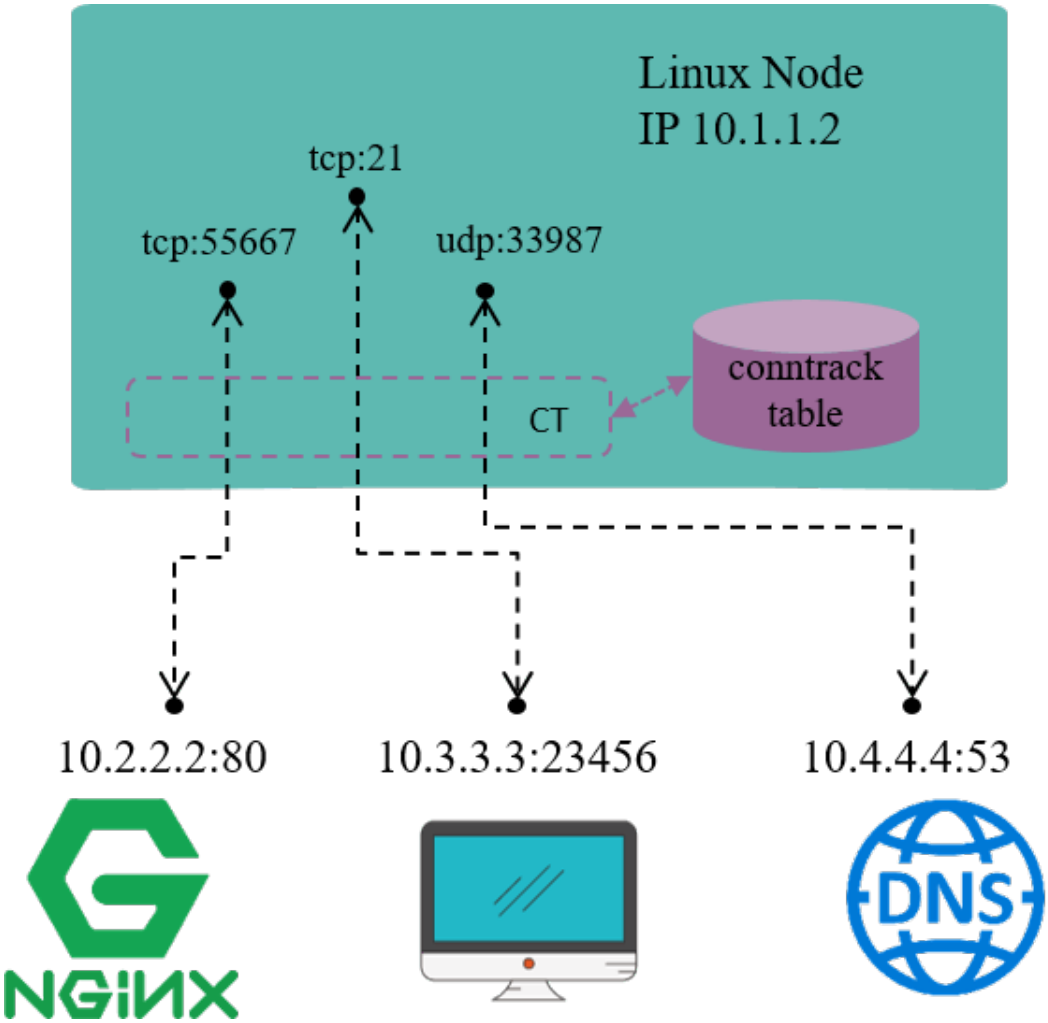
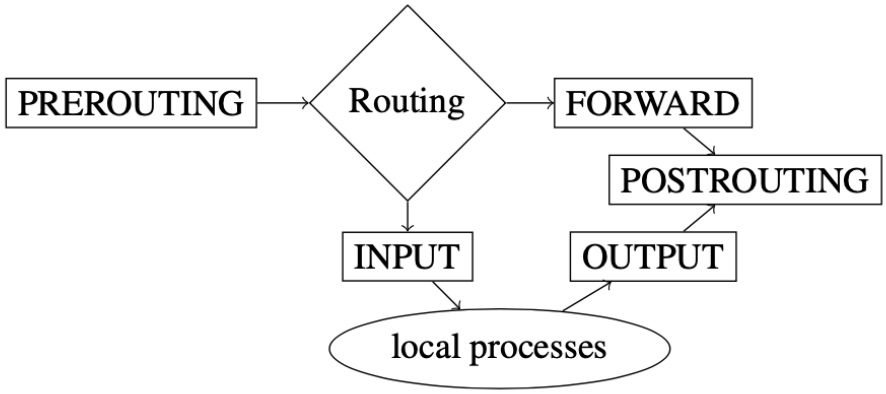


