Deep dive & Getting started with PSA implementation for eBPF

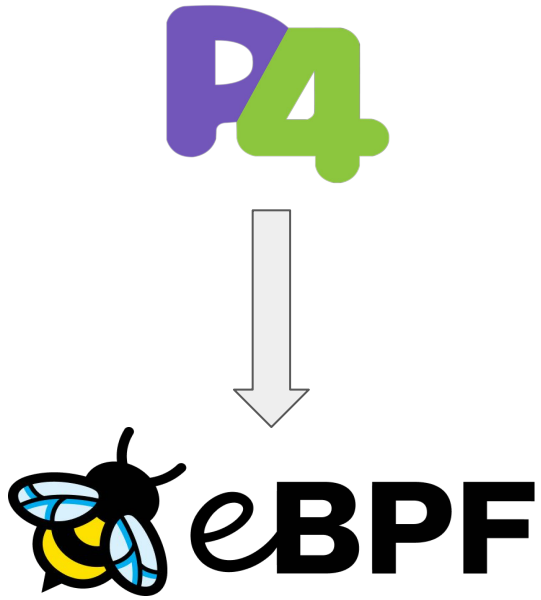
Tomasz Osiński (Intel, ex-ONF),
Mateusz Kossakowski, Jan Palimąka (Orange)

Agenda

- **Introduction to PSA-eBPF**
- **Background:**
  - eBPF
  - Short overview of Portable Switch Architecture (PSA)
- **Deep dive into PSA design & implementation for eBPF:**
  - How PSA is mapped to eBPF subsystem?
  - How PSA externs are implemented?
- **Getting started with PSA-eBPF:**
  - psabpf C library and CLI tool
  - Hands-on session
  - Integration with Mininet
  - Troubleshooting
- **Summary:**
  - Programming guidelines
  - Future work
  - How to get started? How to get involved?

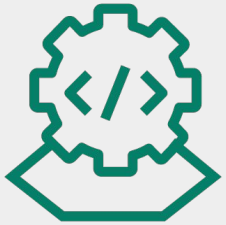
PSA implementation for eBPF (aka PSA-eBPF)

- **eBPF** is a popular technology for flexible and high-speed packet processing on the Linux OS
- P4 compiler already implements eBPF backend
  - P4 developers can write P4 programs and compile them to eBPF instructions that are further injected into the Linux kernel to process packets
  - legacy eBPF backend only supports packet filtering with limited capabilities (only a few P4 externs)
- **PSA-eBPF** is an extension to P4-eBPF, implementing a fully-featured Portable Switch Architecture (PSA) for eBPF



