

Introduction to eBPF

2021-03-27 | 术业专攻 | 433 | 0 | 33k | 1:01

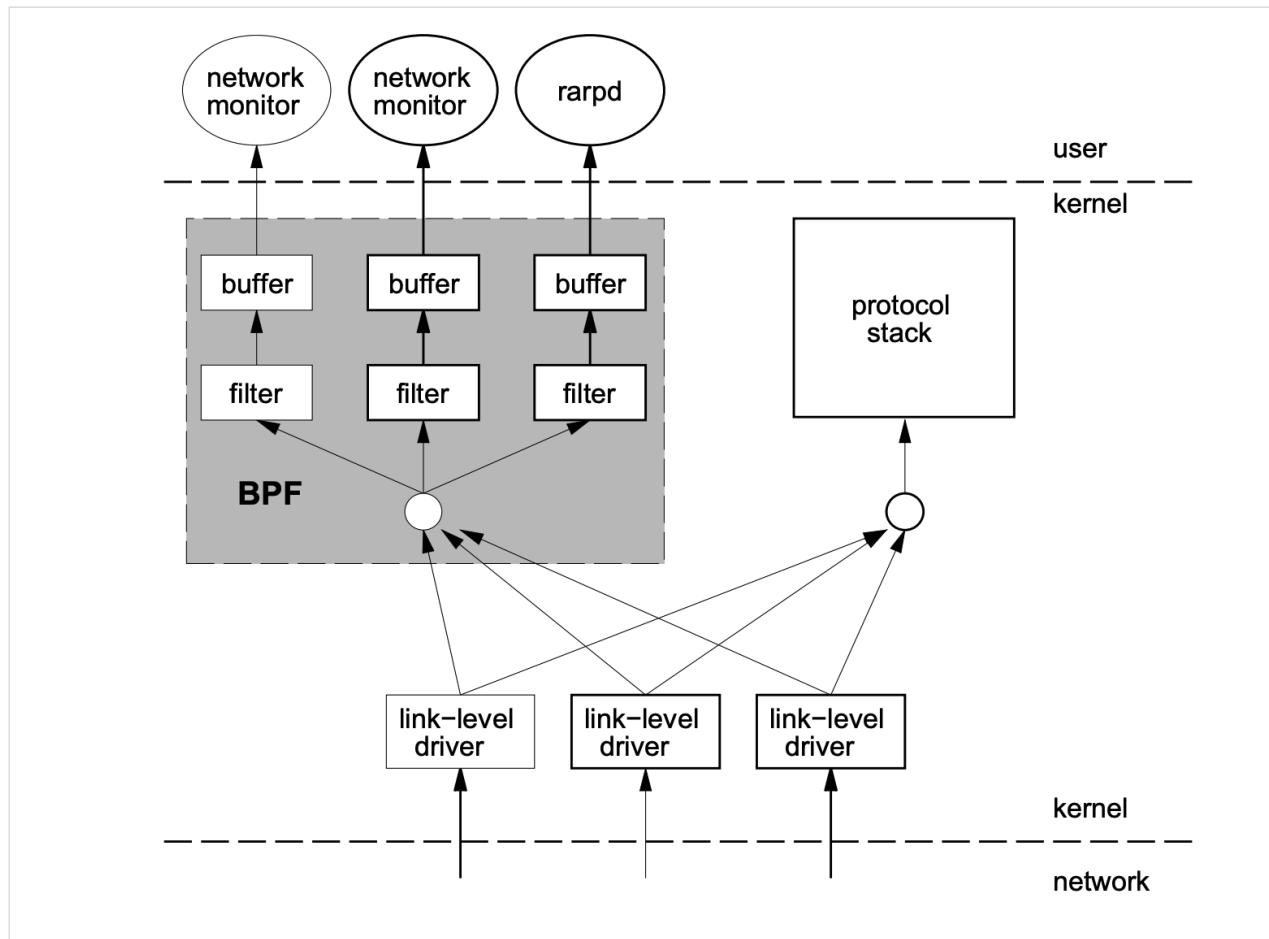
eBPF 源于 BPF，本质上是处于内核中的一个高效与灵活的虚类虚拟机组件，以一种安全的方式在许多内核 hook 点执行字节码。BPF 最初的目的是用于高效网络报文过滤，经过重新设计，eBPF 不再局限于网络协议栈，已经成为内核顶级的子系统，演进为一个通用执行引擎。开发者可基于 eBPF 开发性能分析工具、软件定义网络、安全等诸多场景。本文将介绍 eBPF 的前世今生，并构建一个 eBPF 环境进行开发实践，文中所有的代码可以在我的 Github 中找到。

技术背景

发展历史

BPF，是类 Unix 系统上数据链路层的一种原始接口，提供原始链路层封包的收发。1992 年，Steven McCanne 和 Van Jacobson 写了一篇名为 The BSD Packet Filter: A New Architecture for User-level Packet Capture 的论文。在文中，作者描述了他们如何在 Unix 内核实现网络数据包过滤，这种新的技术比当时最先进的数据包过滤技术快 20 倍。





BPF 在数据包过滤上引入了两大革新:

- 一个新的虚拟机 (VM) 设计, 可以有效地工作在基于寄存器结构的 CPU 之上
- 应用程序使用缓存只复制与过滤数据包相关的数据, 不会复制数据包的所有信息, 这样可以最大程度地减少BPF 处理的数据

由于这些巨大的改进, 所有的 Unix 系统都选择采用 BPF 作为网络数据包过滤技术, 直到今天, 许多 Unix 内核的派生系统中 (包括 Linux 内核) 仍使用该实现。tcpdump 的底层采用 BPF 作为底层包过滤技术, 我们可以在命令后面增加 `-d` 来查看 tcpdump 过滤条件的底层汇编指令。



```

1 $ tcpdump -d 'ip and tcp port 8080'
2 (000) ldh      [12]
3 (001) jeq      #0x800          jt 2    jf 12
4 (002) ldb      [23]
5 (003) jeq      #0x6            jt 4    jf 12
6 (004) ldh      [20]
7 (005) jset     #0x1fff        jt 12   jf 6
8 (006) ldxb    4*([14]&0xf)
9 (007) ldh      [x + 14]
10 (008) jeq     #0x1f90        jt 11   jf 9
11 (009) ldh      [x + 16]
12 (010) jeq     #0x1f90        jt 11   jf 12

```



```
13 (011) ret      #262144
14 (012) ret      #0
```

2014年初，Alexei Starovoitov实现了eBPF（extended Berkeley Packet Filter）。经过重新设计，eBPF演进为一个通用执行引擎，可基于此开发性能分析工具、软件定义网络等诸多场景。**eBPF最早出现在3.18内核中，此后原来的BPF就被称为经典BPF，缩写cBPF（classic BPF）**，cBPF现在已经基本废弃。现在，Linux内核只运行eBPF，内核会将加载的cBPF字节码透明地转换成eBPF再执行。

eBPF与cBPF

eBPF新的设计针对现代硬件进行了优化，所以eBPF生成的指令集比旧的BPF解释器生成的机器码执行得更快。扩展版本也增加了虚拟机中的寄存器数量，将原有的2个32位寄存器增加到10个64位寄存器。由于寄存器数量和宽度的增加，开发人员可以使用函数参数自由交换更多的信息，编写更复杂的程序。总之，这些改进使eBPF版本的速度比原来的BPF提高了4倍。

维度	cBPF	eBPF
内核版本	Linux 2.1.75 (1997年)	Linux 3.18 (2014年) [4.x for kprobe/uprobe/tracepoint/perf-event]
寄存器数目	2个：A, X	10个：R0-R9, 另外R10是一个只读的帧指针 - R0 eBPF中内核函数的返回值和退出值 - R1 - R5 eBPF程序在内核中的参数值 - R6 - R9 内核函数将保存的被调用者callee保存的寄存器 - R10 一个只读的堆栈帧指针
寄存器宽度	32位	64位
存储	16个内存位：M[0-15]	512字节堆栈，无限制大小的map存储
限制的内核调用	非常有限，仅限于JIT特定	有限，通过bpf_call指令调用
目标事件	数据包、seccomp-BPF	数据包、内核函数、用户函数、跟踪点PMCs等

2014年6月，eBPF扩展到用户空间，这也成为了BPF技术的转折点。正如Alexei在提交补丁的注释中写到：「这个补丁展示了eBPF的潜力」。当前，eBPF不再局限于网络栈，已经成为内核顶级的子系统。

eBPF与内核模块



对比 Web 的发展，eBPF 与内核的关系有点类似于 JavaScript 与浏览器内核的关系，eBPF 相比于直接修改内核和编写内核模块提供了一种新的内核可编程的选项。eBPF 程序架构强调安全性和稳定性，看上去更像内核模块，但与内核模块不同，eBPF 程序不需要重新编译内核，并且可以确保 eBPF 程序运行完成，而不会造成系统的崩溃。

维度	Linux 内核模块	eBPF
kprobes/tracepoints	支持	支持
安全性	可能引入安全漏洞或导致内核 Panic	通过验证器进行检查，可以保障内核安全
内核函数	可以调用内核函数	只能通过 BPF Helper 函数调用
编译性	需要编译内核	不需要编译内核，引入头文件即可
运行	基于相同内核运行	基于稳定 ABI 的 BPF 程序可以编译一次，各处运行
与应用程序交互	打印日志或文件	通过 perf_event 或 map 结构
数据结构丰富性	一般	丰富
入门门槛	高	低
升级	需要卸载和加载，可能导致处理流程中断	原子替换升级，不会造成处理流程中断
内核内置	视情况而定	内核内置支持

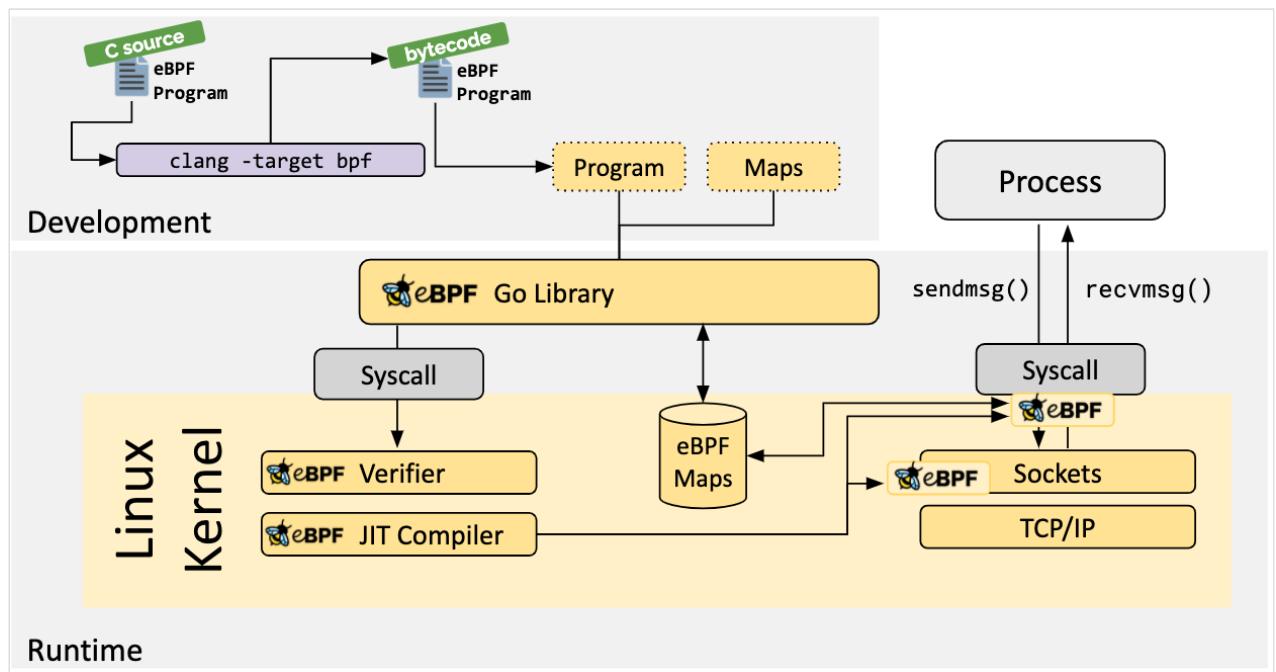
eBPF 架构

eBPF 分为用户空间程序和内核程序两部分：

- 用户空间程序负责加载 BPF 字节码至内核，如需要也会负责读取内核回传的统计信息或者事件详情
- 内核中的 BPF 字节码负责在内核中执行特定事件，如需要也会将执行的结果通过 maps 或者 perf-event 事件发送至用户空间
- 其中用户空间程序与内核 BPF 字节码程序可以使用 map 结构实现双向通信，这为内核中运行的 BPF 字节码程序提供了更加灵活的控制

eBPF 整体结构图如下：





用户空间程序与内核中的 BPF 字节码交互的流程主要如下：

1. 使用 LLVM 或者 GCC 工具将编写的 BPF 代码程序编译成 BPF 字节码
2. 使用加载程序 Loader 将字节码加载至内核
3. 内核使用验证器（Verifier）组件保证执行字节码的安全性，以避免对内核造成灾难，在确认字节码安全后将其加载对应的内核模块执行
4. 内核中运行的 BPF 字节码程序可以使用两种方式将数据回传至用户空间
 - maps 方式可用于将内核中实现的统计摘要信息（比如测量延迟、堆栈信息）等回传至用户空间；
 - perf-event 用于将内核采集的事件实时发送至用户空间，用户空间程序实时读取分析；

eBPF 限制

eBPF 技术虽然强大，但是为了保证内核的处理安全和及时响应，内核中的 eBPF 技术也给予了诸多限制，当然随着技术的发展和演进，限制也在逐步放宽或者提供了对应的解决方案。