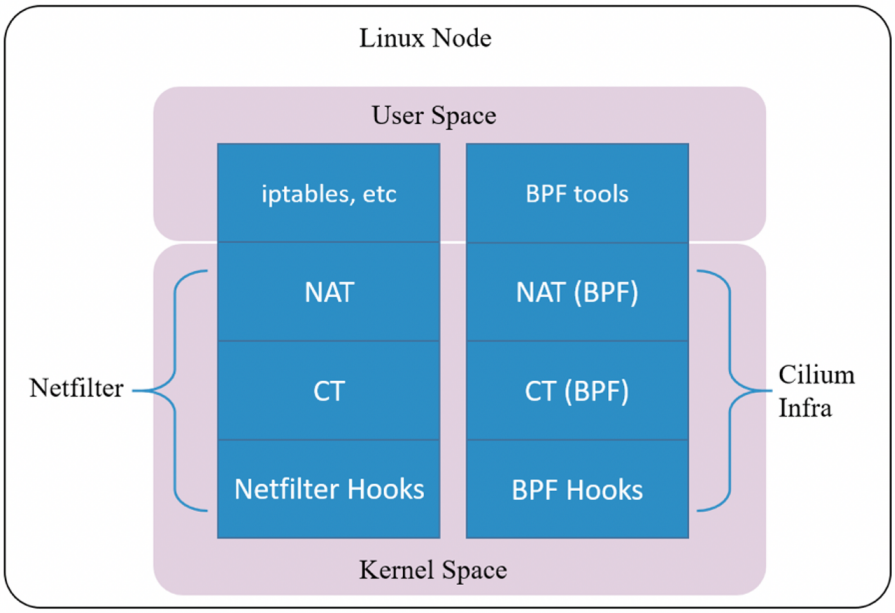
程序创坊
Author: cugriver@163.com

项目背景

连接跟踪（conntrack）是网络应用非常非常的基础，比如有状态防火墙（firewall），网络地址转换（nat）,负载均衡（lb）。Linux conntrack 是基于 netfilter 实现的，如图所示，分别在 PREROUTING, POSTROUTING 位置前和后对网络报文进行跟踪；



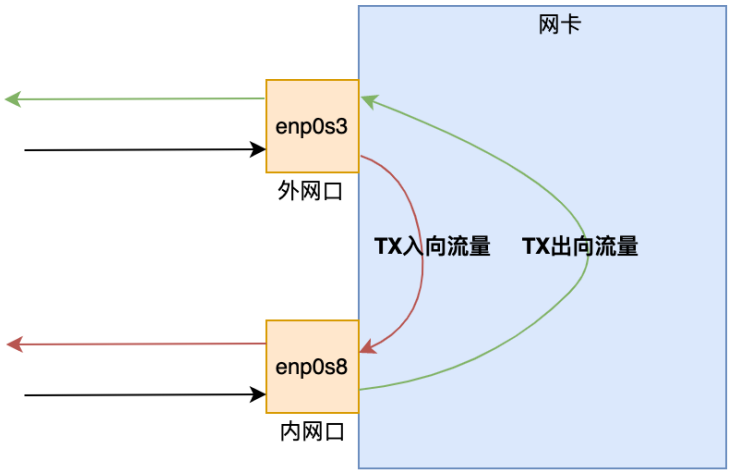
但是 XDP 位置在进入网络栈之前，无法利用到内核栈的 conntrack 能力，Cilium 应该遇到同样的问题， 所以 Cilium 基于 eBPF 实现了 conntrack；如图所示，



