# 简介

BPF，即Berkeley Packet Filter，是一个古老的网络封包过滤机制。它允许从用户空间**注入一段简短的字节码到内核**来定制封包处理逻辑。Linux从2.5开始移植了BPF，tcpdump就是基于BPF的应用。

所谓eBPF（extended BPF），则是从3.18引入的，对BPF的改造和功能增强：

1. 使用类似于X86的体系结构，eBPF设计了一个通用的RISC指令集，支持11个64bit寄存器（32bit子寄存器）r0-r10，使用512字节的栈空间
2. **引入了JIT编译**，取代了BPF解释器。eBPF程序直接被编译为目标体系结构的机器码
3. 和网络子系统进行了解耦。它的数据模型是通用的，eBPF程序可以挂钩到`Kprobe`或`Tracepoint`
4. 使用Maps来存储全局数据，这是一种通用的键值存储。可用作不同eBPF程序、eBPF和用户空间程序的**状态共享**
5. 助手函数（Helper Functions），这些函数供eBPF程序调用，可以实现封包改写、Checksum计算、封包克隆等能力
6. 尾调用（Tail Calls），可以用于将程序控制权从一个eBPF转移给另外一个。老版本的eBPF对程序长度有4096字节的限制，通过尾调用可以规避
7. 用于Pin对象（Maps、eBPF程序）的**伪文件系统**
8. 支持将eBPF Offload给智能硬件的基础设施

以上增强，让eBPF不仅仅限于网络封包处理，当前eBPF的应用领域包括：

1. 网络封包处理：XDP、TC、socket progs、kcm、calico、cilium等
2. 内核跟踪和性能监控：KProbes、UProbes、TracePoints
3. 安全领域：Secomp、landlock等。例如阻止部分类型的系统调用

现在BPF一般都是指eBPF，而老的BPF一般称为cBPF（classic BPF）。

性能是eBPF的另外一个优势，由于**所有代码都在内核空间运行，避免了复制数据到用户空间、上下文切换等开销。**甚至编译过程都在尽可能的优化，例如助手函数会被内联到eBPF程序中，避免函数调用的开销。

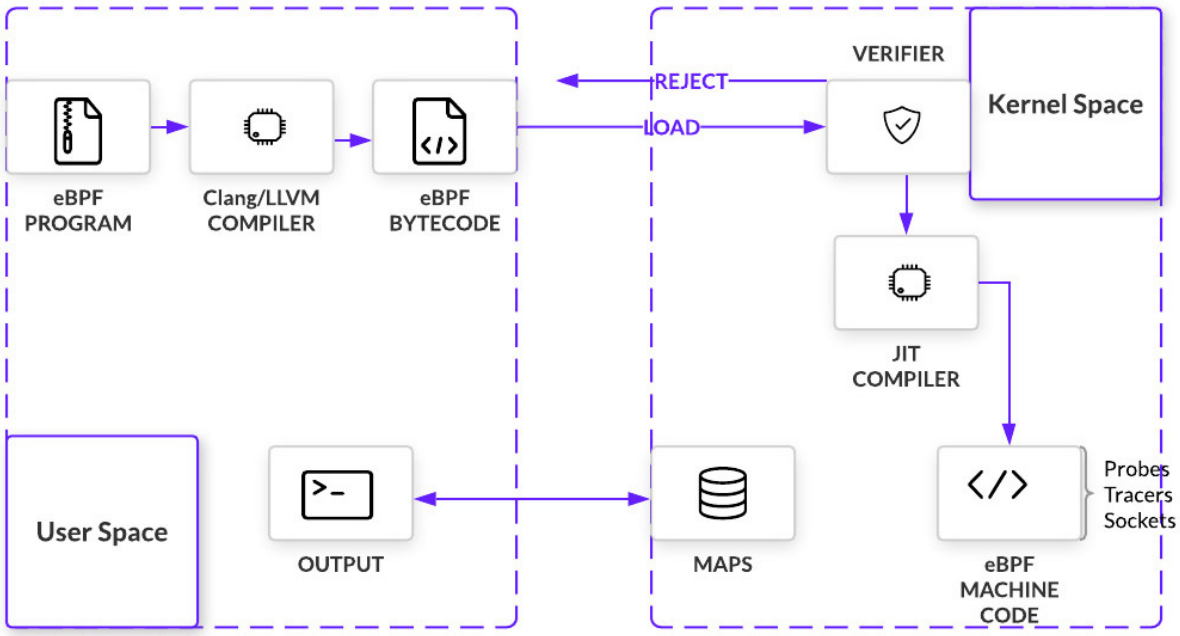
用户提供的代码在内核中运行，安全性需要得到保证。eBPF校验器会对字节码进行各方面的检查，确保它不会导致内核崩溃或锁死。

eBPF具有非常好的灵活性、动态性，可以随时的注入、卸载，不需要重启内核或者中断网络连接。

eBPF程序可以在不同体系结构之间移植。

# eBPF基础

## BPF架构




如上图所示，eBPF应用程序，从开发到运行的典型流程如下：

1. 利用Clang，将C语言开发的代码编译为eBPF object文件
2. 在用户空间将eBPF object文件载入内核。载入前，可能对object文件进行各种修改。这一步骤，可能通过iproute2之类的BPF ELF loader完成，也可能通过自定义的控制程序完成
3. BPF Verifier在VM中进行安全性校验
4. JIT编译器将字节码编译为机器码，返回BPF程序的文件描述符
5. 使用文件描述符将BPF程序挂钩到某个子系统（例如networking）的挂钩点。子系统有可能将BPF程序offload给硬件（例如智能网卡）
6. 用户空间通过eBPF Map和内核空间交换数据，获知eBPF程序的执行结果

# 挂钩点

eBPF程序以事件驱动的方式执行，具体来说，就是在**内核的代码路径上，存在大量挂钩点**（Hook Point）。eBPF程序会注册到某些挂钩点，当内核运行到挂钩点后，就执行eBPF程序。

挂钩点主要包括以下几类：

1. 网络事件，例如封包到达
2. Kprobes / Uprobes
3. 系统调用
4. 函数的入口/退出点

# BPF Verifier

在加载之后，BPF校验器负责验证eBPF程序是否安全，它会模拟所有的执行路径，并且：

1. 检查程序控制流，发现循环
2. 检测越界的跳转、不可达指令
3. 跟踪Context的访问、栈移除
4. 检查unpriviledged的指针泄漏
5. 检查助手函数调用参数

# BPF JITs

在校验之后，eBPF程序被JIT编译器编译为Native代码。

# BPF Maps

键值对形式的存储，通过文件描述符来定位，值是不透明的Blob（任意数据）。用于跨越多次调用共享数据，或者与用户空间应用程序共享数据。

一个eBPF程序可以**直接访问最多64个Map**，多个eBPF程序可以共享同一Map。

# Pinning

BPF Maps和程序都是内核资源，仅能通过文件描述符访问到。文件描述符对应了内核中的匿名inodes。

用户空间程序可以使用大部分基于文件描述符的API，但是**文件描述符是限制在进程的生命周期内的**，这导致Map难以被共享。比较显著的例子是iproute2，当tc或XDP加载eBPF程序之后，自身会立刻退出。这导致无法从用户空间访问Map。

为了解决上面的问题，引入了一个最小化的、**内核空间中的BPF文件系统。BPF程序和Map会被pin到一个被称为object pinning的进程。**bpf系统调用有两个命令BPF_OBJ_PIN、BPF_OBJ_GET分别用于钉住、取回对象。

tc这样的工具就是利用Pinning在ingress/egress端共享Map。

# 尾调用

尾调用允许一个BPF程序调用另外一个，这种调用**没有函数调用那样的开销。**其实现方式是long jump，重用当前stack frame。

注意：只用相同类型的BPF程序才能相互尾调用。

要使用尾调用，需要一个BPF_MAP_TYPE_PROG_ARRAY类型的Map，其内容目前必须由用户空间产生，值是需要被尾调用的BPF程序的文件描述符。通过助手函数bpf_tail_call触发尾调用，内核会将此调用内联到一个特殊的BPF指令。

# BPF-BPF调用

BPF - BPF调用是一个新添加的特性。在此特性引入之前，典型的BPF C程序需要将所有可重用的代码声明为always_inline的，这样才能确保LLVM生成的object包含所有函数。这会导致函数在每个object文件中都反复（只要它被调用超过一次）出现，增加体积。

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

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

#ifndef __inline
# define __inline                         \
   inline __attribute__((always_inline))
#endif

// 总是内联
static __inline int foo(void)
{
    return XDP_DROP;
```

```
}

__section("prog")
int xdp_drop(struct xdp_md *ctx)
{
    return foo();
}

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

总是需要内联的原因是BPF的Loader/Verifier/Interpreter/JITs不支持函数调用。但是从**内核4.16和LLVM 6.0开始，此限制消除**，BPF程序不再总是需要always_inline。上面程序的__inline可以去掉了。

目前x86_64/arm64的JIT编译器支持BPF to BPF调用，这是很重要的性能优化，因为它大大简化了生成的object文件的尺寸，对CPU指令缓存更加友好。

JIT编译器**为每个函数生成独立的映像**（Image），并且**在JIT的最后一个步骤中修复映像中的函数调用地址。**

到5.9为止，你不能同时使用BPF-BPF调用（BPF子程序）和尾调用。从5.10开始，可以混合使用，但是仍然存在一些限制。此外，**混合使用两者可能导致内核栈溢出**，原因是尾调用在跳转之前仅会unwind当前栈帧。

# Offloading

BPF网络程序，特别是tc和XDP，提供了将BPF代码offload给NIC执行的特性。这个特性需要驱动的支持。

# BPF前端工具

能够加载BPF程序的前端工具有很多，包括bcc、perf、iproute2等。内核也在tools/lib/bpf目录下提供了用户空间库，被perf用来加载BPF追踪应用程序到内核。这是一个通用的库，你也可以直接调用它。BCC是面向追踪的工具箱。内核在samples/bpf下也提供了一些BPF示例，这些示例解析Object文件，并且直接通过系统调用将其载入内核。

基于不同前端工具，实现BPF程序的语法、语义（例如对于段名的约定）有所不同。

# 相关sysctl

## /proc/sys/net/core/bpf_jit_enable

启用或禁用BPF JIT编译器：

0 仅用，仅仅使用解释器，默认值

1 启用JIT编译器

2 启用JIT编译器并且生成debugging trace到内核日志

设置为2，可以使用bpf_jit_disasm处理debugging trace

## /proc/sys/net/core/bpf_jit_harden

启用或禁用JIT加固，加固和性能是对立的，但是可以缓和JIT spraying：

0 禁用JIT加固，默认值

1 对非特权用户启用

2 对所有用户启用

## /proc/sys/net/core/bpf_jit_kallsyms

启用或禁用JITed的程序的内核符号导出（导出到/proc/kallsyms），这样可以和perf工具一起使用，还能够让内核对BPF程序的地址感知（用于stack unwinding）：

0 启用

1 仅对特权用户启用

## /proc/sys/kernel/unprivileged_bpf_disabled

是否启用非特权的bpf系统调用。默认启用，一旦禁用，重启前无法恢复启用状态。不会影响seccomp等不使用bpf2系统调用的cBPF程序：

0 启用

1 禁用

# 助手函数

eBPF程序可以调用助手函数，完成各种任务，例如：

1. 在Map中搜索、更新、删除键值对
2. 生成伪随机数
3. 读写隧道元数据
4. 尾调用 —— 将eBPF程序链在一起
5. 执行套接字相关操作，例如绑定、查询Cookies、重定向封包
6. 打印调试信息
7. 获取系统启动到现在的时间

助手函数是定义在内核中的，有一个白名单，决定哪些内核函数可以被eBPF程序调用。

根据eBPF的约定，**助手函数的参数数量不超过5。**

编译后，助手函数的代码是内联到eBPF程序中的，因而不存在函数调用的开销（栈帧处理开销、CPU流水线预取指令失效开销）。

返回int的类型的助手函数，通常操作成功返回0，否则返回负数。如果不是如此，会特别说明。

助手函数不可以随意调用，不同类型的eBPF程序，可以调用不同的助手函数子集。

# iproute2

iproute2提供的BPF前端，主要用来载入BPF网络程序，这些程序的类型包括XDP、tc、lwt。只要是为iproute2编写的BPF程序，共享统一的加载逻辑。

## XDP

### 加载XDP程序

编译好的XDP类型（BPF_PROG_TYPE_XDP）的BPF程序，可以使用如下命令载入到支持XDP的网络设备：

```
ip link set dev eth0 xdp obj prog.o
```

上述命令假设程序位于名为prog的段中。如果不使用默认段名，则需要指定sec参数：

```
# 如果程序放在foobar段
ip link set dev em1 xdp obj prog.o sec foobar
```

如果程序没有标注段，也就是位于默认的.text段，则也可以用上面的命令加载。

如果已经存在挂钩到网络设备的XDP程序，默认情况下命令会报错，可以用-force参数强制替换：

```
ip -force link set dev em1 xdp obj prog.o
```

大多数支持XDP的驱动，能够原子的替换XDP程序，而不会引起流量中断。出于性能的考虑同时只能有一个XDP程序挂钩， 可以利用前文提到的尾调用来组织多个XDP程序。

如果网络设备挂钩了XDP程序，则ip link命令会显示xdp标记和程序的ID。使用bpftool传入ID可以查看更多细节信息。

## 卸载XDP程序

```
ip link set dev eth0 xdp off
```

## XDP操作模式

iproute2实现了XDP所支持的三种操作模式：

1. xdpdrv：即native XDP，**BPF程序在驱动的接收路径的最早时刻被调用。**这是正常的XDP模式，上游内核的所有主要10G/40G+网络驱动（包括virtio）都实现了XDP支持，也就是可使用该模式
2. xdpoffload：由智能网卡的驱动（例如Netronome的nfp驱动）实现，将整个XDP程序offload到硬件中，网卡每接收到封包都会执行XDP程序。该模式比native XDP的性能更高，缺点是，并非所有助手函数、Map类型可用。
3. xdpgeneric：即generic XDP，作为尚不支持native XDP的驱动的试验台。挂钩点比native XDP晚很多，已经进入网络栈的主接收路径，生成了skb对象，因此性能比native XDP差很多，不会用于生产环境

在切换驱动的XDP模式时，驱动通常需要重新配置它的接收/发送Rings，以保证接收到的封包线性的（linearly）存放到单个内存页中。

调用ip link set dev xxx xdp命令时，内核会首先尝试在native XDP模式下载入，如果驱动不支持，则自动使用generic XDP模式。要强制使用native XDP，则可以使用：

```
#                           强制使用native XDP
ip -force link set dev eth0 xdpdrv obj prog.o
```

使用类似的方式可以强制使用xdpgeneric、xdpoffload。

**切换操作模式目前不能原子的进行，但是在单个操作模式下替换XDP程序则可以。**

使用`verb`选项，可以显示详尽的BPF校验日志：

```
ip link set dev eth0 xdp obj xdp-example.o verb
```

除了从文件加载BPF程序，也可以直接从BPF伪文件系统中得到程序并使用：

```
ip link set dev eth0 xdp pinned /sys/fs/bpf/prog
#                               m:表示BPF文件系统的挂载点，默认/sys/fs/bpf/
ip link set dev eth0 xdp pinned m:prog
```

# tc

对于为tc设计的BPF程序（BPF_PROG_TYPE_SCHED_CLS、BPF_PROG_TYPE_SCHED_ACT），可以使用tc命令加载并挂钩到网络设备。和XDP不同，tc程序没有对驱动的依赖。

`clsact`是4.1引入了一个特殊的dummy qdisc，它持有classifier和action，但是不能执行实际的queueing。要挂钩BPF classifier，clsact是必须启用的：

```
tc qdisc add dev eth0 clsact
```

clsact提供了两个特殊的钩子`ingress`、`egress`，对应了BPF classifier可用的两个挂钩点。这两个钩子位于网络数据路径的中心位置，任何封包都必须经过

下面的命令，将BPF程序挂钩到eth0的ingress路径上：

```
tc filter add dev eth0 ingress bpf da obj prog.o
```

下面的命令将BPF程序挂钩到eth0的egress路径上：

```
tc filter add dev eth0 egress bpf da obj prog.o
```

ingress钩子在内核中由`__netif_receive_skb_core() -> sch_handle_ingress()`调用。

egress钩子在内核中由`__dev_queue_xmit() -> sch_handle_egress()`调用。

**clsact是以无锁方式处理的，支持挂钩到虚拟的、没有队列概念的网络设备，例如veth。**

`da`即direct-action模式，这是推荐的模式，应当总是在命令中指定。da模式表示BPF classifier不需要调用外部的tc action模块，因为BPF程序会将封包修改、转发或者其它动作都完成，这正是BPF性能优势所在。

类似XDP，如果不使用默认的段名，需要用sec选项：

```
tc filter add dev eth0 egress bpf da obj prog.o sec foobar
```

已经挂钩到设备的tc程序的列表，可以用下面的命令查看：

```
tc filter show dev em1 ingress
filter protocol all pref 49152 bpf

     # 针对的L3协议     优先级       分类器类型   分类器句柄
filter protocol all    pref 49152 bpf        handle 0x1
  # 从prog.o的ingress段加载了程序
  prog.o:[ingress]
  # BPF程序运行在da模式
  direct-action
  # 程序ID是全局范围唯一的BPF程序标识符，可以被bpftool使用
  id 1
  # 程序指令流的哈希，哈希可以用来关联到Object文件，perf报告栈追踪的时候使用此哈希
  tag c5f7825e5dac396f


tc filter show dev em1 egress
filter protocol all pref 49152 bpf
```

```
filter protocol all pref 49152 bpf handle 0x1 prog.o:[egress] direct-action id 2
tag b2fd5adc0f262714
```

tc可以挂钩多个BPF程序，这和XDP不同，它提供了多个其它的、可以链接在一起的classifier。尽管如此，单个da模式的BPF程序可以满足所有封包操作需求，它可以直接返回action断言，例如`TC_ACT_OK`,`TC_ACT_SHOT`。使用单个BPF程序是推荐的用法。

除非打算自动替换挂钩的BPF程序，建议初次挂钩时明确的指定pref和handle，这样，在后续手工替换的时候就不需要查询获取pref、handle：

```
tc filter add dev eth0 ingress pref 1 handle 1 bpf da obj prog.o sec foobar
```

使用下面的命令原子的替换BPF程序：

```
tc filter replace dev eth0 ingress pref 1 handle 1 bpf da obj prog.o sec foobar
```

要移除所有以及挂钩的BPF程序，执行：

```
tc filter del dev eth0 ingress
tc filter del dev eth0 egress
```

要从网络设备上移除整个clsact qdisc，可以：

```
tc qdisc del dev eth0 clsact
```

类似于XDP程序，tc程序也支持offload给职能网卡。你需要首先启用hw-tc-offload：

```
ethtool -K eth0 hw-tc-offload on
```

然后再启用clsact并挂钩BPF程序。XDP和tc的offloading不能同时开启。

# netdevsim

内核提供了一个dummy驱动netdevsim，它实现了XDP/tc BPF的offloading接口，用于测试目的。

下面的命令可以启用netdevsim设备：

```
modprobe netdevsim
echo "1 1" > /sys/bus/netdevsim/new_device
devlink dev
# netdevsim/netdevsim1
devlink port
# netdevsim/netdevsim1/0: type eth netdev eth0 flavour physical
ip l
# 4: eth0: <BROADCAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
mode DEFAULT group default qlen 1000
#     link/ether 2a:d5:cd:08:d1:3f brd ff:ff:ff:ff:ff:ff
```

# XDP

# 简介

在网络封包处理方面，出现过一种提升性能的技术 —— 内核旁路（Kernel Bypass ）：完全在用户空间实现网络驱动和整个网络栈，避免上下文切换、内核网络层次、中断处理。具体实现包括Intel的DPDK（Data Plane Development Kit）、Cisco的VPP等。

内核旁路技术的缺点是：

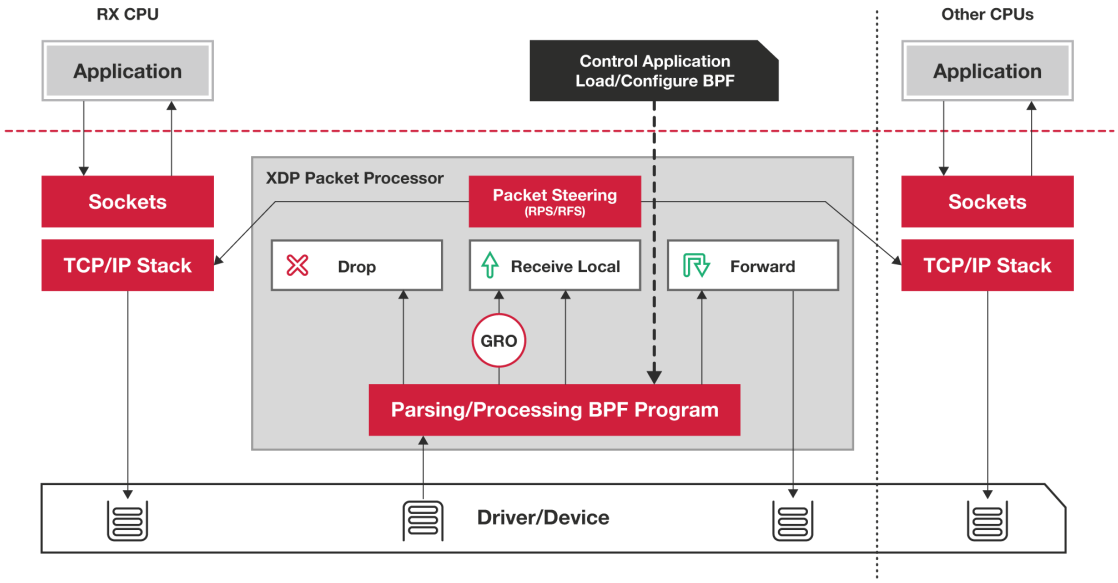
1. 作为硬件资源的抽象层，内核是经过良好测试和验证的。在用户空间重新实现驱动，稳定性、可复用性欠佳
2. 实现网络栈也是困难的
3. 作为一个沙盒，网络处理程序难以和内核其它部分集成/交互
4. 无法使用内核提供的安全层