- eBPF 程序不能调用任意的内核参数，只限于内核模块中列出的 BPF Helper 函数，函数支持列表也随着内核的演进在不断增加。
- eBPF 程序不允许包含无法到达的指令，防止加载无效代码，延迟程序的终止。
- eBPF 程序中循环次数限制且必须在有限时间内结束，这主要是用来防止在 kprobes 中插入任意的循环，导致锁住整个系统；解决办法包括展开循环，并为需要循环的常见用途添加辅助函数。Linux 5.3 在 BPF 中包含了对有界循环的支持，它有一个可验证的运行时间上限。
- eBPF 堆栈大小被限制在 MAX_BPF_STACK，截止到内核 Linux 5.8 版本，被设置为 512；参见 include/linux/filter.h，这个限制特别是在栈上存储多个字符串缓冲区时：一个char[256]缓冲区会消



耗这个栈的一半。目前没有计划增加这个限制，解决方法是改用 bpf 映射存储，它实际上是无限的。

```
1 /* BPF program can access up to 512 bytes of stack space. */  
2 #define MAX_BPF_STACK 512
```

- eBPF 字节码大小最初被限制为 4096 条指令，截止到内核 Linux 5.8 版本，当前已将放宽至 100 万指令（BPF_COMPLEXITY_LIMIT_INSNs），参见：include/linux/bpf.h，对于无权限的BPF程序，仍然保留4096条限制（BPF_MAXINSNS）；新版本的 eBPF 也支持了多个 eBPF 程序级联调用，虽然传递信息存在某些限制，但是可以通过组合实现更加强大的功能。

```
1 #define BPF_COMPLEXITY_LIMIT_INSNs 1000000 /* yes. 1M insns */
```

eBPF 实战

在深入介绍 eBPF 特性之前，让我们 `Get Hands Dirty`，切切实实的感受 eBPF 程序到底是什么，我们该如何开发 eBPF 程序。随着 eBPF 生态的演进，现在已经有越来越多的工具链用于开发 eBPF 程序，在后文也会详细介绍：

- 基于 bcc 开发：bcc 提供了对 eBPF 开发，前段提供 Python API，后端 eBPF 程序通过 C 实现。特点是简单易用，但是性能较差。
- 基于 libbpf-bootstrap 开发：libbpf-bootstrap 提供了一个方便的脚手架
- 基于内核源码开发：内核源码开发门槛较高，但是也更加切合 eBPF 底层原理，所以这里以这个方法作为示例

内核源码编译

系统环境如下，采用腾讯云 CVM，Ubuntu 20.04，内核版本 5.4.0

```
1 $ uname -a  
2 Linux VM-1-3-ubuntu 5.4.0-42-generic #46-Ubuntu SMP Fri Jul 10 00:24:02 UTC
```

首先安装必要依赖：

```
1 sudo apt install -y bison build-essential cmake flex git libedit-dev pkg-config  
2 python zlib1g-dev libssl-dev libelf-dev libcap-dev libfl-dev llvm clang
```

```
3      gcc-multilib luajit libluajit-5.1-dev libncurses5-dev libclang-dev clang
```

一般情况下推荐采用 apt 方式的安装源码，安装简单而且只安装当前内核的源码，源码的大小在 200M 左右。

```
1 # apt-cache search linux-source
2
3 # apt install linux-source-5.4.0
```

源码安装至 /usr/src/ 目录下。

```
1 $ ls -hl
2 total 4.0K
3 drwxr-xr-x 4 root root 4.0K Nov  9 13:22 linux-source-5.4.0
4 lrwxrwxrwx 1 root root   45 Oct 15 10:28 linux-source-5.4.0.tar.bz2 -> lin
5 $ tar -jxvf linux-source-5.4.0.tar.bz2
6 $ cd linux-source-5.4.0
7
8 $ cp -v /boot/config-$(uname -r) .config # make defconfig 或者 make menucon
9 $ make headers_install
10 $ make modules_prepare
11 $ make scripts      # 可选
12 $ make M=samples/bpf  # 如果配置出错，可以使用 make oldconfig && make prepare
```

编译成功后，可以在 samples/bpf 目录下看到一系列的目标文件和二进制文件。

Hello World

前面说到 eBPF 通常由内核空间程序和用户空间程序两部分组成，现在 samples/bpf 目录下有很多这种程序，内核空间程序以 _kern.c 结尾，用户空间程序以 _user.c 结尾。先不看这些复杂的程序，我们手动写一个 eBPF 程序的 Hello World。

内核中的程序 hello_kern.c：

```
hello_kern.c

1 #include <linux/bpf.h>
2 #include "bpf_helpers.h"
3
4 #define SEC(NAME) __attribute__((section(NAME), used))
5
6 SEC("tracepoint/syscalls/sys_enter_execve")
```

```
7 int bpf_prog(void *ctx)
8 {
9     char msg[] = "Hello BPF from houmin!\n";
10    bpf_trace_printk(msg, sizeof(msg));
11    return 0;
12 }
13
14 char _license[] SEC("license") = "GPL";
```

函数入口

上述代码和普通的C语言编程有一些区别。

- 程序的入口通过编译器的 `pragama __section("tracepoint/syscalls/sys_enter_execve")` 指定的。
- 入口的参数不再是 `argc, argv`, 它根据不同的 prog type 而有所差别。我们的例子中, prog type 是 `BPF_PROG_TYPE_TRACEPOINT`, 它的入口参数就是 `void *ctx`。

头文件

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

这个头文件的来源是kernel source header file。它安装在 `/usr/include/linux/bpf.h` 中。

它提供了bpf 编程需要的很多symbol。例如

- enum bpf_func_id 定义了所有的kerne helper function 的id
- enum bpf_prog_type 定义了内核支持的所有的prog 的类型。
- struct __sk_buff 是bpf 代码中访问内核struct sk_buff的接口。

等等

```
#include "bpf_helpers.h"
```

来自libbpf , 需要自行安装。我们引用这个头文件是因为调用了`bpf_printk()`。这是一个kernel helper function。

程序解释

这里我们简单解读下内核态的 `ebpf` 程序, 非常简单:

- `bpf_trace_printk` 是一个 eBPF helper 函数, 用于打印信息到 `trace_pipe` (`/sys/kernel/debug/tracing/trace_pipe`), 详见[这里](#)

- 代码声明了 `SEC` 宏，并且定义了 GPL 的 License，这是因为加载进内核的 eBPF 程序需要有 License 检查，类似于内核模块

加载 BPF 代码

用户态程序 `hello_user.c`

```
hello_user.c

1 #include <stdio.h>
2 #include "bpf_load.h"
3
4 int main(int argc, char **argv)
5 {
6     if(load_bpf_file("hello_kern.o") != 0)
7     {
8         printf("The kernel didn't load BPF program\n");
9         return -1;
10    }
11
12    read_trace_pipe();
13    return 0;
14 }
```

在用户态 `bpf` 程序中，解读如下：

- 通过 `load_bpf_file` 将编译出的内核态 bpf 目标文件加载到内核
- 通过 `read_trace_pipe` 从 `trace_pipe` 读取 trace 信息，打印到控制台中

修改 `samples/bpf` 目录下的 `Makefile` 文件，在对应的位置添加以下三行：

```
hostprogs-y += hello
hello-objs := bpf_load.o hello_user.o
always += hello_kern.o
```

重新编译，可以看到编译成功的文件

```
$ make M=samples/bpf
$ ls -hl samples/bpf/hello*
-rwxrwxr-x 1 ubuntu ubuntu 404K Mar 30 17:48 samples/bpf/hello
-rw-rw-r-- 1 ubuntu ubuntu 317 Mar 30 17:47 samples/bpf/hello_kern.c
```

```
5 -rw-rw-r-- 1 ubuntu ubuntu 3.8K Mar 30 17:48 samples/bpf/hello_kern.o
6 -rw-rw-r-- 1 ubuntu ubuntu 246 Mar 30 17:47 samples/bpf/hello_user.c
7 -rw-rw-r-- 1 ubuntu ubuntu 2.2K Mar 30 17:48 samples/bpf/hello_user.o
```

进入到对应的目录运行 hello 程序，可以看到输出结果如下：

```
$ sudo ./hello
<...>-102735 [001] .... 6733.481740: 0: Hello BPF from houmin!
<...>-102736 [000] .... 6733.482884: 0: Hello BPF from houmin!
<...>-102737 [002] .... 6733.483074: 0: Hello BPF from houmin!
```

代码解读

前面提到 `load_bpf_file` 函数将 LLVM 编译出来的 eBPF 字节码加载进内核，这到底是如何实现的呢？

- 经过搜查，可以看到 `load_bpf_file` 也是在 `samples/bpf` 目录下实现的，具体的参见 `bpf_load.c`
- 阅读 `load_bpf_file` 代码可以看到，它主要是解析 ELF 格式的 eBPF 字节码，然后调用 `load_and_attach` 函数
- 在 `load_and_attach` 函数中，我们可以看到其调用了 `bpf_load_program` 函数，这是 libbpf 提供的函数。
- 调用的 `bpf_load_program` 中的 `license`、`kern_version` 等参数来自于解析 eBPF ELF 文件，`prog_type` 来自于 bpf 代码里面 SEC 字段指定的类型。

```
1 static int load_and_attach(const char *event, struct bpf_insn *prog, int s
2 {
3     bool is_socket = strncmp(event, "socket", 6) == 0;
4     bool is_kprobe = strncmp(event, "kprobe/", 7) == 0;
5     bool is_kretprobe = strncmp(event, "kretprobe/", 10) == 0;
6     bool is_tracepoint = strncmp(event, "tracepoint/", 11) == 0;
7     bool is_raw_tracepoint = strncmp(event, "raw_tracepoint/", 15) == 0;
8     bool is_xdp = strncmp(event, "xdp", 3) == 0;
9     bool is_perf_event = strncmp(event, "perf_event", 10) == 0;
10    bool is_cgroup_skb = strncmp(event, "cgroup/skb", 10) == 0;
11    bool is_cgroup_sk = strncmp(event, "cgroup/sock", 11) == 0;
12    bool is_sockops = strncmp(event, "sockops", 7) == 0;
```

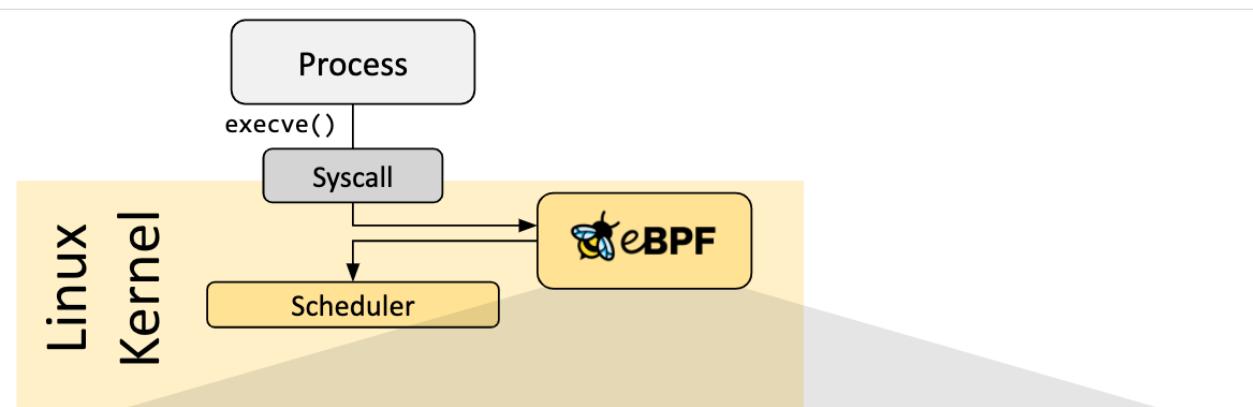
```

13     bool is_sk_skb = strncmp(event, "sk_skb", 6) == 0;
14     bool is_sk_msg = strncmp(event, "sk_msg", 6) == 0;
15
16 //...
17
18     fd = bpf_load_program(prog_type, prog, insns_cnt, license, kern_ve
19                         bpf_log_buf, BPF_LOG_BUF_SIZE);
20     if (fd < 0) {
21         printf("bpf_load_program() err=%d\n%s", errno, bpf_log_buf
22         return -1;
23     }
24 //...
25 }
```

