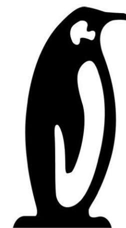
# Cilium's BPF kernel datapath revamped

Daniel Borkmann & Nikolay Aleksandrov, Isovalent



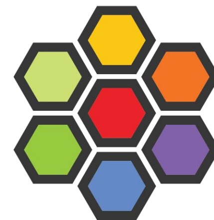
Linux Plumbers Conference
Dublin, Ireland September 12-14, 2022

# Agenda: Ongoing development items

- ➔ Part 1: Interference at prio 1 handle 1 ...
- ➔ Part 2: tc object model vs BPF links
- ➔ Part 3: Revamped design for tc BPF datapath
- ➔ Part 4: Integration of BPF links for tc

# Part 1: Interference at prio 1 handle 1 ...

# Cilium's BPF datapath overview



Host / initial netns

Pod / own netns

veth

eBPF

eBPF

(optional)

vxlan, geneve

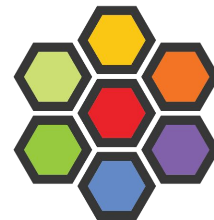
veth

veth

eBPF

eBPF

# Cilium's BPF datapath overview



Host / initial netns

Pod / own netns

tc BPF

veth

eBPF

tc BPF

eBPF

(optional)

vxlan, geneve

tc BPF

veth

veth

tc BPF

XDP BPF

eBPF

eBPF

sock_addr/sk BPF

eBPF

# Cilium’s BPF datapath overview

Host / initial netns

tc BPF

veth

eBPF

tc BPF

eBPF

(optional)

vxlan, geneve

tc BPF

XDP BPF

eBPF

eBPF

sock_addr/sk BP

**Generally:**

- Cilium assumes network ownership from a K8s CNI PoV
- Various CNI chaining options exist and work
  - Usually only around IPAM and netdev setup delegation, but not around BPF
  - Example: AWS VPC CNI plugin

**For tc BPF:**

- Cilium sets up netdevs, moves them into target netns, installs Pod addresses/routes
- Installs single cls_bpf in ‘da’ mode with: prio 1, handle 1
- Option for advanced users to customize prio
  - All bets are off in this case and up to user
  - Example: if prog at prio 1 returns TC_ACT_OK then Cilium is bypassed
  - Should be well-understood in general

# Tale from user staging env ...

[xyz] is reporting issues in a dev environment cluster.

They report "the clusters have become unhealthy and multiple pods are in CrashLoop", though it is not clear if they are talking about workload pods, or cilium pods. Current best guess is that it is just workload pods that are impacted.

They do report that the health check in the `cilium status" output in the "bad" cluster is reporting nodes that are "unreachable" in terms of Endpoints, but not in terms of the Node. This suggests routing issues related to pod CIDRs, but not the underlying fabric.

# Tale from user staging env ...

- They have other environments with pretty much the same configuration and Cilium version which is working
- In this environment, there are nightly reboots. That's probably the most significant difference they can think of
- They deleted all policies so the policy drops noted earlier are not relevant as we are still seeing connectivity failures
- We ran various network connectivity tests across different nodes
  - Realized there are good nodes and bad nodes
  - We asked them for sysdumps targeting the good and bad nodes
- They will also pause the nightly reboots to see if the nodes stay in a consistent state. They suspect that the node reboots might be exposing an issue with their configuration (outside of Cilium)

# Tale from user staging env ...

**... and debugging continued!**

- ➔ No failures in Cilium agent log, looked all reasonably normal
- ➔ User ‘sysdump’ with all configs and Cilium state looked fine
- ➔ No reports of packet drops, no issues from a policy angle
- ➔ Routes looked good, nothing suspicious from netfilter

# Tale from user staging env ...

**... until we noticed:**

bpftool-net-show.md

```
xdp:

tc:
lo(1) clsact/ingress classifier_ingress_security1_4026531992_12332 id 769
lo(1) clsact/egress classifier_egress_security1_4026531992_12332 id 770
lxc00d3490b101f(21) clsact/ingress classifier_ingress_security21_4026531992_12332 id 1575
lxc00d3490b101f(21) clsact/ingress bpf_lxc.o:[from-container] id 1555
lxc00d3490b101f(21) clsact/egress classifier_egress_security21_4026531992_12332 id 1576
lxc8e6b5222b8cc(57) clsact/ingress classifier_ingress_security57_4026531992_12332 id 1887
lxc8e6b5222b8cc(57) clsact/ingress bpf_lxc.o:[from-container] id 1877
lxcad80c832f124(159) clsact/ingress classifier_ingress_security159_4026531992_12332 id 2647
lxcad80c832f124(159) clsact/ingress bpf_lxc.o:[from-container] id 2637

flow_dissector:
```

# Tale from user staging env ...

**... until we noticed:**

lxc devices are created by Cilium, one for each Pod.

bpftool-net-show.md

```
xdp:

tc:
lo(1) clsact/ingress classifier_ingress_security1_4026531992_12332 id 769
lo(1) clsact/egress classifier_egress_security1_4026531992_12332 id 770
lxc00d3490b101f(21) clsact/ingress classifier_ingress_security21_4026531992_12332 id 1575
lxc00d3490b101f(21) clsact/ingress bpf_lxc.o:[from-container] id 1555
lxc00d3490b101f(21) clsact/egress classifier_egress_security21_4026531992_12332 id 1576
lxc8e6b5222b8cc(57) clsact/ingress classifier_ingress_security57_4026531992_12332 id 1887
lxc8e6b5222b8cc(57) clsact/ingress bpf_lxc.o:[from-container] id 1877
lxcad80c832f124(159) clsact/ingress classifier_ingress_security159_4026531992_12332 id 2647
lxcad80c832f124(159) clsact/ingress bpf_lxc.o:[from-container] id 2637

flow_dissector:
```

# Tale from user staging env ...

**... until we noticed:**

bpftool-net-show.md

```
xdp:

tc:
lo(1) clsact/ingress classifier_ingress_security1_4026531992_12332 id 769
lo(1) clsact/egress classifier_egress_security1_4026531992_12332 id 770
f(21) clsact/ingress classifier_ingress_security21_4026531992_12332 id 1575
f(21) clsact/ingress bpf_lxc.o:[from-container] id 1555
f(21) clsact/egress classifier_egress_security21_4026531992_12332 id 1576
c(57) clsact/ingress classifier_ingress_security57_4026531992_12332 id 1887
(57) clsact/ingress bpf_lxc.o:[from-container] id 1877
clsact/ingress classifier_ingress_security159_4026531992_12332 id 2647
24(159) clsact/ingress bpf_lxc.o:[from-container] id 2637

flow_dissector:
```

We do attach to clsact + {ingress, egress} hooks.

