bar() 的返回值（在 R0 中）与 foo() 的返回值（在 R7 中）相加
bpf_exit
```

在 JIT 成 x86_64 之后，可能长下面这样：

> 将“eBPF 寄存器 -> x86_64 硬件寄存器”映射关系贴到这里方便下面程序对照

```
R0 -> rax
R1 -> rdi  // 传参，调用方（caller）保存
R2 -> rsi  // 传参，调用方（caller）保存
R3 -> rdx  // 传参，调用方（caller）保存
R4 -> rcx  // 传参，调用方（caller）保存
R5 -> r8   // 传参，调用方（caller）保存
R6 -> rbx  // 被调用方（callee）保存
R7 -> r13  // 被调用方（callee）保存
R8 -> r14  // 被调用方（callee）保存
R9 -> r15  // 被调用方（callee）保存
R10 -> rbp // 被调用方（callee）保存
```

```
// x86_64 指令格式：注意源和目的寄存器的顺序与 BPF 指令是相反的
// <指令>  <源寄存器/常量>  <目的寄存器>

// 下面这几行是 x86_64 的初始化指令，与 eBPF 还没有直接对应关系
// 解读参考：https://stackoverflow.com/questions/41912684/what-is-the-purpo
push   %rbp                // 将帧指针（frame pointer）在栈地址空间中前移，即栈
mov    %rsp, %rbp          // 将栈指针（stack pointer）保存到 %rbp 位置（即上-
sub    $0x228, %rsp        // 栈指针 rsp -= 0x228（栈向下增长，这一行表示再分配
mov    %rbx, -0x228(%rbp)  // 将 %rbx（对应 eBPF R6）的值保存到新分配空间的起始
mov    %r13, -0x220(%rbp)  // 将 %r13 （对应 eBPF R7）的值保存到下一个位置（起始
                           // 接下来还应该有三条指令，分别将 r14、15、rbp 依次保
                           // 这样，这 5 条指令占用 5*8 = 40byte = 0x28 字节。
                           // 0x228 - 0x28 = 0x200 = 512 字节，也就是 eBPF


// 下面这段与上面的 eBPF 指令能一一对应上
mov    %rdi, %rbx          // R6 = R1
mov    $0x2, %esi          // R2 = 2
mov    $0x3, %edx          // R3 = 3
mov    $0x4, %ecx          // R4 = 4
mov    $0x5, %r8d          // R5 = 5
callq  foo
mov    %rax, %r13          // R7 = R0
mov    %rbx, %rdi          // R1 = R6
mov    $0x6, %esi          // R2 = 6
mov    $0x7, %edx          // R3 = 7
mov    $0x8, %ecx          // R4 = 8
mov    $0x9, %r8d          // R5 = 9
callq  bar
add    %r13, %rax          // R7 += R0

mov    -0x228(%rbp), %rbx
mov    -0x220(%rbp), %r13
leaveq
retq
```

下面是对应的 C 代码：

```
u64 bpf_filter(u64 ctx) {
    return foo(ctx, 2, 3, 4, 5) + bar(ctx, 6, 7, 8, 9);
}
```

内核函数 `foo()` 和 `bar()` 原型：

```
u64 (*)(u64 arg1, u64 arg2, u64 arg3, u64 arg4, u64 arg5);
```

它们从规定好的寄存器中获取传入参数，并将函数返回值放到 `%rax` 寄存器，也就是 eBPF 中的 R0 寄存器。 起始和结束的汇编片段**（prologue and epilogue）是由 JIT emit 出来的，是解释器内置的**。

> 上面添加了对起始汇编片段的一些解读，尤其是：为什么**“eBPF 虚拟机的最大栈空间是 512 字节”**。 译注。

R0-R5 are scratch registers，因此 eBPF 程序需要在多次函数调用之间保存这些值。 下面这个程序例子是不合法的：

