eBPF Programming

Sebastiano Miano

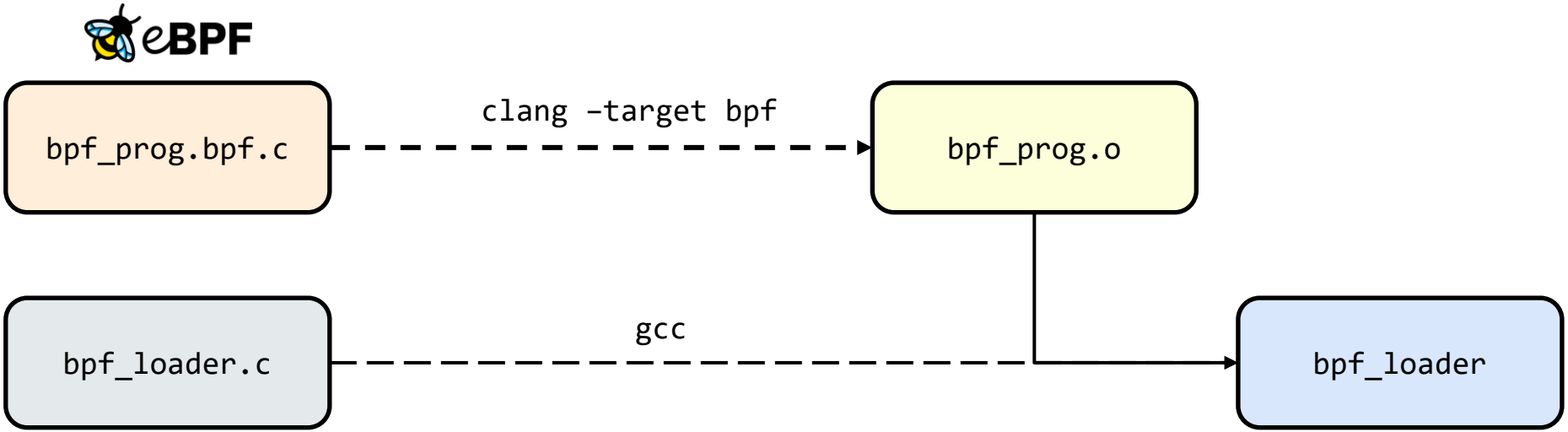
PhD School @ TMA '24 - May 21st, Dresden

RESTART

POLITECNICO
MILANO 1863

The structure of BPF programs

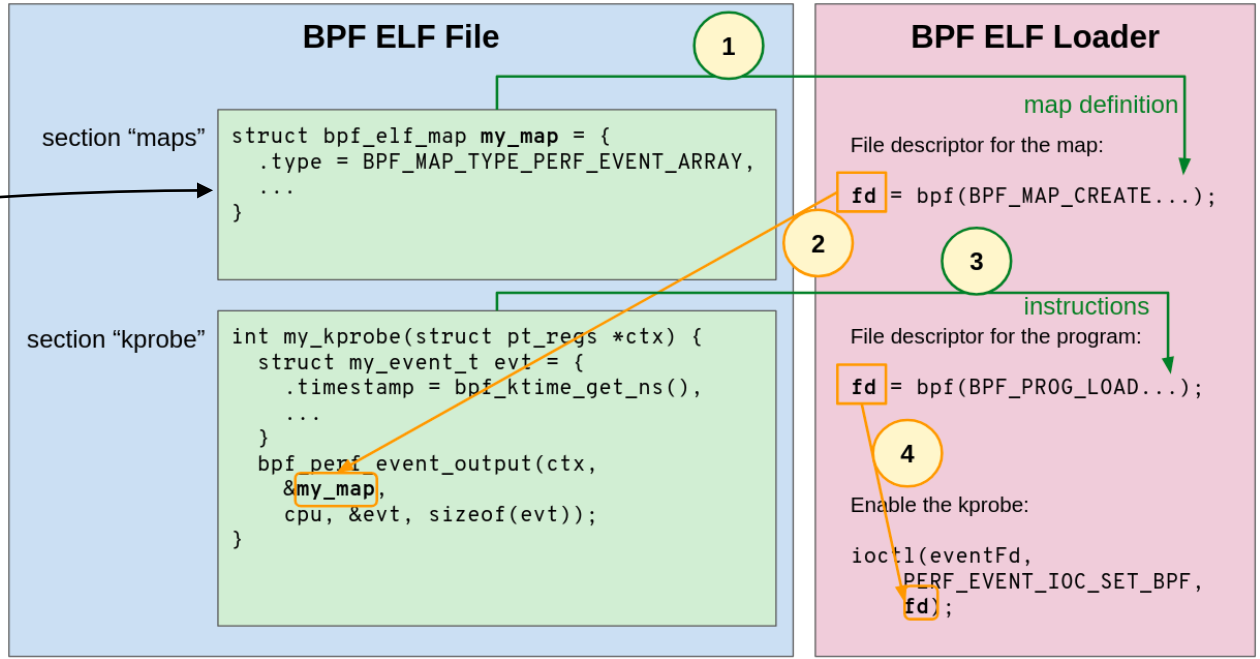
- A project aimed at developing BPF program usually consist of two types of source files:
  - The source code file of the BPF program **running in the kernel**
  - The source file of the **user state program** use to load the BPF program in the kernel



eBPF program: ELF File

- BPF is usually compiled from C, using Clang, and linked into a single ELF file

ELF sections are used to distinguish map definitions and executable code.



Develop BPF programs with  LIBBPF

- Working directly with the kernel's BPF subsystem can be complex and challenging

- **LIBBPF** bridges this gap by providing a set of high-level APIs that simplify the eBPF kernel interactions
- In this way developers can focus on code logic of their eBPF apps

Develop BPF programs with  LIBBPF

The key functionality provided by **LIBBPF** are:

- **Loading eBPF bytecode:**
  - Provides APIs for loading eBPF bytecode from ELF files
- **Attaching eBPF Programs:**
  - Offers APIs for attaching eBPF program to various kernel hook points, such as tracepoints, kprobes, network sockets, and XDP points.
- **Managing eBPF Maps:**
  - Provides APIs for creating, deleting, updating and querying eBPF maps
- **Helper Functions:**
  - Includes a collection of helper functions that simplify common eBPF operations, both on BPF-side and userspace-side

