CLOUDFLARE

Marking Packets With Rich Metadata

Arthur Fabre
Jakub Sitnicki

Linux Plumbers Conference 2024

Metadata today

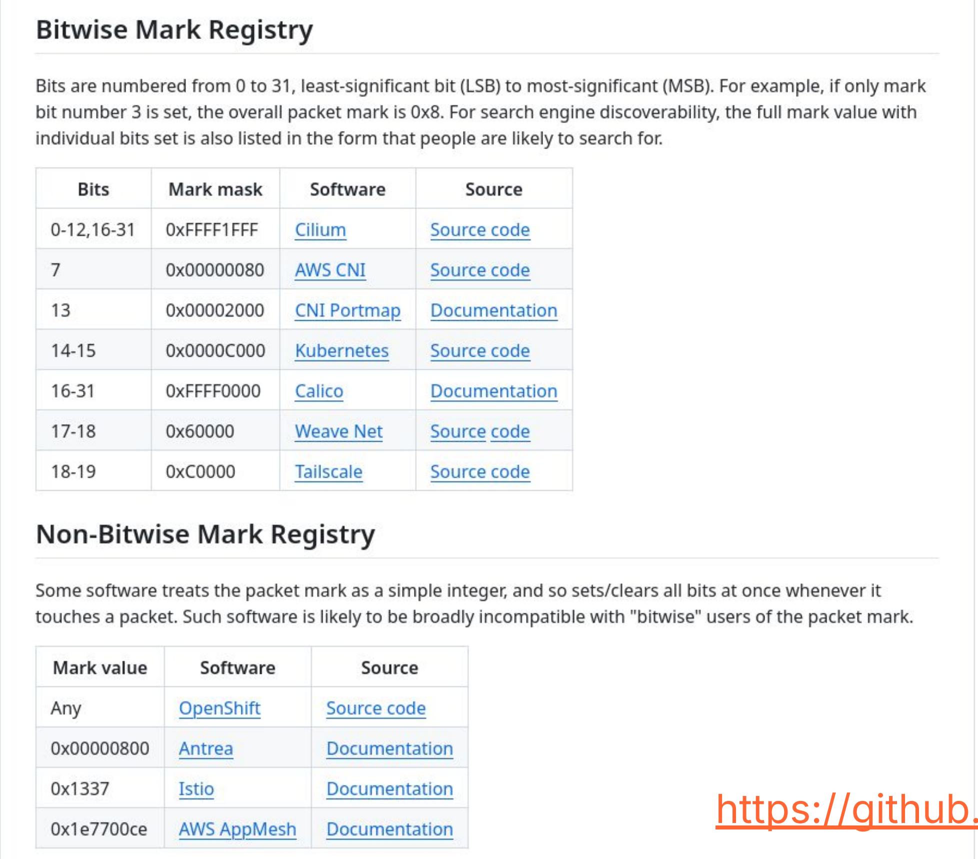
Metadata today

- `sk_buff->mark`
- Throughout network stack:
    - BPF
    - `fw_mark`
    - `ct_mark`
    - `xfrm_mark`
    - `SO_MARK`

What is the mark?

- Nothing
- Everything
- 32 bit cocktail:
  - routing
  - firewalling
  - nating
  - en/decryption
- LPC 2020: Packet mark in the Cloud Native world

Mark everything



Bitwise Mark Registry

Bits are numbered from 0 to 31, least-significant bit (LSB) to most-significant (MSB). For example, if only mark bit number 3 is set, the overall packet mark is 0x8. For search engine discoverability, the full mark value with individual bits set is also listed in the form that people are likely to search for.

Bits	Mark mask	Software	Source
0-12,16-31	0xFFFF1FFF	Cilium	Source code
7	0x00000080	AWS CNI	Source code
13	0x00002000	CNI Portmap	Documentation
14-15	0x0000C000	Kubernetes	Source code
16-31	0xFFFF0000	Calico	Documentation
17-18	0x60000	Weave Net	Source code
18-19	0xC0000	Tailscale	Source code

Non-Bitwise Mark Registry

Some software treats the packet mark as a simple integer, and so sets/clears all bits at once whenever it touches a packet. Such software is likely to be broadly incompatible with "bitwise" users of the packet mark.

Mark value	Software	Source
Any	OpenShift	Source code
0x00000800	Antrea	Documentation
0x1337	Istio	Documentation
0x1e7700ce	AWS AppMesh	Documentation

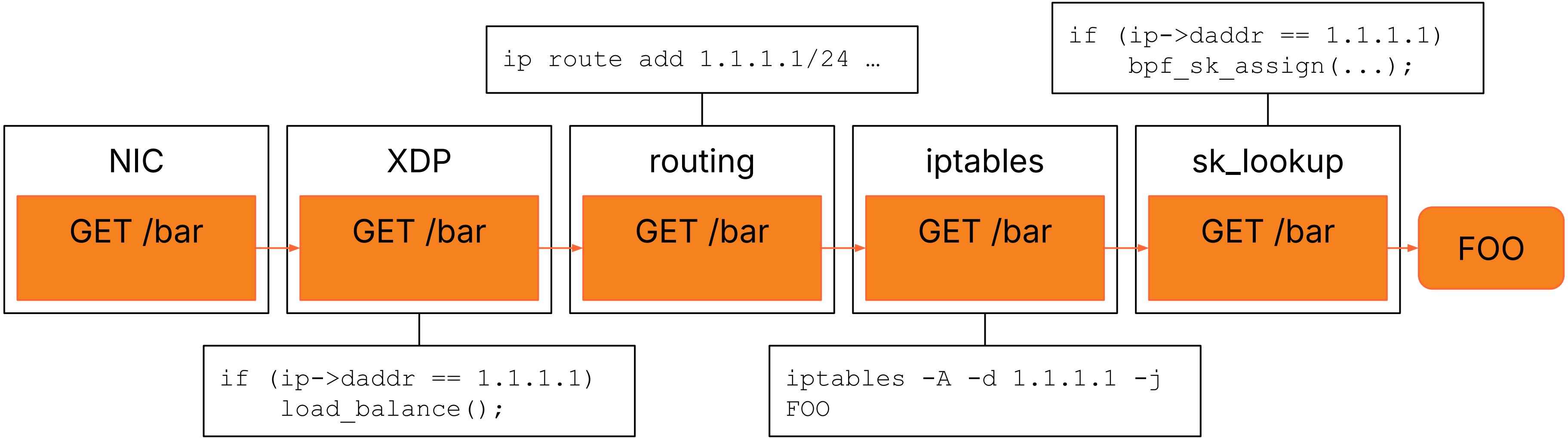
https://github.com/fwmark/registry

More!

- How many bits can I use?
- Which ones?
- Will it interfere with other services?
- We shouldn’t need a registry.

Big MetaDreams

Dispatch to services



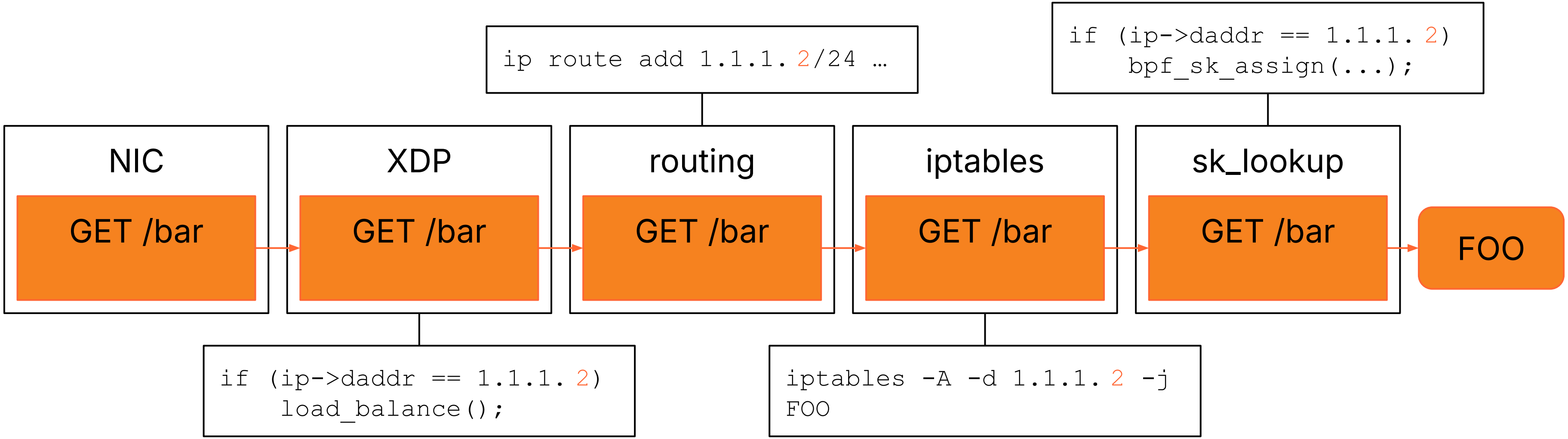
```
ip route add 1.1.1.1/24 …
```

```
if (ip->daddr == 1.1.1.1)
    bpf_sk_assign(...);
```

```
if (ip->daddr == 1.1.1.1)
    load_balance();
```

```
iptables -A -d 1.1.1.1 -j
FOO
```

Dispatch to services



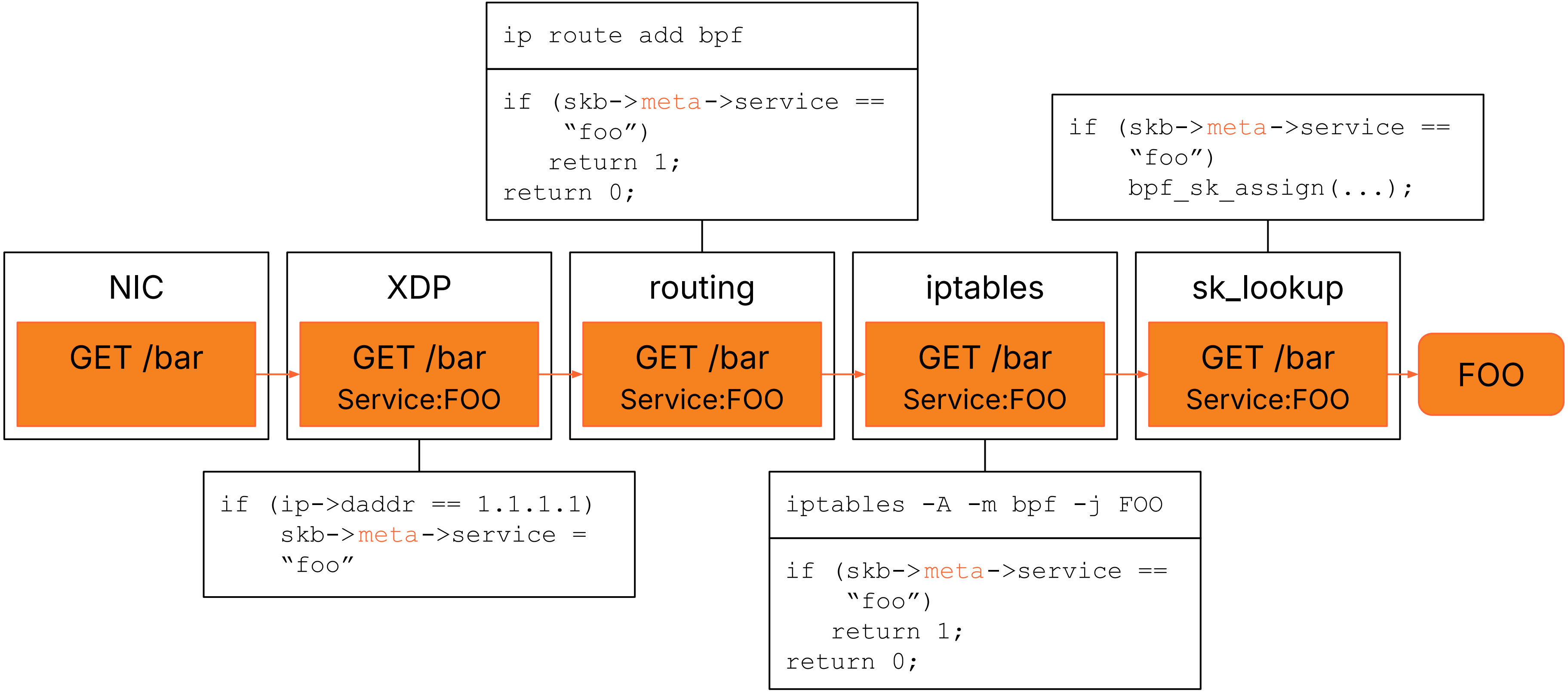
```
ip route add 1.1.1.2/24 …
```

```
if (ip->daddr == 1.1.1.2)
    bpf_sk_assign(...);
```

```
if (ip->daddr == 1.1.1.2)
    load_balance();
```

```
iptables -A -d 1.1.1.2 -j
FOO
```

Dispatch to services



```
ip route add bpf
```

```
if (skb->meta->service ==
     "foo")
   return 1;
return 0;
```

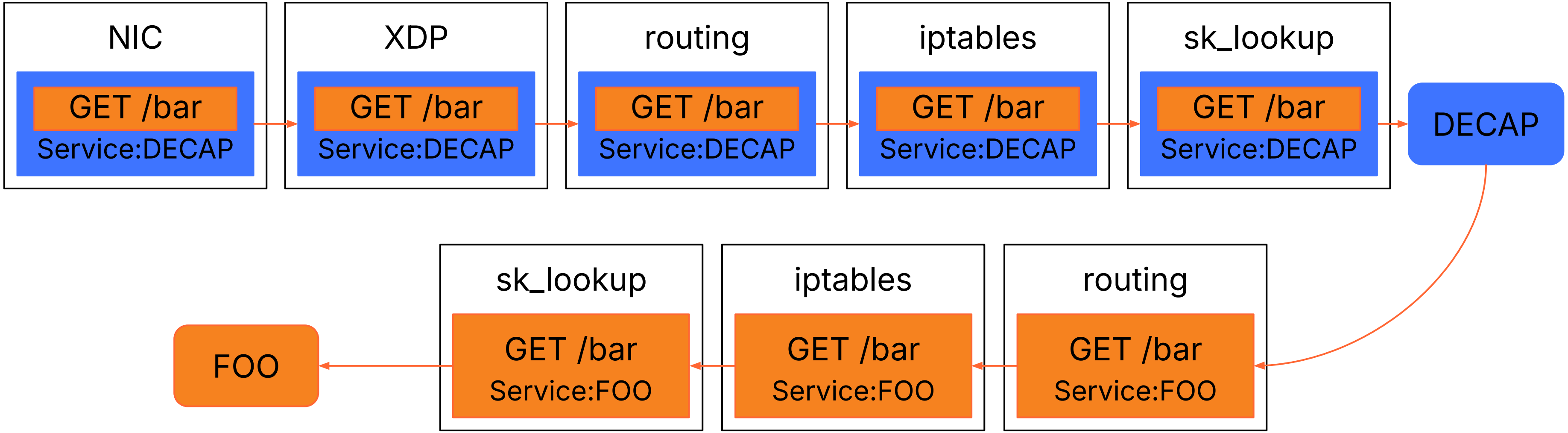
```
if (skb->meta->service ==
     "foo")
     bpf_sk_assign(...);
```

```
if (ip->daddr == 1.1.1.1)
     skb->meta->service =
     "foo"
```

```
iptables -A -m bpf -j FOO
```

