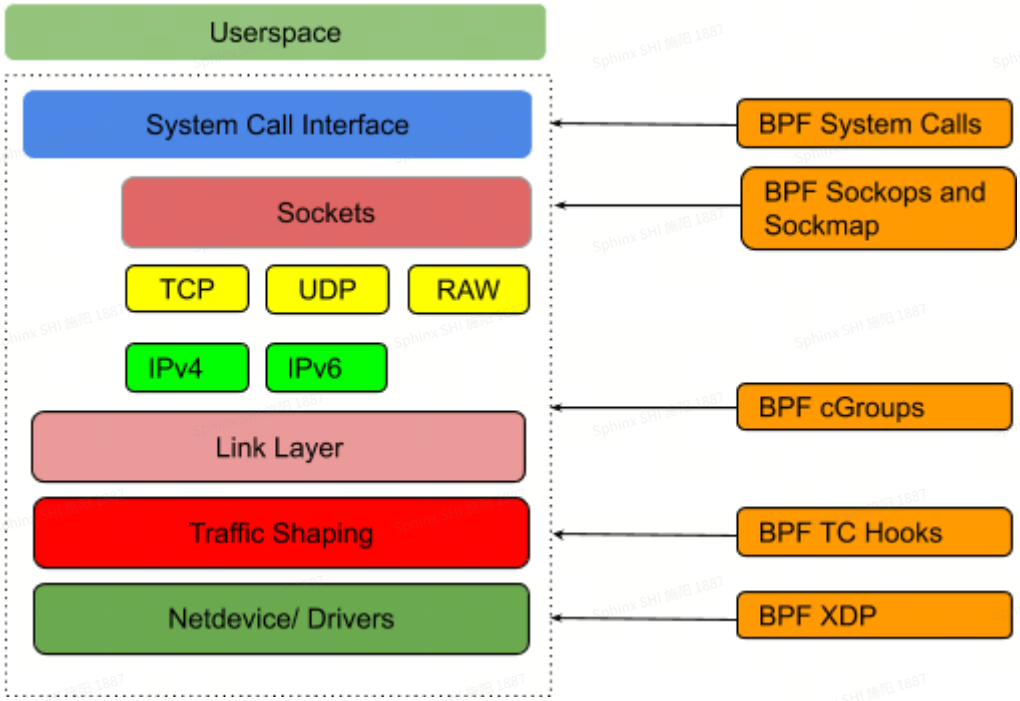
Linux eBPF socket 重定向

1. 原理



每个eBPF程序都属于特定的类型，不同类型eBPF程序的出发事件是不同的。

网络类eBPF程序可以分为XDP程序、TC程序、套接字程序以及cgroup程序:

- XDP：在网络驱动程序刚刚收到数据包的时候触发执行，支持卸载到网卡硬件，常用于**防火墙**和**四层负载均衡**
- TC：在网卡队列接收或发送的时候触发执行，运行在内核协议栈中，常用于流量控制
- 套接字：在套接字发生创建、修改、收发数据等变化的时候触发执行，运行在内核协议栈中，常用于过滤、观测或重定向套接字网络包。
  其中 `BPF_PROG_TYPE_SOCK_OPS` 、 `BPF_PROG_TYPE_SK_SKB` 、 `BPF_PROG_TYPE_SK_MSG` 等都可以用于套接字重定向
- cgroup：在cgroup内所有进程的套接字创建、修改选项、连接等情况下触发执行，常用于过滤和控制cgroup内多个进程的套接字

