Cilium Datapath Plugins

Jordan Rife (jrife@google.com, GH: jrife)

Agenda

Summary: *“Hive for the dataplane” – a plugin framework for the Cilium dataplane.*

- [5m] Background
- [5m] Datapath Plugins
- [10m] Proof Of Concept
- [10m] Q&A

Background

Motivation

- Hive improves modularity in the control plane allowing for
  - Easier testing
  - Explicit dependency tracking
  - Easier extension
- In contrast, the dataplane (`bpf/`) is harder to understand and extend
  - For Cilium itself, the monolithic code structure makes the dataplane harder to understand, test, etc.
  - Third party extension is difficult without embedding code, and upstreaming isn't always possible for non-generic changes.

Custom Calls

- Custom calls already provide a mechanism for datapath extension
- With `--enable-custom-calls` users can inject custom BPF programs into a `PROG_ARRAY`.
- Limited in scope, only called at the end of Cilium's BPF programs.
- Not great for extension in the middle of the Cilium datapath (e.g. before service resolution, before policy enforcement, etc.)

Custom Calls ++

- Can we extend the number of hook points throughout the datapath to enhance utility?

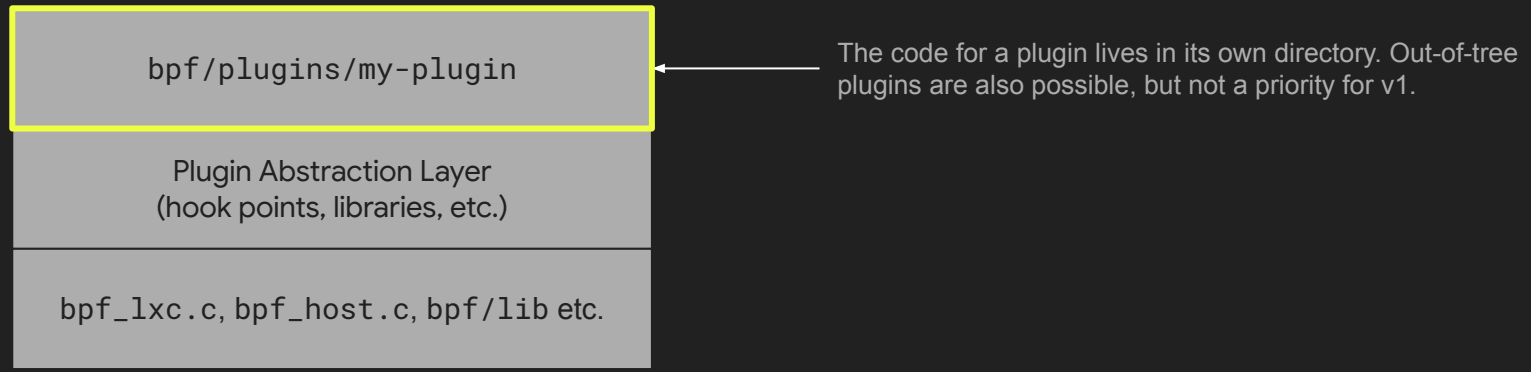
Goals of Datapath Plugins

Enable greater datapath modularity and extensibility

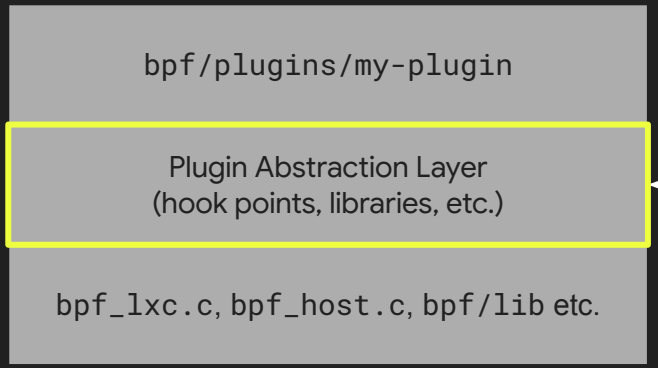
- Provide a set of well-defined hook points throughout the Cilium datapath.
- Spec out the datapath plugin structure. Where and how are hooks are implemented?
- Provide the plumbing that loads and injects plugin hooks into the Cilium datapath.
- Provide configuration for enabling or disabling datapath plugins.