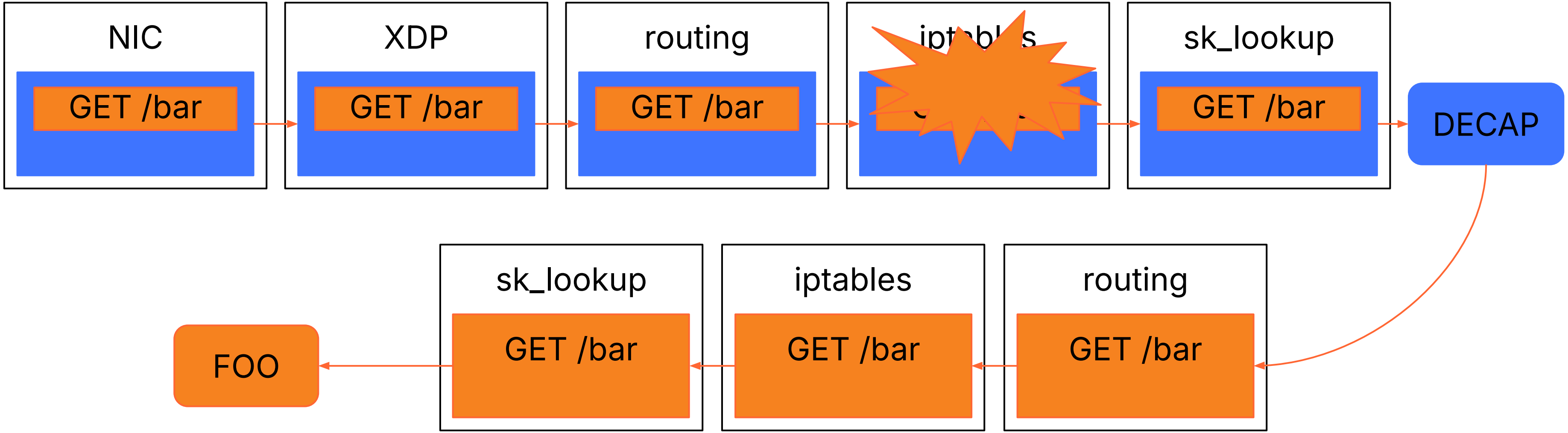
```
if (skb->meta->service ==
     "foo")
   return 1;
return 0;
```

Dispatch to services

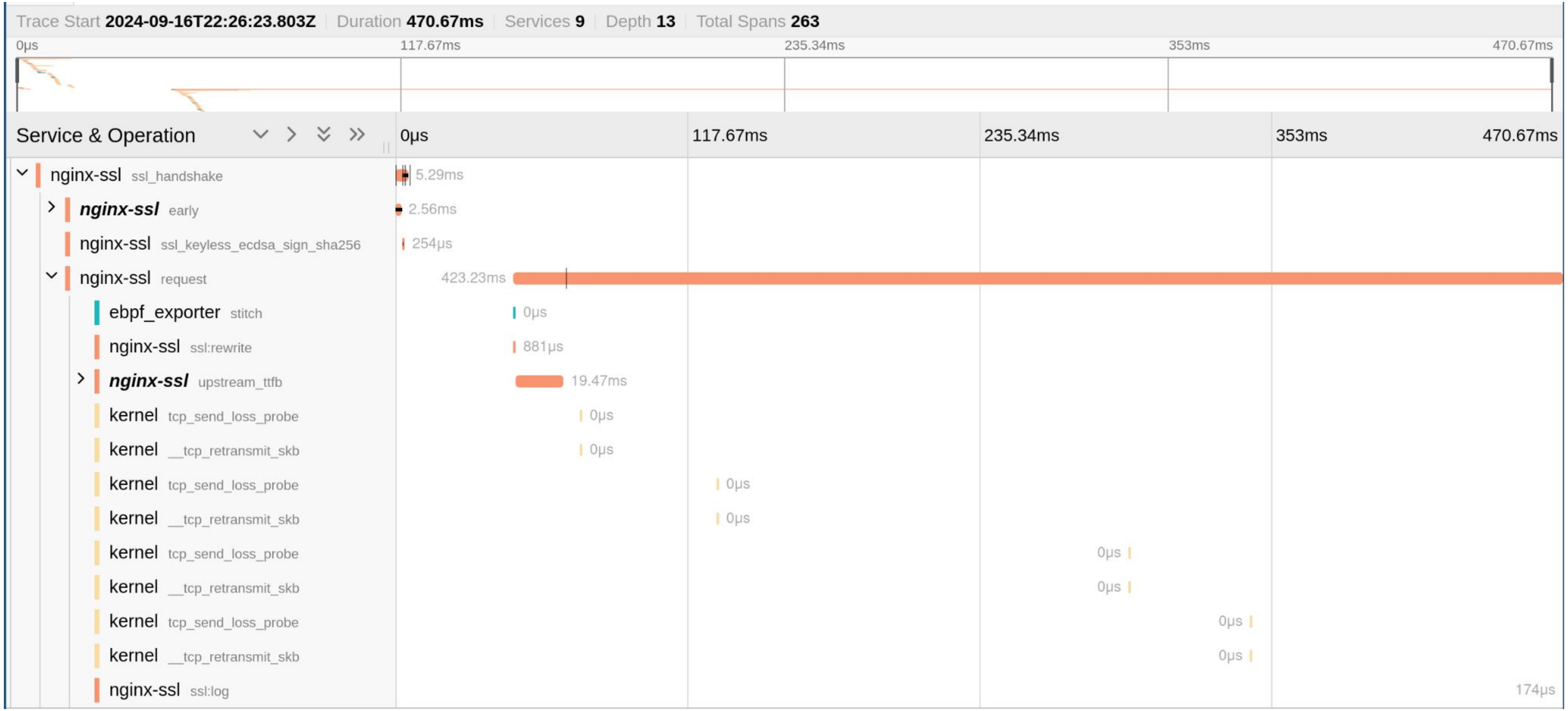


Packet tracing

- Packet drops
- Performance problems



Packet tracing



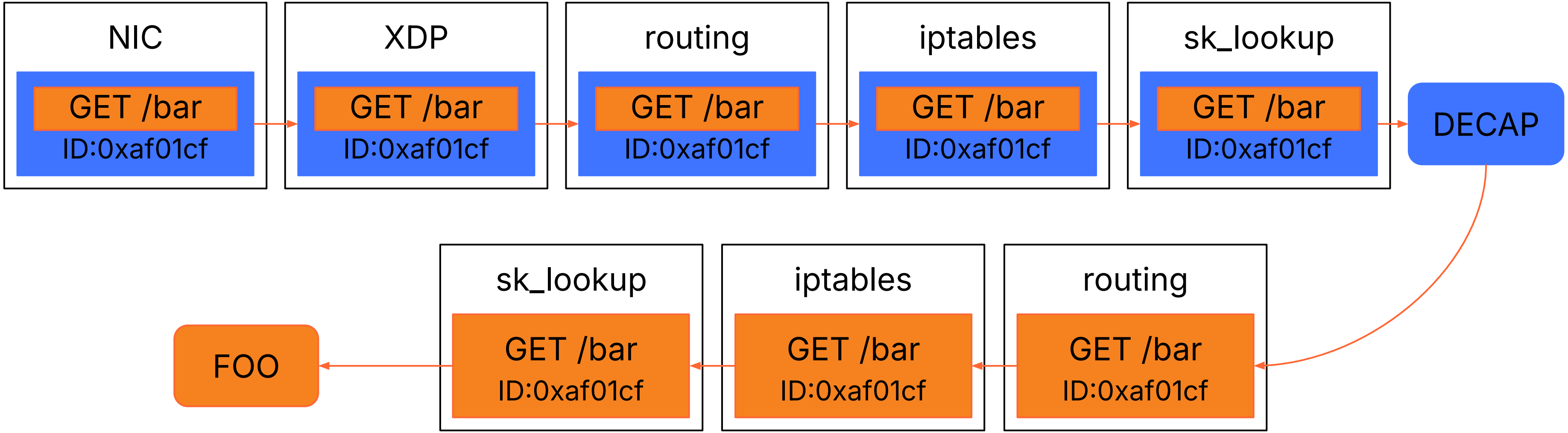
Trace Start **2024-09-16T22:26:23.803Z** | Duration **470.67ms** | Services **9** | Depth **13** | Total Spans **263**

Service & Operation	0μs · 117.67ms · 235.34ms · 353ms · 470.67ms
nginx-ssl ssl_handshake	5.29ms
nginx-ssl early	2.56ms
nginx-ssl ssl_keyless_ecdsa_sign_sha256	254μs
nginx-ssl request	423.23ms
ebpf_exporter stitch	0μs
nginx-ssl ssl:rewrite	881μs
nginx-ssl upstream_ttfb	19.47ms
kernel tcp_send_loss_probe	0μs
kernel __tcp_retransmit_skb	0μs
kernel tcp_send_loss_probe	0μs
kernel __tcp_retransmit_skb	0μs
kernel tcp_send_loss_probe	0μs
kernel __tcp_retransmit_skb	0μs
kernel tcp_send_loss_probe	0μs
kernel __tcp_retransmit_skb	0μs
nginx-ssl ssl:log	174μs

Packet tracing

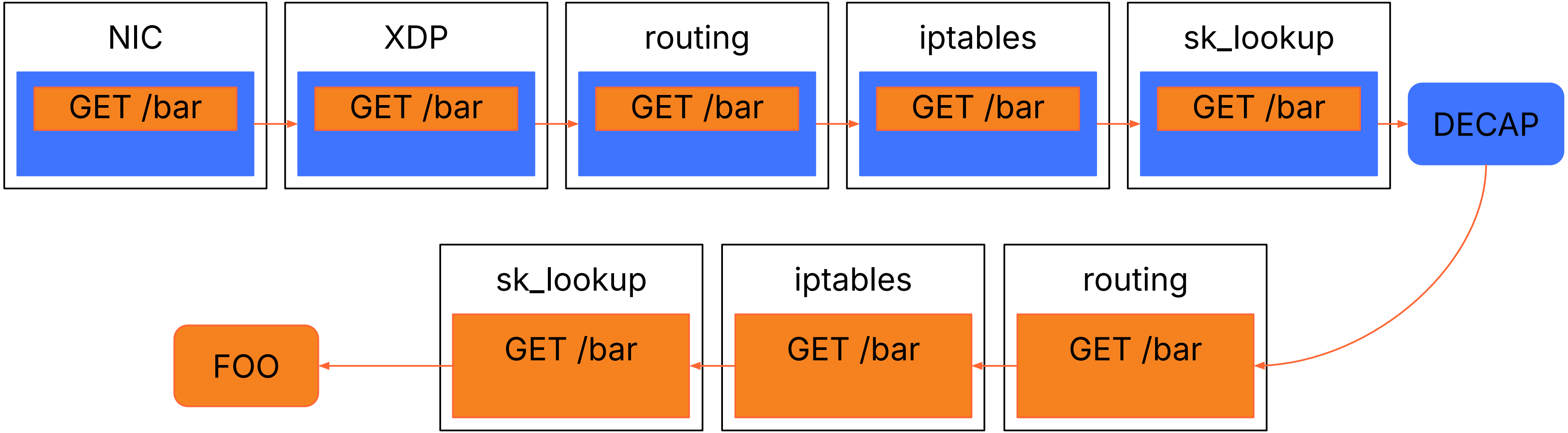
- 5-tuple
  - Doesn’t work through encap / decap
  - Doesn’t work across userspace
- `sk_buff`
  - https://github.com/cilium/pwru
  - Doesn’t work across `skb_clone()`
  - Doesn’t work across network namespaces

Packet identification



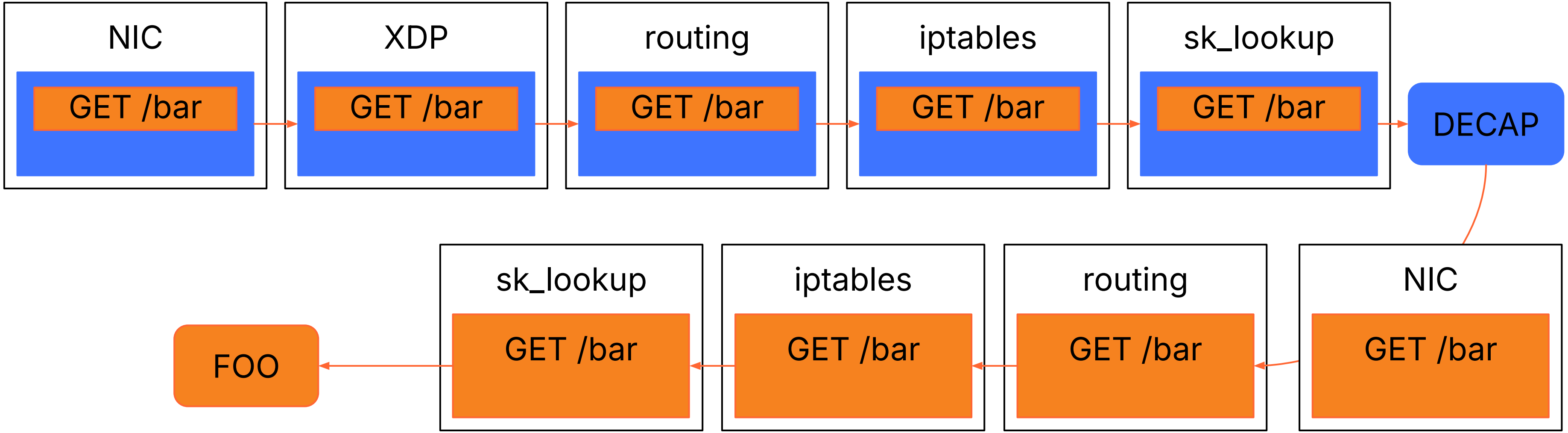
Network metadata

- Internet?
- Tunneled?



Network metadata

- Internet?
- Tunneled?