PSA-eBPF components

PSA-eBPF compiler

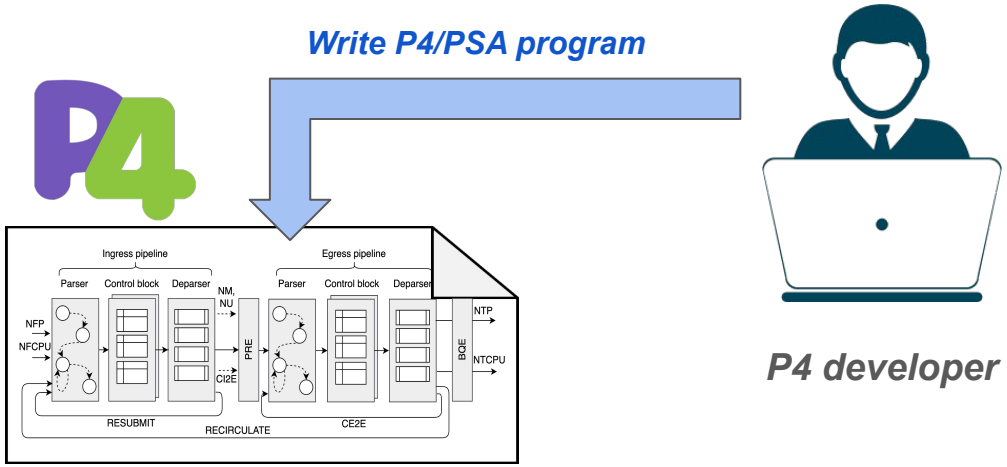


github.com/p4lang/p4c/tree/main/backends/ebpf/psa

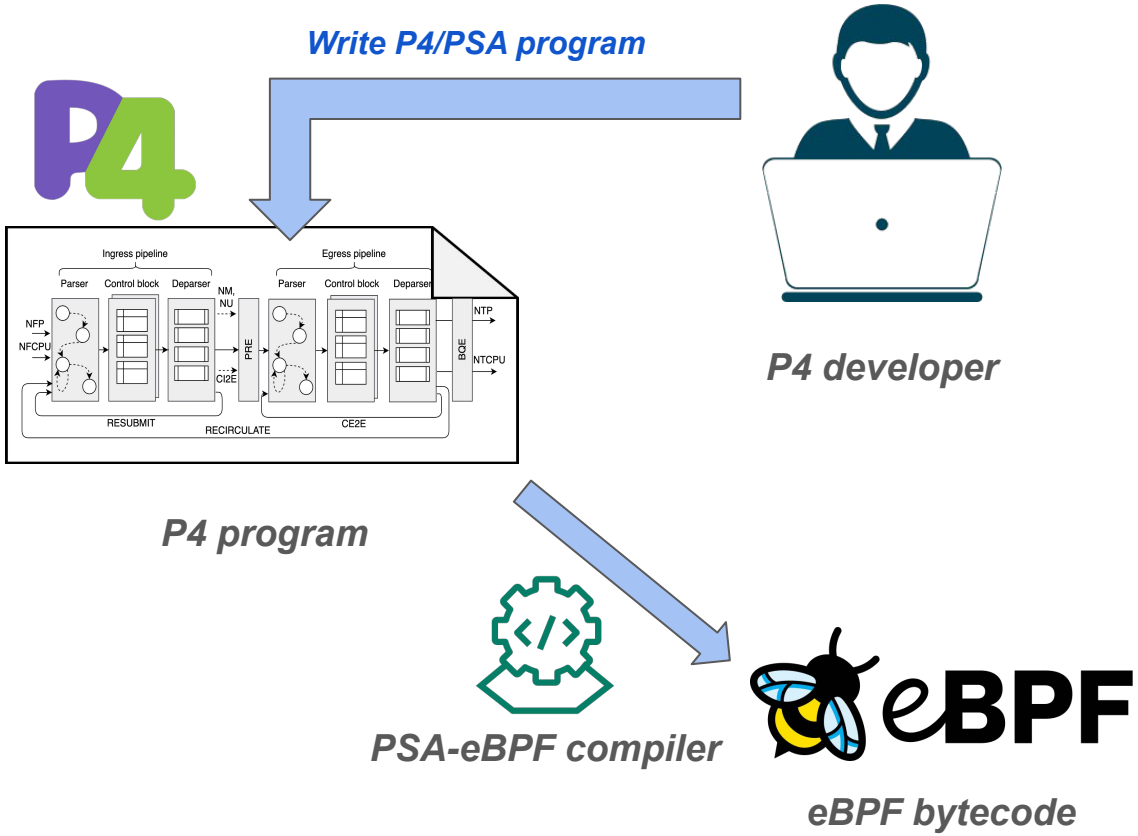
psabpf API / CLI tool

github.com/P4-Research/psabpf

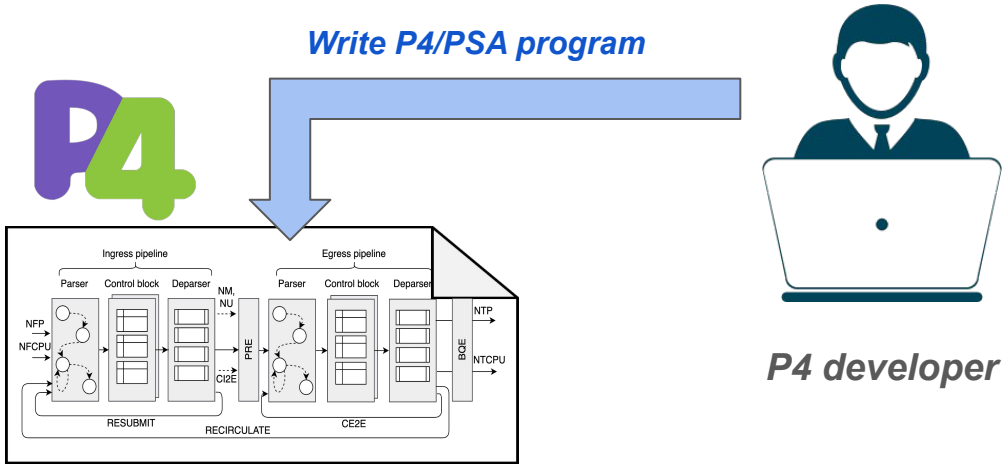
Overview of the PSA-eBPF workflow

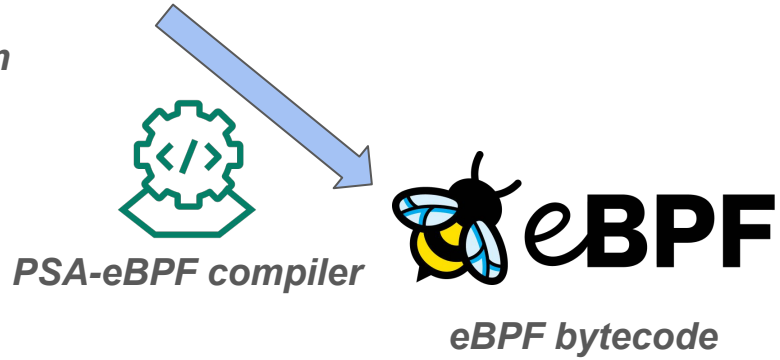


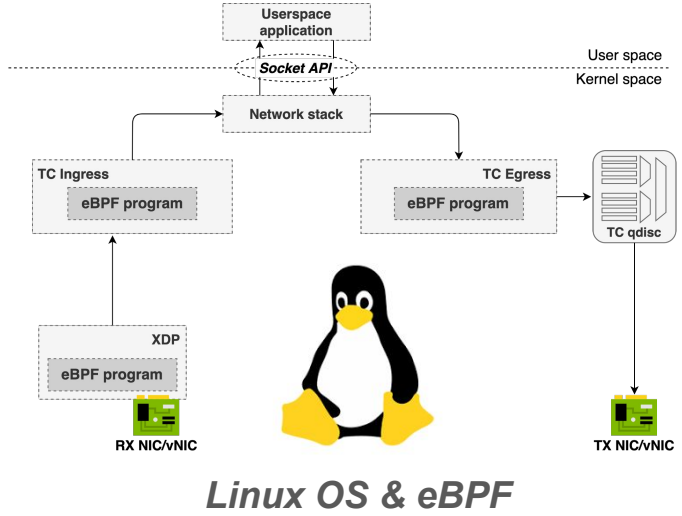
Overview of the PSA-eBPF workflow



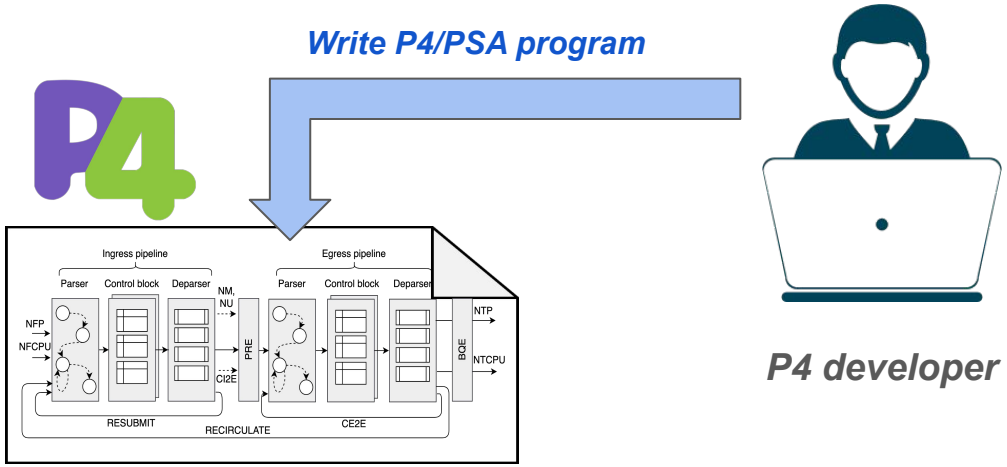
Overview of the PSA-eBPF workflow







Overview of the PSA-eBPF workflow



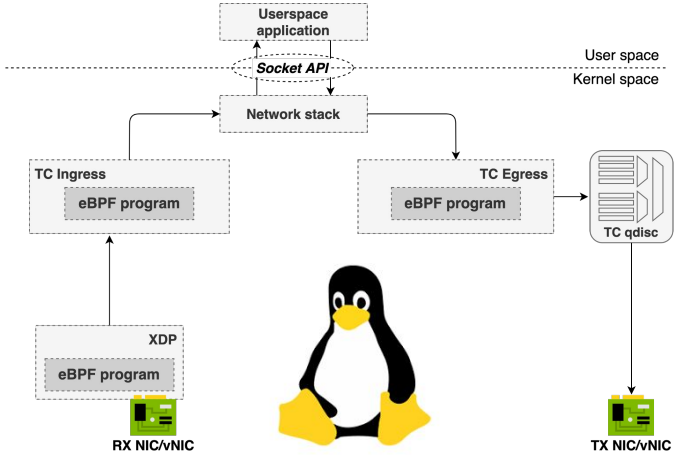
P4 program

P4 developer

PSA-eBPF compiler

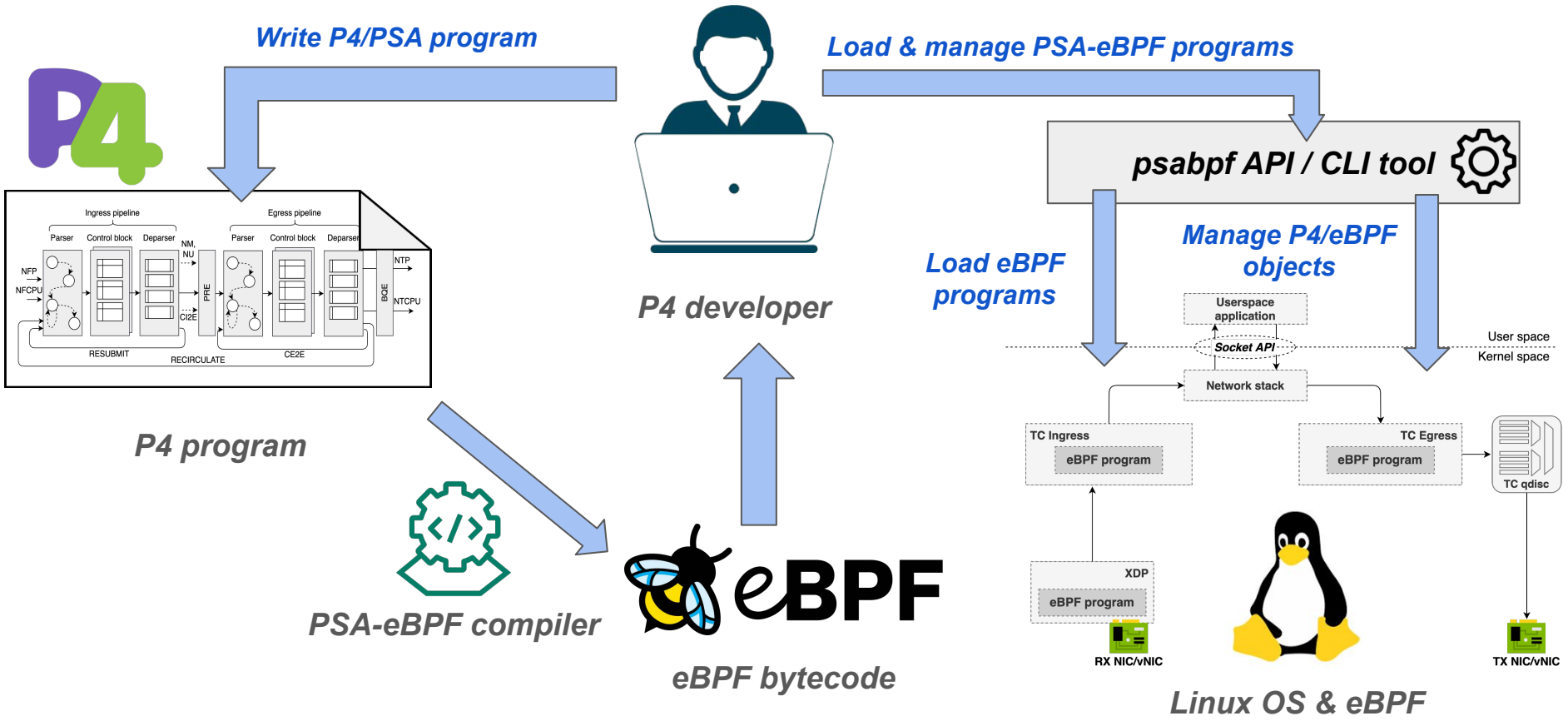
eBPF bytecode

psabpf API / CLI tool



Linux OS & eBPF

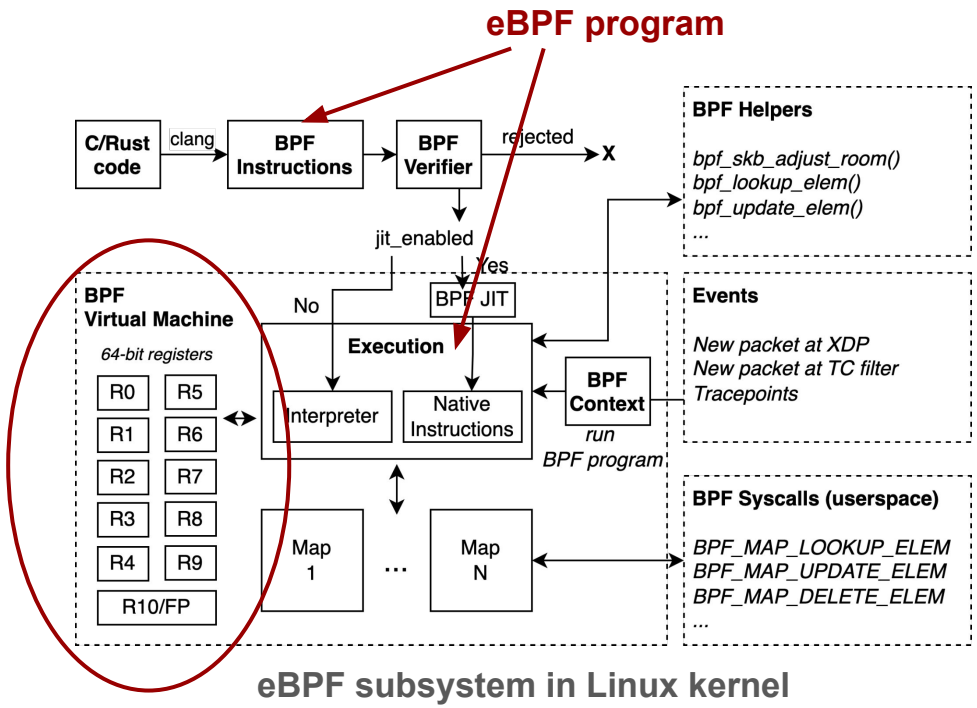
Overview of the PSA-eBPF workflow



Background

Introduction to eBPF

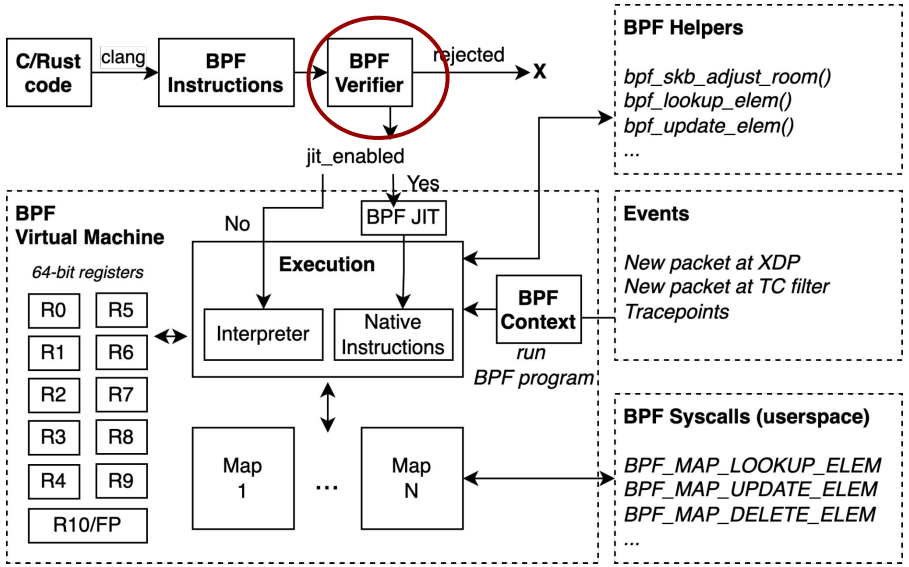
- extended Berkeley Packet Filter (eBPF)
  - **eBPF** provides **in-kernel virtual machine that can safely run eBPF programs**
  - **eBPF program** - set of instructions
    - JIT-ed or interpreted
  - eBPF programs can be injected at runtime (no kernel extensions, no reboot)



Cilium provides excellent [BPF and XDP Reference Guide]

Introduction to eBPF

- extended Berkeley Packet Filter (eBPF)
  - **eBPF** provides **in-kernel virtual machine that can safely run eBPF programs**
  - **eBPF program** - set of instructions
    - JIT-ed or interpreted
  - eBPF programs can be injected at runtime (no kernel extensions, no reboot)
- eBPF concepts:
  - **BPF verifier** - static in-kernel checker, verifies eBPF program before it's injected into the kernel

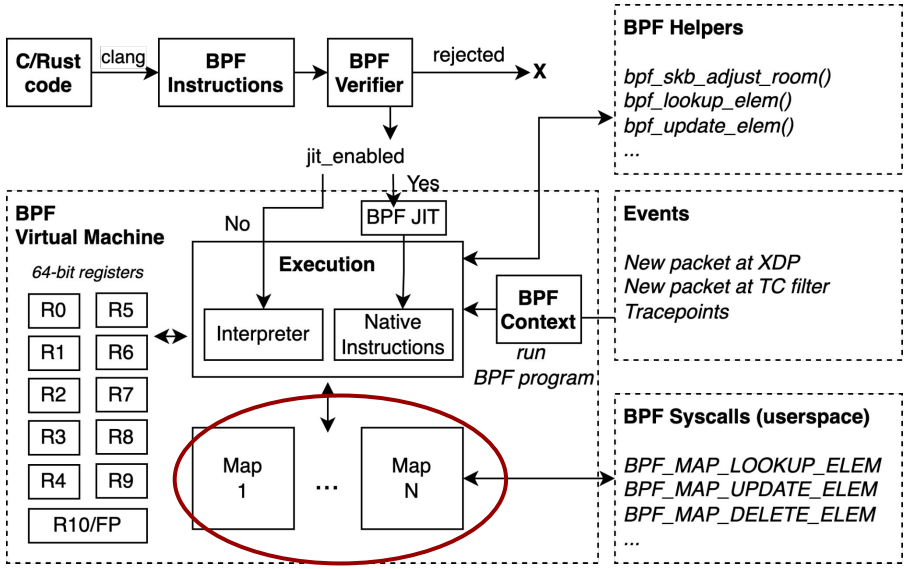


eBPF subsystem in Linux kernel

Cilium provides excellent [BPF and XDP Reference Guide]