```
bpf_mov   R1, 1
bpf_call  foo
bpf_mov   R0, R1
bpf_exit
```

在执行 `call` 之后，寄存器 **R1-R5 包含垃圾值，不能读取**。 内核中的校验器（in-kernel eBPF verifier）负责验证 eBPF 程序的合法性。

6.4 eBPF 程序最大指令数限制

eBPF 最初限制最大指令数 4096，现在已经将这个限制放大到了 **100 万条**。

cBPF 和 eBPF 都是**两操作数指令**（two operand instructions），有 助于 JIT 编译时将 eBPF 指令一一映射成 x86 指令。

6.5 eBPF 程序上下文（ctx）参数

触发解释器执行时，传递给它的上下文指针（the input context pointer）是一个通用结构体， **结构体中的信息是由具体场景来解析的**。例如

- 对于 seccomp 来说，R1（也就是 `ctx`）指向 seccomp_data，
- 对于 eBPF filter（包括从 cBPF 转换过来的）来说，R1 **指向一个 skb**。

6.6 cBPF -> eBPF 转换若干问题

内部的 cBPF -> eBPF 转换格式如下：

```
op:16, jt:8, jf:8, k:32    ==>    op:8, dst_reg:4, src_reg:4, off:16, imm
```

- 目前已经实现了 87 条 eBPF 指令；
- 8bit 的 `op` 字段最多支持 256 条，因此还有扩展空间，可用于增加新指令；
- Some of them may use 16/24/32 byte encoding；
- eBPF 指令必须是 8 字节对齐的，以保持后向兼容。

eBPF 是一个**通用目的 RISC 指令集**。在将 cBPF 转成 eBPF 的过程中，不是每个寄存器和每条指令都会用到。例如，

- socket filters 没有用到 `exclusive add` 指令，而 tracing filters 可能会在维护 事件计数器时用到这种 add；
- socket filters 也没有用到 R9 寄存器，但更加复杂的 filters 可能会用完所有的寄 存器（还不够），因此还有用的 spill/fill to stack 之类的技术。

某种意义上来说，eBPF 作为一个**通用汇编器**（generic assembler），是性能优化的最后手段，

- socket filters 和 seccomp 就是把它当做汇编器在用；
- Tracing filters 可以用它作为汇编器，从内核生成代码。

6.7 eBPF 的安全性

内核内使用（in kernel usage）可能没有安全顾虑，因为生成的 eBPF 代码只是用于优化内核内部代码路径，不会暴露给用户空间。eBPF 的安全问题可能会出自校验器本身

（TBD ）。因此在上述这些场景，可以把它作为一个安全的指令集来使用。

与 cBPF 类似，eBPF 运行在一个确定性的受控环境中，内核能依据下面两个步骤，轻松地对 **程序的安全性**作出判断：

1. 首先进行**深度优先搜索**（depth-first-search），禁止循环；其他 CFG 验证。
2. 以上一步生成的指令作为输入，**遍历所有可能的执行路径**。具体说 就是模拟每条指令的执行，**观察寄存器和栈的状态变化**。

7 eBPF 字节码编码（opcode encoding）

为方便 cBPF 到 eBPF 的转换，eBPF 复用了 cBPF 的大部分 opcode encoding。

7.1 算术和跳转指令

对于算术和跳转指令（arithmetic and jump instructions），eBPF 的 8bit `op` 字段进一步分为三部分：

```
+----------------+--------+--------------------+
|   4 bits       |  1 bit |   3 bits           |
| operation code | source | instruction class  |
+----------------+--------+--------------------+
(MSB)                                      (LSB)
```

最后的 3bit 表示的是指令类型，包括：

```
==================     ===============
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
===================     ===============
```

BPF_ALU 和 BPF_JMP 的 operand

当 `BPF_CLASS(code) == BPF_ALU or BPF_JMP` 时，第 4 bit 表示的源操作数（source operand）可以是：

```
BPF_K     0x00 // 32bit 立即数作为源操作数 (use 32-bit immediate as source o
BPF_X     0x08 // 对 cBPF, 表示用寄存器 X        作为源操作数
               // 对 eBPF, 表示用寄存器 src_reg 作为源操作数
```

BPF_ALU 和 BPF_ALU64 (eBPF) 的 opcode

`BPF_CLASS(code) == BPF_ALU or BPF_ALU64 [ in eBPF ]`, `BPF_OP(code)` 可以是：