# Tale from user staging env ...

**... until we noticed:**

bpftool-net-show.md

```
xdp:

tc:
lo(1) clsact/ingress classifier_ingress_security1_4026531992_12332 id 769
lo(1) clsact/egress classifier_egress_security1_4026531992_12332 id 770
lxc0 ingress classifier_ingress_security21_4026531992_12332 id 1575
lxc0 ingress bpf_lxc.o:[from-container] id 1555
lxc0 egress classifier_egress_security21_4026531992_12332 id 1576
lxc8 ingress classifier_ingress_security57_4026531992_12332 id 1887
lxc8 ingress bpf_lxc.o:[from-container] id 1877
lxca ress classifier_ingress_security159_4026531992_12332 id 2647
lxca bpf_lxc.o:[from-container] id 2637

flow_dissector:
```

bpf_lxc programs are installed by Cilium w/ ‘from-container’ or ‘to-container’

# Tale from user staging env ...

**... until we noticed:**

bpftool-net-show.md

```
xdp:

tc:
lo(1) clsact/ingress classifier_ingress_security1_4026531992_12332 id 769
lo                  ifier_egress_security1_4026531992_12332 id 770
lx                  t/ingress classifier_ingress_security21_4026531992_12332 id 1575
lx                  t/ingress bpf_lxc.o:[from-container] id 1555
lx                    ress classifier_egress_security21_4026531992_12332 id 1576
lx                   /ingress classifier_ingress_security57_4026531992_12332 id 1887
lxc8e6b5222b8cc(57) clsact/ingress bpf_lxc.o:[from-container] id 1877
lxcad80c832f124(159) clsact/ingress classifier_ingress_security159_4026531992_12332 id 2647
lxcad80c832f124(159) clsact/ingress bpf_lxc.o:[from-container] id 2637

flow_dissector:
```

But these are not!

# Tale from user staging env ...

**... until we noticed:**

classifier_egress_security

Pull requests Issues Marketplace Explore

| | |
|---|---|
| Repositories | 0 |
| Code | 0 |
| Commits | 3 |
| Issues | 0 |
| Discussions | 0 |
| Packages | 0 |
| Marketplace | 0 |

## 3 commit results

DataDog/datadog-agent
[CWS] TC classifiers (#10553) ...
4 people committed on 24 Mar
Verified 186d25a

BFG7274/datadog-l
[CWS] TC classifiers (#10553) ...
4 people committed on 24 Mar
Verified 186d25a

# Tale from user staging env ...

**... until we noticed:**

k8s-pods-20220701-213855.txt

```
datadog    datadog-cluster-agent-696bb7957b-m8f24    0/1    CrashLoopBackOff    66
datadog    dd-agent-2l7mv                            4/4    Running             0
datadog    dd-agent-496c5                            4/4    Running             0
datadog    dd-agent-4dmb6                            4/4    Running             0
datadog    dd-agent-5btwx                            4/4    Running             0
datadog    dd-agent-5kjvv                            4/4    Running             0
datadog    dd-agent-5znxb                            4/4    Running             0
datadog    dd-agent-6q8j2                            4/4    Running             0
datadog    dd-agent-6srg2                            4/4    Running             0
datadog    dd-agent-745wq                            4/4    Running             0
datadog    dd-agent-b595q                            4/4    Running             0
datadog    dd-agent-ch887                            4/4    Running             0
datadog    dd-agent-kngfq                            4/4    Running             0
datadog    dd-agent-kv95k                            4/4    Running             0
datadog    dd-agent-kx4v6                            4/4    Running             0
datadog    dd-agent-lmnjd                            4/4    Running             0
datadog    dd-agent-mdpc4                            4/4    Running             0
datadog    dd-agent-qjd7b                            4/4    Running             0
```

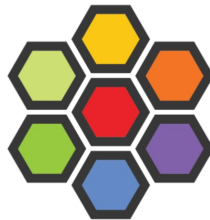
**Bingo!**

# Tale from user staging env ...

**... tl;dr:**

- ➔ 3rd party agent was replacing all **cls_bpf** instances and removing programs underneath us
- ➔ Periodically attaching to all devices with same **prio 1**, **handle 1** which we use
- ➔ Cilium agent couldn't see issue and assumed all is fine
- ➔ Removing 3rd party DaemonSet and restarting Cilium one, everything worked again

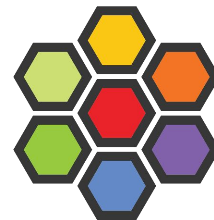
# Tale from user staging env ...



**... how can we solve the ownership problem? Enter BPF links!**

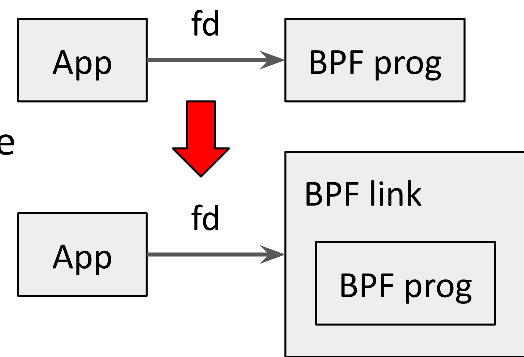
```
Subject: [PATCH bpf-next 0/3] Introduce pinnable bpf_link kernel abstraction
Date: Fri, 28 Feb 2020 14:39:45 -0800   [thread overview]
Message-ID: <20200228223948.360936-1-andriin@fb.com> (raw)

This patch series adds bpf_link abstraction, analogous to libbpf's already
existing bpf_link abstraction. This formalizes and makes more uniform existing
bpf_link-like BPF program link (attachment) types (raw tracepoint and tracing
links), which are FD-based objects that are automatically detached when last
file reference is closed. These types of BPF program links are switched to
using bpf_link framework.

FD-based bpf_link approach provides great safety guarantees, by ensuring there
is not going to be an abandoned BPF program attached, if user process suddenly
exits or forgets to clean up after itself. This is especially important in
production environment and is what all the recent new BPF link types followed.

One of the previously existing  inconveniences of FD-based approach, though,
was the scenario in which user process wants to install BPF link and exit, but
let attached BPF program run. Now, with bpf_link abstraction in place, it's
easy to support pinning links in BPF FS, which is done as part of the same
patch #1. This allows FD-based BPF program links to survive exit of a user
process and original file descriptor being closed, by creating an file entry
in BPF FS. This provides great safety by default, with simple way to opt out
for cases where it's needed.
```

# BPF links as ‘container’ object for BPF progs

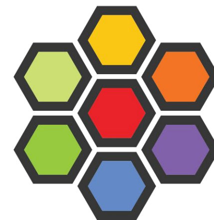


**BPF links:**

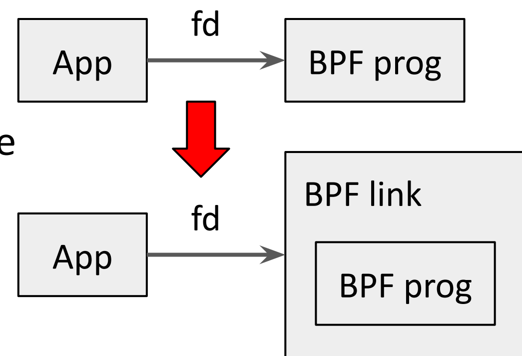
- ➔ Represents attachment of BPF prog to BPF hook point
  - Abstraction ‘containing’ BPF program
  - Holds (single) reference to keep BPF program alive
  - Hook points do not reference BPF link, only application fd or pinning does
  - Holds meta-data specific to attachment
  - Create/Update/Detach/Get{Next,FdById}
  - Application deals with link fd instead of program fd, meaning, program fd is safe to close after link is created



# BPF links as ‘container’ object for BPF progs

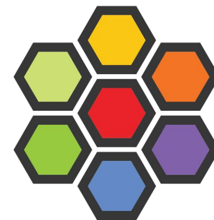


**BPF links:**




- ➔ Represents attachment of BPF prog to BPF hook point
  - Abstraction ‘containing’ BPF program
  - Holds (single) reference to keep BPF program alive
  - Hook points do not reference BPF link, only application fd or pinning does
  - Holds meta-data specific to attachment
  - Create/Update/Detach/Get{Next,FdById}
  - Application deals with link fd instead of program fd, meaning, program fd is safe to close after link is created
- ➔ Explicitly allows to prevent prog detachment on process exit when link pinned (e.g. think of tracing app, can be upgraded on the fly while prog continues to run)