Datapath Plugins

Overview

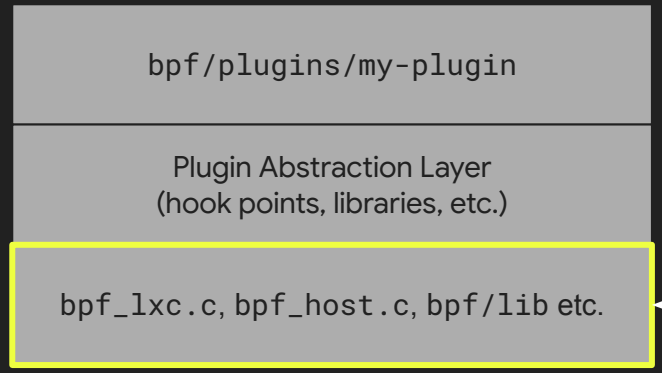

bpf/plugins/my-plugin
Plugin Abstraction Layer
(hook points, libraries, etc.)
bpf_lxc.c, bpf_host.c, bpf/lib etc.
The code for a plugin lives in its own directory. Out-of-tree plugins are also possible, but not a priority for v1.

Overview



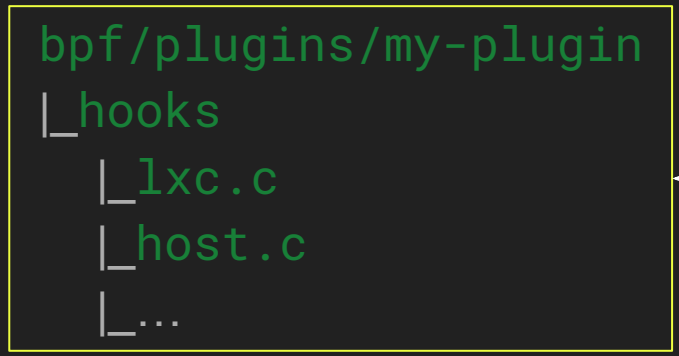
Plugins implement hooks for well-defined hook points. Plugin libraries (e.g. `plugin.h`) may provide a set of helpers that remain stable between Cilium versions.

Overview



Implementation details may change down here but the plugin contract remains intact. In an ideal scenario, plugins keep working between Cilium versions.

Anatomy Of A Plugin



A plugin’s code lives in a self-contained directory.

Anatomy Of A Plugin

```
bpf/plugins/my-plugin
|_hooks
  |_lxc.c
  |_host.c
  |_...
```