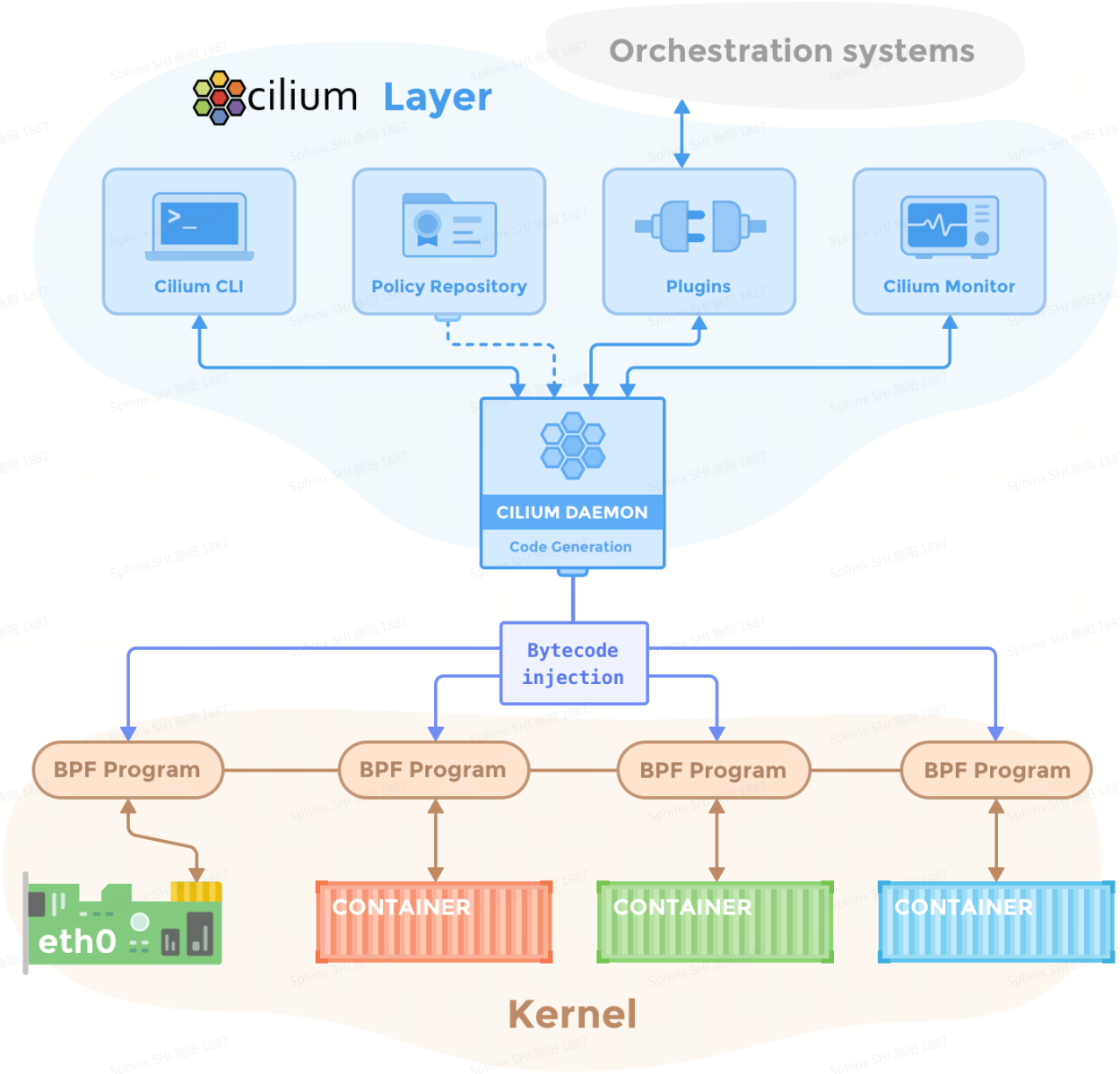
因此针对网络转发的优化，通常可以在XDP与套接字阶段进行优化。

> XDP的性能往往是最好的

2. 实践

对于源和目的端都在同一台机器的应用来说，可以通过eBPF绕过整个TCP/IP协议栈，直接将数据发送到socket对端（原理与Cilium相仿）。

> https://github.com/cilium/cilium



2.1 优化细节

套接字eBPF程序工作在内核空间中，无需把网络数据发送到用户空间就能完成转发。

具体来说，使用套接字映射转发网络包需要以下几个步骤：

1. 创建套接字映射（即全局映射表记录所有的socket信息）
2. 在 `BPF_PROG_TYPE_SOCK_OPS` 类型的 eBPF 程序中，将新创建的套接字存入套接字映射中
3. 在流解析类的 eBPF 程序（如 `BPF_PROG_TYPE_SK_SKB` 或 `BPF_PROG_TYPE_SK_MSG` ）中，从套接字映射中提取套接字信息，并调用 BPF 辅助函数转发网络包
4. 加载并挂载eBPF程序到套接字事件

2.2 Code

2.2.1 socketops

1. 系统中有 socket 操作时（例如 connection establishment、tcp retransmit 等），触发执行
2. 执行逻辑：提取 socket 信息，并以 key \& value 形式存储到 sockmap

代码块

```
__section("sockops") // 加载到 ELF 中的 `sockops` 区域，有 socket operations 时触发执行
int bpf_sockmap(struct bpf_sock_ops *skops)
{
    switch (skops->op) {
        case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB: // 被动建连
        case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:  // 主动建连
            if (skops->family == 2) {             // AF_INET
                bpf_sock_ops_ipv4(skops);         // 将 socket 信息记录到到 sockmap
            }
            break;
        default:
            break;
    }
    return 0;
}
```