eBPF 特性

Hook Overview

eBPF 程序都是事件驱动的，它们会在内核或者应用程序经过某个确定的 Hook 点的时候运行，这些 Hook 点都是提前定义的，包括系统调用、函数进入/退出、内核 tracepoints、网络事件等。



```

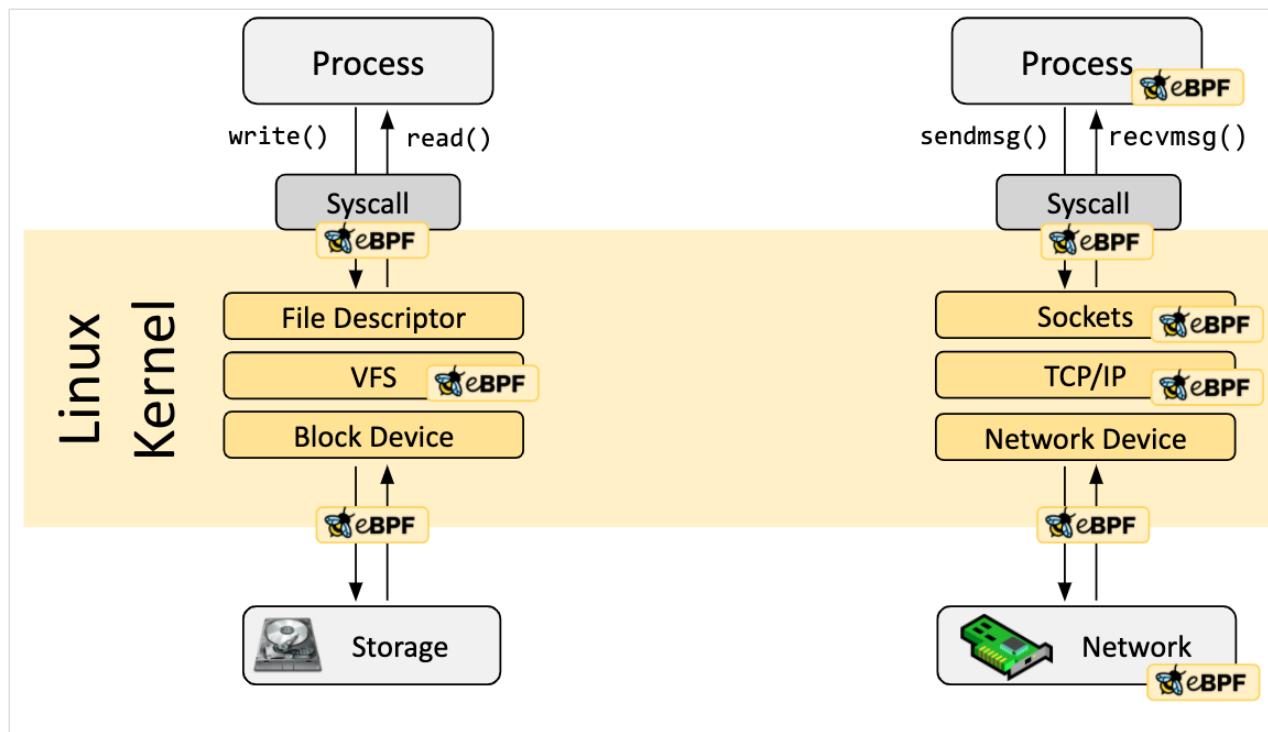
int syscall__ret_execve(struct pt_regs *ctx)
{
    struct comm_event event = {
        .pid = bpf_get_current_pid_tgid() >> 32,
        .type = TYPE_RETURN,
    };

    bpf_get_current_comm(&event.comm, sizeof(event.comm));
    comm_events.perf_submit(ctx, &event, sizeof(event));

    return 0;
}
```



如果针对某个特定需求的 Hook 点不存在，可以通过 `kprobe` 或者 `uprobe` 来在内核或者用户程序的几乎所有地方挂载 eBPF 程序。



Verification

With great power there must also come great responsibility.

每一个 eBPF 程序加载到内核都要经过 `Verification`，用来保证 eBPF 程序的安全性，主要包括：

- 要保证 加载 eBPF 程序的进程有足够的特权级，除非节点开启了 `unprivileged` 特性，只有特权级的程序才能够加载 eBPF 程序
 - 内核提供了一个配置项 `/proc/sys/kernel/unprivileged_bpf_disabled` 来禁止非特权用户使用 `bpf(2)` 系统调用，可以通过 `sysctl` 命令修改
 - 比较特殊的一点是，这个配置项特意设计为**一次性开关** (one-time kill switch)，这意味着一旦将它设为 `1`，就没有办法再改为 `0` 了，除非重启内核
 - 一旦设置为 `1` 之后，只有初始命名空间中有 `CAP_SYS_ADMIN` 特权的进程才可以调用 `bpf(2)` 系统调用。Cilium 启动后也会将这个配置项设为 `1`:

```
1 $ echo 1 > /proc/sys/kernel/unprivileged_bpf_disabled
```

- 要保证 eBPF 程序不会崩溃或者使得系统出故障
- 要保证 eBPF 程序不能陷入死循环，能够 `runs to completion`

- 要保证 eBPF 程序必须满足系统要求的大小，过大的 eBPF 程序不允许被加载进内核
- 要保证 eBPF 程序的复杂度有限，Verifier 将会评估 eBPF 程序所有可能的执行路径，必须能够在有限时间内完成 eBPF 程序复杂度分析

JIT Compilation

Just-In-Time (JIT) 编译用来将通用的 eBPF 字节码翻译成与机器相关的指令集，从而极大加速 BPF 程序的执行：

- 与解释器相比，它们可以降低每个指令的开销。通常，指令可以 1:1 映射到底层架构的原生指令
- 这也会减少生成的可执行镜像的大小，因此对 CPU 的指令缓存更友好
- 特别地，对于 CISC 指令集（例如 x86），JIT 做了很多特殊优化，目的是为给定的指令产生可能的最短操作码，以降低程序翻译过程所需的空间

64 位的 x86_64、arm64、ppc64、s390x、mips64、sparc64 和 32 位的 arm、x86_32 架构都内置了 in-kernel eBPF JIT 编译器，它们的功能都是一样的，可以用如下方式打开：



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

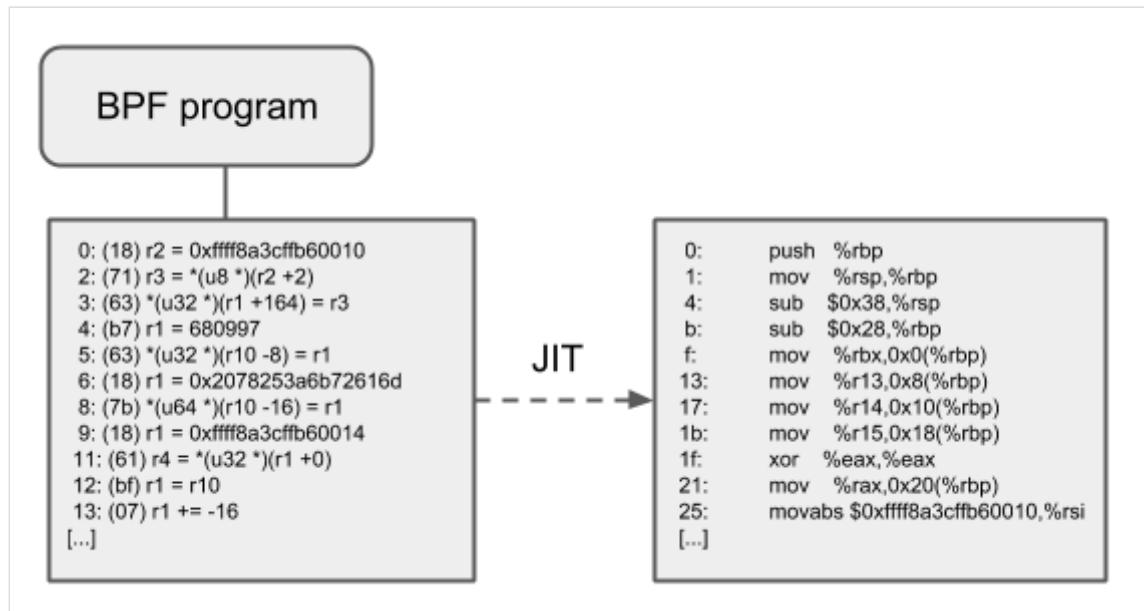
32 位的 mips、ppc 和 sparc 架构目前内置的是一个 cBPF JIT 编译器。这些只有 cBPF JIT 编译器的架构，以及那些甚至完全没有 BPF JIT 编译器的架构，需要通过内核中的解释器（in-kernel interpreter）执行 eBPF 程序。

要判断哪些平台支持 eBPF JIT，可以在内核源文件中 grep HAVE_EBPF_JIT：



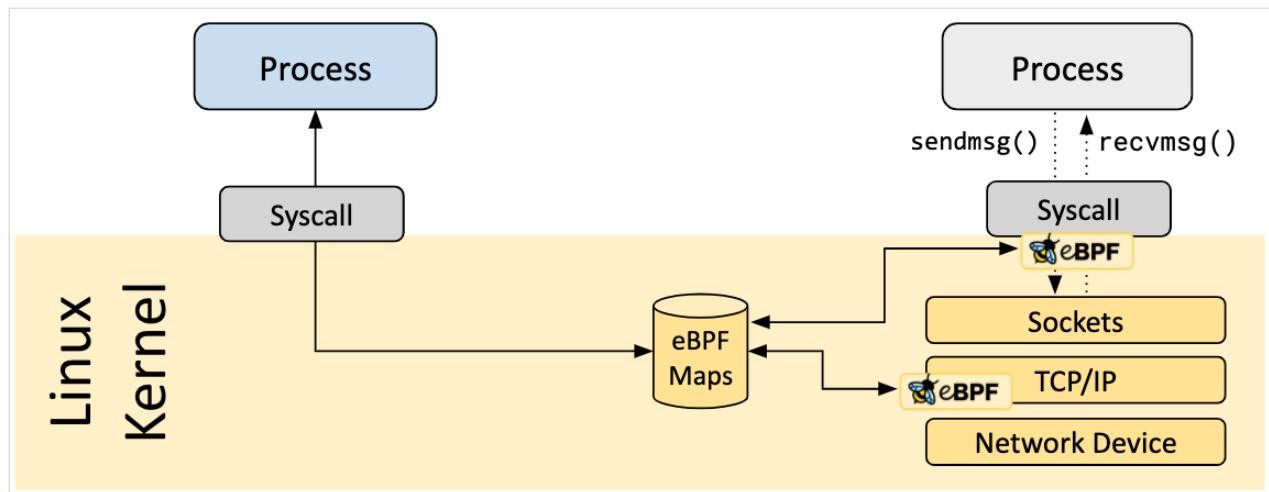
```
1 $ git grep HAVE_EBPF_JIT arch/
2 arch/arm/Kconfig:      select HAVE_EBPF_JIT    if !CPU_ENDIAN_BE32
3 arch/arm64/Kconfig:     select HAVE_EBPF_JIT
4 arch/powerpc/Kconfig:   select HAVE_EBPF_JIT    if PPC64
5 arch/mips/Kconfig:      select HAVE_EBPF_JIT    if (64BIT && !CPU_MICROMIPS)
6 arch/s390/Kconfig:      select HAVE_EBPF_JIT    if PACK_STACK && HAVE_MARCH_
7 arch/sparc/Kconfig:     select HAVE_EBPF_JIT    if SPARC64
8 arch/x86/Kconfig:       select HAVE_EBPF_JIT    if X86_64
```