Introduction to eBPF

- extended Berkeley Packet Filter (eBPF)
  - **eBPF** provides **in-kernel virtual machine that can safely run eBPF programs**
  - **eBPF program** - set of instructions
    - JIT-ed or interpreted
  - eBPF programs can be injected at runtime (no kernel extensions, no reboot)
- eBPF concepts:
  - **BPF verifier** - static in-kernel checker, verifies eBPF program before it's injected into the kernel
  - **BPF maps** - persistent key-value store (e.g., based on hash table)
    - can be accessed from userspace or from eBPF program

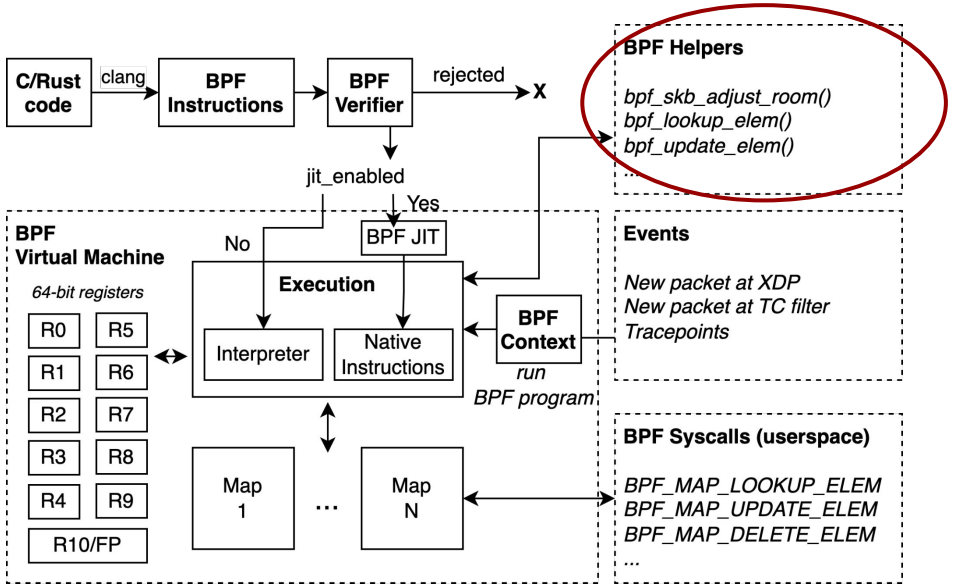


eBPF subsystem in Linux kernel

Cilium provides excellent [BPF and XDP Reference Guide]

Introduction to eBPF

- extended Berkeley Packet Filter (eBPF)
  - **eBPF** provides **in-kernel virtual machine that can safely run eBPF programs**
  - **eBPF program** - set of instructions
    - JIT-ed or interpreted
  - eBPF programs can be injected at runtime (no kernel extensions, no reboot)
- eBPF concepts:
  - **BPF verifier** - static in-kernel checker, verifies eBPF program before it's injected into the kernel
  - **BPF maps** - persistent key-value store (e.g., based on hash table)
    - can be accessed from userspace or from eBPF program
  - **BPF helpers** - external kernel functions that can be called from within eBPF program

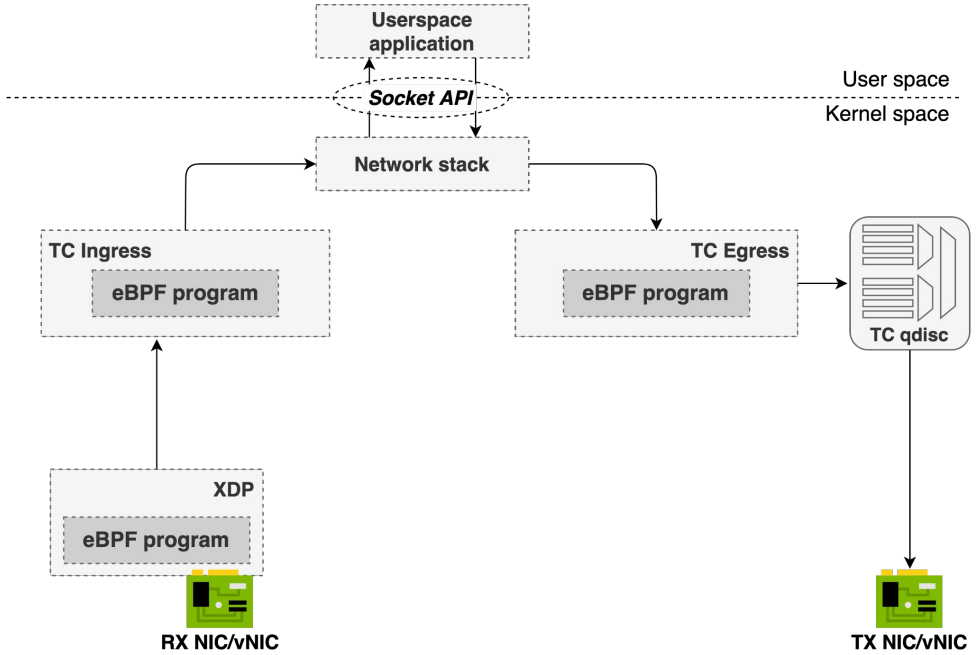


eBPF subsystem in Linux kernel

Cilium provides excellent [BPF and XDP Reference Guide]

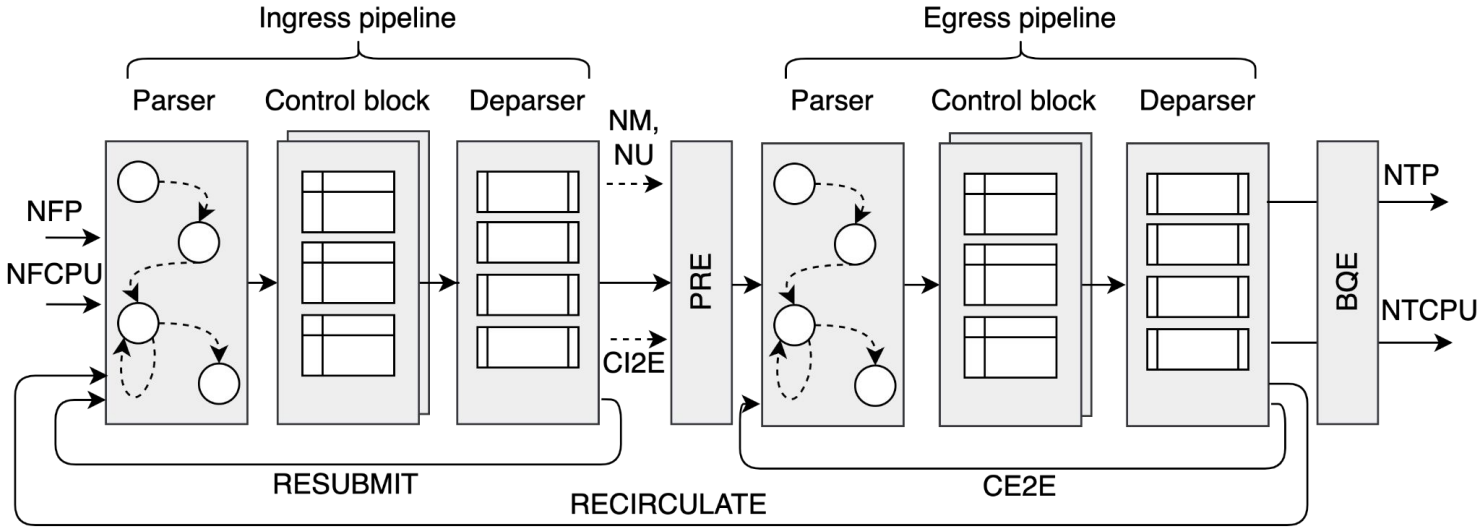
eBPF hooks

- **BPF hook** is an attachment point for eBPF programs
- Linux provides many BPF hooks..
- BPF hooks for packet processing:
  - XDP (eXpress Data Path)
    - runs in the NIC driver
    - Ingress only
    - high-speed packet processing
    - limited capabilities (e.g., no QoS)
  - TC (Traffic Control)
    - runs before Linux TCP/IP stack
    - Ingress + Egress
    - lower performance
    - more feature-rich (e.g., integrated with TC qdisc for QoS)



Portable Switch Architecture (PSA)

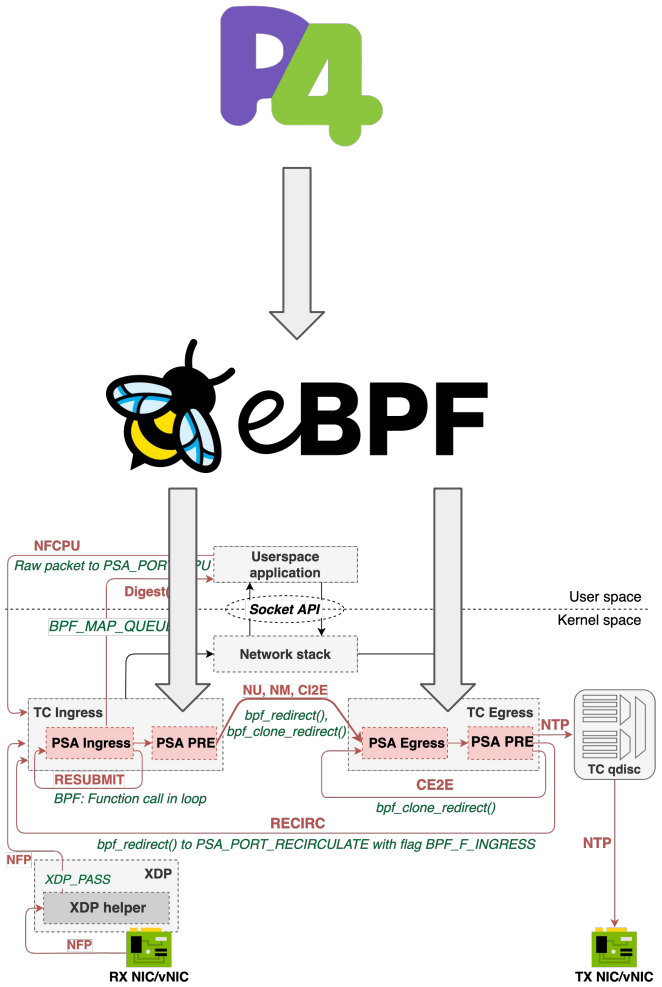
- *"The Portable Switch Architecture is a target architecture that describes common capabilities of network switch devices that process and forward packets across multiple interface ports."*
- PSA consists of:
  - 2 P4-programmable pipelines (ingress/egress)
  - Packet Replication Engine (PRE) and Buffer Queueing Engine (BQE)
  - Basic primitives (e.g., to send or drop packet) and set of PSA externs (e.g., Meter, Register, etc.)
  - Pre-defined packet paths (RESUBMIT, RECIRCULATE, etc.)