```
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 */
```

BPF_JMP 和 BPF_JMP32 (eBPF) 的 opcode

`BPF_CLASS(code) == BPF_JMP or BPF_JMP32 [ in eBPF ]`, `BPF_OP(code)` 可以

是：

```
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 '<=' */
```

指令 **`BPF_ADD | BPF_X | BPF_ALU`** 在 cBPF 和 eBPF 中都表示 32bit 加法：

- cBPF 中只有两个寄存器，因此它表示的是 `A += X`；
- eBPF 中表示的是 `dst_reg = (u32) dst_reg + (u32) src_reg` 。

类似的， **`BPF_XOR | BPF_K | BPF_ALU`** 表示：

- cBPF 中： `A ^= imm32`；
- eBPF 中： `src_reg = (u32) src_reg ^ (u32) imm32` 。

BPF_MISC 与 BPF_ALU64（eBPF 64bit 寄存器加法操作）

cBPF 用 BPF_MISC 类型表示 A = X 和 X = A 操作。

eBPF 中与此对应的是 `BPF_MOV | BPF_X | BPF_ALU` 。 由于 eBPF 中没有 BPF_MISC 操作，因此 class 7 空出来了，用作了新指令类型 BPF_ALU64，表示 64bit BPF_ALU 操作。 因此， `BPF_ADD | BPF_X | BPF_ALU64` 表示 64bit 加法，例如 `dst_reg = dst_reg + src_reg` 。

cBPF/eBPF BPF_RET 指令的不同

cBPF 用整个 BPF_RET class 仅仅表示一个 `ret` 操作，非常浪费。 其 `BPF_RET | BPF_K` 表示将立即数 imm32 拷贝到返回值寄存器，然后退出函数。

eBPF 是模拟 CPU 建模的，因此 eBPF 中 `BPF_JMP | BPF_EXIT` 只表示退出函数操作。 eBPF 程序自己负责在执行 BPF_EXIT 之前，将返回值拷贝到 R0。

BPF_JMP 与 eBPF BPF_JMP32

Class 6 in eBPF 用作 BPF_JMP32，表示的意思与 BPF_JMP 一样，但操作数是 32bit 的。

7.2 加载指令（load/store）

load and store 指令的 8bit code 进一步分为三部分：

```
+--------+--------+-------------------+
| 3 bits | 2 bits |   3 bits          |
|  mode  |  size  | instruction class |
+--------+--------+-------------------+
(MSB)                             (LSB)
```

2bit 的 size modifier 可以是：

```
BPF_W   0x00    /* word */
BPF_H   0x08    /* half word */
BPF_B   0x10    /* byte */
BPF_DW  0x18    /* eBPF only, double word */
```

表示的是 load/store 操作的字节数：

```
B  - 1 byte
H  - 2 byte
W  - 4 byte
DW - 8 byte (eBPF only)
```

Mode modifier 可以是：

```
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_XADD 0xc0  /* eBPF only, exclusive add */
```

两个 eBPF non-generic 指令：BPF_ABS 和 BPF_IND，用于访问 skb data

eBPF 有两个 non-generic 指令，用于兼容 cBPF：

1. `BPF_ABS | <size> | BPF_LD`
2. `BPF_IND | <size> | BPF_LD`

二者用来**访问数据包中的数据**（packet data）。

1. 这两个指令 **cBPF 中就有了，必须 eBPF 也必须要支持**，而且 eBPF 解释器还要**高效地执行**这两条指令。
2. 执行这个两个指令时，**传递给解释器的上下文**必须是 `struct *sk_buff`， 并且**隐含了 7 个操作数**：
   - R6 作为**隐式输入**，存放的必须是 `struct *sk_buff`；
   - R0 作为**隐式输出**，存放的是从包中读取的数据；
   - R1-R5 作为 scratch registers，不能在多次 `BPF_ABS | BPF_LD` 或 `BPF_IND | BPF_LD` 指令之间在这几个寄存器中保存数据（每次调用执行之后，都会将这些寄存器置空）；
3. 这些指令还有隐含的程序退出条件。当 eBPF 程序试图访问数据包边界之外的数据时，解释器 会终止（abort）程序的执行。因此，eBPF JIT 编译器也必须实现这个特性。 `src_reg` 和 imm32 字段是这些指令的显式输入。

看个例子， **`BPF_IND | BPF_W | BPF_LD`** 翻译成 C 语言表示： **`R0 = ntohl(*(u32 *)(((struct sk_buff *) R6)->data + src_reg + imm32))`** 。 过程中会用到 R1-R5（R1-R5 were scratched）。

通用 eBPF load/store 指令

与 cBPF 指令集不同，eBPF 有通用 load/store 操作：