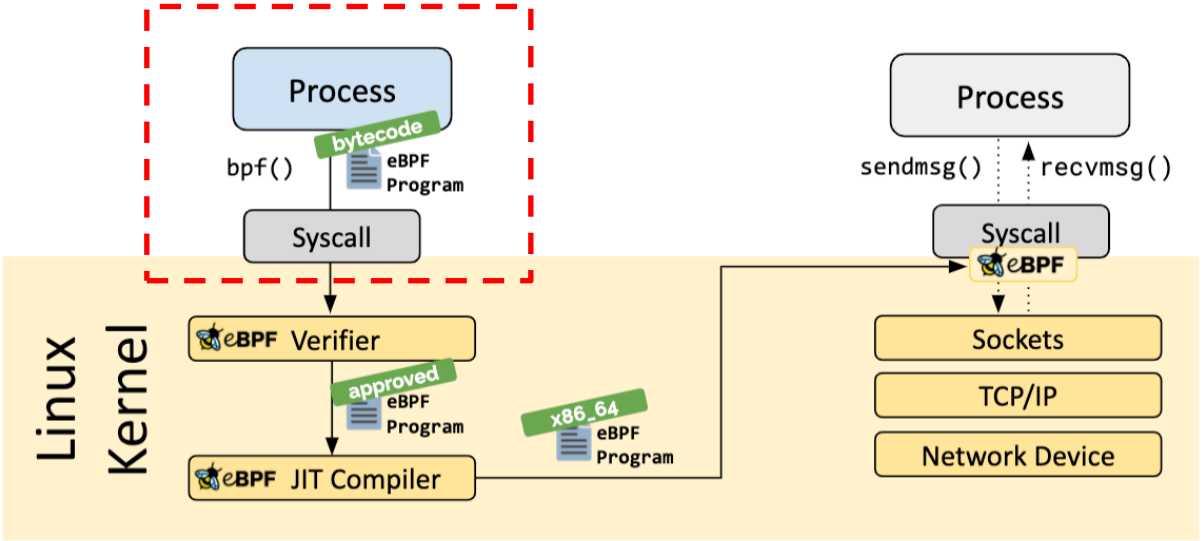
BPF App Lifecycle – 4 phases

- A BPF application consists of
  - One or more BPF programs (either cooperating or completely independent)
  - BPF maps
  - Global variables

- We can distinguish **4 phases** in the BPF life cycle

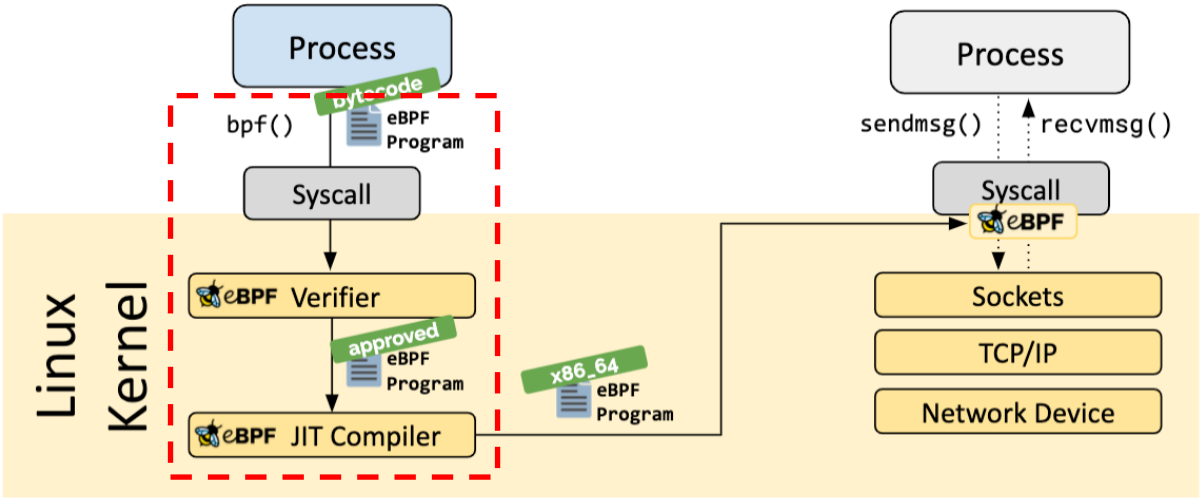
BPF App Lifecycle – Open Phase

- **Open Phase**: In this phase, libbpf parses the BPF object file and discovers BPF maps, BPF programs and global variables.
  - After the app is opened, userspace apps can make additional adjustments
  - Setting BPF program types
  - Pre-setting initial variables for global variables



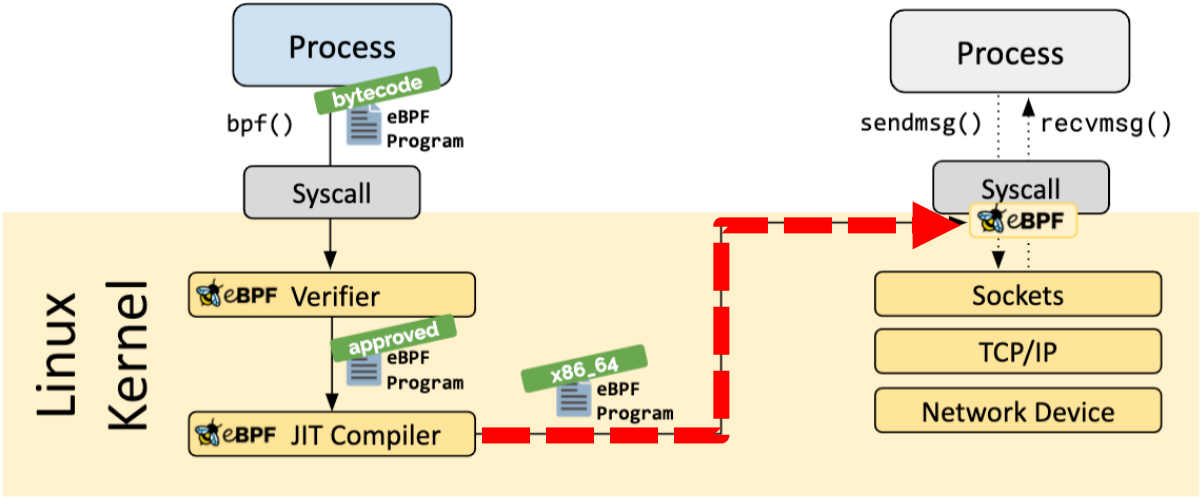
BPF App Lifecycle – Load Phase

- **Load Phase**: In this phase, libbpf creates BPF maps, resolves various relocations, and verifies and loads BPF programs into the kernel.
  - After that, the program is loaded into the kernel



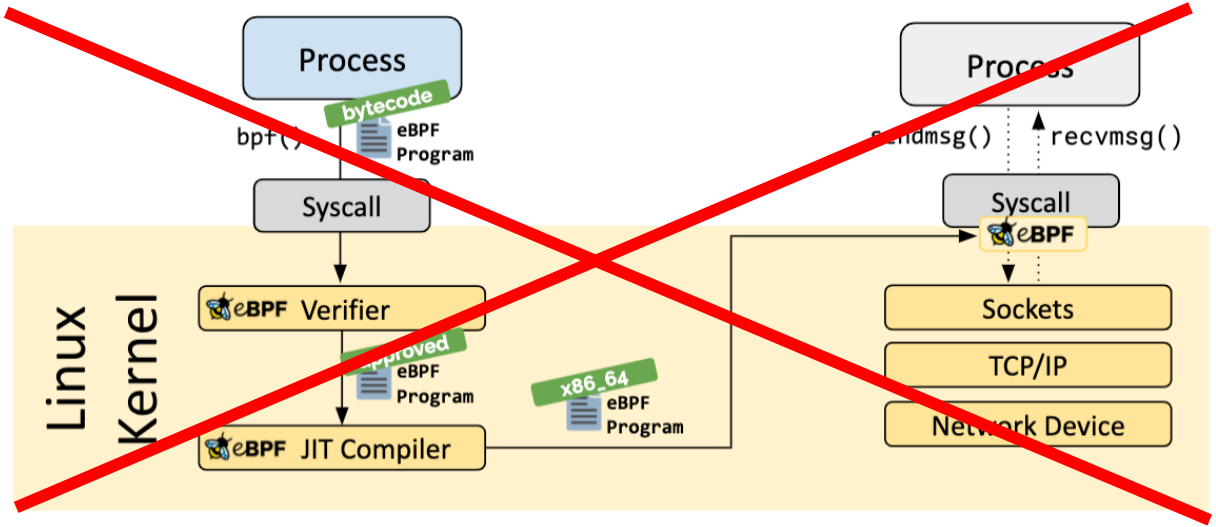
BPF App Lifecycle – Attachment Phase

- Attachment Phase: In this phase, libbpf attaches BPF programs to various BPF hook points.
  - During this phase, BPF programs perform useful work such as processing packets, or updating BPF maps and global variables that can be read from user space.