**eXpress Data Path，为内核提供了一个基于eBPF的、高性能的、可编程的、运行在驱动层的封包处理框架**，它提升性能的思路和内核旁路技术相反 —— 完全在内核空间实现封包处理逻辑，例如过滤、映射、路由等。XDP通过在网络接收路径的最早期挂钩eBPF程序来实现高速封包过滤。最早期意味着：NIC驱动刚刚从receiver rings接收到封包，任何高成本操作，例如分配skb并将封包推入网络栈，尚未进行。

XDP的起源来自于对DDoS攻击的防范。Cloudflare依赖（leverages heavily on）iptables进行封包过滤，在配置相当好的服务器上，可以处理1Mpps的流量。但是当出现DDoS攻击时，流量会高达3Mpps，这会导致Linux系统overflooded by IRQ请求，直到系统变得不稳定。

由于Cloudflare希望继续使用iptables以及其它内核网络栈功能，它不考虑使用DPDK这样的完全控制硬件的方案，而是使用了所谓部分内核旁路（partial kernel bypass），NIC的一部分队列继续附到内核，另外一部分队列则附到一个用户空间应用程序，此程序决定封包是否应该被丢弃。通过在网络栈的最底部就决定是否应该丢弃封包，需要经由内核网络子系统的封包数量大大减少了。

Cloudflare利用了Netmap工具包实现部分内核旁路。但是这个思路可以延伸为，在内核网络栈中增加一个Checkpoint，这个点应该离NIC接收到封包的时刻尽可能的近。这个Checkpoint将把封包交给用户编写的程序，决定是应该丢弃，还是继续正常处理路径。




XDP对应的BPF程序类型是：BPF_PROG_TYPE_XDP。XDP程序可以读写封包，调用助手函数解析封包、计算Checksum，这些操作都不会牵涉系统调用的开销（都在内核空间执行）。

尽管XDP的基本用途是，尽早的决定封包是否应该丢弃。但是，由于网络函数无非是读、写、转发、丢弃等原语的组合，XDP可以用来实现任何网络功能。

XDP的主要优势包括：

1. 可以使用各种内核基础设施，例如路由表、套接字、网络栈

2. 运行在内核中，使用和内核其它部分一致的安全模型
3. 运行在内核中，不需要跨越用户/内核空间边界，能够灵活的转发封包给其它内核实体，例如命名空间、网络栈
4. 支持动态替换XDP程序，不会引起网络中断
5. 保证封包的线性（linearly）布局，封包位于单个DMAed内存页中，访问起来很方便
6. 保证封包有256字节可用的额外headroom，可以用于（使用助手函数`bpf_xdp_adjust_head`、`bpf_xdp_adjust_meta`）添加自定义的封装包头

从内核4.8+开始，主要发行版中XDP可用，大部分10G+网络驱动支持XDP。

# 应用场景

## DDoS缓解

XDP的高性能特征，让它非常适合实现DDoS攻击缓解，以及一般性防火墙。

## 封包转发

BPF程序可以对封包进行任意的修改，甚至是通过助手函数任意的增减headroom大小实现封装/解封装。

处理完的封包通过XDP_REDIRECT动作即可转发封包给其它NIC，或者转发给其它CPU（利用BPF的cpumap）

## 负载均衡

使用XDP_TX动作，hairpinned LB可以将修改后的封包从接收它的网卡发送回去。

## 流量采样和监控

XDP支持将部分或截断的封包内容存放到无锁的per-CPU的内存映射ring buffer中。此ring buffer由Linux perf基础设施提供，可以被用户空间访问。

# 编程接口

## xdp_buff

在XDP中，代表当前封包的结构是：

```
struct xdp_buff {
    // 内存页中，封包数据的开始点指针
    void *data;
    // 内存页中，封包数据的结束点指针
    void *data_end;
    // 最初和和data指向同一位置。后续可以被bpf_xdp_adjust_meta()调整，向data_hard_start方向移动
    // 可以用于为元数据提供空间。这种元数据对于正常的内核网络栈是不可见的，但是能够被tc BPF程序读取，
    // 因为元数据会从XDP传送到skb中
    // data_meta可以仅仅适用于在尾调用之间传递信息，类似于可被tc访问的skb->cb[]
    void *data_meta;
    // XDP支持headroom，这个字段给出页中，此封包可以使用的，最小的地址
    // 如果封包被封装，则需要调用bpf_xdp_adjust_head()，将data向data_hard_start方向移动
    // 解封装时，也可以使用bpf_xdp_adjust_head()移动指针
    void *data_hard_start;
    // 提供一些额外的per receive queue元数据，这些元数据在ring setup time生成
```

```
    struct xdp_rxq_info *rxq;
};

// 接收队列信息
struct xdp_rxq_info {
    struct net_device *dev;
    u32 queue_index;
    u32 reg_state;
} ____cacheline_aligned; // 缓存线（默认一般是64KB），CPU以缓存线为单位读取内存到CPU高速缓存
```

它通过BPF context传递给XDP程序。

## xdp_action

```
enum xdp_action {
    // 提示BPF出现错误，和DROP的区别仅仅是会发送一个trace_xdp_exception追踪点
    XDP_ABORTED = 0,
    // 应当在驱动层丢弃封包，不必再浪费额外资源。对于DDos缓和、一般性防火墙很有用
    XDP_DROP,
    // 允许封包通过，进入网络栈进行常规处理
    // 处理此封包的CPU后续将分配skb，将封包信息填充进去，然后传递给GRO引擎
    XDP_PASS,
    // 将封包从接收到的网络接口发送回去，可用于实现hairpinned LB
    XDP_TX,
    // 重定向封包给另外一个NIC
    XDP_REDIRECT,
};
```

这个枚举是XDP程序需要返回的断言，告知驱动应该如何处理封包。

# tc

关于tc的基础知识，参考基于tc的网络QoS管理。

## tc程序简介

BPF可以和内核的tc层一起工作。tc程序和XDP程序有以下不同：

1. tc程序的BPF输入上下文是skb_buff，而非xdp_buff。在XDP之后，内核会解析封包，存入skb_buff。解析的开销导致tc程序的性能远低于XDP，但是，tc程序可以访问skb的mark, pkt_type, protocol, priority, queue_mapping, napi_id, cb[]数组, hash, tc_classid, tc_index等字段，以及VLAN元数据、XDP传送来的自定义元数据。BPF上下文`struct __sk_buff`定义在`linux/bpf.h`
2. tc程序可以挂钩到ingress/egress网络路径上，XDP则仅仅能挂钩到ingress路径
3. tc程序对驱动层没有依赖，可以挂钩到任何类型的网络设备。除非启用tc BPF程序的offloading

尽管tc程序的挂钩点没有XDP那么早，但是仍然是**在内核网络路径的早期。它在GRO运行之后，任何协议处理之前执行。**iptables PREROUTING、nftables ingress hook等工具也在相同的位置挂钩。

## tc程序工作方式

工作在tc层的BPF程序，是从一个名为`cls_bpf`的过滤器运行的。tc程序**不但可以读取skb的元数据、封包内容，还能够对封包进行任意修改，甚至使用action verdict终止tc处理过程。**

过滤器**cls_bpf可以挂钩1-N个BPF程序**，当有多个BPF程序情况下，前面的程序返回verdict `TC_ACT_UNSPEC` 会导致继续执行后面的BPF程序。使用多个BPF程序的缺点是，需要反复解析封包，导致性能降低。

cls_bpf有一个direct-action（da）模式，这样BPF程序能够直接返回action verdict，决定封包命运，结束tc处理流水线。

tc BPF程序也支持在运行时动态更新，而不会中断任何网络流量。

cls_bpf可以挂钩到的ingress/egress钩子，均被一个伪（不在qdisc树形结构中）排队规则 `sch_clsact` 管理。对于ingress qdisc来说，sche_clsact可以作为一个drop-in的替代品。对于在 `__dev_queue_xmit()` 中执行的egress钩子，需要强调，sche_clsact不在内核的root qdisc锁下运行。因此，**不管是ingress/egress，使用sche_clsact时tc BPF程序都是以无锁方式执行的，这和典型的qdisc完全不同。**此外需要注意，**sch_clsact执行期间不会发生抢占。**

典型情况下，egress方向会有附到网络设备的qdisc，例如sch_htb、sch_fq，它们其中有些是classful qdisc。classful qdisc会通过 `tcf_classify()` 调用分类器，cls_bpf也可以被挂到这种qdisc上。这时，BPF程序在root qdisc下运行，可能面临锁争用问题。

为了达到最大性能（减少锁争用），可以考虑这样的用法：使用sch_clsact + cls_bpf，在root qdisc锁之外，完成任务繁重的封包分类工作，并且设置skb->mark或skb->priority。然后，由运行在root qdisc锁下的sch_htb快速的根据skb字段完成分类、塑形操作。

sch_clsact + cls_bpf组合使用时，如果cls_bpf是da模式、只包含单个BPF程序、且位于ingress网络路径，则支持offload给智能网卡。

# 编程接口

## __sk_buff

在tc BPF程序中，代表当前封包的结构是 `__sk_buff`，这种结构叫UAPI（user space API of the kernel），可以访问内核 `sk_buff` 结构的某些字段。

```
struct __sk_buff {
    __u32 len;
    __u32 pkt_type;
    __u32 mark;
    __u32 queue_mapping;
    __u32 protocol;
    __u32 vlan_present;
    __u32 vlan_tci;
    __u32 vlan_proto;
    __u32 priority;
    __u32 ingress_ifindex;
    __u32 ifindex;
    __u32 tc_index;
    __u32 cb[5];
    __u32 hash;
    __u32 tc_classid;
    __u32 data;
    __u32 data_end;
    __u32 napi_id;

    /* Accessed by BPF_PROG_TYPE_sk_skb types from here to ... */
    __u32 family;
    __u32 remote_ip4;    /* Stored in network byte order */
    __u32 local_ip4;    /* Stored in network byte order */
```

```
    __u32 remote_ip6[4];    /* Stored in network byte order */
    __u32 local_ip6[4];     /* Stored in network byte order */
    __u32 remote_port;      /* Stored in network byte order */
    __u32 local_port;       /* stored in host byte order */
    /* ... here. */

    __u32 data_meta;
    __bpf_md_ptr(struct bpf_flow_keys *, flow_keys);
    __u64 tstamp;
    __u32 wire_len;
    __u32 gso_segs;
    __bpf_md_ptr(struct bpf_sock *, sk);
};
```

## verdicts

tc ingress/egress钩子能够返回的verdict定义在：

```
// 未指定，如果有多个BPF程序，会继续运行下一个。如果没有更多BPF程序
// 则提示内核在没有任何side-effect的情况下继续处理skb
#define TC_ACT_UNSPEC         (-1)
// 从tc BPF程序角度，TC_ACT_OK、TC_ACT_RECLASSIFY等价
#define TC_ACT_OK               0
// 提示内核丢弃封包，在ingress方向，网络栈上层无法看到封包；在egress方向，封包不会被发出
#define TC_ACT_SHOT             2
// 从tc BPF程序角度，TC_ACT_STOLEN、TC_ACT_QUEUED、TC_ACT_TRAP等价
// 类似于TC_ACT_SHOT，区别：
//   TC_ACT_SHOT导致内核通过kfree_skb()释放封包并返回NET_XMIT_DROP作为即时的反馈
//   TC_ACT_STOLEN导致内核通过consume_skb()释放封包，并且返回NET_XMIT_SUCCESS，
//      效果是上层以为封包是成功发送的
#define TC_ACT_STOLEN           4
// 利用助手函数bpf_redirect()，重定向封包到相同/不同设备的ingress/egress路径
#define TC_ACT_REDIRECT         7
```

# 应用场景

## 容器网络策略

对于容器来说，容器网络命名空间和初始网络命名空间通过一对veth连接。我们可以在宿主机端实现网络策略：

1. 主机侧的egress，对应容器的ingress
2. 主机侧的ingress，对应容器的egress

将tc BPF程序挂钩到宿主机veth的egress/ingress钩子即可。

对于veth这样的虚拟设备，XDP是不适合的。因为内核在虚拟设备这里，单纯在一个skb上操作，**XDP由于一些限制，无法和克隆的skb一起工作。**克隆skb在内核TCP/IP栈被大量使用，用来存放重传的数据段，这里XDP钩子会被直接跳过。此外，XDP需要线性化（放到单个页）整个skb，这也导致严重的性能影响。

## 转发和负载均衡

容器工作负载的东西向流量是主要的目标场景。

不像XDP仅作用在ingress上，tc BPF可以在某些场景下，应用到容器egress方向。在对容器透明的前提下，可以利用BPF在egress进行NAT和LB处理，利用bpf_redirection助手函数，**BPF可以将封包转到任何接口的ingress/egress路径，不需要网桥之类的设备辅助。**

## 流采样和监控

类似XDP，流采样和监控可以通过高性能、无锁的per-CPU内存映射的perf ring buffer实现，依此tc BPF程序可以调用助手函数`bpf_skb_event_output()`推送定制数据、完整/截断的封包内容到用户空间。

由于tc BPF程序可以同时挂到ingress/egress，因此可以为任何节点实现双向的监控。

BPF程序可以预先做一些聚合，而不是把所有东西推送到用户空间。

## 预处理封包调度

如上文所提到，sch_clsact的egress钩子，即`sch_handle_egress()`，在获取内核qdisc root锁之前运行。这种无锁的特征让tc BPF程序适合执行较重（耗时）的分类任务，并将分类结果设置到skb的某些字段，在后续交由常规的、有锁的qdisc进行塑形和重排。

# 开发环境

# 内核和工具

本节介绍如何创建完整的BPF开发环境。尽管手工构建iproute2和Linux内核是非必须的（主流发行版已经内置），但是测试最新特性、或者需要贡献BPF补丁到内核、iproute2时则需要。

安装构建需要的软件：

```
sudo apt-get install -y make gcc libssl-dev bc libelf-dev libcap-dev \
  clang gcc-multilib llvm libncurses5-dev git pkg-config libmnl-dev bison flex \
  graphviz
```

## 构建内核

BPF相关的新特性在内核的net-next分支上开发，最后的BPF fixes则在net分支上。

注意打开以下内核配置项：

```
CONFIG_CGROUP_BPF=y
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_NET_SCH_INGRESS=m
CONFIG_NET_CLS_BPF=m
CONFIG_NET_CLS_ACT=y
# 如果目标体系结构支持JIT，自动y
CONFIG_BPF_JIT=y
CONFIG_LWTUNNEL_BPF=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF_EVENTS=y
CONFIG_TEST_BPF=m
```

从新编译的内核启动后，运行BPF自我测试套件，应该全部通过：

```
cd tools/testing/selftests/bpf/
make
sudo ./test_verifier
```

## 构建iproute2

iproute2具有独立的Git仓库：

```
git clone https://git.kernel.org/pub/scm/network/iproute2/iproute2.git
```

master分支对应内核的net分支，net-next分支对应内核的net-next分支。

执行下面的命令编译：

```
cd iproute2/
./configure --prefix=/usr
# 确保输出：
# ELF support: yes
# 这样iproute2才能处理LLVM BPF后端产生的ELF文件
sudo make install
```

## 构建bpftool

bpftools是BPF程序、Map的调试、introspection工具。它位于内核源码树的tools/bpf/bpftool/

```
cd tools/bpf/bpftool/
make
sudo make install
```

## 构建libbpf

# 工具链

LLVM 3.7+是当前唯一提供BPF后端的编译套件，GCC目前不支持。主流发行版默认启用了LLVM的BPF后端支持，因此直接安装clang和llvm包足够将C编译为BPF Object文件。

通过下面的命令确认你的LLVM支持BPF目标：

```
llc --version
LLVM (http://llvm.org/):
  LLVM version 10.0.0

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

  Registered Targets:
    ...
    # 默认情况下，bpf目标使用编译它的CPU的端序
    bpf        - BPF (host endian)
    # 这两个目标用于交叉编译
    bpfeb      - BPF (big endian)
```

```
    bpfel      - BPF (little endian)
```

## 编译命令

对于下面这个最简单的XDP Drop程序：

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

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

__section("prog")
int xdp_drop(struct xdp_md *ctx)
{
    return XDP_DROP;
}

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

可以这样编译：

```
clang -O2 -Wall -target bpf -c xdp-example.c -o xdp-example.o
# 加载，需要4.12或更高版本
# ip link set dev em1 xdp obj xdp-example.o
```

# BPF CO-RE

所谓CO-RE是指一次编译，到处运行（Compile Once – Run Everywhere）。

编写可移植性（能够跨越不同内核版本运行）的BPF程序是一项挑战，BPF CO-RE能够辅助这一过程。

## 问题

BPF程序本质上是一段直接插入内核的代码，它可以访问（被允许访问的）所有内核内部状态。BPF程序无法控制所运行内核的内存布局：

1. 不同内核版本，结构体字段的顺序可能不同，某些字段可能在新版本被内嵌到子结构中，字段的类型可能出现不兼容的变更
2. 根据内核构建时的配置不同，某些字段可能被注释掉

并不是所有BPF程序都需要访问内核数据结构，例如opensnoop，它依赖 kprobes/tracepoints 来追踪程序打开了那些文件，仅仅需要捕获一些系统调用参数，而系统调用提供稳定的ABI，不会随着内核版本而变化。

不幸的是，opensnoop这样的程序是少数派，不访问内核数据结构也限制了程序的能力。

**内核中的BPF机制提供了一个有限集合的稳定接口**，BPF程序可以依赖这套结构，不用担心跨内核的兼容性问题。尽管底层结构会发生变化，但是BPF提供的抽象保持稳定。这种接口的一个例子是`__sk_buff`，它是内核数据结构`sk_buff`的稳定接口，可以访问sk_buff的一部分重要字段。对**sk_buff字段的访问，会被透明的转写（可以需要多次pointer chasing）为对sk_buff对应字段的访问。类似**sk_buff的、针对特定类型BPF程序的接口抽象还有不少，如果你编写这些类型的BPF程序，可能（如果接口可以满足需求）不用担心可移植性问题。

一旦有了读取内核内部原始状态的需求，例如读取`struct task_struct`的某个字段，可移植性问题就出现了。如果某些内核为此结构添加了额外字段，如何保证读取的不是垃圾数据？如果出现了字段重命名，例如4.7将task_struct的fs字段重命名为fsbase，又该怎么办？这些可移植性问题都让你不能简单的使用开发机上的内核头文件来构建BPF程序。

# BCC方案

解决可移植性问题的一个方案是BCC，使用BCC时，BPF内核程序的C源码作为字符串嵌入在你的用户空间程序（控制程序）中，当控制程序被部署到目标环境后，BCC会调用内嵌的Clang/LLVM，拉取本地内核头文件，执行即席的BPF程序构建。如果字段可能在某些环境下compiled-out，你只需要在源码中使用相应的#ifdef/#else。BCC方案有一些关键的缺点：

1. Clang/LLVM组合非常大，你需要将其部署到所有运行BPF程序的机器
2. Clang/LLVM在编译时很消耗资源，如果在程序启动时编译BPF代码，可能会生产环境的工作负载产生不利影响
3. 目标机器上可能不包含匹配的内核头文件
4. 开发和测试的迭代变得痛苦，可能在运行时得到很多编译错误
5. 太多的magic，难以定位错误，你需要记住命名约定、自动为tracepoint生成的结构、依赖代码重写来读取内核数据/获取kprobe参数
6. 读写BPF Map时需要编写半面向对象的C代码，和内核中发生的不完全匹配
7. 仍然需要在用户空间编写大量的样板代码

# CO-RE原理

BPF CO-RE将软件栈所有层次 —— 内核、用户空间BPF loader库（libbpf）、编译器（Clang）——的必要功能/数据片段整合到一起，来降低编写可移植性BPF程序的难度。CO-RE需要下列组件的谨慎集成和协作：

1. BTF类型信息：允许捕获关于内核、BPF程序的类型/代码的关键信息
2. Clang为BPF程序C代码提供了express the intent和记录relocation信息的手段
3. BPF loader（libbpf）根据内核的BTF和BPF程序，调整编译后的BPF代码，使其适合在目标内核上运行
4. 对于BPF CO-RE不可知的内核，提供了一些高级的BPF特性，满足高级场景