```
BPF_MEM  | <size> | BPF_STX:  *(size *)(dst_reg + off) = src_reg
BPF_MEM  | <size> | BPF_ST :  *(size *)(dst_reg + off) = imm32
BPF_MEM  | <size> | BPF_LDX:  dst_reg = *(size *)(src_reg + off)
BPF_XADD | BPF_W  | BPF_STX:  lock xadd *(u32 *)(dst_reg + off16) += src_
BPF_XADD | BPF_DW | BPF_STX:  lock xadd *(u64 *)(dst_reg + off16) += src_
```

其中，size 是：

1. `BPF_B`
2. `BPF_H`
3. `BPF_W`
4. `BPF_DW`

注意：这里不支持 1 或 2 字节的原子递增操作。

加载 64bit 立即数的 eBPF 指令

eBPF 有一个 16-byte instruction: `BPF_LD | BPF_DW | BPF_IMM` ， 功能是将 64bit 立即数（imm64）加载到寄存器：

1. 这条指令由两个连续 `struct bpf_insn` 8-byte blocks 组成，会被解释器解释为单个指令，
2. 功能就是上面提到的，将一个 64bit 立即值 load 到 `dst_reg` 。

cBPF 中有一个类似指令 `BPF_LD | BPF_W | BPF_IMM` ， 功能是将一个 32bit 立即值（imm）加载到寄存器。

8 eBPF 校验器（eBPF verifier）

eBPF 程序的安全是通过两个步骤来保证的：

- 首先做一次 DAG 检查，确保没有循环，并执行其他 CFG validation。特别地，这会检查程序中是否有 无法执行到的指令（unreachable instructions，虽然 cBPF checker 是允许的）。
- 第二步是从程序的第一条指令开始，遍历所有的可能路径。这一步会模拟执行每一条指令，在过程中观察寄存器和栈的状态变化。

8.1 模拟执行

- 程序开始时，**R1 中存放的是上下文指针**（ `ctx` ），类型是 `PTR_TO_CTX` 。
  - 接下来，如果校验器看到 **R2=R1，那 R2 的类型也变成了 PTR_TO_CTX**，并且接下来就能用在表达式的右侧。
  - 如果 R1=PTR_TO_CTX 接下来的指令是 **R2=R1+R1 ，那 R2=SCALAR_VALUE**，因为**两个合法指针相加，得到的是一个非法指针**。（在“secure”模式下， 校验器会拒绝任何类型的指针运算，以确保内核地址不会泄露给 unprivileged users）。
- **从来没有写入过数据的寄存器是不可读的**，例如：