# BPF links as ‘container’ object for BPF progs



**BPF links:**

- ➔ Co-exists with non-link attachments for {single,multi}-attach supported hooks
- ➔ Key properties regarding attachment
  - ◆ BPF links cannot replace other BPF links
  - ◆ BPF links cannot replace non-BPF links
  - ◆ non-BPF links cannot replace BPF links
  - ◆ (non-BPF links can replace non-BPF links)

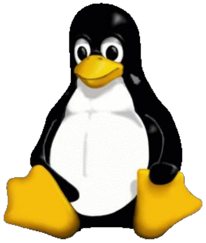
# BPF links as ‘container’ object for BPF progs

**BPF links:**

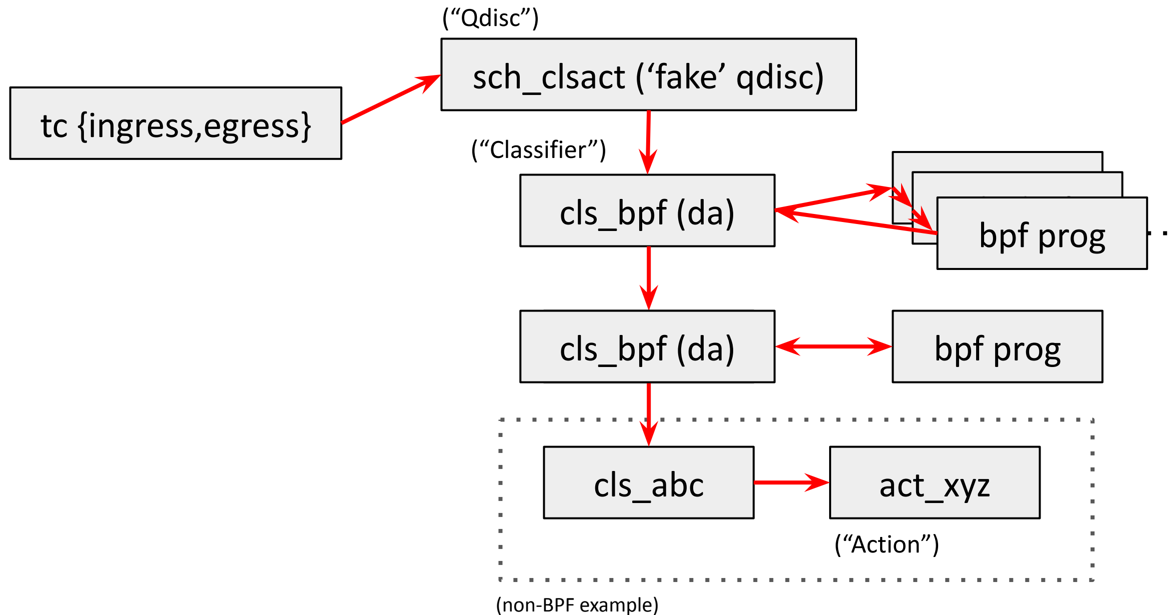
- ➔ 9 link types exist today, mostly relevant to tracing and partially networking
  - raw tracepoint, tracing, perf_event, kprobe_multi
  - XDP, netns, cgroup, struct_ops, iter
- ➔ BUT: no tc BPF link today!

# Part 2: tc object model vs BPF links

# tc recap in a nutshell



**tc objects relevant for BPF attachment:**




**da:** 'direct action', meaning BPF program returns a verdict from below instead of calling act_xyz. Crucial (!) otherwise tools like Cilium wouldn't exist today as legacy tc doesn't scale.

TC_ACT_UNSPEC:
- Continue in pipeline

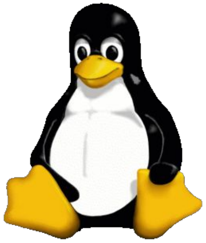
TC_ACT_OK:
- Terminate and pass to stack↑/driver↓

TC_ACT_SHOT:
- Terminate and drop

TC_ACT_REDIRECT:
- Terminate and forward to given netdev

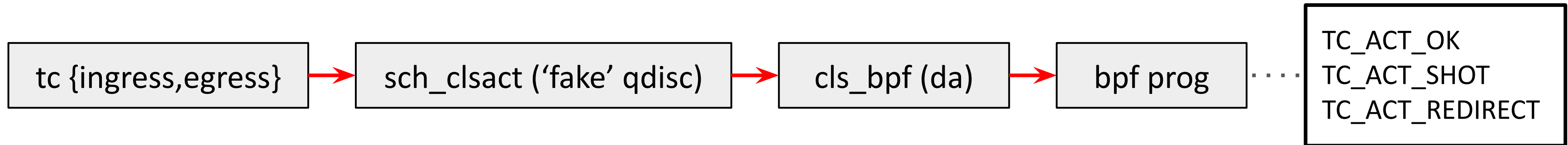
TC_ACT_*:
- Rest has not much relevance for BPF, mostly dups of above or unsupported

# tc recap in a nutshell

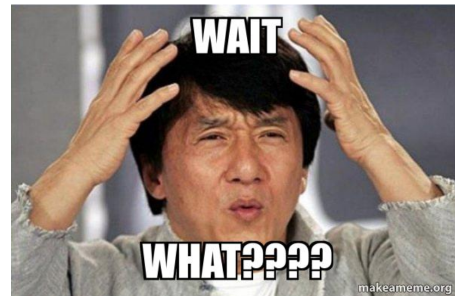


**tc objects in context of Cilium:**

- ➔ Single entry point with immediate termination
- ➔ Needed given BPF program implements complex policy/firewalling, load-balancing, local forwarding to K8s Pods, tunnel/ipsec/wireguard mesh, etc



# tc objects and BPF links?

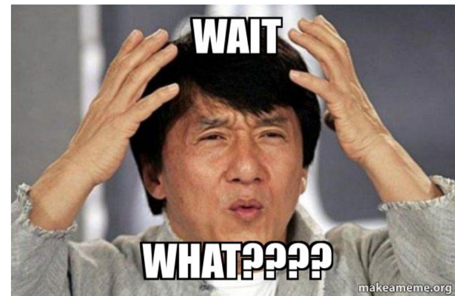



**How to marry both layers together ...**

- ➔ Tricky, tc has its own object model and configuration which does not fit well with BPF link
- ➔ Think of BPF link semantics for tracing
  - ◆ Link keeps the prog alive, so tracing can continue in pinned link when process exits, and link ownership can be taken again by new process
  - ◆ This kind of exists with cls_bpf except for the ‘ownership’ part

[ 🧵 https://lore.kernel.org/bpf/094a8c0f-d781-d2a2-d4cd-721b20d75edd@iogearbox.net/ ]

# tc objects and BPF links?



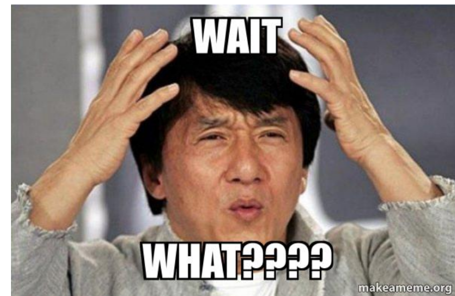

**Original thoughts and goals:**

➔ Meta: Safe auto-detachment of tc BPF programs

knowing, leading to really bad consequences. It's especially important for applications that are deployed fleet-wide and that don't "control" hosts they are deployed to. If such application crashes and no one notices and does anything about that, BPF program will keep running draining resources or even just, say, dropping packets. We at FB had outages due to such permanent BPF attachment semantics. With FD-based bpf_link we are getting a framework, which allows safe, auto-detachable behavior by default, unless application explicitly opts in w/ bpf_link__pin().

