BPF datapath extensions for K8s workloads

Daniel Borkmann & Martynas Pumpcutis, Cilium.io
Cilium’s Load Balancer in one picture

Main principle: Operating as close as possible to the socket for E-W and as close as possible to the driver for N-S.

- Handles external traffic (N-S) for services
  - Consistent hashing through Maglev
  - DSR or SNAT for remote backends
  - Wildcarded IPv4/v6 n-tuple based PCAP exporter with ingress & egress observability points

- Handles internal traffic (E-W) for services
  - Uses connect(), sendmsg(), recvmsg(), getpeername() BPF cgroup v2 hooks
  - Handles service backend health-probing
  - Hooks into special bind() & tc/BPF logic to then locally craft health checking packets

- K8s / L4LB Node
- BPF at socket layer
- BPF L4LB at XDP/tc layer

Client

MAC | IP | TCP/UDP

Backend

MAC | IP | IP | TCP/UDP
Agenda: Ongoing development items

➔ Part 1: The cgroup v1/v2 interference problem
➔ Part 2: TCP pacing for Pods from initns
➔ Part 3: Managed neighbor entries and fib extensions
➔ Part 4: Wildcarded BPF map lookups
Part 1: The cgroup v1/v2 interference
Cgroup v2 layout on (bare metal) K8s node

```bash
# tree -d -L 1 /sys/fs/cgroup/
/sys/fs/cgroup/
├── blkio
│   ├── cpu -> cpu,cpuacct
│   │   ├── cpuacct -> cpu,cpuacct
│   │   └── cpu,cpuacct
│   ├── cpuset
│   ├── devices
│   ├── freezer
│   ├── hugetlb
│   ├── memory
│   │   ├── net_cls -> net_cls,net_prio
│   │   │   ├── net_cls,net_prio
│   │   │   └── net_prio -> net_cls,net_prio
│   │   └── perf_event
│   └── pids
│       └── rdma
│           └── systemd
│               └── unified
├── cgroup v2
│   └── socket LB (bind(2))
├── cgroup v1
│   └── socket LB (connect(2), sendmsg(2), recvmsg(2), getpeername(2))
└── Host (initns)
```

Pod nginx
Pod httpd
Cgroup v2 layout on KIND (K8s in Docker)

K8s Node A:

nodens: /sys/fs/cgroup/
initns: /sys/fs/cgroup/docker-node-a/

Pod nginx
Pod httpd
veth0
veth1
br0

K8s Node B:

nodens: /sys/fs/cgroup/
initns: /sys/fs/cgroup/docker-node-b/

Pod client
veth0
veth2

Node running KIND
initns: /sys/fs/cgroup/
Cgroup v2 layout on KIND (K8s in Docker)
Cgroup v1/v2 interference: Context

The case for saving 8 byte in the socket structure

→ Assumption back in 2015: “no reason to mix cgroup v1/v2”
  ○ struct sock_cgroup_data is a union with v1/v2 data
  ○ cgroup v1 net_cls/net_prio tags vs cgroup v2 pointer

→ Reality check: Environments today have both flavors mounted
Cgroup v1/v2 interference: Context

Retrieving socket’s cgroup v2 pointer in fast-path:

```c
static inline struct cgroup *sock_cgroup_ptr(struct sock_cgroup_data *skcd)
{
    #if defined(CONFIG_CGROUP_NET_PRIO) || defined(CONFIG_CGROUP_NET_CLASSID)
        unsigned long v;
    
    /*
     * @skcd->val is 64bit but the following is safe on 32bit too as we
     * just need the lower ulong to be written and read atomically.
     */
    v = READ_ONCE(skcd->val);

    if (v & 3)
        return &cgrp_dfl_root.cgrp;

    return (struct cgroup *)(unsigned long)v ?: &cgrp_dfl_root.cgrp;
#else
    return (struct cgroup *)(unsigned long)skcd->val;
#endif
}
```
Cgroup v1/v2 interference: Context

Retrieving socket’s cgroup v2 pointer in fast-path:

```c
static inline struct cgroup *sock_cgroup_ptr(struct sock_cgroup_data *skcd) {
    #if defined(CONFIG_CGROUP_NET_PRIO) || defined(CONFIG_CGROUP_NET_CLASSID)
        unsigned long v;

        /*
         * @skcd->val is 64bit but the following is safe on 32bit too as we
         * just need the lower ulong to be written and read atomically.
         */
        v = READ_ONCE(skcd->val);

        if (v & 3)
            return &cgrp_dfl_root.cgrp;

        return (struct cgroup *)(unsigned long)v ?: &cgrp_dfl_root.cgrp;
    #else
        return (struct cgroup *)(unsigned long)skcd->val;
    #endif
}
```