BPF App Lifecycle – Tear down Phase

- **Tear down phase**: In this phase, libbpf detaches BPF programs and unloads them from the kernel.
  - BPF maps are destroyed
  - All the resources used by the BPF app are freed



BPF Object Skeleton File

- BPF skeleton is an alternative interface to libbpf APIs for working with BPF objects.
- Skeleton code abstract away generic libbpf APIs to significantly simplify code for manipulating BPF programs from user space
- Skeleton code includes a bytecode representation of the BPF object file, simplifying the process of distributing your BPF code.
    - With BPF bytecode embedded, there are no extra files to deploy along with your application binary.

Generate Skeleton File

- The skeleton header file (.skel.h) for a specific object file can be generated using bpftool
  - bpftool is a versatile command-line utility designed for inspecting and managing eBPF programs and maps within the Linux kernel

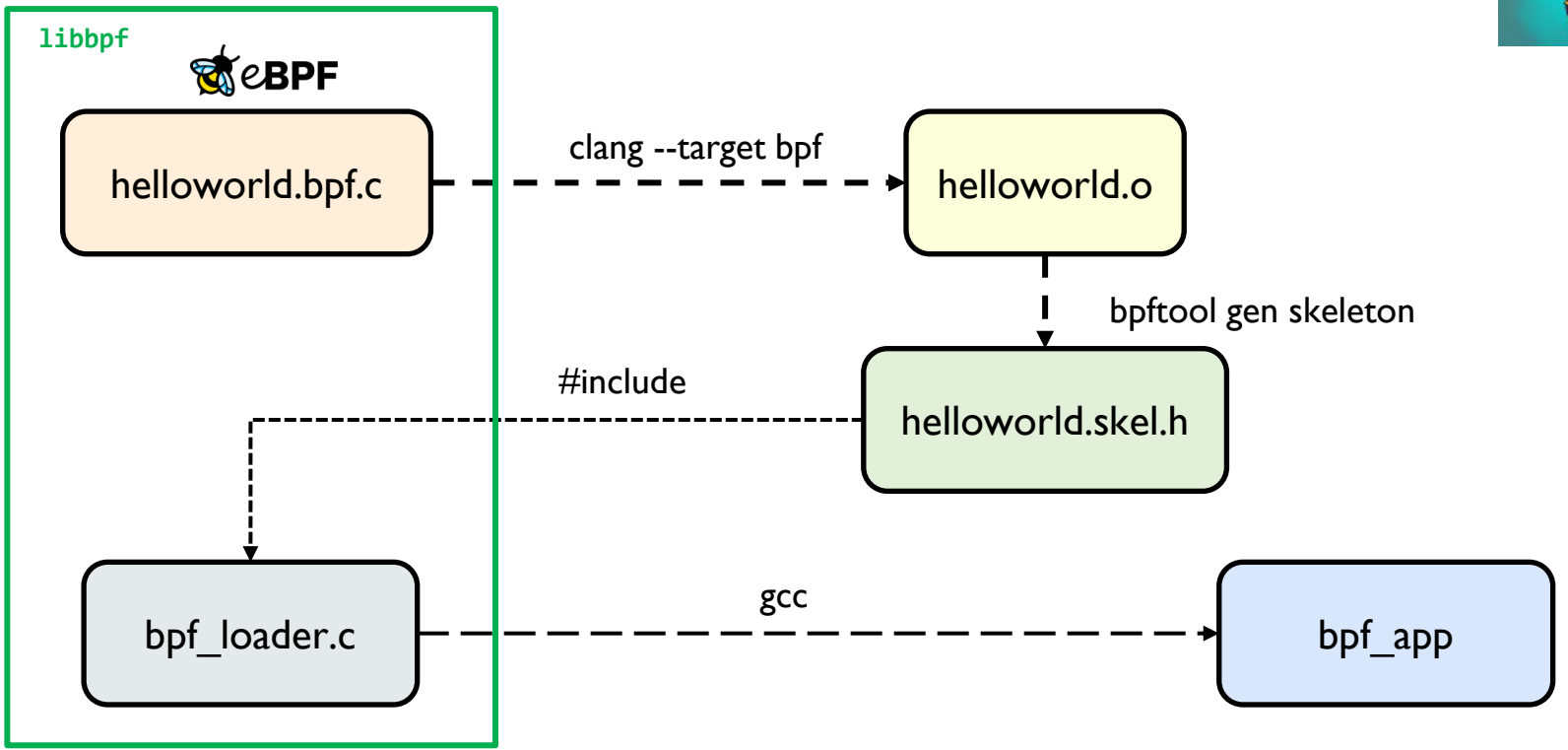
- The generated BPF skeleton provides the following custom functions that correspond to the BPF lifecycle, each of them prefixed with the specific object name.

Generate Skeleton File

- The generated BPF skeleton provides the following custom functions that correspond to the BPF lifecycle, each of them prefixed with the specific object name:

1. <name>__open(): creates and opens BPF application
2. <name>__load(): instantiates, loads, and verifies BPF application parts
3. <name>__attach(): attaches all auto-attachable BPF programs (it's optional, you can have more control by using libbpf APIs directly)
4. <name>__destroy(): detaches all BPF programs and frees up all used resources

Develop BPF programs with BPF Skeletons



eBPF Maps

What is an eBPF map

- eBPF maps are a generic data structure for storage of different data types

- Data types are generally treated as binary blobs, so a user just specifies the size of the key and the size of the value at map-creation time
    - In other words, a key/value for a given map can have an arbitrary structure.

- The map handles are file descriptors, and multiple maps can be created and accessed by multiple programs

Defining an eBPF map

- Maps can be defined directly in the BPF file.
- When the program is compiled a separate ELF section is created for maps
- When the program is loaded, the libbpf loaded creates the map and assigns a file descriptor to it

```
struct {
__uint(type, <map_type>);
__type(key, <key_type>);
__type(value, <value_type>);
__uint(max_entries, <# entries>);
} <map_name> SEC(".maps");
```

eBPF map types

- Several types of map are available.
- The most common are:
  - BPF_ARRAY: a static array of data stored sequentially
  - BPF_HASH: data are stored in a hash table. A hash of the provided key is computed and used as an index to access the corresponding value.
  - BPF_LRU_HASH: A hash-table that purges elements that have not recently been used.
  - BPF_LPM_TRIE: data are stored as an ordered search tree. As the name suggests, this map allows performing lookups based on the Longest Prefix Match algorithm.
  - BPF_PROG_ARRAY: this is a special map whose values contains only file descriptor to other eBPF programs. Used to implement tail calls.