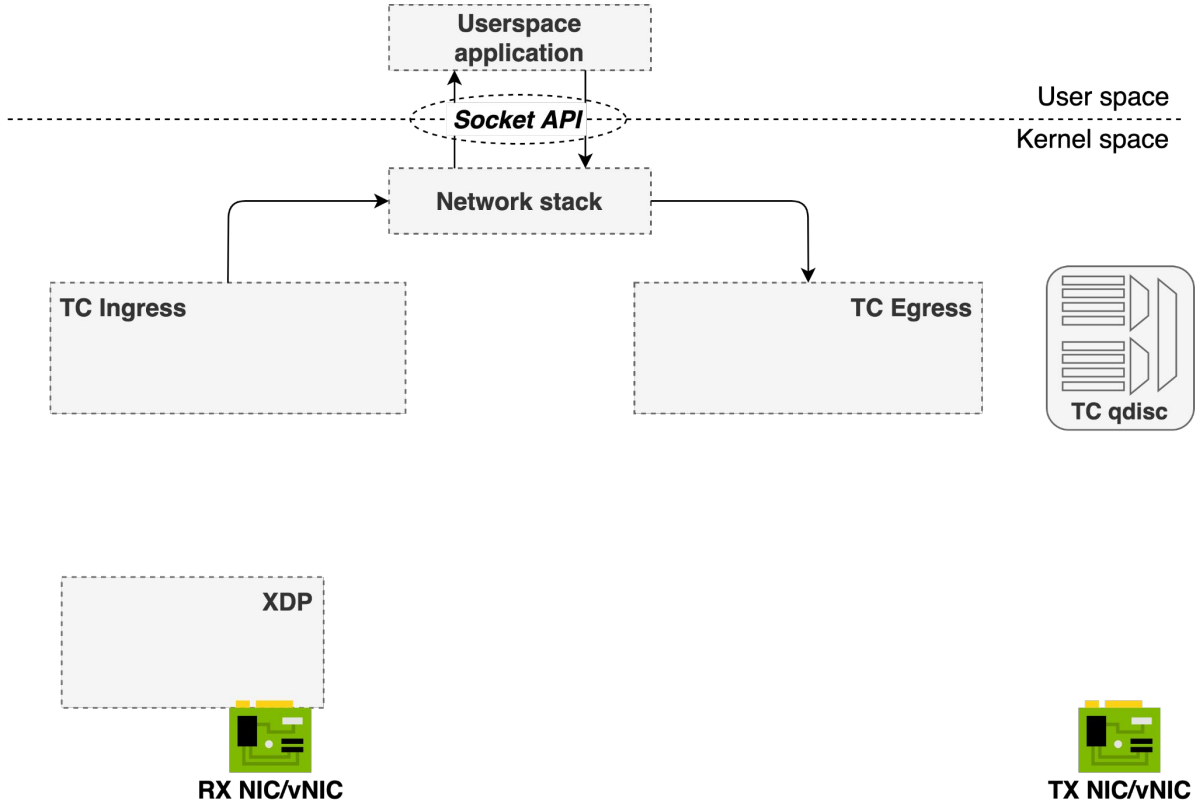
Deep dive into PSA-eBPF design & implementation

PSA-eBPF compiler

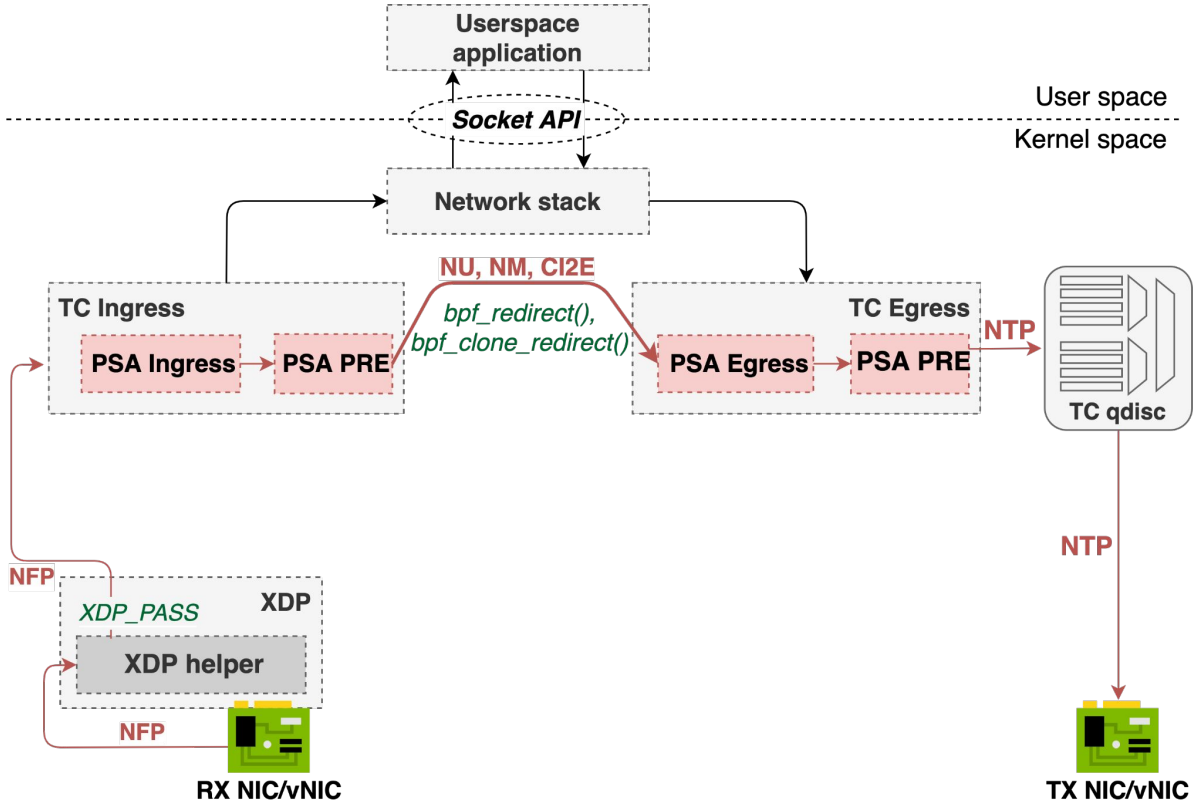
- PSA-eBPF compiler
  - translates P4 programs written for PSA to eBPF programs
  - extends the eBPF backend of the open-source P4 compiler:
    - https://github.com/p4lang/p4c/tree/main/backends/ebpf/psa
- PSA-eBPF compiler generates a set of eBPF programs
  - **General-purpose TC-based design** that implements ***any*** PSA program at the cost of performance
  - High-performance XDP-based design coming up next..