  ```
  bpf_mov R0 = R2
  bpf_exit
  ```

  将会被拒绝，因为程序开始之后，R2 还没有初始化过。
- 内核函数**执行完成后，R1-R5 将被重置为不可读状态**，R0 保存函数的返回值。
- 由于 R6-R9 是被调用方（callee）保存的，因此它们的状态在函数调用结束之后还是有效的。

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

以上程序是合法的。如果换成了 `R0 = R1` ，就会被拒绝。

8.2 load/store 指令检查

load/store 指令只有当**寄存器类型合法时**才能执行，这里的类型包括 PTR_TO_CTX、PTR_TO_MAP、PTR_TO_STACK。会对它们做边界和对齐检查。例如：

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

将会被拒，因为执行到第三行时，R1 并不是一个合法的指针类型。

8.3 定制化校验器，限制程序只能访问 `ctx` 特定字段

程序开始时，R1 类型是 PTR_TO_CTX（指向通用类型 `struct bpf_context` 的指针）。可以**通过 callback 定制化校验器**，指定 size 和对齐，来 **限制 eBPF 程序只能访问 ctx 的特定字段**。例如，下面的指令：

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

- 如果 R6=PTR_TO_CTX，通过 `is_valid_access()` callback，校验器就能知道从 offset 8 处读取 4 个字节的操作是合法的，否则，校验器就会拒绝这个程序。
- 如果 R6=PTR_TO_STACK，那访问就应该是对齐的，而且在栈空间范围内，即 `[-MAX_BPF_STACK, 0)`。在这里例子中 offset 是 8，因此校验会失败，因为超出 栈空间边界。

8.4 读取栈空间

只有程序**向栈空间写入数据后，校验器才允许它从中读取数据**。cBPF 通过 `M[0-15]` memory slots 执行类似的检查，例如

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

是非法程序。因为虽然 R10 是只读寄存器，类型 PTR_TO_STACK 也是合法的，并且 `R10 - 4` 也在栈边界内，但在这次读取操作之前，并没有往这个位置写入数据。

8.5 其他

- 指针寄存器（pointer register）spill/fill 操作也会被跟踪，因为 **对一些程序来说，四个 (R6-R9) callee saved registers 显然是不够的**。
- 可通过 `bpf_verifier_ops->get_func_proto()` 来**定制允许执行哪些函数**。 eBPF 校验器会检查寄存器与参数限制是否匹配。调用结束之后，R0 用来存放函数返回值。
- **函数调用**是扩展 eBPF 程序功能的主要机制，但每种类型的 BPF 程 序能用到的函数是不同的，例如 socket filters 和 tracing 程序。
- 如果一个函数设计成对 eBPF 可见的，那必须从安全的角度对这个函数进行考量。校验 器会保证调用该函数时，参数都是合法的。
- cBPF 中， seccomp 的安全限制与 socket filter 是不同的，它依赖**两个级联的校验器**：
  - 首先执行 cBPF verifier，
  - 然后再执行 seccomp verifier

  而在 **eBPF 中，所有场景都共用一个（可配置的）校验器**。

更多关于 eBPF 校验器的信息，可参考 kernel/bpf/verifier.c。

9 寄存器值跟踪（register value tracking）

为保证 eBPF 程序的安全，校验器必须跟踪每个**寄存器**和**栈上每个槽位**（stack slot）值的范围。这是通过 **`struct bpf_reg_state`** 实现的，定义在 include/linux/bpf_verifier.h，它**统一了对标量和指针类型的跟踪**（scalar and pointer values）。

每个**寄存器状态**都有一个**类型**，

- `NOT_INIT`：该寄存器还未写入数据
- `SCALAR_VALUE`：标量值，不可作为指针
- 指针类型

9.1 9 种指针类型

依据它们**指向的数据结构类型**，又可以分为：

1. `PTR_TO_CTX`：指向 **`bpf_context`** 的指针。
2. `CONST_PTR_TO_MAP`：指向 **`struct bpf_map`** 的指针。是**常量**（const），因为不允许对这种类型指针进行算术操作。
3. `PTR_TO_MAP_VALUE`：指向 bpf **map 元素**的指针。
4. `PTR_TO_MAP_VALUE_OR_NULL`：指向 bpf map 元素的指针，可为 NULL。**访问 map 的操作**会返回这种类型的指针。**禁止算术操作**。
5. `PTR_TO_STACK`：帧指针（Frame pointer）。
6. `PTR_TO_PACKET`：指向 **`skb->data`** 的指针。
7. `PTR_TO_PACKET_END`：指向 **`skb->data + headlen`** 的指针。禁止算术操作。
8. `PTR_TO_SOCKET`：指向 **`struct bpf_sock_ops`** 的指针，内部有引用计数。
9. `PTR_TO_SOCKET_OR_NULL`：指向 **`struct bpf_sock_ops`** 的指针，或 NULL。

socket lookup 操作会返回这种类型。**有引用计数**， 因此程序在执行结束时，必须通过 socket release 函数释放引用。禁止算术操作。

这些指针都称为 base 指针。

9.2 指针偏移（offset）触发寄存器状态更新

实际上，很多有用的指针都是 base 指针加一个 offset（指针算术运算的结果）， 这是通过两方面来个跟踪的：

1. ‘fixed offset’（**固定偏移**）： offset 是个常量（例如，立即数）。
2. ‘variable offset’（**可变偏移**）： offset 是个变量。这种类型还用在 SCALAR_VALUE 跟踪中，来跟踪寄存器值的可能范围。

校验器对可变 offset 的知识包括：

1. 无符号类型：最小和最大值；
2. 有符号类型：最小和最大值；
3. 关于每个 bit 的知识，以 ‘tnum’ 的格式: 一个 u64 ‘mask’ 加一个 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。 例如，如果从内存加载一个字节到寄存器，那该寄存器的前 56bit 已知是全零，而后 8bit 是未知的 —— 表示为 `tnum（0x0; 0xff）`。如果我们将这个值与 0x40 进行 OR 操作，就得到 `（0x40; 0xbf）`；如果加 1 就得到 `（0x0; 0x1ff）`，因为可能的进位操 作。

9.3 条件分支触发寄存器状态更新

除了算术运算之外，条件分支也能更新寄存器状态。例如，如果判断一个 SCALAR_VALUE 大于 8，那

- 在 true 分支，这个变量的最小值 `umin_value` （unsigned minimum value）就是

9；

- 在 false 分支，它的最大值就是 umax_value of 8。

9.4 有符号比较触发寄存器状态更新

有符号比较（BPF_JSGT or BPF_JSGE）也会相应更新有符号变量 的最大最小值。

有符合和无符号边界的信息可以结合起来；例如如果一个值先判断小于无 符号 8，后判断大于有符合 4，校验器就会得出结论这个值大于无符号 4，小于有符号 8，因为这个边界不会跨正负边界。

9.5 `struct bpf_reg_state` 的 `id` 字段

`struct bpf_reg_state` 结构体有一个 `id` 字段，