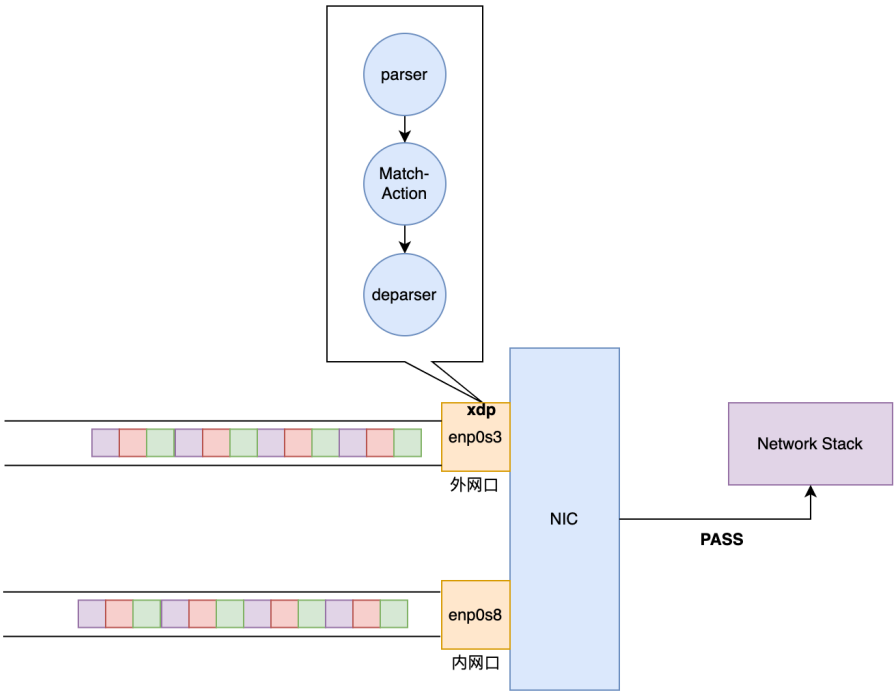
换句话说，只要具备 Hook 能力，能拦截进出主机的每个报文，完全可以实现一套连接跟踪功能， 这个该项目的核心思路。

项目设计

从网络数据路径上分析， Linux XDP HOOK 只能处理入向报文，无法处理主机主动出向报文，所以该项目只能做网络数据转发。网络报文是双向的， 为了区分出入向流量，需要两张网络设备，如图所示：



要实现连接跟踪，首先要解析网络报文，提取 Flow 项（五元组），然后建立连接信息数据库（conntrack table），最后拦截网络报文信息，不断更新数据库。如图所示



将 eBPF 程序挂载到网络设备上，Parse-Match Action-Deparse 每一个网络报文，遇到处理不了的可以送到内核栈处理，可以成为网络栈的一种延伸。

Conntrack Table	ebpf map
Parer/Deparser	bpf_tail_call
Match Action	todo

BPF Map 定义

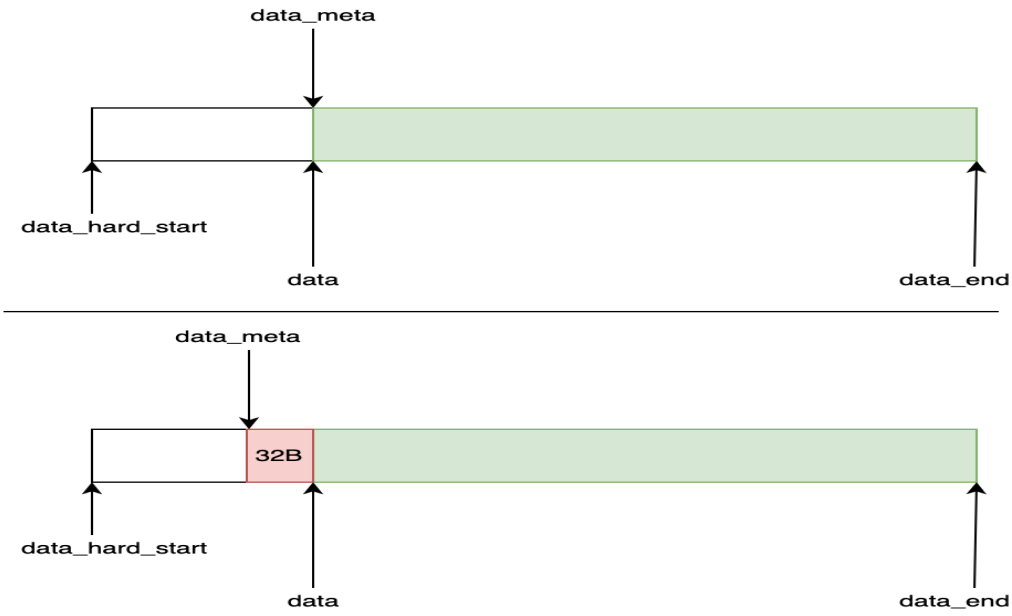
- meta ： 存储元数据，每个序号都要含义，比如 Index:0 表示系统启动时间 ktime
- jt ： Jump Table, 用于 bpf_tail_call 程序跳转
- ctt ： Conntrack Table 存储 flow

```
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key,  __u32);
    __type(value,__u64);
    __uint(max_entries, 4);
} meta SEC(".maps"); //meta table

struct {
   __uint(type,         BPF_MAP_TYPE_PROG_ARRAY);
   __type(key,          0x04);
   __type(value,        0x04);
   __uint(max_entries,  30);
} jt SEC(".maps");      //jump table

struct {
   __uint(type,         BPF_MAP_TYPE_LRU_HASH);
   __type(key,          struct ipv4_ct_tuple);
   __type(value,        struct ipv4_ct_entry);
   __uint(max_entries,  1024);
} ctt SEC(".maps");      //conntrack table
```

