
KubeCon
CloudNativeCon
Europe 2023
TiKV

Red Hat

KubeCon | CloudNativeCon

Europe 2023

eBPF 201

Donald Hunter & Sanjeev Rampal

Emerging Technologies, Red Hat

Overview

- Goal of this talk:
  - Provide a tutorial and guidelines for best practices for eBPF development
  - Targeted at non-kernel developers, Kubernetes audience

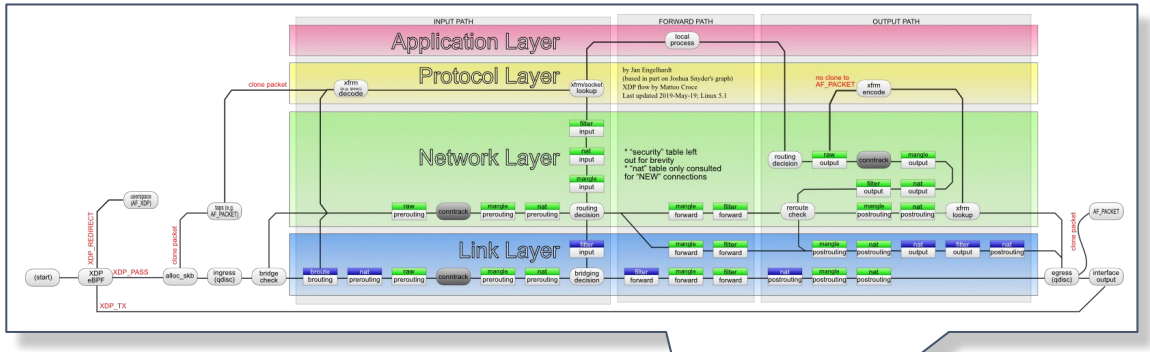
> **"I am a Kubernetes developer / operator, not a Linux kernel developer. I may need to do some eBPF development or operations, what do I need to know?"**

  - A "201 tutorial" => Focus on topics beyond "What is eBPF ?"
  - Writing maintainable, portable eBPF programs across teams
  - Based on our experiences as newbies learning eBPF development

Agenda

- Brief eBPF technology introduction
- Types of Applications that can benefit from eBPF
- Modern eBPF Technology overview
- Understanding eBPF portability, kernel data types, CO-RE
- Development recommendations & best practices
- Deployment, Operations, Kubernetes specific aspects
- Review of BPF code best practices

eBPF Technology Introduction




```
int trigger_func(int arg1, int arg2)
{
    // user space code
}
```

```
SEC("uprobe//proc/self/exe:trigger_func")
int BPF_UPROBE(handle_trigger_func, int arg1, int arg2)
{
    // BPF code
}
```

