Overview of BPF networking hooks

+ User experience in Meta
Linux network stack
(super high level)
SEC(“xdp”)

Where is it run?

- Running at driver (ingress)
- Before skb creation. Speed!
SEC("xdp")

Header handling

- \([\text{xdp->data}, \text{xdp->data_end})\]
- Multi-buffer: bpf_xdp_{load,store}_bytes()
- encap or decap: bpf_xdp_adjust_{head,tail}
SEC("xdp")

Actions on the packet (xdp_md)

- XDP_PASS to pass up to the kernel stack
- XDP_TX to send it out at the same interface
- bpf_xdp_redirect() to send out at a different interface or cpu
  - devmap and cpumap
SEC(“xdp”)

Where is the sk?

- Routing is not done yet. No sk.
- bpf_sk[c]_lookup_{tcp,udp}
  - TCP SYN-ACK reply with syn-cookie
SEC("tc")

Where is it run?

- tc ingress (after gro) or
- tc egress (before gso)
SEC(“tc”) header handling

- Directly through skb->[data, data_end)
- bpf_skb_pull_data() and bpf_skb_{load,store}_bytes() for the non-linear part
- encap and decap: bpf_skb_adjust_room()
SEC(“tc”)  

With skb, there are more, eg.

- skb->hwtstamp for ingress
- skb->protocol
- ...etc
**SEC(“tc”)**

**Action on skb**

- **TC_ACT_OK**
- **TC_ACT_SHOT**
- **TC_ACT_REDIRECT + bpf_redirect*()**
  - eg. fast path to redirect packets between phy-eth to container veth
- At egress, set the skb->tstamp (EDT) + fq
  - Bandwidth shaping. Flexible and fast.
SEC(“tc”)

Where is the sk?

- **Ingress**
  - No (routing is not done).
  - bpf_sk_lookup*(). With bpf_sk_assign(), earlier demux.
- **Egress**
  - skb->sk

```plaintext
sendmsg,
recvmsg,
connect,
...
TCP/UDP
IP[6J +
Routing
TC
GRO / GSO
Driver
NIC HW
```
IP[46] and TCP/UDP

UDP only
- cgroup/sendmsg[46]

TCP/UDP
- cgroup/bind, connect[46]
- cgroup/get(sock, peer)name[46]
- cgroup/(get, set)sockopt

TCP only
- bpf-tcp-cc
- (cgroup)/sockops

- cgroup_skb/egress

TCP/UDP
- cgroup_skb/ingress

- sk_reuseport (attach to a sk)

- sk_lookup (attach to netns)

Routing locally
SEC(“sk_lookup”), a netns hook

- SEC(“sk_lookup”) bpf prog can pick a sk different from the skb’s ip/port
- TCP: A listening socket
- UDP: Any sk that is ready to receive packet
- skb is available. Read only.
SEC(“sk_reuseport”), attach to sk

- Reuseport sk(s) is a group of sockets bind()-ed to the same IP[6] and port
- By default, kernel picks one by hash
SEC(“sk_reuseport”)

SEC(“sk_reuseport”) bpf prog can:

- Pick the sk by numa node
- Avoid picking the sk that its process is exiting. eg. process restart.
- Even migrate the not-yet accepted TCP sk from a closing listen sk to another sk
- Connection-ID in QUIC
- skb is available. Read only.
SEC(“cgroup/{ingress,egress}”)

- IPv6 and IPv4 skb only
- skb is available. Read only.
- sk is available. skb->sk
SEC("cgroup/{ingress,egress}""")

• Ingress
  – Reject packets based on container policy

• Egress
  – Set the delivery time (EDT) in skb->tstamp (+ fq) to limit bandwidth usage per cgroup
  – Return NET_XMIT_{DROP, CN} to signal the tcp stack to tcp_enter_cwr()
SEC(".struct_ops") aka bpf-tcp-cc

- A TCP CC (congestion control) fully implemented in BPF
- Enable congestion control experts to test ideas and collect data faster in production
We have >=2 bpf-tcp-cc in production

- One is for background bulk traffic and saving $$$ by not over provisioning bandwidth.
- One is bpf_dctcp that has different ongoing experiments
- Ideas on using one-way-delay is also brewing
SEC(".struct_ops") aka bpf-tcp-cc

bpf-tcp-iter +
bpf_setsockopt(TCP_CONGESTION)

• Retire old cc faster on the long-lived connection
SEC("sockops")

One logical bpf hook but different callbacks:

- BPF_SOCK_OPS_TCP_LISTEN_CB
- BPF_SOCK_OPS_TCP_CONNECT_CB
- BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB
- BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB
  - bpf_setsockopt(TCP_CONGESTION) based on peer address
  - Fallback to cubic, bbr, or others instead of the default reno if ECN is not supported
SEC("sockops"), contd

- BPF_SOCK_OPS_TIMEOUT_INIT
  - Configure initial RTO for SYN and SYN-ACK
- BPF_SOCK_OPS_*_HDR_OPT_CB
  - Read and write tcp header option
  - Add server id in SYN (similar to the connection-id in QUIC)
  - Add max delay ack + bpf_setsockopt(TCP_BPF_DELACK_MAX or TCP_BPF_RTO_MIN)
cgroup bpf hooks at the syscall code path

- Bind, connect[46], sendmsg[46], get{sock,peer}name[46]
  - Change the local or remote address
- {set,get}sockopt()
  - Do extra setsockopt
    - userspace setsockopt(IPV6_TCLASS, background_dscp) and the bpf prog does bpf_setsockopt(TCP_CONGESTION, background_cc).
    - Userspace invent new sockopt, supported by the bpf sk storage
- The recent bpf lsm cgroup hooks (by Stanislav Fomichev) enabled more syscall code path
Q: The bpf prog has a sk. Why a sk bpf helper is not available?

- A cgroup prog SEC("sockops"). Where is bpf_sk_cgroup_id()?
- SEC("tc") can call bpf_get_socket_cookie(skb)
  - but cannot do "sk = bpf_sk_lookup_tcp();" and then "bpf_get_socket_cookie(sk);"
Q: When calling a “sk” helper, should ctx be passed or sk be passed?

eg. long bpf_setsockopt(void *bpf_socket, ....) and the comments:

• The same for bpf_get_socket_cookie. bpf.h has:
  – bpf_get_socket_cookie(struct bpf_sock_addr *ctx)
  – bpf_get_socket_cookie(struct bpf_sock_ops *ctx)
  – bpf_get_socket_cookie(struct sock *sk)
  – However, there is always ctx->sk for both bpf_sock_addr and bpf_sock_ops case.
Q: If the SEC("tc") prog can access the cgroup (skb->sk), can it access the bpf cgroup storage?
A: No. The tc prog is not a cgroup prog.

Q: Why sk, task, and inode storage is not limited by prog types? Even tracing bpf prog can use it.
A: ...

Q: A bpf prog can access multiple sk storage map. Why a cgroup bpf prog can only access one cgroup storage map?
A: ...