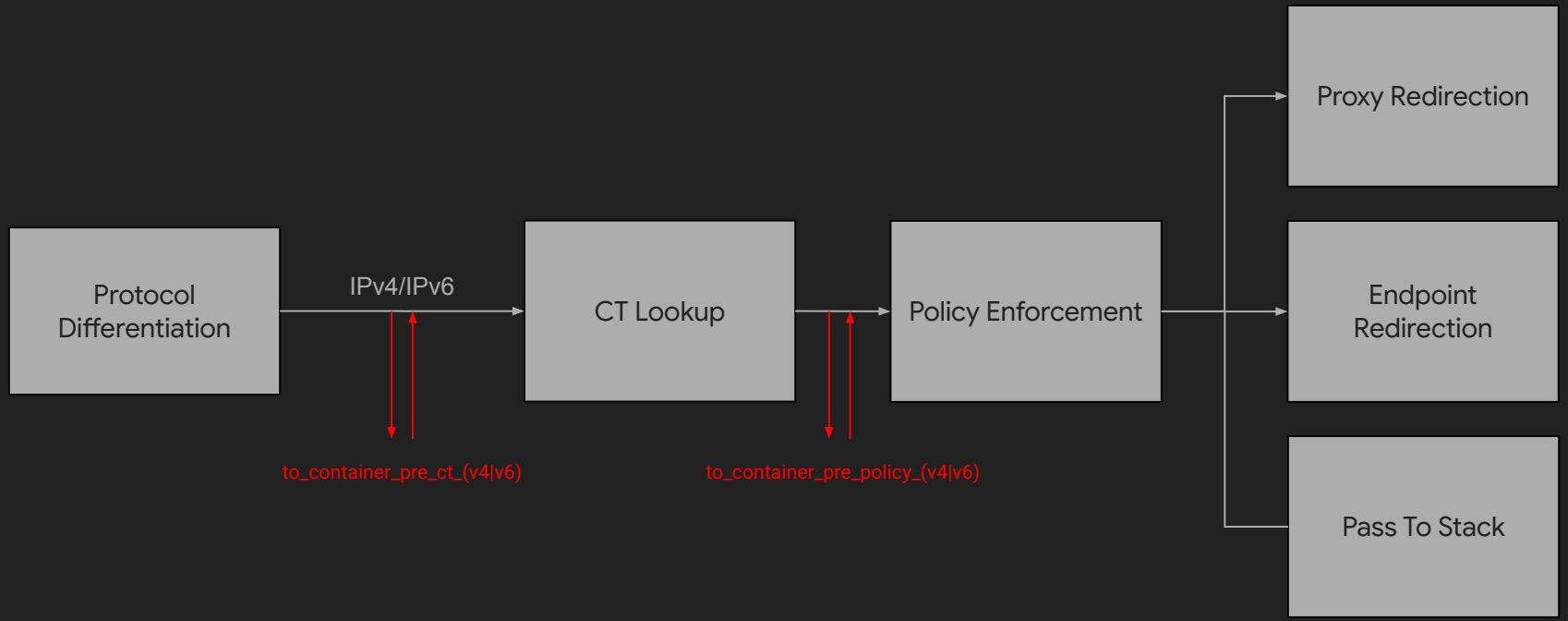

`hooks` contains various C files with well-known names. Hooks for a pod live in `lxc.c`, hooks for host and netdev interfaces live in `host.c`, etc. Corresponds to `bpf_lxc.c`, `bpf_host.c`, etc.

Each one is compiled and loaded similarly to its `bpf_*.c` counterpart. For example `lxc.c` is compiled into a template and generated per endpoint.