Interacting with maps - Kernel

- Interacting with eBPF maps happens through some lookup/update/delete primitives.
- From the BPF side, you can use the following functions to interact with maps:

```
void *bpf_map_lookup_elem(map, void *key);
void bpf_map_update_elem(map, void *key, void *val, __u64 flags);
void bpf_map_delete_elem(map, void *key);
```

- The flags argument in bpf_map_update_elem() allows to define semantics on whether the element exists:

```
#define BPF_ANY       0 /* create new element or update existing */
#define BPF_NOEXIST   1 /* create new element only if it didn't exist */
#define BPF_EXIST     2 /* only update existing element */
```

Interacting with maps - Userspace

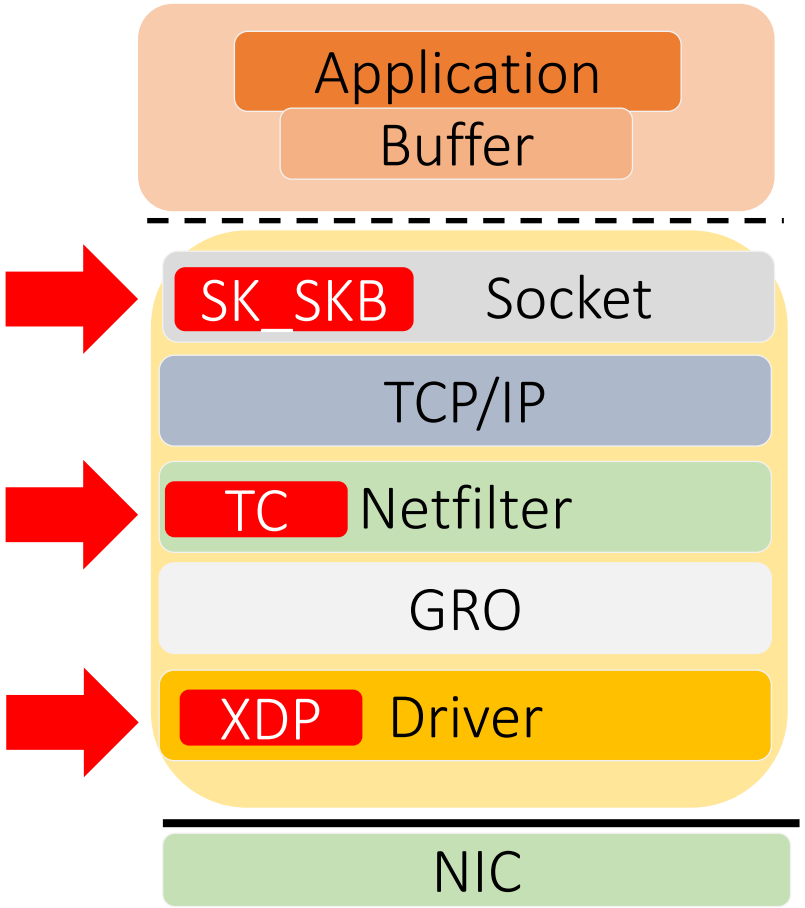
- Interacting with an eBPF map from userspace, happens through the bpf syscall and a file descriptor.

```
/* Userspace helpers */
int bpf_map_lookup_elem(int fd, void *key, void *value);
int bpf_map_update_elem(int fd, void *key, void *value, __u64 flags);
int bpf_map_delete_elem(int fd, void *key);
/* Only userspace: */
int bpf_map_get_next_key(int fd, void *key, void *next_key);
```

- Key and value are passed as void pointers
- Given the memory separation between kernel and userspace, this is a copy of the value

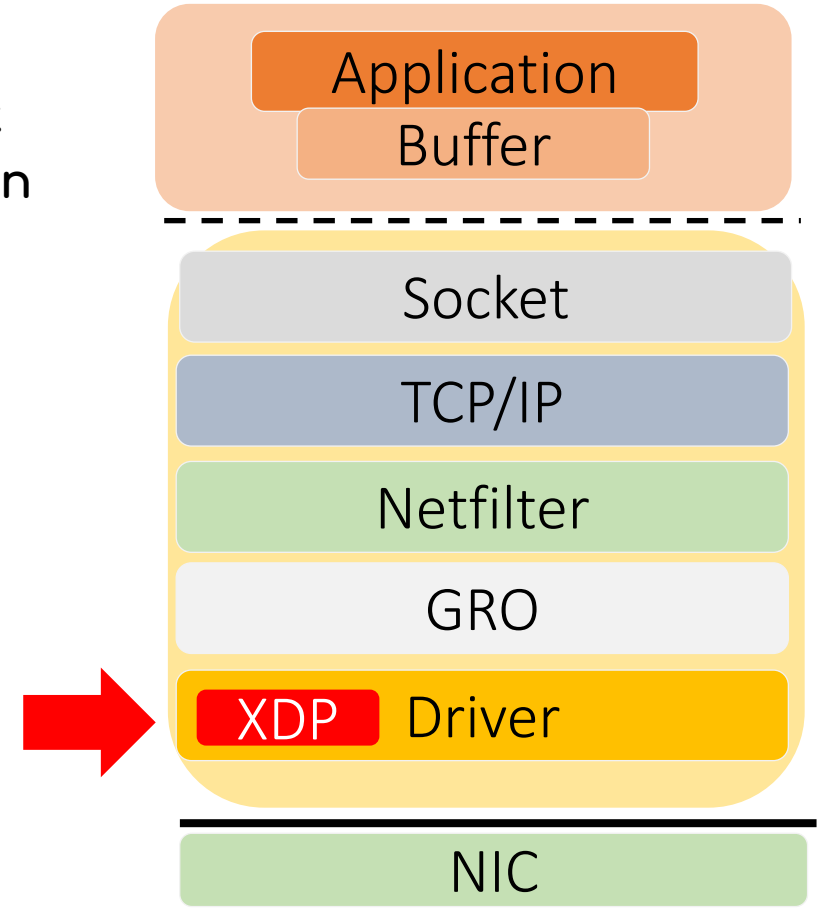
eBPF Networking Hook points

- Several hook points (a.k.a. kernel events) for networking:
- Located at different levels of the stack
- Opens the possibility to implement packet processing programs at different layers of the stack
  - eXpress Data Path (XDP)
  - Traffic Control (TC)
  - Socket SKB (SK_SKB)
  - There are many more...