If cgroup v1 tagging is used on the socket, fallback to cgroup v2 root.
Cgroup v1/v2 interference: Context

Retrieving socket’s cgroup v2 pointer in fast-path:

```c
static inline struct cgroup *sock_cgroup_ptr(struct sock_cgroup_data *skcd)
{
    #if defined(CONFIG_CGROUP_NET_PRIO) || defined(CONFIG_CGROUP_NET_CLASSID)
        unsigned long v;
    
    /*
    * @skcd->val is 64bit but the following is safe on 32bit too as we
    * just need the lower ulong to be written and read atomically.
    */
    v = READ_ONCE(skcd->val);

    if (v & 3)
        return &cgrp_dfl_root.cgrp;
    
    return (struct cgroup *)(unsigned long)v ?: &cgrp_dfl_root.cgrp;

else
    return (struct cgroup *)(unsigned long)skcd->val;

#endif
}
```

If cgroup v1 tagging is used on the socket, fallback to cgroup v2 root.

Problematic for today’s environments!
Cgroup v2 layout on KIND (K8s in Docker)

K8s Node A:
- Pod nginx
- Pod httpd
- veth0
- veth1

K8s Node B:
- Pod client
- veth0
- veth2

nodens: /sys/fs/cgroup/
inits: /sys/fs/cgroup/docker-node-a/
nodens: /sys/fs/cgroup/
inits: /sys/fs/cgroup/docker-node-b/

Host (initns)

Fallback to initns root!

net_cls usage

Socket LB / A

(eBPF)

Socket LB / B

Node running KIND
initns: /sys/fs/cgroup/

(nothing attached here)
Cgroup v2 layout on KIND (K8s in Docker)

**K8s Node A:**
- Pod nginx
- Pod httpd

**K8s Node B:**
- Pod client

**Host (initns):**
- Socket LB / B
- Socket LB / A

- net_cls usage
- BPF cgroup programs are being **bypassed**. No policy enforcement possible!
- Agent on Node A cannot do anything about it.

Node running KIND
- initns: /sys/fs/cgroup/

Fallback to initns root!
Cgroup v1/v2 interference: Recap

v2 cgroup management complex and cumbersome

➔ Incompatible to cgroup namespaces or non-root cgroup paths
➔ v2-to-v1 switch on the socket leaks v2 object references
➔ Unreliable v2 invocation hinders adoption of BPF cgroup programs
  ○ Independent 3rd party agents inevitably step on each other
  ○ Distros usually enable everything for max compatibility
Approach to fixing the cgroup v1/v2 interference

Fix: biting the bullet and detangle the two ...

```c
static inline struct cgroup *sock_cgroup_ptr(struct sock_cgroup_data *skcd)
{
    return skcd->cgroup;
}
```

➔ struct sock_cgroup_data always holds reliable cgroup pointer
➔ Implicitly also addresses the v2 reference count leaks
➔ Fix along with test cases has been [upstreamed](#) recently
Part 2: TCP pacing for Pods from initns
K8s Pod-specific ingress/egress bandwidth annotation:

- Handled by K8s CNI plugins (e.g. Cilium or bandwidth plugin)
- Semantics for rate enforcement points defined by plugin:
  - K8s bandwidth plugin uses combination of ifb & tbf qdisc
  - Cilium natively implements EDT via BPF & fq qdisc for egress
Pod

-kubernetes.io/egress-bandwidth: "50M"

eth0

MQ

FQ

FQ

FQ

FQ

Packet (skb) departure timestamp management

Multi-queue aware Packet scheduler

NIC queues

Q1

Q4
BPF datapath walk-through: Overview forwarding

See LPC 2020 for helper details

Host / initial netns

Upper stack (IP, netfilter / routing, ...)