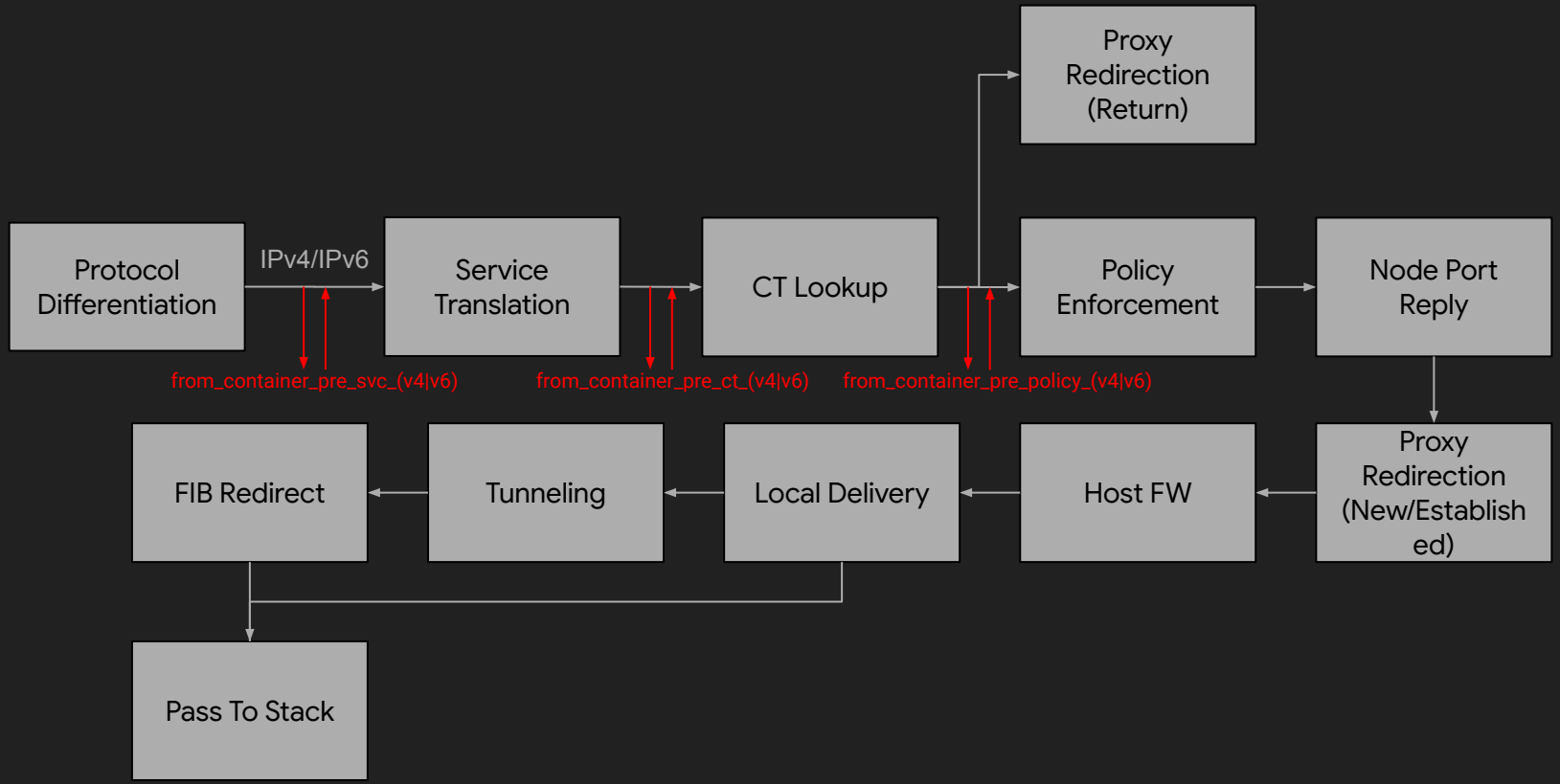
Choosing Hook Points

- Well-defined hook points “lock in” certain guarantees
  - Imposes ordering constraints going forward if the “plugin contract” is to be maintained between versions.
  - Reduces flexibility in how the core datapath code can be changed.
- However, there exist some fundamental phases / constraints already
  - Some examples:
    - Conntrack lookup needs to happen before policy enforcement (container egress+ingress)
    - Service backend resolution needs to happen before policy enforcement
  - Hook points at these boundaries are useful.