Maps

BPF Map 是驻留在内核空间中的高效 Key/Value store，包含多种类型的 Map，由内核实现其功能，具体实现可以参考 我的这篇博文。

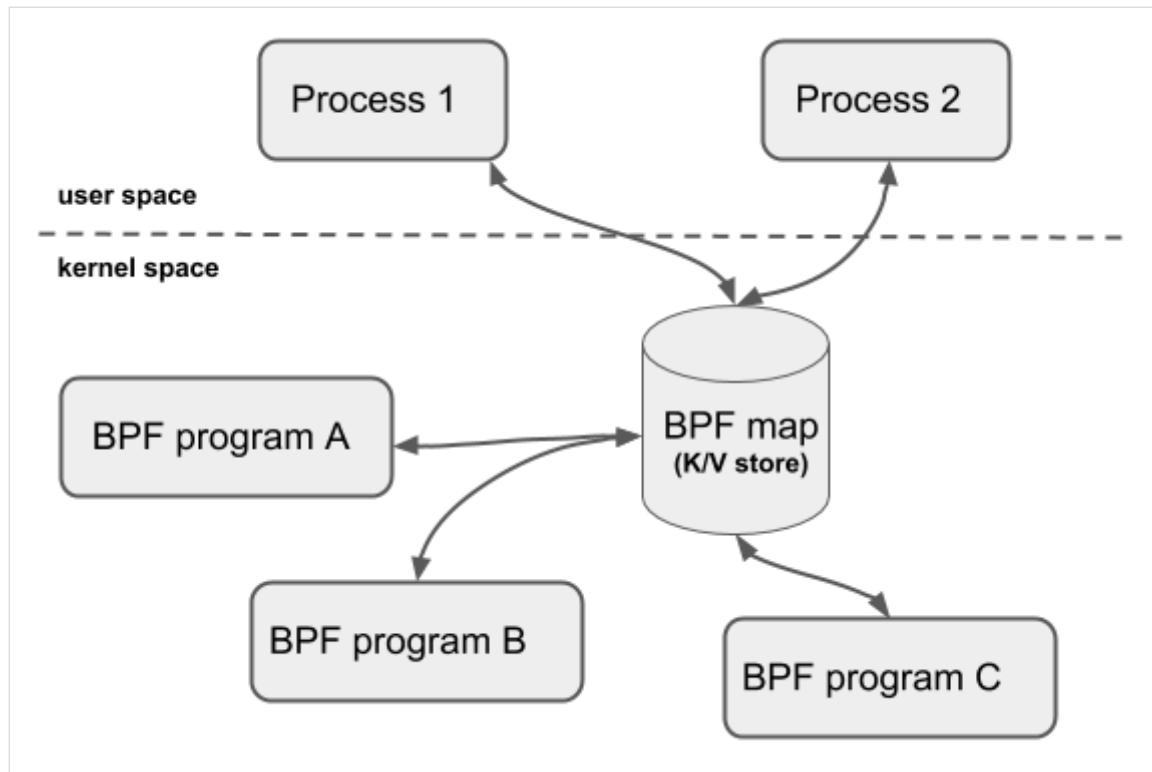


BPF Map 的交互场景有以下几种：

- BPF 程序和用户态程序的交互：BPF 程序运行完，得到的结果存储到 map 中，供用户态程序通过文件描述符访问
- BPF 程序和内核态程序的交互：和 BPF 程序以外的内核程序交互，也可以使用 map 作为中介
- BPF 程序间交互：如果 BPF 程序内部需要用全局变量来交互，但是由于安全原因 BPF 程序不允许访问全局变量，可以使用 map 来充当全局变量
- BPF Tail call：Tail call 是一个 BPF 程序跳转到另一 BPF 程序，BPF 程序首先通过 `BPF_MAP_TYPE_PROG_ARRAY` 类型的 map 来知道另一个 BPF 程序的指针，然后调用 `tail_call()` 的 helper function 来执行 Tail call



共享 map 的 BPF 程序不要求是相同的程序类型，例如 tracing 程序可以和网络程序共享 map，单个 BPF 程序目前最多可直接访问 64 个不同 map。



当前可用的 **通用 map** 有：

- `BPF_MAP_TYPE_HASH`
- `BPF_MAP_TYPE_ARRAY`
- `BPF_MAP_TYPE_PERCPU_HASH`
- `BPF_MAP_TYPE_PERCPU_ARRAY`
- `BPF_MAP_TYPE_LRU_HASH`
- `BPF_MAP_TYPE_LRU_PERCPU_HASH`
- `BPF_MAP_TYPE_LPM_TRIE`

以上 map 都使用相同的一组 BPF 辅助函数来执行查找、更新或删除操作，但各自实现了不同的后端，这些后端各有不同的语义和性能特点。随着多CPU架构的成熟发展，BPF Map也引入了 **per-cpu** 类型，如 `BPF_MAP_TYPE_PERCPU_HASH`、`BPF_MAP_TYPE_PERCPU_ARRAY` 等，当你使用这种类型的BPF Map时，每个CPU都会存储并看到它自己的Map数据，从属于不同CPU之间的数据是互相隔离的，这样做的好处是，在进行查找和聚合操作时更加高效，性能更好，尤其是你的BPF程序主要是在做收集时间序列型数据，如流量数据或指标等。

当前内核中的 **非通用 map** 有：

- `BPF_MAP_TYPE_PROG_ARRAY`：一个数组 map，用于 hold 其他的 BPF 程序
- `BPF_MAP_TYPE_PERF_EVENT_ARRAY`
- `BPF_MAP_TYPE_CGROUP_ARRAY`：用于检查skb中的cgroup2成员信息



- `BPF_MAP_TYPE_STACK_TRACE`：用于存储栈跟踪的MAP
- `BPF_MAP_TYPE_ARRAY_OF_MAPS`：持有（hold）其他 map 的指针，这样整个 map 就可以在运行时实现原子替换
- `BPF_MAP_TYPE_HASH_OF_MAPS`：持有（hold）其他 map 的指针，这样整个 map 就可以在运行时实现原子替换

Helper Calls

eBPF 程序不能够随意调用内核函数，如果这么做的话会导致 eBPF 程序与特定的内核版本绑定，相反它内核定义的一系列 Helper functions。Helper functions 使得 BPF 能够通过一组内核定义的稳定的函数调用来从内核中查询数据，或者将数据推送到内核。所有的 BPF 辅助函数都是核心内核的一部分，无法通过内核模块来扩展或添加。当前可用的 BPF 辅助函数已经有几十个，并且数量还在不断增加，你可以在 Linux Manual Page: bpf-helpers 看到当前 Linux 支持的 Helper functions。



不同类型的 BPF 程序能够使用的辅助函数可能是不同的，例如：

- 与 attach 到 tc 层的 BPF 程序相比，attach 到 socket 的 BPF 程序只能够调用前者可以调用的辅助函数的一个子集
- `lightweight tunneling` 使用的封装和解封装辅助函数，只能被更低的 tc 层使用；而推送通知到用户态所使用的事件输出辅助函数，既可以被 tc 程序使用也可以被 XDP 程序使用

所有的辅助函数都共享同一个通用的、和系统调用类似的函数方法，其定义如下：

```
1 u64 fn(u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)
```

内核将辅助函数抽象成 `BPF_CALL_0()` 到 `BPF_CALL_5()` 几个宏，形式和相应类型的系统调用类似，这里宏的定义可以参见 `include/linux/filter.h`。以 `bpf_map_update_elem` 为例，可以看到它通过调用相应 map 的回调函数完成更新 map 元素的操作：

/kernel/bpf/helpers.c

```
1 BPF_CALL_4(bpf_map_update_elem, struct bpf_map *, map, void *, key,
2             void *, value, u64, flags)
3 {
4     WARN_ON_ONCE(!rcu_read_lock_held());
5     return map->ops->map_update_elem(map, key, value, flags);
6 }
7
8 const struct bpf_func_proto bpf_map_update_elem_proto = {
9     .func          = bpf_map_update_elem,
10    .gpl_only      = false,
11    .ret_type       = RET_INTEGER,
12    .arg1_type     = ARG_CONST_MAP_PTR,
13    .arg2_type     = ARG_PTR_TO_MAP_KEY,
14    .arg3_type     = ARG_PTR_TO_MAP_VALUE,
15    .arg4_type     = ARG_ANYTHING,
16};
```

这种方式有很多优点：

虽然 cBPF 允许其加载指令（load instructions）进行超出范围的访问（overload），以便从一个看似不可能的包偏移量（packet offset）获取数据以唤醒多功能辅助函数，但每个 cBPF JIT 仍然需要为这个 cBPF 扩展实现对应的支持。而在 eBPF 中，JIT 编译器会以一种透明和高效的方式编译新加入的辅助函数，这意味着 JIT 编译器只需要发射（emit）一条调用指令（call instruction），因为寄存器映射的方式使得 BPF 排列参数的方式（assignments）已经和底层架构的调用约定相匹配了。这使得基于辅助函数扩展核心内核（core kernel）非常方便。**所有的 BPF 辅助函数都是核心内核的一部分，无法通过内核模块（kernel module）来扩展或添加。**

前面提到的函数签名还允许校验器执行类型检测（type check）。上面的 `struct bpf_func_proto` 用于存放校验器必需知道的所有关于该辅助函数的信息，这样校验器可以确保辅助函数期望的类型和 BPF 程序寄存器中的当前内容是匹配的。

参数类型范围很广，从任意类型的值，到限制只能为特定类型，例如 BPF 栈缓冲区（stack buffer）的 `pointer/size` 参数对，辅助函数可以从这个位置读取数据或向其写入数据。对于这种情况，校验器还可以执行额外的检查，例如，缓冲区是否已经初始化过了。

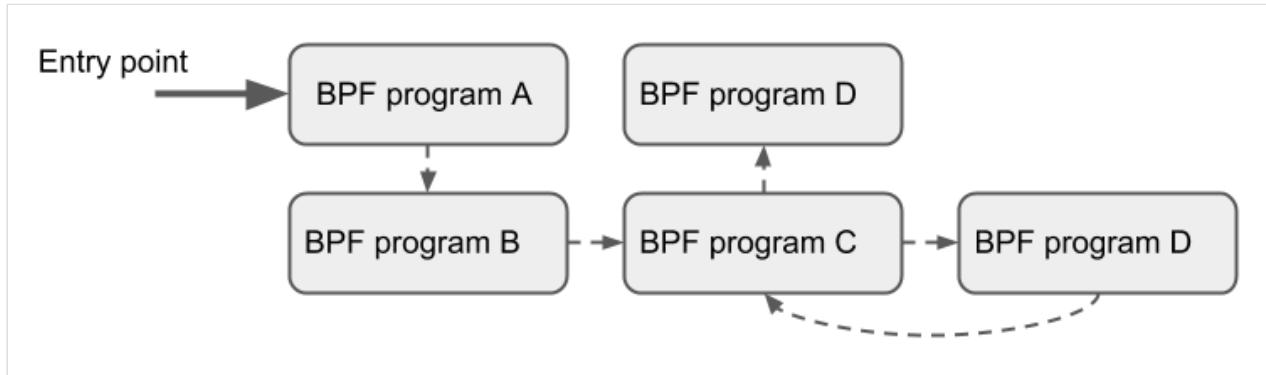
Tail Calls

尾调用的机制是指：一个 BPF 程序可以调用另一个 BPF 程序，并且调用完成后不用返回到原来的程序。

- 和普通函数调用相比，这种调用方式开销最小，因为它是用长跳转（long jump）实现的，复用了原来的栈帧（stack frame）



- BPF 程序都是独立验证的，因此要传递状态，要么使用 per-CPU map 作为 scratch 缓冲区，要么如果是 tc 程序的话，还可以使用 `skb` 的某些字段（例如 `cb[]`）
- 相同类型的程序才可以尾调用，而且它们还要与 JIT 编译器相匹配，因此要么是 JIT 编译执行，要么是解释器执行（invoke interpreted programs），但不能同时使用两种方式



BPF to BPF Calls

除了 BPF 辅助函数和 BPF 尾调用之外，BPF 核心基础设施最近刚加入了一个新特性：`BPF to BPF calls`。在这个特性引入内核之前，典型的 BPF C 程序必须 将所有需要复用的代码进行特殊处理，例如，在头文件中声明为 `always_inline`。当 LLVM 编译和生成 BPF 对象文件时，所有这些函数将被内联，因此会在生成的对象文件中重 复多次，导致代码尺寸膨胀：



```

1 #include <linux/bpf.h>
2
3 #ifndef __section
4 # define __section(NAME) \
5     __attribute__((section(NAME), used))
6 #endif
7
8 #ifndef __inline
9 # define __inline \
10     inline __attribute__((always_inline))
11 #endif
12
13 static __inline int foo(void)
14 {
15     return XDP_DROP;
16 }
17
18 __section("prog")
19 int xdp_drop(struct xdp_md *ctx)
20 {
21     return foo();
22 }
  
```



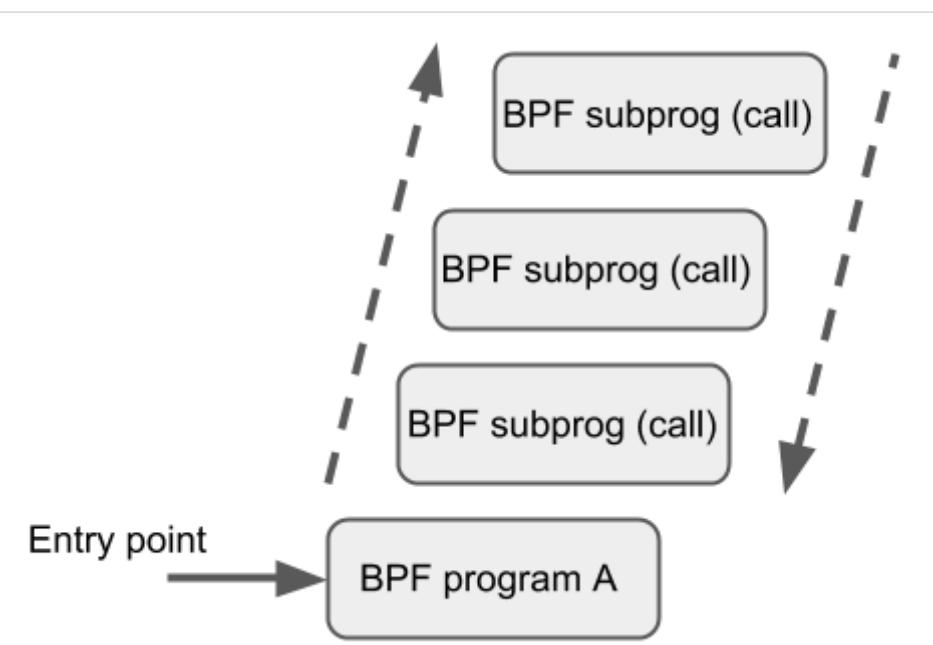
23

24 char __license[] __section("license") = "GPL";