XDP Hook

- XDP allows to run eBPF programs at the driver level, before skb allocation
- This is the earliest hook-point, suitable for high-performance processing

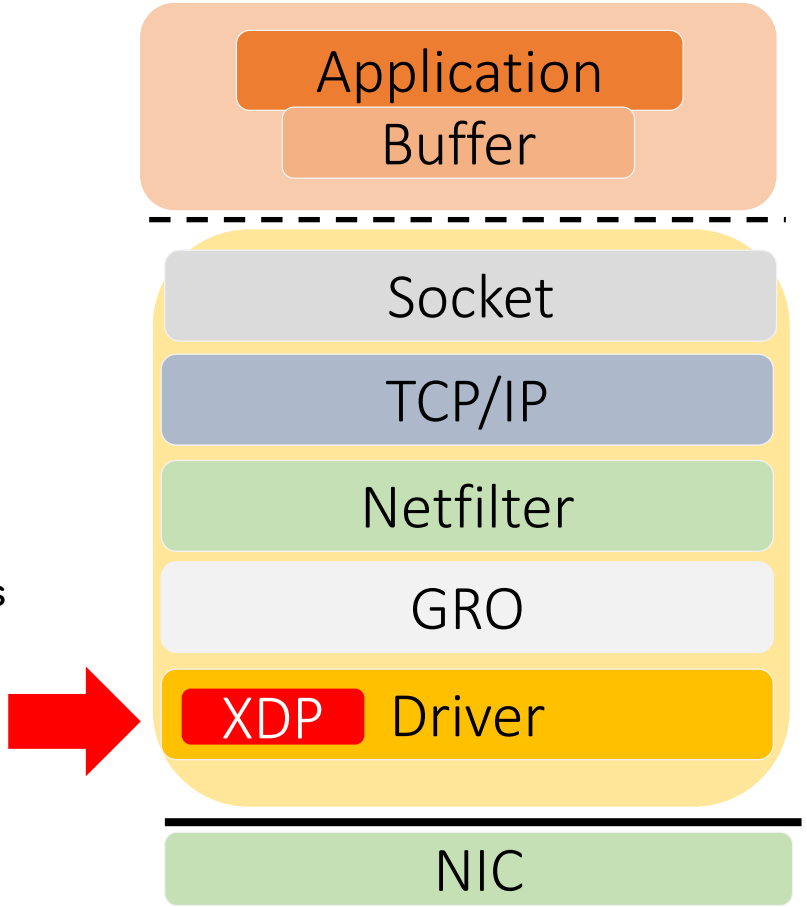


[1] https://ebpf-docs.dylanreimerink.nl/linux/program-type/BPF_PROG_TYPE_XDP/

XDP Hook – execution modes

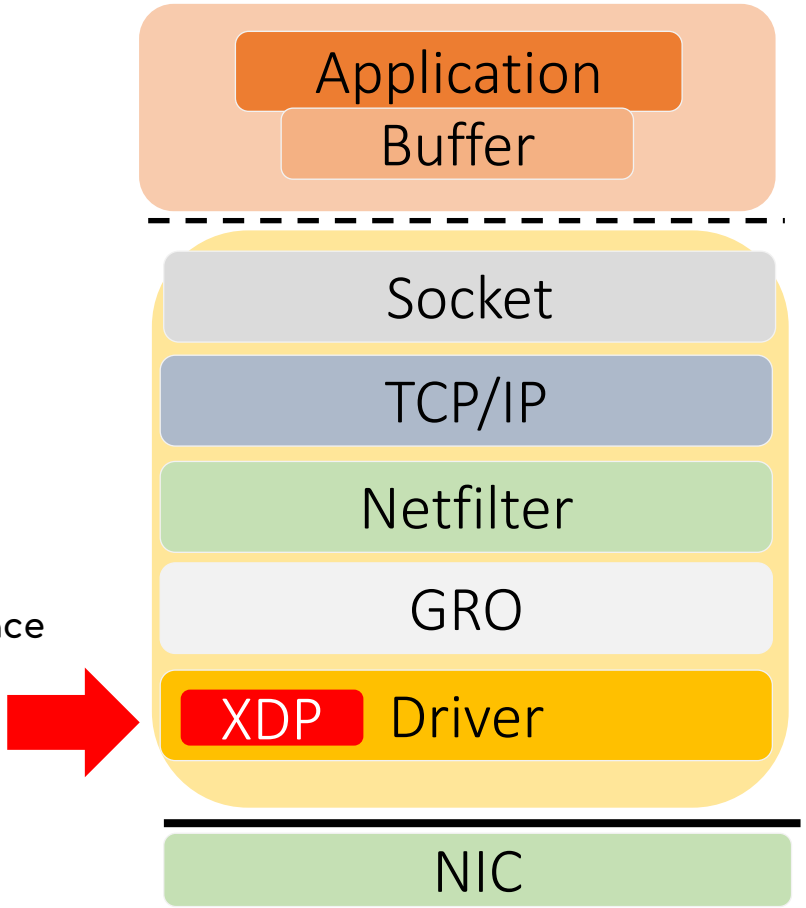
- XDP program supports three execution modes:

- XDP Native:
  - Requires specific support for the driver, which has to adhere with the XDP memory model
- XDP Generic:
  - Mainly intended for testing XDP programs in case driver does not have support
- Offloaded XDP



XDP Hook – return values

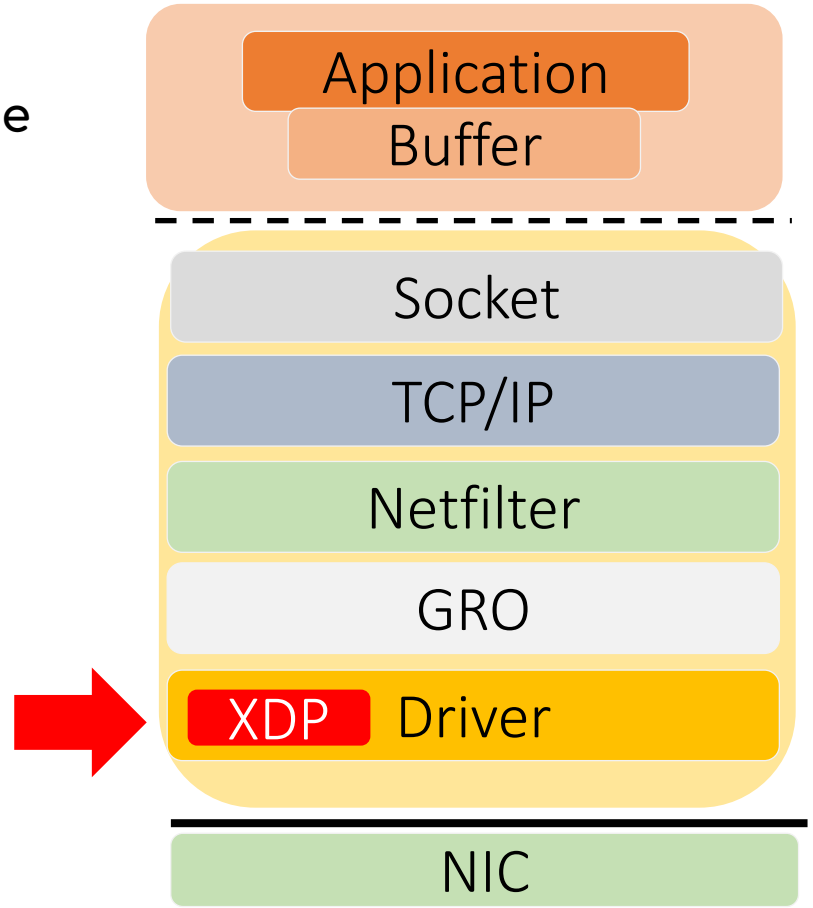
- The return value indicates what action the kernel should take with the packet, the following values are permitted:
- XDP_ABORTED/XDP_DROP
  - Both will drop the packet
- XDP_PASS
  - Allow the packet to continue up to the kernel networking stack
- XDP_TX
  - Retransmit the packet out of the same interface it was received on
- XDP_REDIRECT
  - Transmit the packet out of another interface