syscall API

Verifier

xdp, tc,
cgroup sockops

kprobes,
uprobes

tracepoints

BPF VM

Helpers

Maps

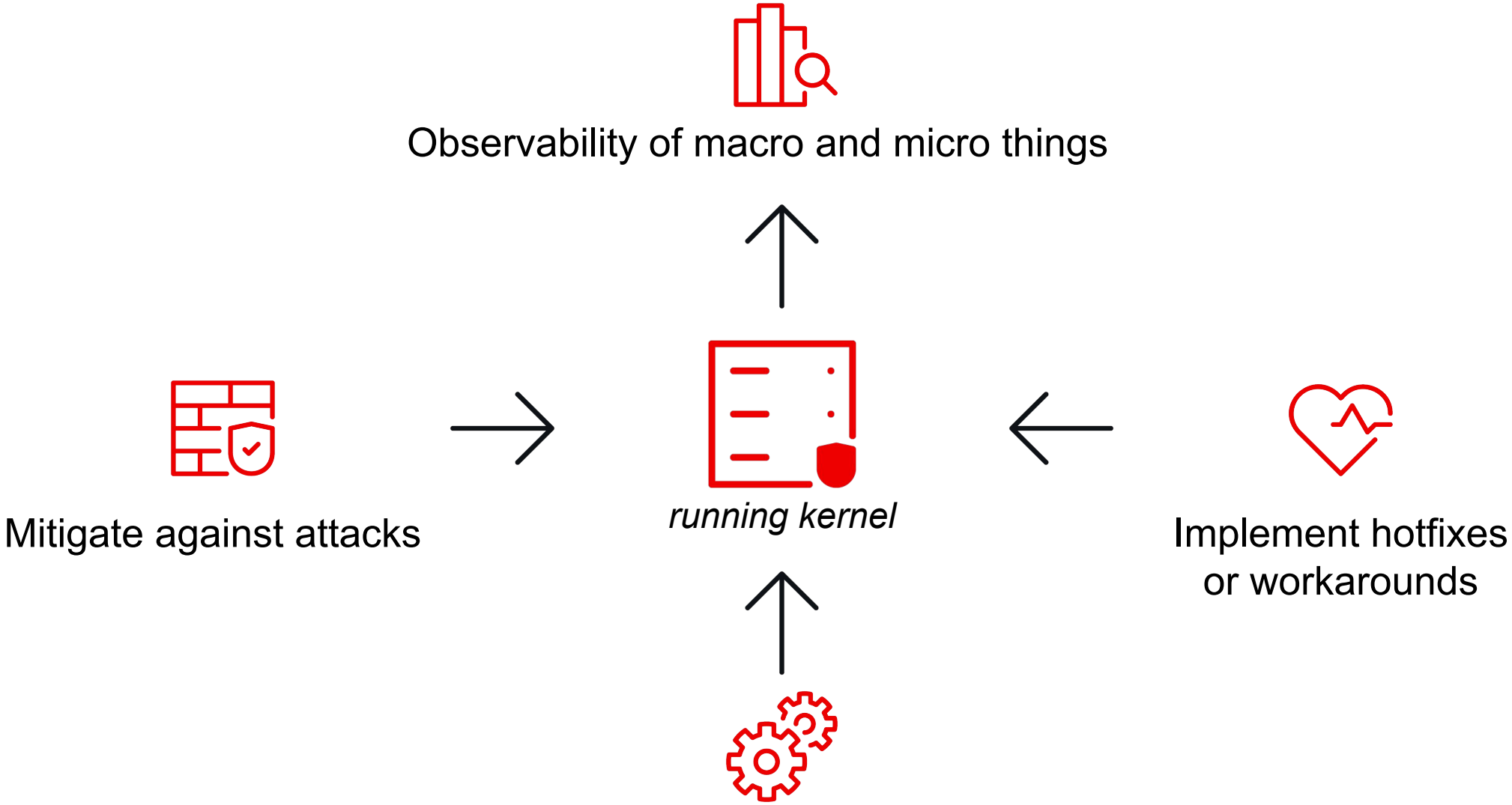
kfuncs

Linux kernel

1700+ tracepoints:

```
tcp:tcp_retransmit_skb
sock:sock_recv_length
sock:sock_send_length
sock:sk_data_ready
net:net_dev_queue
net:net_dev_xmit_timeout
net:net_dev_xmit
net:net_dev_start_xmit
skb:consume_skb
skb:kfree_skb
...
```

eBPF Applications



eBPF technology maturity

Phase 0 “Classic BPF”	Phase 1 eBPF early days	Phase 2 Modern eBPF	Future eBPF innovations
Berkeley Packet Filter	eBPF VM, base instruction set Initial set of bpf helpers Early BCC based tooling (continuing)	BTF CO-RE libbpf kfuncs (& many more tools) BPF Type Format Compile Once Run Everywhere	E.g. Dynamic (& safe) memory allocation, greater access to kernel internals, operational tooling, eBPF supply chain …

Focus of this presentation.

Recommended for new eBPF apps

Modern eBPF portability: key concepts

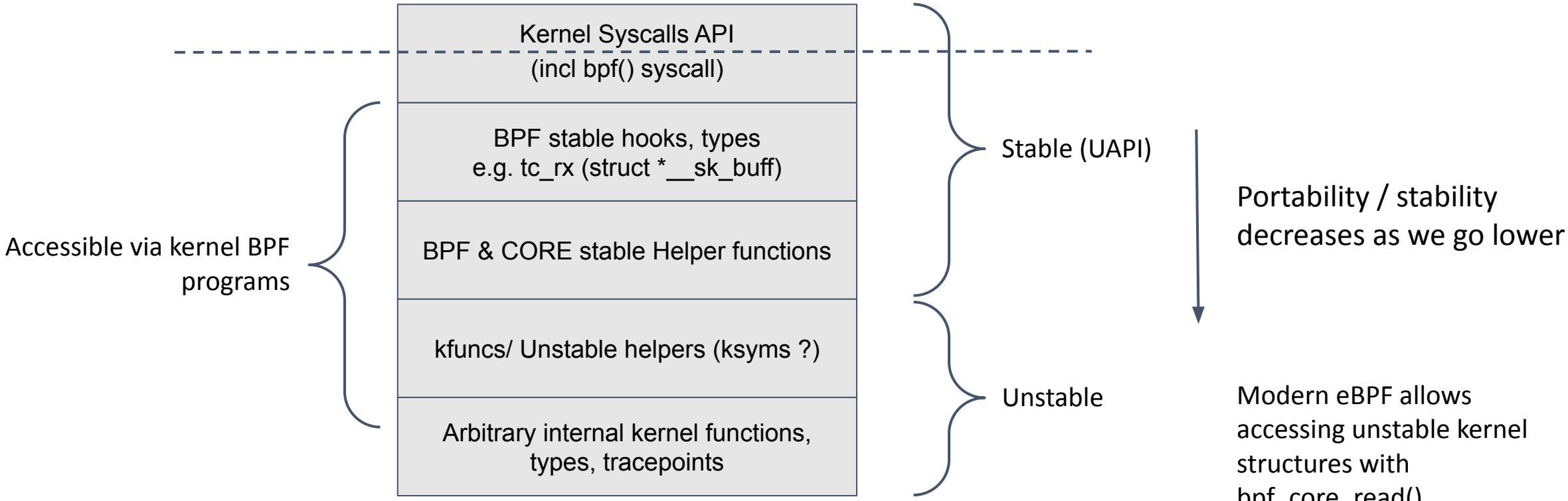
- “BPF C”
  - A version of C with restrictions as well as extensions beyond ANSI C designed for use in “kernel safe” and CO-RE portable eBPF programs
- BTF:
  - BPF Type Format
  - Efficient symbol metadata table format used for eBPF CO-RE, debugging
- CO-RE:
  - Compile Once Run Everywhere
  - Technology within Clang, libbpf to use BTF and relocate types across kernel changes
- libbpf
  - Userspace C library for manipulation of BPF programs (with CO-RE) and some kernel space utilities
  - Part of kernel source tree up to date with core BPF functions