## BTF

即BPF Type Format，类似于DWARF调试信息，但是没有那么generic和verbose。它是一种空间高效的、紧凑的、有足够表达能力的格式，足以描述C程序的所有类型信息。由于它的简单性和BPF去重算法，对比DWARF，BTF能够缩小100x的尺寸。现在，在运行时总是保留BTF信息是常见做法，它对应内核选项`CONFIG_DEBUG_INFO_BTF=y`，在Ubuntu 20.10开始默认开启。

BTF能够用来增强BPF verifier的能力，能够允许BPF代码直接访问内核内存，不需要`bpf_probe_read()`。

对于CO-RE来说，更重要的是，内核通过`/sys/kernel/btf/vmlinux`暴露了权威的、自描述的BTF信息。执行下面的命令，你可以得到一个可编译的C头文件：

```
bpftool btf dump file /sys/kernel/btf/vmlinux format c
```

此文件通常命名为`vmlinux.h`，其中包含了所有的内核类型信息，甚至包含那些不会通过kernel-devel包暴露的信息。

## 编译器支持

为了启用CO-RE，并且让BPF loader（libbpf）来为正在运行的（目标）内核调整BPF程序，Clang被扩展，增加了一些built-ins。

这些built-ins会发出（emit）BTF relocations，BTF relocations是BPF程序需要读取什么信息的高层描述。假设程序需要访问task_struct->pid，Clang会将其记录：需要访问pid_t类型的、名为pid、位于task_struct结构中的字段。这样，即使字段顺序调整，甚至pid字段被放入一个内嵌的匿名结构体/联合体中，BPF程序仍然能够正确访问到pid字段。

能够捕获（进而重定位）的信息不单单是字段偏移量，还包括字段是否存在、字段的size。甚至对于位域（bitfield）字段，也能够捕获足够多的信息，让对它的访问能够被重定位。

## BPF loader

BPF loader在加载程序时，会利用前述的（构建机的）内核BTF信息、Clang重定位信息，并读取当前内核的BTF信息，对BPF程序（ELF object文件）进行裁减（custom tailored） —— 解析和匹配所有类型、字段，更新字段偏移量，以及其它可重定位数据 —— 确保程序在当前内核上能够正确运行。

## 内核

要支持CO-RE，内核不需要更多的改变（除了开启CONFIG_DEBUG_INFO_BTF）。被BPF loader（libbpf）处理过的BPF程序，对于内核来说，和在本机编译的BPF程序是完全等价的。

# CO-RE现状

截至2021年，BPF CO-RE是很成熟的技术，在大量生产环境下运行。

由于引入了BPF CO-RE，超过25个BCC工具被转换为libbpf +BPF CO-RE方式编写。由于越来越多的Linux发行版（Ubuntu 20.10、RHEL 8.2+）默认开启BTF，BPF CO-RE相关工具的适用面变得越来越广，可以替代笨重的、基于python的BCC工具。

BPF CO-RE在不同BPF应用领域被广泛接受，包括追踪、性能监控、安全/审计，甚至网络BPF程序。

要使用BPF CO-RE，可以考虑从脚手架项目libbpf-bootstrap开始。

# 使用CO-RE

## 解除内核依赖

为了避免依赖于系统头文件，可以生成包含所有内核类型的vmlinux.h：

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

这样你的代码中就不需要包含各种内核头文件、也不必须安装kernel-devel包了：

```
#include <linux/sched.h>
#include <linux/fs.h>
```

由于BTF（以及DWARF）不会记录宏信息，因此某些常用的宏可能没有包含在vmlinux.h中，好在其中大部分可以通过bpf_helpers.h访问。

## 读取内核结构

使用BCC时，你可以直接访问：

```
pid_t pid = task->pid;
```

BCC会自动将其重写为对`bpf_probe_read()`的调用。

使用CO-RE的时候，由于没有BCC这种代码重写机制，为了打到同样效果，你可能需要：

1. libbpf + BPF_PROG_TYPE_TRACING：如果编写的是这类程序，你可以直接写：

```
pid_t pid = task->pid;
```

而不需要bpf_probe_read()调用。要实现可移植性，则需要将上述代码包围到`__builtin_preserve_access_index`中：

```
pid_t pid = __builtin_preserve_access_index(({ task->pid; }));
```

2. 对于其它类型BPF程序，你不能直接访问结构字段，而需要：

```
pid_t pid; bpf_probe_read(&pid, sizeof(pid), &task->pid);
```

要实现可移植性，则需要：

```
pid_t pid; bpf_core_read(&pid, sizeof(pid), &task->pid);
// 或者
bpf_probe_read(&pid, sizeof(pid), __builtin_preserve_access_index(&task->pid));
```

进行pointer chasing时，使用bpf_probe_read()/bpf_core_read()会变得痛苦：

```
u64 inode = task->mm->exe_file->f_inode->i_ino;
```

你需要逐步的分配指针临时变量，逐步读取字段，非常麻烦。 幸运的是，CO-RE提供了助手宏：

```
u64 inode = BPF_CORE_READ(task, mm, exe_file, f_inode, i_ino);

// 或者
u64 inode;
BPF_CORE_READ_INTO(&inode, task, mm, exe_file, f_inode, i_ino);
```

类似的，和`bpf_probe_read_str()`对应的CO-RE函数是`bpf_core_read_str()`，以及助手宏`BPF_CORE_READ_STR_INTO()`。

要检查字段是否在目标内核存在，可以使用`bpf_core_field_exists()`宏：

```
pid_t pid = bpf_core_field_exists(task->pid) ? BPF_CORE_READ(task, pid) : -1;
```

某些内部的、非UAPI的内核枚举值，可能跨内核版本时发生变动，甚至依赖于特定的内核配置（例如cgroup_subsys_id），这导致硬编码任何值都是不可靠的。使用Enum relocation宏`bpf_core_enum_value_exists()`和`bpf_core_enum_value()`，可以检查特定枚举值是否存在，并捕获它的值。Enum relocation重定向的一个重要用途是检测BPF助手函数是否存在，如果不存在则使用旧版本的替代物。

要捕获（如果不确定某个字段是否在别的内核版本中发生了类型变更）字段的size，可以使用`bpf_core_field_size()`：

```
u32 comm_sz = bpf_core_field_size(task->comm);
```

位域字段的读取，可以使用：

```
struct tcp_sock *s = ...;

// 读取s->is_cwnd_limited对应的位域字段
bool is_cwnd_limited = BPF_CORE_READ_BITFIELD(s, is_cwnd_limited);

// 或者
u64 is_cwnd_limited;
BPF_CORE_READ_BITFIELD_PROBED(s, is_cwnd_limited, &is_cwnd_limited);
```

## 内核版本和配置差异

某些情况下，内核之间不是简单的结构性差异：

1. 同一含义的字段可能被重命名
2. 字段的含义可能改变，例如从4.6开始，task_struct.utime/stime从原先的以jiffies为单位改为纳秒为单位

内核配置的差异，可能会出现某些内核下无法读取字段的情况。

CO-RE提供处理这些问题的辅助机制是libbpf提供的extern Kconfig variables和struct flavors。

BPF程序可以定义具有知名名称的变量，例如LINUX_KERNEL_VERSION；或者一个匹配内核Kconfig键的变量，例如CONFIG_HZ。libbpf能够自动设置这些外部变量为匹配当前内核的值，BPF verifier也会跟踪这些变量，并进行高级的流分析和dead code消除。

```
//                                声明外部Kconfig变量
extern u32 LINUX_KERNEL_VERSION __kconfig;
extern u32 CONFIG_HZ __kconfig;

u64 utime_ns;

if (LINUX_KERNEL_VERSION >= KERNEL_VERSION(4, 11, 0))
    utime_ns = BPF_CORE_READ(task, utime);
else
    /* convert jiffies to nanoseconds */
    utime_ns = BPF_CORE_READ(task, utime) * (1000000000UL / CONFIG_HZ);
```

struct flavors则用于解决内核存在不兼容类型的情况。实际上就是为不同版本的内核定义不同的结构：

```
// 新版本内核使用此结构
struct thread_struct {
    ...
```

```
    u64 fsbase;
    ...
};

// 4.6或者更低版本使用此结构
// 三下划线及其后面的部分，被认为是结构的一个flavor，flavor部分会被libbpf忽略，
// 这意味着在进行relocation时thread_struct___v46仍然对应着运行中的内核的thread_struct结构

struct thread_struct___v46 { /* ___v46 is a "flavor" part */
    ...
    u64 fs;
    ...
};

extern int LINUX_KERNEL_VERSION __kconfig;
...

struct thread_struct *thr = ...;
u64 fsbase;
if (LINUX_KERNEL_VERSION > KERNEL_VERSION(4, 6, 0))
    // 强制转型为flavor，从而抽取需要的字段
    fsbase = BPF_CORE_READ((struct thread_struct___v46 *)thr, fs);
else
    fsbase = BPF_CORE_READ(thr, fsbase);
```

如果没有struct flavors，你就不能编写可移植的BPF程序。你只能通过#ifdef条件编译为多个BPF object文件，然后在控制程序中，判断当前内核版本，然后选择加载匹配的BPF object。

## 根据用户配置修改行为

即使知道目标内核版本、配置，BPF程序可能仍然不知道如何从内核中读取需要的数据。这种情况下，可以通过用户空间的控制程序进行精确的判断，然后通过BPF Map传递一个配置信息给BPF程序，BPF程序根据此配置信息改变自己的行为。这种做法的缺点是：

1. BPF程序每次都需要读取Map的成本，对于高性能BPF程序不可忽略
2. 配置信息即使在BPF程序启动后是不可变的（没有代码去改它），但是对于BPF verifier来说，仍然是一个黑盒。BPF verifier不能根据配置信息来裁减掉dead code，或者进行其它高级的代码分析。这样，为新版本内核（假设这个版本引入了新的助手函数）编写的分支，在旧版本内核上无法被裁减，从而可能破坏程序（无法通过校验，以为助手函数不存在）

解决上述缺点的方法是使用只读全局变量。变量的值由控制程序加载BPF object文件后设置（修改ELF文件）。这不会带来Map查询的成本，BPF verifer会将其此变量作为常量看待，从而裁减dead code。

```
/* global read-only variables, set up by control app */
const bool use_fancy_helper;
const u32 fallback_value;

...

u32 value;
if (use_fancy_helper)
    value = bpf_fancy_helper(ctx);
else
    value = bpf_default_helper(ctx) * fallback_value;
```

通过BPF skeleton，可以很容易的从用户空间修改ELF文件。

### 编译BPF程序

利用上文提到的BPF CO-RE提供的多种能力，编写好代码后，你需要用Clang 10+版本，编译得到BPF object文件。

### 生成BPF skeleton

从编译好的BPF object文件，可以利用`bpftool gen skeleton`自动生成BPF skeleton。

### 编写控制程序

将BPF skeleton（头文件）包含到你的用户空间控制程序中，获得打开、加载、挂钩BPF程序，以及修改BPF对象等能力。

# eBPF编程

## C编程要点

使用C语言编写eBPF程序，需要注意：

1. 可以访问助手函数、上下文对象
2. 程序的入口点通过段来指定，而非main函数
3. 在对象文件中包含多个入口点是允许的
4. 所有库函数调用被内联，因而运行时不存在函数调用的开销
5. 没有全局变量（5.5-）
6. 没有循环
7. 没有常量
8. LLVM的内置函数一般是可用的、并且被内联
9. 栈空间大小限制为512字节

### 内联一切

除非使用支持BPF-BPF调用的4.16+内核（和LLVM6.0+），所有函数都需要被内联，没有函数调用（老版本LLVM上）或共享库调用。

BPF程序不能使用共享库，公共代码可以放在头文件中，被主程序include。尽管不能使用共享库，但是通过include头文件来使用静态内联函数、宏定义是很常见的。

为了确保这一点，需要为所有作为库使用的函数标注__inline：

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

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

#ifndef __inline
# define __inline                         \
   // 使用always_inline，因为仅仅使用inline编译器仍然可能在
   // 代码过大的情况下不内联，从而导致在ELF文件中生成一个relocation entry
   // iproute2这样的ELF loader不能解析BPF程序
   // 仅仅BPF Map是ELF loader能够处理的relication entry
   inline __attribute__((always_inline))
```

```
#endif

static __inline int foo(void)
{
    return XDP_DROP;
}

__section("prog")
int xdp_drop(struct xdp_md *ctx)
{
    return foo();
}

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

从4.16开始支持在BPF程序中使用BPF-BPF函数调用，libbpf v0.2+也对此特性提供的完整的支持，确保代码的relocations/adjustment正确进行。你可以去掉`__always_inline`，甚至用`__noinline`强制不得进行内联。

非内联的global函数从5.5+开始被支持，但是和static函数比起来具有不同的语义以及校验约束。

## 每个程序一个段

BPF的C程序依赖段（section）注解。典型的C程序被结构化为三个或更多的段，BPF ELF loader**通过段的名字来抽取、准备相关的信息**，以便载入程序和Map。

例如，iproute2使用maps和license作为默认的段名称，来获取创建Map所需元数据，以及BPF程序的License信息。加载时，License信息也被推入内核，这样某些仅仅在GPL协议下暴露的函数（例如bpf_ktime_get_ns和bpf_probe_read）允许被调用，确保BPF程序的License兼容性。

其它的段名，都专用于BPF程序代码。下面的代码使用了ingress/egress两个段，可以被tc加载并挂钩到网络设备的ingress/egress钩子：

```
#include <linux/bpf.h>
#include <linux/pkt_cls.h>
#include <stdint.h>
#include <iproute2/bpf_elf.h>

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

#ifndef __inline
# define __inline                         \
   inline __attribute__((always_inline))
#endif

// 共享的Map是全局变量，因此访问它的时候需要同步
#ifndef lock_xadd
# define lock_xadd(ptr, val)              \
   ((void)__sync_fetch_and_add(ptr, val))
#endif

// 这个宏用于将BPF助手函数映射到C代码
// 函数map_lookup_elem被映射到定义uapi/linux/bpf.h在中的枚举值
BPF_FUNC_map_lookup_elem
```

```
// 在载入内核后，Verifier会检查传入的参数是否为期望的类型，并且将对助手函数的调用指向真实函数的
调用
#ifndef BPF_FUNC
# define BPF_FUNC(NAME, ...)              \
   (*NAME)(__VA_ARGS__) = (void *)BPF_FUNC_##NAME
#endif

static void *BPF_FUNC(map_lookup_elem, void *map, const void *key);
// static void (*map_lookup_elem)(void *map, const void *key) = (void
*)BPF_FUNC_map_lookup_elem;




// 这个是共享的Map，类型struct bpf_elf_map是iproute2定义的
// iproute2提供了公共的BPF ELF loader，因此struct bpf_elf_map对于XDP和tc程序来说是一样
的
//                          必须放在maps段，这样loader才能发现
//                          可以定义多个Map，都必须放在maps段
struct bpf_elf_map acc_map __section("maps") = {
    .type           = BPF_MAP_TYPE_ARRAY,
    .size_key       = sizeof(uint32_t),
    .size_value     = sizeof(uint32_t),
    // 该Map被Pin到PIN_GLOBAL_NS，这意味着Map将被tc钉为BPF伪文件系统中的位于
    // /sys/fs/bpf/tc/globals/目录下的节点。对于此acc_map，节点路径为
    // /sys/fs/bpf/tc/globals/acc_map
    // global是跨越多个Object文件的全局命名空间。如果不同BPF程序中均有名为acc_map
    // 的Map映射到PIN_GLOBAL_NS，这这些程序会共享统一Map。仅仅第一个载入的BPF程序会触发
    // Map的创建，后续载入的程序直接使用

    // 如果取值PIN_NONE则不会映射为BPF文件系统中的节点，当tc退出后，无法从用户空间访问Map
    .pinning        = PIN_GLOBAL_NS,
    .max_elem       = 2,
};

// 这个是共享的内联函数
static __inline int account_data(struct __sk_buff *skb, uint32_t dir)
{
    uint32_t *bytes;
    // 将Map传递给助手函数
    bytes = map_lookup_elem(&acc_map, &dir);
    if (bytes)
            lock_xadd(bytes, skb->len);

    return TC_ACT_OK;
}

// 两个段，都会调用account_data往Map中写入数据
__section("ingress")
int tc_ingress(struct __sk_buff *skb)
{
    return account_data(skb, 0);
}

__section("egress")
int tc_egress(struct __sk_buff *skb)
{
    return account_data(skb, 1);
```

```
}

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

使用下面的命令编译：

```
clang -O2 -Wall -target bpf -c tc-example.c -o tc-example.o
```

利用tc加载该程序：

```
tc qdisc add dev eth0 clsact
tc filter add dev eth0 ingress bpf da obj tc-example.o sec ingress
tc filter add dev eth0 egress bpf da obj tc-example.o sec egress

tc filter show dev eth0 ingress
# filter protocol all pref 49152 bpf
# filter protocol all pref 49152 bpf handle 0x1 tc-example.o:[ingress] direct-
action id 1 tag c5f7825e5dac396f

tc filter show dev em1 egress
# filter protocol all pref 49152 bpf
# filter protocol all pref 49152 bpf handle 0x1 tc-example.o:[egress] direct-
action id 2 tag b2fd5adc0f262714

mount | grep bpf
# sysfs on /sys/fs/bpf type sysfs (rw,nosuid,nodev,noexec,relatime,seclabel)
# bpf on /sys/fs/bpf type bpf (rw,relatime,mode=0700)

tree /sys/fs/bpf/
# /sys/fs/bpf/
# +-- ip -> /sys/fs/bpf/tc/
# +-- tc
# |   +-- globals
# |       +-- acc_map
# +-- xdp -> /sys/fs/bpf/tc/
```

一旦有封包通过eth0接口，则acc_map中的计数器值就会增加。

## 没有全局变量

除非内核版本在5.5+以上，BPF程序中没有普通C程序中的全局变量。5.5+的全局变量底层仍然是基于BPF Map实现的。

作为变通方案，可以使用BPF_MAP_TYPE_PERCPU_ARRAY类型的Map，这种Map为每个CPU核心存储一个任意大小的值。由于**BPF程序在运行过程中绝不会被抢占**，在此Map中初始化一个临时的缓冲区（例如为了突破栈的大小限制）用作全局变量是安全的。在**发生尾调用的情况下也不会发生抢占，Map的内容不会消失。**

对于任何需要跨越多次BPF程序运行保存的状态，都使用普通BPF Map即可。

## 没有常量字符串或数组

由于一切内联、不支持全局变量，定义`const`的字符串或其它数组都是不被支持的，生成在ELF文件中的relocation entry会被BPF ELF loaders拒绝。

打印调试消息可以利用助手函数trace_printk：

```
static void BPF_FUNC(trace_printk, const char *fmt, int fmt_size, ...);

#ifndef printk
# define printk(fmt, ...)                                       \
    ({                                                          \
        char ____fmt[] = fmt;                                   \
        trace_printk(____fmt, sizeof(____fmt), ##__VA_ARGS__); \
    })
#endif


// 调用
printk("skb len:%u\n", skb->len);

// 在用于空间使用下面的命令查看打印的消息
// tc exec bpf dbg
```

不建议在生产环境使用trace_printk()助手函数，因为类似于`"skb len:%u\n"`这样的字符串必须在每次调用时载入到BPF Stack，此外助手函数最多支持5个参数。

对于网络应用，可以使用`skb_event_output()`或者`xdp_event_output()`代替，它们允许从BPF程序传递自定义的结构体，外加一个可选的packet sample到perf event ring buffer。

## 使用LLVM内置函数

除了助手函数之外，BPF程序不能发起任何函数调用，因此公共库代码需要实现为内联函数。此外LLVM提供的一些builtins可以被使用，并且总是保证被内联：

```
#ifndef memset
# define memset(dest, chr, n)   __builtin_memset((dest), (chr), (n))
#endif

#ifndef memcpy
# define memcpy(dest, src, n)   __builtin_memcpy((dest), (src), (n))
#endif

#ifndef memmove
# define memmove(dest, src, n)  __builtin_memmove((dest), (src), (n))
#endif
```

内置函数memcmp()存在一些边缘场景，会导致不发生inline，因此在LLVM解决此问题之前不推荐使用。

## 尚不支持循环

BPF Verifier会检查程序代码，确保其不包含循环，目的是确保程序总是能停止。

只有非常特殊形式的循环被允许：

```
// 使用该指令，内核5.3+不再需要
#pragma unroll
    // 循环的upper buounds是常量
    for (i = 0; i < IPV6_MAX_HEADERS; i++) {
        // ...
    }
```

另外一种比较刁钻的实现循环的方式是，自我尾调用， 使用一个BPF_MAP_TYPE_PERCPU_ARRAY作为变量存储。这种方式的循环次数是动态的，但是最多迭代34次（初始程序，加上最多33次尾调用）。

## 使用尾调用

尾调用提供了一种灵活的、在运行时原子的修改程序行为的方式，它的做法是从一个BPF程序跳转到另外一个，同时保留当前栈帧。尾调用没有return的概念，当前程序直接被替换掉。

发起尾调用时必须使用BPF_MAP_TYPE_PROG_ARRAY类型的Map，传递目标BPF程序所在索引。

使用尾调用可以非常灵活的对程序进行“分区”。例如挂钩到XDP或tc的根BPF程序可以发起对索引为0的BPF程序的尾调用，后者执行流量采样，然后跳转到BPF程序1，在此应用防火墙策略。封包在此被丢弃，或进一步尾调用2来处理，BPF程序2修改封包并将其从网络接口发出。