Pod / own netns

bpf_redirect_peer()

bpf_fib_lookup() + bpf_redirect() or bpf_redirect_neigh()
BPF datapath walk-through: Works today

1) skb->sk preserved across netns switch
2) BPF program at tc ingress marking all Pod traffic into aggregate
3) BPF program at tc egress setting skb->tstamp based on aggregate’s rate limit
4) mq + fq leaf qdiscs, skb->sk from Pod still retained until here

Host / initial netns

Pod / own netns
BPF datapath walk-through: Next steps

Host / initial netns

Pod / own netns

**mq +fq** leaf qdiscs cannot enforce rate for Pod’s socket

$ skb->tstamp = 0 $ reset on netns switch

Setting socket to BBR or SO_MAX_PACING_RATE
BPF datapath walk-through: Next steps

Situation today: Unstable throughput

```
root@apoc:/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
    Recv  Send  Send
Socket  Socket  Message  Elapsed
  Size    Size    Size    Time    Throughput
bytes  bytes  bytes  secs.  10^6bits/sec
  87380   16384   16384  40.04    655.52

root@apoc:/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
    Recv  Send  Send
Socket  Socket  Message  Elapsed
  Size    Size    Size    Time    Throughput
bytes  bytes  bytes  secs.  10^6bits/sec
  87380   16384   16384  40.07    1274.70

root@apoc:/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
    Recv  Send  Send
Socket  Socket  Message  Elapsed
  Size    Size    Size    Time    Throughput
bytes  bytes  bytes  secs.  10^6bits/sec
  87380   16384   16384  40.07    1519.32

root@apoc:/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
    Recv  Send  Send
Socket  Socket  Message  Elapsed
  Size    Size    Size    Time    Throughput
bytes  bytes  bytes  secs.  10^6bits/sec
  87380   16384   16384  40.06    849.96
```

Setting socket to BBR or SO_MAX_PACING_RATE

```
skb->tstamp = 0
reset on netns switch
```

4Gbit/s
# BPF datapath walk-through: Next steps

PoC hack to retain skb->tstamp:

<table>
<thead>
<tr>
<th>Host / initial netns</th>
<th>Pod / own netns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recv Send Send</td>
<td>Send</td>
</tr>
<tr>
<td>Socket Socket Message Elapsed</td>
<td></td>
</tr>
<tr>
<td>Size Size Size Time Throughput</td>
<td></td>
</tr>
<tr>
<td>bytes bytes bytes secs. 10^6bits/sec</td>
<td></td>
</tr>
<tr>
<td>87380 16384 16384 40.01 3976.04</td>
<td></td>
</tr>
</tbody>
</table>

Setting socket to BBR or SO_MAX_PACING_RATE

<table>
<thead>
<tr>
<th>4Gbit/s</th>
</tr>
</thead>
</table>

skb->tstamp = 0
reset on netns switch
Rationale on today’s timestamp reset

Kernel uses different clock bases for skb->tstamp:

- Ingress is CLOCK_TAI, egress is CLOCK_MONOTONIC (as is fq)
- Forwarding from RX to TX would cause drop in fq due to overreaching fq’s drop horizon (given clock’s offsets)
- No means to figure out clock base from skb->tstamp, hence reset
Rationale on today’s timestamp reset

Can skb->tstamp be normalized to a single base?

➔ Initially TCP EDT was based on CLOCK_TAI as well
➔ **Nodes were seen** where improper RTC setup caused clock discontinuities of +50yrs during boot
➔ Confused fq which lead to drops, thus broke TCP
  ○ Hence CLOCK_MONOTONIC & reset on direction switch
Approach to fixing the timestamp reset

Adding new skb->tstamp_base bit (defines: 0 ➔ TAI, 1 ➔ MONO)

➔ skb_set_tstamp_{mono,tai}(skb, ktime) helper used by RX and TX
➔ fq_enqueue() detects TAI clock and resets skb->tstamp
➔ All skb->tstamp = 0 due to forwarding are then removed
  ○ skb_mstamp_ns union could be removed as well
➔ net_timestamp_check() must be deferred in RX after tc ingress
Part 3: Managed neighbor/fib extensions
Use case: Cilium’s XDP L4LB
Current state: Cilium’s XDP L4LB

XDP LB receives packet to svcIP/port, forwards to backendIP/port:

➔ BPF: Either DNAT & SNAT or DSR with IPIP/IP6IP6 encapsulation
➔ In both cases outer header has backendIP as destination
➔ bpf_fib_lookup() used to piggyback on neighbor resolution
➔ Pushed back out via XDP_TX (transparent of phys/bond device)
Current state: Cilium’s XDP L4LB

Neighbor resolution under XDP:

➔ Neighbor entry must be present in table, cannot resolve from XDP
➔ Agent currently resolves entries manually which is a pain point
➔ Pushes resolution as NUD_PERMANENT into neighbor table
Current state: Neighbor entry management
Current state: Neighbor entry management

Node1: 10.0.0.1

Node2: 10.0.0.2

api-server

Pod nginx

name: Node2
status: ...
address: 10.0.0.2
Current state: Neighbor entry management

Kubernetes cluster

Node1: 10.0.0.1
- Cilium
- api-server

Node2: 10.0.0.2
- Cilium
- Pod nginx

ARPING 10.0.0.2?

address: 10.0.0.2

name: Node2
status: ...