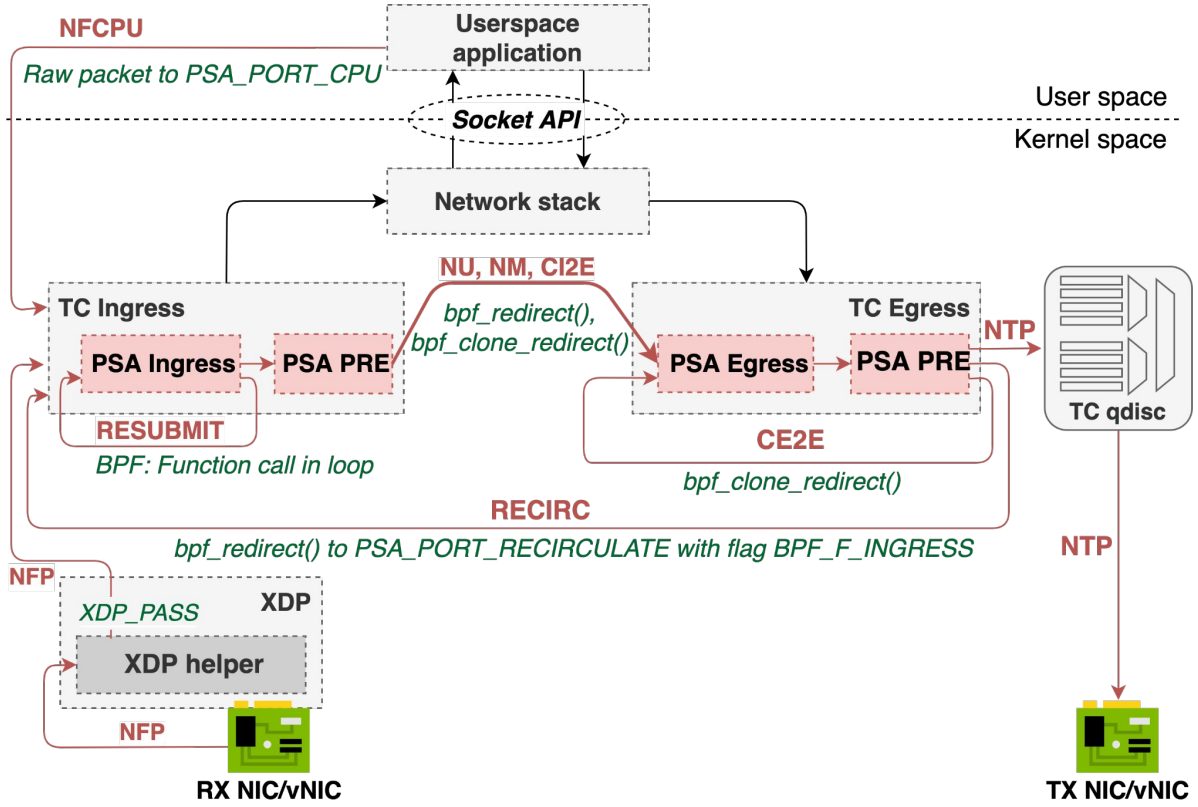
TC-based design of PSA-eBPF



TC-based design of PSA-eBPF

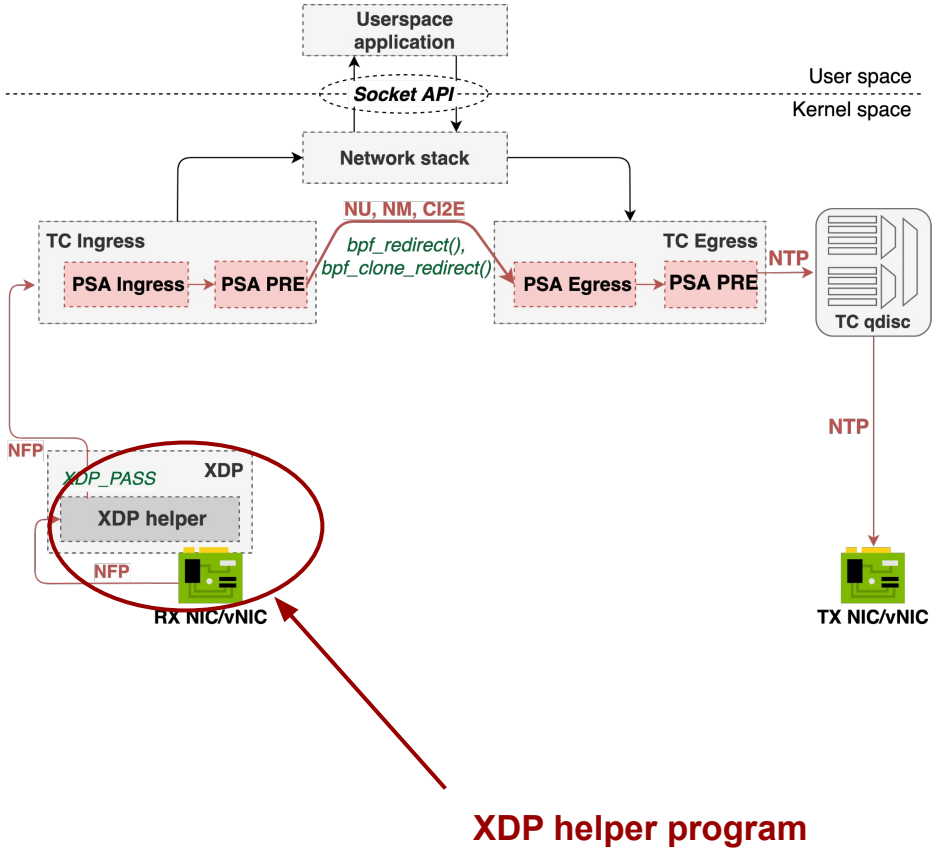


TC-based design of PSA-eBPF



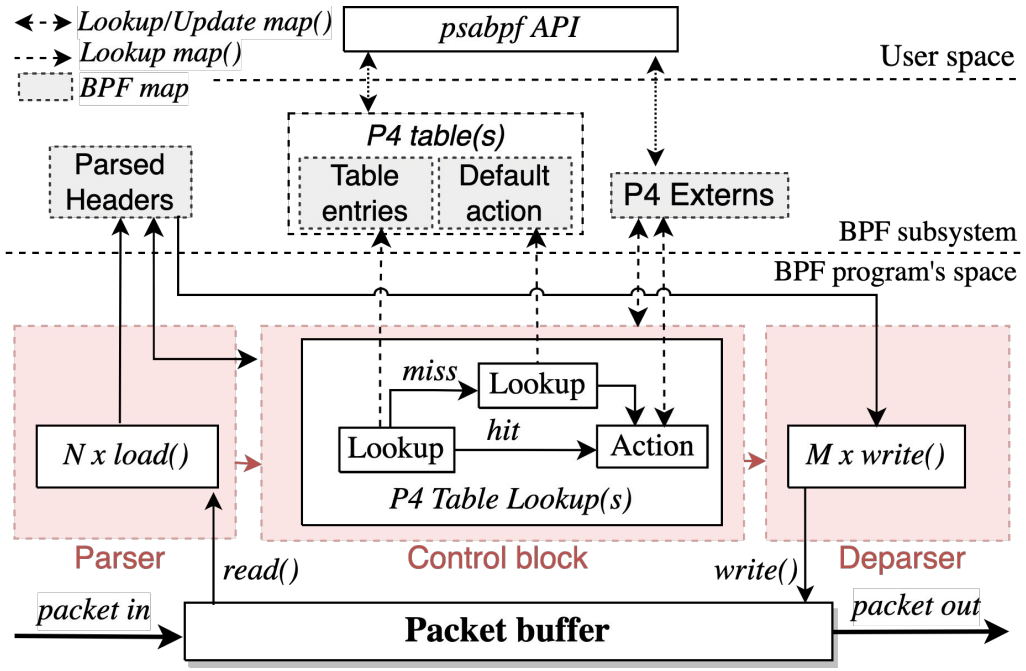
XDP helper program

- Non-programmable eBPF program attached to XDP
- Makes TC protocol-independent..
  - Reads original EtherType & saves it as “XDP2TC” metadata
  - Replaces original EtherType with IPv4
  - Sends “XDP2TC” metadata up to TC
  - TC Ingress must restore original EtherType from “XDP2TC” metadata
- XDP2TC metadata modes (compiler flag):
  - **meta** - uses *bpf_xdp_adjust_meta()* helper + *skb->data_meta* field to send EtherType to TC
  - **head** - uses *bpf_xdp_adjust_head()* helper to prepend a packet with EtherType
  - **cpumap** (testing only) - uses BPF per-CPU array map to transfer EtherType from XDP to TC
- What are the consequences?
  - You’ll see malformed packets (invalid EtherType) when sniffing interfaces with tcpdump/tshark
  - eBPF program must be attached to “native” XDP hook ⟶ some NIC drivers may not support native XDP, for vEth pairs XDP program must be attached to both ends…



PSA pipeline to eBPF translation

- PSA pipeline (Ingress/Egress) is translated to single eBPF program
  - PSA pipeline consists of Parser, Control block and Deparser
  - eBPF program operates on packet buffer via *skb* (socket buffer) descriptor
- Parser:
  - packet headers are read by load() functions and saved into BPF array map
  - supports ValueSet, lookahead(), advance(), length()..
- Control block:
  - P4 table is translated to set of BPF maps (depending on match kind) + BPF array map storing default action
  - Most P4 externs uses a combination of BPF maps + extern-specific code in data plane
- Deparser:
  - constructs outgoing packet based on packet headers stored in the “Parsed Headers” map



P4 tables & match kinds

- Each P4 table is translated to:
  - BPF map(s) depending on the P4 match kind
  - BPF array map with a single element storing default P4 action
- **exact** table - a P4 table consisting of exact match kinds only
  - translated to **BPF hash map**
- **lpm** tables - a P4 table with at least one lpm match field (no ternary/range)
  - translated to **BPF LPM_TRIE map**
- **ternary** tables - a P4 table with at least one ***ternary*** field
  - no built-in eBPF primitive
  - translated to **set of BPF hash & array maps implementing** Tuple Space Search
- **range** tables are not supported yet!
  - can be implemented by naive range-to-prefix conversion

PSA externs

- Most PSA externs uses a combination of eBPF maps & helpers to achieve the goal
- Refer to PSA-eBPF documentation for more technical details

PSA extern	eBPF map types	Data plane code
Action Profile	Additional BPF hash map storing mapping between member ID and action spec.	2 lookups (instead of 1) to BPF maps
Action Selector	3 additional BPF maps: - BPF map of maps storing ActionSelector groups and its members - BPF map storing action for empty group - BPF map for member ID -> action spec. mapping	Sequential lookups to BPF maps + set of eBPF instructions calculating hash value from selector fields to dynamically select member from ActionSelector group
Counter	BPF array map per instance	Lookup to BPF array map + *__sync_fetch_and_add()*
DirectCounter	-	*__sync_fetch_and_add()*
Digest	BPF queue map (FIFO) per instance	*bpf_map_push_elem()*

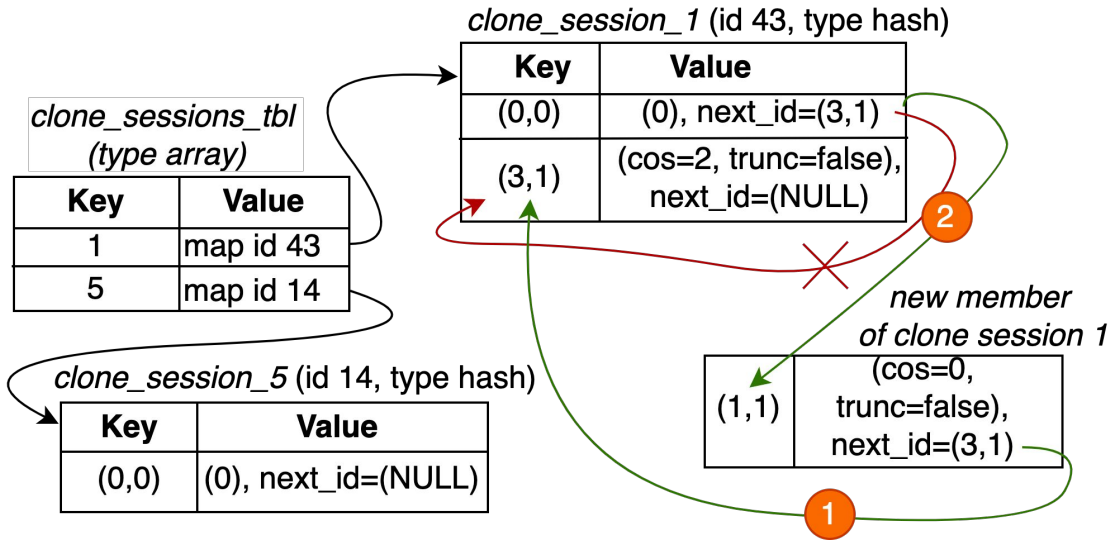
PSA externs

- Most PSA externs uses a combination of eBPF maps & helpers to achieve the goal
- Refer to PSA-eBPF documentation for more technical details

PSA extern	**eBPF map types**	**Data plane code**
Meter	BPF hash map + BPF spinlock	*bpf_ktime_get_ns()* + lookup & update to modify Meter state
DirectMeter	BPF spinlock only (for exact tables, lpm/ternary not supported)	*bpf_ktime_get_ns()* + update to modify Meter state
Register	BPF map per instance: *array* if width type <= bit<32>, *hash* otherwise	*bpf_map_lookup_elem*() for Register read(), *bpf_map_update_elem*() for Register write()
Hash, Checksum, Internet Checksum	-	Set of eBPF instructions calculating hash or checksum from a given data.
Random	-	*bpf_get_prandom_u32()* to get pseudo-random number

Packet Replication Engine (PRE) in PSA-eBPF

- PRE functionality is integrated into eBPF programs for Ingress (CI2E) & Egress (CE2E)
- Clone sessions/multicast groups are organized as ***BPF map of maps***
- Data plane implementation:
  - eBPF program performs lookup to an outer array map based on clone_session_id or multicast_group (from PSA metadata) to get an inner map
  - The inner map stores all clone session/multicast group members
  - eBPF prog iterates over inner map entries and calls ***bpf_clone_redirect()*** for each member

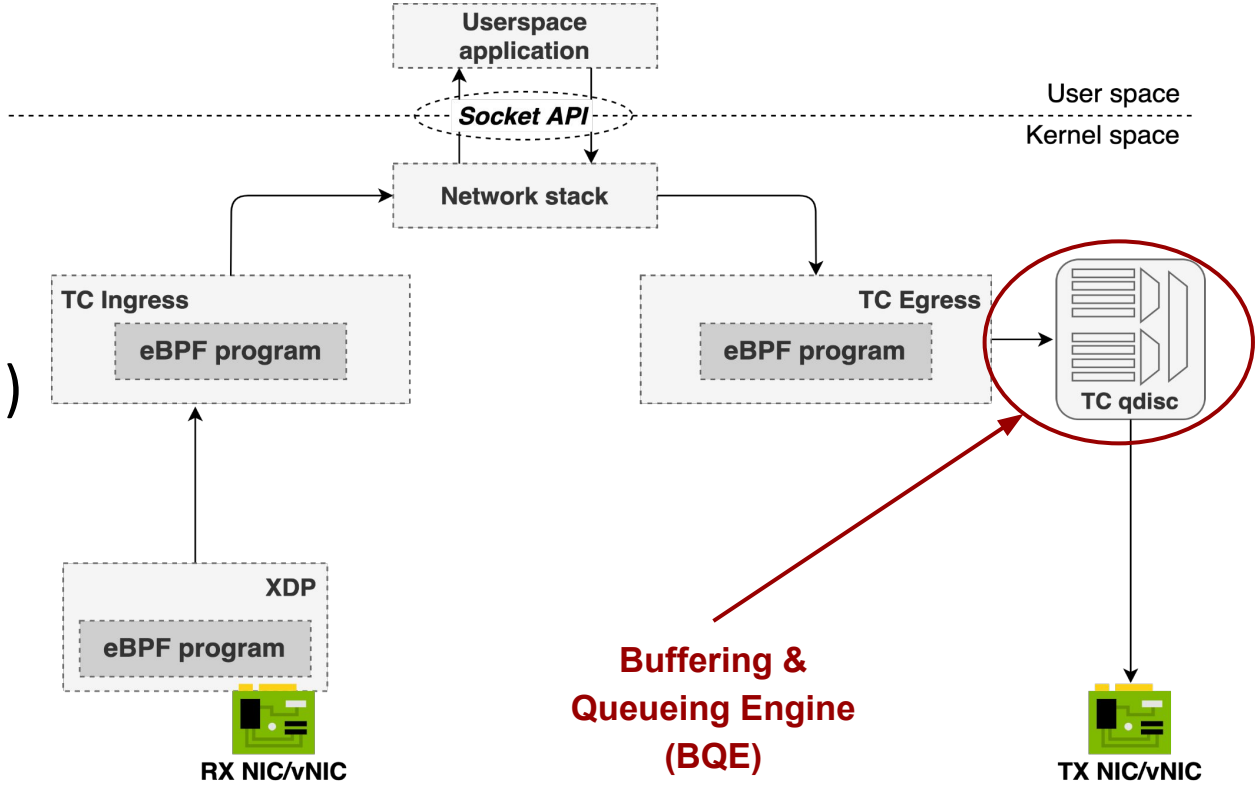


Memory organization of PRE in PSA-eBPF:

1) a new clone session member is allocated and the next element is set as an element located in the front of the list
2) the head of the list is updated to point to a new element

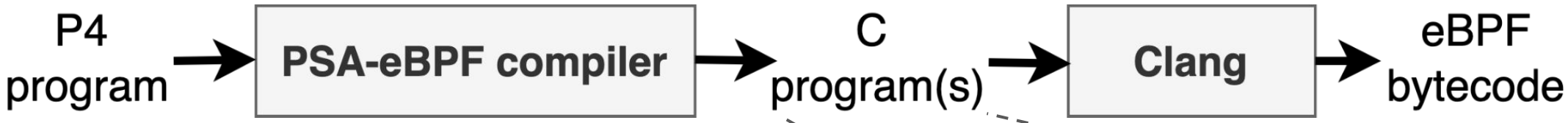
Buffering & Queueing Engine (BQE)