```
// include/linux/bpf_verifier.h

    /* For PTR_TO_PACKET, used to find other pointers with the same varia
     * offset, so they can share range knowledge.
     * For PTR_TO_MAP_VALUE_OR_NULL this is used to share which map value
     * came from, when one is tested for != NULL.
     * For PTR_TO_MEM_OR_NULL this is used to identify memory allocation
     * for the purpose of tracking that it's freed.
     * For PTR_TO_SOCKET this is used to share which pointers retain the
     * same reference to the socket, to determine proper reference freein
     */
    u32 id;
```

如注释所述，该字段针对不同指针类型有不同用途，下面分别解释。

`PTR_TO_PACKET`

`id` 字段对**共享同一 variable offset 的多个 PTR_TO_PACKET 指针** 都是可见的，这对**skb 数据的范围检查**非常重要。举个例子：

```
1:  A = skb->data // A 是指向包数据的指针
2:  B = A + var2  // B 是从 A 开始往前移动 var2 得到的地址
3:  A = A + 4     // A 往前移动 4 个字节
```

在这个程序中，寄存器 **A 和 B 将将共享同一个 `id`**，

- A 已经从最初地址向前移动了 4 字节（有一个固定偏移 +4），
- 如果这个边界通过校验了，也就是确认小于 `PTR_TO_PACKET_END`，那现在 **寄存器 B 将有一个范围至少为 4 字节的可安全访问范围**。

更多关于这种指针的信息，见下面的 'Direct packet access' 章节。

`PTR_TO_MAP_VALUE`

与上面的用途类型，具体来说：

1. 这一字段对共享同一基础指针的多个 PTR_TO_MAP_VALUE 指针可见；
2. 这些指针中，**只要一个指针经验证是非空的，就认为其他指针（副本）都是非空的**（因此减少重复验证开销）；

另外，与 range-checking 类型，跟踪的信息（the tracked information）还用于**确保指针访问的正确对齐**。 例如，在大部分系统上，packet 指针都 4 字节对齐之后再加 2 字节。如果一个程序将这个指针加 14（跳过 Ethernet header）然后读取 IHL，并将指针再加上 `IHL * 4`，最终的指针将有一个 `4n + 2` 的 variable offset，因此，加 2（`NET_IP_ALIGN`） gives a 4-byte alignment，因此通过这个指针进行 word-sized accesses 是安全的。

`PTR_TO_SOCKET`

与上面用途类似，只要一个指针验证是非空的，其他共享同一 `id` 的 PTR_TO_SOCKET 指针就都是非空的；此外， 还**负责跟踪指针的引用**（reference tracking for the pointer）。

PTR_TO_SOCKET 隐式地表示对一个 `struct sock` 的引用。为确保引用没有泄露，需

要强制对引用进行非空（检查）， 如果非空（non-NULL），将合法引用传给 socket release 函数。

10 直接数据包访问（direct packet access）

对于 cls_bpf 和 act_bpf eBPF 程序，校验器允许**直接通过 `skb->data` 和 `skb->data_end` 指针访问包数据**。

10.1 简单例子