Hook Point Example: Container Ingress



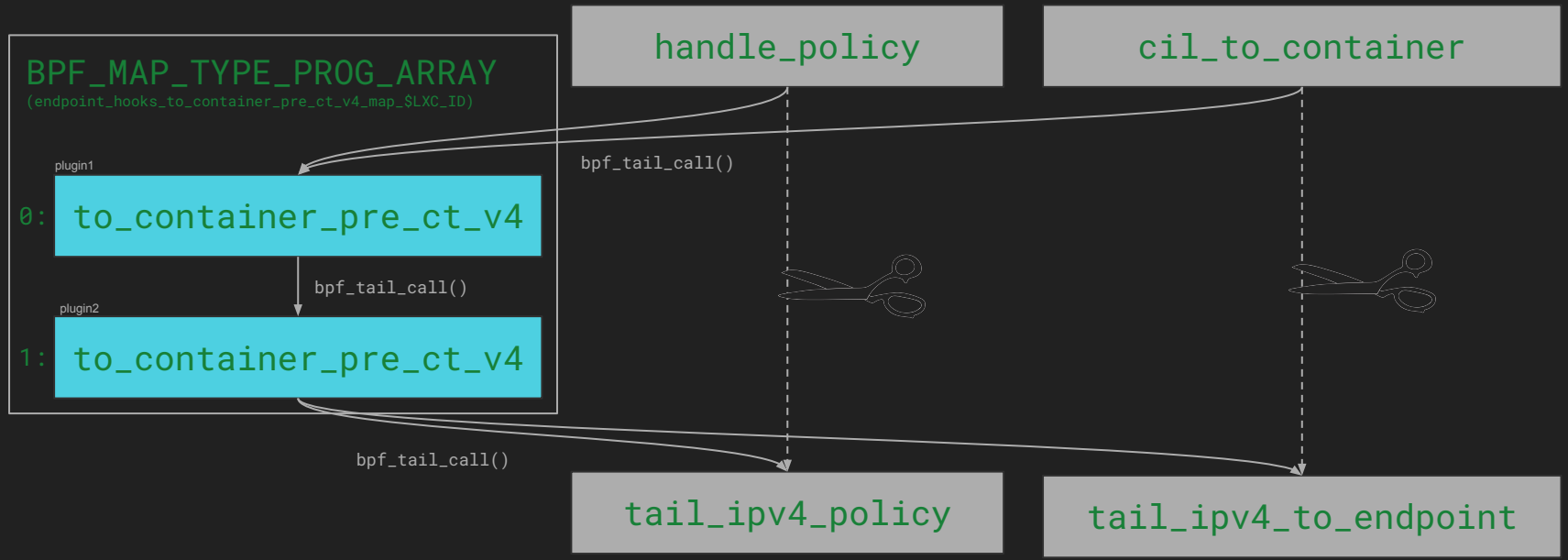
Hook Point Example: Container Egress



Implementing Hook Points

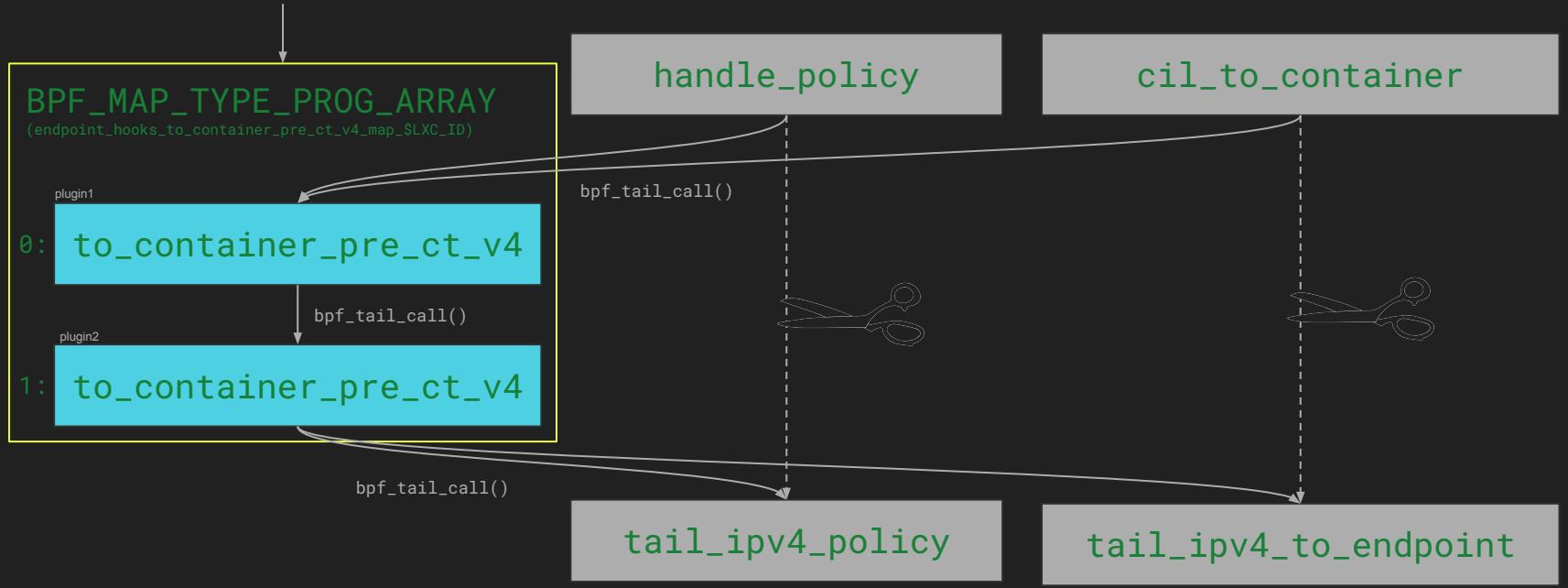
- Alternative 1: Multi-attachment to TC/TCX hook points
  - **Pros:** Minimally invasive. Requires *no* changes to the Cilium BPF code structure.
  - **Cons:** Inflexible. Cannot insert hooks into the middle of the Cilium datapath.
- Alternative 2: BPF-to-BPF + Tail Calls
  - **Pros:** Minimally invasive and flexible. Requires *minimal* changes to the Cilium BPF code structure.
  - **Cons:** Not practical. Imposes a stack size limit (256 bytes) which makes verifier reject most programs in `bpf_lxc.c`, `bpf_host.c` when combined with this technique.
- **Alternative 3: Tail Call Interposition**
  - Where there is an existing tail call boundary (e.g. `A` tail calls to `B`), we can insert our own chain of tail calls: `A` → `P1` → `P2` → … → `PN` → `B`.
  - **Pros:** More flexible than alternative one and imposes no stack size limit
  - **Cons:** Most invasive and less flexible than alternative two. Need to break apart programs to introduce hook points. Code structure is oriented around the need for hook points.