XDP Hook – context

- The context of an XDP program is the struct xdp_md object

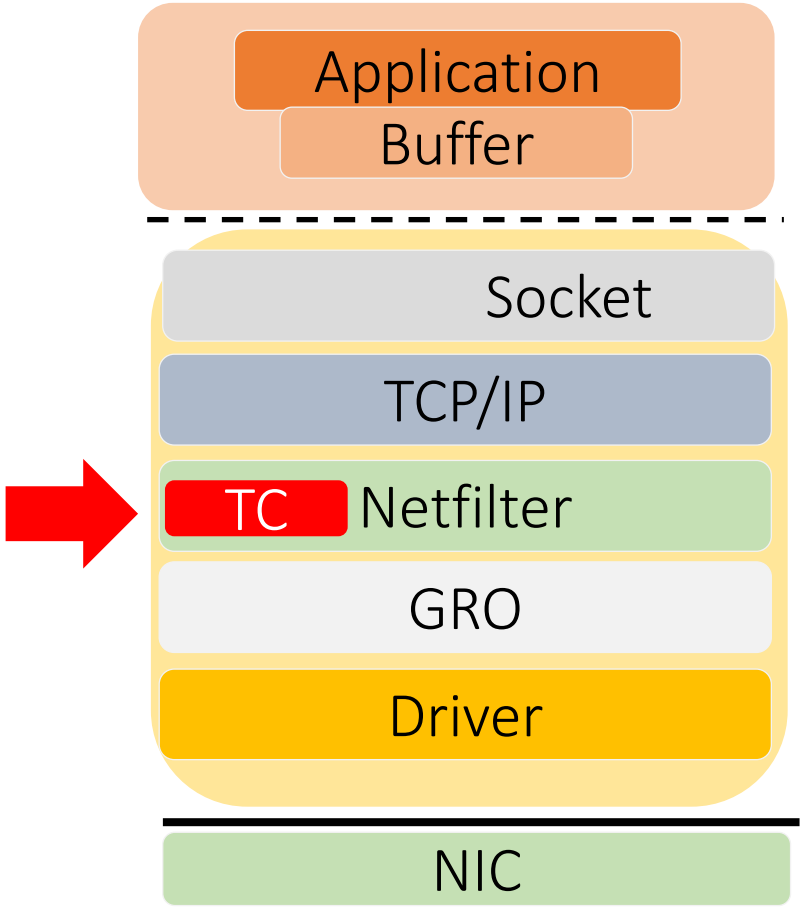
```
struct xdp_md {
__u32 data;
__u32 data_end;
__u32 data_meta;
__u32 ingress_ifindex;
__u32 rx_queue_index;
};
```



Traffic Control (TC) hook

- tc hook is located in the Traffic Control subsystem that helps in policing, classifying, shaping, and scheduling network traffic.

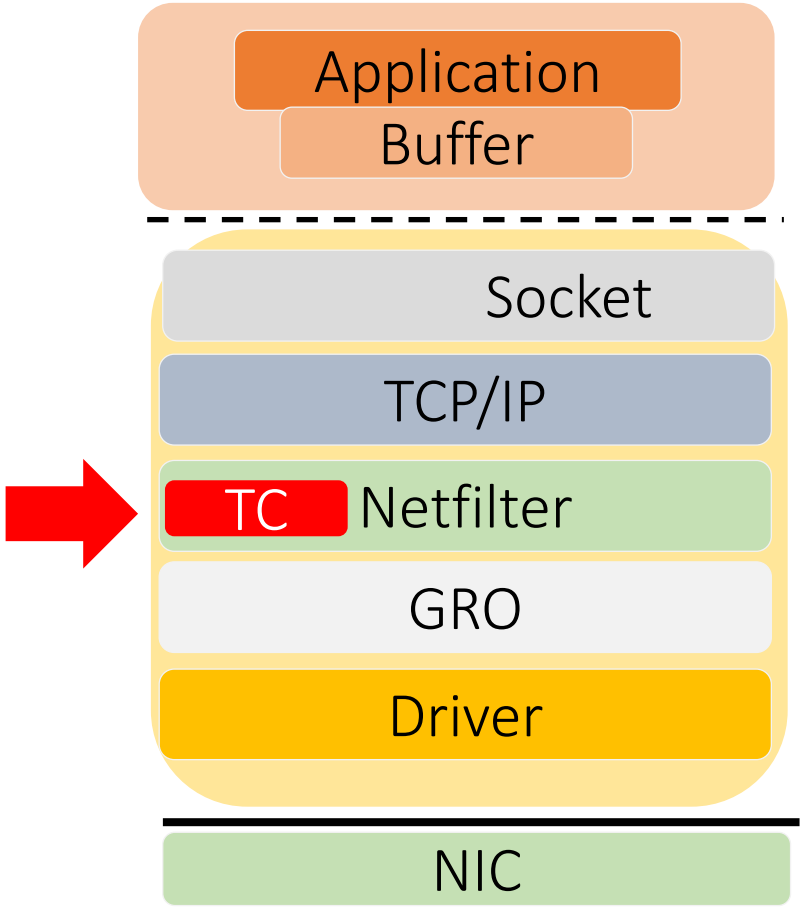
- An eBPF program attached to the tc hook has access to the sk_buff structure and can change things such as:
  - Priority
  - Queue mapping
  - Classid (to define a queue discipline)



Traffic Control (TC) hook - execution modes

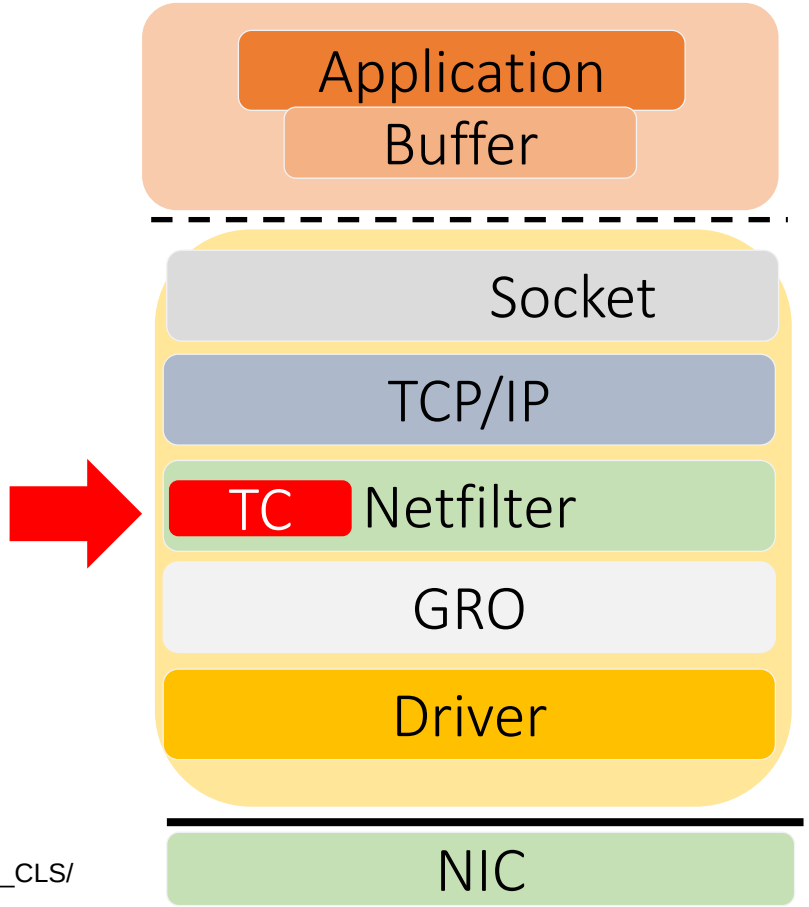
- Regular classifier
  - By default, when a BPF classifier is attached to a qdisc it will act as any other classifier
  - A return value of -1 indicates the default class should be picked, a return value of 0 means the filter did not match