```
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 */
```

上面从包数据中加载 2 字节的操作是安全的，因为**程序编写者在第五行主动检查了数据边界**： `if（skb->data + 14 > skb->data_end）goto err`，这意味着能执行到第 6 行时（fall-through case）， R3（ `skb->data` ）至少有 14 字节的直接可访问数据，因此 **校验器将其标记为 `R3=pkt(id=0,off=0,r=14)`**：

- `id=0` 表示**没有额外的变量加到这个寄存器上**；
- `off=0` 表示**没有额外的常量 offset**；
- `r=14` 表示**安全访问的范围**，即 `[R3，R3+14）` 指向的字节范围都是 OK 的。

这里注意 **R5 被标记为 `R5=pkt(id=0,off=14,r=14)`**，

- 它也指向包数据，但**常量 14 加到了寄存器**，因为它执行的是 `skb->data + 14`，
- 因此可访问的范围是 `[R5，R5 + 14 – 14）`，也就是 0 个字节。

10.2 复杂例子

下面是个更复杂一些的例子：

```
R0=inv1 R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt_end R5=pkt(id=0,off=14,r=14
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,of
19:  r1 = *(u8 *)(r3 +4)
```

校验器标记信息解读

第 18 行之后，寄存器 R3 的状态是 `R3=pkt(id=2,off=0,r=8)`，

- `id=2` 表示**之前已经跟踪到两个 `r3 += rX` 指令**，因此 r3 指向某个包内的某个 offset，由于程序员在 18 行已经做了 `if (r3 + 8 > r1) goto err` 检查，因此**安全范围**是 `[R3, R3 + 8)`。
- 校验器**只允许对 packet 寄存器执行 add/sub** 操作。其他操作会将 寄存器状态设为 **SCALAR_VALUE，这个状态是不允许执行 direct packet access 的**。

操作 `r3 += rX` **可能会溢出，变得比起始地址 skb->data 还小**，校验器必须要能检查出这种情况。 因此当它看到 `r3 += rX` 指令并且 rX 比 16bit 值还大时，接下来的任何将 r3 与 `skb->data_end` 对比的操作都**不会返回范围信息**，因此尝试通过 这个指针读取数据的操作都会收到 **invalid access to packet** 错误。 例如，

- `r4 = *(u8 *)(r3 +12)` 之后，r4 的状态是 `R4=inv(id=0,umax_value=255,var_off=(0x0; 0xff))`，意思是 寄存器的 upper 56 bits 肯定是 0，但对于低 8bit 信息一无所知。 在执行完 `r4 *= 14` 之后，状态变成 `R4=inv(id=0,umax_value=3570,var_off=(0x0; 0xfffe))`，因为一个 8bit 值乘以 14 之后， 高 52bit 还是 0，此外最低 bit 位为 0，因为 14 是偶数。
- 类似地， `r2 >>= 48` 使得 `R2=inv(id=0,umax_value=65535,var_off=(0x0; 0xffff))`，因为移位是无符号扩展。 这个逻辑在函数 `adjust_reg_min_max_vals()` 中实现，它又会调用
  - `adjust_ptr_min_max_vals()`
  - `adjust_scalar_min_max_vals()`

对应的 C 代码

最终的结果是：eBPF 程序编写者可以像使用普通 C 语言一样访问包数据：