[ 🧵 https://lore.kernel.org/bpf/094a8c0f-d781-d2a2-d4cd-721b20d75edd@iogearbox.net/ ]

# tc objects and BPF links?

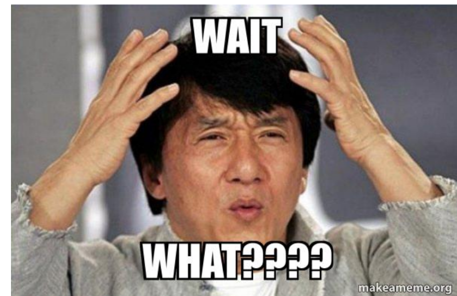



**Original thoughts and goals:**

```
bpf program is an object. That object has an owner or multiple owners.
A user process that holds a pointer to that object is a shared owner.
FD is such pointer. FD == std::shared_ptr<bpf_prog>.
Holding that pointer guarantees that <bpf_prog> will not disappear,
but it says nothing that the program will keep running.
For [ku]probe,tp,fentry,fexit there was always <bpf_link> in the kernel.
It wasn't that formal in the past until most recent Andrii's patches,
but the concept existed for long time. FD == std::shared_ptr<bpf_link>
connects a kernel object with <bpf_prog>. When that kernel objects emits
an event the <bpf_link> guarantees that <bpf_prog> will be executed.
```

[ 🧵 https://lore.kernel.org/bpf/094a8c0f-d781-d2a2-d4cd-721b20d75edd@iogearbox.net/ ]

# tc objects and BPF links?



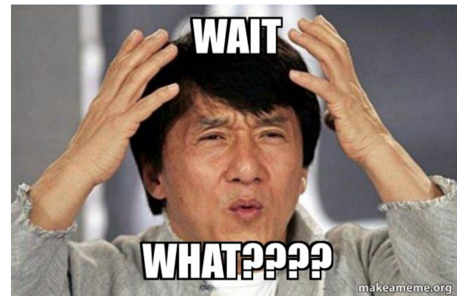

**Original thoughts and goals:**

➔ But flexibility is needed:

```
I think it depends on the environment, and yes, whether the orchestrator
of those progs controls the host [networking] as in case of Cilium. We
actually had cases where a large user in prod was accidentally removing
the Cilium k8s daemon set (and hence the user space agent as well) and only
noticed 1hrs later since everything just kept running in the data path as
expected w/o causing them an outage. So I think both attachment semantics
```

[ 🧵 https://lore.kernel.org/bpf/094a8c0f-d781-d2a2-d4cd-721b20d75edd@iogearbox.net/ ]

# tc objects and BPF links?

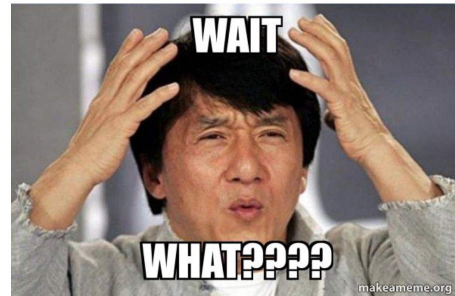



**Original thoughts and goals:**

- ➔ BPF link would lock cls_bpf program detachment, but semantics from tracing are not straight forward transferrable given we don’t have fd-based objects
  - ◆ sch_clsact can just wipe the cls_bpf, so the link would have to lock the former (or even the netdev) which gets into weird layering
  - ◆ Other cls_* objects don’t fit into the big picture but intrusive to tc internals

[ 🧵 https://lore.kernel.org/bpf/094a8c0f-d781-d2a2-d4cd-721b20d75edd@iogearbox.net/ ]

# tc objects and BPF links?




**Original thoughts and goals:**

➔ Earlier attempts intrusive/non-fitting to tc core, for example:

```
diff --git a/include/net/sch_generic.h b/include/net/sch_generic.h
index f7a6e14491fb..bacd70bfc5ed 100644
--- a/include/net/sch_generic.h
+++ b/include/net/sch_generic.h
@@ -341,7 +341,11 @@ struct tcf_proto_ops {
        int                     (*tmplt_dump)(struct sk_buff *skb,
                                              struct net *net,
                                              void *tmplt_priv);
-
+#if IS_ENABLED(CONFIG_NET_CLS_BPF)
+       int                     (*bpf_link_change)(struct net *net, struct tcf_proto *tp,
+                                                   struct bpf_prog *filter, void **arg, u32 handle,
+                                                   u32 gen_flags);
+#endif
        struct module           *owner;
        int                     flags;
 };
```

[ https://lore.kernel.org/bpf/20210604063116.234316-1-memxor@gmail.com/ ]

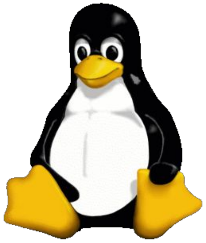
# tc objects and BPF links?

**Additional thoughts and goals:**

- ➔ Cilium: Safe ownership model of tc BPF programs
- ➔ While cooperation between components can be possible (TC_ACT_UNSPEC), it is also clear that it cannot be realized implicitly
  - ◆ Most cloud native components shipped via containers, developed by disjoint set of teams/companies/etc
  - ◆ Verdict conflicts possible, so pipeline model here to stay with explicit cooperation between BPF programs when feasible
  - ◆ However: can't make two programs work correctly together if they think they own the datapath
  - ◆ BPF link as safeguard to protect accidental stepping over each other

# Part 3: Revamped design for BPF tc datapath

# Revamped design for tc BPF datapath

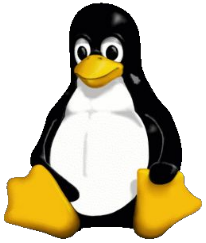


**Let’s take a step back for a moment**

- ➔ Back in 2015 we added (e)BPF support for cls_bpf because “at that time it kind of fit”
- ➔ Fast forward to 2022 its usage has skyrocketed, so ... can we do better today?

[ https://lore.kernel.org/netdev/cover.1425208501.git.daniel@iogearbox.net/ ]

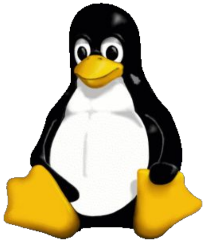
# Revamped design for tc BPF datapath



**Let's take a step back for a moment**

- ➔ Back in 2015 we added (e)BPF support for cls_bpf because “at that time it kind of fit”
- ➔ Fast forward to 2022 its usage has skyrocketed, so ... can we do better today?
- ➔ Lessons learned from a *software datapath* perspective
  - ◆ Relevant parts today mainly actual Qdiscs like sch_fq, sch_fq_codel
  - ◆ ‘Fake’ Qdisc ingress/egress hook (with few exceptions) mainly used for:
    - Slow-path fallback for hardware offloads (e.g. ovs)
    - tc BPF software datapath
  - ◆ From UX cls_bpf too hard to use ... libbpf with tc BPF API simplified it a lot