Current state: Neighbor entry management
Current state: Neighbor entry management

Kubernetes cluster

Node1: 10.0.0.1

Node2: 10.0.0.2

Cilium

Pod nginx

api-server

ARPING 10.0.0.2?

10.0.0.2: aa:bb:cc:dd:ee:ff

name: Node2
status: ...
address: 10.0.0.2
Current state: Neighbor entry management

Kubernetes cluster

Node1: 10.0.0.1
- api-server
- L2 neigh:
  10.0.0.2 -> aa:bb:cc:dd:ee:ff (PERM)

Node2: 10.0.0.2
- Pod nginx

Cilium

ARPING 10.0.0.2?

10.0.0.2: aa:bb:cc:dd:ee:ff
Current state: Neighbor entry management

Node1: 10.0.0.1
- api-server
- L2 neigh:
  10.0.0.2 -> aa:bb:cc:dd:ee:ff (PERM)

Node2: 10.0.0.2
- Pod nginx

Cilium

ARPING 10.0.0.2?

Periodic

10.0.0.2: aa:bb:cc:dd:ee:ff

Kubernetes cluster
Problems with current approach

➔ How often to arping? (Currently once every 5 min)
➔ Buggy, for example:
  ○ An obsolete NUD_PERMANENT entry for the api-server node is fatal after agent restart if the former’s L2 address changed
  ○ No auto-updates from active traffic processed by the local stack
➔ Duplicating logic of net/ipv4/arp.c
➔ Need an equivalent for IPv6’s ND
Managed neighbor entry: Rationale

➔ Control plane (here: Cilium agent) requirements
  ○ Netlink route lookup: backendIP in same L2 or via GW IP?
  ○ Pushes L3 (without L2) addresses into neighbor table
➔ Neighboring subsystem auto-resolves them
➔ Periodically keeps them in REACHABLE state
➔ Option to avoid GC eviction
➔ Visibility for agent restart to resync/clean obsolete L3 entries
Managed neighbor entry: Design

→ We can piggyback on NTF_USE | NTF_EXT_LEARNED neigh flag
  ○ Gets us quite close already:
    ● Triggers one-time resolution via neigh_event_send()
    ● Updates STALE state upon external/internal traffic events
    ● Ensures that neigh entries are not added to GC list
Managed neighbor entry: Design

We can piggyback on NTF_USE | NTF_EXT_LEARNED neigh flag

- What it does not do:
  - No self-managed auto-refresh to get back to REACHABLE from STALE state due to inactivity
  - Creation flags not propagated back to user space
  - Not retained upon carrier-down events (like NUD_PERMANENT)
Managed neighbor entry: Design

→ Proposal: New NUD_MANAGED state for neigh entry creation
  ○ Volatile pseudo-state (not fixed as in NUD_PERMANENT):
    ● Implies NTF_USE and adds entry to a per-neigh table list
  ○ Uses delayed system-wq to trigger neigh_event_send() for entries
  ○ Triggered on BASE_REACHABLE_TIME/2 with slack
  ○ NUD_MANAGED can be combined with NTF_EXT_LEARNED
  ○ Retained upon carrier-down & refreshed once up again
Managed neighbor entry: iproute2 example

→ Entry creation via ‘nud managed’:
  ○ ip neigh replace 192.168.1.99 dev enp5s0 extern_learn nud managed

→ Entry dump (including flag propagation fix):
  ○ 192.168.1.99 dev enp5s0 lladdr 98:9b:cb:05:2e:ae use extern_learn REACHABLE
FIB extensions 1/2: Source IP address selection for SNAT

Outside 10.0.0.100/24

Node1

Pod client 172.16.0.1

$ curl 10.0.0.100

tc: BPF masquerade

192.168.0.1/24
10.0.0.1/24
10.0.0.100/24
FIB extensions 1/2: Source IP address selection for SNAT

```
#define IPV4_MASQ \ 192.168.0.1
src_ip4 = IPV4_MASQ;
snat_v4(skb, src_ip4);
```