```
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)
    ...;
```

相比使用 LD_ABS 之类的指令，这种程序写起来方便多了。

11 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

> 以上介绍非常简单，更多信息可参考：
>
> - BPF 进阶笔记（二）：BPF Map 类型详解：使用场景、程序示例
> - BPF 进阶笔记（三）：BPF Map 内核实现
>
> 译注。

12 Pruning（剪枝）

校验器实际上**并不会模拟执行程序的每一条可能路径**。

对于每个新条件分支：校验器首先会查看它自己当前已经跟踪的所有状态。如果这些状态 已经覆盖到这个新分支，该分支就会被剪掉（pruned）—— 也就是说之前的状态已经被接受 （previous state was accepted）能证明当前状态也是合法的。

举个例子：

1. 当前的状态记录中，r1 是一个 packet-pointer
2. 下一条指令中，r1 仍然是 packet-pointer with a range as long or longer and at least as strict an alignment，那 r1 就是安全的。

类似的，如果 r2 之前是 `NOT_INIT`，那就说明之前任何代码路径都没有用到这个寄存器，因此 r2 中的任何值（包括另一个 NOT_INIT）都是安全的。

实现在 `regsafe()` 函数。

Pruning 过程不仅会看寄存器，还会看栈（及栈上的 spilled registers）。 只有证明二者都安全时，这个分支才会被 prune。这个过程实现在 `states_equal()` 函数。

13 理解 eBPF 校验器提示信息

提供几个不合法的 eBPF 程序及相应校验器报错的例子。

The following are few examples of invalid eBPF programs and verifier error messages

as seen in the log:

13.1 程序包含无法执行到的指令

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

Error:

```
unreachable insn 1
```

13.2 程序读取未初始化的寄存器

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

Error:

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

13.3 程序退出前未设置 R0 寄存器

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

Error:

```
0: (bf) r2 = r1
```

```
1: (95) exit
R0 !read_ok
```

13.4 程序访问超出栈空间

```
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
```

13.5 未初始化栈内元素，就传递该栈地址

```
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
```

13.6 程序执行 map_lookup_elem() 传递了非法的 map_fd

```
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
```

13.7 程序未检查 `map_lookup_elem()` 的返回值是否为空就开始使用

```
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'
```

13.8 程序访问 map 内容时使用了错误的字节对齐

程序虽然检查了 `map_lookup_elem()` 返回值是否为 NULL，但接下来使用了错误的对齐：

```
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
```

13.9 程序在 fallthrough 分支中使用了错误的字节对齐访问 map 数据

程序检查了 `map_lookup_elem()` 返回值是否为 NULL，在 `if` 分支中使用了正确的字节对齐， 但在 fallthrough 分支中使用了错误的对齐：

```
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'
```

13.10 程序执行 `sk_lookup_tcp()`，未检查返回值就直接将其置 NULL

```
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
```

这里的信息提示是 socket reference 未释放，说明 `sk_lookup_tcp()` 返回的是一个非空指针， 直接置空导致这个指针再也无法被解引用。

13.11 程序执行 `sk_lookup_tcp()` 但未检查返回值是否为空

```
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
```

这里的信息提示是 socket reference 未释放，说明 `sk_lookup_tcp()` 返回的是一个非空指针， 直接置空导致这个指针再也无法被解引用。

14 测试（testing）

内核自带了一个 BPF 测试模块，覆盖了 cBPF 和 eBPF 的很多测试场景，能用来测试解释 器和 JIT 编译器。源码见 `lib/test_bpf.c` ，编译是 Kconfig 启用：

```
CONFIG_TEST_BPF=m
```

编译之后用 `insmod` 或 `modprobe` 加载 `test_bpf` 模块。 测试结果带有 `ns` 精度的时间戳日志，打印到内核日志（ `dmesg` 查看）。

15 其他（misc）

Also trinity, the Linux syscall fuzzer, has built-in support for BPF and SECCOMP-BPF kernel fuzzing.

本文作者

The document was written in the hope that it is found useful and in order to give potential BPF hackers or security auditors a better overview of the underlying architecture.

- Jay Schulist jschlst@samba.org

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

« [译] LLVM EBPF 汇编编程（2020）

[译] LINUX 异步 I/O 框架 IO_URING：基本原理、程序示例与性能压测（2020） »