之所以要这样做是因为 BPF 程序的加载器、校验器、解释器和 JIT 中都缺少对函数调用的支持。从 Linux 4.16 和 LLVM 6.0 开始，这个限制得到了解决，BPF 程序不再需要到处使用 `always_inline` 声明了。因此，上面的代码可以更自然地重写为：

```
1 #include <linux/bpf.h>
2
3 #ifndef __section
4 # define __section(NAME) \
5     __attribute__((section(NAME), used))
6 #endif
7
8 static int foo(void)
9 {
10     return XDP_DROP;
11 }
12
13 __section("prog")
14 int xdp_drop(struct xdp_md *ctx)
15 {
16     return foo();
17 }
18
19 char __license[] __section("license") = "GPL";
```

BPF 到 BPF 调用是一个重要的性能优化，极大减小了生成的 BPF 代码大小，因此 对 CPU 指令缓存 (instruction cache, i-cache) 更友好。



BPF 辅助函数的调用约定也适用于 BPF 函数间调用：

- `r1 - r5` 用于传递参数，返回结果放到 `r0`
- `r1 - r5` 是 scratch registers，`r6 - r9` 像往常一样是保留寄存器
- 最大嵌套调用深度是 `8`
- 调用方可以传递指针（例如，指向调用方的栈帧的指针）给被调用方，但反过来不行

当前，BPF 函数间调用和 BPF 尾调用是不兼容的，因为后者需要复用当前的栈设置（stack setup），而前者会增加一个额外的栈帧，因此不符合尾调用期望的布局。

BPF JIT 编译器为每个函数体发射独立的镜像（emit separate images for each function body），稍后在最后一通 JIT 处理（final JIT pass）中再修改镜像中函数调用的地址。已经证明，这种方式需要对各种 JIT 做最少的修改，因为在实现中它们可以将 BPF 函数间调用当做常规的 BPF 辅助函数调用。

Object Pinning

BPF map 和程序作为内核资源只能通过文件描述符访问，其背后是内核中的匿名 inode。这带来了很多优点：

- 用户空间应用程序能够使用大部分文件描述符相关的 API
- 传递给 Unix socket 的文件描述符是透明工作等等

但同时，文件描述符受限于进程的生命周期，使得 map 共享之类的操作非常笨重，这给某些特定的场景带来了很多复杂性。

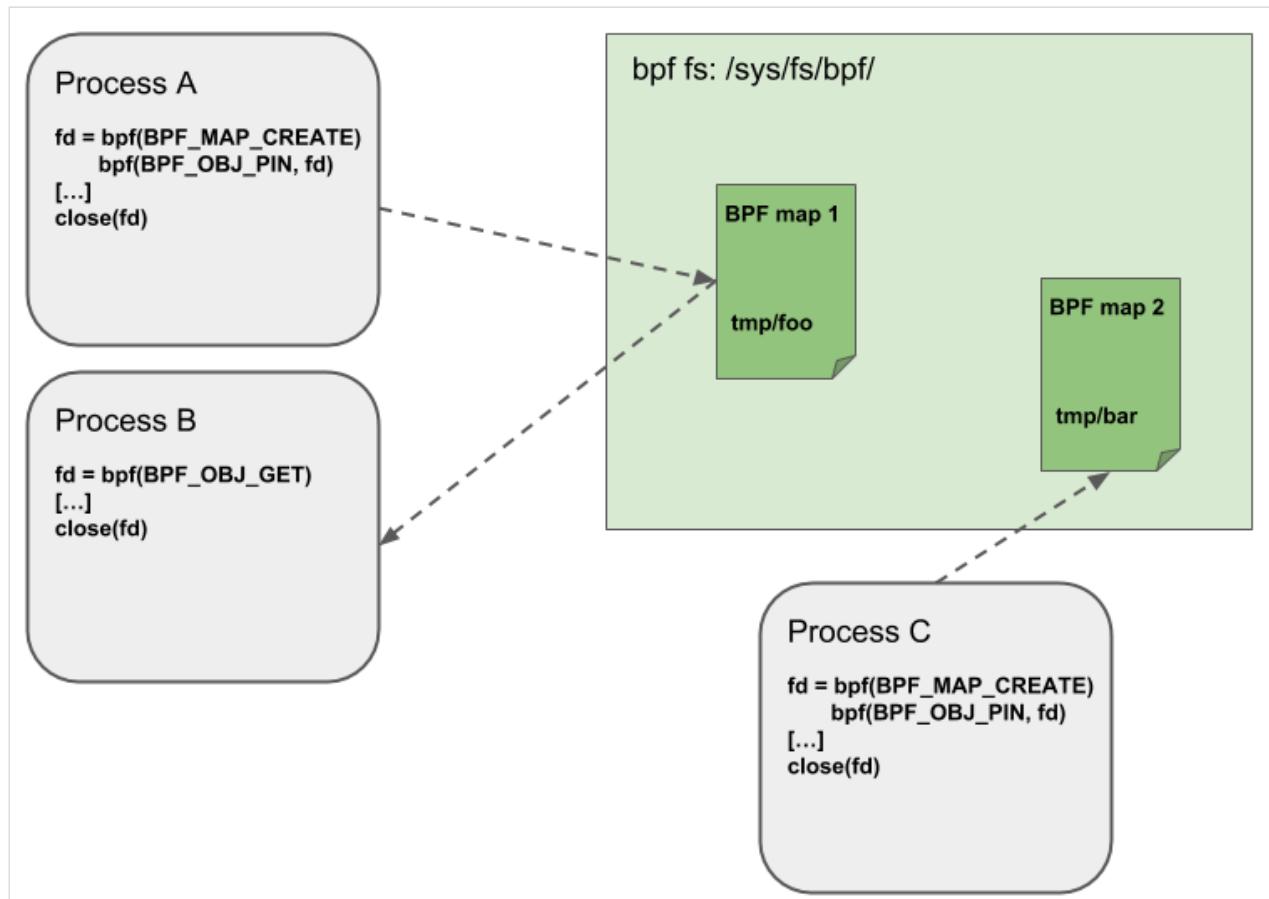
例如 iproute2，其中的 tc 或 XDP 在准备环境、加载程序到内核之后最终会退出。在这种情况下，从用户空间也无法访问这些 map 了，而本来这些 map 其实是很有用的。例如，在 data path 的 ingress 和 egress 位置共享的 map（可以统计包数、字节数、PPS 等信息）。另外，第三方应用可能希望在 BPF 程序运行时监控或更新 map。

为了解决这个问题，内核实现了一个最小内核空间 BPF 文件系统，BPF map 和 BPF 程序都可以 pin 到这个文件系统内，这个过程称为 object pinning。BPF 相关的文件系统不是单例模式（singleton），它支持多挂载实例、硬链接、软连接等等。

相应的，BPF 系统调用扩展了两个新命令，如下图所示：

- `BPF_OBJ_PIN`：钉住一个对象
- `BPF_OBJ_GET`：获取一个被钉住的对象





Hardening

Protection Execution Protection

为了避免代码被损坏，BPF 会在程序的生命周期内，在内核中将 BPF 解释器解释后的整个镜像 (`struct bpf_prog`) 和 JIT 编译之后的镜像 (`struct bpf_binary_header`) 锁定为只读的。在这些位置发生的任何数据损坏（例如由于某些内核 bug 导致的）会触发通用的保护机制，因此会造成内核崩溃而不是允许损坏静默地发生。

查看哪些平台支持将镜像内存（image memory）设置为只读的，可以通过下面的搜索：



```

1 $ git grep ARCH_HAS_SET_MEMORY | grep select
2 arch/arm/Kconfig:      select ARCH_HAS_SET_MEMORY
3 arch/arm64/Kconfig:    select ARCH_HAS_SET_MEMORY
4 arch/s390/Kconfig:     select ARCH_HAS_SET_MEMORY
5 arch/x86/Kconfig:      select ARCH_HAS_SET_MEMORY
  
```

`CONFIG_ARCH_HAS_SET_MEMORY` 选项是不可配置的，因此平台要么内置支持，要么不支持，那些目前还不支持的架构未来可能也会支持。

Mitigation Against Spectre



为了防御 Spectre v2) 攻击, Linux 内核提供了 `CONFIG_BPF_JIT_ALWAYS_ON` 选项, 打开这个开关后 BPF 解释器将会从内核中完全移除, 永远启用 JIT 编译器:

- 如果应用在一个基于虚拟机的环境, 客户机内核将不会复用内核的 BPF 解释器, 因此可以避免某些相关的攻击
- 如果是基于容器的环境, 这个配置是可选的, 如果 JIT 功能打开了, 解释器仍然可能会在编译时被去掉, 以降低内核的复杂度
- 对于主流架构 (例如 `x86_64` 和 `arm64`) 上的 JIT 通常都建议打开这个开关

将 `/proc/sys/net/core/bpf_jit_harden` 设置为 `1` 会为非特权用户的 JIT 编译做一些额外的加固工作。这些额外加固会稍微降低程序的性能, 但在有非受信用户在系统上进行操作的情况下, 能够有效地减小潜在的受攻击面。但与完全切换到解释器相比, 这些性能损失还是比较小的。对于 `x86_64` JIT 编译器, 如果设置了 `CONFIG_RETPOLINE`, 尾调用的间接跳转 (indirect jump) 就会用 `retpoline` 实现。写作本文时, 在大部分现代 Linux 发行版上这个配置都是打开的。

Constant Blinding

当前, 启用加固会在 JIT 编译时盲化 (blind) BPF 程序中用户提供的所有 32 位和 64 位常量, 以防御 **JIT spraying 攻击**, 这些攻击会将原生操作码作为立即数注入到内核。这种攻击有效是因为: **立即数驻留在可执行内核内存 (executable kernel memory) 中**, 因此某些内核 bug 可能会触发一个跳转动作, 如果跳转到立即数的开始位置, 就会把它们当做原生指令开始执行。

盲化 JIT 常量通过对真实指令进行随机化 (randomizing the actual instruction) 实现。在这种方式中, 通过对指令进行重写, 将原来**基于立即数的操作转换成基于寄存器的操作**。指令重写将加载值的过程分解为两部分:

1. 加载一个盲化后的 (blinded) 立即数 `rnd ^ imm` 到寄存器
2. 将寄存器和 `rnd` 进行异或操作 (`xor`)

这样原始的 `imm` 立即数就驻留在寄存器中, 可以用于真实的操作了。这里介绍的只是加载操作的盲化过程, 实际上所有的通用操作都被盲化了。下面是加固关闭的情况下, 某个程序的 JIT 编译结果:



```
1 $ echo 0 > /proc/sys/net/core/bpf_jit_harden
2
3 ffffffa034f5e9 + <x>:
4 [...]
5 39:    mov      $0xa8909090,%eax
6 3e:    mov      $0xa8909090,%eax
7 43:    mov      $0xa8ff3148,%eax
8 48:    mov      $0xa89081b4,%eax
9 4d:    mov      $0xa8900bb0,%eax
10 52:   mov      $0xa810e0c1,%eax
11 57:   mov      $0xa8908eb4,%eax
```



```
12 5c:    mov    $0xa89020b0,%eax
13 [...]
```

加固打开之后，以上程序被某个非特权用户通过 BPF 加载的结果（这里已经进行了常量盲化）：

```
1 $ echo 1 > /proc/sys/net/core/bpf_jit_harden
2
3 ffffffff034f1e5 + <x>:
4 [...]
5 39:    mov    $0xe1192563,%r10d
6 3f:    xor    $0x4989b5f3,%r10d
7 46:    mov    %r10d,%eax
8 49:    mov    $0xb8296d93,%r10d
9 4f:    xor    $0x10b9fd03,%r10d
10 56:   mov    %r10d,%eax
11 59:   mov    $0x8c381146,%r10d
12 5f:   xor    $0x24c7200e,%r10d
13 66:   mov    %r10d,%eax
14 69:   mov    $0xeb2a830e,%r10d
15 6f:   xor    $0x43ba02ba,%r10d
16 76:   mov    %r10d,%eax
17 79:   mov    $0xd9730af,%r10d
18 7f:   xor    $0xa5073b1f,%r10d
19 86:   mov    %r10d,%eax
20 89:   mov    $0x9a45662b,%r10d
21 8f:   xor    $0x325586ea,%r10d
22 96:   mov    %r10d,%eax
23 [...]
```