# Revampe[...]apath



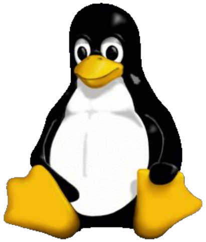
**Lets take a ste**

- ➔ Back [...]cls_bpf because "at that time it kind of fit"
- ➔ Fast f[...]ted, so ... can we do better today?
- ➔ Lesso[...]perspective
  - [...]ual Qdiscs like sch_fq, sch_fq_codel
  - [...] (with few exceptions) mainly used for:
    - [...]r hardware offloads (e.g. ovs)
    - [...]path
  - [...] ... libbpf with tc BPF API simplified it a lot

```c
DECLARE_LIBBPF_OPTS(bpf_tc_hook, tc_hook,
        .ifindex = LO_IFINDEX, .attach_point = BPF_TC_INGRESS);
DECLARE_LIBBPF_OPTS(bpf_tc_opts, tc_opts,
        .handle = 1, .priority = 1);
bool hook_created = false;
struct tc_bpf *skel;
int err;

libbpf_set_strict_mode(LIBBPF_STRICT_ALL);
libbpf_set_print(libbpf_print_fn);

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

/* The hook (i.e. qdisc) may already exists because:
 *   1. it is created by other processes or users
 *   2. or since we are attaching to the TC ingress ONLY,
 *      bpf_tc_hook_destroy does NOT really remove the qdisc,
 *      there may be an egress filter on the qdisc
 */
err = bpf_tc_hook_create(&tc_hook);
if (!err)
        hook_created = true;
if (err && err != -EEXIST) {
        fprintf(stderr, "Failed to create TC hook: %d\n", err);
        goto cleanup;
}

tc_opts.prog_fd = bpf_program__fd(skel->progs.tc_ingress);
err = bpf_tc_attach(&tc_hook, &tc_opts);
if (err) {
        fprintf(stderr, "Failed to attach TC: %d\n", err);
        goto cleanup;
}
```

(Extract from example)

# Revampe[...]apath



**Lets take a ste**

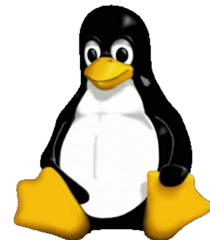
- ➔ Back [...] cls_bpf because "at that time it kind of fit"
- ➔ Fast f[...] ted, so ... can we do better today?
- ➔ Lesso[...] perspective
  - [...] ual Qdiscs like sch_fq, sch_fq_codel
  - [...] (with few exceptions) mainly used for:
    - [...] r hardware offloads (e.g. ovs)
    - [...] path
  - [...] ... libbpf with tc BPF API simplified it a lot

```
DECLARE_LIBBPF_OPTS(bpf_tc_hook, tc_hook,
        .ifindex = LO_IFINDEX, .attach_point = BPF_TC_INGRESS);
DECLARE_LIBBPF_OPTS(bpf_tc_opts, tc_opts,
        .handle = 1, .priority = 1);
bool hook_created = false;
struct tc_bpf *skel;
int err;

libbpf_set_strict_mode(LIBBPF_STRICT_ALL);
libbpf_set_print(l

skel = tc_bpf__ope
if (!skel) {
        fprintf(st
        return 1;
}

/* The hook (i.e. qdisc) ma                exists because:
 *   1. it is created by other processes or users
 *   2. or since we are attaching to the TC ingress ONLY,
 *      bpf_tc_hook_destroy does NOT really remove the qdisc,
 *      there may be an egress filter on the qdisc
 */
err = bpf_tc_hook_create(&tc_hook);
if (!err)
        hook_created = true;
if (err && err != -EEXIST) {
        fprintf(stderr, "Failed to create TC hook: %d\n", err);
        goto cleanup;
}

tc_opts.prog_fd = bpf_program__fd(skel->progs.tc_ingress);
err = bpf_tc_attach(&tc_hook, &tc_opts);
if (err) {
        fprintf(stderr, "Failed to attach TC: %d\n", err);
        goto cleanup;
}
```

BUT: Given no ownership model, the detachment doesn't remove the sch_clsact / cls_bpf.

(Extract from example)

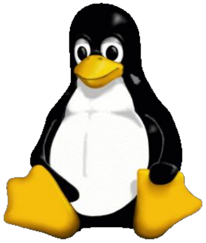
# Revamped design for tc BPF datapath



**Let’s take a step back for a moment**

- ➔ Requirements for fresh design today:
    - ◆ fd-based, so that BPF link blends in perfectly

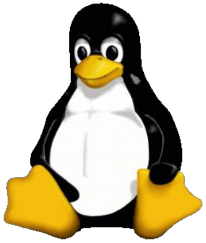
# Revamped design for tc BPF datapath



**Let’s take a step back for a moment**

- ➔ Requirements for fresh design today:
  - ◆ fd-based, so that BPF link blends in perfectly
  - ◆ Multi-attach required, but must be efficient

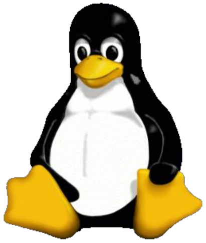
# Revamped design for tc BPF datapath



**Let’s take a step back for a moment**

➔ Requirements for fresh design today:
- fd-based, so that BPF link blends in perfectly
- Multi-attach required, but must be efficient
- Minimal overhead entry point into BPF program

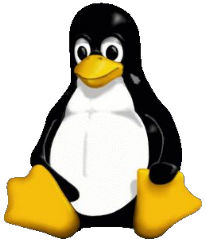
# Revamped design for tc BPF datapath



**Let’s take a step back for a moment**

- ➔ Requirements for fresh design today:
  - fd-based, so that BPF link blends in perfectly
  - Multi-attach required, but must be efficient
  - Minimal overhead entry point into BPF program
  - Easy-to-program/consume API for developers

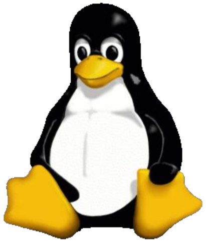
# Revamped design for tc BPF datapath



**Let’s take a step back for a moment**

- ➔ Requirements for fresh design today:
  - ◆ fd-based, so that BPF link blends in perfectly
  - ◆ Multi-attach required, but must be efficient
  - ◆ Minimal overhead entry point into BPF program
  - ◆ Easy-to-program/consume API for developers
  - ◆ Not ‘polluting’ stack with yet another hook

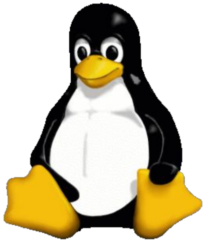
# Revamped design for tc BPF datapath



**Let's take a step back for a moment**

- ➔ Requirements for fresh design today:
  - fd-based, so that BPF link blends in perfectly
  - Multi-attach required, but must be efficient
  - Minimal overhead entry point into BPF program
  - Easy-to-program/consume API for developers
  - Not 'polluting' stack with yet another hook
  - Must integrate with old-style cls_bpf for migration path

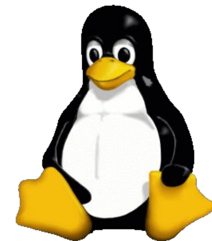
# Revamped design for tc BPF datapath



**Let’s take a step back for a moment**

- ➔ Requirements for fresh design today:
  - fd-based, so that BPF link blends in perfectly
  - Multi-attach required, but must be efficient
  - Minimal overhead entry point into BPF program
  - Easy-to-program/consume API for developers
  - Not ‘polluting’ stack with yet another hook
  - Must integrate with old-style cls_bpf for migration path
  - Must support tc BPF programs 1:1 (or very close to it)