- PSA-eBPF leverages TC qdisc to map BQE functionality
- TC qdisc is a powerful QoS engine:
  - Shaping, scheduling, policing, dropping..
  - Classful disciplines (e.g., HQoS)
  - Classless disciplines (e.g., fq_codel)
- PSA-eBPF <-> TC qdisc interface via PSA *class_of_service*:
  - skb->priority is set based on PSA class_of_service
  - skb->priority is analyzed by TC qdisc to select QoS class



Getting started with PSA-eBPF

Compile P4/PSA to eBPF

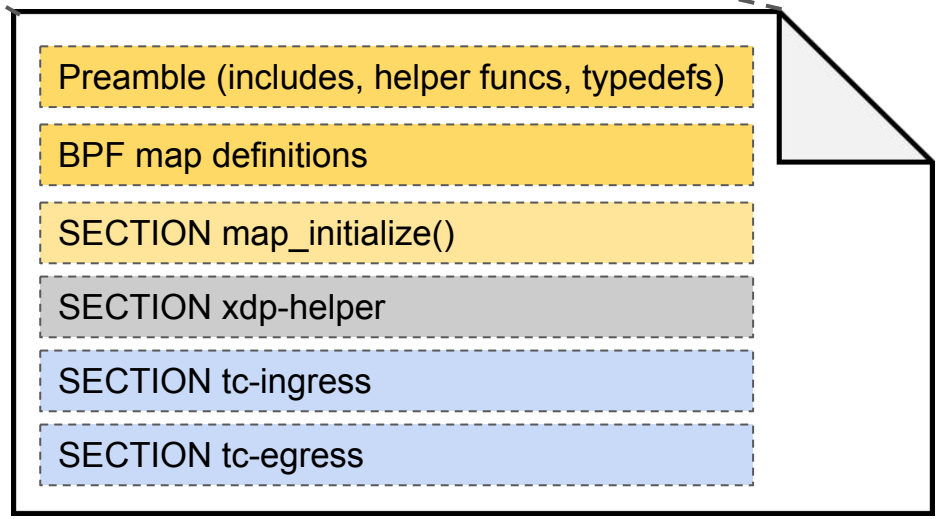


To compile:

```
$ p4c-ebpf -arch psa demo.p4 -o demo.c
$ P4C=$(path_to_p4c_repository)/backends/ebpf/runtime
$ clang -O2 -g -c -DPSA_PORT_RECIRCULATE=2 -I$P4C/usr/include/
-I$P4C/ -emit-llvm -DBTF -o demo.bc demo.c
$ llc -march=bpf -mcpu=generic -filetype=obj -o demo.o demo.bc
```

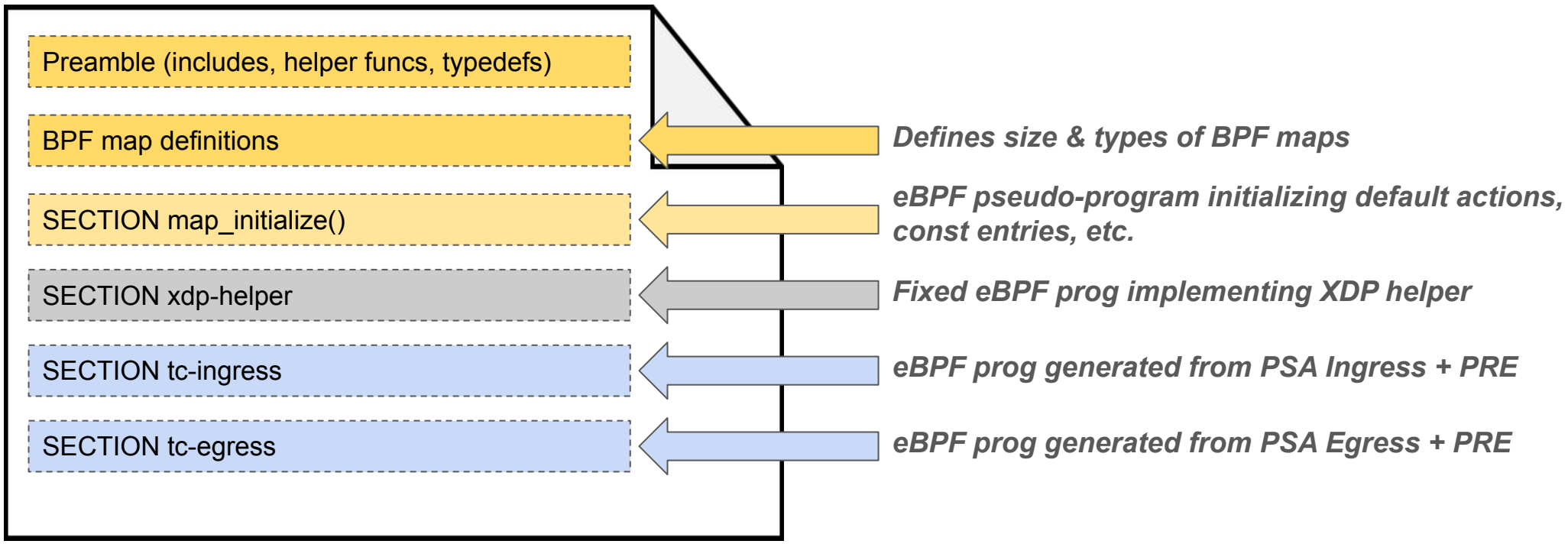
Two-step process automated via Makefile:

```
$ make -f ${P4C_REPO}/backends/ebpf/runtime/kernel.mk \
BPFOBJ=out.o P4FILE=program.p4 ARGS="-DPSA_PORT_RECIRCULATE=2" \
psa
```



Structure of C program generated by PSA-eBPF compiler

Structure of C program generated by PSA-eBPF compiler



psabpf - a low-level library + psabpf-ctl CLI

- Designed to manage PSA-eBPF programs in almost every aspect:
  - Load & unload programs from eBPF subsystem
  - Attach & detach programs from network interfaces
  - Manage P4 tables & externs
  - Configure Packet Replication Engine
- psabpf is a low-level library to manage PSA-eBPF programs:
  - Exposes C API
  - Hides dependency on libbpf and its API
  - Use BTF (BPF Type Format) to operate on eBPF maps
  - p4info isn't used

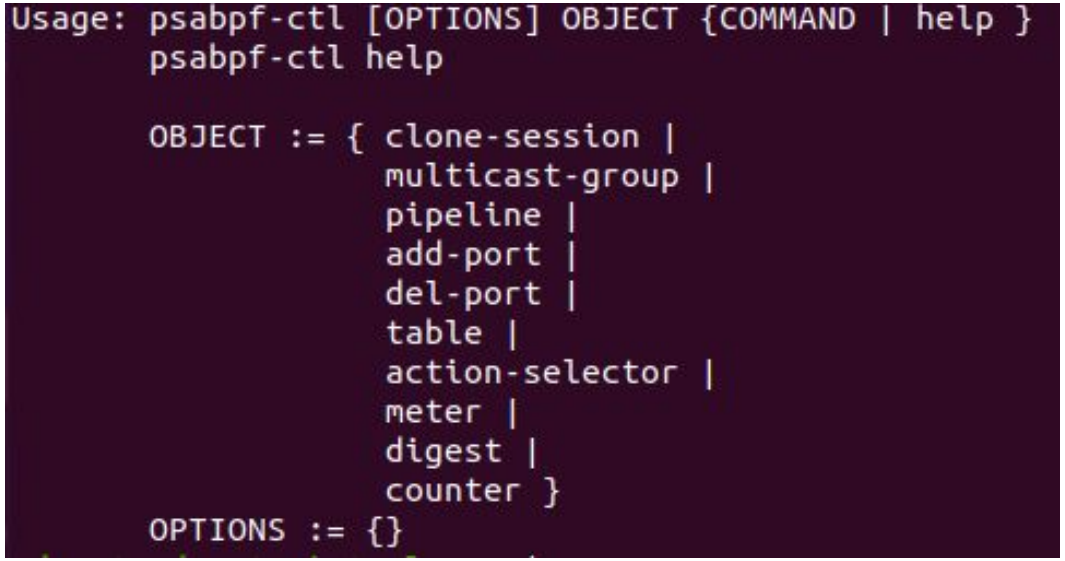
```c
void psabpf_digest_ctx_init(psabpf_digest_context_t *ctx);
void psabpf_digest_ctx_free(psabpf_digest_context_t *ctx);
int psabpf_digest_ctx_name(psabpf_context_t *psabpf_ctx, psabpf_digest_context_t *ctx, const char *name);
/* Will initialize digest, but must be later freed */
int psabpf_digest_get_next(psabpf_digest_context_t *ctx, psabpf_digest_t *digest);
void psabpf_digest_free(psabpf_digest_t *digest);

psabpf_struct_field_t * psabpf_digest_get_next_field(psabpf_digest_context_t *ctx, psabpf_digest_t *digest);
```

psabpf C API for Digest extern

psabpf-ctl - a CLI tool

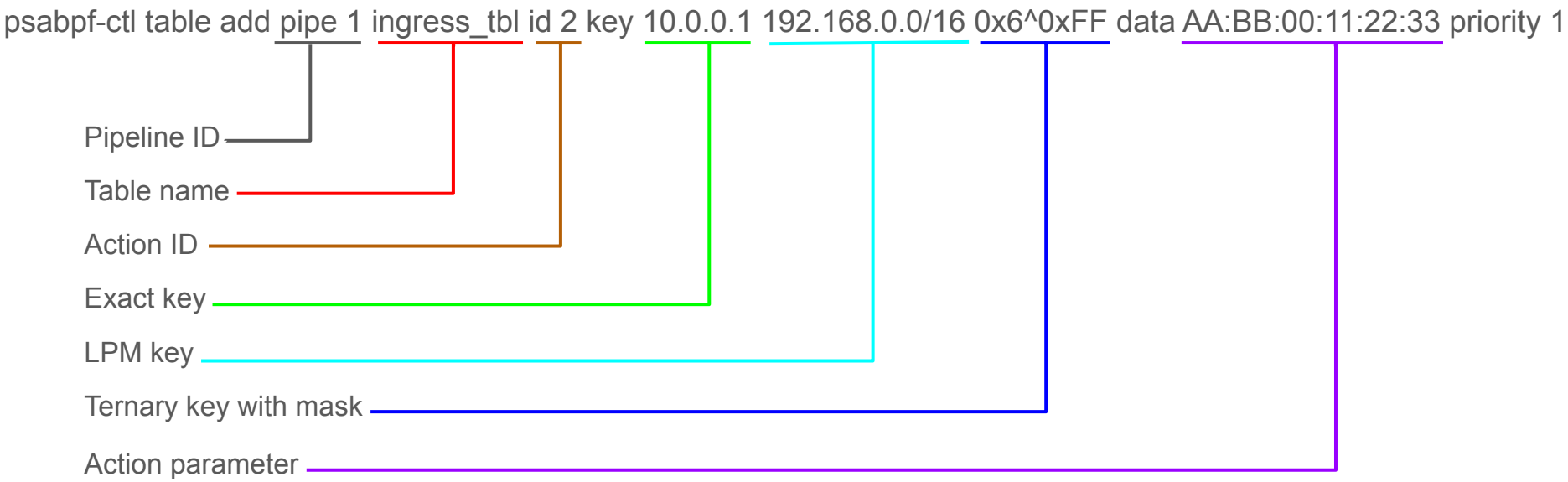
- psabpf-ctl is a CLI tool bundled within psabpf repository
  - Uses C API from psabpf
  - Currently provides CLI commands for: pipeline management, add/remove ports, clone sessions, multicast groups, table management, operations on externs (Action Selector, Meter, Digest, Counter)
- Both psabpf C API & psabpf-ctl are still under development
  - some externs are not supported yet