Hook Points: to_container_pre_ct_v4

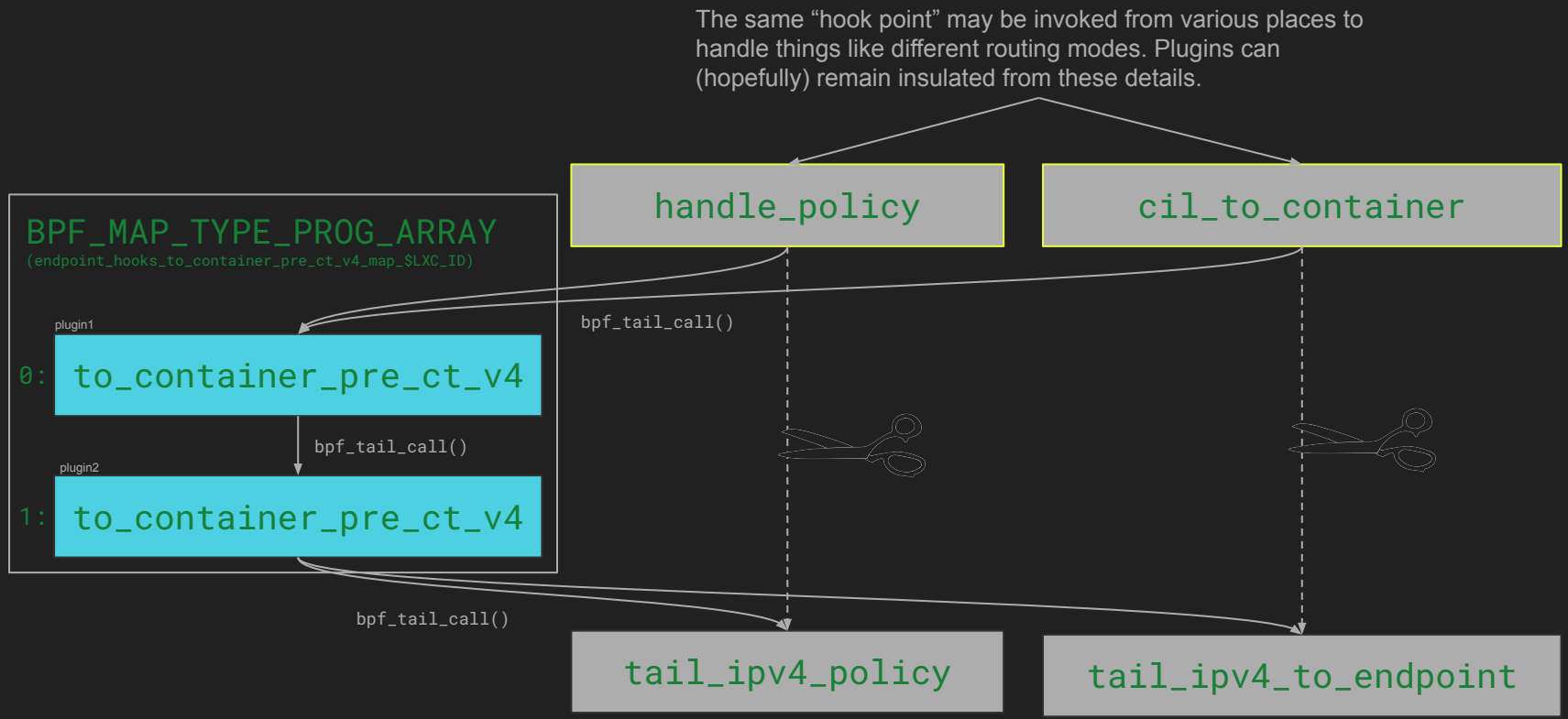


Hook Points: to_container_pre_ct_v4

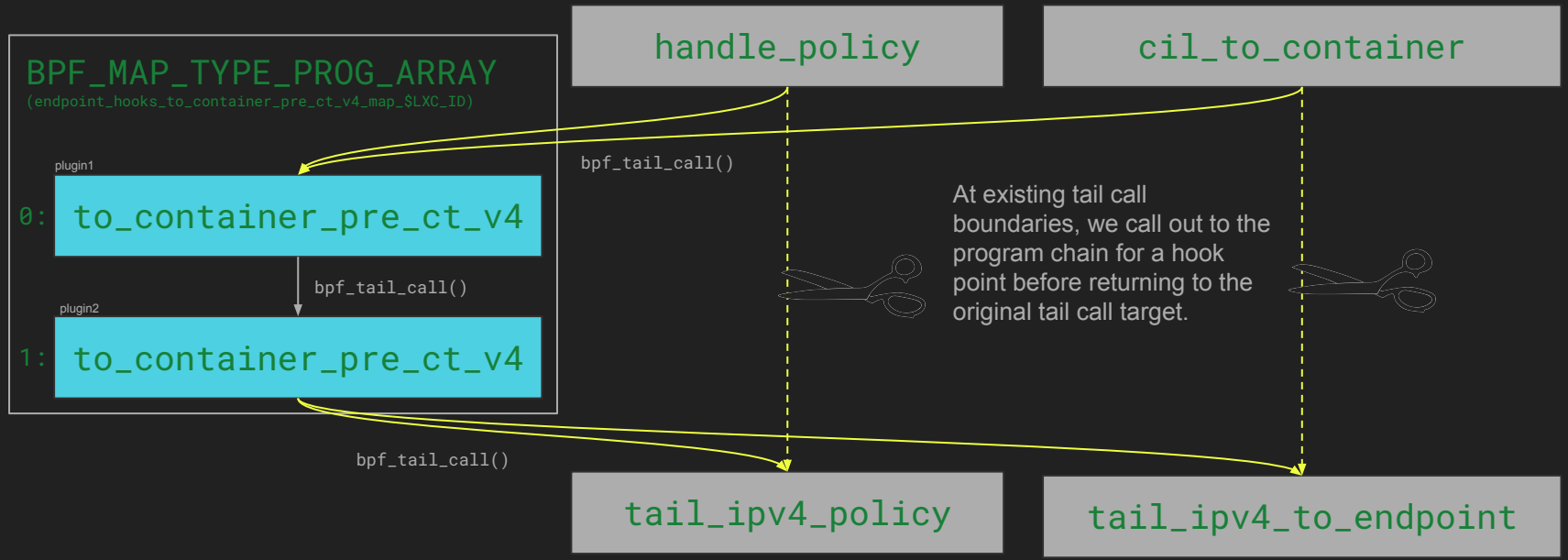
There exists a PROG_ARRAY per endpoint and hook point. A chain of tail calls connects the same handler from various plugins.