```
#ifndef __stringify
# define __stringify(X)   #X
#endif

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

#ifndef __section_tail
# define __section_tail(ID, KEY)          \
   __section(__stringify(ID) "/" __stringify(KEY))
#endif

#ifndef BPF_FUNC
# define BPF_FUNC(NAME, ...)              \
   (*NAME)(__VA_ARGS__) = (void *)BPF_FUNC_##NAME
#endif

#define BPF_JMP_MAP_ID   1

static void BPF_FUNC(tail_call, struct __sk_buff *skb, void *map,
                     uint32_t index);

// 创建一个eBPF程序数组并且钉到BPF文件系统的全局命名空间下的/jmp_map节点
struct bpf_elf_map jmp_map __section("maps") = {
    .type           = BPF_MAP_TYPE_PROG_ARRAY,
    .id             = BPF_JMP_MAP_ID,
    .size_key       = sizeof(uint32_t),
    .size_value     = sizeof(uint32_t),
```

```
    .pinning        = PIN_GLOBAL_NS,
    .max_elem       = 1,
};

// iproute2的BPF ELF loader能够识别标记为__section_tail()的块，将其存放到某个程序数组中
// 第一个参数ID用于决定存放到哪个数组，第二个参数用于决定存放到数组的哪个索引
// 不仅仅是tc，任何iproute2支持的BPF程序类型（XDP，lwt等）都可以使用这种标记
__section_tail(BPF_JMP_MAP_ID, 0)
int looper(struct __sk_buff *skb)
{
    printk("skb cb: %u\n", skb->cb[0]++);
    tail_call(skb, &jmp_map, 0);
    return TC_ACT_OK;
}

// 主程序
__section("prog")
int entry(struct __sk_buff *skb)
{
    skb->cb[0] = 0;
    // 发起尾调用
    tail_call(skb, &jmp_map, 0);
    return TC_ACT_OK;
}

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

钉在BPF伪文件系统的BPF程序数组，可被用户空间程序查询或修改，tc也提供了更新BPF程序的命令：

```
#                   更换globals/jmp_map的 0索引元素
#                                              用new.o的 foo段代替
tc exec bpf graft m:globals/jmp_map key 0 obj new.o sec foo
```

## 受限的栈空间

BPF程序的栈空间仅有512字节，因此编码时需要小心。要使用一个较大的缓冲区，可以从BPF_MAP_TYPE_PERCPU_ARRAY类型的Map分配。

## 去除字节对齐补白

现代编译器默认情况下会进行字节边界对齐 —— 结构体成员被对齐到是它们长度的整数倍的内存边界，空出的部分自动补白：

```
struct called_info {
    u64 start;  // 8-byte
    u64 end;    // 8-byte
    u32 sector; // 4-byte
}; // size of 20-byte ?

printf("size of %d-byte\n", sizeof(struct called_info)); // size of 24-byte

// Actual compiled composition of struct called_info
// 0x0(0)                    0x8(8)
//  ↓_________________________↓
//  |        start (8)        |
//  |_________________________|
```

```
//  |         end  (8)       |
//  |________________________|
//  |  sector(4) |  PADDING  | <= address aligned to 8
//  |____________|___________|     with 4-byte PADDING.
```

由于字节对齐的原因，结构体的大小通常比期望的大。

BPF Verifier会检查栈的边界，确保程序不会越界（512字节）访问，或者访问未初始化的栈区域。使用带补白的结构作为Map的值，可能导致在`bpf_prog_load()`时报invalid indirect read from stack错。

你需要使用pack指令移除补白：

```
#pragma pack(4)
struct called_info {
    u64 start;  // 8-byte
    u64 end;    // 8-byte
    u32 sector; // 4-byte
}; // size of 20-byte ?

printf("size of %d-byte\n", sizeof(struct called_info)); // size of 20-byte

// Actual compiled composition of packed struct called_info
// 0x0(0)                   0x8(8)
//  |________________________|
//  |        start (8)       |
//  |________________________|
//  |         end  (8)       |
//  |________________________|
//  |  sector(4) |             <= address aligned to 4
//  |____________|                 with no PADDING.
```

移除字节对齐补白后，会导致CPU内存访问效率的降低，在某些体系结构下，不对齐的访问（unaligned access）可能被Verifier拒绝。

所以，最优的方式是人工添加仅用于实现字节对齐的pad字段：

```
struct called_info {
    u64 start;  // 8-byte
    u64 end;    // 8-byte
    u32 sector; // 4-byte
    u32 pad;    // 4-byte
}; // size of 24-byte ?

printf("size of %d-byte\n", sizeof(struct called_info)); // size of 24-byte

// Actual compiled composition of struct called_info with explicit padding
// 0x0(0)                   0x8(8)
//  |________________________|
//  |        start (8)       |
//  |________________________|
//  |         end  (8)       |
//  |________________________|
//  |  sector(4) |  pad (4)  | <= address aligned to 8
//  |____________|___________|     with explicit PADDING.
```

## 无效引用问题

诸如bpf_skb_store_bytes之类的助手函数，会导致封包的size发生变化。由于Verifier无法在运行时跟踪这种变化，因此一旦调用了这类助手函数，对数据的引用（指针）立刻会被Verifer无效化（invalidated）：

```
struct iphdr *ip4 = (struct iphdr *) skb->data + ETH_HLEN;

skb_store_bytes(skb, l3_off + offsetof(struct iphdr, saddr), &new_saddr, 4, 0);

// Verifier会拒绝下面的代码，因为此处ip4这个指针已经无效了，不能解引用
if (ip4->protocol == IPPROTO_TCP) {
    // do something
}
```

解决办法是重新获取引用：

```
struct iphdr *ip4 = (struct iphdr *) skb->data + ETH_HLEN;

skb_store_bytes(skb, l3_off + offsetof(struct iphdr, saddr), &new_saddr, 4, 0);

// 重新获取引用
ip4 = (struct iphdr *) skb->data + ETH_HLEN;

if (ip4->protocol == IPPROTO_TCP) {
    // do something
}
```

# bpftool

bpftool是内核提供（tools/bpf/bpftool/）的，主要的BPF内省（introspection）和调试工具。它支持：

1. Dump出当前加载到系统中的所有BPF程序，以及Maps
2. 列出指定程序使用的Maps
3. Dump中Map的所有键值对
4. 对Map键值对进行增删改查
5. 将Map或程序钉到BPF伪文件系统

指定bpftool操作的目标时，可以使用ID，或者目标在BPF伪文件系统中的路径。

## 查看程序和Map

列出所有程序：

```
bpftool prog
```

输出为JSON：

```
# 对于所有子命令，都支持输出JSON
bpftool prog --json --pretty
```

列出所有Map：

```
bpftool map
```

查看特定程序：

```
bpftool prog show id 406
#    程序类型为sched_cls，即BPF_PROG_TYPE_SCHED_CLS
406: sched_cls  tag e0362f5bd9163a0a
#    加载此程序的用户和时间
     loaded_at Apr 09/16:24  uid 0
#    指令序列长度为11144字节
#                    JIT编译后的映像为7721字节
#                                程序本身（不包含Map）占用空间122888字节
#                                                    使用的Maps列表
     xlated 11144B  jited 7721B  memlock 12288B  map_ids 18,20,8,5,6,14
```

## Dump程序指令

使用下面的命令可以dump出BPF程序的指令：

```
# bpftool prog dump xlated id 406
 0: (b7) r7 = 0
 1: (63) *(u32 *)(r1 +60) = r7
 2: (63) *(u32 *)(r1 +56) = r7
 3: (63) *(u32 *)(r1 +52) = r7
[...]
47: (bf) r4 = r10
48: (07) r4 += -40
49: (79) r6 = *(u64 *)(r10 -104)
50: (bf) r1 = r6
51: (18) r2 = map[id:18]                    # <-- 使用ID为18的Map
53: (b7) r5 = 32
54: (85) call bpf_skb_event_output#5656112  # <-- 调用助手函数
55: (69) r1 = *(u16 *)(r6 +192)
[...]
```

使用下面的命令可以dump出程序JIT后的汇编指令：

```
# bpftool prog dump jited id 406
 0:        push   %rbp
 1:        mov    %rsp,%rbp
 4:        sub    $0x228,%rsp
 b:        sub    $0x28,%rbp
 f:        mov    %rbx,0x0(%rbp)
13:        mov    %r13,0x8(%rbp)
17:        mov    %r14,0x10(%rbp)
1b:        mov    %r15,0x18(%rbp)
1f:        xor    %eax,%eax
21:        mov    %rax,0x20(%rbp)
25:        mov    0x80(%rdi),%r9d
```

### Dump Map

Dump整个Map：

```
bpftool map dump id 5
```

# libbpf

libbpf是一个C/C++库，作为内核的一部分进行维护，位于tools/lib/bpf目录下。内核自带的eBPF代码样例均依赖于此库。libbpf提供了一个eBPF loader，用于处理LLVM生成的ELF文件，将其载入内核。libbpf中的一部分特性源自BCC，它也包含了一些额外的功能，例如全局变量、BPF Skeletons。

# BPF系统调用

为了支持eBPF相关操作，例如载入eBPF程序、挂钩到特定事件、创建和访问eBPF Map，Linux中引入了一个新的系统调用`bpf`。

该系统调用的签名如下：

```
//      命令       用于内核和用户空间的数据交互   attr的字节数
int bpf(int cmd, union bpf_attr *attr,      unsigned int size);
// cmd有很多，要么和eBPF程序交互、要么和eBPF Map交互，或者同时和两者交互
```

### BPF_PROG_LOAD

该命令用于载入eBPF程序。载入的时候需要指明程序的类型。程序类型决定了以下事项：

1. 程序在何处挂钩
2. 校验器允许程序调用哪些助手函数
3. 是否允许直接访问网络封包数据
4. 传递给程序的**第一个参数的对象的类型**

可以看到，程序类型规定了eBPF程序的API接口。某些时候，定义一个新的程序类型，仅仅是为了限制可调用函数的列表，例如BPF_PROG_TYPE_CGROUP_SKB、BPF_PROG_TYPE_SOCKET_FILTER

### BPF_MAP_CREATE

用于创建eBPF Maps，参考下文。

# BPF程序类型

### BPF_PROG_TYPE_SOCKET_FILTER

网络封包过滤器

### BPF_PROG_TYPE_KPROBE

挂钩到一个KProbe，BPF程序在某个内核函数被调用时触发

### BPF_PROG_TYPE_SCHED_CLS

网络流量控制（TC）的分类器（classifier）

### BPF_PROG_TYPE_SCHED_ACT

网络流量控制（TC）动作（action）

### BPF_PROG_TYPE_TRACEPOINT

挂钩到一个Tracepoint，当执行到内核特定的代码路径时触发BPF程序

### BPF_PROG_TYPE_XDP

在设备驱动接收路径上运行的网络封包过滤器

### BPF_PROG_TYPE_PERF_EVENT

决定是否应当触发一个Perf Event Handler

### BPF_PROG_TYPE_CGROUP_SKB

为控制组提供的网络封包过滤器

### BPF_PROG_TYPE_CGROUP_SOCK

同上，但是允许修改套接字选项

### BPF_PROG_TYPE_LWT_*

用于轻量级隧道（lightweight tunnels ）的网络封包过滤器

### BPF_PROG_TYPE_SOCK_OPS

用于设置套接字选项

### BPF_PROG_TYPE_SK_SKB

用于在套接字之间转发封包

### BPF_PROG_CGROUP_DEVICE

决定是否允许一个设备操作

## BPF Map

部分Map专供特定的助手函数使用，以实现特殊任务。

### 定义Map

要在BPF程序中定义一个Map，需要用到如下结构：

```
struct bpf_map_def {
        unsigned int type; // Map类型
        unsigned int key_size; // 键的长度
        unsigned int value_size; // 值的长度
        unsigned int max_entries; // 最大键值对数量
        unsigned int map_flags; // 标记
        unsigned int inner_map_idx;
        unsigned int numa_node;
};
```

下面是一个例子：

```
struct bpf_map_def SEC("maps") my_map = {
        .type = BPF_MAP_TYPE_ARRAY,
        .key_size = sizeof(int),
        .value_size = sizeof(u64),
        .max_entries = MAX_CPU,
};
```

在加载阶段，`bpf_load.c`会扫描BPF object的ELF头以发现Map定义，并调用tools/lib/bpf/bpf.c中的`bpf_create_map_node()`或`bpf_create_map_in_map_node()`来创建Map。这两个函数实际上是调用`bpf`系统调用的`BPF_MAP_CREATE`命令。

除非你在编写lwt或tc等类型的BPF程序，你都应该使用上面的方式来定义Map。tc之类的程序使用iproute2作为loader，可以使用下面的结构来定义Map：

```
#define PIN_GLOBAL_NS           2

struct bpf_elf_map {
        __u32 type;
        __u32 size_key;
        __u32 size_value;
        __u32 max_elem;
        __u32 flags;
        __u32 id;
        __u32 pinning;
};

struct bpf_elf_map SEC("maps") tun_iface = {
        .type = BPF_MAP_TYPE_ARRAY,
        .size_key = sizeof(int),
        .size_value = sizeof(int),
        .pinning = PIN_GLOBAL_NS,
        .max_elem = 1,

};
```

## 钉住Map

所谓Pinning是指通过文件系统路径来暴露Map。对于tc之类的iproute2加载的程序，可以使用这些宏：

```
#define PIN_NONE        0
#define PIN_OBJECT_NS        1
// 钉到/sys/fs/bpf/tc/globals/
#define PIN_GLOBAL_NS        2
```

其它程序，你可以手工调用libbpf钉住Map：`bpf_obj_pin(fd, path)`。其它程序可以调用`mapfd = bpf_obj_get(pinned_file_path);`获得Map的文件描述符。

## 操控Map

检查头文件`linux/bpf_types.h`你会发现，不同的Map的操作，是由`bpf_map_ops`结构所引用的不同函数指针实现的。

```
BPF_MAP_TYPE(BPF_MAP_TYPE_ARRAY, array_map_ops)
BPF_MAP_TYPE(BPF_MAP_TYPE_PERCPU_ARRAY, percpu_array_map_ops)
```

不过对于BPF开发者来说，所有Map都可以在eBPF或用户空间程序中，通过`bpf_map_lookup_elem()`和`bpf_map_update_elem()`函数访问。

# Map类型

所有的Map类型定义在枚举`bpf_map_type`中。

## BPF_MAP_TYPE_HASH

哈希表。

## BPF_MAP_TYPE_ARRAY

Array Map，为快速查找优化。

键是数组索引值（4字节，64bit），不支持删除键值。所有其它Array Map都具有此特征。

## BPF_MAP_TYPE_PROG_ARRAY

存放对应eBPF程序的文件描述符的数组。用于实现`bpf_tail_call()`需要的跳转表（jump tables）。

## BPF_MAP_TYPE_PERCPU_ARRAY

per-CPU Array Map，也就是每个CPU对应一个值。**键仍然是数字，值则是和CPU个数相同的数组**：

```
long values[nr_cpus];
ret = bpf_map_lookup_elem(map_fd, &next_key, values);
if (ret) {
    perror("Error looking up stat");
    continue;
}
for (i = 0; i < nr_cpus; i++) {
    sum += values[i];
}
```

此Map可用于代替栈上变量，分配大的缓冲区，以解决栈空间仅512字节的问题。亦可用作全局变量，以解决较旧版本内核中BPF程序没有原生全局变量支持的问题

由于值是per-CPU的，而执行中的BPF程序不会被抢占。因此只要正确编码（仅访问当前CPU的值），就不会产生竞态条件。对于**会被频繁执行的代码路径，一般会考虑per-CPU的Map。**

## BPF_MAP_TYPE_PERF_EVENT_ARRAY

即perfbuf，per-CPU的缓冲区。在内核空间，供`bpf_perf_event_output()`函数使用，调用该函数，可以输出指定类型的结构到缓冲区。

在用户空间，可以进行epoll，当有数据输出时会得到通知。

## BPF_MAP_TYPE_RINGBUF

即ringbuf，perfbuf的继任，所有CPU共享的一个环形缓冲区。

## BPF_MAP_TYPE_CGROUP_ARRAY

在用户空间，存放cgroup的文件描述符。

在内核空间，调用`bpf_skb_under_cgroup()`来检查skb是否和Map中指定索引的cgroup关联。

## BPF_MAP_TYPE_PERCPU_HASH

per-CPU的哈希表

## BPF_MAP_TYPE_LRU_HASH

使用LRU算法的哈希表

## BPF_MAP_TYPE_LRU_PERCPU_HASH

per-CPU的使用LRU算法的哈希表

## BPF_MAP_TYPE_LPM_TRIE

最长前缀匹配的字典树（trie），可用于IP地址范围匹配。

## BPF_MAP_TYPE_STACK_TRACE

存储栈追踪信息。

## BPF_MAP_TYPE_ARRAY_OF_MAPS

Map的数组。

## BPF_MAP_TYPE_HASH_OF_MAPS

Map的Map。

## BPF_MAP_TYPE_DEVICE_MAP

存储和查找网络设备引用。

### BPF_MAP_TYPE_SOCKMAP

存储和查找套接字，允许基于助手函数，实现套接字重定向。

## 助手函数

助手函数由libbpf库提供，定义在`bpf_helpers.h`中。

### bpf_map_lookup_elem

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

查找eBPF中和一个Key关联的条目。如果找不到条目，返回NULL

### bpf_map_update_elem

签名：`int bpf_map_update_elem(struct bpf_map *map, const void *key, const void *value, u64 flags)`

添加或修改一个Key关联的值。flags可以是以下之一：

1. BPF_NOEXIST 键值必须不存在，即执行添加操作。不能和BPF_MAP_TYPE_ARRAY、BPF_MAP_TYPE_PERCPU_ARRAY联用
2. BPF_EXIST，键值必须存在，即执行更新操作
3. BPF_ANY，更新或修改

### bpf_map_delete_elem

签名：`int bpf_map_delete_elem(struct bpf_map *map, const void *key)`

从eBPF Map中删除条目

### bpf_probe_read

签名：`int bpf_probe_read(void *dst, u32 size, const void *src)`

对于Tracing用途的eBPF程序，可以安全的从src读取size字节存储到dst

### bpf_ktime_get_ns

签名：`u64 bpf_ktime_get_ns(void)`

读取系统从启动到现在的纳秒数，返回ktime

### bpf_trace_printk

签名：`int bpf_trace_printk(const char *fmt, u32 fmt_size, ...)`

类似于printk()的调试工具，从DebugFS打印由fmt所定义的格式化字符串到`/sys/kernel/debug/tracing/trace`。最多支持3个额外的u64参数。

每当此函数被调用，它都会打印一行到trace，格式取决于配置`/sys/kernel/debug/tracing/trace_options`。默认格式如下：

```
# 当前任务的名字
#       当前任务的PID
#              当前CPU序号
#                     每个字符表示一个选项
#                           时间戳
#                                          BPF使用的指令寄存器的Fake值
telnet-470   [001] .N.. 419421.045894: 0x00000001: <formatted msg>
```

可以使用的格式化占位符：`%d, %i, %u, %x, %ld, %li, %lu, %lx, %lld, %lli, %llu, %llx, %p, %s`。不支持长度、补白等修饰符。

该函数比较缓慢，应该仅用于调试目的。

## bpf_get_prandom_u32

签名：`u32 bpf_get_prandom_u32(void)`

获得一个伪随机数。

## bpf_get_smp_processor_id

签名：`u32 bpf_get_smp_processor_id(void)`

得到SMP处理器ID，需要注意，所有eBPF都在禁止抢占的情况下运行，这意味着在eBPF程序的执行过程中，此ID不会改变。

## bpf_skb_store_bytes

签名：`int bpf_skb_store_bytes(struct sk_buff *skb, u32 offset, const void *from, u32 len, u64 flags)`

存储缓冲区from的len字节到，skb所关联的封包的offset位置。flags是以下位域的组合：

1. BPF_F_RECOMPUTE_CSUM：自动重新计算修改后的封包的Checksum
2. BPF_F_INVALIDATE_HASH：重置`skb->hash``skb->swhash``skb->l4hash`为0

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_skb_load_bytes

签名：`int bpf_skb_load_bytes(const struct sk_buff *skb, u32 offset, void *to, u32 len)`

从skb中的offset位置读取len长的数据，存放到to缓冲区。

从4.7开始，该函数的功能基本被直接封包访问（direct packet access）代替 —— `skb->data`和`skb->data_end`给出了封包数据的位置。如果希望一次性读取大量数据到eBPF，仍然可以使用该函数。