Example: Struct sk_buff below is considered equivalent to the kernel’s full struct sk_buff and the sub-fields are dynamically relocated / aligned.

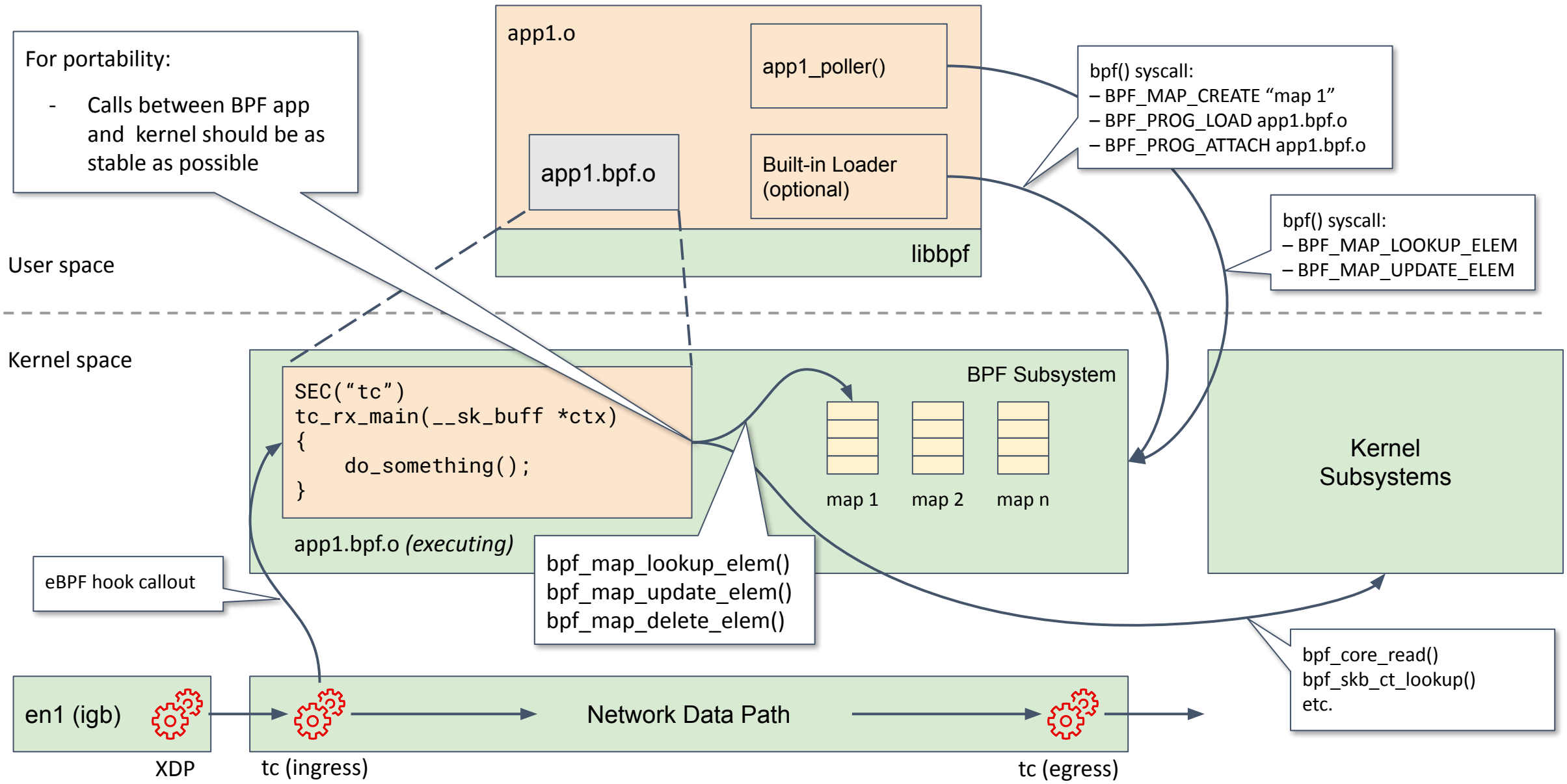
```
struct sk_buff {
     unsigned char *data;
     unsigned int len;
}
__attribute__((preserve_access_index));
```