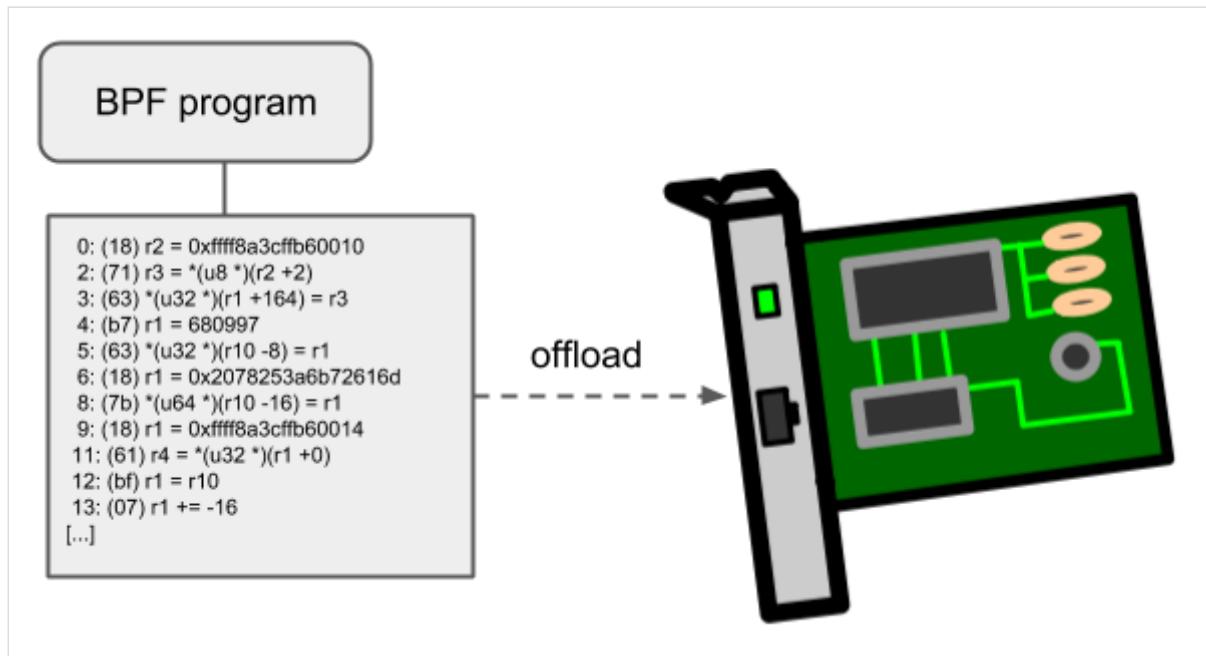
两个程序在语义上是一样的，但在第二种方式中，原来的立即数在反汇编之后的程序中不再可见。同时，加固还会禁止任何 JIT 内核符合 (kallsyms) 暴露给特权用户，JIT 镜像地址不再出现在 `/proc/kallsyms` 中。

Offloads

BPF 网络程序，尤其是 tc 和 XDP BPF 程序在内核中都有一个 offload 到硬件的接口，这样就可以直接在网卡上执行 BPF 程序。

当前，Netronome 公司的 `nfp` 驱动支持通过 JIT 编译器 offload BPF，它会将 BPF 指令翻译成网卡实现的指令集。另外，它还支持将 BPF maps offload 到网卡，因此 offloaded BPF 程序可以执行 map 查找、更新和删除操作。





eBPF 接口

BPF 系统调用

eBPF 提供了 `bpf()` 系统调用来对 BPF Map 或 程序进行操作，其函数原型如下：

```

1 #include <linux/bpf.h>
2 int bpf(int cmd, union bpf_attr *attr, unsigned int size);

```

函数有三个参数，其中：

- `cmd` 指定了 bpf 系统调用执行的命令类型，每个 cmd 都会附带一个参数 `attr`
- `bpf_attr union` 允许在内核和用户空间之间传递数据，确切的格式取决于 `cmd` 这个参数
- `size` 这个参数表示 `bpf_attr union` 这个对象以字节为单位的大小

`cmd` 可以为一下几种类型，基本上可以分为操作 eBPF Map 和操作 eBPF 程序两种类型：

- `BPF_MAP_CREATE`：创建一个 eBPF Map 并且返回指向该 Map 的文件描述符
- `BPF_MAP_LOOKUP_ELEM`：在某个 Map 中根据 key 查找元素并返回其 value
- `BPF_MAP_UPDATE_ELEM`：在某个 Map 中创建或者更新一个元素 key/value 对
- `BPF_MAP_DELETE_ELEM`：在某个 Map 中根据 key 删除一个元素
- `BPF_MAP_GET_NEXT_KEY`：在某个 Map 中根据 key 查找元素然后返回下一个元素的 key
- `BPF_PROG_LOAD`：校验并加载 eBPF 程序，返回与该程序关联的文件描述符


```

3             __u32 kern_version, char *log_buf,
4             size_t log_buf_sz)
5 {
6     struct bpf_load_program_attr load_attr;
7
8     memset(&load_attr, 0, sizeof(struct bpf_load_program_attr));
9     load_attr.prog_type = type;
10    load_attr.expected_attach_type = 0;
11    load_attr.name = NULL;
12    load_attr.insns = insns;
13    load_attr.insns_cnt = insns_cnt;
14    load_attr.license = license;
15    load_attr.kern_version = kern_version;
16
17    return bpf_load_program_xattr(&load_attr, log_buf, log_buf_sz);
18 }
19
20 int bpf_load_program_xattr(const struct bpf_load_program_attr *load_attr,
21                           char *log_buf, size_t log_buf_sz)
22 {
23     // ...
24     fd = sys_bpf_prog_load(&attr, sizeof(attr));
25     if (fd >= 0)
26         return fd;
27     // ...
28 }
29
30 static inline int sys_bpf_prog_load(union bpf_attr *attr, unsigned int siz
31 {
32     int fd;
33
34     do {
35         fd = sys_bpf(BPF_PROG_LOAD, attr, size);
36     } while (fd < 0 && errno == EAGAIN);
37
38     return fd;
39 }
```

使用 eBPF Map 的命令

和前面一样，查看 `libbpf` 中 `bpf_create_map` 的实现，可以看到最终也调用了 bpf 系统调用：



/tools/lib/bpf/bpf.c

```

1 int bpf_create_map(enum bpf_map_type map_type, int key_size,
2                   int value_size, int max_entries, u32 map_flags)
```

```

2             _value_size, _max_entries, __use_map_tags)
3     {
4         struct bpf_create_map_attr map_attr = {};
5
6         map_attr.map_type = map_type;
7         map_attr.map_flags = map_flags;
8         map_attr.key_size = key_size;
9         map_attr.value_size = value_size;
10        map_attr.max_entries = max_entries;
11
12        return bpf_create_map_xattr(&map_attr);
13    }
14
15 int bpf_create_map_xattr(const struct bpf_create_map_attr *create_attr)
16 {
17     union bpf_attr attr;
18
19     memset(&attr, '\0', sizeof(attr));
20
21     attr.map_type = create_attr->map_type;
22     attr.key_size = create_attr->key_size;
23     attr.value_size = create_attr->value_size;
24     attr.max_entries = create_attr->max_entries;
25     attr.map_flags = create_attr->map_flags;
26     if (create_attr->name)
27         memcpy(attr.map_name, create_attr->name,
28                min(strlen(create_attr->name), BPF_OBJ_NAME_LEN - 1));
29     attr.numa_node = create_attr->numa_node;
30     attr.btf_fd = create_attr->btf_fd;
31     attr.btf_key_type_id = create_attr->btf_key_type_id;
32     attr.btf_value_type_id = create_attr->btf_value_type_id;
33     attr.map_ifindex = create_attr->map_ifindex;
34     attr.inner_map_fd = create_attr->inner_map_fd;
35
36     return sys_bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
37 }

```

libbpf 中 bpf_map_lookup_elem 的实现:

/tools/lib/bpf/bpf.c

```

1 int bpf_map_lookup_elem(int fd, const void *key, void *value)
2 {
3     union bpf_attr attr;
4
5     memset(&attr, 0, sizeof(attr));

```

```
6     attr.map_fd = fd;
7     attr.key = ptr_to_u64(key);
8     attr.value = ptr_to_u64(value);
9
10    return sys_bpf(BPF_MAP_LOOKUP_ELEM, &attr, sizeof(attr));
11 }
```

libbpf 中 bpf_map_update_elem 的实现:

```
/tools/lib/bpf/bpf.c
```

```
1 int bpf_map_update_elem(int fd, const void *key, const void *value,
2                         __u64 flags)
3 {
4     union bpf_attr attr;
5
6     memset(&attr, 0, sizeof(attr));
7     attr.map_fd = fd;
8     attr.key = ptr_to_u64(key);
9     attr.value = ptr_to_u64(value);
10    attr.flags = flags;
11
12    return sys_bpf(BPF_MAP_UPDATE_ELEM, &attr, sizeof(attr));
13 }
```

libbpf 中 bpf_map_delete_elem 的实现:

```
/tools/lib/bpf/bpf.c
```

```
1 int bpf_map_delete_elem(int fd, const void *key)
2 {
3     union bpf_attr attr;
4
5     memset(&attr, 0, sizeof(attr));
6     attr.map_fd = fd;
7     attr.key = ptr_to_u64(key);
8
9     return sys_bpf(BPF_MAP_DELETE_ELEM, &attr, sizeof(attr));
10 }
```

libbpf 中 bpf_map_get_next_key 的实现:

```
/tools/lib/bpf/bpf.c
```

```
1 int bpf_map_get_next_key(int fd, const void *key, void *next_key)
2 {
3     union bpf_attr attr;
4
5     memset(&attr, 0, sizeof(attr));
6     attr.map_fd = fd;
7     attr.key = ptr_to_u64(key);
8     attr.next_key = ptr_to_u64(next_key);
9
10    return sys_bpf(BPF_MAP_GET_NEXT_KEY, &attr, sizeof(attr));
11 }
```

注意，这里的 `libbpf` 函数和之前提到的 `helper functions` 还不太一样，你可以在 Linux Manual Page: `bpf-helpers` 看到当前 Linux 支持的 `Helper functions`。以 `bpf_map_update_elem` 为例，eBPF 程序通过调用 `helper function`，其参数如下：



```
1 struct msg {
2     __s32 seq;
3     __u64 cts;
4     __u8 comm[MAX_LENGTH];
5 };
6
7 struct bpf_map_def SEC("maps") map = {
8     .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
9     .key_size = sizeof(int),
10    .value_size = sizeof(__u32),
11    .max_entries = 0,
12 };
13
14 void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

这里的第一个参数来自于 `SEC(".maps")` 语法糖创建的 `bpf_map`。

对于用户态程序，则其函数原型如下，其中通过 `fd` 来访问 eBPF map。



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

BPF 程序类型

函数 `BPF_PROG_LOAD` 加载的程序类型规定了四件事：



- 程序可以附加在哪里
- 验证器允许调用内核中的哪些帮助函数
- 网络包的数据是否可以直接访问
- 作为第一个参数传递给程序的对象类型

实际上，程序类型本质上定义了一个API。甚至还创建了新的程序类型，以区分允许调用的不同的函数列表（比如 `BPF_PROG_TYPE_CGROUP_SKB` 对比 `BPF_PROG_TYPE_SOCKET_FILTER`）。

bpf 程序会被hook到内核不同的hook点上。不同的hook点的入口参数，能力有所不同。因而定义了不同的 prog type。不同的prog type 的bpf程序能够调用的kernel function 集合也不一样。当bpf 程序加载到内核时，内核的verifier程序会根据bpf prog type，检查程序的入口参数，调用了哪些 helper function。

目前内核支持的eBPF程序类型列表如下所示：

- `BPF_PROG_TYPE_SOCKET_FILTER`：一种网络数据包过滤器
- `BPF_PROG_TYPE_KPROBE`：确定kprobe是否应该触发
- `BPF_PROG_TYPE_SCHED_CLS`：一种网络流量控制分类器
- `BPF_PROG_TYPE_SCHED_ACT`：一种网络流量控制动作
- `BPF_PROG_TYPE_TRACEPOINT`：确定 tracepoint是否应该触发
- `BPF_PROG_TYPE_XDP`：从设备驱动程序接收路径运行的网络数据包过滤器
- `BPF_PROG_TYPE_PERF_EVENT`：确定是否应该触发perf事件处理程序
- `BPF_PROG_TYPE_CGROUP_SKB`：一种用于控制组的网络数据包过滤器
- `BPF_PROG_TYPE_CGROUP_SOCK`：一种由于控制组的网络包筛选器，它被允许修改套接字选项
- `BPF_PROG_TYPE_LWT_*`：用于轻量级隧道的网络数据包过滤器
- `BPF_PROG_TYPE SOCK_OPS`：一个用于设置套接字参数的程序
- `BPF_PROG_TYPE_SK_SKB`：一个用于套接字之间转发数据包的网络包过滤器
- `BPF_PROG_CGROUP_DEVICE`：确定是否允许设备操作

随着新程序类型的添加，内核开发人员同时发现也需要添加新的数据结构。

举个例子`BPF_PROG_TYPE_SCHED_CLS` bpf prog，能够访问哪些bpf helper function呢？让我们来看看源代码是如何实现的。

每一种prog type 会定义一个 `struct bpf_verifier_ops` 结构体。当 prog load 到内核时，内核会根据它的 type，调用相应结构体的`get_func_proto` 函数。



```
1 const struct bpf_verifier_ops tc_cls_act_verifier_ops = {  
2     .get_func_proto          = tc_cls_act_func_proto,
```