Hardware metadata

- Receive timestamp
- RSS hash
- VLAN tag

Rich SKB metadata

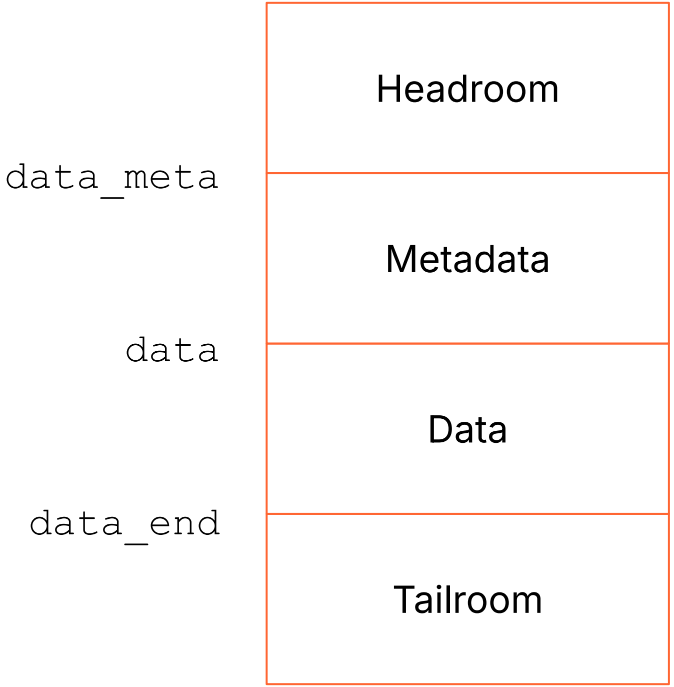
Requirements

- No allocations
  - No `struct skb_ext`
- No growing `sk_buff`
- Persistent
  - No `sk_buff->cb`

XDP metadata

```
struct xdp_md {
      __u32 data;
      __u32 data_end;
      __u32 data_meta;
}
```

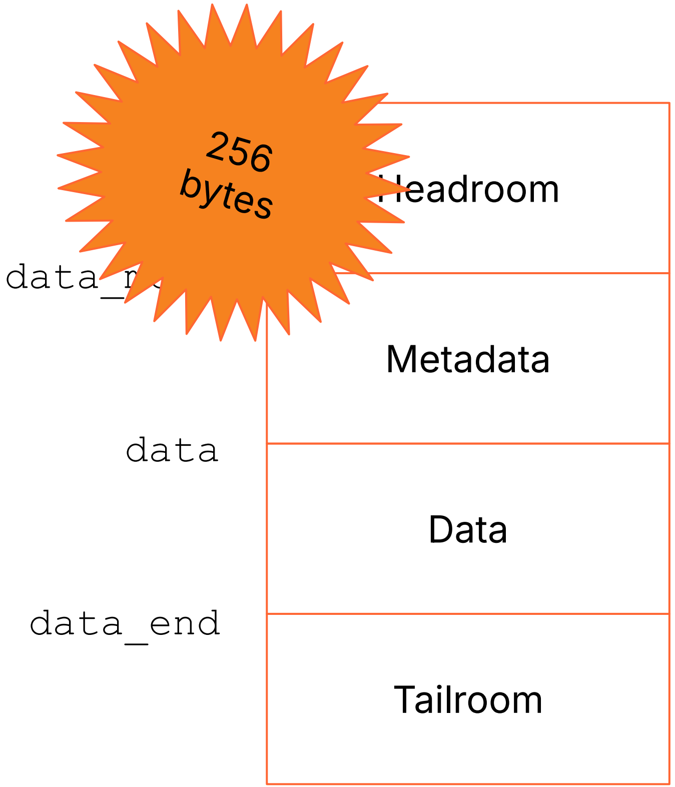
```
bpf_xdp_adjust_meta()
```



XDP metadata

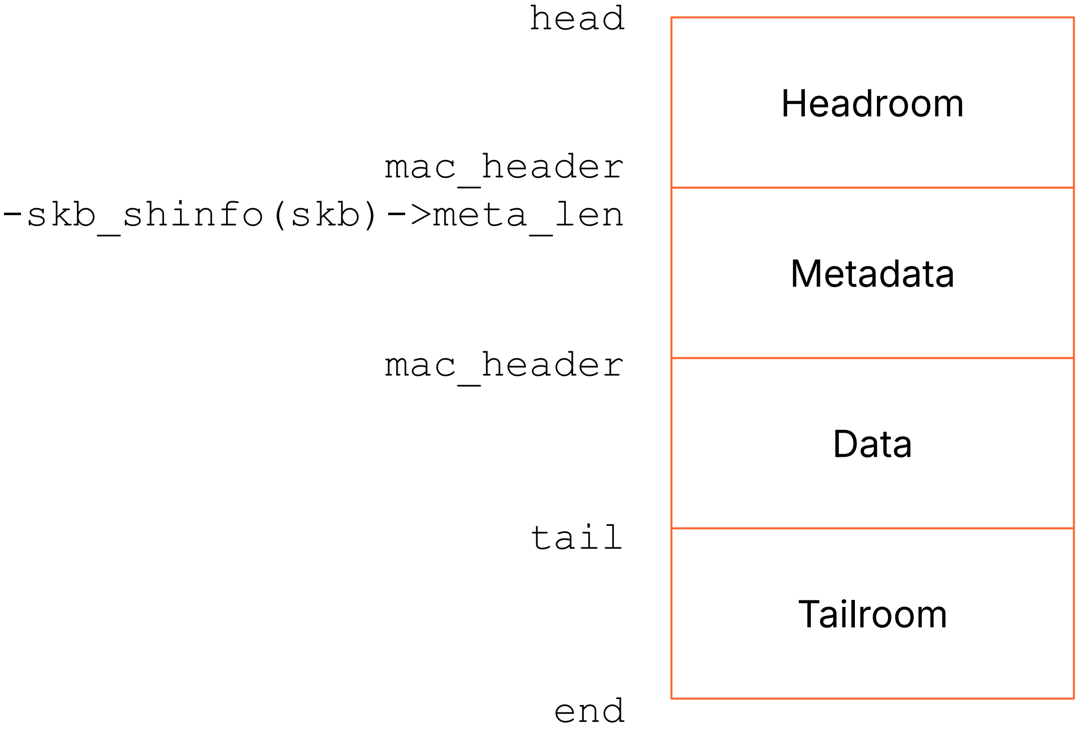
```
struct xdp_md {
	__u32 data;
	__u32 data_end;
	__u32 data_meta;
}
```

```
bpf_xdp_adjust_meta()
```



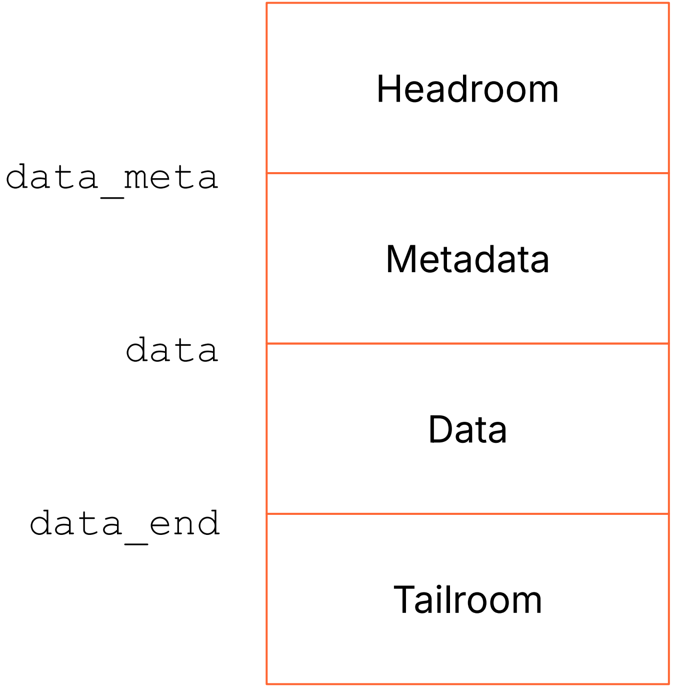
Kernel sk_buff metadata

```
struct sk_buff {
      unsigned char  *head;
     __u16           mac_header;
      sk_buff_data_t tail;
      sk_buff_data_t end;
}
```



TC __sk_buff metadata

```
struct __sk_buff {
      __u32 data;
      __u32 data_end;
      __u32 data_meta;
}
```



Beyond TC: Socket filters

- Direct access to `data` and `data_meta`
  - `CAP_PERMON` & `CAP_BPF`
- Fields already exist

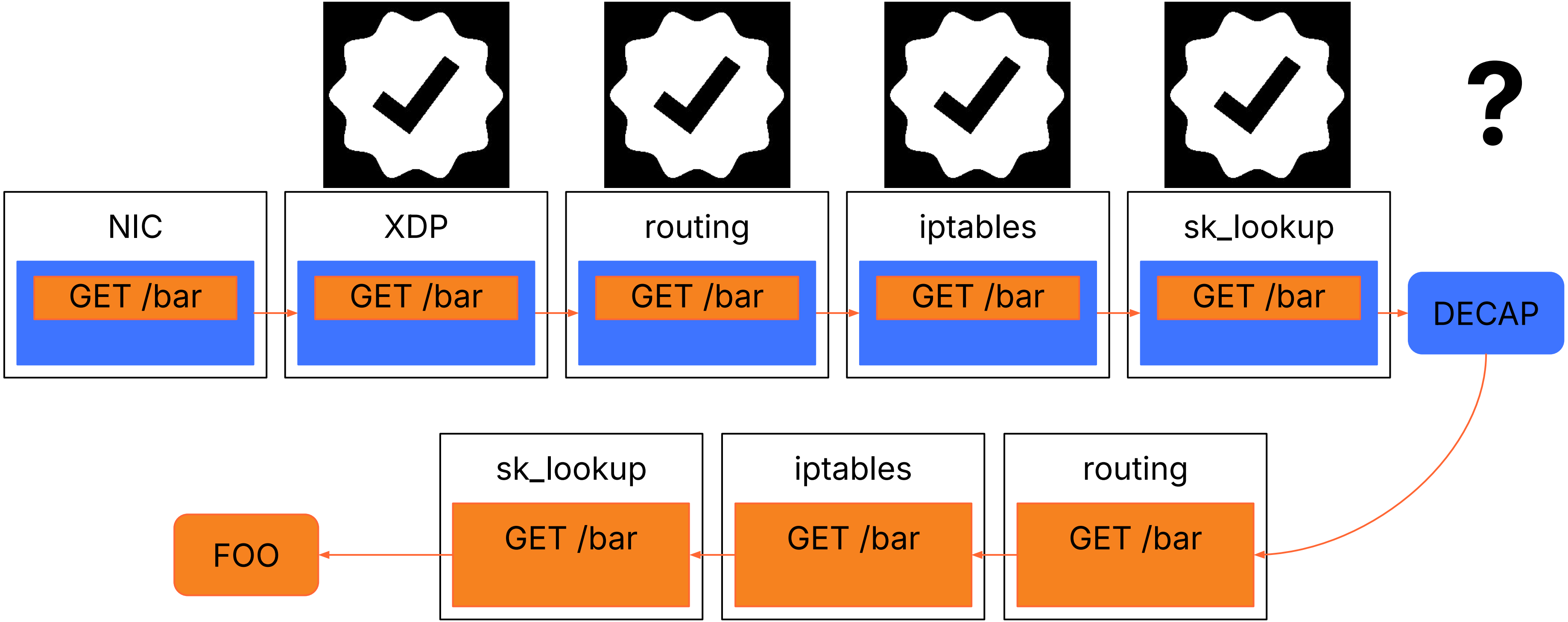
Beyond TC: sk_lookup

- No `__sk_buff`
- Mirror socket filter API
- Add direct access to:
  - `data_meta`
  - `data_meta_end`

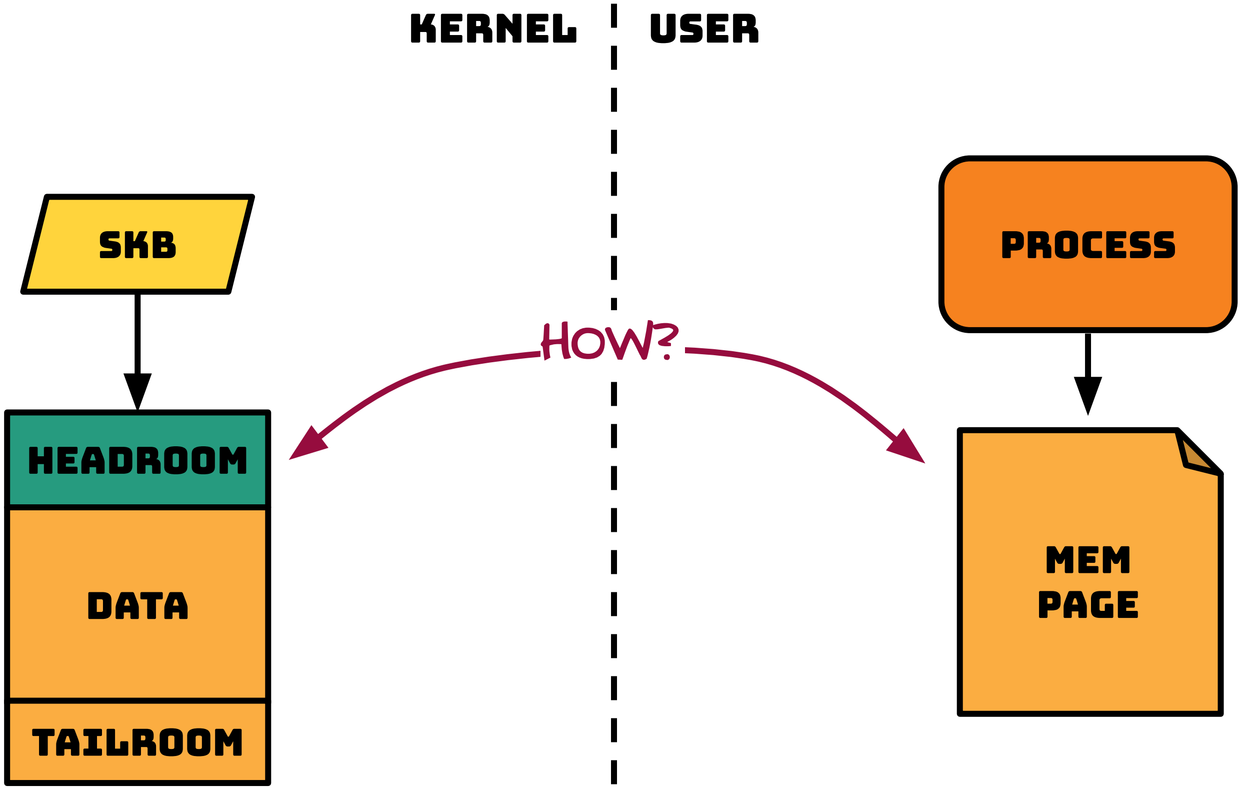
Beyond TC: limitations

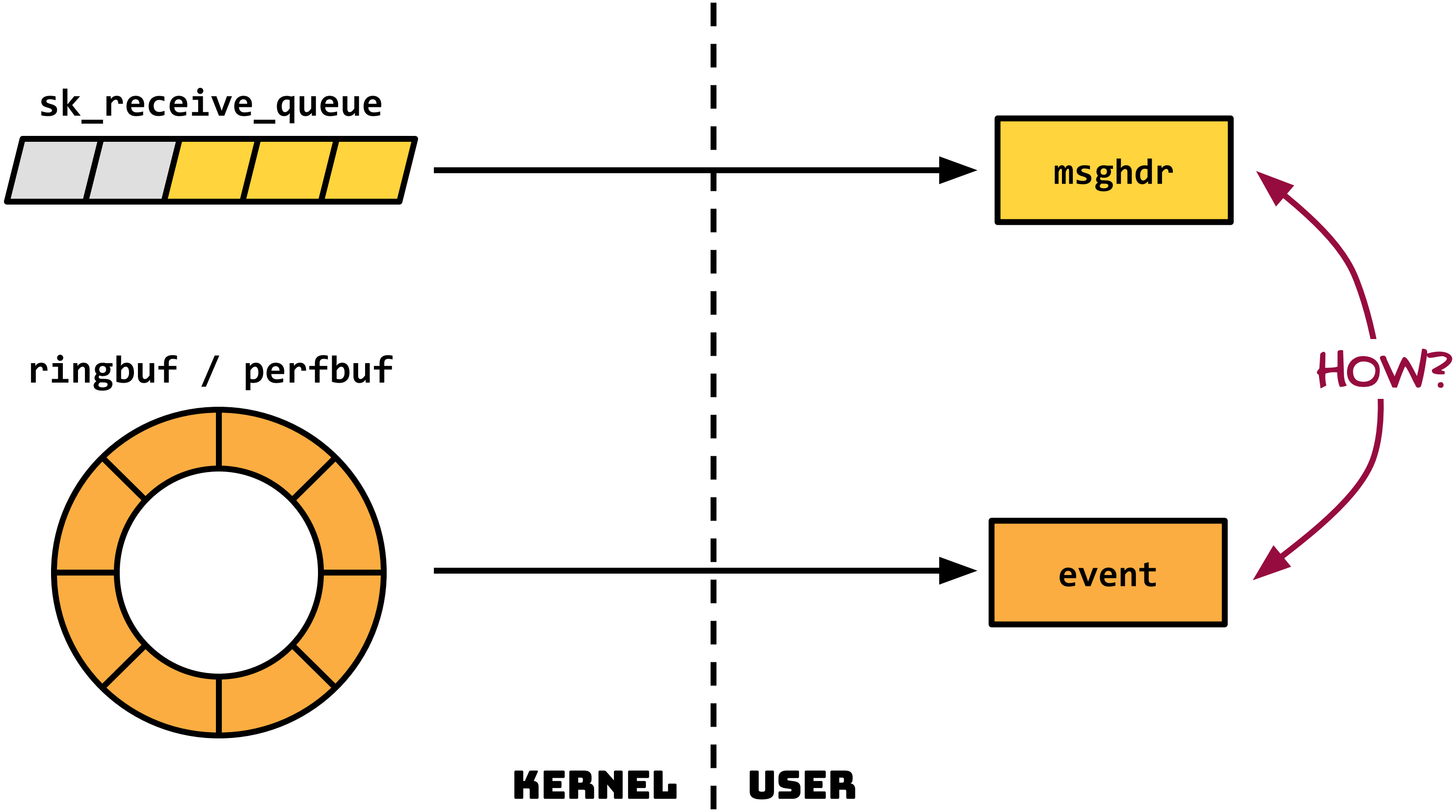
- No BPF for routing
  - Fallback to mark
- Local traffic?
  - New hook?