Levels of kernel programming stability

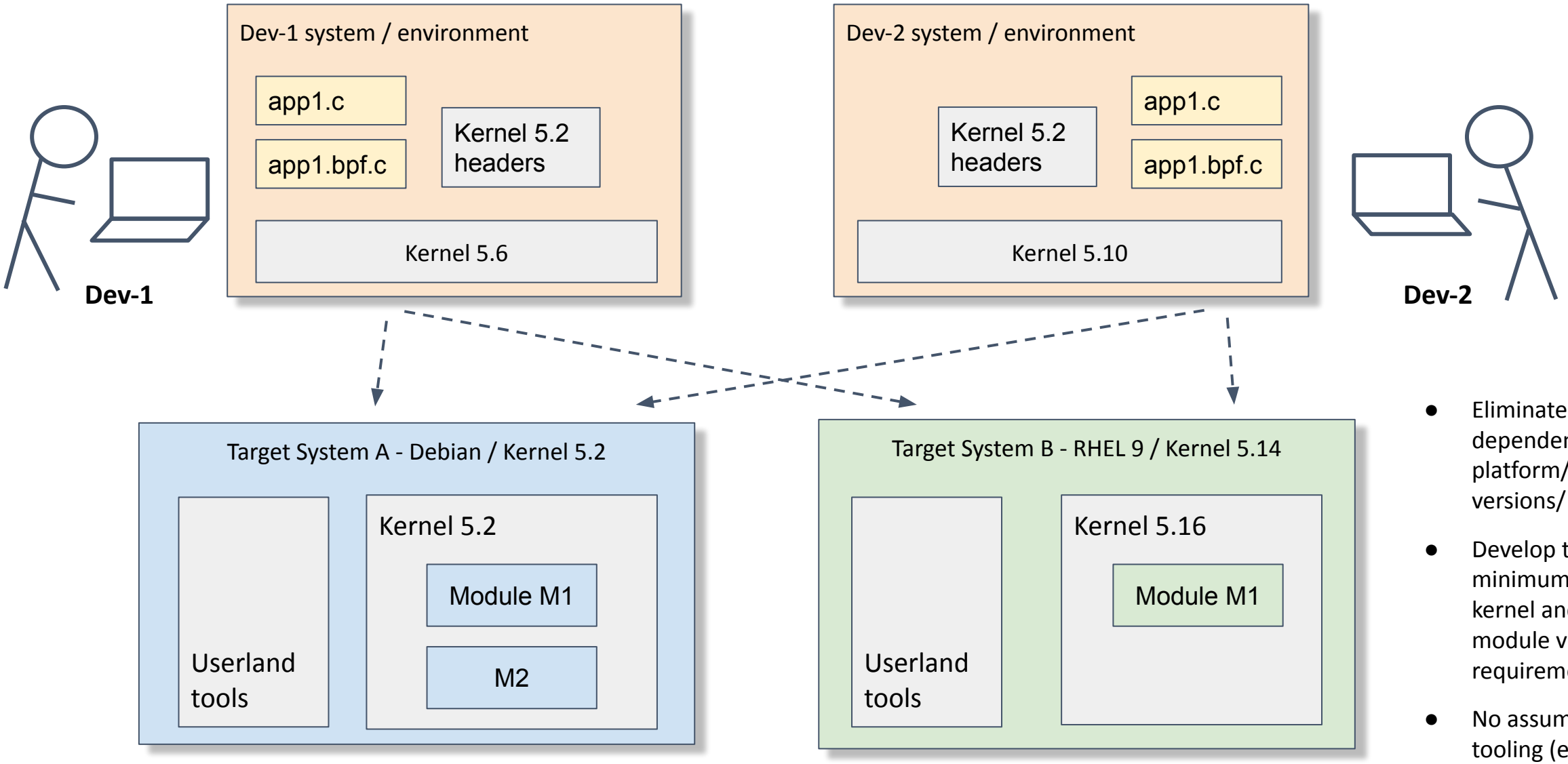
BPF loader & user space app
(with CAP_BPF, CAP_NET_ADMIN …)



Architecture of a running eBPF Program



eBPF Team Development Model



- Eliminate any dependency on dev platform/ kernel versions/ headers
- Develop towards minimum Target kernel and kernel module versions requirements
- No assumptions on tooling (eg bpftool)

Development Recommendations for BPF Projects

1: Plan for Target platform versions

Example:

RHEL 8+, Ubuntu 20.04+, Amazon Linux 2

CPU Architectures: x86 / AMD64, ARM64

From distro docs:

RHEL 8 => Kernel 4.18 RHEL 9 => 5.14

Ubuntu 20.04 => Kernel 5.4

Amazon Linux 2 => Kernels 4.14, 5.4, 5.10

Determine the minimum kernel version app needs:

Read the docs or test target platforms

E.g. cgroup/connect4 programs in 4.17+

BPF function calls in 4.16+

Either: write app to use eBPF functions only available in kernel 4.14

Or: probe and exit for kernels older than 4.18

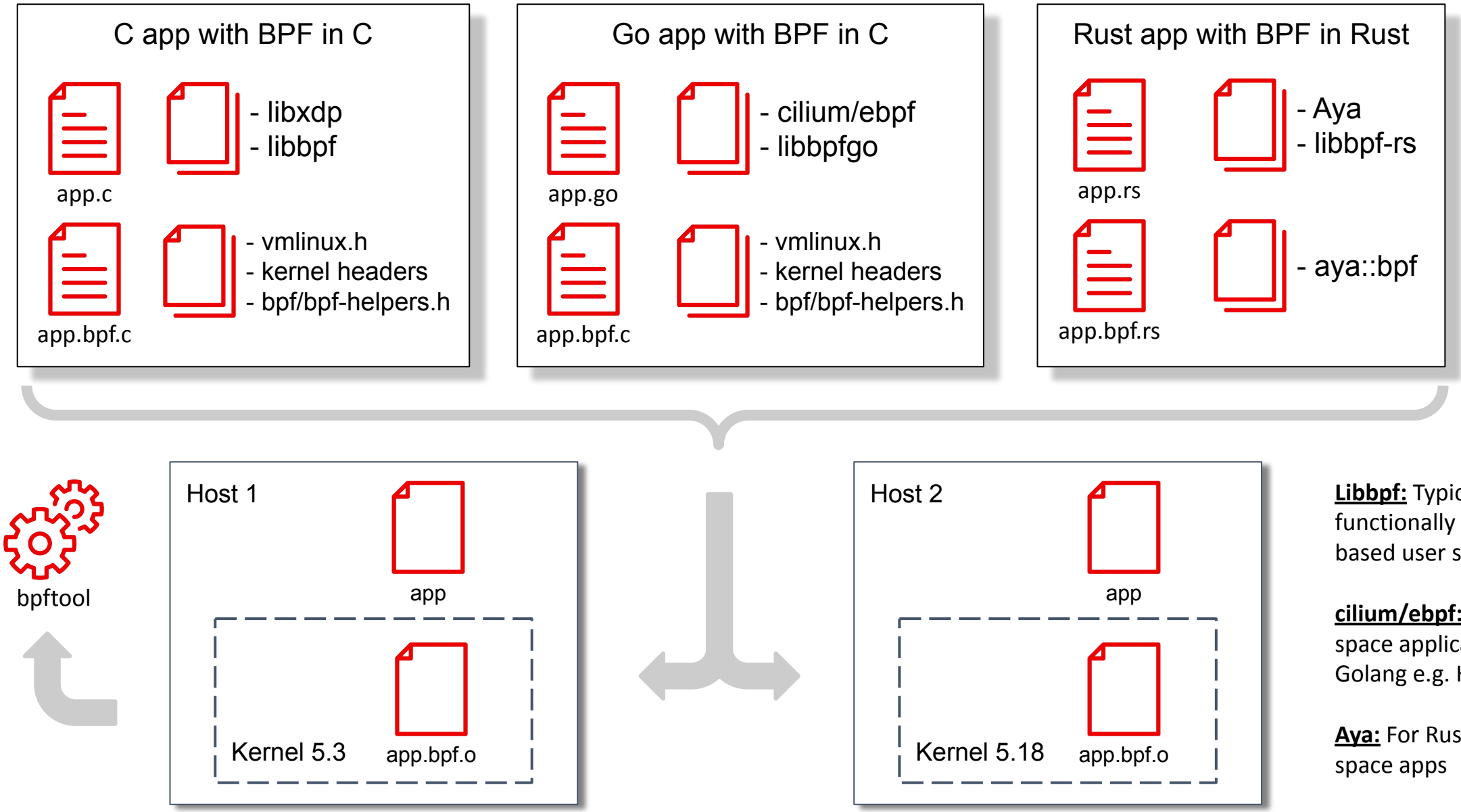
LRU hash	4.10	BPF_MAP_TYPE_LRU_HASH
LPM trie	4.11	BPF_MAP_TYPE_LPM_TRIE
Ring buffer	5.8	BPF_MAP_TYPE_RINGBUF
Bloom filter	5.16	BPF_MAP_TYPE_BLOOM_FILTER

BPF function calls	4.16
cgroup/connect4	4.17
cgroup/connect6	4.17
BPF bounded loops	5.3

bpf_dynptr_read()	5.19
bpf_dynptr_write()	5.19
bpf_fib_lookup()	4.18
bpf_timer_start()	5.15
bpf_timer_cancel()	5.15

https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

2: Pick your programming stack



Libbpf: Typically, most functionally complete. C based user space apps.