BPF Prog 定义

```
PROG(ingress)(struct xdp_md *ctx) {
    // 处理入向报文
}
PROG(egress)(struct xdp_md *ctx) {
    // 处理出向报文
}
PROG(prs_eth)(struct xdp_md *ctx) {
    // 解析以太网协议
}
PROG(prs_ipv4)(struct xdp_md *ctx) {
    // 解析 IPv4 协议
}
PROG(prs_icmp)(struct xdp_md *ctx) {
    // 解析 ICMP 协议
}
PROG(prs_tcp)(struct xdp_md *ctx) {
    // 解析 TCP 协议
}
PROG(prs_udp) (struct xdp_md *ctx) {
    // 解析 UDP 协议
}
PROG(ct)(struct xdp_md *ctx) {
    // 处理 CT
}
PROG(prs_end)(struct xdp_md *ctx) {
    // Deparser
}
```

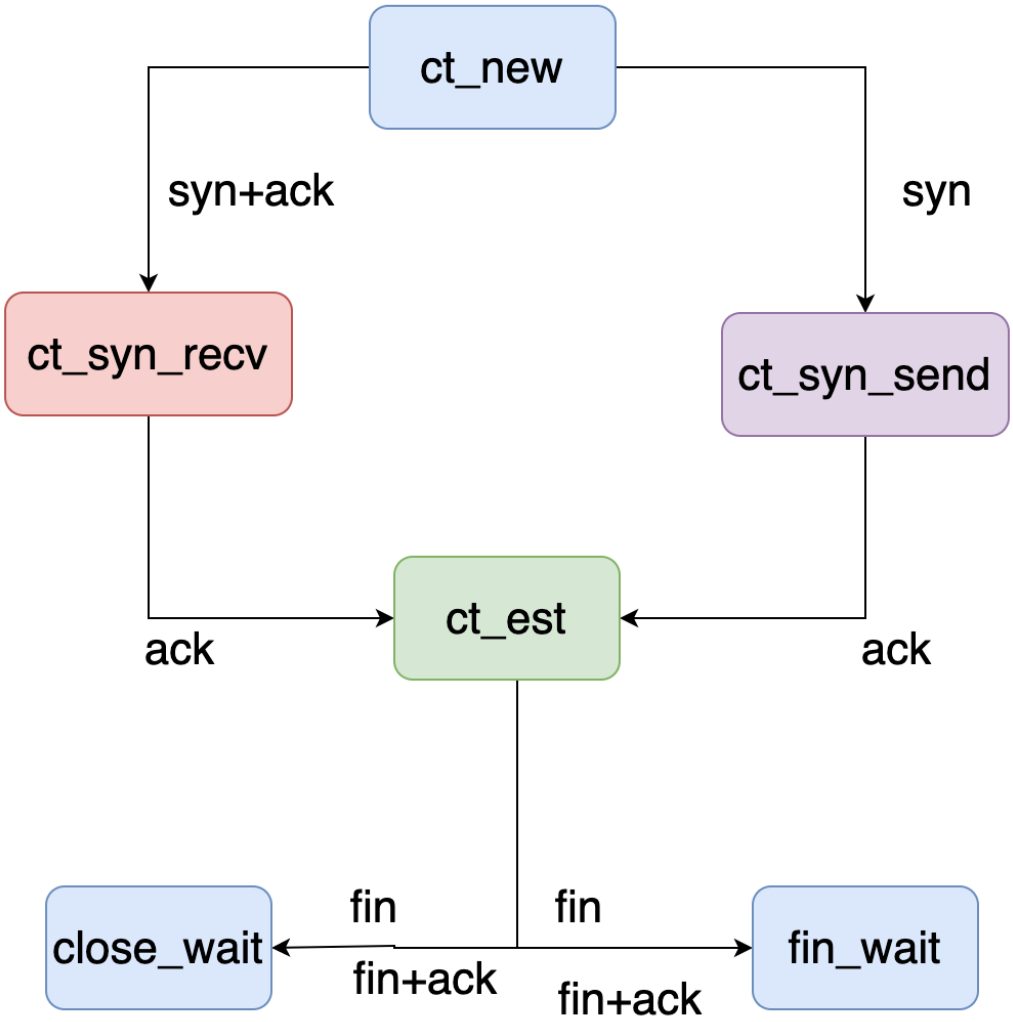
Parser 逐层解析报文，例如，解析以太网、解析 IPv4、解析 ICMP/TCP/UDP 报文。每个 Prog eBPF 程序入参是 `struct xdp_md *ctx`，是如何在 `PROG(prs_ipv4)` 中避免重复解析 eth。 基于 bpf_xdp_adjust_meta 函数在报文前扩展内置协议。此项目，扩展内置协议是 `struct flow` 存储 Flow 需要的参数。




```
struct xdp_buff {
      void *data;
      void *data_end;
      void *data_meta;
      void *data_hard_start;
      struct xdp_rxq_info *rxq;
   };

struct flow {
    __be32  sip;
    __be32  dip;
    __be32  sport:16,
            dport:16;
    __u32   proto:8,
            delta:8,
            bytes:16;
    __u32   urg:1,
            ack:1,
            psh:1,
            rst:1,
            syn:1,
            fin:1,
            rel:1,
            dir:1,
            reserved:24;
} __attribute__((packed));
```

CT 状态转换