Sample usage of psabpf-ctl

```
table tbl {
    key = {
        hdr.ipv4.src_addr : exact;
        hdr.ipv4.dst_addr : lpm;
        hdr.ipv4.protocol  : ternary;
    }
    actions = { act_A; act_B; NoAction;}
} /* action ID:     1        2          0 */
```

sample P4 table



Load & initialize PSA-eBPF programs with psabpf

- Load PSA-eBPF programs into eBPF subsystem
  - Programs verified by in-kernel BPF verifier
- Pin BPF objects under /sys/fs/bpf/pipelineN
  - N is a pipeline ID
  - Maps are pinned under /sys/fs/bpf/pipelineN/maps/ subdirectory
- Initialize maps by running map_initilize() with *bpf_prog_test_run()*
  - default values, const entries, etc.
  - Called only once
- Done by single psabpf-ctl command:

```
$ psabpf-ctl pipeline load id N FILENAME.o
```

Attach/detach ports with psabpf

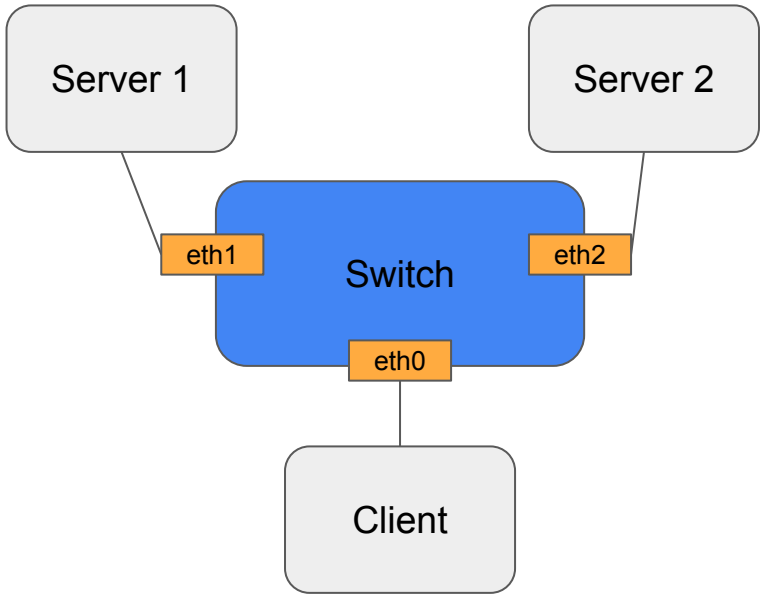
- Loaded eBPF programs must be attached to network interface to process packets
- There are three programs to attach for every single interface:
  - TC Ingress
  - TC Egress
  - XDP helper
- Single command to attach/detach an interface:

```
$ psabpf-ctl add-port pipe N dev INTERFACE (to attach interface)
$ psabpf-ctl del-port pipe N dev INTERFACE (to detach interface)
```

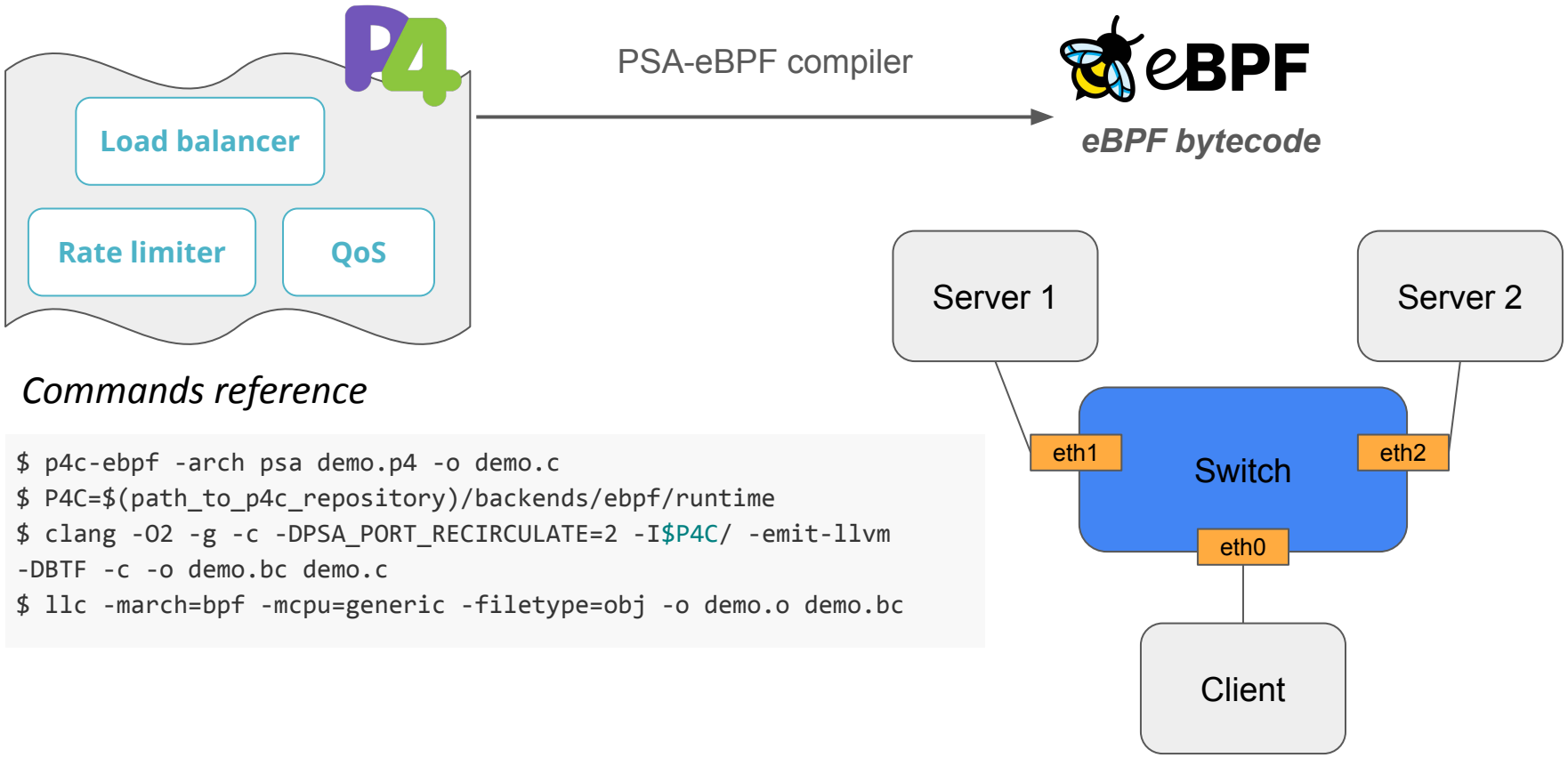
Hands-on session

Hands-on session plan

- Demo topology:
  - 4 Linux namespaces (client, switch, server1, server2)
  - All-in-one P4 program compiled to eBPF
- Hands-on session steps:
  - How to compile P4/PSA program to eBPF?
  - How to load PSA/eBPF programs with psabpf-ctl?
  - How to attach PSA/eBPF program to ports?
  - **Scenario 1: Load balancer**
  - **Scenario 2: Rate limiter**
  - **Sceanrio 3: Traffic prioritization (QoS)**
- Steps to reproduce:
  - https://github.com/P4-Research/psa-ebpf-demo



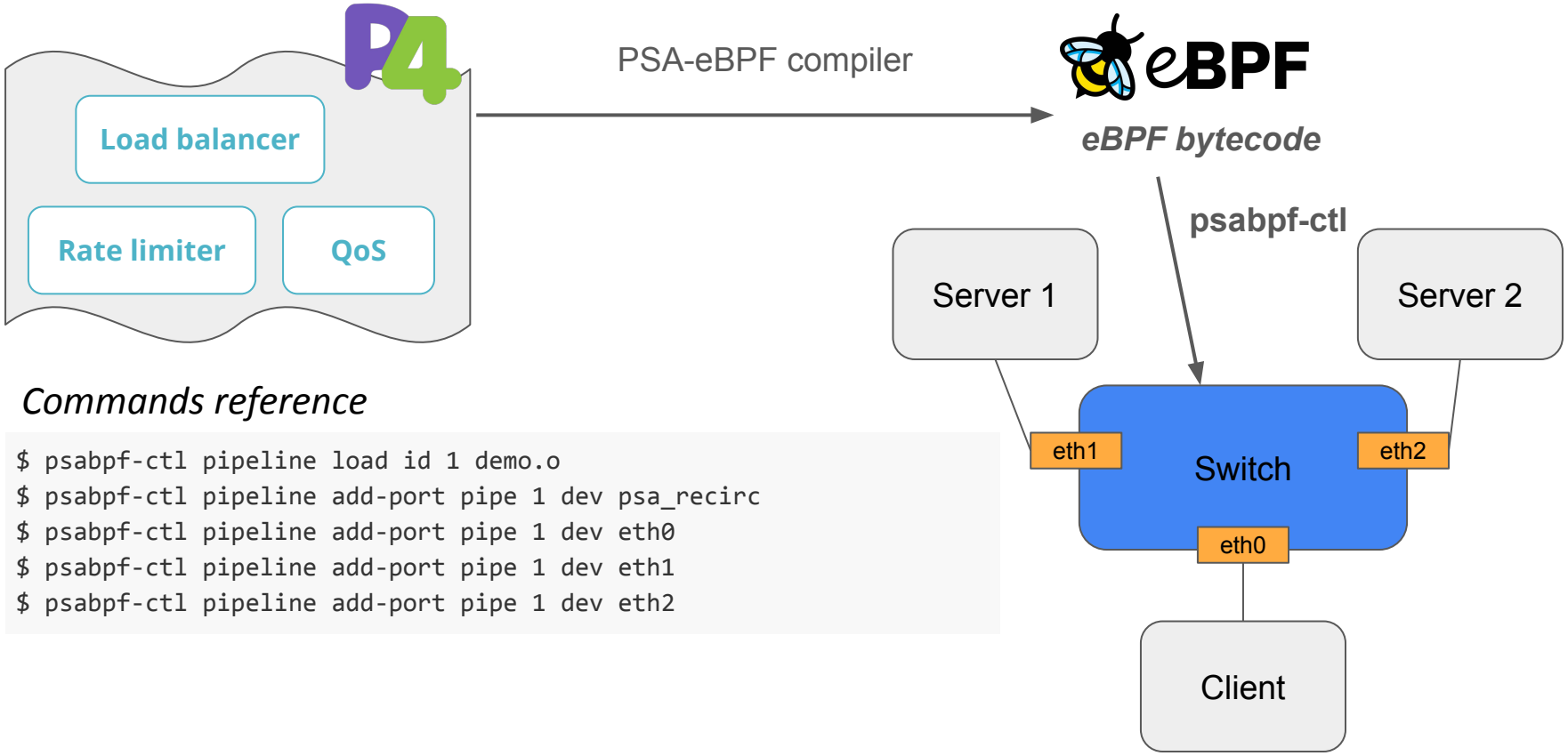
How to compile P4/PSA program to eBPF?



Commands reference

```
$ p4c-ebpf -arch psa demo.p4 -o demo.c
$ P4C=$(path_to_p4c_repository)/backends/ebpf/runtime
$ clang -O2 -g -c -DPSA_PORT_RECIRCULATE=2 -I$P4C/ -emit-llvm
-DBTF -c -o demo.bc demo.c
$ llc -march=bpf -mcpu=generic -filetype=obj -o demo.o demo.bc
```

How to load PSA/eBPF programs and attach to ports?