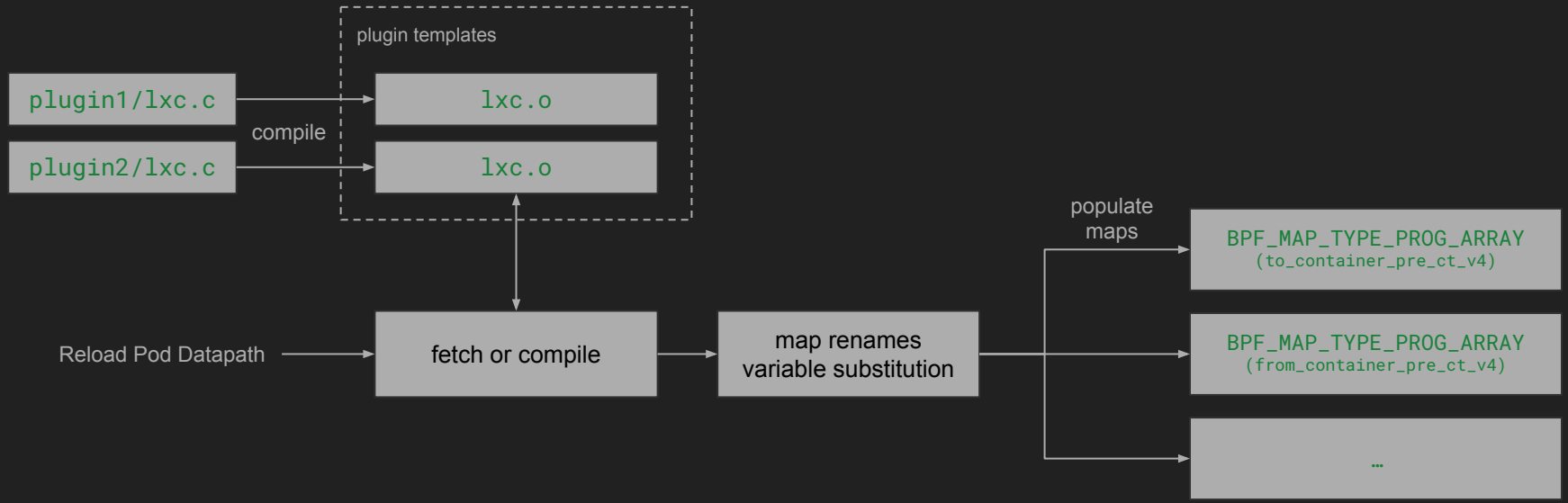
Hook Points: to_container_pre_ct_v4



Hook Points: to_container_pre_ct_v4

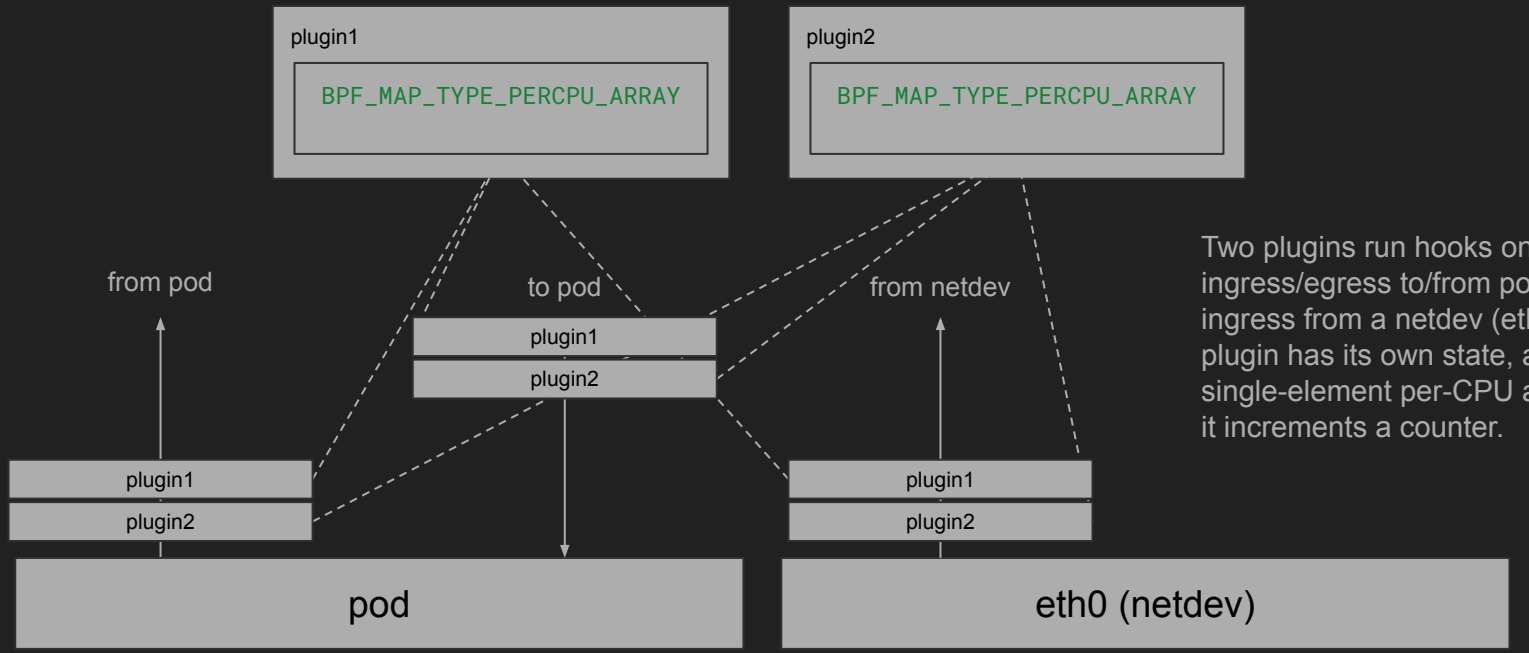


Building And Loading Plugins



Proof Of Concept

Scenario



Two plugins run hooks on ingress/egress to/from pods and on ingress from a netdev (eth0). Each plugin has its own state, a single-element per-CPU array where it increments a counter.

Demo

Q&A