cilium/ebpf: When user space application is in Golang e.g. K8s

Aya: For Rust based user space apps

3: Use portable kernel type definitions

Know your kernel API surface

- Define just what you need, not all of vmlinux.h:

```
struct sk_buff {
    unsigned char *data;
    unsigned int len;
} __attribute__((preserve_access_index));
```

- Start with vmlinux.h from oldest target version (eg 4.18), then prune
- Easier to review, and to maintain in future
- Do not use kernel headers package from dev system

```
struct sk_buff {
        union {
                struct {
                        struct sk_buff
                        struct sk_buff
                        union {
                                struct r
                                long uns
                        };
                };
                struct rb_node rbnode;
                struct list_head list;
                struct llist_node ll_nc
        };
        union {
                struct sock *sk;
                int ip_defrag_offset;
        };
        union {
                ktime_t tstamp;
                u64 skb_mstamp_ns;
        };
        char cb[48];
        union {
                struct {
                        long unsigned i
                        void (*destruct
                };
                struct list_head tcp_ts
                long unsigned int _sk_r
        };
        long unsigned int _nfct;
        unsigned int len;
        unsigned int data_len;
        __u16 mac_len;
        __u16 hdr_len;
        __u16 queue_mapping;
...
```

```
...
                                u32 vlan_all;
                                struct {
                                        __be16 vlan_proto;
                                        __u16 vlan_tci;
                                };
                        };
                        union {
                                unsigned int napi_id;
                                unsigned int sender_cpu;
                        };
                        u16 alloc_cpu;
                        __u32 secmark;
                        union {
                                __u32 mark;
                                __u32 reserved_tailroom;
                        };
                        union {
                                __be16 inner_protocol;
                                __u8 inner_ipproto;
                        };
                        __u16 inner_transport_header;
                        __u16 inner_network_header;
                        __u16 inner_mac_header;
                        __be16 protocol;
                        __u16 transport_header;
                        __u16 network_header;
                        __u16 mac_header;
                } headers;
        };
        sk_buff_data_t tail;
        sk_buff_data_t end;
        unsigned char *head;
        unsigned char *data;
        unsigned int truesize;
        refcount_t users;
        struct skb_ext *extensions;
};
```

4: Use Kernel version & config probing

Check execution kernel version against planned supported target versions:

```
#include <bpf/bpf_helpers.h>

extern int LINUX_KERNEL_VERSION __kconfig;

int probe_kernel() {

        if (LINUX_KERNEL_VERSION > KERNEL_VERSION(4, 18, 0)) {
                /* we are on a supported kernel version */
        } else {
                /* log error and exit gracefully */
        }

        ...

}
```

Probe for individual kernel features or struct definitions:

```
extern bool CONFIG_LWTUNNEL_BPF __kconfig __weak;

if (CONFIG_LWTUNNEL_BPF) {
        /* configure lwtunnel from BPF */
}

if (bpf_core_type_exists(struct bpf_ringbuf)) {
        /* use ringbuf instead of perf buffers */
}
```

https://nakryiko.com/posts/bpf-core-reference-guide/

5: Use new eBPF program mgmt infra

Some **operational challenges** with eBPF today

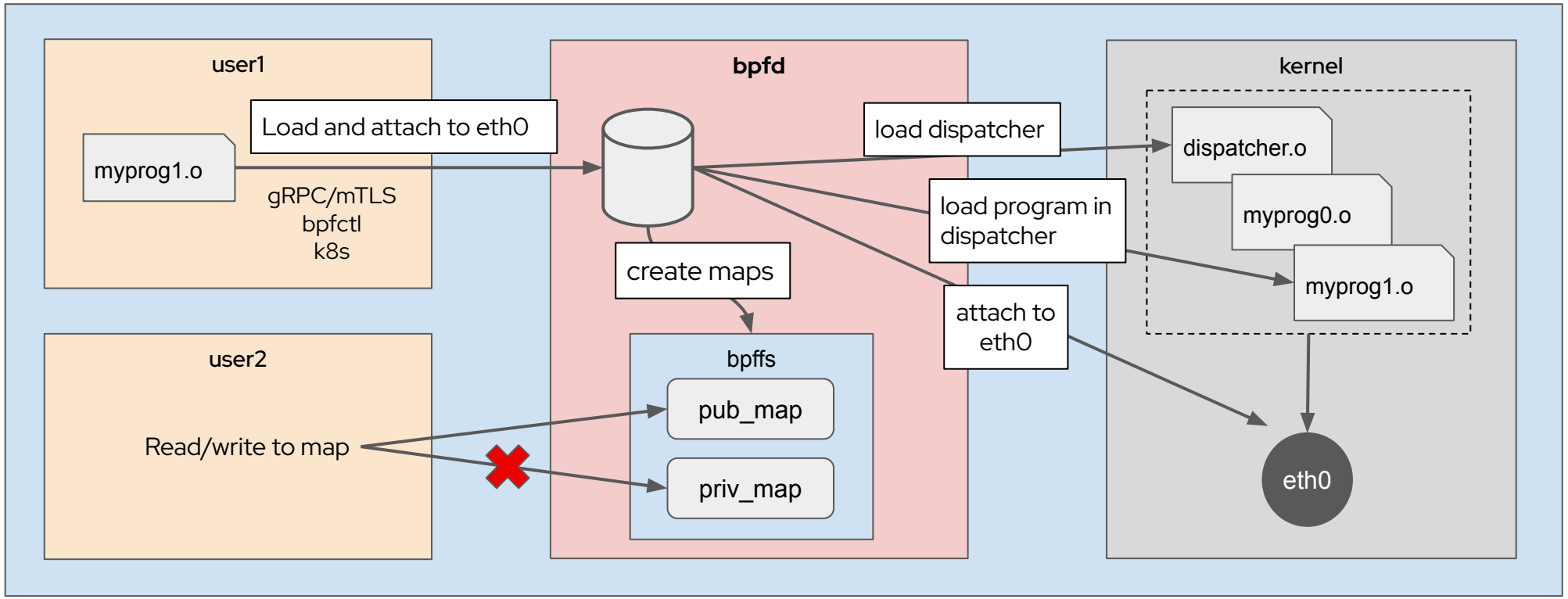
- Security
  - BPF-enabled apps require at least CAP_BPF permissions, maybe more
  - E.g. Net-Observ: CAPS: BPF, PERFMON, NET_ADMIN, SYS_RESOURCE
- Handling multiple BPF programs
  - Sharing the same hook points
  - Programs can Interfere with each other
- Preventing and Debugging BPF-related problems
- Duplicated code and functionality needed to load and manage BPF programs