CT 状态转换跟踪 flow 信息，像 UDP,ICMP 等无状态相对 TCP 更简单，TCP 本身面向连接且有状态转换相对复杂。TCP 的连接跟踪状态转换图如下：



实验环境

Namespace 网络环境

```
#创建 namespace
ip netns del ns1
ip netns del ns2

ip netns add ns1
ip netns add ns2
```

```
#创建 veth 虚拟网络设备
ip link add ns1-wan type veth peer name ns1-lan
ip link add ns2-wan type veth peer name ns2-lan

ip link set ns1-lan up
ip link set ns1-wan up
ip link set ns2-lan up
ip link set ns2-wan up

ip link set ns1-wan netns ns1
ip link set ns2-wan netns ns2

ip netns exec ns1  ip link set dev ns1-wan up
ip netns exec ns2  ip link set dev ns2-wan up

ip addr add 172.20.56.1/24 brd 172.20.56.255 dev ns1-lan
ip addr add 172.30.56.1/24 brd 172.30.56.255 dev ns2-lan

sysctl -w net.ipv4.conf.all.forwarding=1
sysctl -w net.ipv6.conf.all.forwarding=1

ip netns exec ns1 ip addr add 172.20.56.105/24 brd 172.20.56.255 dev ns1-wan
ip netns exec ns1 ip route add default via 172.20.56.1 dev ns1-wan

ip netns exec ns2 ip addr add 172.30.56.105/24 brd 172.30.56.255 dev ns2-wan
ip netns exec ns2 ip route add default via 172.30.56.1 dev ns2-wan
```

安装 CT 程序

```
dpkg -i ct-0.0.1-0-x86_64.deb
systemctl start ct
```

挂载 eBPF 程序

```
ip link set dev ns1-lan xdp pinned /sys/fs/bpf/prog/xdp_0
ip link set dev ns2-lan xdp pinned /sys/fs/bpf/prog/xdp_1
```

源代码

```
https://github.com/advancevillage/ct
```