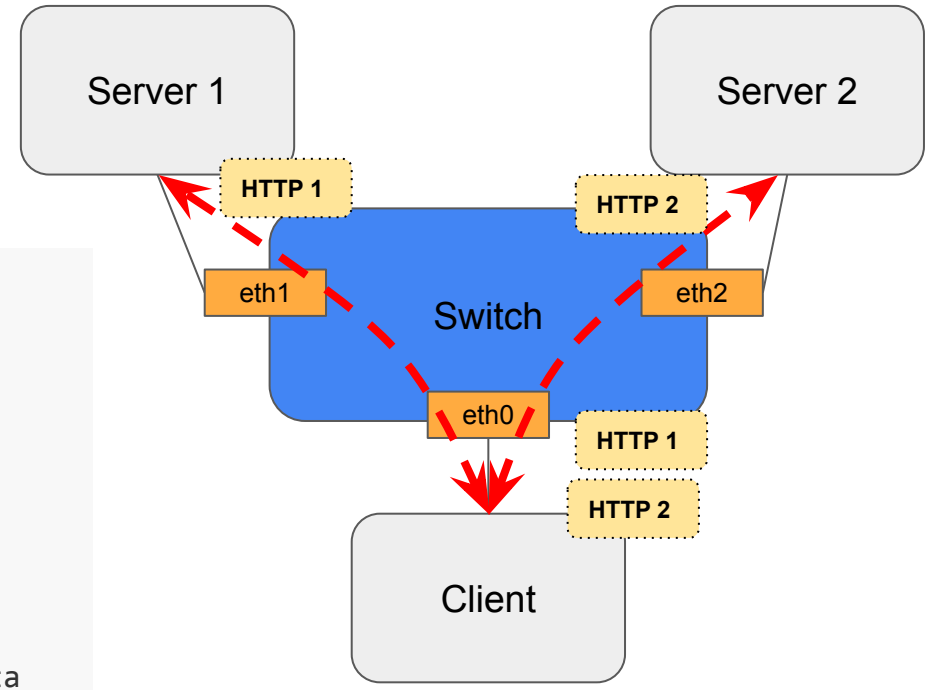


Commands reference

```
$ psabpf-ctl pipeline load id 1 demo.o
$ psabpf-ctl pipeline add-port pipe 1 dev psa_recirc
$ psabpf-ctl pipeline add-port pipe 1 dev eth0
$ psabpf-ctl pipeline add-port pipe 1 dev eth1
$ psabpf-ctl pipeline add-port pipe 1 dev eth2
```

Scenario 1: Load balancer

- Performs load-balancing between HTTP servers running on Server 1 and 2
  - Uses Action Selector to dynamically select output port based on 5-tuple



Commands reference

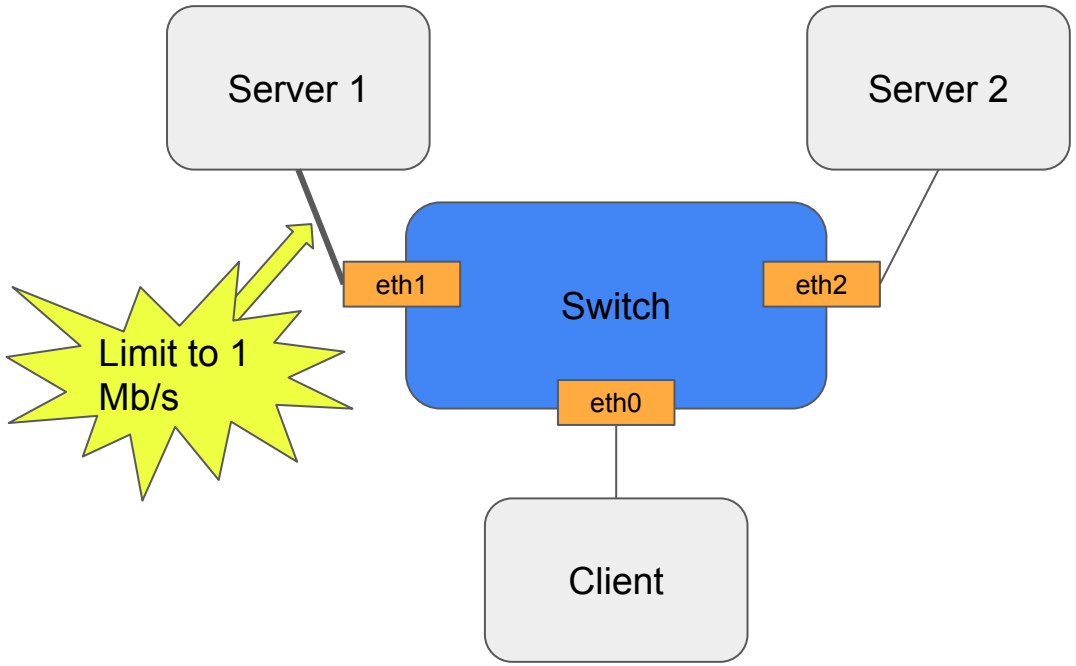
```
$ psabpf-ctl action-selector add_member pipe 1 DemoIngress_as id 2 data 1
SW:IT:CH:MA:C0:01 SE:RV:ER:MA:C0:01 17.0.0.1
$ psabpf-ctl action-selector add_member pipe 1 DemoIngress_as id 2 data 2
SW:IT:CH:MA:C0:02 SE:RV:ER:MA:C0:02 17.0.0.2
$ psabpf-ctl action-selector create_group pipe 1 DemoIngress_as
$ psabpf-ctl action-selector add_to_group pipe 1 DemoIngress_as 4 to 1
$ psabpf-ctl action-selector add_to_group pipe 1 DemoIngress_as 5 to 1
$ psabpf-ctl action-selector add_member pipe 1 DemoIngress_as id 3 data 0
SW:IT:CH:MA:C0:00 CL:IE:NT:MA:C0:00 10.192.168.44
$ psabpf-ctl table add pipe 1 DemoIngress_tbl_routing ref key "10.192.168.44/32" data
group 1
$ psabpf-ctl table update pipe 1 DemoIngress_tbl_routing ref key "10.0.0.1/24" data 6
```

Scenario 2: Rate limiter

- Performs rate limiting for UDP flows between Client and Server 1
  - Uses P4 Meter to rate-limit traffic on output port

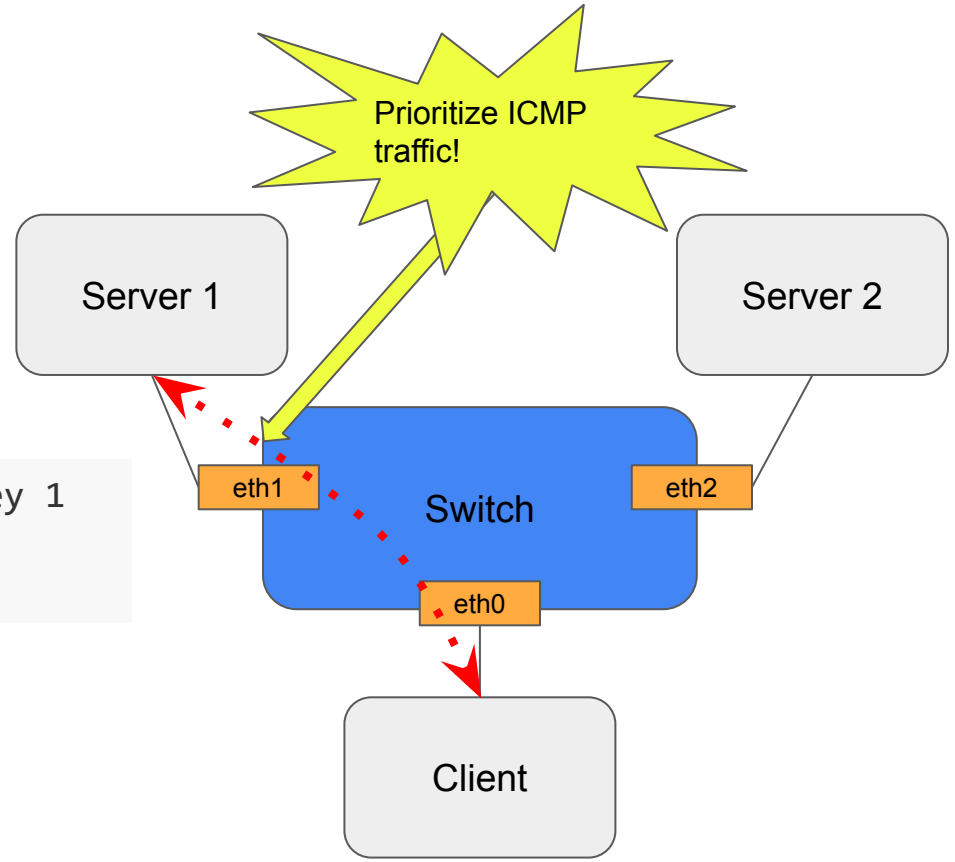
Commands reference

```
# 1 Mb/s -> PIR&CIR=125kbytes/s, PBS&CBS=10kbytes
$ psabpf-ctl meter update pipe 1 DemoIngress_meter index 1
125000:10000 125000:10000
```



Scenario 3: Traffic prioritization (QoS)

- Performs traffic prioritization for ICMP flows
  - Uses class_of_service to determine QoS class in TC qdisc



Commands reference

```
$ psabpf-ctl table add pipe 1 DemoIngress_qos_classifier id 1 key 1
```

Live session

Run PSA-eBPF with Mininet!

- We've prepared a basic Python/Mininet wrapper on top of psabpf-ctl: https://github.com/P4-Research/psabpf/blob/master/mininet/lib/psabpf_mn.py
- Mininet demo!
  - https://github.com/P4-Research/psabpf/tree/master/mininet

Troubleshooting PSA-eBPF

- *--trace* compiler flag - programs will log what they are doing
  - *bpftool prog tracelog*
  - *cat /sys/kernel/debug/tracing/trace_pipe*
- bpftool - https://github.com/torvalds/linux/tree/master/tools/bpf/bpftool
  - *bpftool prog show* - list loaded programs into eBPF subsystem
  - *bpftool net show* - list interfaces with attached programs
  - *bpftool map show, bpftool map dump* - list maps and dump its content
  - *bpftool prog dump* - allow to manually inspect generated eBPF bytecode
- Remove programs from eBPF subsystem
  - *psabpf-ctl del-port pipe N dev INTERFACE* - detaches programs from interface
  - *psabpf-ctl pipeline unload id N* - removes program from eBPF subsystem
- Clear state of eBPF subsystem
  - rm -rf /sys/fs/bpf/pipelineN and detach programs from all interfaces
  - Reboot OS :)

Summary

Programming guidelines

- **Avoid egress processing** - if the PSA Egress pipeline is empty, the PSA-eBPF compiler optimizes the eBPF egress program out, resulting in a noticeable performance gain
- **Minimize P4 table lookup** - each P4 table lookup contributes additional processing overhead, even if the result is not used. If possible, minimize table lookups (e.g., guard P4 table lookup by header validity or exit pipeline earlier)
- **Use index type equal or shorter than bit<32> for PSA externs** - in such case, PSA-eBPF compiler uses a more efficient BPF array map for Counters, Registers and Meters.
- **If possible, prefer direct externs over indirect externs** - indirect externs (Counter/Meter) require additional lookup to a BPF map, so using direct externs reduces the processing overhead as only a single lookup is generated

Summing up..

- PSA-eBPF brings the power of P4/PSA to Linux & eBPF and enables programming advanced use cases for end-host networking with P4!
- Possible advanced use cases:
  - P4-programmable K8s plugin (e.g., with Host-INT support)
  - software-based P4 implementation of 5G UPF
  - P4-programmable datapath for BIRD (or other routing daemons)
- Future work:
  - P4Runtime integration
  - missing features (e.g., range matching)
  - meet parity with the latest Linux kernel version

How to get started/involved?

- We’re eager to support you in using PSA-eBPF in your projects!
- PSA-eBPF documentation is a good starting point
- Feel free to open Github issues to report a bug or ask questions!
- Reach out to us on #p4-ebpf channel in the P4 Lang Slack
- To learn more about eBPF refer to eBPF/XDP guide from ebpf.io

Thank You

tomasz.osinski@intel.com / osinstom@gmail.com
mateusz.kossakowski@orange.com
jan.palimaka@orange.com