```
3     .convert_ctx_access      = tc_cls_act_convert,
4 };
```

对于 BPF_PROG_TYPE_SCHED_CLS 类型的 BPF 代码，Verifier 会调用 `tc_cls_act_func_proto`，以检查程序调用的 helper function 是否都是合法的。

BPF 代码调用时机

每一种 prog type 的调用时机都不同。

BPF_PROG_TYPE_SCHED_CLS

BPF_PROG_TYPE_SCHED_CLS 的调用过程如下。

Egress 方向

egress 方向上，tcp/ip 协议栈运行之后，有一个 hook 点。这个 hook 点可以 attach BPF_PROG_TYPE_SCHED_CLS type 的 egress 方向的 bpf prog。在这段 bpf 代码执行之后，才会运行 qos, tcpdump, xmit 到网卡 driver 的代码。在这段 bpf 代码中你可以修改报文里面的内容，地址等。修改之后，通过 tcpdump 可以看到，因为 tcpdump 代码在此之后才执行。



```
1 static int __dev_queue_xmit(struct sk_buff *skb, struct net_device *sb_dev
2
3 {
4     skb = sch_handle_egress(skb, &rc, dev);
5     // enqueue tc qos
6     // dequeue tc qos
7     // dev_hard_start_xmit
8     // tcpdump works here! dev_queue_xmit_nit
9     // nic driver->ndo_start_xmit
10 }
```

Ingress 方向

ingress 方向上，在 deliver to tcp/ip 协议栈之前，在 tcpdump 之后，有一个 hook 点。这个 hook 点可以 attach BPF_PROG_TYPE_SCHED_CLS type 的 ingress 方向的 bpf prog。在这里你也可以修改报文。但是修改之后的结果在 tcpdump 中是看不到的。



```
1 static int __netif_receive_skb_core(struct sk_buff **pskb, bool pfmemalloc,
2                                     struct packet_type **ppt_prev)
3 {
4     // generic xdp bpf hook
5     // tcpdump
```

```
6         // tc ingress hook  
7         skb = sch_handle_ingress(skb, &pt_prev, &ret, orig_dev, &an  
8         // deliver to tcp/ip stack or bridge/ipvlan device  
9     }
```

执行入口 `cls_bpf_classify`

无论 egress还是ingress 方向，真正执行bpf 指令的入口都是 `cls_bpf_classify`。它遍历 `tcf_proto`中的 bpf prog link list，对每一个bpf prog 执行`BPF_PROG_RUN`(`prog->filter, skb`)

```
1 static int cls_bpf_classify(struct sk_buff *skb, const struct tcf_proto *t  
2                               struct tcf_result *res)  
3 {  
4     struct cls_bpf_head *head = rcu_dereference_bh(tp->root);  
5     struct cls_bpf_prog *prog;  
6  
7     list_for_each_entry_rcu(prog, &head->plist, link) {  
8         int filter_res;  
9  
10        if (tc_skip_sw(skb))  
11            filter_res = prog->exts_integrated ? TC_ACT_UNSPEC :  
12        } else if (at_ingress) {  
13            /* It is safe to push/pull even if skb_shared() */  
14            __skb_push(skb, skb->mac_len);  
15            bpf_compute_data_pointers(skb);  
16            filter_res = BPF_PROG_RUN(prog->filter, skb);  
17            __skb_pull(skb, skb->mac_len);  
18        } else {  
19            bpf_compute_data_pointers(skb);  
20            filter_res = BPF_PROG_RUN(prog->filter, skb);  
21    }
```

`BPF_PROG_RUN` 会执行 JIT compile 的bpf 指令，如果内核不支持JIT，则会调用解释器执行bpf的 byte code。

`BPF_PROG_RUN` 传给bpf prog的入口参数是`skb`，其类型是 `struct sk_buff`，定义在文件 `include/linux/skbuff.h`中。

但是在bpf 代码中，为了安全，不能直接访问 `sk_buff`。bpf中是通过访问 `struct __sk_buff` 来访问`struct sk_buff`的。`__sk_buff` 是 `sk_buff` 的一个子集，是`sk_buff`面向bpf 程序的接口。bpf代码中对 `__sk_buff` 的访问会在verifier程序中翻译成对`sk_buff`相应fileds的访问。

在加载bpf prog的时候， verifier会调用上面 `tc_cls_act_verifier_ops` 结构体里面的 `tc_cls_act_convert_ctx_access`的钩子。它最终会调用下面的函数修改ebpf的指令，使得对 `__sk_buff` 的访问变成对 `struct sk_buff` 的访问。

BPF Attach type

一种 type 的bpf prog 可以挂到内核中不同的hook点，这些不同的hook点就是不同的attach type。

其对应关系在下面函数 中定义了。



```
1 attach_type_to_prog_type(enum bpf_attach_type attach_type)
2 {
3     switch (attach_type) {
4         case BPF_CGROUP_INET_INGRESS:
5         case BPF_CGROUP_INET_EGRESS:
6             return BPF_PROG_TYPE_CGROUP_SKB;
7         case BPF_CGROUP_INET_SOCK_CREATE:
8         case BPF_CGROUP_INET_SOCK_RELEASE:
9         case BPF_CGROUP_INET4_POST_BIND:
10        case BPF_CGROUP_INET6_POST_BIND:
11            return BPF_PROG_TYPE_CGROUP_SOCK;
12        .....
13 }
```

当bpf prog 通过系统调用bpf() attach到具体的hook点时，其入口参数中就需要指定attach type。

有趣的是，BPF_PROG_TYPE_SCHED_CLS 类型的 bpf prog 不能通过bpf系统调用来attach，因为它没有定义对应的 attach type。故它的 attach 需要通过netlink interface 额外的实现，还是非常复杂的。

常用 prog type 介绍

内核中的 prog type 目前有30种。每一种type 能做的事情有所差异，这里只讲讲我平时工作用过的几种。

理解一种prog type的最好的方法是

- 查表 attach_type_to_prog_type，得到它的 attach type，
- 再搜索内核代码，看这些 attach type 在内核哪里被调用了。
- 最后看看它的入口参数和 return value 的处理过程，基本就能理解其作用了。