## bpf_l3_csum_replace

签名：`int bpf_l3_csum_replace(struct sk_buff *skb, u32 offset, u64 from, u64 to, u64 size)`

重新计算L3（IP）的Checksum。计算是增量进行的，因此助手函数必须知道被修改的头字段的前值（from）、修改后的值（to），以及被修改字段的字节数（size，2或4）。你亦可将from和size设置为0，并将字段修改前后的差存放到to。offset用于指示封包的IP Checksum的位置

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_l4_csum_replace

签名：`int bpf_l4_csum_replace(struct sk_buff *skb, u32 offset, u64 from, u64 to, u64 flags)`

重新计算L4（TCP/UDP/ICMP）的Checksum。计算是增量进行的，因此助手函数必须知道被修改的头字段的前值（from）、修改后的值（to），以及被修改字段的字节数（存放在flags的低4bit，2或4）。你亦可将from和flags低4bit设置为0，并将字段修改前后的差存放到to。offset用于指示封包的IP Checksum的位置。

flags的高位用于存放以下标记：

1. BPF_F_MARK_MANGLED_0，如果Checksum是null，则不去修改它，除非设置了BPF_F_MARK_ENFORCE
2. CSUM_MANGLED_0，对于导致Checksum为null的更新操作，设置此标记
3. BPF_F_PSEUDO_HDR，提示使用pseudo-header来计算Checksum

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_tail_call

签名：`int bpf_tail_call(void *ctx, struct bpf_map *prog_array_map, u32 index)`

这是一个特殊的助手函数，用于触发尾调用 —— 跳转到另外一个eBPF程序。新程序将使用一样的栈帧，但是被调用者不能访问调用者在栈上存储的值，以及寄存器。

使用场景包括：

1. 突破eBPF程序长度限制
2. 在不同条件下进行跳转（到子程序）

出于安全原因，可以连续执行的尾调用次数是受限制的。限制定义在内核宏MAX_TAIL_CALL_CNT中，默认32，无法被用户空间访问

当调用发生后，程序尝试跳转到prog_array_map（BPF_MAP_TYPE_PROG_ARRAY类型的Map）的index索引处的eBPF程序，并且将当前ctx传递给它。

如果调用成功，则当前程序被替换掉，不存在函数调用返回。如果调用失败，则不产生任何作用，当前程序继续运行后续指令。失败的原因包括：

1. 指定的index不存在eBPF程序
2. 当前尾调用链的长度超过限制

## bpf_clone_redirect

签名：`int bpf_clone_redirect(struct sk_buff *skb, u32 ifindex, u64 flags)`

克隆skb关联的封包，并且重定向到由ifindx所指向的网络设备。入站/出站路径都可以用于重定向。标记BPF_F_INGRESS用于确定是重定向到入站（ingress）还是出站（egress）路径，如果该标记存在则入站。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_redirect

签名：`int bpf_redirect(u32 ifindex, u64 flags)`

重定向封包到ifindex所指向的网络设备。类似于bpf_clone_redirect，但是不会进行封包克隆，因而性能较好。缺点是，redirect操作实际上是在eBPF程序返回后的某个代码路径上发生的。

除了XDP之外，入站/出站路径都可以用于重定向。标记BPF_F_INGRESS用于指定是ingress还是egress。当前XDP仅仅支持重定向到egress接口，不支持设置flag

对于XDP，成功返回XDP_REDIRECT，出错返回XDP_ABORTED。对于其它eBPF程序，成功返回TC_ACT_REDIRECT，出错返回TC_ACT_SHOT

## bpf_redirect_map

签名：`int bpf_redirect_map(struct bpf_map *map, u32 key, u64 flags)`

将封包重定向到map的key键指向的endpoint。根据map的类型，它的值可能指向：

1. 网络设备，用于转发封包到其它ports
2. CPU，用于重定向XDP帧给其它CPU，仅仅支持Native（驱动层支持的） XDP

flags必须置零。

当重定向给网络设备时，该函数比bpf_redirect性能更好。这是由一系列底层实现细节之一决定的，其中之一是该函数会以bulk方式将封包发送给设备。

如果成功返回XDP_REDIRECT，否则返回XDP_ABORTED。

## bpf_sk_redirect_map

签名：`int bpf_sk_redirect_map(struct bpf_map *map, u32 key, u64 flags)`

将封包重定向给map（类型BPF_MAP_TYPE_SOCKMAP）的key所指向的套接字。ingress/egress接口都可以用于重定向。标记BPF_F_INGRESS用于确定是不是ingress。

如果成功返回SK_PASS，否则返回SK_DROP。

## bpf_sock_map_update

签名：`int bpf_sock_map_update(struct bpf_sock_ops *skops, struct bpf_map *map, void *key, u64 flags)`

添加/更新map的条目，skopts作为key的新值。flags是以下其中之一：

1. BPF_NOEXIST，仅添加
2. BPF_EXIST，仅更新
3. BPF_ANY，添加或更新

## bpf_skb_vlan_push

签名：`int bpf_skb_vlan_push(struct sk_buff *skb, __be16 vlan_proto, u16 vlan_tci)`

将vlan_proto协议的vlan_tci（VLAN Tag控制信息）Push给skb关联的封包，并且更新Checksum。需要注意ETH_P_8021Q和ETH_P_8021AD的vlan_proto是不一样的，这里使用前者。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_skb_vlan_pop

签名：`int bpf_skb_vlan_pop(struct sk_buff *skb)`

弹出skb关联的封包的VLAN头。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_skb_get_tunnel_key

签名：`int bpf_skb_get_tunnel_key(struct sk_buff *skb, struct bpf_tunnel_key *key, u32 size, u64 flags)`

获取隧道（外层报文）的元数据，skb关联的封包的隧道元数据被填充到key，长度size。标记BPF_F_TUNINFO_IPV6提示隧道是基于IPv6而非IPv4。

`bpf_tunnel_key`是一个容器结构，它将各种隧道协议的主要参数都存入其中，这样eBPF程序可以方便的根据封装（外层）报文的头来作出各种决定。

对端的IP地址被存放在`key->remote_ipv4`或`key->remote_ipv6`

通过`key->tunnel_id`可以访问隧道的ID，通常映射到VNI（虚拟网络标识符），调用`bpf_skb_set_tunnel_key()`函数需要用到

下面这个示例用在隧道一端的TC Ingress接口，可以过滤掉对端隧道IP不是10.0.0.1的封包：

```
int ret;
struct bpf_tunnel_key key = {};

ret = bpf_skb_get_tunnel_key(skb, &key, sizeof(key), 0);
if (ret < 0)
        return TC_ACT_SHOT;     // drop packet

if (key.remote_ipv4 != 0x0a000001)
        return TC_ACT_SHOT;     // drop packet

return TC_ACT_OK;               // accept packet
```

支持VxLAN、Geneve、GRE、IPIP等类型的隧道。

## bpf_skb_set_tunnel_key

签名：`int bpf_skb_set_tunnel_key(struct sk_buff *skb, struct bpf_tunnel_key *key, u32 size, u64 flags)`

为skb关联的封包生成隧道元数据。隧道元数据被设置为长度为size的bpf_tunnel_key结构。flags是如下位域的组合：

1. BPF_F_TUNINFO_IPV6 指示隧道基于IPv6而非IPv4
2. BPF_F_ZERO_CSUM_TX 对于IPv4封包，添加一个标记到隧道元数据，提示应该跳过Checksum计算，将其置零
3. BPF_F_DONT_FRAGMENT，添加一个标记到隧道元数据，提示封包不得被分片（fragmented）
4. BPF_F_SEQ_NUMBER，添加一个标记到隧道元数据，提示发送封包之前，需要添加sequence number

示例：

```
struct bpf_tunnel_key key;
// populate key ...
bpf_skb_set_tunnel_key(skb, &key, sizeof(key), 0);
bpf_clone_redirect(skb, vxlan_dev_ifindex, 0);
```

## bpf_skb_get_tunnel_opt

签名：`int bpf_skb_get_tunnel_opt(struct sk_buff *skb, u8 *opt, u32 size)`

从skb关联的封包中获取隧道选项元数据，并且将原始的隧道选项信息存储到大小为size的opt中。

## bpf_skb_set_tunnel_opt

签名：`int bpf_skb_set_tunnel_opt(struct sk_buff *skb, u8 *opt, u32 size)`

将隧道选项元数据设置给skb关联的封包。

## bpf_skb_change_proto

签名：`int bpf_skb_change_proto(struct sk_buff *skb, __be16 proto, u64 flags)`

将skb的协议改为proto。目前仅仅支持将IPv4改为IPv6。助手函数会做好底层工作，例如修改套接字缓冲的大小。eBPF程序需要调用`skb_store_bytes`填充必要的新的报文头字段，并调用`bpf_l3_csum_replace`、`bpf_l4_csum_replace`重新计算Checksum。

该助手函数的主要意义是执行一个NAT64操作。

在内部实现上，封包的GSO（generic segmentation offload）类型标记为dodgy，因而报文头被检查，TCP分段被GSO/GRO引擎重新分段。

flags必须清零，这个参数暂时没有使用。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_csum_diff

签名：`s64 bpf_csum_diff(__be32 *from, u32 from_size, __be32 *to, u32 to_size, __wsum seed)`

计算两个缓冲区from到to的checksum difference。

## bpf_skb_change_type

签名：`int bpf_skb_change_type(struct sk_buff *skb, u32 type)`

修改封包类型，即设置`skb->pkt_type`为type。主要用途是将skb改为PACKET_HOST。type的取值：

1. PACKET_HOST 单播给本机的封包
2. PACKET_BROADCAST 广播封包
3. PACKET_MULTICAST 组播封包
4. PACKET_OTHERHOST单播给其它机器的封包

## bpf_skb_change_head

签名：`int bpf_skb_change_head(struct sk_buff *skb, u32 len, u64 flags)`

增长封包的headroom，增长len长度，调整MAC头的偏移量。如果需要，该函数会自动扩展和重新分配内存。

该函数可以用于在L3的skb上，推入一个MAC头，然后将其重定向到L2设备。

flags为保留字段，全部置空。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_skb_under_cgroup

签名：`int bpf_skb_under_cgroup(struct sk_buff *skb, struct bpf_map *map, u32 index)`

检查skb是否是由BPF_MAP_TYPE_CGROUP_ARRAY类型的Map的index位置所指向的CGroup2的descendant。

返回值：

0 ： 不是目标Cgroup2的descendant

1：是目标Cgroup2的descendant

负数：出错

## bpf_set_hash_invalid

签名：`void bpf_set_hash_invalid(struct sk_buff *skb)`

无效化`skb->hash`。在通过直接封包访问修改报文头之后调用此函数，以提示哈希值以及过期，内核下一次访问哈希或者调用bpf_get_hash_recalc时会触发哈希值的重新计算。

## bpf_get_hash_recalc

签名：`u32 bpf_get_hash_recalc(struct sk_buff *skb)`

获取封包哈希值`skb->hash`，如果该字段没有设置（特别是因为封包修改导致哈希被清空）则计算并设置哈希。后续可以直接访问skb->哈希获取哈希值。

调用bpf_set_hash_invalid()、bpf_skb_change_proto()、bpf_skb_store_bytes()+BPF_F_INVALIDATE_HASH标记，都会导致哈希值清空，并导致下一次bpf_get_hash_recalc()调用重新生成哈希值。

## bpf_set_hash

签名：`u32 bpf_set_hash(struct sk_buff *skb, u32 hash)`

设置完整哈希值到skb->hash

## bpf_skb_change_tail

签名：`int bpf_skb_change_tail(struct sk_buff *skb, u32 len, u64 flags)`

Resize(trim/grow) skb关联的封包到len长。flags必须置零。

改变封包长度后，eBPF程序可能需要调用bpf_skb_store_bytes、bpf_l3_csum_replace、bpf_l3_csum_replace等函数填充数据、重新计算Checksum。

一般用于回复ICMP控制报文。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_skb_pull_data

签名：`int bpf_skb_pull_data(struct sk_buff *skb, u32 len)`

所谓non-linear的skb，是指被fragmented的skb，即有一部分数据没有存放在skb所在内存，而是存放在其它内存页（可能有多个），并通过skb_shared_info记录这些数据位置。

当skb是non-linear的、并且不是所有len长是linear section的一部分的前提下，拉取skb的non-linear数据。确保skb的len字节是可读写的。如果len设置为0，则拉取拉取skb的整个长度的数据。

进行封包直接访问时，通过`skb->data_end`来测试某个偏移量是否在封包范围内，可能因为两个原因失败：

1. 偏移量是无效的
2. 偏移量对应的数据是在skb的non-linear部分中

该助手函数可以用来一次性拉取non-linear数据，然后再进行偏移量测试和数据访问。

此函数确保skb是uncloned，这是直接封包访问的前提。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

# bpf_get_socket_cookie

签名：`u64 bpf_get_socket_cookie(struct sk_buff *skb)`

如果skb关联到一个已知的套接字，则得到套接字的cookie（由内核生成），如果尚未设置cookie，则生成之。一旦cookie生成，在套接字的生命周期范围内都不会改变。

该助手用于监控套接字网络流量统计信息，它在网络命名空间范围内为套接字提供唯一标识。

# bpf_get_socket_uid

签名：`u32 bpf_get_socket_uid(struct sk_buff *skb)`

获得套接字的owner UID。如果套接字是NULL，或者不是full socket（time-wait状态，或者是一个request socket），则返回overflowuid，overflowuid不一定是socket的实际UID。

# bpf_csum_update

签名：`s64 bpf_csum_update(struct sk_buff *skb, __wsum csum)`

如果驱动已经为整个封包提供了Checksum，那么此函数将csum加到`skb->csum`字段上，其它情况下返回错误。

该助手函数应当和`bpf_csum_diff()`联合使用，典型场景是，通过封包直接访问修改了封包内容之后，进行Checksum更新。

# bpf_get_route_realm

签名：`u32 bpf_get_route_realm(struct sk_buff *skb)`

得到路由的Realm，也就是skb的destination的tclassid字段。这个字段是用户提供的一个tag，类似于net_cls的classid。不同的是，这里的tag关联到路由条目（destination entry）。

可以在clsact TC egress钩子中调用此函数，或者在经典的classful egress qdiscs上使用。不能在TC ingress路径上使用。

要求内核配置选项CONFIG_IP_ROUTE_CLASSID。

返回skb关联的封包的路由的realm。

## bpf_setsockopt

签名：`int bpf_setsockopt(struct bpf_sock_ops *bpf_socket, int level, int optname, char *optval, int optlen)`

针对bpf_socket关联的套接字发起一个setsockopt()操作，此套接字必须是full socket。optname为选项名，optval/optlen指定了选项值，level指定了选项的位置。

该函数实际上实现了setsockopt()的子集，支持以下level：

1. SOL_SOCKET，支持选项SO_RCVBUF, SO_SNDBUF, SO_MAX_PACING_RATE, SO_PRIORITY, SO_RCVLOWAT, SO_MARK
2. IPPROTO_TCP，支持选项TCP_CONGESTION, TCP_BPF_IW, TCP_BPF_SNDCWND_CLAMP
3. IPPROTO_IP，支持选项IP_TOS
4. IPPROTO_IPV6，支持选项IPV6_TCLASS

## bpf_skb_adjust_room

签名：`int bpf_skb_adjust_room(struct sk_buff *skb, u32 len_diff, u32 mode, u64 flags)`

增加/缩小skb关联的封包的数据的room，增量为len_diff。mode可以是：

1. BPF_ADJ_ROOM_NET，在网络层调整room，即在L3头上增加/移除room space

flags必须置零。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_xdp_adjust_head

签名：`int bpf_xdp_adjust_head(struct xdp_buff *xdp_md, int delta)`

移动`xdp_md->data`delta字节，delta可以是负数。

该函数准备用于push/pop headers的封包。

调用此助手函数会导致封包缓冲区改变，因此在加载期间校验器对指针的校验将失效，必须重新校验。

## bpf_xdp_adjust_meta

签名：`int bpf_xdp_adjust_meta(struct xdp_buff *xdp_md, int delta)`

调整 xdp_md->data_meta所指向的地址delta字节。该操作改变了存储在xdp_md->data中的地址信息。

## bpf_get_current_task

签名：`u64 bpf_get_current_task(void)`

获取当前Task结构的指针。

## bpf_get_stackid

签名：`int bpf_get_stackid(struct pt_reg *ctx, struct bpf_map *map, u64 flags)`

获取一个用户/内核栈，得到其ID。需要传入ctx，即当前追踪程序在其中执行的上下文对象，以及一个BPF_MAP_TYPE_STACK_TRACE类型的Map。通过flags指示需要跳过多少栈帧（0-255），masked with BPF_F_SKIP_FIELD_MASK。flags的其它位如下：

1. BPF_F_USER_STACK 收集用户空间的栈，而非内核栈
2. BPF_F_FAST_STACK_CMP 基于哈希来对比栈
3. BPF_F_REUSE_STACKID 如果两个不同的栈哈希到同一个stackid，丢弃旧的

## bpf_get_current_pid_tgid

签名：`u64 bpf_get_current_pid_tgid(void)`

返回一个包含了当前tgid和pid的64bit整数。值为`current_task->tgid << 32 | current_task->pid`

## bpf_get_current_uid_gid

签名：`u64 bpf_get_current_uid_gid(void)`

返回一个包含了当前GID和UID的整数。值为`current_gid << 32 | current_uid`

## bpf_get_current_comm

签名：`int bpf_get_current_comm(char *buf, u32 size_of_buf)`

将当前任务的comm属性拷贝到长度为size_of_buf的buf中。comm属性包含可执行文件的路径

调用成功时助手函数确保buf是NULL-terminated。如果失败，则填满0

## bpf_get_cgroup_classid

签名：`u32 bpf_get_cgroup_classid(struct sk_buff *skb)`

得到当前任务的classid，即skb所属的net_cls控制组的classid。该助手函数可用于TC的egress路径，不能用于ingress路径。

Linux支持两个版本的Cgroups，v1和v2，用户可以混合使用。但是，net_cls是v1特有的Cgroup。这意味着此助手函数和run on cgroups（v2 only）的eBPF程序不兼容，套接字一次仅仅能携带一个版本Cgroup的数据。

内核必须配置CONFIG_CGROUP_NET_CLASSID=y/m才能使用此助手函数。

返回classid，或者0，即默认的没有被配置的classid。

## bpf_probe_write_user

签名：`int bpf_probe_write_user(void *dst, const void *src, u32 len)`

尝试在以一个安全方式来写入src的len字节到dst中。仅仅对于运行在用户上下文的线程可用，dst必须是有效的用户空间地址。

由于TOC-TOU攻击的原因，此助手函数不得用于实现任何类型的安全机制。

此函数用于试验目的，存在的导致系统、进程崩溃的风险。当调用了此函数的eBPF程序被挂钩后，内核日志会打印一条警告信息，包含PID和进程名信息。

## bpf_probe_read_str

签名：`int bpf_probe_read_str(void *dst, int size, const void *unsafe_ptr)`

从unsafe_ptr拷贝一个NULL结尾的字符串到dst，size包含结尾的NULL字符。如果字符串长度小于size，不会补NUL；如果字符串长度大于size，则截断（保证填充字符串结尾NULL）

## bpf_current_task_under_cgroup

签名：`int bpf_current_task_under_cgroup(struct bpf_map *map, u32 index)`

检查当前正在运行的探针是否在map的index所指向的Cgroup2之下。

## bpf_get_numa_node_id

签名：`int bpf_get_numa_node_id(void)`

得到当前NUMA节点的ID。该函数的主要目的是用于选取本地NUMA节点的套接字。

## bpf_perf_event_output

签名：`int bpf_perf_event_output(struct pt_reg *ctx, struct bpf_map *map, u64 flags, void *data, u64 size)`

将长度为size的blob写入到Map所存放的特殊BPF perf event。map的类型是BPF_MAP_TYPE_PERF_EVENT_ARRAY

perf event必须具有属性：

sample_type = PERF_SAMPLE_RAW

type = PERF_TYPE_SOFTWARE

config = PERF_COUNT_SW_BPF_OUTPUT

flags用于指定写入到数组的索引。masked by BPF_F_INDEX_MASK，如果指定BPF_F_CURRENT_CPU则取当前CPU的值。

当前程序的ctx也需要传递给助手函数。

在用户空间，希望读取值的程序需要针对perf event调用perf_event_open()，然后将文件描述符存储到Map中。这个操作必须在eBPF程序第一次写入数据到Map之前完成。参考内核中的例子samples/bpf/trace_output_user.c

要和用户空间进行数据交互，该函数优于bpf_trace_printk()，性能更好。适合从eBPF程序stream数据给用户空间读取。

## bpf_perf_event_read

签名：`u64 bpf_perf_event_read(struct bpf_map *map, u64 flags)`

读取一个perf event counter的值，该助手函数操作BPF_MAP_TYPE_PERF_EVENT_ARRAY类型的Map。这个Map本质上是一个数组，它的size和CPU数量一致，其值和对应CPU相关。取哪个CPU的值，masked by BPF_F_INDEX_MASK，如果指定BPF_F_CURRENT_CPU则取当前CPU的值。