# Revamped design for tc BPF datapath

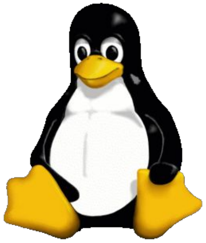


**Overview**

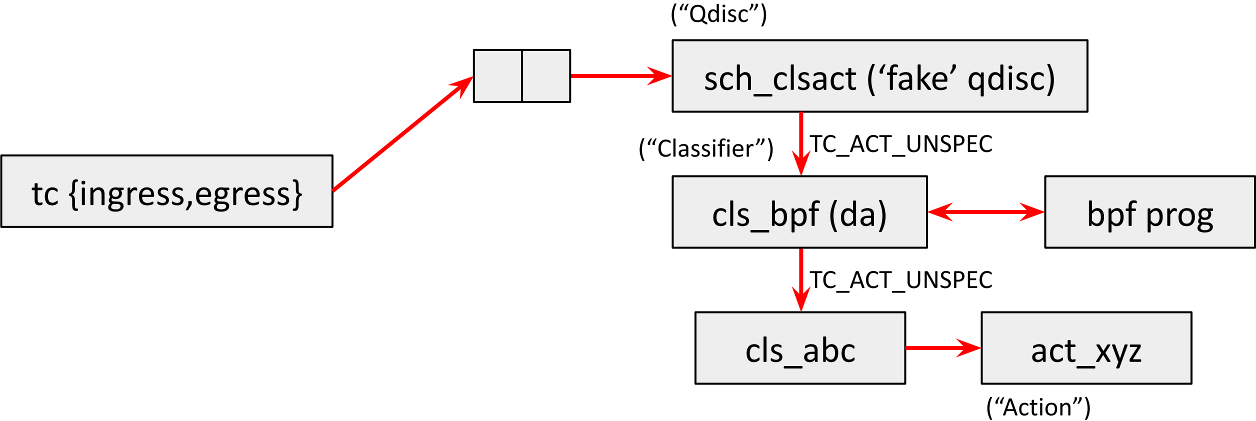


Old-style, no change in behavior

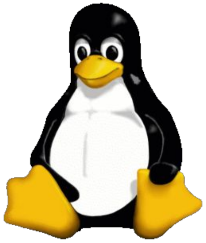
# Revamped design for tc BPF datapath



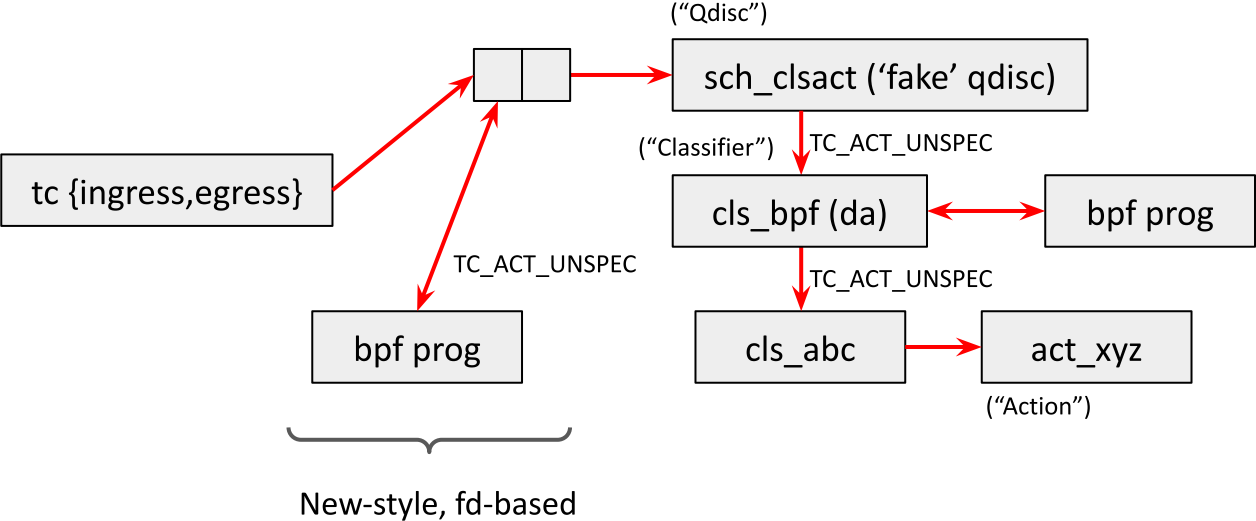
**Overview**



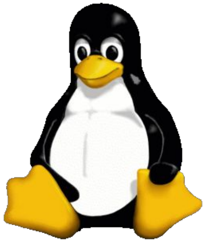
# Revamped design for tc BPF datapath



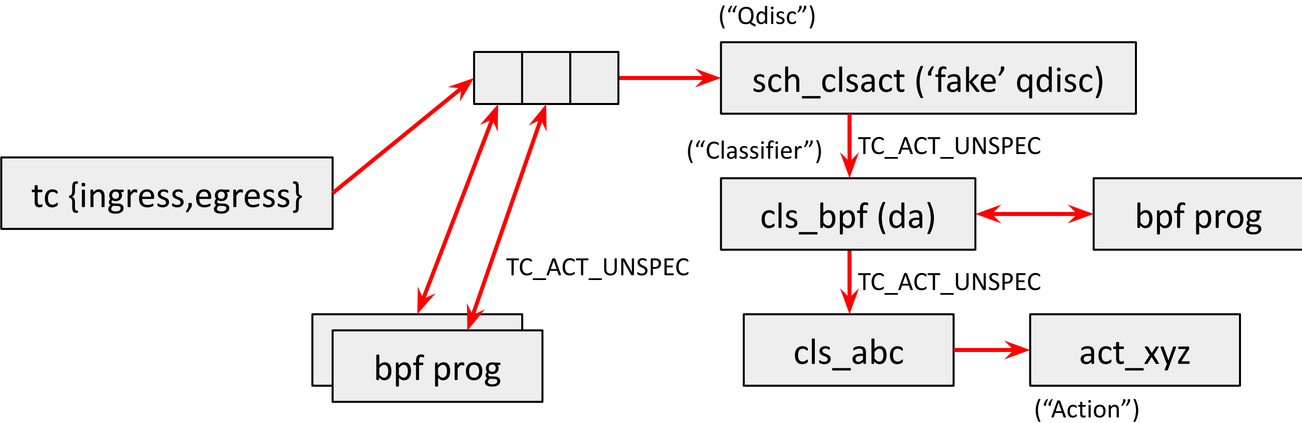
**Overview**



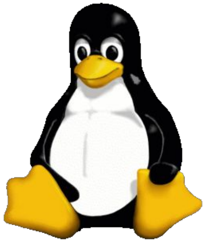
# Revamped design for tc BPF datapath



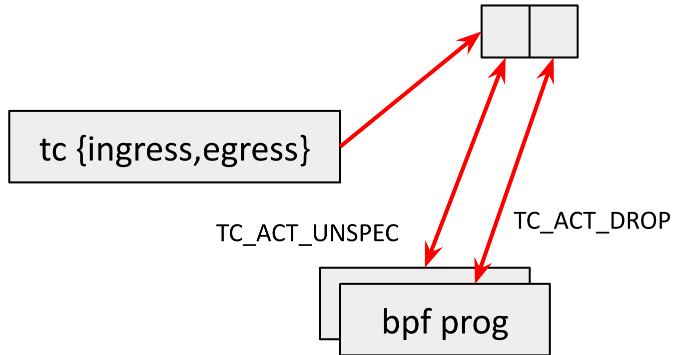
**Overview**



# Revamped design for tc BPF datapath

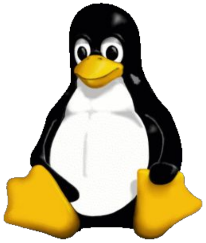


## Overview

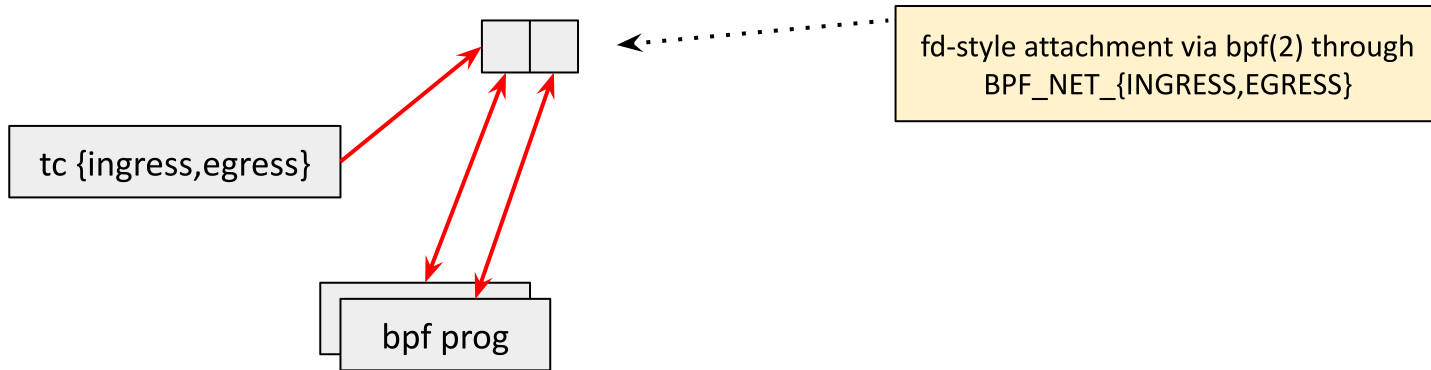



(But, if no old-style used, then not invoked at all.)

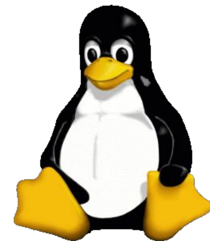
# Revamped design for tc BPF datapath



**Overview**



# Revamped design for tc BPF datapath

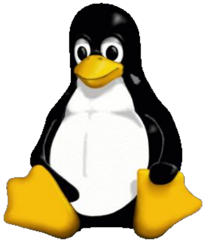


__netif_receive_skb_core:

```
skip_taps:
#ifdef CONFIG_NET_INGRESS
	if (static_branch_unlikely(&ingress_needed_key)) {
		bool another = false;

		nf_skip_egress(skb, true);
		skb = sch_handle_ingress(skb, &pt_prev, &ret, orig_dev,
					 &another);
		if (another)
			goto another_round;
		if (!skb)
			goto out;

		nf_skip_egress(skb, false);
		if (nf_ingress(skb, &pt_prev, &ret, orig_dev) < 0)
			goto out;
	}
#endif
```

Entry as-is

# Revamped design for tc BPF datapath

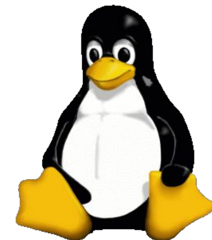


sch_handle_ingress:

```
struct sch_entry *entry = rcu_dereference_bh(skb->dev->sch_ingress);

if (!entry)
        return skb;
if (*pt_prev) {
        *ret = deliver_skb(skb, *pt_prev, orig_dev);
        *pt_prev = NULL;
}

qdisc_skb_cb(skb)->pkt_len = skb->len;
sch_set_ingress(skb, true);

switch (sch_run_progs(entry, skb, true)) {
case TC_ACT_UNSPEC:
case TC_ACT_OK:
        break;
default:
case TC_ACT_SHOT:
        kfree_skb_reason(skb, SKB_DROP_REASON_TC_INGRESS);
        return NULL;
case TC_ACT_REDIRECT:
        __skb_push(skb, skb->mac_len);
        if (skb_do_redirect(skb) == -EAGAIN) {
                __skb_pull(skb, skb->mac_len);
                *another = true;
                break;
        }