- Direct actions:
  - eBPF programs are typically used for direct actions
  - When attached in direct action mode, the eBPF program will act as both a classifier and an action



Traffic Control (TC) hook – return values

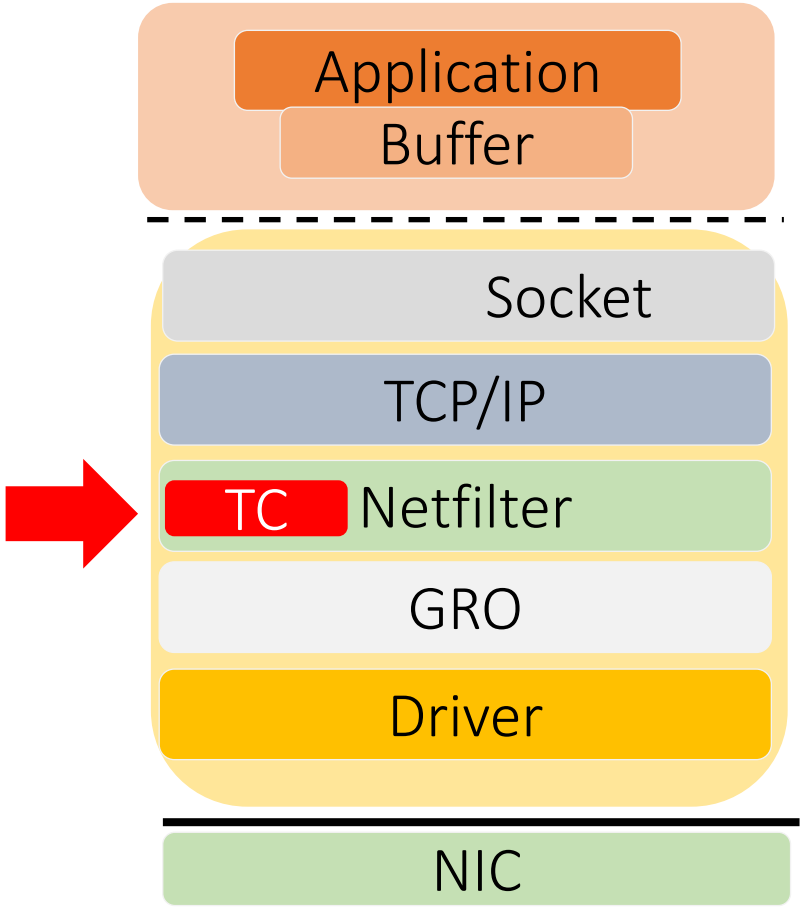
- TC_ACT_UNSPEC (-1)
  - Signals that the default configured action should be taken.
- TC_ACT_OK (0)
  - Signals that the packet should proceed.
- TC_ACT_SHOT (2)
  - Signals that the packet should be dropped, no other TC processing should happen.
- TC_ACT_REDIRECT (7)
  - Signals that the packet should be redirected



[1] https://ebpf-docs.dylanreimerink.nl/linux/program-type/BPF_PROG_TYPE_SCHED_CLS/

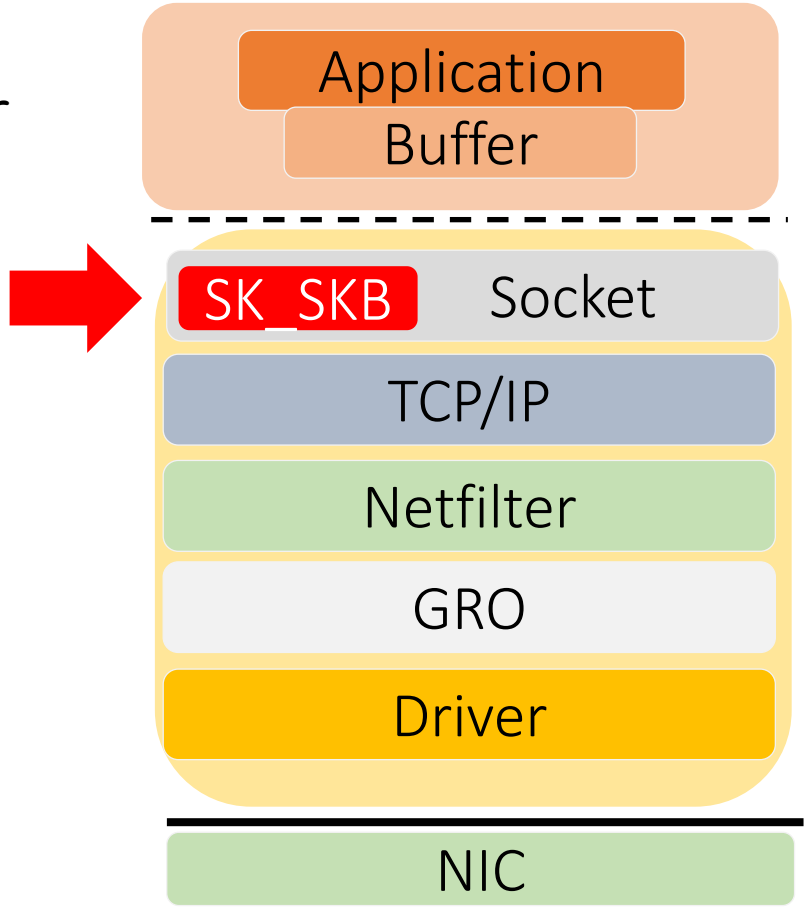
Traffic Control (TC) hook – context

- The TC CLS program is called by the kernel with a __sk_buff context
- The struct __sk_buff is a "mirror" of the struct sk_buff program type which is actually used by the kernel.
- Accesses to the struct __sk_buff pointer are seamlessly transformed into accesses into the real socket buffer
  - This indirection exists to provide a stable ABI for programs