在4.13之前，仅仅支持hardware perf event。

成功时返回计数器值，否则返回负数。

考虑使用`bpf_perf_event_read_value`代替此函数。

# BPF Skeleton

一个BPF Application由1-N个BPF program、BPF Maps、全局变量组成。所有BPF program（各自对应一个ELF section）、用户空间（的控制）程序可以共享Map/全局变量。

## 管理生命周期

BPF Application通常会经历以下生命周期阶段：

1. Open：控制程序打开BPF object文件，解析了programs/map/global vars，但是尚未在内核中创建这些对象。打开BPF object文件后，控制程序可能进行一些调整，例如设置程序类型、设置全局变量的值，等等
2. Load：在此阶段，BPF Maps被创建，各种relocations被解析，BPF programs被载入内核并校验。这个阶段结束时，BPF程序的所有组件都被校验并且存在于内核，但是BPF program还不会被内核执行，可以保证在没有竞态条件的前提下，对BPF Map进行初始化
3. Attach：BPF programs被挂钩到相应的BPF挂钩点（例如tracepoint、kprobes、cgroup hooks、网络封包处理流水线...），BPF program开始工作，读写BPF Maps和全局变量
4. Teardown：BPF程序被detach并unload，BPF Map被销毁

通过bpftool生成的BPF Skeleton，包含了触发上述阶段的函数：

1. `<name>__open()`：创建并打开BPF Application
2. `<name>__load()`：实例化、加载、校验BPF Application组件
3. `<name>__attach()`：将BPF programs挂钩到内核的hook point。你也可以选择直接使用libbpf API进行细粒度控制
4. `<name>__destroy()`：销毁BPF programs并释放所有资源

## 访问全局变量

在内核空间，访问全局变量使用普通的C语法，你甚至可以取地址并将其传递给助手函数。

在控制程序中，你需要通过BPF skeleton来访问这些变量：

1. `skel->rodata`，访问只读变量（常量）
2. `skel->bss`，访问可变的、以0初始化的变量
3. `skel->data`，访问可变的、非0初始化的变量

在用户空间对这些变量的修改，会立刻反映到内核空间。

# BPF Ring Buffer

BPF ring buffer是一个新的BPF数据结构，它解决了BPF perf buffer（现有的给用户空间发送数据的事实标准工具）的内存效率、事件re-ordering问题。

ringbuf提供了兼容perfbuf的API，便于迁移。同时也提供了新的reserve/commit API，具有更好的易用性。

## 对比perfbuf

当BPF程序将收集的信息发送到用户空间，供后续处理时，通常会利用perfbuf。perfbuf是per-CPU的环形缓冲区，允许内核和用户空间进行高效的数据交换。但是由于它的per-CPU设计，会导致两个缺点：内存低效、事件乱序。

因为这些缺点，内核5.8+开始，BPF提供了新的数据结构，BPF ring buffer。它是一个多生产者、单消费者（MPSC）的队列，可以安全的被多CPU环境共享。

ringbuf支持perfbuf的特性：

1. 可变长度数据记录
2. 基于内存映射区域，高效的从用户空间读取数据的能力，不需要内存拷贝或者执行系统调用
3. 支持epoll通知，或者忙循环（最小化延迟）

## 内存低效问题

perfbuf为每个CPU都分配了独立的缓冲区，开发者可能需要进行权衡：

1. 如果分配足够大的缓冲，则会浪费内存，特别是核心数很多的情况下
2. 如果分配较小的缓冲，那么出现事件spike时，会丢失数据

而ringbuf使用单个缓冲区，因而能够容易应对spike，同时不需要太大的内存消耗。

## 事件乱序问题

在追踪相关事件（例如进程启动/退出、网络连接生命周期事件）的时候，事件顺序非常重要。

使用perfbuf时，如果两个相关事件在很短事件内（若干ms）被不同CPU处理，则可能发出到用户空间的顺序是错乱的。

## 关于性能

在所有应用场景下，BPF ringbuf都有着和perfbuf可比较的性能。

唯一需要考虑使用perfbuf的场景是在NMI (non-maskable interrupt)上下文下运行的BPF程序，例如处理cpu-cycles之类的perf事件。由于ringbuf内部使用一个轻量的自旋锁，在NMI上下文下可能发生锁争用并导致reserve失败。

## 用法对比

为了简化代码的迁移，ringbuf提供了一套类似于perfbuf的API，本节做一个对比。

下面的数据结构代表BPF程序需要收集的一个事件：

```
#define TASK_COMM_LEN 16
#define MAX_FILENAME_LEN 512

struct event {
    int pid;
    char comm[TASK_COMM_LEN];
    char filename[MAX_FILENAME_LEN];
};
```

perfbuf和ringbuf对应不同的Map类型：

```
/*** perfbuf ***/
struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(int));
    __uint(value_size, sizeof(int));
} pb SEC(".maps");

/*** ringbuf ***/
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    // 和perfbuf不同，ringbuf的尺寸可以在内核端定义
    // 但是，在用户空间也可以通过bpf_map__set_max_entries()指定或覆盖ringbuf的尺寸
    // 单位是字节，必须是内核页（一般都是4KB）大小的整数倍，并且是2的幂
    __uint(max_entries, 256 * 1024 /* 256 KB */);
} rb SEC(".maps");
```

由于event超过512字节，因此不能直接在栈上分配，需要存储到BPF_MAP_TYPE_PERCPU_ARRAY：

```
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __uint(max_entries, 1);  // 只需要1个元素即可
    __type(key, int);
    __type(value, struct event);
} heap SEC(".maps");
```

内核空间代码：

```
// 挂钩到sched:sched_process_exec，每当成功的exec()系统调用后触发
SEC("tp/sched/sched_process_exec")
int handle_exec(struct trace_event_raw_sched_process_exec *ctx)
{
    unsigned fname_off = ctx->__data_loc_filename & 0xFFFF;
    struct event *e;
    int zero = 0;
    // 获得事件的指针
    e = bpf_map_lookup_elem(&heap, &zero);
    if (!e) /* can't happen */
        return 0;

    // 填充字段
    e->pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&e->comm, sizeof(e->comm));
    bpf_probe_read_str(&e->filename, sizeof(e->filename), (void *)ctx +
fname_off);

    /*** 填充数据的API不同 ***/
    /*** perfbuf ***/
    // 输出追踪样本，要求在perfbuf中保留e大小的缓冲区，然后通知用户空间有数据可用
    bpf_perf_event_output(ctx, &pb, BPF_F_CURRENT_CPU, e, sizeof(*e));
        /*** ringbuf ***/
    // 不需要传递ctx对象
    bpf_ringbuf_output(&rb, e, sizeof(*e), 0);

    return 0;
}
```

用户空间代码：

```
struct perf_buffer *pb = NULL;
struct perf_buffer_opts pb_opts = {};
struct perfbuf_output_bpf *skel;

/*** 回调函数的签名不同 ***/
/*** perfbuf ***/
void handle_event(void *ctx, int cpu, void *data, unsigned int data_sz)
{
    const struct event *e = data;
    struct tm *tm;
    char ts[32];
    time_t t;
    time(&t);
    tm = localtime(&t);
```

```
    strftime(ts, sizeof(ts), "%H:%M:%S", tm);

    printf("%-8s %-5s %-7d %-16s %s\n", ts, "EXEC", e->pid, e->comm, e-
>filename);
}
/*** ringbuf ***/
int handle_event(void *ctx, void *data, size_t data_sz)
{
    // ...
}

/*** 创建用户空间缓冲区的API不同 ***/
/*** perfbuf ***/
pb_opts.sample_cb = handle_event; // 回调函数
//                      指向内核空间对应物，也就是那个Map
//                                                    为每个CPU分配8个页，也就是32KB的缓
冲区
pb = perf_buffer__new(bpf_map__fd(skel->maps.pb), 8 /* 32KB per CPU */,
&pb_opts);
if (libbpf_get_error(pb)) {
    err = -1;
    fprintf(stderr, "Failed to create perf buffer\n");
    goto cleanup;
}
/*** ringbuf ***/
rb = ring_buffer__new(bpf_map__fd(skel->maps.rb), handle_event, NULL, NULL);
if (!rb) {
    err = -1;
    fprintf(stderr, "Failed to create ring buffer\n");
    goto cleanup;
}


// 开始epoll轮询
while (!exiting) {
    /*** 轮询接口不同 ***/
    /*** perfbuf ***/
    err = perf_buffer__poll(pb, 100 /* timeout, ms */);
    /*** ringbuf ***/
    err = ring_buffer__poll(rb, 100 /* timeout, ms */);

    // ...
}
```

## reserve/commit

上面的perfbuf兼容API，具有与perfbuf类似的缺点：

1. 额外的内存拷贝：你需要额外的内存空间来构建追踪样本对象，然后才能将它拷贝到缓冲区中
2. very late data reservation：构建（可能需要采集多种内存数据）追踪样本对象的工作可能是无意义的，如果缓冲区中剩余空间（由于用户空间程序处理缓慢或者事件burst）不足，构建的对象无处存放。使用xxx_output()接口不能提前感知缓冲区剩余空间，从而避免不必要的样本对象构造

ringbuf提供了一套新的API，应对上述缺点。

1. `bpf_ringbuf_reserve()`：尝试在缓冲区中预定空间，这个操作可以在构建样本对象之前就进行，尽早获取空间不足的状况。如果预定成功，则返回指针，并且可以保证后续可以将数据

commit到其中；如果预定失败，则返回NULL，我们可以跳过后续的、无意义的操作
2. `bpf_ringbuf_commit()`：将样本对象发送到缓冲区

此外，reserve的空间，在commit之前，对于用户空间是不可见的。因此你可以直接用它来构造样本对象，不用担心半初始化的对象被看到，同时达到节约内存的目的。

唯一的限制是：reserve的空间大小，必须能够被BPF verifier感知（常量）。如果样本尺寸是动态的，则只能使用`bpf_ringbuf_output()`并且承受内存拷贝的代价。

用法示例：

```
// 不再需要到per-CPU array中预先构造对象：
//   e = bpf_map_lookup_elem(&heap, &zero);
e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);

// 不再需要将per-CPU array中的对象拷贝到缓冲区
//   bpf_ringbuf_output(&rb, e, sizeof(*e), 0);
bpf_ringbuf_submit(e, 0);
```

## 通知用户空间

不管是perfbuf还是ringbuf，如果内核每写入一个样本，就通知（也就是唤醒在poll/epoll上等待的）用户空间程序，其开销是相当大的。

perfbuf的解决办法是，允许用户空间程序构造perfbuf时，指定每发生N个样本才唤醒一次，这可能会让你丢失最多N-1个样本。

ringbuf则允许为`bpf_ringbuf_output()`或`bpf_ringbuf_commit()`指定额外的标记：

1. BPF_RB_FORCE_WAKEUP 本次写入样本，强制唤醒用户空间程序
2. BPF_RB_NO_WAKEUP本次写入样本，不会唤醒用户空间程序

如果不指定上述标记，ringbuf会根据用户空间程序lagging的情况决定是否唤醒。不指定标记通常是安全的默认值。

# 调试BPF程序

BPF没有常规的调试工具，支持设置断点、探查变量、单步跟踪的那种。

## bpf_printk

使用`bpf_printk(fmt, args...)`可以打印一些信息到`/sys/kernel/debug/tracing/trace_pipe`，来帮助你理解发生了什么，这个函数最多支持3个args。

这个函数的成本很高，不能在生产环境使用。

## bpf_trace_printk

助手函数`long bpf_trace_printk(const char *fmt, __u32 fmt_size, ...)`是bpf_printk的wrapper。

# libbpf-bootstrap

这是libbpf提供的脚手架，可以让你快速开始编写自己的eBPF程序。该脚手架目前提供一些demo程序：

1. minimal：最小化的、能编译、加载、运行的BPF hello world程序

2. bootstrap：现实可用的、可移植的、全功能的BPF程序，依赖BPF CO-RE。需要当前使用的内核开启`CONFIG_DEBUG_INFO_BTF=y`，该demo还示例了5.5+支持的BPF全局变量、5.8+支持的BPF ring buffer

libbpf-bootstrap自带了libbpf（作为git子模块）和bpftool（仅x84），避免依赖你的Linux发行版所携带的特定版本。libbpf依赖的库包括libelf、zlib，需要确保已经安装。

执行下面的命令获得脚手架：

```
git clone https://github.com/libbpf/libbpf-bootstrap.git
git submodule init
```

目录结构如下：

```
.
├── examples
│   ├── c  # C代码示例
│   │   ├── bootstrap.bpf.c  # 在内核空间执行的BPF程序
│   │   ├── bootstrap.c      # 在用户空间执行的逻辑，负责加载BPF字节码，在程序生命周期内和
BPF程序交互
│   │   ├── bootstrap.h      # 用户/内核空间程序共享的头
│   │   ├── CMakeLists.txt
│   │   ├── fentry.bpf.c
│   │   ├── fentry.c
│   │   ├── kprobe.bpf.c
│   │   ├── kprobe.c
│   │   ├── Makefile
│   │   ├── minimal.bpf.c
│   │   ├── minimal.c
│   │   ├── uprobe.bpf.c
│   │   ├── uprobe.c
│   │   └── xmake.lua
│   └── rust # Rust代码示例
│       ├── Cargo.lock
│       ├── Cargo.toml
│       └── xdp
│           ├── build.rs
│           ├── Cargo.lock
│           ├── Cargo.toml
│           └── src
│               ├── bpf
│               │   ├── vmlinux.h -> ../../../../../vmlinux/vmlinux.h
│               │   └── xdppass.bpf.c
│               └── main.rs
├── libbpf # 自带的libbpf
├── LICENSE
├── README.md
├── tools  # 自带bpftool，用于为你的BPF代码构建BPF skeleton、生成vmlinux.h头
│   ├── bpftool
│   ├── cmake
│   │   ├── FindBpfObject.cmake
│   │   └── FindLibBpf.cmake
│   └── gen_vmlinux_h.sh  # 用于生成自定义的vmlinux.h
└── vmlinux
    ├── vmlinux_508.h
    └── vmlinux.h -> vmlinux_508.h  # 预生成的vmlinux.h
```

# minimal

最小化的样例，不依赖BPF CO-RE，可以在很老版本的内核上运行。该程序会安装一个每秒触发一次的tracepoint handler，它使用`bpf_printk()`和外部通信，你可以通过/sys/kernel/debug/tracing/trace_pipe查看输出。

构建此样例：

```
cd examples/c
make minimal
```

内核空间代码：

```
// 包含基本的BPF相关的类型和常量。为了使用内核空间BPF API，例如助手函数的flags，需要引入此头
#include <linux/bpf.h>
// 由libbpf提供，包含了大部分常用宏、常量、助手函数，几乎每个BPF程序都会使用
#include <bpf/bpf_helpers.h>
// 这个变量定义你的代码的License，内核强制要求此字段存在，某些功能对于非GPL兼容的License不可用
//              必须定义在license段
char LICENSE[] SEC("license") = "Dual BSD/GPL";
// 全局变量，要求内核版本5.5+，全局变量甚至可以从用户空间读写
// 可以用于配置BPF程序、存放轻量的统计数据、在内核和用户空间传递数据
int my_pid = 0;

// 这里定义了BPF内核程序
// 这个注解，说明了需要创建的BPF程序类型，以及如何挂钩到内核
//   tracepoint BPF程序
//      在进入write系统调用时触发
SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
    // 调用助手函数，获得PID（内核的术语叫TGID）。为了节约空间，很多助手函数使用高低字节存储不同的数据
    int pid = bpf_get_current_pid_tgid() >> 32;

    // 判断是否是关注进程发出的系统调用
    if (pid != my_pid)
        return 0;
    // 如果是，打印信息到/sys/kernel/debug/tracing/trace_pipe
    // 注意，由于性能问题，这个函数不能用于生产环境
    bpf_printk("BPF triggered from PID %d.\n", pid);

    return 0;
}

// 你还可以定义更多的BPF程序，只需要为它们声明适当的SEC即可。所有这些程序共享全局变量
```

用户空间代码：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
// 由bpftool自动生成的，映射了minimal.bpf.c的高层结构的BPF skeleton
// 编译后的BPF object被内嵌到此头文件（也就是用户空间代码）中，简化了开发和部署
```

```c
// 文件路径 .output/<app>.skel.h
#include "minimal.skel.h"

// 此回调打印libbpf日志到控制台
static int libbpf_print_fn(enum libbpf_print_level level, const char *format,
va_list args)
{
    return vfprintf(stderr, format, args);
}

static void bump_memlock_rlimit(void)
{
    struct rlimit rlim_new = {
        .rlim_cur    = RLIM_INFINITY,
        .rlim_max    = RLIM_INFINITY,
    };

    if (setrlimit(RLIMIT_MEMLOCK, &rlim_new)) {
        fprintf(stderr, "Failed to increase RLIMIT_MEMLOCK limit!\n");
        exit(1);
    }
}

int main(int argc, char **argv)
{
    struct minimal_bpf *skel;
    int err;

    // 为所有libbpf日志设置回调函数
    libbpf_set_print(libbpf_print_fn);

    // 增大内核内部的per-user内存限制，允许BPF子系统为程序、Map分配足够的资源
    bump_memlock_rlimit();

    // 打开BPF skeleton
    skel = minimal_bpf__open();
    if (!skel) {
        fprintf(stderr, "Failed to open BPF skeleton\n");
        return 1;
    }

    // 访问BSS段中的全局变量
    skel->bss->my_pid = getpid();

    // 加载和校验BPF程序
    err = minimal_bpf__load(skel);
    if (err) {
        fprintf(stderr, "Failed to load and verify BPF skeleton\n");
        goto cleanup;
    }

    // 挂钩BPF程序，在此即注册tracepoint handler
    // libbpf能够根据SEC注解，自动的为大部分BPF程序类型（tracepoints, kprobes等）选择适当
的挂钩点
    // 如果不能满足需求，可以调用libbpf提供的函数手工挂钩的API
    err = minimal_bpf__attach(skel);
    if (err) {
        fprintf(stderr, "Failed to attach BPF skeleton\n");
```

```
        goto cleanup;
    }

    printf("Successfully started! Please run `sudo cat
/sys/kernel/debug/tracing/trace_pipe` "
           "to see output of the BPF programs.\n");

    // 触发write系统调用，进而触发BPF程序
    for (;;) {
        /* trigger our BPF program */
        fprintf(stderr, ".");
        sleep(1);
    }

cleanup:
    // 清除所有内核/用户空间的资源
    // 在大部分情况下，即使程序崩溃，没有清理，内核也会作自动清理
    minimal_bpf__destroy(skel);
    return -err;
}
```

构建此程序的Makefile：

```
# SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause)
OUTPUT := .output
CLANG ?= clang
LLVM_STRIP ?= llvm-strip
BPFTOOL ?= $(abspath ../../tools/bpftool)
LIBBPF_SRC := $(abspath ../../libbpf/src)
LIBBPF_OBJ := $(abspath $(OUTPUT)/libbpf.a)
# vmlinux.h由bpftool从当前运行的内核中抽取出，包含了所有内核使用的类型定义
VMLINUX := ../../vmlinux/vmlinux.h
# 使用自己的libbpf API头文件、Linux UAPI头文件，避免依赖当前系统的（可能缺失或过期的）头文件
INCLUDES := -I$(OUTPUT) -I../../libbpf/include/uapi -I$(dir $(VMLINUX))
# 使用-g保留调试信息
CFLAGS := -g -Wall
# 体系结构信息会传递给后续的BPF构建步骤，使用bpf_tracing.h中低级别tracing助手函数需要此信息
ARCH := $(shell uname -m | sed 's/x86_64/x86/')

# 这个示例项目包含多个demo，每个对应一个APP
APPS = minimal bootstrap uprobe kprobe fentry

# 获取Clang在当前系统上默认的includes目录，当以-target bpf编译时，这些目录被显式添加到
# include列表。如果不这样做，某些体系结构/发行版下，体系结构特定的目录可能missing，诸如
# asm/types.h, asm/byteorder.h, asm/socket.h, asm/sockios.h, sys/cdefs.h之类
# 的头文件可能missing
CLANG_BPF_SYS_INCLUDES = $(shell $(CLANG) -v -E - </dev/null 2>&1 \
    | sed -n '/<...> search starts here:/,/End of search list./{ s| \(/.*\)|-
idirafter \1|p }')

ifeq ($(V),1)
    Q =
    msg =
else
    Q = @
    msg = @printf '  %-8s %s%s\n'                    \
              "$(1)"                          \
```

```
              "$(patsubst $(abspath $(OUTPUT))/%,%,$(2))"    \
              "$(if $(3), $(3))";
    MAKEFLAGS += --no-print-directory
endif

.PHONY: all
all: $(APPS)

.PHONY: clean
clean:
    $(call msg,CLEAN)
    $(Q)rm -rf $(OUTPUT) $(APPS)