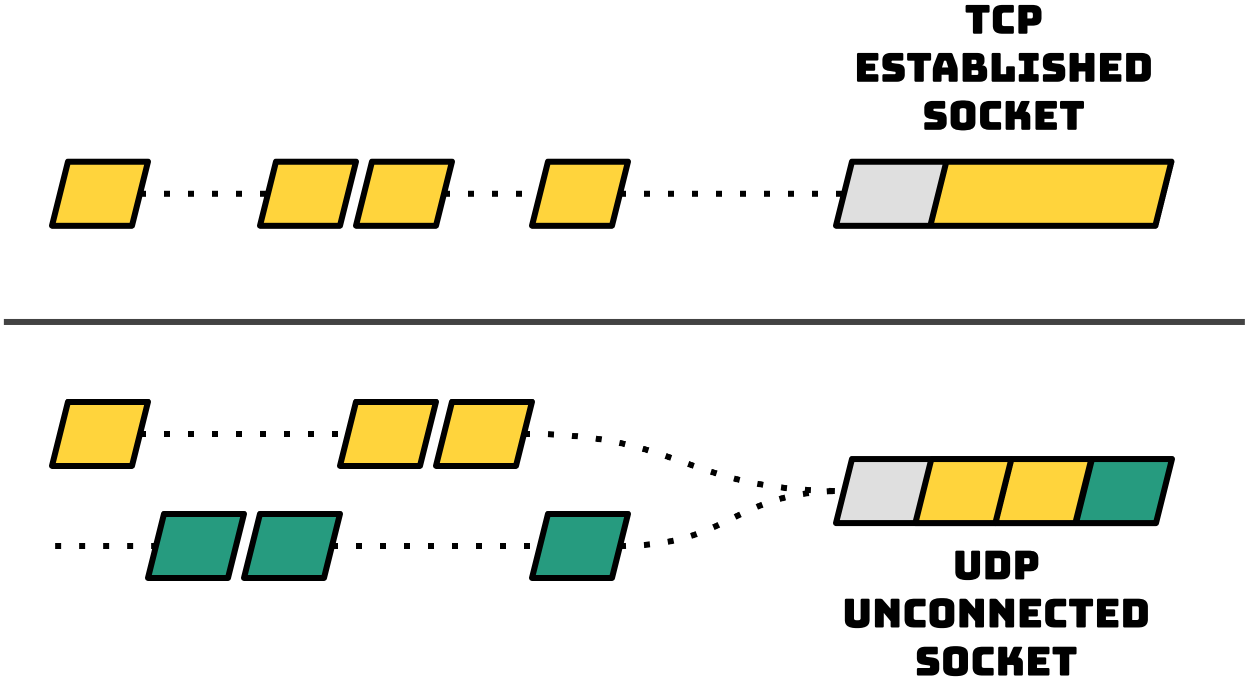
Beyond the SKB



How to pass SKB metadata to user-space and back?


KERNEL
USER
SKB
HEADROOM
DATA
TAILROOM
HOW?
PROCESS
MEM
PAGE


sk_receive_queue
msghdr
ringbuf / perfbuf
event
HOW?
KERNEL
USER



READ / WRITE
SKB METADATA

…

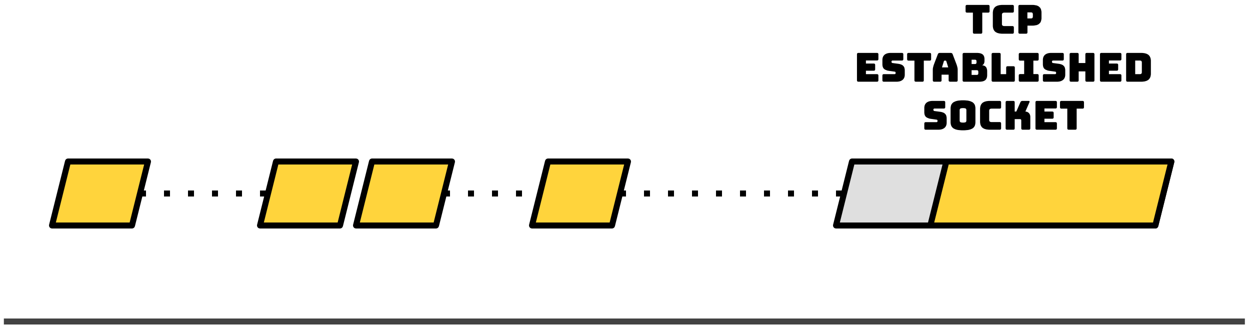
ONCE

VS

MANY TIMES

…

PER SOCKET
LIFETIME



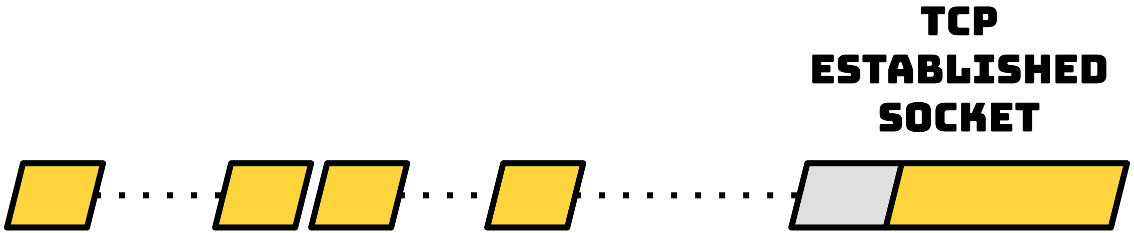
READ / WRITE
SKB METADATA

...

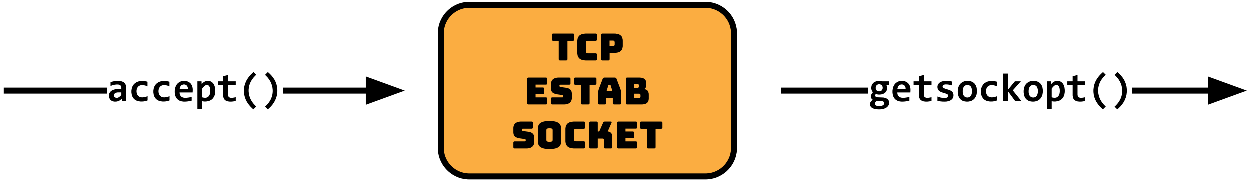
ONCE

...

PER SOCKET
LIFETIME



TCP ESTABLISHED SOCKET



accept()

TCP ESTAB SOCKET

getsockopt()

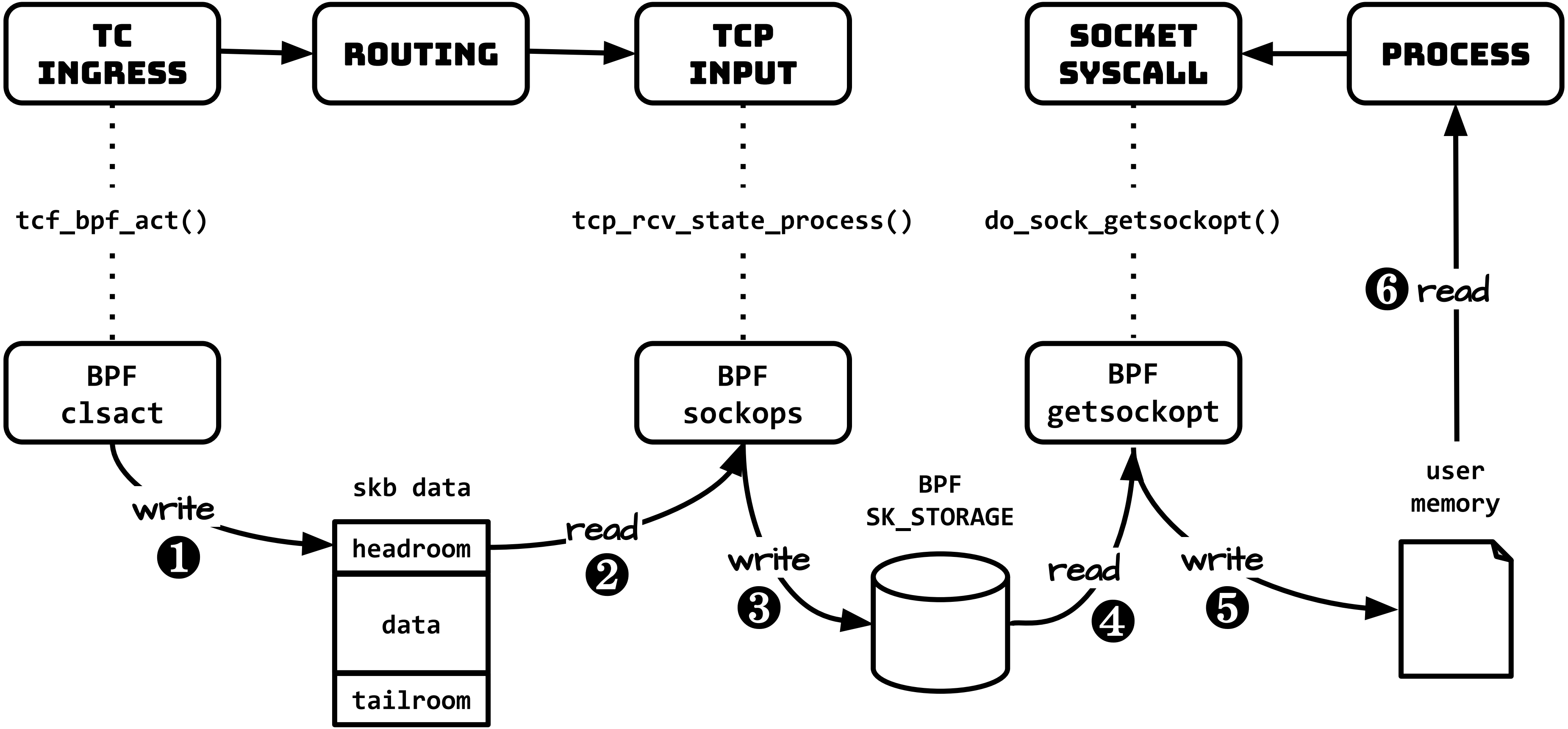
READ / WRITE
SKB METADATA

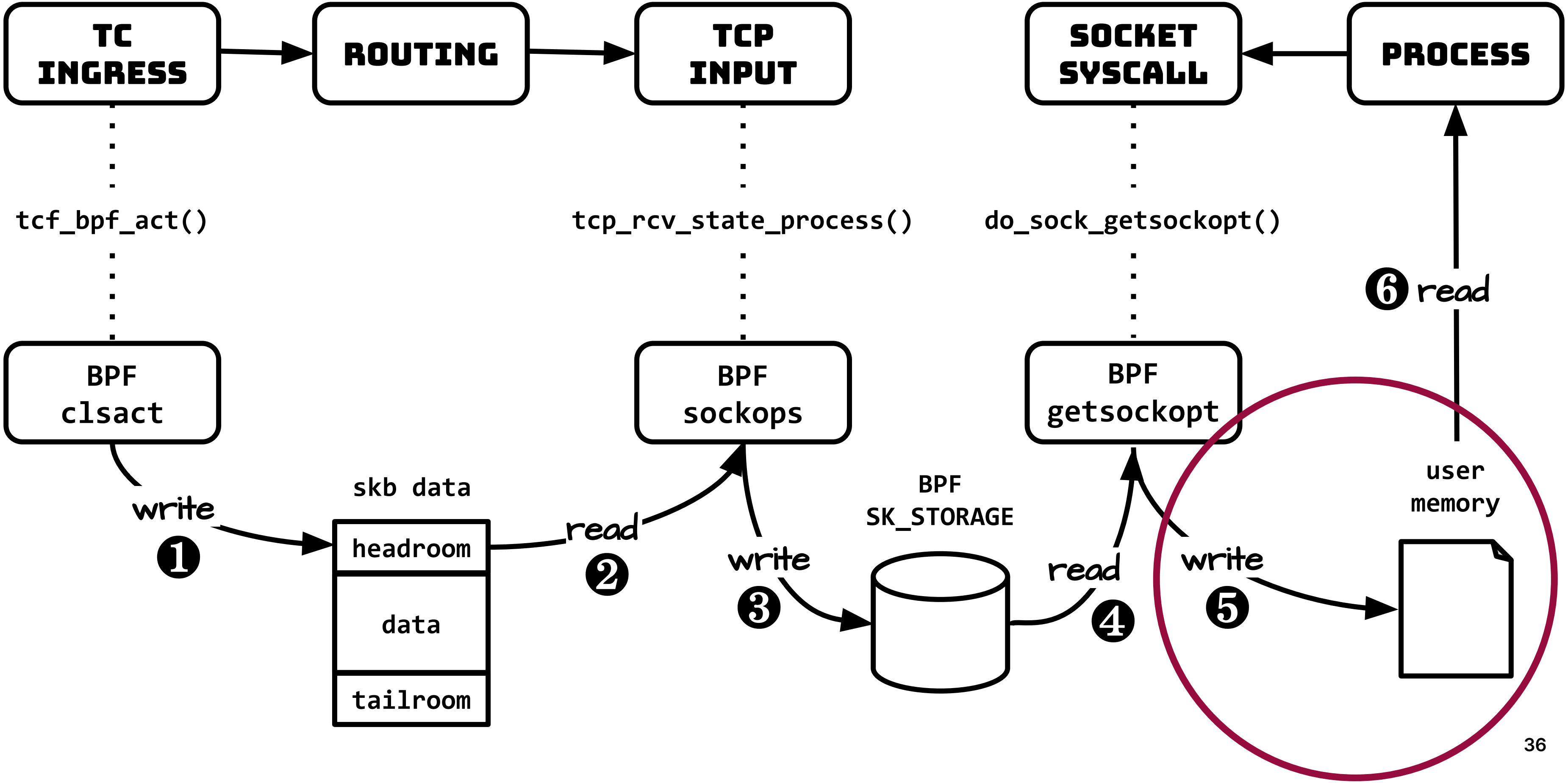
...

ONCE

...