Node1

Pod client
172.16.0.1

Outside
10.0.0.100/24

$ curl 10.0.0.100

$ curl 10.0.0.100

192.168.0.1/24

10.0.0.1/24

10.0.0.100/24

Node1

eth0

192.168.0.1/24

10.0.0.1/24
FIB extensions 1/2: Source IP address selection for SNAT

```
#define IPV4_MASQ 192.168.0.1
src_ip4 = IPV4_MASQ;
snat_v4(skb, src_ip4);
```

Node1

Pod client

$ curl 10.0.0.100

Outside

10.0.0.100/24

192.168.0.1/24

10.0.0.1/24

tc: BPF masquerade

FIB extensions 1/2: Source IP address selection for SNAT
FIB extensions 1/2: Source IP address selection for SNAT

#define IPV4_MASQ 192.168.0.1
src_ip4 = IPV4_MASQ;
snat_v4(skb, src_ip4);

Pod client
172.16.0.1

10.0.0.1/24

Outside
10.0.0.100/24

Reply is sent to 192.168.0.1

$ curl 10.0.0.100

Reply is sent to 192.168.0.1

$ curl 10.0.0.100

10.0.0.1/24

$ curl 10.0.0.100

10.0.0.1/24

192.168.0.1/24

Node1

eth0

tc: BPF masquerade

FIB extensions 1/2: Source IP address selection for SNAT
FIB extensions 1/2: Source IP address selection for SNAT

```c
bpf_skb_fib_lookup(&params);

snat_v4(skb, params.ipv4_src);
```

Node1

Pod client

172.16.0.1

$ curl 10.0.0.100

Outside

10.0.0.100/24

192.168.0.1/24

10.0.0/24

tc: BPF masquerade

eth0

$ curl 10.0.0.100
Proposed solution for source address selection

➔ Use bpf_{xdp,skb}_fib_lookup() for source IP address selection
  ○ Requires changes to the BPF helper implementation

➔ Introduction of a new BPF_FIB_LOOKUP_SET_SRC flag
  ○ Sets the fib_params.ipv{4,6}_src address to:
    fib_result_prefsrc() / fib6_info.fib6_src

➔ Another benefit: No need to hardcode IP addresses into the datapath
FIB extensions 2/2: Redirect in multi-homed network

// LB selects “nginx-1”
ifindex = IFINDEX_BY_SUBNET(1.1.1.1/24);
redirect(skb, ifindex);

xdp/tc: BPF L4LB
FIB extensions 2/2: Redirect in multi-homed network

// LB selects “nginx-1”
ifindex = IFINDEX_BY_SUBNET(1.1.1.1/24);
redirect(skb, ifindex);

xdp/tc: BPF L4LB

L4LB node

FIB table
1.1.1.0/24 dev eth1
2.2.2.0/24 dev eth2

Pod nginx-1
1.1.1.1/24

Pod nginx-2
2.2.2.2/24
// LB selects “nginx-1”
bpf_skbfib_fib_lookup(
&params);
redirect(skb,
params.ifindex);

def FIB table
1.1.1.0/24 dev eth1 2.2.2.0/24 dev eth2

L4LB node

FIB extensions 2/2: Redirect in multi-homed network
Proposed solution for target ifindex selection

- Use `bpf_{xdp,skb}_fib_lookup()` to determine ifindex
  - Requires fixing the BPF helper implementation, too
- Do not require ifindex when `!BPF_FIB_LOOKUP_DIRECT`
  - “params->ifindex = dev->ifindex;” already exists
  - Is current behavior a bug?
- Commit for making params->ifindex optional (to be upstreamed)
Part 4: Wildcarded BPF map lookups
Current state: Cilium XDP L4LB use case

Flexible LB traffic recorder to correlate inbound/outbound pkts:

- Introspection on path taken from fabric to L4LBs to L7 proxies/backends
- Higher-level API for out-of-band programming of L4LB agents
- Hubble then constructing PCAP for offline troubleshooting
$ hubble record "0.0.0.0/0 0 192.168.33.11/32 80 TCP"
Started recording. Press CTRL+C to stop.
2021-05-19T10:54:07Z Status: 345 packets (27445 bytes) written

$ cilium recorder list
ID  Capture Length  Wildcard Filters       ->  192.168.33.11/32:80 TCP
4241 full 0.0.0.0/0:0       ->  192.168.33.11/32:80 TCP

Users  Priority  Wildcard Masks       ->  ffffffff:ffff ff
1  56 00000000:0       ->  ffffffff:ffff ff
$
Cilium XDP L4LB: PCAP recorder overview