```
1 include/uapi/linux/bpf.h
2
3 enum bpf_prog_type {
4 }
```



BPF_PROG_TYPE_SOCKET_FILTER

是第一个被添加到内核的程序类型。当你attach一个bpf程序到socket上，你可以获取到被socket处理的所有数据包。socket过滤不允许你修改这些数据包以及这些数据包的目的地。仅仅是提供给你观察这些数据包。在你的程序中可以获取到诸如protocol type类型等。

以 tcp 为 example，调用的地点是 `tcp_v4_rcv->tcp_filter->sk_filter_trim_cap` 作用是过滤报文，或者 trim报文。udp, icmp中也有相关的调用。

BPF_PROG_TYPE_SOCK_OPS

在 tcp 协议 event 发生时调用的bpf 钩子，定义了15种event。这些event的 attach type 都是 BPF_CGROUP_SOCK_OPS。不同的调用点会传入不同的enum, 比如：

- BPF_SOCK_OPS_TCP_CONNECT_CB 是主动 tcp connect call 的；
- BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB是被动connect 成功时调用的。

主要作用：tcp 调优，event 统计等。

BPF_PROG_TYPE_SOCK_OPS 这种程序类型，允许你当数据包在内核网络协议栈的各个阶段传输的时候，去修改套接字的链接选项。他们attach到cgroups上，和BPF_PROG_TYPE_CGROUP_SOCK以及 BPF_PROG_TYPE_CGROUP_SKB很像，但是不同的是，他们可以在整个连接的生命周期内被调用好多次。你的bpf程序会接受到一个op的参数，该参数代表内核将通过套接字链接执行的操作。因此，你知道在链接的生命周期内何时调用该程序。另一方面，你可以获取ip地址，端口等。你还可以修改链接的链接的选项以设置超时并更改数据包的往返延迟时间。

举个例子，Facebook 使用它来为同一数据中心内的连接设置短恢复时间目标（RTO）。RTO是一种时间，它指的是网络在出现故障后的恢复时间，这个指标也表示网络在受到不可接受到情况下的，不能被使用的时间。Facebook认为，在同一数据中心中，应该有一个很短的RTO,Facebook修改了这个时间，使用bpf程序。

BPF_PROG_TYPE_CGROUP_SOCK_ADDR

它对应很多attach type，一般在bind, connect 时调用，传入 sock 的地址。

主要作用：例如 cilium中 clusterip 的实现，在主动 connect 时，修改了目的ip地址，就是利用这个。

BPF_PROG_TYPE_CGROUP_SOCK_ADDR，这种类型的程序使您可以在由特定cgroun控制的用户空间程序中操纵IP地址和端口号。在某些情况下，当您要确保一组特定的用户空间程序使用相同的IP地址和端口号时，系统将使用多个IP地址.当您将这些用户空间程序放在同一cgroun中时，这些BPF程序使您可以灵活地操作这些绑定。这样可以确保这些应用程序的所有传入和传出连接均使用BPF程序提供的IP和端口。

BPF_PROG_TYPE_SK_MSG

BPF_PROG_TYPE_SK_MSG, These types of programs let you control whether a message sent to a socket should be delivered.当内核创建了一个socket，它会被存储在前面提到的map中。当你 attach一个程序到这个socket map的时候，所有的被发送到那些socket的message都会被filter。在

filter message之前，内核拷贝了这些data，因此你可以读取这些message，而且可以给出你的决定：例如，SK_PASS和SK_DROP。

BPF_PROG_TYPE_SK_SKB

调用点：tcp sendmsg 时会调用。

主要作用：做sock redirect 用的。

BPF_PROG_TYPE_SK_SKB，这类程序可以让你获取socket maps和socket redirects。socket maps可以让你获得一些socket的引用。当你有了这些引用，你可以使用相关的helpers，去重定向一个 incoming 的 packet，从一个socket去另外一个socket.这在使用BPF来做负载均衡时是非常有用的。你可以在socket之间转发网络数据包，而不需要离开内核空间。Cilium和facebook的Katranc 广泛的使用这种类型的程序去做流量控制。

BPF_PROG_TYPE_CGROUP_SOCKOPT

调用点：getsockopt, setsockopt

BPF_PROG_TYPE_KPROBE

类似 ftrace 的kprobe，在函数出入口的 hook 点，debug 用的。

BPF_PROG_TYPE_TRACEPOINT

类似 ftrace 的 tracepoint。

BPF_PROG_TYPE_SCHED_CLS

如上面的例子

BPF_PROG_TYPE_XDP

网卡驱动收到packet时，尚未生成 sk_buff 数据结构之前的一个hook点。

BPF_PROG_TYPE_XDP 允许你的 bpf 程序，在网络数据包到达 kernel 很早的时候。在这样的bpf程序中，你仅仅可能获取到一点点的信息，因为kernel还没有足够的时间去处理。因为时间足够的早，所以你可以在网络很高的层面上去处理这些 packet。

XDP定义了很多的处理方式，例如

- XDP_PASS 就意味着，你会把packet交给内核的另一个子系统去处理
- XDP_DROP就意味着，内核应该丢弃这个数据包
- XDP_TX意味着，你可以把这个包转发到network interface card(NIC)第一次接收到这个包的时候

BPF_PROG_TYPE_CGROUP_SKB



BPF_PROG_TYPE_CGROUP_SKB 允许你过滤整个 cgroup 的网络流量。在这种程序类型中，你可以在网络流量到达这个 cgroup 中的程序前做一些控制。内核试图传递给同一 cgroup 中任何进程的任何数据包都将通过这些过滤器之一。同时，您可以决定 cgroup 中的进程通过该接口发送网络数据包时该怎么做。其实，你可以发现它和 BPF_PROG_TYPE_SOCKET_FILTER 的类型很类似。最大的不同是 cgroup_skb 是 attach 到这个 cgroup 中的所有进程，而不是特殊的进程。在 container 的环境中，bpf 是非常有用的。

- ingress 方向上，tcp 收到报文时 (tcp_v4_rcv)，会调用这个 bpf 做过滤。
- egress 方向上，ip 在出报文时 (ip_finish_output) 会调用它做丢包过滤 输入参数是 skb。

BPF_PROG_TYPE_CGROUP_SOCK

在 sock create, release, post_bind 时调用的。主要用来做一些权限检查的。

BPF_PROG_TYPE_CGROUP_SOCK，这种类型的 bpf 程序允许你，在一个 cgroup 中的任何进程打开一个 socket 的时候，去执行你的 Bpf 程序。这个行为和 CGROUP_SKB 的行为类似，但是它是提供给你 cgroup 中的进程打开一个新的 socket 的时候的情况，而不是给你网络数据包通过的权限控制。这对于为可以打开套接字的程序组提供安全性和访问控制很有用，而不必分别限制每个进程的功能。

eBPF 工具链

bcc

BCC 是 BPF 的编译工具集合，前端提供 Python/Lua API，本身通过 C/C++ 语言实现，集成 LLVM/Clang 对 BPF 程序进行重写、编译和加载等功能，提供一些更人性化的函数给用户使用。

虽然 BCC 竭尽全力地简化 BPF 程序开发人员的工作，但其“黑魔法”（使用 Clang 前端修改了用户编写的 BPF 程序）使得出现问题时，很难找到问题的所在以及解决方法。必须记住命名约定和自动生成的跟踪点结构。且由于 libbcc 库内部集成了庞大的 LLVM/Clang 库，使其在使用过程中会遇到一些问题：

1. 在每个工具启动时，都会占用较高的 CPU 和内存资源来编译 BPF 程序，在系统资源已经短缺的服务器上运行可能引起问题；
2. 依赖于内核头文件包，必须将其安装在每个目标主机上。即便如此，如果需要内核中未 export 的内容，则需要手动将类型定义复制/粘贴到 BPF 代码中；
3. 由于 BPF 程序是在运行时才编译，因此很多简单的编译错误只能在运行时检测到，影响开发体验。

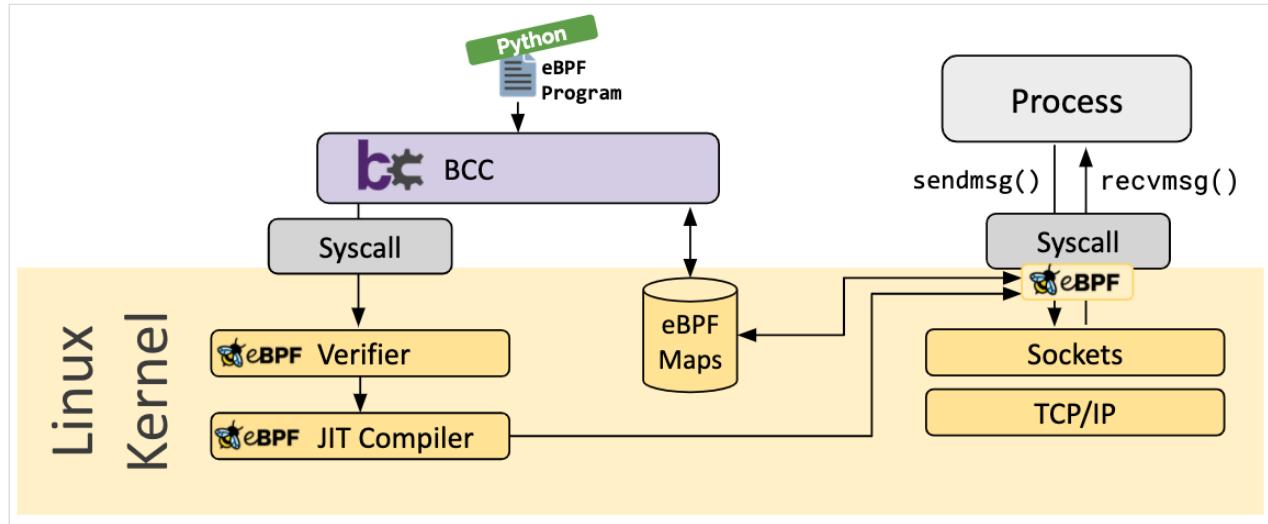
随着 BPF CO-RE 的落地，我们可以直接使用内核开发人员提供的 libbpf 库来开发 BPF 程序，开发方式和编写普通 C 用户态程序一样：一次编译生成小型的二进制文件。Libbpf 作为 BPF 程序加载器，接管了重定向、加载、验证等功能，BPF 程序开发者只需要关注 BPF 程序的正确性和性能即可。这种方式将开销降到了最低，且去除了庞大的依赖关系，使得整体开发流程更加顺畅。

性能优化大师 Brendan Gregg 在用 libbpf + BPF CO-RE 转换一个 BCC 工具后给出了性能对比数据：



As my colleague Jason pointed out, the memory footprint of opensnoop as CO-RE is much lower than opensnoop.py. 9 Mbytes for CO-RE vs 80 Mbytes for Python.

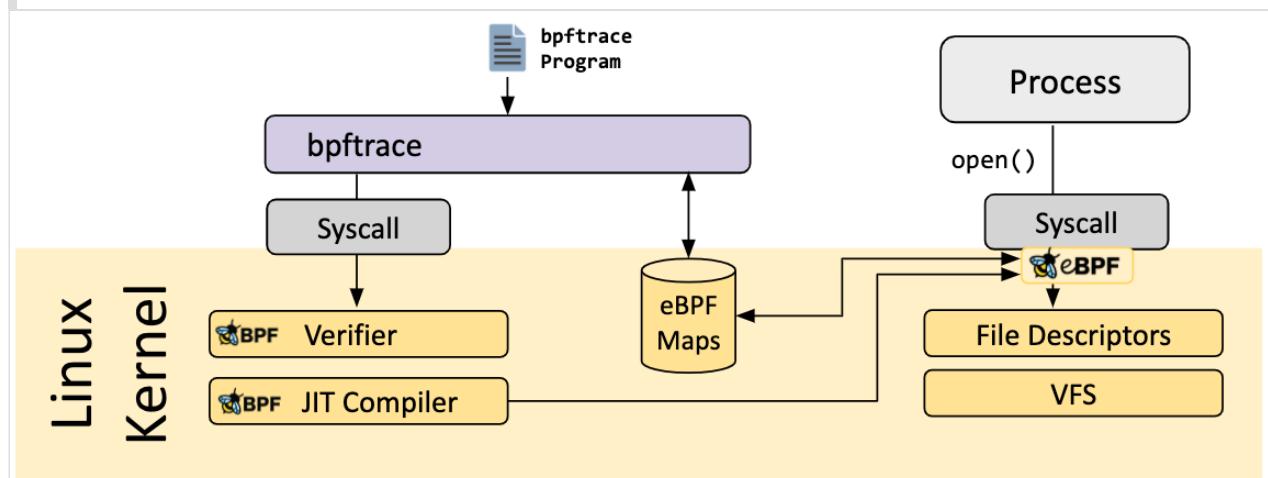
我们可以看到在运行时相比 BCC 版本，libbpf + BPF CO-RE 版本节约了近 9 倍的内存开销，这对于物理内存资源已经紧张的服务器来说会更友好。



关于 BCC 可以参考 我的这篇文章介绍

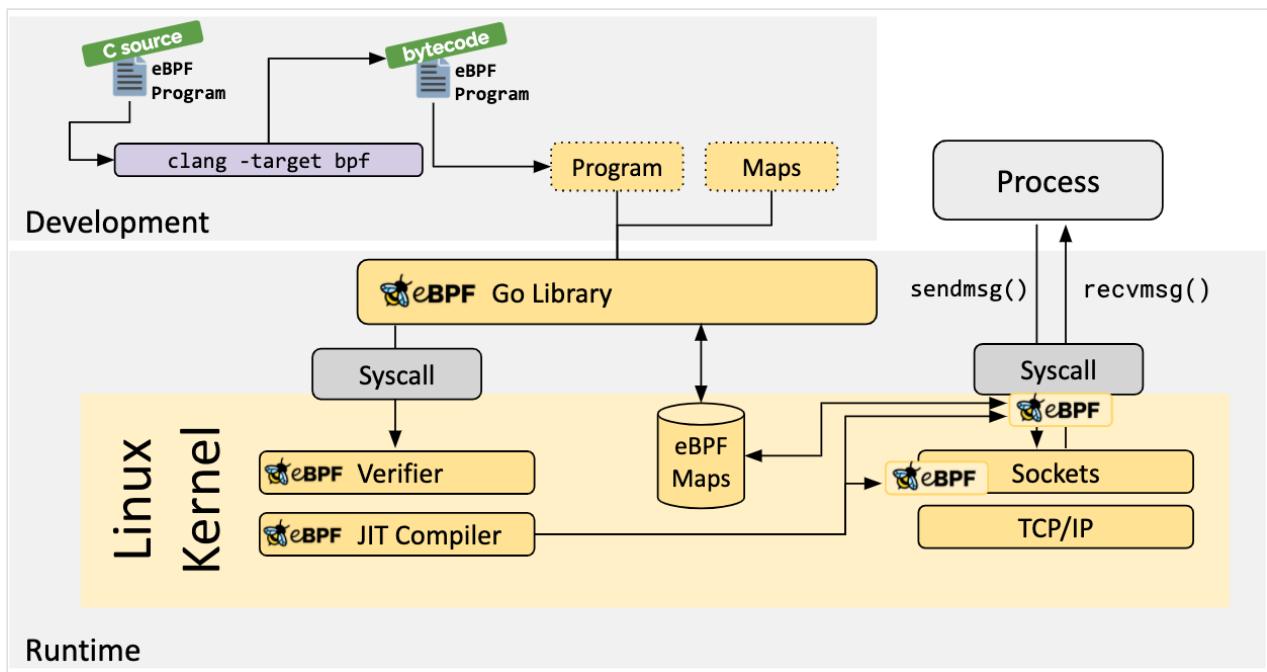
bpftrace

bpftrace is a high-level tracing language for Linux eBPF and available in recent Linux kernels (4.x). bpftrace uses LLVM as a backend to compile scripts to eBPF bytecode and makes use of BCC for interacting with the Linux eBPF subsystem as well as existing Linux tracing capabilities: kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes), and tracepoints. The bpftrace language is inspired by awk, C and predecessor tracers such as DTrace and SystemTap.

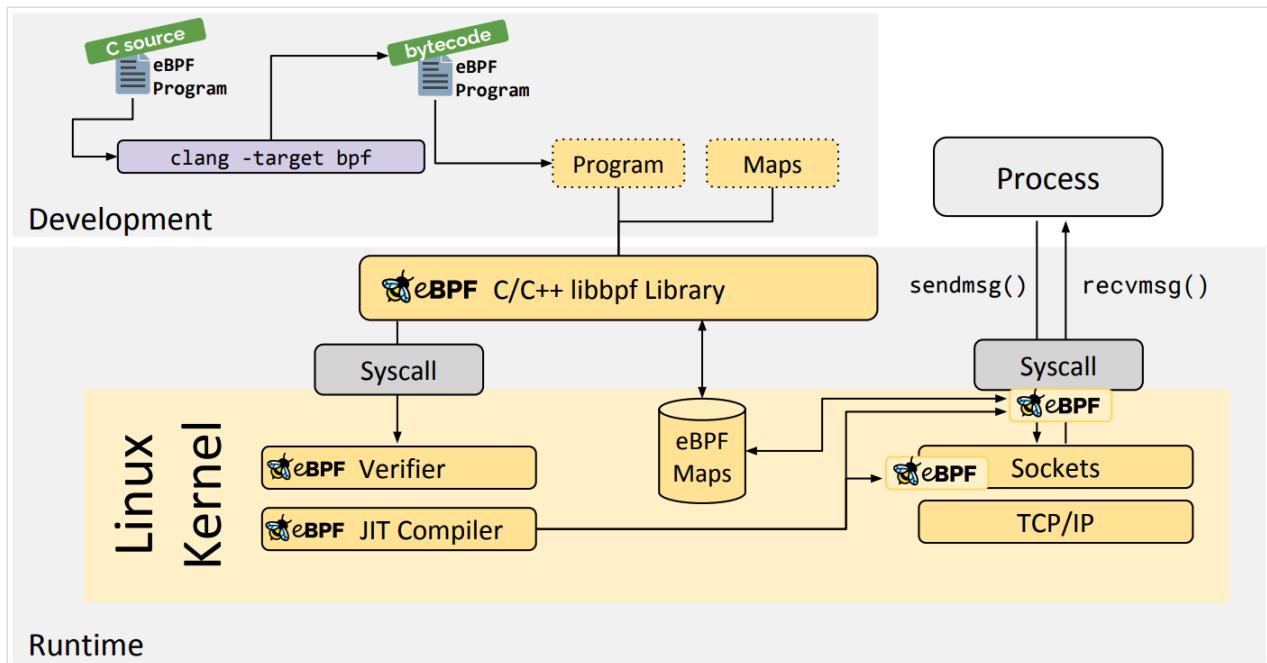


eBPF Go Library





libbpf



参考资料

- The BSD Packet Filter: A New Architecture for User-level Packet Capture, Steven McCanne and Van Jacobso, December 19, 1992
- eBPF Documentation: What is eBPF?
- LWN: A thorough introduction to eBPF
- Cilium Documentation: BPF and XDP Reference Guide
- eBPF summit: The Future of eBPF based Networking and Security
- eBPF – The Future of Networking & Security



- eBPF – Rethinking the Linux Kernel
- Linux Manual Page: bpf(2)
- Linux Manual Page: bpf-helpers
- Linux Kernel Documentation: Linux Socket Filtering aka Berkeley Packet Filter (BPF)
- Dive into BPF: a list of reading material
- LWN: eBPF materials
- 基于 Ubuntu 20.04 的 eBPF 环境搭建

相关文章推荐

- eBPF 指令集
- eBPF tc 子系统
- Linux Traffic Control
- 网卡聚合 Bonding
- Linux网络包收发流程

打赏

本文作者: Houmin

本文链接: <http://houmin.cc/posts/2c811c2c/>

版权声明: 本博客所有文章除特别声明外, 均采用  BY-NC-SA 许可协议。转载请注明出处!

 网络

 linux

 tc

 BPF

 tracing

 XDP

◀ TCPdump 原理与实现

eBPF Map 操作 ➤

昵称

邮箱

网址(<http://>)



Just go go

M+



提交

来发评论吧~

Powered By [Valine](#)
v1.4.18

God said: Let there be light.——【Holy Bible】

© 2019 – 2022 ❤️ Houmin | 🎵 3.6m | 🎥 108:01

👤 34315 | 👁 79236