PER SOCKET
LIFETIME

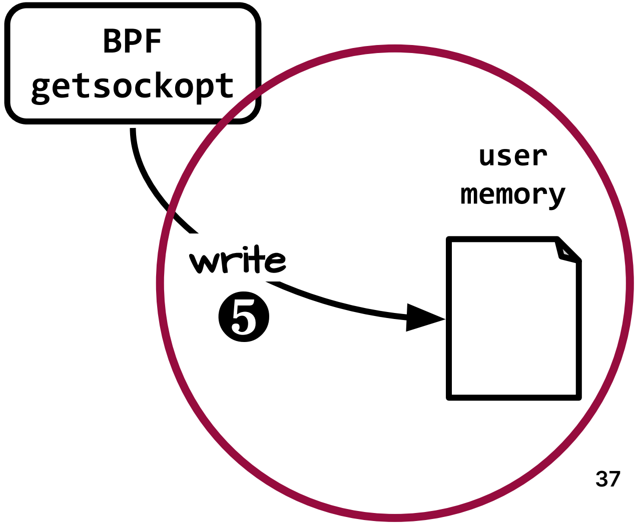

TC INGRESS
ROUTING
TCP INPUT
SOCKET SYSCALL
PROCESS
tcf_bpf_act()
tcp_rcv_state_process()
do_sock_getsockopt()
BPF clsact
BPF sockops
BPF getsockopt
❻ read
user memory
write ❶
skb data
headroom
data
tailroom
read ❷
write ❸
BPF SK_STORAGE
read ❹
write ❺


TC INGRESS
ROUTING
TCP INPUT
SOCKET SYSCALL
PROCESS
tcf_bpf_act()
tcp_rcv_state_process()
do_sock_getsockopt()
BPF clsact
BPF sockops
BPF getsockopt
❻ read
user memory
skb data
headroom
data
tailroom
write ❶
read ❷
write ❸
BPF SK_STORAGE
read ❹
write ❺

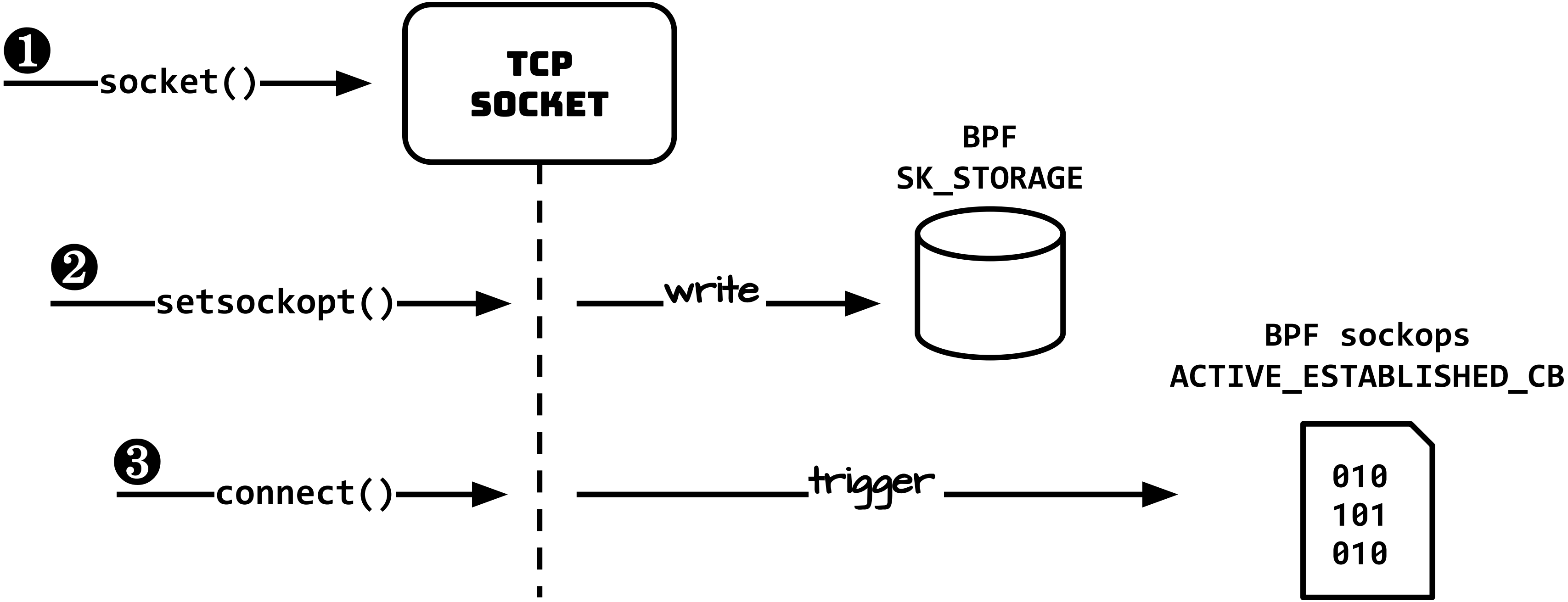
[RFC bpf-next 0/5] Sleepable BPF programs on cgroup {get,set}sockopt

https://lore.kernel.org/all/20230722052248.1062582-1-kuifeng@meta.com/

… but we have < 4 KiB
for skb metadata
anyway

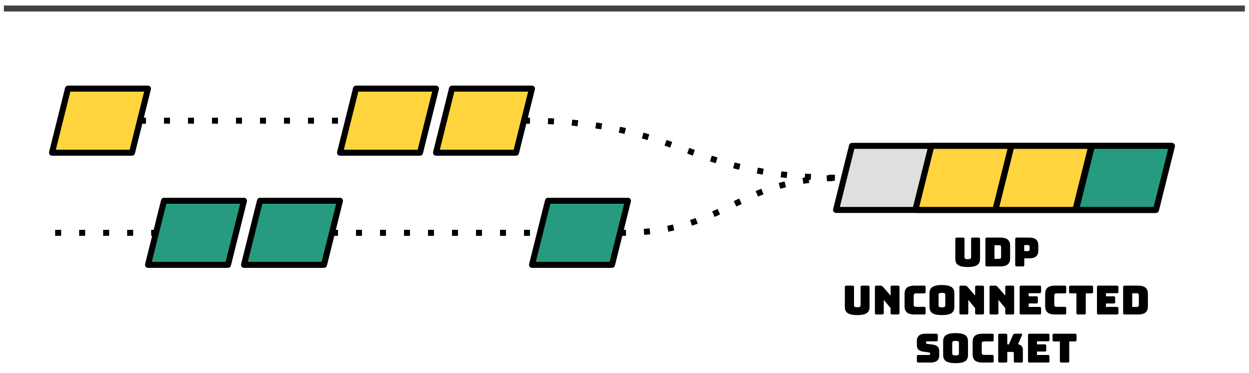


Can we do the same for outgoing connections?



```
SOL_BPF = 0xEB9F
```

Can we reserve a socket level value for BPF?



READ / WRITE
SKB METADATA

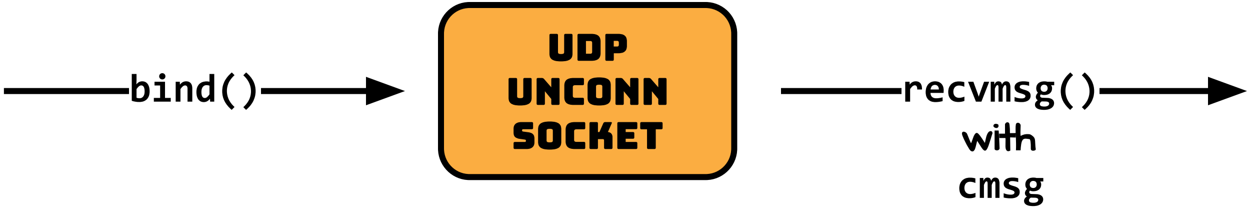
...

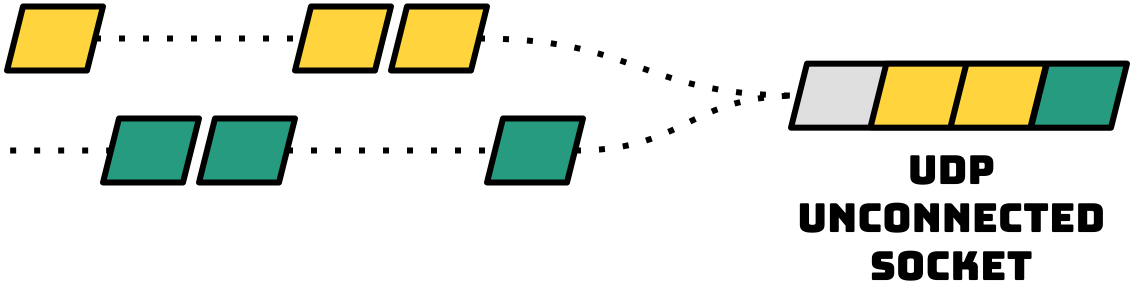
<u>MANY TIMES</u>

...

PER SOCKET
LIFETIME

https://man7.org/linux/man-pages/man3/cmsg.3.html





READ / WRITE SKB METADATA

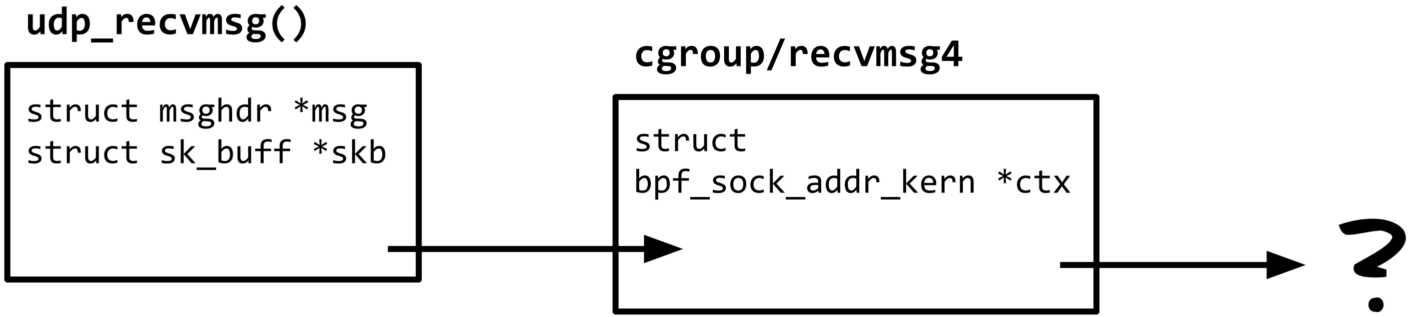
...

MANY TIMES

...

PER SOCKET LIFETIME

https://man7.org/linux/man-pages/man3/cmsg.3.html



udp_recvmsg()

```
struct msghdr *msg
struct sk_buff *skb
```

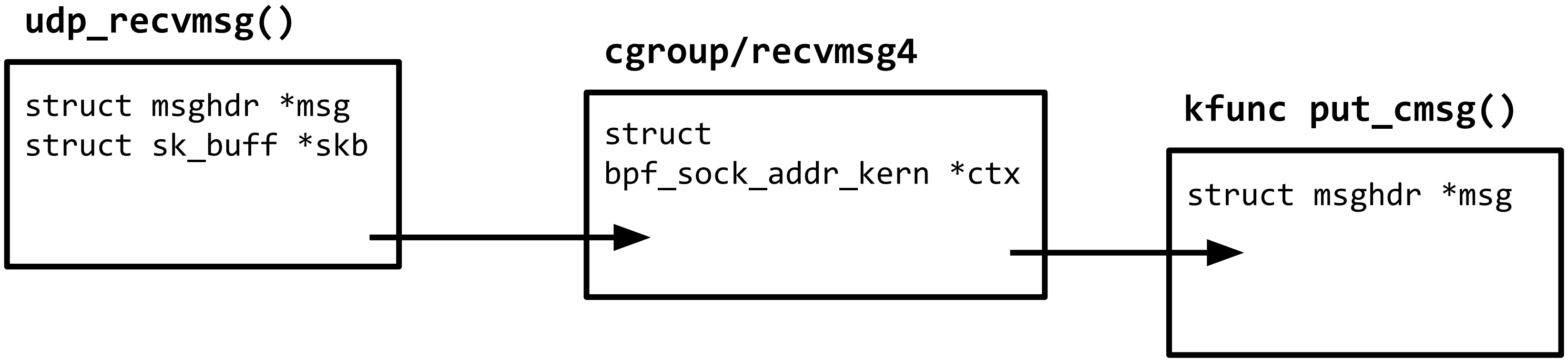
cgroup/recvmsg4

```
struct
bpf_sock_addr_kern *ctx
```

?

```
int put_cmsg(struct msghdr * msg, int level, int type, int len, void *data)
{
        // …

        if (msg->msg_control_is_user) {
                struct cmsghdr __user *cm = msg->msg_control_user;

                check_object_size(data, cmlen - sizeof(*cm), true);

                if (!user_write_access_begin(cm, cmlen))
                        goto efault;

                unsafe_put_user(cmlen, &cm->cmsg_len, efault_end);
                unsafe_put_user(level, &cm->cmsg_level, efault_end);
                unsafe_put_user(type, &cm->cmsg_type, efault_end);
                unsafe_copy_to_user(CMSG_USER_DATA(cm), data,
                                    cmlen - sizeof(*cm), efault_end);
                user_write_access_end();
        } // …
}
```

https://elixir.bootlin.com/linux/v6.10/source/net/core/scm.c#L231



udp_recvmsg()

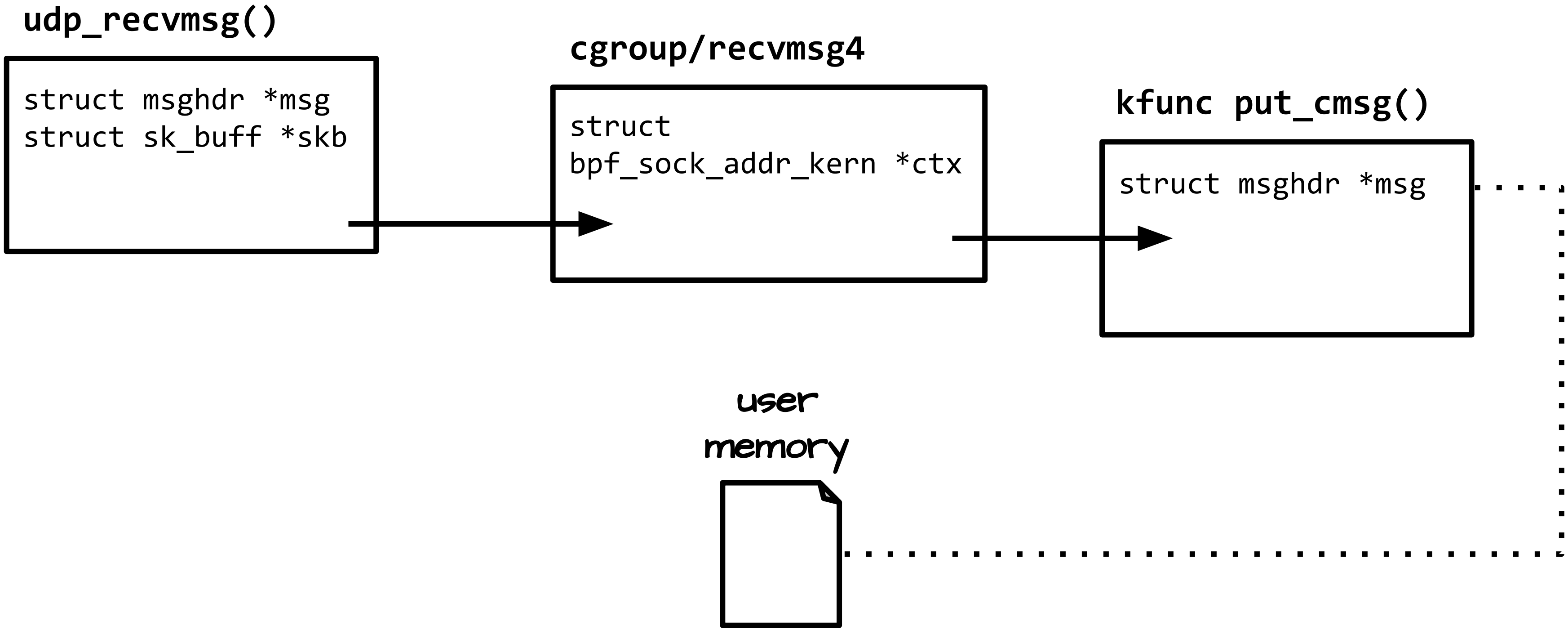
```
struct msghdr *msg
struct sk_buff *skb
```

cgroup/recvmsg4

```
struct
bpf_sock_addr_kern *ctx
```

kfunc put_cmsg()

```
struct msghdr *msg
```



udp_recvmsg()