1) `cilium_capture_in()`
   - v4/v6 wildcard classification
   - per-cpu cache: capture info
   - push original ingress pkt

2) `cilium_capture_out()`
   - per-cpu cache: prior match?
   - push corresponding egress pkt

L4LB Node

- eth0
- xdp/tc: BPF L4LB
- XDP_TX
- Perf RB

`cilium_capture()`
- meta data header e.g. recorder ID
- pcap_pkthdr with MONO time
- full or partial payload capture

DNAT + SNAT or DSR with IPIP Encapsulation
PCAP recorder: Classifier rules

1) `cilium_capture_in()`
   - v4/v6 wildcard classification

One ‘Recorder’ consists of:
- Source CIDR, destination CIDR
- Source Port, destination Port (0: any, n: direct match)
- Currently unsupported: n-m range
- Protocol (0: any, n: direct match)
PCAP recorder: Classifier rules

1) `cilium_capture_in()`
   - v4/v6 wildcard classification

eBPF

L4LB Node

eth0

xdp/tc: BPF L4LB

One ‘Recorder’ consists of:
- Source CIDR, destination CIDR
- Source Port, destination Port (0/n)
- Currently unsupported: n-m range
- Protocol (0/n)

Agent:
- User API for programming recorders
- Tracking different masks from rules
- Regens datapath on mask set change
- v4/v6 hashtable each for rule lookup
PCAP recorder: Classifier rules

```c
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
    struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

    [...]

    __Pragma("unroll")
    for (i = 0; i < size; i++) {
        cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
        match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
        if (match)
            return match;
    }
    return NULL;
}
```
PCAP recorder: Classifier rules

```c
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
    struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

    [...]

    _Pragma("unroll")
    for (i = 0; i < size; i++) {
        cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
        match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
        if (match)
            return match;
    }

    return NULL;
}

Dynamic, ordered mask set, regenerated by agent on the fly.
```
PCAP recorder: Classifier rules

static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
    struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

    [...]

    _Pragma("unroll")
    for (i = 0; i < size; i++) {
        cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
        match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
        if (match)
            return match;
    }

    return NULL;
}
PCAP recorder: Classifier rules

```c
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
    struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

    [...]

    __Pragma("unroll")
    for (i = 0; i < size; i++) {
        cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
        match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
        if (match)
            return match;
    }

    return NULL;
}
```

Using masked key (lkey) for the hashtable lookup.
PCAP recorder: Classifier rules

```c
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
    struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };
    ...

    __Pragma("unroll")
    for (i = 0; i < size; i++) {
        cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
        match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
        if (match)
            return match;
    }

    return NULL;
}
```

Holds Recorder ID and capture length.
static __always_inline void
cilium_capture4_masked_key(const struct capture4_wcard *orig,
                        const struct capture4_wcard *mask,
                        struct capture4_wcard *out)
{
  out->daddr = orig->daddr & mask->daddr;
  out->saddr = orig->saddr & mask->saddr;
  out->dport = orig->dport & mask->dport;
  out->sport = orig->sport & mask->sport;
  out->nexthdr = orig->nexthdr & mask->nexthdr;
  out->dmask = mask->dmask;
  out->smask = mask->smask;
}

Masked key (lkey) generation for the map lookup.
Problems with current approach

➔ “Poor man’s version” of wildcard match:
  ○ Assumes small number of masks, *but* allows for large number of matches within the mask set: acceptable for our use-case
  ○ Requires expensive on-the-fly recompilation on mask set change
  ○ Linearity for probing different masks

➔ Works on old kernels, but loop unrolling risks verifier complexity issues
Native wildcard-supported BPF map

- Ideally native BPF map to avoid costly code regeneration:
  - ‘Very fast’ lookup time (Millions/sec)
  - ‘Reasonably fast’ update time (Thousands/sec)

- First use-case dates back to 2018 in context of BPF + OVS to implement Megaflows in BPF, effort stalled however
Native wildcard-supported BPF map

➔ Potential map candidate: TupleMerge (Eric Torng et al.)
   ○ Current state-of-the-art in classification algorithms
Next step on our agenda: PoC implementation for BPF runtime
Thanks! Questions, feedback, comments?

➔ Try it out:  https://cilium.link/kubeproxy-free
➔ Cilium:       https://github.com/cilium/cilium
➔ PoC code:     https://git.kernel.org/[...]/dborkman/bpf.git
                  https://github.com/brb/linux