对于两端都在本节点的socket来说，这段代码会执行两次

- 源端发送 SYN 时会产生一个事件，命中 case 2
- 目的端发送 SYN+ACK 时会产生一个事件，命中 case 1

因此对于每一个成功建连的 socket，sockmap 中会有两条记录（key 不同）

2.2.2 socket map

代码块

```
static inline
void bpf_sock_ops_ipv4(struct bpf_sock_ops *skops)
{
    struct sock_key key = {};
    int ret;

    extract_key4_from_ops(skops, &key);

```

```
9       ret = sock_hash_update(skops, &sock_ops_map, &key, BPF_NOEXIST);
10      if (ret != 0) {
11          printk("sock_hash_update() failed, ret: %d\n", ret);
12      }
13
14      printk("sockmap: op %d, port %d --> %d\n", skops->op, skops->local_port,
    bpf_ntohl(skops->remote_port));
15  }
```

1. 调用 `extract_key4_from_ops()` 从 `struct bpf_sock_ops *skops（socket metadata）` 中提取 key
2. 调用 `sock_hash_update()` 将 key:value 写入全局的 `sockmap sock_ops_map`，这 个变量定义在我们的头文件中

2.2.3 sockmap key

map 的类型可以是：

- BPF_MAP_TYPE_SOCKMAP
- BPF_MAP_TYPE_SOCKHASH

sockmap，key定义如下：

代码块

```
1   struct bpf_map_def __section("maps") sock_ops_map = {
2           .type           = BPF_MAP_TYPE_SOCKHASH,
3           .key_size       = sizeof(struct sock_key),
4           .value_size     = sizeof(int),              // 存储 socket
5           .max_entries    = 65535,
6           .map_flags      = 0,
7   };
8
9   struct sock_key {
10          uint32_t sip4;    // 源 IP
11          uint32_t dip4;    // 目的 IP
12          uint8_t  family;  // 协议类型
13          uint8_t  pad1;    // this padding required for 64bit alignment
14          uint16_t pad2;    // else ebpf kernel verifier rejects loading of the
    program
15          uint32_t pad3;
16          uint32_t sport;   // 源端口
17          uint32_t dport;   // 目的端口
18  } __attribute__((packed));
```

key 的提取

代码块

```
static inline
void extract_key4_from_ops(struct bpf_sock_ops *ops, struct sock_key *key)
{
    // keep ip and port in network byte order
    key->dip4 = ops->remote_ip4;
    key->sip4 = ops->local_ip4;
    key->family = 1;

    // local_port is in host byte order, and remote_port is in network byte order
    key->sport = (bpf_htonl(ops->local_port) >> 16);
    key->dport = FORCE_READ(ops->remote_port) >> 16;
}
```

> 使用sock_hash_update() 将 socket 信息写入到 sockmap

2.2.4 socket 重定向

功能需求

1. 拦截所有的 sendmsg 系统调用，从消息中提取 key
2. 根据 key 查询 sockmap，找到这个 socket 的对端，然后绕过 TCP\/IP 协议栈，直接将 数据重定向过去

要完成这个功能，需要：

1. 在 socket 发起 sendmsg 系统调用时触发执行
2. 关联到前面已经创建好的 sockmap，因为要去里面查询 socket 的对端信息

拦截sendmsg系统调用

代码块

```
__section("sk_msg") // 加载目标文件（ELF ）中的 `sk_msg` section，`sendmsg` 系统调用时触发执行
int bpf_redir(struct sk_msg_md *msg)
{
    struct sock_key key = {};
    extract_key4_from_msg(msg, &key);
    msg_redirect_hash(msg, &sock_ops_map, &key, BPF_F_INGRESS);
    return SK_PASS;
}
```

当 attach 了这段程序的 socket 上有 sendmsg 系统调用时，内核就会执行这段代码。它会：

1. 从 socket metadata 中提取 key
2. 调用 `bpf_socket_redirect_hash()` 寻找对应的 socket，并根据 flag（BPF_F_INGRESS），将数据重定向到 socket 的某个 queue

提取 key

从 socket message 中提取 key

代码块

```
static inline
void extract_key4_from_msg(struct sk_msg_md *msg, struct sock_key *key)
{
    key->sip4 = msg->remote_ip4;
    key->dip4 = msg->local_ip4;
    key->family = 1;

    key->dport = (bpf_htonl(msg->local_port) >> 16);
    key->sport = FORCE_READ(msg->remote_port) >> 16;
}
```

msg_redirect_hash() 也是我们定义的一个宏，最终调用的是 BPF 内置的辅助函数