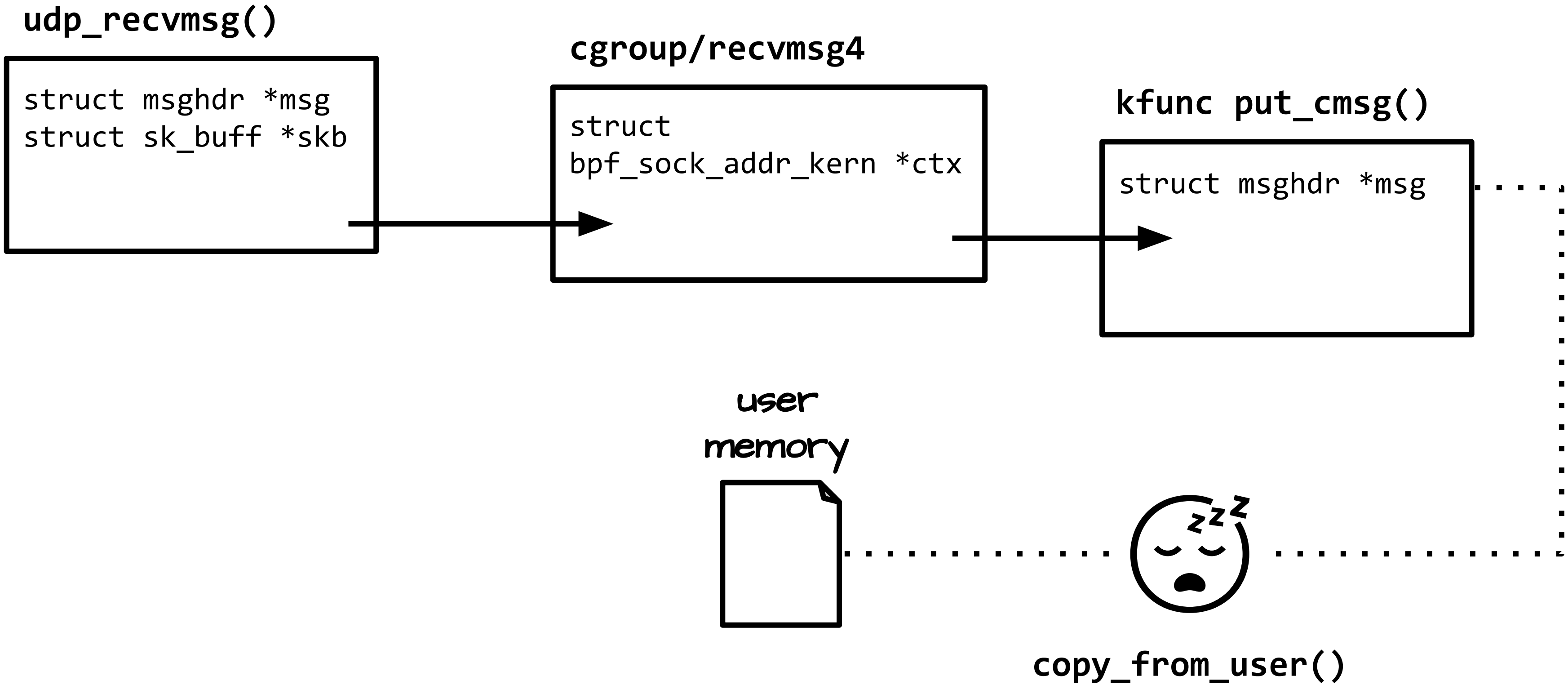
```
struct msghdr *msg
struct sk_buff *skb
```

cgroup/recvmsg4

```
struct
bpf_sock_addr_kern *ctx
```

kfunc put_cmsg()

```
struct msghdr *msg
```



udp_recvmsg()

```
struct msghdr *msg
struct sk_buff *skb
```

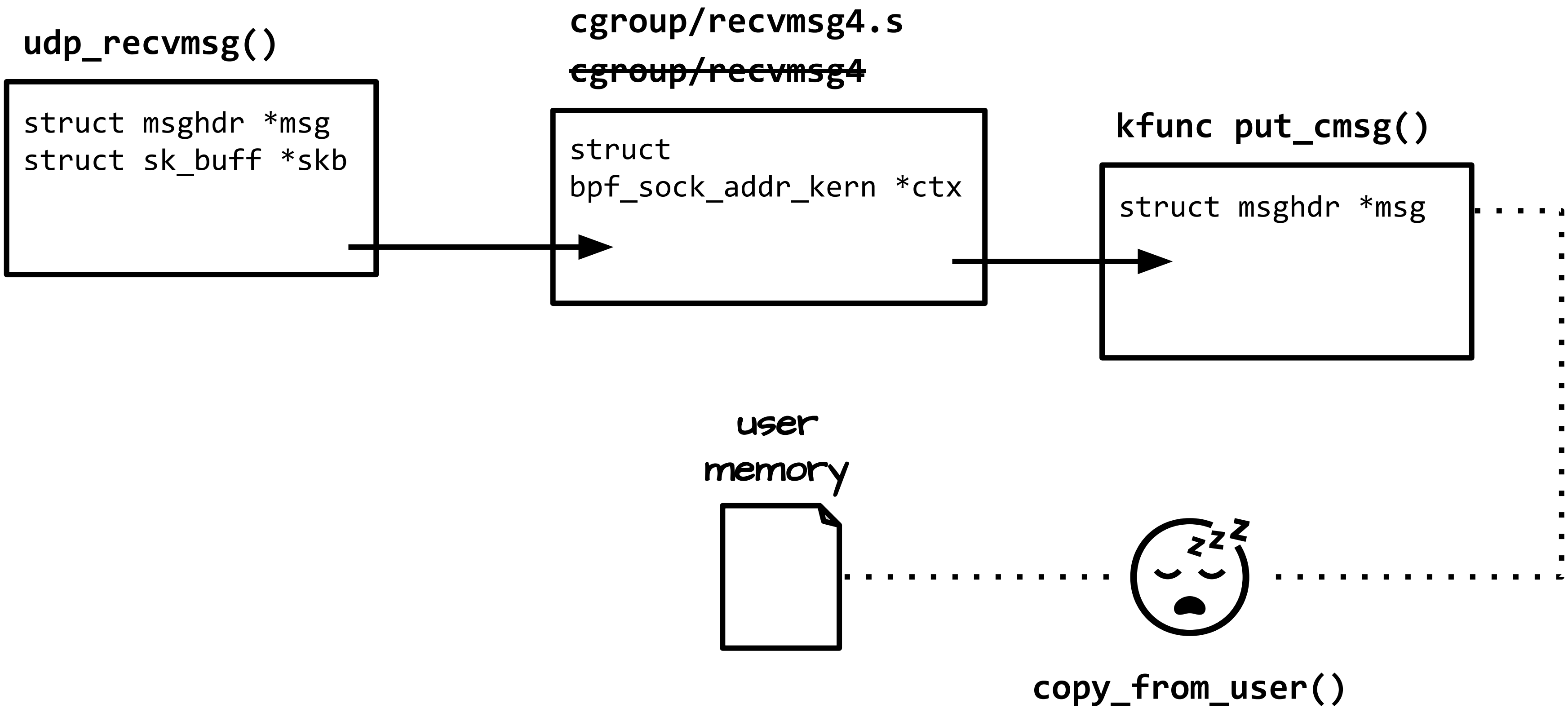
cgroup/recvmsg4

```
struct
bpf_sock_addr_kern *ctx
```

kfunc put_cmsg()

```
struct msghdr *msg
```

copy_from_user()



udp_recvmsg()

struct msghdr *msg
struct sk_buff *skb

cgroup/recvmsg4.s
~~cgroup/recvmsg4~~

struct
bpf_sock_addr_kern *ctx

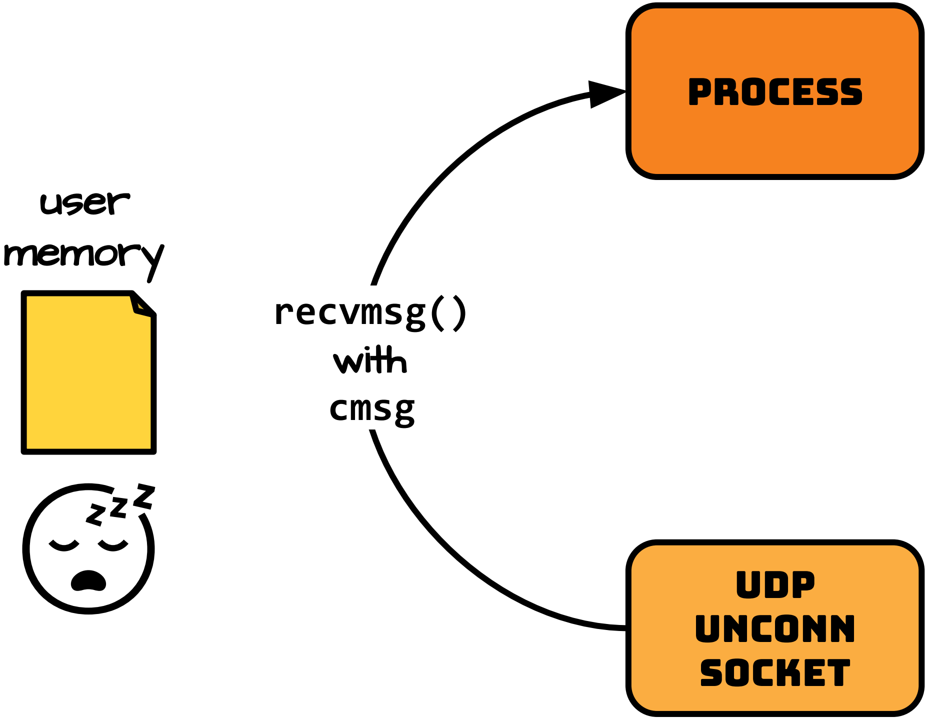
kfunc put_cmsg()