        return NULL;
case TC_ACT_CONSUMED:
        consume_skb(skb);
        return NULL;
}
return skb;
```

Only action codes which are actually used in BPF context

# Revamped design for tc BPF datapath

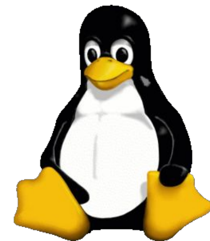


sch_run_progs:

```
const struct bpf_prog_array_item *item;
const struct bpf_prog *prog;
int ret = TC_ACT_UNSPEC;

if (needs_mac)
        __skb_push(skb, skb->mac_len);
item = &entry->items[0];
while ((prog = READ_ONCE(item->prog))) {
        bpf_compute_data_pointers(skb);
        ret = bpf_prog_run(prog, skb);
        if (ret != TC_ACT_UNSPEC)
                break;
        item++;
}
if (needs_mac)
        __skb_pull(skb, skb->mac_len);
return ret;
```

Main loop over cache friendly program array

# Revamped design for tc BPF datapath

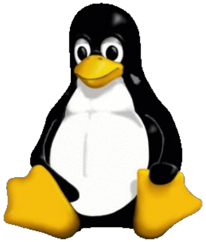


sch_cls_ingress (old-style):

```
tc_skb_cb(skb)->mru = 0;
tc_skb_cb(skb)->post_ct = false;

miniq = dev_sch_entry_pair(skb->dev->sch_ingress)->miniq;
if (!miniq)
        return TC_ACT_UNSPEC;
mini_qdisc_bstats_cpu_update(miniq, skb);
__skb_pull(skb, skb->mac_len);
ret = tcf_classify(skb, miniq->block, miniq->filter_list, &res, false);
__skb_push(skb, skb->mac_len);
/* Only tcf related quirks below. */
switch (ret) {
case TC_ACT_SHOT:
        mini_qdisc_qstats_cpu_drop(miniq);
        break;
case TC_ACT_OK:
case TC_ACT_RECLASSIFY:
        skb->tc_index = TC_H_MIN(res.classid);
        ret = TC_ACT_OK;
        break;
case TC_ACT_STOLEN:
case TC_ACT_QUEUED:
case TC_ACT_TRAP:
        ret = TC_ACT_CONSUMED;
        break;
case TC_ACT_CONSUMED:
        /* Bump refcount given skb is now in use elsewhere. */
        skb_get(skb);
        break;
}
return ret;
```

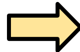
Old-style tc remaps into more generic action codes

# Revamped design for tc BPF datapath



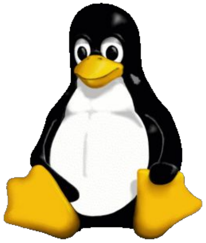
## Comparison for BPF point to entry

```
bpf_prog_a04f5eef06a7f555()
// list: for each cls_bpf_prog: bpf_prog_run()
cls_bpf_classify()
// list: for each tp: tp->classify()
// (return path: conditional tc_skb_ext_alloc)
tcf_classify()
sch_handle_ingress()
__netif_receive_skb_list_core()
netif_receive_skb_list_internal()
```