Solution: Consider new bpf tools

- New tool: **bpfd** https://github.com/redhat-et/bpfd
- Note: Also, leverage existing tools such as bpftool (for use cases other than above)

What is bpfd ?

- A system daemon for managing BPF programs
- Improved security, multi-program management:
  - Instead of every app needing CAP_BPF, CAP_NETADMIN etc, they can have lowered privileges & interact with the kernel via bpfd
- Includes a K8s operator and APIs to support managing BPF programs on K8s/ Openshift.



bpfd: K8s operational model

Step1:

Build a BPF program, compile it to BPF Bytecode & package in OCI image

```
podman build \
    --build-arg PROGRAM_NAME=go-xdp-counter \
    --build-arg SECTION_NAME=stats \
    --build-arg PROGRAM_TYPE=xdp \
    --build-arg BYTECODE_FILENAME=bpf_bpfel.o \
    --build-arg KERNEL_COMPILE_VER=$(uname -r) \
    -f ../../packaging/container-deployment/Containerfile.bytecode \
    -t quay.io/$USER/go-xdp-counter-bytecode:latest .
```

Step 2:

Create a BpfProgramConfig object to describe the program attributes, such as attachpoint, priority nodeselector, and bpf-bytecode image tag.

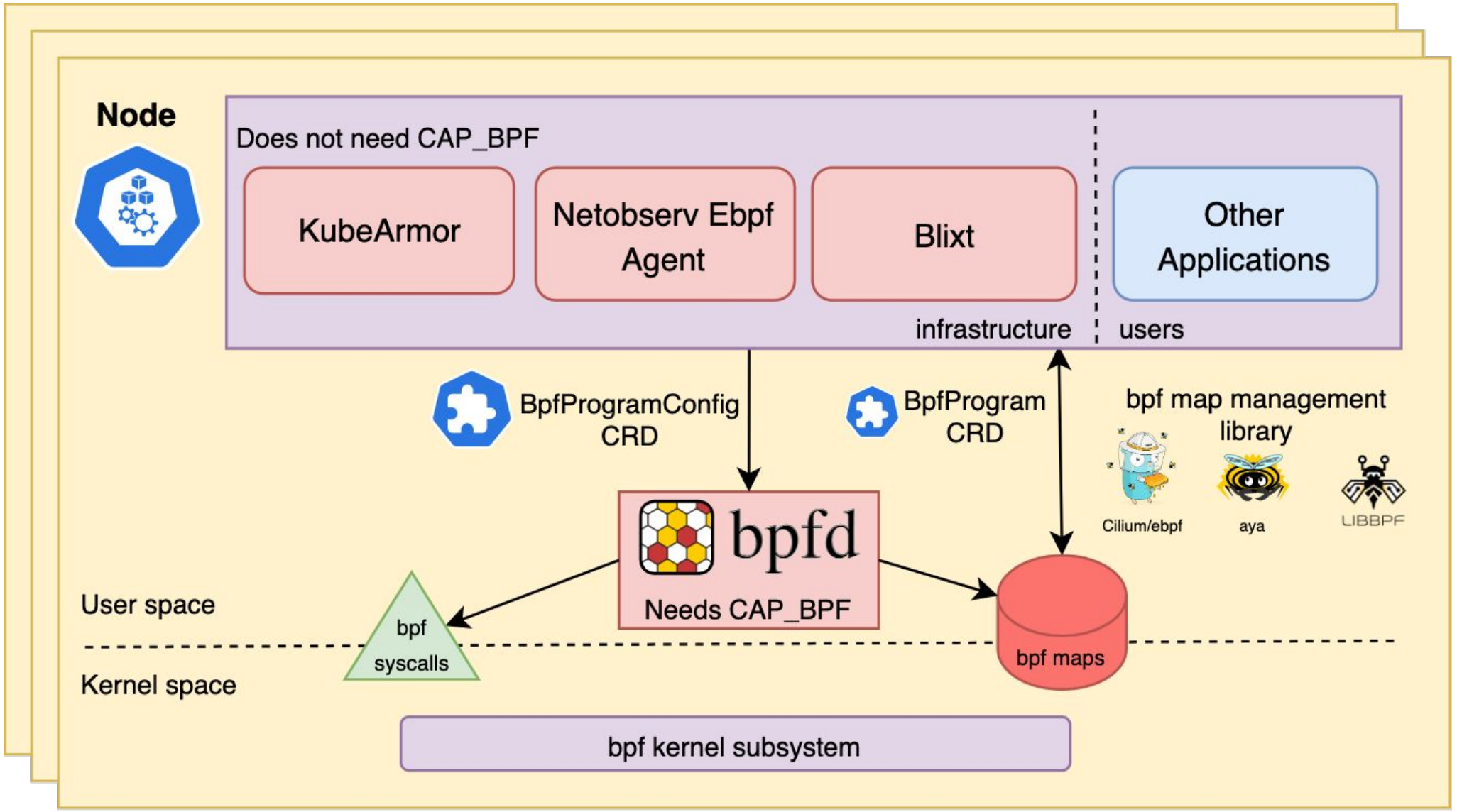
```
apiVersion: bpfd.io/v1alpha1
kind: BpfProgramConfig
metadata:
  labels:
    app.kubernetes.io/name: bpfprogramconfig
  name: go-xdp-counter-example
spec:
  ## Must correspond to image section name
  name: stats
  type: XDP
  # Select all nodes
  nodeselector: {}
  attachpoint:
    networkmultiattach:
      interfaceselector:
        primarynodeinterface: true
      priority: 55
  bytecode: image://quay.io/bpfd-bytecode/go-xdp-counter:latest
```

Step 3:

Write an application which works with the maps defined by your BPF programs

Interact with the maps via the map pin-points stored in each node's bpfProgram

bpfd: Deploying BPF in Kubernetes



BPF Code Best Practises

https://github.com/donaldh/bpf-playground

Ring buffer for sending events to user space

```
struct {
        __uint(type, BPF_MAP_TYPE_RINGBUF);