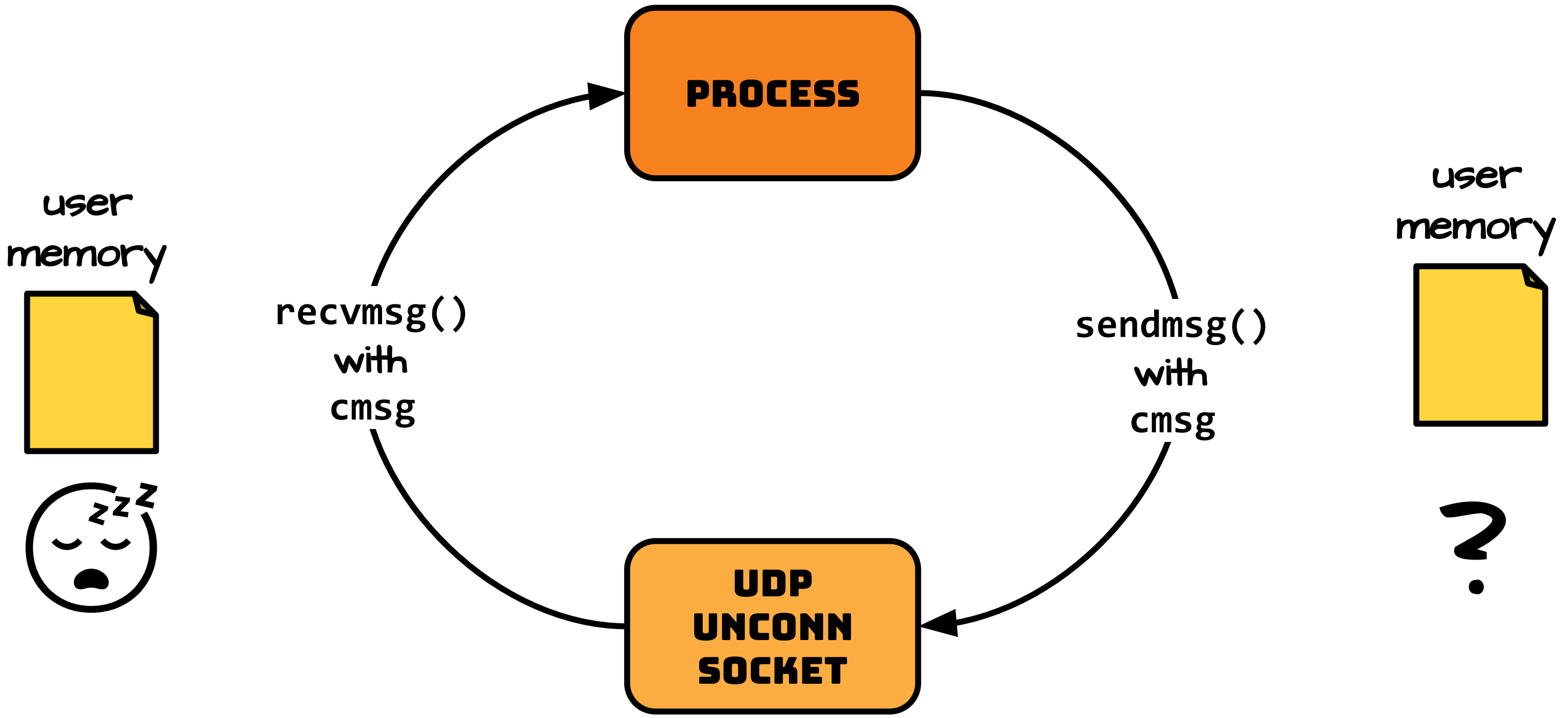
struct msghdr *msg

user
memory

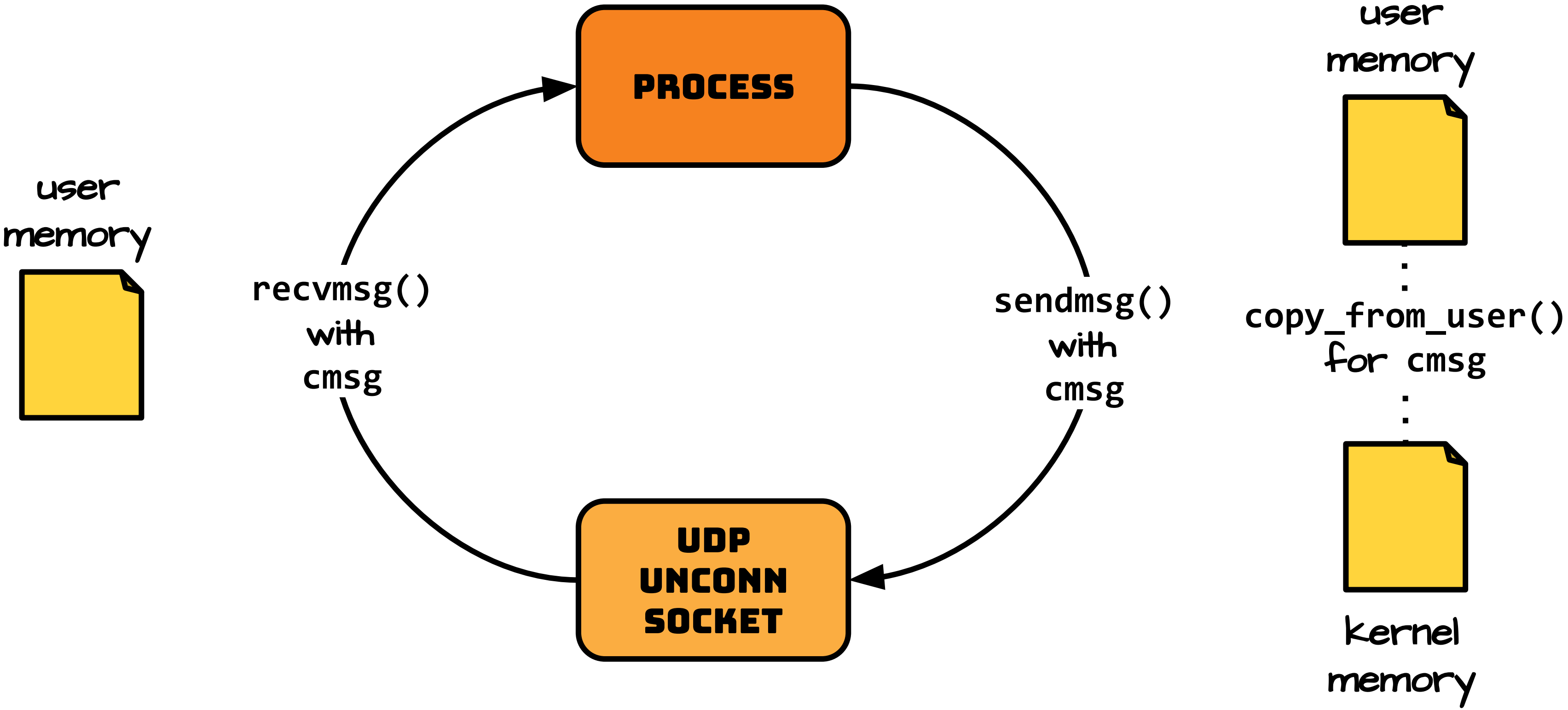
copy_from_user()

```
SEC("cgroup/recvmsg4.s")
int udp4_cmsg_put(struct bpf_sock_addr *ctx)
{
        struct msghdr *msg;
        char v = 42;
        int r;

        msg = bpf_sock_addr_msg_acquire(ctx);
        if (!msg)
                goto out;

        r = bpf_msg_put_cmsg(msg, SOL_BPF, SO_BPF_ANSWER, &v, sizeof(v));
        if (r)
                __sync_fetch_and_add(&error_count, 1);

        bpf_msg_release(msg);
out:
        return CG_OK;
}
```

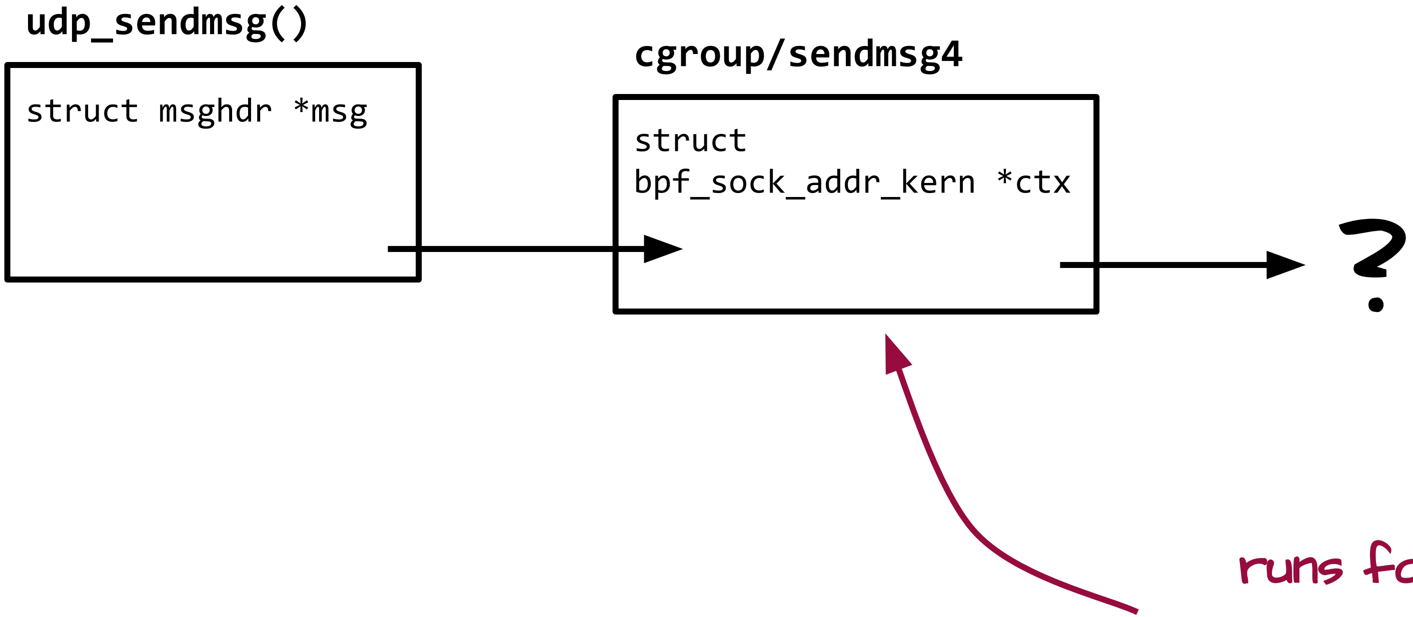

user memory
recvmsg() with cmsg
PROCESS
UDP UNCONN SOCKET



user memory
PROCESS
recvmsg() with cmsg
sendmsg() with cmsg
UDP UNCONN SOCKET
user memory
?



https://elixir.bootlin.com/linux/v6.10/source/net/socket.c#L2562



udp_sendmsg()

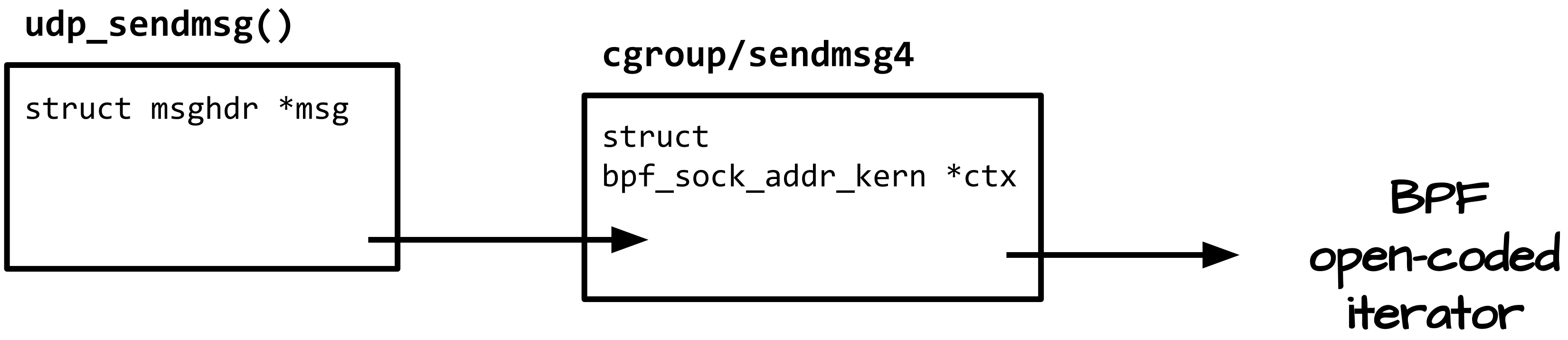
struct msghdr *msg

cgroup/sendmsg4

struct
bpf_sock_addr_kern *ctx

?

runs for
unconnected
sockets only...



udp_sendmsg()

struct msghdr *msg

cgroup/sendmsg4

struct
bpf_sock_addr_kern *ctx

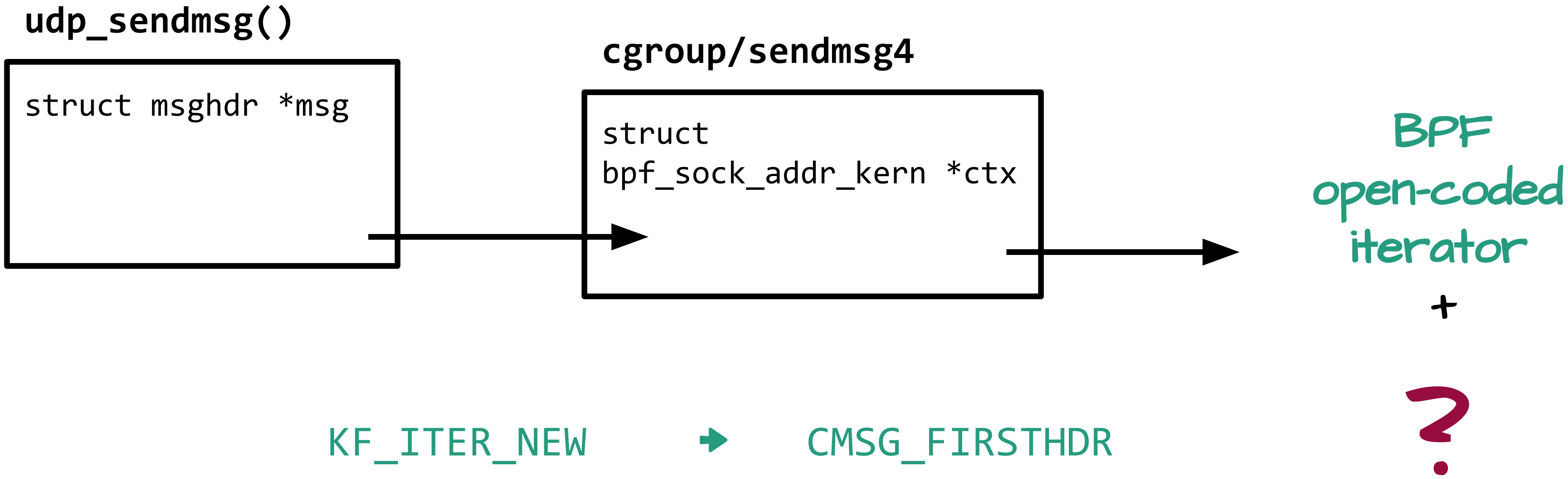
BPF
open-coded
iterator

KF_ITER_NEW	→	CMSG_FIRSTHDR
KF_ITER_NEXT	→	CMSG_NXTHDR
KF_ITER_DESTROY	→	∅

```
SEC("cgroup/sendmsg4")
int udp4_cmsg_get(struct bpf_sock_addr *ctx)
{
        struct cmsghdr *cmsg;
        struct msghdr *msg;
        int count = 0;

        msg = bpf_sock_addr_msg_acquire(ctx);
        if (!msg)
                goto out;

        bpf_for_each(cmsghdr, cmsg, msg) {
                if (cmsg->cmsg_level == SOL_BPF && cmsg->cmsg_type == SO_BPF_ANSWER)
                        count++;
        }

        bpf_msg_release(msg);
out:
        return CG_OK;
}
```

```
SEC("cgroup/sendmsg4")
int udp4_cmsg_get(struct bpf_sock_addr *ctx)
{
        struct cmsghdr *cmsg;
        struct msghdr *msg;
        int count = 0;

        msg = bpf_sock_addr_msg_acquire(ctx);
        if (!msg)
                goto out;

        bpf_for_each(cmsghdr, cmsg, msg) {
                if (cmsg->cmsg_level == SOL_BPF && cmsg->cmsg_type == SO_BPF_ANSWER)
                        count++;
        }

        bpf_msg_release(msg);
out:
        return CG_OK;
}
```

```
bpf_iter_cmsghdr_new()
bpf_iter_cmsghdr_next()
bpf_iter_cmsghdr_destroy()
```



udp_sendmsg()

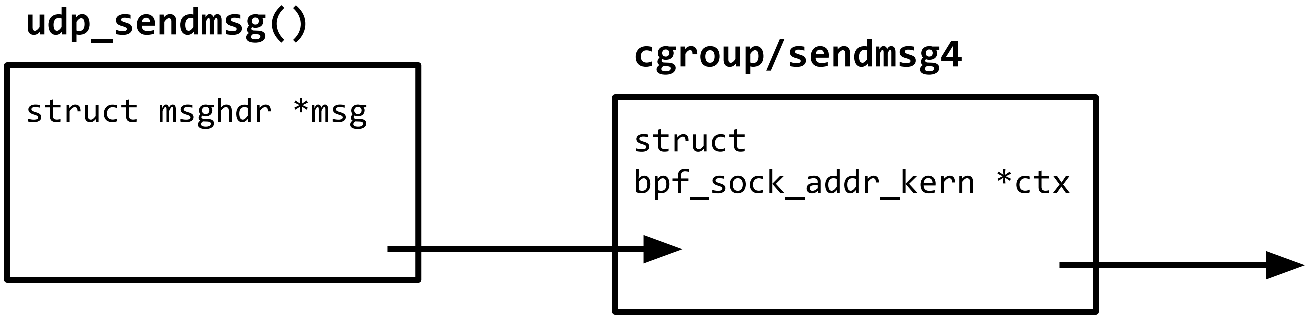
struct msghdr *msg

cgroup/sendmsg4

struct
bpf_sock_addr_kern *ctx

BPF
open-coded
iterator
+
?