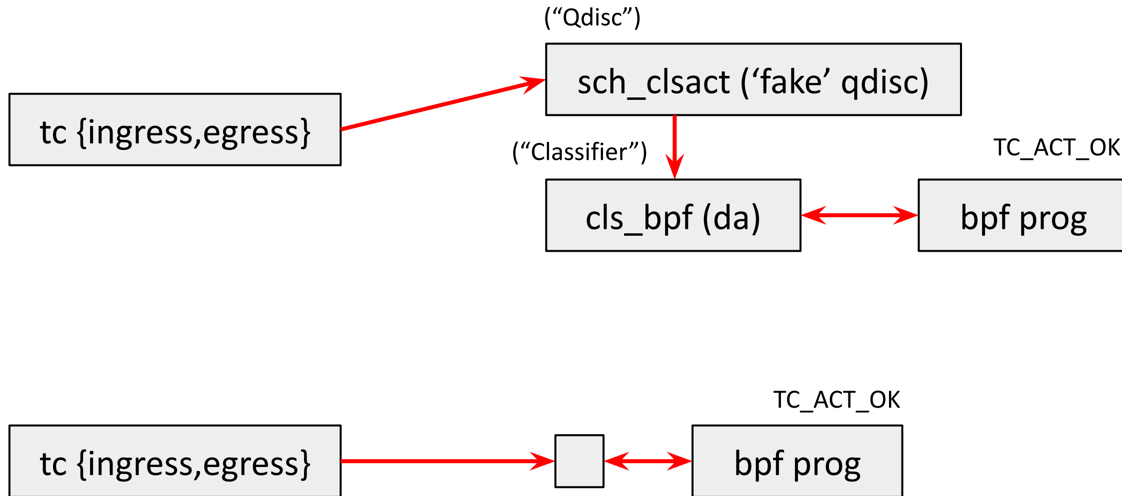


```
bpf_prog_a04f5eef06a7f555()
// array: for each item: bpf_prog_run()
sch_run_progs()
sch_handle_ingress()
__netif_receive_skb_list_core()
netif_receive_skb_list_internal()
```

# Revamped design for tc BPF datapath



## Comparison for BPF point to entry



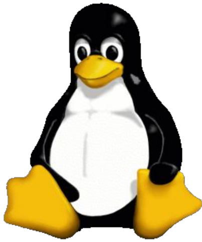

(ubench when cache hot)

**before:** 59 cycles

**after:** 33 cycles

(AMD Ryzen 9 3950X, defaults wrt Spectre mitigations, disabled IRQ, disabled CPU freq scaling, disabled idle states, pinned to 1 core)

# Revamped design for tc BPF datapath

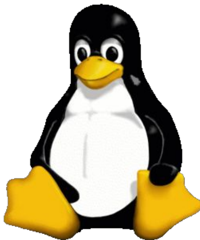


## User API walkthrough

User application for attaching BPF_NET_{INGRESS,EGRESS}:

```
DECLARE_LIBBPF_OPTS(bpf_prog_attach_opts, opt);
int prio = 0; // == auto or #num
int ifindex = 1;

[...]

opt.flags = BPF_F_REPLACE;
opt.attach_priority = prio;
err = bpf_prog_attach_opts(prog_fd, ifindex, BPF_NET_INGRESS, &opt);
```

# Revamped design for tc BPF datapath

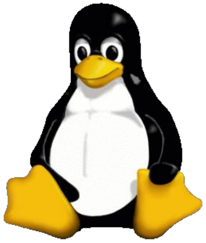


## User API walkthrough

User application for query:

```
__u32 prog_cnt, attach_flags = 0;

[...]

prog_cnt = 0;
err = bpf_prog_query(ifindex, BPF_NET_INGRESS, 0, &attach_flags,
                     NULL, &prog_cnt);
[...]

ASSERT_EQ(prog_cnt, 1, "prog_cnt");

memset(progs, 0, sizeof(progs));
prog_cnt = ARRAY_SIZE(progs);
err = bpf_prog_query(ifindex, BPF_NET_INGRESS, 0, &attach_flags,
                     progs, &prog_cnt);
[...]

ASSERT_EQ(prog_cnt, 1, "prog_cnt");
ASSERT_EQ(progs[0].prog_id, id1, "prog[0]_id");
ASSERT_EQ(progs[0].link_id, 0, "prog[0]_link");
ASSERT_EQ(progs[0].prio, 1, "prog[0]_prio");
ASSERT_EQ(progs[1].prog_id, 0, "prog[1]_id");
```

# Revamped design for tc BPF datapath

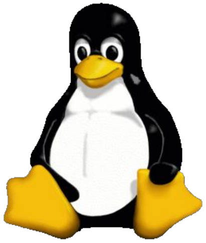


## User API walkthrough

User application for detaching BPF_NET_{INGRESS,EGRESS}:

```
DECLARE_LIBBPF_OPTS(bpf_prog_detach_opts, opt);
[...]

opt.attach_priority = 1;
err = bpf_prog_detach_opts(0, ifindex, BPF_NET_EGRESS, &opt);
[...]
```

# Revamped design for tc BPF datapath

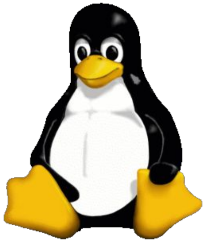


**Short summary**

- ➔ fd-based tc BPF prog installation via bpf(2) where BPF links are then implemented upon
  - For ease of initial migration supports also regular BPF attach API
- ➔ Multi-attach for ingress/egress entry array currently up to 32 slots each
  - Same prio concept as rest of tc, including prio auto-allocation via IDR
  - Same TC_ACT_* semantics to process/terminate pipeline
- ➔ Compat with old-style tc framework and same hook reuse
  - For old-style there's in-kernel API for sch_clsact/ingress to attach

# Part 4: Integration of BPF links for tc

# Integration of BPF links for tc BPF



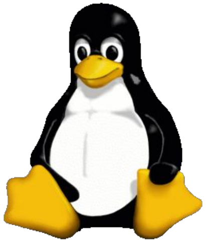
**Kernel-side**

- ➔ Supported for fd-based tc BPF datapath
- ➔ Contains device, priority, location attributes
- ➔ Implements attach/(atomic) update/detach
- ➔ Implements link attach semantics mentioned earlier to solve ownership problem described in Part 1

```
struct bpf_tc_link {
        struct bpf_link link;
        struct net_device *dev;
        u32 priority;
        u32 location;
};
```

```
static const struct bpf_link_ops bpf_tc_link_lops = {
        .release        = sch_link_release,
        .detach         = sch_link_detach,
        .dealloc        = sch_link_dealloc,
        .update_prog    = sch_link_update,
        .show_fdinfo    = sch_link_fdinfo,
        .fill_link_info = sch_link_fill_info,
};
```

# Integration of BPF links for tc BPF



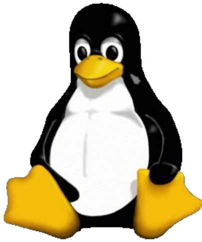
## User API walkthrough

Bare dummy programs:

```
SEC("tc/ingress")
int tc_handler_in(struct __sk_buff *skb)
{
        [...]
        return TC_ACT_UNSPEC;
}

SEC("tc/egress")
int tc_handler_eg(struct __sk_buff *skb)
{
        [...]
        return TC_ACT_UNSPEC;
}
```

Two new tc-style sections

# Integration of BPF links for tc BPF



## User API walkthrough

User application for attaching both:

```
        int ifindex = 1;
        int prio = 0; // == auto or #num
        struct test_tc_link *skel;
        struct bpf_link *link;

        skel = test_tc_link__open_and_load();
        if (!ASSERT_OK_PTR(skel, "skel_load"))
                goto cleanup;

        link = bpf_program__attach_tc(skel->progs.tc_handler_eg, ifindex, prio);
        if (!ASSERT_OK_PTR(link, "link_attach"))
                goto cleanup;
        skel->links.tc_handler_eg = link;

        link = bpf_program__attach_tc(skel->progs.tc_handler_in, ifindex, prio);
        if (!ASSERT_OK_PTR(link, "link_attach"))
                goto cleanup;
        skel->links.tc_handler_in = link;

        [...]
cleanup:
        test_tc_link__destroy(skel);
```

# Thanks! Questions, feedback, comments?

**PoC:** https://github.com/cilium/linux/commits/pr/bpf-tc-links