        __uint(max_entries, 4096);
} dns_events SEC(".maps");
```

Struct definition shared between BPF and user space code.

Types must be compatible with user space `#include` files, e.g. `__u32`, `__u64`

```
struct dns_event {
        __u64 duration;
        char ifname[IFNAMSIZ];
        __u32 srcip;
        __u32 dstip;
        __u16 length;
        unsigned char payload[MAXMSG];
        __u16 id;
        __u16 flags;
};
```

Interface with user space

Interface with kernel

CO-RE enabled struct definitions.

Contains subset of kernel definitions with just the required members.

```
struct sk_buff {
        unsigned char *data;
        unsigned int len;
} __attribute__((preserve_access_index));

struct trace_event_raw_net_dev_template {
        struct sk_buff *skbaddr;
} __attribute__((preserve_access_index));
```

```
SEC("tracepoint/net/net_dev_queue")
int trace_net_packets(struct trace_event_raw_net_dev_template *ctx) {
        unsigned char *data = BPF_CORE_READ(ctx, skbaddr, data);
        unsigned int len = BPF_CORE_READ(ctx, skbaddr, len);

        do_trace(data, len);

        return BPF_OK;
}
```

CO-RE macros used to access structure values

Strongly typed maps thanks to CO-RE

```
struct {
        __uint(type, BPF_MAP_TYPE_HASH);
        __uint(max_entries, 1024);
        __type(key, struct request_key);
        __type(value, struct request_val);
} requests SEC(".maps");
```

Documentation Landscape

eBPF Foundation: eBPF Documentation

Kernel docs: BPF Documentation, BPF Maps, Program Types, BPF kfuncs

Man pages: bpf-helpers(7), bpftool(8)

Library Docs: libbpf, libxdp, Aya, Cilium ebpf

'Reference' Blogs: BPF CO-RE Reference Guide

Examples: libbpf-bootstrap, Practical BPF Examples

There is *still* a lot of tribal knowledge.

Red Hat

KubeCon | CloudNativeCon

Europe 2023

Thank You

KubeCon CloudNativeCon Europe 2023

KubeCon CloudNativeCon North America 2021

Please scan the QR Code above
to leave feedback on this session