$(OUTPUT) $(OUTPUT)/libbpf:
    $(call msg,MKDIR,$@)
    $(Q)mkdir -p $@

# 1. 构建 libbpf 为静态库，存放在.output目录下
#    如果希望和系统的libbpf共享库链接，去掉这个目标
$(LIBBPF_OBJ): $(wildcard $(LIBBPF_SRC)/*.[ch] $(LIBBPF_SRC)/Makefile) |
$(OUTPUT)/libbpf
    $(call msg,LIB,$@)
    $(Q)$(MAKE) -C $(LIBBPF_SRC) BUILD_STATIC_ONLY=1              \
            OBJDIR=$(dir $@)/libbpf DESTDIR=$(dir $@)              \
            INCLUDEDIR= LIBDIR= UAPIDIR=                    \
            install

# 2. 构建BPF object
$(OUTPUT)/%.bpf.o: %.bpf.c $(LIBBPF_OBJ) $(wildcard %.h) $(VMLINUX) | $(OUTPUT)
    $(call msg,BPF,$@)
    # 必须使用-g -O2      目标必须是bpf 为bpf_tracing.h定义必要的宏
    #
    $(Q)$(CLANG) -g -O2 -target bpf -D__TARGET_ARCH_$(ARCH) $(INCLUDES)
$(CLANG_BPF_SYS_INCLUDES) -c $(filter %.c,$^) -o $@
    $(Q)$(LLVM_STRIP) -g $@ # 去除DWARF信息，从来不会用到。由于BPF程序最终以文本形式嵌入
到BPF skeleton，因此要尽量精简

# 3. 生成BPF skeletons，依赖2
$(OUTPUT)/%.skel.h: $(OUTPUT)/%.bpf.o | $(OUTPUT)
    $(call msg,GEN-SKEL,$@)
    $(Q)$(BPFTOOL) gen skeleton $< > $@

# 4. 构建用户空间程序object，依赖3
$(patsubst %,$(OUTPUT)/%.o,$(APPS)): %.o: %.skel.h

$(OUTPUT)/%.o: %.c $(wildcard %.h) | $(OUTPUT)
    $(call msg,CC,$@)
    $(Q)$(CC) $(CFLAGS) $(INCLUDES) -c $(filter %.c,$^) -o $@

# 5. 构建用户空间程序
$(APPS): %: $(OUTPUT)/%.o $(LIBBPF_OBJ) | $(OUTPUT)
    $(call msg,BINARY,$@)
    $(Q)$(CC) $(CFLAGS) $^ -lelf -lz -o $@

# delete failed targets
.DELETE_ON_ERROR:

# keep intermediate (.skel.h, .bpf.o, etc) targets
```

```
.SECONDARY:
```

## bootstrap

minimal是最小化的BPF程序实例，在现代Linux环境下开发BPF程序，bootstrap可以作为一个不错的开始点。它实现的功能包括：

1. 命令行参数解析
2. 信号处理
3. 多个BPF程序之间的交互 —— 通过Map共享状态
4. 使用BPF ring buffer来发送数据到用户空间
5. 用于行为参数化的全局常量，以及如何通过修改段数据初始化常量值
6. 使用BPF CO-RE和vmlinux.h来读取内核`struct task_struct`暴露的进程的额外信息

该程序依赖BPF CO-RE，需要内核配置`CONFIG_DEBUG_INFO_BTF=y`。

公共头文件：

```
#ifndef __BOOTSTRAP_H
#define __BOOTSTRAP_H

#define TASK_COMM_LEN 16
#define MAX_FILENAME_LEN 127

struct event {
    int pid;
    int ppid;
    unsigned exit_code;
    unsigned long long duration_ns;
    char comm[TASK_COMM_LEN];
    char filename[MAX_FILENAME_LEN];
    bool exit_event;
};

#endif /* __BOOTSTRAP_H */
```

内核空间代码：

```
// 该头文件包含内核的所有数据类型，通过gen_vmlinux_h.sh自动生成

// 其中的类型应用了__attribute__((preserve_access_index))注解，可以让Clang
// 生成BPF CO-RE relocations，这让libbpf能够自动的将BPF代码是配到宿主机内核的内存布局，即使
// 该布局和构建BPF程序时候的主机不一致
// BPF CO-RE relocations是创建pre-compiled、可移植的BPF程序的关键，它不需要在目标及其上部署
// Clang/LLVM工具链。一个被选的技术是BCC的运行时编译，这种技术有多个缺点

// 需要注意，vmlinux.h不能和其它系统级的头文件联合适用，会导致类型重定义和冲突
// 因此开发BPF程序时，仅可使用vmlinux.h、libbpf提供的头、你自己定义的头
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
// 下面两个头文件，用于编写基于BPF CO-RE的追踪应用程序
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>
// 用于/内核空间共享头
```

```c
#include "bootstrap.h"

char LICENSE[] SEC("license") = "Dual BSD/GPL";

// 定义一个哈希表类型的Map
struct {
    // 这几个__开头的是宏bpf_helpers.h中定义的宏，用来定义结构体字段
    // 定义一个BPF_MAP_TYPE_HASH类型的，最大条目8192的，key类型为pid_t，值类型为u64的Map
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 8192);
    __type(key, pid_t);
    __type(value, u64);
} exec_start SEC(".maps");
//            为了让libbpf知道它需要创建BPF Map，必须添加此注解

// 定义一个BPF ring buffer类型的Map
// 用于向用户空间发送数据，本样例使用bpf_ringbuf_reserve()/bpf_ringbuf_submit()来实现最高
// 的易用性和性能
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 256 * 1024);
} rb SEC(".maps");

// 全局常量，对于用户/内核空间均不可变。在BPF程序校验期间，此常量值已知，BPF Verifier可能依据值来裁减
// 某些dead code
// volatile是必须的，可以防止Clang将变量优化掉（直接没了）
const volatile unsigned long long min_duration_ns = 0;

// 本样例由两个BPF程序组成，这个监控exec系统调用
SEC("tp/sched/sched_process_exec")
int handle_exec(struct trace_event_raw_sched_process_exec *ctx)
{
    struct task_struct *task;
    unsigned fname_off;
    struct event *e;
    pid_t pid;
    u64 ts;

    // 得到当前PID
    pid = bpf_get_current_pid_tgid() >> 32;
    // 获取当前内核时间
    ts = bpf_ktime_get_ns();
    // 记录当前进程的创建时间
    bpf_map_update_elem(&exec_start, &pid, &ts, BPF_ANY);

    /* don't emit exec events when minimum duration is specified */
    if (min_duration_ns)
        return 0;

    // 在ring buffer中保留event长度的空间，返回得到的空间指针
    e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);
    if (!e)
        return 0;

    // 得到当前Task对象
    task = (struct task_struct *)bpf_get_current_task();
```

```
    // 并填充event
    e->exit_event = false;
    e->pid = pid;
    // BPF CO-RE读取
    // 相当于e->ppid = task->real_parent->tgid;
    // 但是对于BPF程序来说，BPF verifier需要额外的检查，因为存在读取任意内核内存的可能
    // BPF_CORE_READ()通过简洁的方式完成此过程，并且记录必要的BPF CO-RE relocations，从
而
    // 允许libbpf根据宿主机的内存布局，来调整字段的偏移量
    e->ppid = BPF_CORE_READ(task, real_parent, tgid);
    // 读取当前任务的可执行文件名（不包含路径）
    bpf_get_current_comm(&e->comm, sizeof(e->comm));

    // 获取新进程文件名在ctx的偏移量              移除u32高16位
    fname_off = ctx->__data_loc_filename & 0xFFFF;
    // 从ctx读取文件名，存入e
    bpf_probe_read_str(&e->filename, sizeof(e->filename), (void *)ctx +
fname_off);

    // 提交给用户空间供处理
    bpf_ringbuf_submit(e, 0);
    return 0;
}

// 本样例由两个BPF程序组成，这个监控进程退出，即exit系统调用
SEC("tp/sched/sched_process_exit")
int handle_exit(struct trace_event_raw_sched_process_template* ctx)
{
    struct task_struct *task;
    struct event *e;
    pid_t pid, tid;
    u64 id, ts, *start_ts, duration_ns = 0;

    // 获取正在退出的进程的PID
    id = bpf_get_current_pid_tgid();
    // 高32位是pid，低32位是线程ID
    pid = id >> 32;
    tid = (u32)id;

    // 如果pid和tid不同，则意味着是非主线程退出，忽略
    if (pid != tid)
        return 0;

    // 查找进程开始时间，并计算进程持续时间
    start_ts = bpf_map_lookup_elem(&exec_start, &pid);
    if (start_ts)
        duration_ns = bpf_ktime_get_ns() - *start_ts;
    else if (min_duration_ns)
        return 0;
    // 从Map删除条目
    bpf_map_delete_elem(&exec_start, &pid);

    // 忽略持续时间过短的进程
    if (min_duration_ns && duration_ns < min_duration_ns)
        return 0;

    // 从ring buffer分配空间
    e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);
```

```
    if (!e)
        return 0;

    // 填充并提交给用户空间处理
    task = (struct task_struct *)bpf_get_current_task();

    e->exit_event = true;
    e->duration_ns = duration_ns;
    e->pid = pid;
    e->ppid = BPF_CORE_READ(task, real_parent, tgid);
    e->exit_code = (BPF_CORE_READ(task, exit_code) >> 8) & 0xff;
    bpf_get_current_comm(&e->comm, sizeof(e->comm));

    /* send data to user-space for post-processing */
    bpf_ringbuf_submit(e, 0);
    return 0;
}
```

用户空间代码：

```
#include <argp.h>
#include <signal.h>
#include <stdio.h>
#include <time.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "bootstrap.h"
#include "bootstrap.skel.h"

// 使用libc的argp做命令行参数解析
static struct env {
    bool verbose;
    long min_duration_ms;
} env;

const char *argp_program_version = "bootstrap 0.0";
const char *argp_program_bug_address = "<bpf@vger.kernel.org>";
const char argp_program_doc[] =
"BPF bootstrap demo application.\n"
"\n"
"It traces process start and exits and shows associated \n"
"information (filename, process duration, PID and PPID, etc).\n"
"\n"
"USAGE: ./bootstrap [-d <min-duration-ms>] [-v]\n";

static const struct argp_option opts[] = {
    { "verbose", 'v', NULL, 0, "Verbose debug output" },
    { "duration", 'd', "DURATION-MS", 0, "Minimum process duration (ms) to
report" },
    {},
};

static error_t parse_arg(int key, char *arg, struct argp_state *state)
{
    switch (key) {
    case 'v':
        env.verbose = true;
```

```
        break;
    case 'd':
        errno = 0;
        env.min_duration_ms = strtol(arg, NULL, 10);
        if (errno || env.min_duration_ms <= 0) {
            fprintf(stderr, "Invalid duration: %s\n", arg);
            argp_usage(state);
        }
        break;
    case ARGP_KEY_ARG:
        argp_usage(state);
        break;
    default:
        return ARGP_ERR_UNKNOWN;
    }
    return 0;
}

static const struct argp argp = {
    .options = opts,
    .parser = parse_arg,
    .doc = argp_program_doc,
};

static int libbpf_print_fn(enum libbpf_print_level level, const char *format,
va_list args)
{
    if (level == LIBBPF_DEBUG && !env.verbose)
        return 0;
    return vfprintf(stderr, format, args);
}

static void bump_memlock_rlimit(void)
{
    struct rlimit rlim_new = {
        .rlim_cur    = RLIM_INFINITY,
        .rlim_max    = RLIM_INFINITY,
    };

    if (setrlimit(RLIMIT_MEMLOCK, &rlim_new)) {
        fprintf(stderr, "Failed to increase RLIMIT_MEMLOCK limit!\n");
        exit(1);
    }
}

static volatile bool exiting = false;

static void sig_handler(int sig)
{
    exiting = true;
}

static int handle_event(void *ctx, void *data, size_t data_sz)
{
    // 直接转换为event
    const struct event *e = data;
    struct tm *tm;
    char ts[32];
```

```
    time_t t;

    time(&t);
    tm = localtime(&t);
    strftime(ts, sizeof(ts), "%H:%M:%S", tm);

    if (e->exit_event) {
        printf("%-8s %-5s %-16s %-7d %-7d [%u]", ts, "EXIT", e->comm, e->pid, e-
>ppid, e->exit_code);
        if (e->duration_ns) printf(" (%llums)", e->duration_ns / 1000000);
        printf("\n");
    } else {
        printf("%-8s %-5s %-16s %-7d %-7d %s\n",  ts, "EXEC", e->comm, e->pid,
e->ppid, e->filename);
    }
    return 0;
}

int main(int argc, char **argv)
{
    struct ring_buffer *rb = NULL;
    struct bootstrap_bpf *skel;
    int err;

    // 解析命令行参数
    err = argp_parse(&argp, argc, argv, 0, NULL, NULL);
    if (err)
        return err;

    // 日追回调函数
    libbpf_set_print(libbpf_print_fn);

    // 设置RLIMIT_MEMLOCK
    bump_memlock_rlimit();

    // 去除默认Ctrl-C处理逻辑
    signal(SIGINT, sig_handler);
    signal(SIGTERM, sig_handler);

    // 由于有一个常量需要设置值，因此这里没有使用bootstrap_bpf__open_and_load()
    // 而是先open
    skel = bootstrap_bpf__open();
    if (!skel) {
        fprintf(stderr, "Failed to open and load BPF skeleton\n");
        return 1;
    }

    // 然后设置常量值
    skel->rodata->min_duration_ns = env.min_duration_ms * 1000000ULL;

    // 最后load，加载之后，用户空间只能读取，不能修改 skel->rodata->min_duration_ns
    err = bootstrap_bpf__load(skel);
    if (err) {
        fprintf(stderr, "Failed to load and verify BPF skeleton\n");
        goto cleanup;
    }

    // 挂钩BPF程序到内核
```

```
    err = bootstrap_bpf__attach(skel);
    if (err) {
        fprintf(stderr, "Failed to attach BPF skeleton\n");
        goto cleanup;
    }

    // 创建ring buffer轮询
    // 映射位ring_buffer类型    内核空间对应物                轮询到数据后的回调
    rb = ring_buffer__new(bpf_map__fd(skel->maps.rb), handle_event, NULL, NULL);
    if (!rb) {
        err = -1;
        fprintf(stderr, "Failed to create ring buffer\n");
        goto cleanup;
    }

    // 打印表头
    printf("%-8s %-5s %-16s %-7s %-7s %s\n",
           "TIME", "EVENT", "COMM", "PID", "PPID", "FILENAME/EXIT CODE");
    // 接收到信号前，反复轮询
    while (!exiting) {
        err = ring_buffer__poll(rb, 100 /* timeout, ms */);
        /* Ctrl-C will cause -EINTR */
        if (err == -EINTR) {
            err = 0;
            break;
        }
        if (err < 0) {
            printf("Error polling perf buffer: %d\n", err);
            break;
        }
    }

cleanup:
    // 需要清理ring buffer和BPF
    ring_buffer__free(rb);
    bootstrap_bpf__destroy(skel);

    return err < 0 ? -err : 0;
}
```

## uprobe

上面两个例子都是tracepoint BPF程序。tracepoint是内核中比较稳定的追踪机制，属于静态instrument，它由一系列预定义在内核源码中的挂钩点组成。

kprobe/uprobe则属于动态instrument，在运行时动态的进行instrument，可以对任何函数进行调试追踪。例如在函数的入口、出口地址、或者某一行值入代码，执行到这些代码的时候，你就可以获得上下文信息，例如当前函数的名字、参数、返回值、寄存器甚至全局数据结构信息。

kprobe/uprobe很强大，但是依赖于程序/内核的特定版本，因为函数的签名可能改变，函数也可能被删除。

libbpf-bootstrap提供了一个uprobe样例，此样例能够处理用户空间的entry/exit(return)探针。

内核空间代码：

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

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

char LICENSE[] SEC("license") = "Dual BSD/GPL";

// 函数entry探针
SEC("uprobe/func")
// 这里是调用一个宏，在函数体中可以使用参数 struct pt_regs *ctx
//                      逐个列出函数的参数
int BPF_KPROBE(uprobe, int a, int b)
{
    bpf_printk("UPROBE ENTRY: a = %d, b = %d\n", a, b);
    return 0;
}

// 函数exit探针
SEC("uretprobe/func")
//                            列出函数的返回值
int BPF_KRETPROBE(uretprobe, int ret)
{
    bpf_printk("UPROBE EXIT: return = %d\n", ret);
    return 0;
}
```

用户空间代码：

```c
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "uprobe.skel.h"

//...

/* 通过/proc/self/maps查找进程的base load address, 寻找第一个可执行的（r-xp）内存映射:
 * 绝对起始地址                     此区域的偏移量
 * 5574fd254000-5574fd258000 r-xp 00002000 fd:01 668759
/usr/bin/cat
 * ^^^^^^^^^^^^                    ^^^^^^^^
 * 绝对起始地址 - 此区域的偏移量 即得到进程的base load address
 */
static long get_base_addr() {
    size_t start, offset;
    char buf[256];
    FILE *f;

    f = fopen("/proc/self/maps", "r");
    if (!f)
        return -errno;

    while (fscanf(f, "%zx-%*x %s %zx %*[^\n]\n", &start, buf, &offset) == 3) {
        if (strcmp(buf, "r-xp") == 0) {
            fclose(f);
            return start - offset;
        }
    }
```

```
    fclose(f);
    return -1;
}

// 这个是被追踪的函数
int uprobed_function(int a, int b)
{
    return a + b;
}

int main(int argc, char **argv)
{
    struct uprobe_bpf *skel;
    long base_addr, uprobe_offset;
    int err, i;

    // 注册日志回调钩子、调整rlimit、打开并加载BPF程序

    base_addr = get_base_addr();
    if (base_addr < 0) {
        fprintf(stderr, "Failed to determine process's load address\n");
        err = base_addr;
        goto cleanup;
    }

    // uprobe/uretprobe期望获知被追踪函数的相对偏移量，这个偏移量是相对于process的base
load address的
    // 这里直接得到目标函数的绝对地址，然后减去上面我们得到的base load address，即可得到偏移
量
    //
    // 通常情况下，被追踪的函数不会在当前程序中，这可能需要解析其所在程序的ELF，
    // 函数相对base load address的偏移量 = .text段的偏移量 + 函数在.text段内的偏移量
    //
    //               得到函数的绝对地址
    uprobe_offset = (long)&uprobed_function - base_addr;

    //                    挂钩
    skel->links.uprobe = bpf_program__attach_uprobe(skel->progs.uprobe,
                           false /* entry钩子 */,
                           0 /* 到当前进程 */,
                           "/proc/self/exe", // 当前进程的二进制文件路径
                           uprobe_offset); // 追踪的目标函数的偏移量
    err = libbpf_get_error(skel->links.uprobe);
    if (err) {
        fprintf(stderr, "Failed to attach uprobe: %d\n", err);
        goto cleanup;
    }

    // 这里示例了如何挂钩uprobe/uretprobe到任何现存的/未来创建的、使用同一二进制文件的进程
    skel->links.uretprobe = bpf_program__attach_uprobe(skel->progs.uretprobe,
                              true /* exit钩子 */,
                              -1 /* 任何进程 */,
                              "/proc/self/exe",
                              uprobe_offset);
    err = libbpf_get_error(skel->links.uretprobe);
    if (err) {
        fprintf(stderr, "Failed to attach uprobe: %d\n", err);
```

```
        goto cleanup;
    }

    printf("Successfully started! Please run `sudo cat
/sys/kernel/debug/tracing/trace_pipe` "
           "to see output of the BPF programs.\n");

    for (i = 0; ; i++) {
        // 调用目标函数，触发BPF程序
        fprintf(stderr, ".");
        uprobed_function(i, i + 1);
        sleep(1);
    }

cleanup:
    uprobe_bpf__destroy(skel);
    return -err;
}
```

运行此程序，就可以看到函数uprobed_function每次调用的参数、返回值了。

## kprobe

kprobe和uprobe工作方式很类似，只是挂钩的是内核函数。

libbpf-bootstrap提供了一个kprobe样例，该样例挂钩到do_unlinkat()函数，并且打印PID、文件名、返回值等信息。

内核空间代码：

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

char LICENSE[] SEC("license") = "Dual BSD/GPL";

// 注解里必须包含目标内核函数的名字
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
    pid_t pid;
    const char *filename;

    pid = bpf_get_current_pid_tgid() >> 32;
    // 必须使用此助手函数访问指针参数，fentry则不必
    filename = BPF_CORE_READ(name, name);
    bpf_printk("KPROBE ENTRY pid = %d, filename = %s\n", pid, filename);
    return 0;
}

SEC("kretprobe/do_unlinkat")
int BPF_KRETPROBE(do_unlinkat_exit, long ret)
{
    pid_t pid;

    pid = bpf_get_current_pid_tgid() >> 32;
```

```
    bpf_printk("KPROBE EXIT: pid = %d, ret = %ld\n", pid, ret);
    return 0;
}
```

用户空间代码：

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "kprobe.skel.h"

// ...

static volatile sig_atomic_t stop;

static void sig_int(int signo)
{
    stop = 1;
}

int main(int argc, char **argv)
{
    struct kprobe_bpf *skel;
    int err;

    // 注册日志回调钩子、调整rlimit、打开并加载BPF程序

    // 挂钩处理比起uprobe要简单的多，不需要计算目标函数的地址
    err = kprobe_bpf__attach(skel);
    if (err) {
        fprintf(stderr, "Failed to attach BPF skeleton\n");
        goto cleanup;
    }

    if (signal(SIGINT, sig_int) == SIG_ERR) {
        fprintf(stderr, "can't set signal handler: %s\n", strerror(errno));
        goto cleanup;
    }

    printf("Successfully started! Please run `sudo cat
/sys/kernel/debug/tracing/trace_pipe` "
           "to see output of the BPF programs.\n");

    while (!stop) {
        fprintf(stderr, ".");
        sleep(1);
    }

cleanup:
    kprobe_bpf__destroy(skel);
    return -err;
}
```