KF_ITER_NEW	➔	CMSG_FIRSTHDR
KF_ITER_NEXT	➔	CMSG_NXTHDR
KF_ITER_DESTROY	➔	∅
???	➔	CMSG_DATA



udp_sendmsg()

```
struct msghdr *msg
```

cgroup/sendmsg4

```
struct
bpf_sock_addr_kern *ctx
```

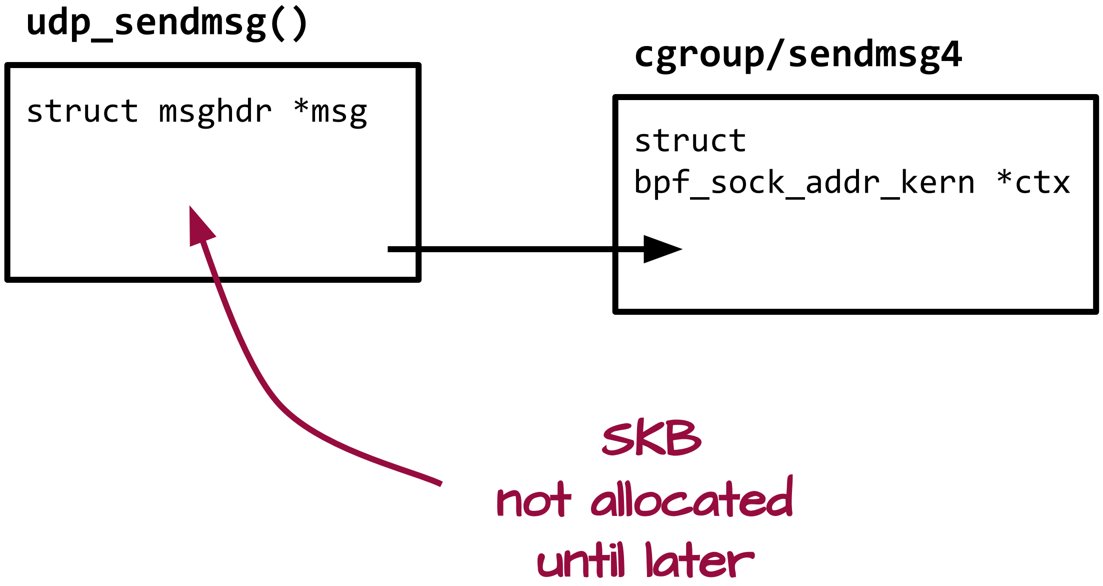
BPF open-coded iterator

+

BPF dynptr

KF_ITER_NEW	➔	CMSG_FIRSTHDR
KF_ITER_NEXT	➔	CMSG_NXTHDR
KF_ITER_DESTROY	➔	∅
bpf_dynptr_init	➔	CMSG_DATA

```
bpf_for_each(cmsghdr, cmsg, msg) {
        struct bpf_dynptr ptr;
        __u32 *data;

        if (cmsg->cmsg_level != SOL_BPF || cmsg->cmsg_type != SO_BPF_ANSWER)
                continue;

        err = bpf_dynptr_from_cmsg(cmsg, &ptr);
        if (err)
                continue;

        data = bpf_dynptr_slice(&ptr, 0, NULL, sizeof(*data));
        if (!data)
                continue;

        bpf_printk("answer %u\n", *data);
}
```



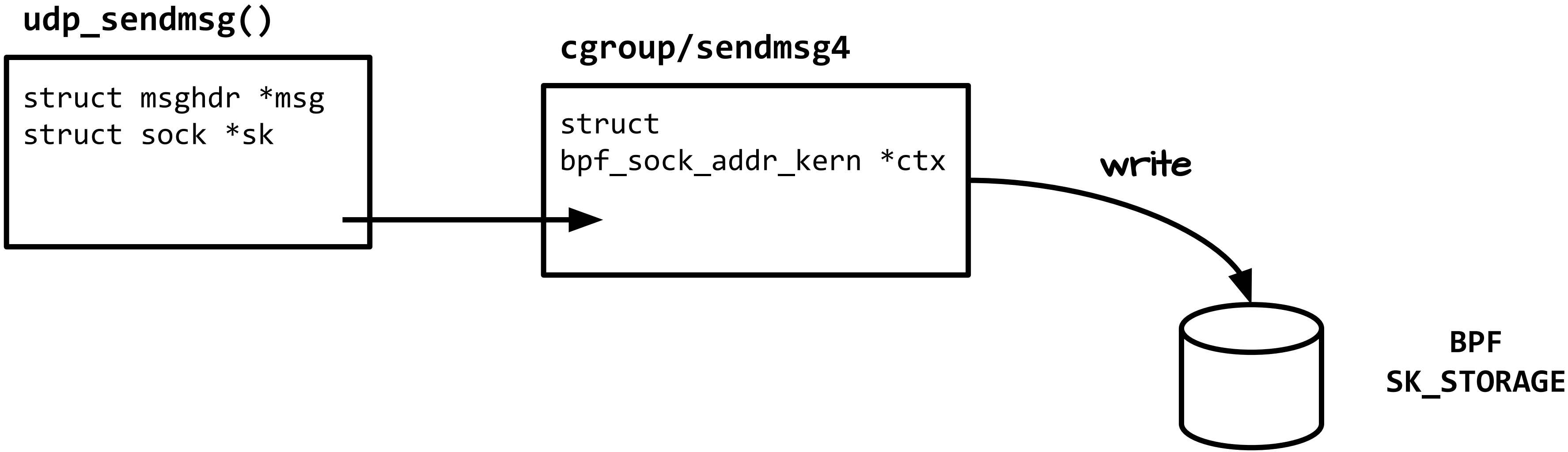
udp_sendmsg()

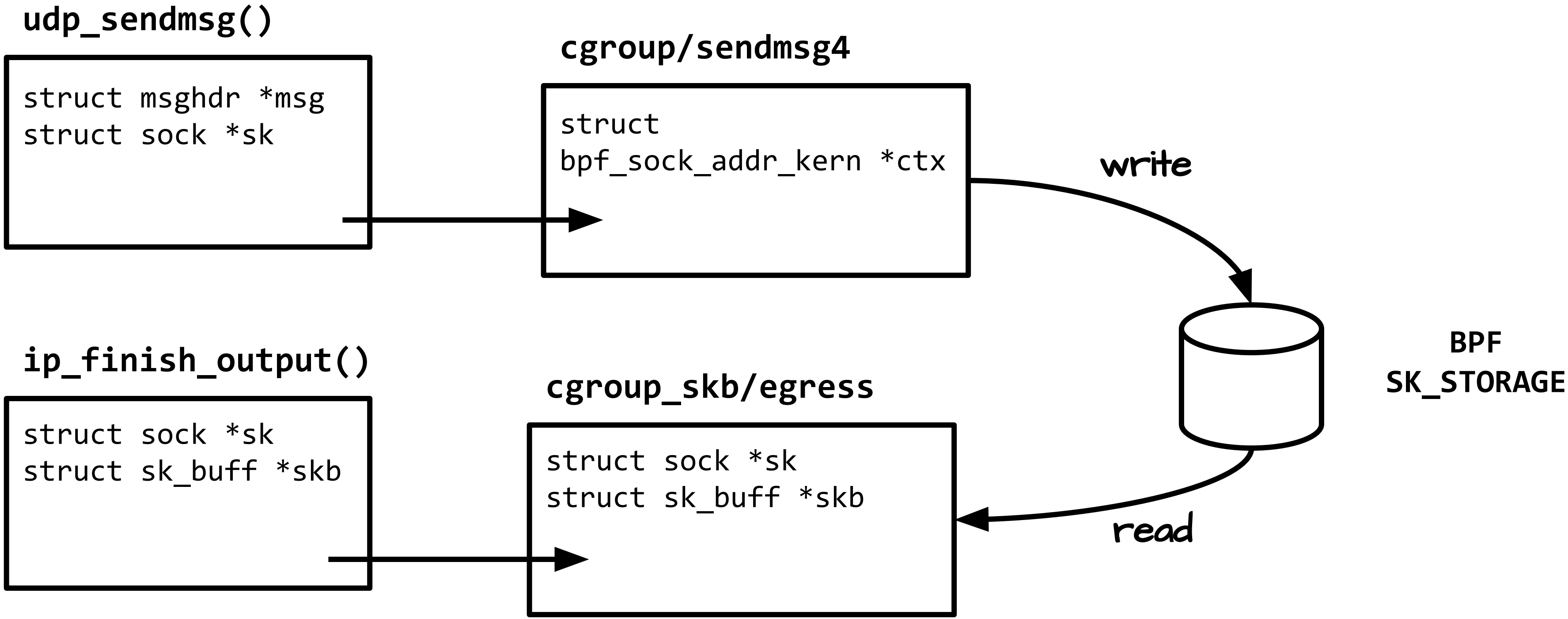
```
struct msghdr *msg
```

cgroup/sendmsg4

```
struct
bpf_sock_addr_kern *ctx
```

SKB
not allocated
until later


udp_sendmsg()
struct msghdr *msg
struct sock *sk
cgroup/sendmsg4
struct
bpf_sock_addr_kern *ctx
write
BPF
SK_STORAGE



udp_sendmsg()

```
struct msghdr *msg
struct sock *sk
```

cgroup/sendmsg4

```
struct
bpf_sock_addr_kern *ctx
```

write

BPF
SK_STORAGE

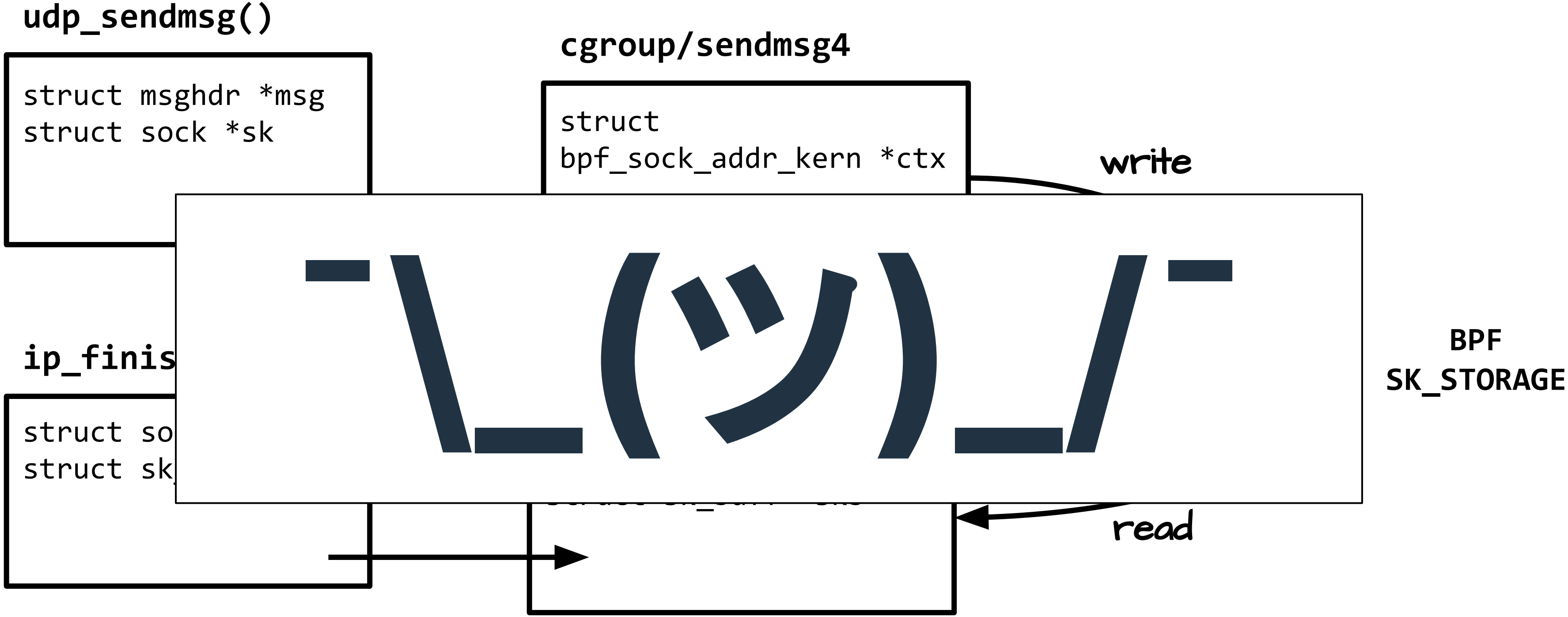
read

ip_finish_output()

```
struct sock *sk
struct sk_buff *skb
```

cgroup_skb/egress

```
struct sock *sk
struct sk_buff *skb
```



udp_sendmsg()

```
struct msghdr *msg
struct sock *sk
```

cgroup/sendmsg4

```
struct
bpf_sock_addr_kern *ctx
```

write

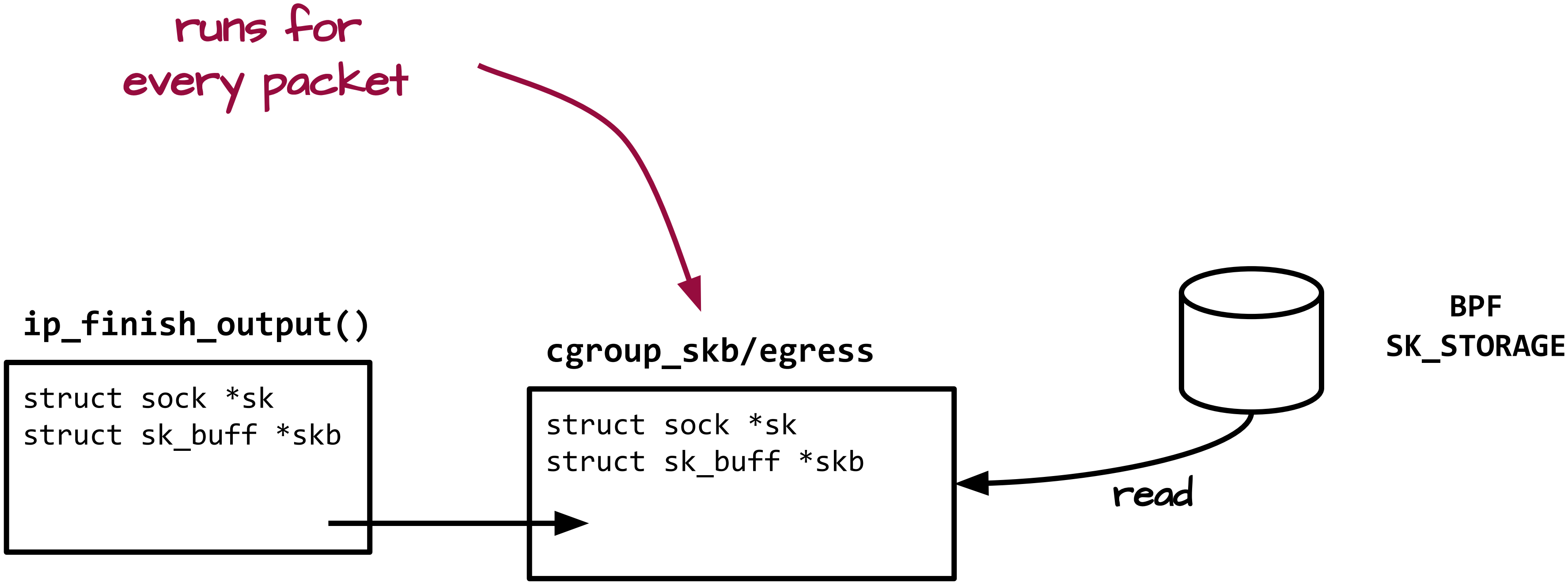
BPF
SK_STORAGE

ip_finis

```
struct so
struct sk
```

read

¯_(ツ)_/¯



runs for
every packet

ip_finish_output()

```
struct sock *sk
struct sk_buff *skb
```

cgroup_skb/egress

```
struct sock *sk
struct sk_buff *skb
```

read

BPF
SK_STORAGE

Can we do better?

Metadata format

MetaElephant

- ~~How many bits can I use?~~
- Which ones?
- Will it interfere with other services?
- We shouldn’t need a registry.

- Add fields at runtime?

Binary Blob?

- Binary blob / struct

- Pros:
  - Simple
- Cons:
  - System-wide agreement
    - Make fields configurable?
  - Can’t move / change existing fields

```
struct meta {
    __u32 service_id;
    __u32 rx_timestamp;
    char source;
    __u64 pkt_id;
}

service_field = “0:32”
```

BTF & CO-RE?

- BTF type
  - System-wide?
  - Per skb?

- Pros:
  - Layout can change.
- Cons:
  - CO-RE assumes types don’t change at runtime.

```
struct meta {
    __u32 service_id;
    __u32 rx_timestamp;
    char source;
    __u64 pkt_id;
    __u32 btf_id;
}

if (btf_id == 4) {
    BPF_CORE_READ(m, 4, …);
} else {
    BPF_CORE_READ(m, 5, …);
}
```

- LPC 2022: XDP gaining access to NIC hardware hints via BTF

Magic map?

- Magic KV like map?
- Explicitly register keys with kernel?

```
bpf_map_lookup_elem(
    &skb->meta,
    SERVICE_ID,
)
```

TLV

- Cons:
  - Need to parse all TLVs each time
- Pros:
  - Very flexible
  - Space efficient

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   META TYPE   |   META LEN     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              DATA              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   SERVICE_ID  |      2         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              FOO               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  RX_TIMESTAMP |      4         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            2343234             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```