# fentry

内核5.5引入了BPF trampoline，可以让内核和BPF程序更快速的相互调用。fentry/fexit是它的一个用例，fentry/fexit基本上等价于kprobe/kretprobe，但是调用BPF程序不会引起额外的overhead。

在XDP开发中BPF trampoline能很大程度上改善BPF相关的网络troubleshooting的体验。BPF trampoline支持将fentry/fexit BPF程序挂钩到任何网络BPF程序上，从而可以看到任何XDP/TC/lwt/cgroup程序的输入输出封包，同时不会产生干扰。

libbpf-bootstrap提供了一个fentry样例，类似于kprobe样例，它也是挂钩到do_unlinkat()函数，从而在文件被删除后触发BPF程序，记录各种信息。

fentry/fexit提升了性能的同时，也改善了开发体验。**访问指针参数时**，不再需要调用助手函数，**可以直接解引用。fexit**和kretprobe比起来，**可以同时返回输入参数和返回值**，后者只能访问返回值。

内核空间代码：

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

char LICENSE[] SEC("license") = "Dual BSD/GPL";

// 注解里必须包含目标内核函数的名字
SEC("fentry/do_unlinkat")
int BPF_PROG(do_unlinkat, int dfd, struct filename *name)
{
    pid_t pid;

    pid = bpf_get_current_pid_tgid() >> 32;
    bpf_printk("fentry: pid = %d, filename = %s\n", pid, name->name);
    return 0;
}

SEC("fexit/do_unlinkat")
//                                可以同时访问入参和返回值
int BPF_PROG(do_unlinkat_exit, int dfd, struct filename *name, long ret)
{
    pid_t pid;

    pid = bpf_get_current_pid_tgid() >> 32;
    bpf_printk("fexit: pid = %d, filename = %s, ret = %ld\n", pid, name->name,
ret);
    return 0;
}
```

用户空间代码：

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "fentry.skel.h"
```

```
static volatile sig_atomic_t stop;

int main(int argc, char **argv)
{
    struct fentry_bpf *skel;
    int err;

    // 注册日志回调钩子、调整rlimit、打开并加载BPF程序

    // 挂钩
    err = fentry_bpf__attach(skel);
    if (err) {
        fprintf(stderr, "Failed to attach BPF skeleton\n");
        goto cleanup;
    }

    while (!stop) {
        fprintf(stderr, ".");
        sleep(1);
    }

cleanup:
    fentry_bpf__destroy(skel);
    return -err;
}
```

# 内核样例程序

内核在samples/bpf/下包含了若干eBPF程序示例。这些程序都包含两部分：

1. `*_kern.c`会编译为BPF内核程序
2. `*_user.c`为用户空间程序，加载上述内核程序

这些样例没有封装，直接使用libbpf提供的函数完成各种功能，包括助手函数调用、ELF加载、BPF挂钩、Map创建和读写。

执行下面的命令构建示例：

```
# 可能需要进行必要的清理
make -C tools clean
make -C samples/bpf clean
make clean

# 当前正在构建版本的内核头文件，复制到usr/include子目录。构建samples会优先使用这些头文件
make headers_install

# 构建BPF样例
make M=samples/bpf
```

## tracex4

eBPF程序代码：

```
#include <linux/ptrace.h>
#include <linux/version.h>
#include <uapi/linux/bpf.h>
```

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

struct pair {
    u64 val;
    u64 ip;
};

// 定义一个Map
struct {
    // eBPF提供多种Map，BPF_MAP_TYPE_HASH是其中之一
    __uint(type, BPF_MAP_TYPE_HASH);
    // 键值的类型
    __type(key, long);
    __type(value, struct pair);
    // 容量
    __uint(max_entries, 1000000);
} my_map SEC(".maps");
        // SEC宏用于在二进制文件中创建一个新的段

// 这个段实际上声明了挂钩点
SEC("kprobe/kmem_cache_free")
// 从Map中删除一个键值
int bpf_prog1(struct pt_regs *ctx)
{
    long ptr = PT_REGS_PARM2(ctx);

    bpf_map_delete_elem(&my_map, &ptr);
    return 0;
}

// 这个段实际上声明了挂钩点
SEC("kretprobe/kmem_cache_alloc_node")
// 添加一个键值到Map
int bpf_prog2(struct pt_regs *ctx)
{
    long ptr = PT_REGS_RC(ctx);
    long ip = 0;

    /* get ip address of kmem_cache_alloc_node() caller */
    BPF_KRETPROBE_READ_RET_IP(ip, ctx);

    struct pair v = {
        .val = bpf_ktime_get_ns(),
        .ip = ip,
    };

    bpf_map_update_elem(&my_map, &ptr, &v, BPF_ANY);
    return 0;
}
char _license[] SEC("license") = "GPL";
u32 _version SEC("version") = LINUX_VERSION_CODE;
```

用于空间程序源码：

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>
#include <sys/resource.h>

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

struct pair {
    long long val;
    __u64 ip;
};

static __u64 time_get_ns(void)
{
    struct timespec ts;

    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000000000ull + ts.tv_nsec;
}

static void print_old_objects(int fd)
{
    long long val = time_get_ns();
    __u64 key, next_key;
    struct pair v;

    key = write(1, "\e[1;1H\e[2J", 12); /* clear screen */

    key = -1;
    while (bpf_map_get_next_key(fd, &key, &next_key) == 0) {
        bpf_map_lookup_elem(fd, &next_key, &v);
        key = next_key;
        if (val - v.val < 1000000000ll)
            /* object was allocated more then 1 sec ago */
            continue;
        printf("obj 0x%llx is %2lldsec old was allocated at ip %llx\n",
               next_key, (val - v.val) / 1000000000ll, v.ip);
    }
}

int main(int ac, char **argv)
{
    struct rlimit r = {RLIM_INFINITY, RLIM_INFINITY};
    struct bpf_link *links[2];
    struct bpf_program *prog;
    struct bpf_object *obj;
    char filename[256];
    int map_fd, i, j = 0;

    if (setrlimit(RLIMIT_MEMLOCK, &r)) {
        perror("setrlimit(RLIMIT_MEMLOCK, RLIM_INFINITY)");
        return 1;
    }

    snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);
```

```
    // 打开编译好的eBPF程序，tracex4_kern.o
    obj = bpf_object__open_file(filename, NULL);
    if (libbpf_get_error(obj)) {
        fprintf(stderr, "ERROR: opening BPF object file failed\n");
        return 0;
    }

    // 载入eBPF程序
    // 调用之后，定义在eBPF中的探针被添加到/sys/kernel/debug/tracing/kprobe_events
    //   cat /sys/kernel/debug/tracing/kprobe_events
    //   p:kprobes/kmem_cache_free kmem_cache_free
    //   r:kprobes/kmem_cache_alloc_node kmem_cache_alloc_node
    if (bpf_object__load(obj)) {
        fprintf(stderr, "ERROR: loading BPF object file failed\n");
        goto cleanup;
    }

    // 找到共享的Map
    map_fd = bpf_object__find_map_fd_by_name(obj, "my_map");
    if (map_fd < 0) {
        fprintf(stderr, "ERROR: finding a map in obj file failed\n");
        goto cleanup;
    }

    // 执行挂钩
    bpf_object__for_each_program(prog, obj) {
        links[j] = bpf_program__attach(prog);
        if (libbpf_get_error(links[j])) {
            fprintf(stderr, "ERROR: bpf_program__attach failed\n");
            links[j] = NULL;
            // 如果出错，解除所有已经注册的钩子
            goto cleanup;
        }
        j++;
    }

    // 读取Map并打印
    for (i = 0; ; i++) {
        print_old_objects(map_fd);
        sleep(1);
    }

cleanup:
    for (j--; j >= 0; j--)
        bpf_link__destroy(links[j]);

    bpf_object__close(obj);
    return 0;
}
```

# XDP样例程序

## 过滤IPv6

这个简单的、可以和iproute2一起工作的XDP样例，判断封包是否是IPv6的，如果是，则丢弃：

```
#define KBUILD_MODNAME "xdp_ipv6_filter"
#include <uapi/linux/bpf.h>
#include <uapi/linux/if_ether.h>
#include <uapi/linux/if_packet.h>
#include <uapi/linux/if_vlan.h>
#include <uapi/linux/ip.h>
#include <uapi/linux/in.h>
#include <uapi/linux/tcp.h>
#include "bpf_helpers.h"

#define DEBUG 1

#ifdef  DEBUG
/* Only use this for debug output. Notice output from  bpf_trace_printk()
 * end-up in /sys/kernel/debug/tracing/trace_pipe
 */
#define bpf_debug(fmt, ...)                     \
        ({                              \
            char ____fmt[] = fmt;               \
            // 这个函数打印消息到 /sys/kernel/debug/tracing/trace_pipe
            bpf_trace_printk(____fmt, sizeof(____fmt),  \
                     ##__VA_ARGS__);            \
        })
#else
#define bpf_debug(fmt, ...) { } while (0)
#endif

// 解析包头得到ethertype
static __always_inline bool parse_eth(struct ethhdr *eth, void *data_end, u16
*eth_type)
{
    u64 offset;

    offset = sizeof(*eth);
    if ((void *)eth + offset > data_end)
        return false;
    *eth_type = eth->h_proto;
    return true;
}

// 定义一个名为prog的段
// 由于我们使用iproute2来挂钩，因此必须用prog这个段名。否则iproute2无法识别
SEC("prog")
// xdp_md结构包含访问封包所需的所有数据
int xdp_ipv6_filter_program(struct xdp_md *ctx)
{
    // 得到封包头尾指针
    void *data_end = (void *)(long)ctx->data_end;
    void *data     = (void *)(long)ctx->data;
    struct ethhdr *eth = data;
    u16 eth_type = 0;
    // 调用parse_eth函数，可以获取封包的ethertype
    if (!(parse_eth(eth, data_end, eth_type))) {
```

```
        bpf_debug("Debug: Cannot parse L2\n");
        return XDP_PASS;
    }

    bpf_debug("Debug: eth_type:0x%x\n", ntohs(eth_type));
    // 判断ethertype是否IPv6
    if (eth_type == ntohs(0x86dd)) {
        return XDP_PASS;
    } else {
        return XDP_DROP;
    }
}

char _license[] SEC("license") = "GPL";
```

编译此程序后得到xdp_ipv6_filter.o文件。要将此文件加载到网络接口，可以使用两种方法：

1. 编写一个用户空间程序，加载对象文件，挂钩到一个网络接口
2. 调用iproute2进行等价操作

某些网络接口在驱动层支持XDP，包括ixgbe, i40e, mlx5, veth, tap, tun, virtio_net等。这种情况下，XDP钩子实现在网络栈的最底部，即NIC从Rx Ring中接收到封包之后，具有最好的性能。对于其它类型的网络接口，XDP钩子则在更高的层次实现，性能相对较差。

我们这里用一对veth来测试：

```
sudo ip link add dev veth0 type veth peer name veth1
sudo ip link set up dev veth0
sudo ip link set up dev veth1

# 挂钩eBPF程序到接口上，程序必须写在prog段中
sudo ip link set dev veth1 xdp object xdp_ipv6_filter.o

# 查看接口信息
sudo ip link show veth1
8: veth1@veth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc noqueue
state UP mode DEFAULT group default qlen 1000
    link/ether 32:05:fc:9a:d8:75 brd ff:ff:ff:ff:ff:ff
    # 可以看到xdp标记
    prog/xdp id 32 tag bdb81fb6a5cf3154 jited
```

为了验证eBPF程序能正常工作，我们通过tcpdump回放一段录制的封包：

```
# 在veth1上录制IPv6封包，如果程序正常工作，应该接收不到
sudo tcpdump "ip6" -i veth1 -w captured.pcap -c 10
tcpdump: listening on veth1, link-type EN10MB (Ethernet), capture size 262144
bytes
# 预期结果：
# 10 packets captured
# 10 packets received by filter
# 0 packets dropped by kernel


# 将IPv4/IPv6混合流量（此pcap中包含10个IPv4，10个IPv6）回放到veth0
# q由于eBPF程序的存在IPv6的封包应该不会发到veth1
sudo tcpreplay -i veth0 ipv4-and-ipv6-data.pcap
```

在/sys/kernel/debug/tracing/trace_pipe 中可以看到日志：

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
tcpreplay-4496  [003] ..s1 15472.046835: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046847: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046855: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046862: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046869: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046878: 0: Debug: eth_type:0x800
tcpreplay-4496  [003] ..s1 15472.046885: 0: Debug: eth_type:0x800
tcpreplay-4496  [003] ..s1 15472.046892: 0: Debug: eth_type:0x800
tcpreplay-4496  [003] ..s1 15472.046903: 0: Debug: eth_type:0x800
tcpreplay-4496  [003] ..s1 15472.046911: 0: Debug: eth_type:0x800
...
```

# BCC

## 转换为CO-RE

BCC提供了开发可移植的BPF程序的一套方法，但是比起新出现的BPF CO-RE有不少缺点，建议切换为BPF CO-RE。

### 过渡

如果你需要同时支持BCC和CO-RE，可以利用`BCC_SEC`宏，对于BCC来说，它会自动定义：

```
#ifdef BCC_SEC
#define __BCC__
#endif

#ifdef __BCC__
/* BCC-specific code */
#else
/* libbpf-specific code */
#endif
```

### 头文件

使用BPF CO-RE时，你不需要包含任何内核头文件（`#include <linux/whatever.h>`），而仅仅需要包含一个`vmlinux.h`即可。

```
#ifdef __BCC__
/* linux headers needed for BCC only */
#else /* __BCC__ */
#include "vmlinux.h"                /* all kernel types */
#include <bpf/bpf_helpers.h>        /* most used helpers: SEC, __always_inline,
etc */
#include <bpf/bpf_core_read.h>      /* for BPF CO-RE helpers */
#include <bpf/bpf_tracing.h>        /* for getting kprobe arguments */
#endif /* __BCC__ */
```

由于vmlinux.h中可能缺少一部分内核通过#define定义的常量，因此你可能需要重新声明他们，这些变量中最常用的，已经声明在`bpf_helpers.h`中了。

## 字段访问

BCC会自动将`tsk->parent->pid`这样的point chasing转换为bpf_probe_read()调用，CO-RE则没有这么好用功能，你需要使用`bpf_core_read.h`中定义的助手函数/宏，并将上述表达式改写为`BPF_CORE_READ(tsk, parent, pid)`形式。

从5.5+开始，`tp_btf`和`fentry`/`fext`类型的BPF程序类型，可以使用原生的C指针访问语法。考虑你的目标环境。

`BPF_CORE_READ`和BCC兼容，也就是说，你可以统一使用该宏，避免重复代码。

## 声明BPF Maps

下面给出BCC和libbpf声明Map语法的对比：

```
/* Array */
#ifdef __BCC__
BPF_ARRAY(my_array_map, struct my_value, 128);
#else
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 128);
    __type(key, u32);
    __type(value, struct my_value);
} my_array_map SEC(".maps");
#endif

/* Hashmap */
#ifdef __BCC__
BPF_HASH(my_hash_map, u32, struct my_value);
#else
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 10240);
    __type(key, u32);
    __type(value, struct my_value);
} my_hash_map SEC(".maps")
#endif

/* per-CPU array */
#ifdef __BCC__
BPF_PERCPU_ARRAY(heap, struct my_value, 1);
#else
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __uint(max_entries, 1);
    __type(key, u32);
    __type(value, struct my_value);
} heap SEC(".maps");
#endif
```

注意：BCC中的Map默认大小一般是10240，使用libbpf时则需要精确的设置。

PERF_EVENT_ARRAY、STACK_TRACE以及一些其它的特殊Map（DEVMAP、CPUMAP）尚不支持BTF类型的键值，需要直接指定key_size/value_siz：

```
/* Perf event array (for use with perf_buffer API) */
#ifdef __BCC__
BPF_PERF_OUTPUT(events);
#else
struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(u32));
    __uint(value_size, sizeof(u32));
} events SEC(".maps");
#endif
```

## 访问BPF Maps

BCC使用伪C++风格的语法，这些语法会自动被改写为BPF助手函数调用。使用CO-RE时候，你需要将：

```
some_map.operation(some, args)
```

改写为：

```
bpf_map_operation_elem(&some_map, some, args);
```

下面是一些例子：

```
#ifdef __BCC__
    struct event *data = heap.lookup(&zero);
#else
    struct event *data = bpf_map_lookup_elem(&heap, &zero);
#endif

#ifdef __BCC__
    my_hash_map.update(&id, my_val);
#else
    bpf_map_update_elem(&my_hash_map, &id, &my_val, 0 /* flags */);
#endif

#ifdef __BCC__
    events.perf_submit(args, data, data_len);
#else
    bpf_perf_event_output(args, &events, BPF_F_CURRENT_CPU, data, data_len);
#endif
```

## 注册BPF Program

使用CO-RE时必须用 `SEC()` 宏标记满足约定的段名：

```
#if !defined(__BCC__)
SEC("tracepoint/sched/sched_process_exec")
#endif
int tracepoint__sched__sched_process_exec(
#ifdef __BCC__
    struct tracepoint__sched__sched_process_exec *args
#else
    struct trace_event_raw_sched_process_exec *args
#endif
) {
/* ... */
}
```

常用的约定例如：

`tp/<category>/<name>` 用于tracepoint

`kprobe/<func_name>` 用于kprobe；`kretprobe/<func_name>` 用于kretprobe

`raw_tp/<name>` 用于raw tracepoint

`cgroup_skb/ingress`，`cgroup_skb/egress`，`cgroup/<subtype>`

## 关于tracepoint

BCC和libbpf关于tracepoint上下文类型的命名有所不同。BCC的格式是 `tracepoint__<category>__<name>`，BCC会在运行时编译的时候，自动生成对应的C类型。

libbpf则没有这种自动生成类型的能力，不过内核已经提供了类似的，包含了所有tracepoint数据的类型，其命名为 `trace_event_raw_<name>`。

某些内核中的tracepoint会复用公共类型，因此上述类型对应关系不一定可用。例如没有 `trace_event_raw_sched_process_exit`，你需要使用 `trace_event_raw_sched_process_template`，具体需要关注内核源码或者vmlinux.h。

BCC和libbpf访问tracepoint上下文大部分字段，用同样的字段名。除了一些可变长度字符串字段。BBC是 `data_loc_<some_field>` 而libbpf是 `__data_loc_<some_field>` 格式。

## 关于kprobe

Kprobe BPF程序以一个 `struct pt_regs` 作为上下文参数，BCC支持直接声明为函数参数，libbpf则需要借助 `BPF_KPROBE` / `BPF_KRETPROBE` 宏：

```
#ifdef __BCC__
int kprobe__acct_collect(struct pt_regs *ctx, long exit_code, int group_dead)
#else
SEC("kprobe/acct_collect")
int BPF_KPROBE(kprobe__acct_collect, long exit_code, int group_dead)
#endif
{
    // 对于libbpf，在这里也可以访问名为ctx的pt_regs*
}
```

另外一个需要注意的点，在4.17+，系统调用函数发生了重命名，例如 `sys_kill` 被重命名为 `__x64_sys_kill`（对于x86，其它平台下有不同的前缀）。使用kprobe时需要注意到这种变化，如果可能，尽量使用tracepoint。

从5.5+开始，可以考虑用`raw_tp`/`fentry`/`fexit`代替`tracepoint`/`kprobe`/`kretprobe`。这些新探针具有更好的性能、易用性。

### 处理编译时#if

在BCC代码中依赖预处理器（#ifdef 或者 #if）是流行的做法，原因通常是为了适配不同版本的内核、根据程序配置启用/禁用代码片段。CO-RE提供类似的能力：Kconfig externs和struct flavors。

# 附录

## 相关项目

### BCC

参考：基于BCC进行性能追踪和网络监控

BCC（BPF Compiler Collection，BPF编译器集合）是一个工具箱，它提供了很多追踪用途的BPF程序，加载这些程序时使用BCC提供的Python接口。

### bpftrace

一个DTrace风格的动态追踪工具，使用LLVM作为后端，将脚本编译为BPF字节码，并使用BCC和内核的BPF tracing基础设施进行交互。和原生BCC相比，它提供了更加高级的、用于实现追踪脚本的语言。

### perf

参考：利用perf剖析Linux应用程序

perf_events是内核2.6+的一部分，用户空间工具perf在包linux-tools-common中。它有三种工作模式，其中最新的是4.4引入的BPF模式，支持载入tracing BPF程序。

## 参考


1. Dive into BPF: a list of reading material
2. https://docs.cilium.io/en/v1.9/bpf
3. list of eBPF helper functions