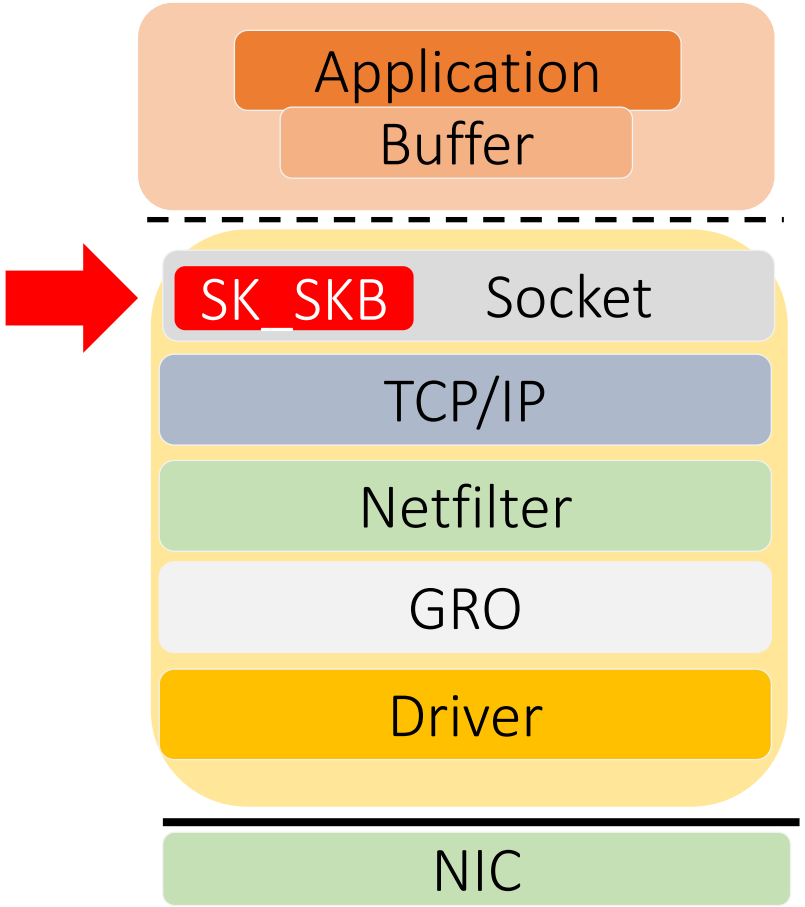
Socket hook (SK_SKB)

- SK_SKB is located at the socket layer after the TCP/IP processing
- Consequence: eBPF programs attached to this hook access to segments (not packets)
- Socket SKB programs are called on L4 data streams to parse L7 messages and/or to determine if the L4/L7 messages should be allowed, blocked or redirected.



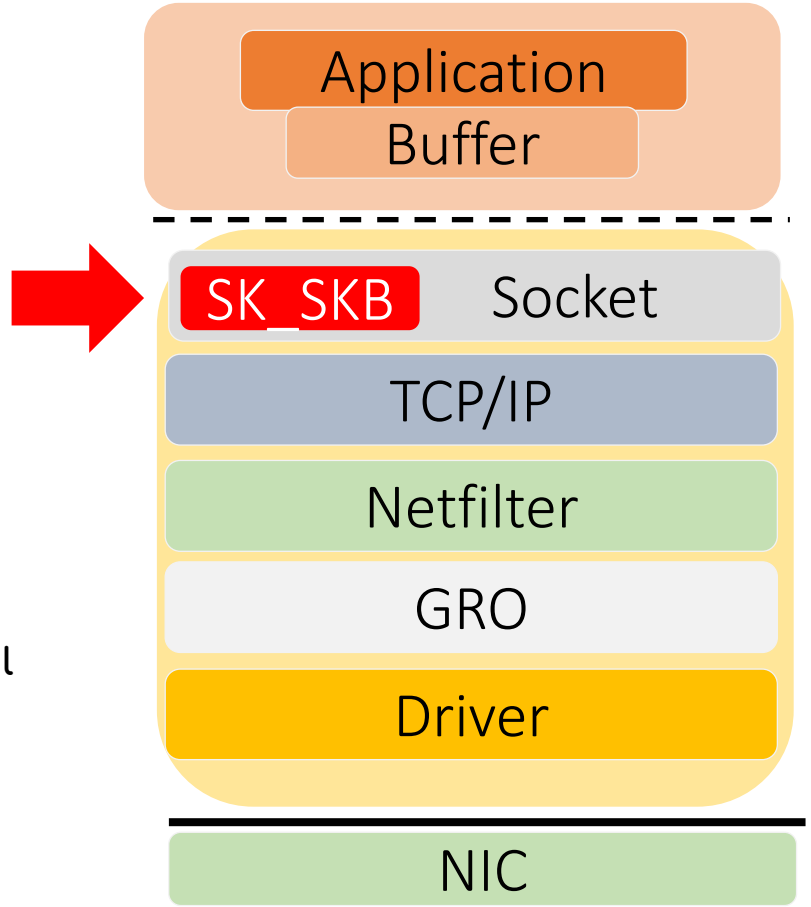
Socket hook (SK_SKB) – Attachment types (1)

- Socket SKB programs are attached to SOCKMAP or SOCKHASH maps
- Are invoked when messages get received on the sockets which are part of the map the program is attached to.
- The exact purpose of the program differs depending on its attach type.



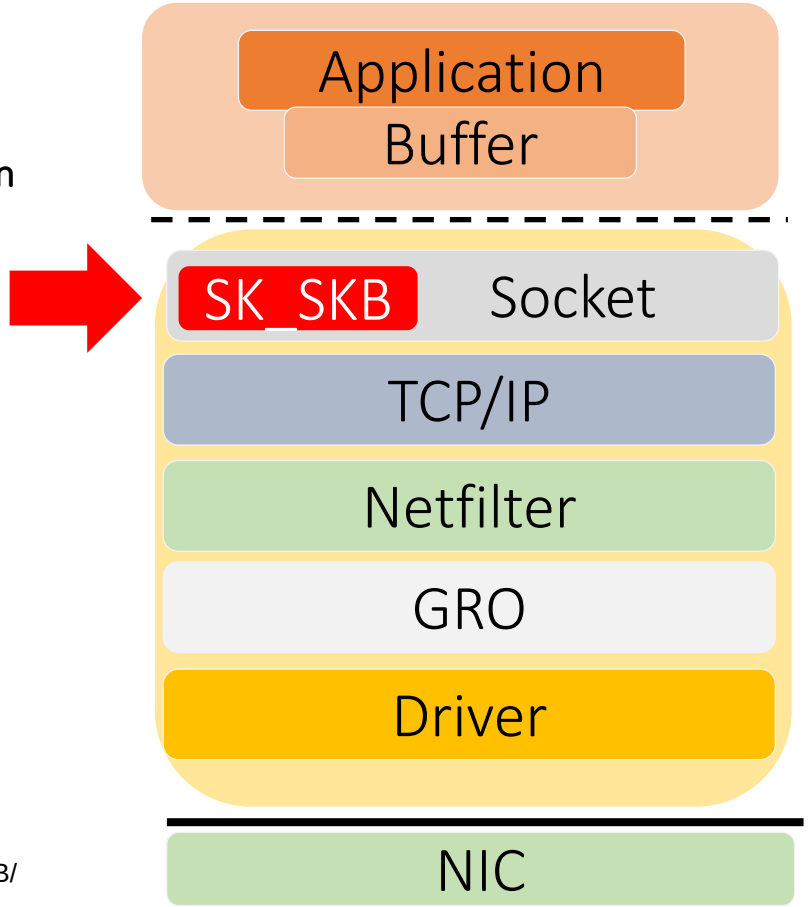
Socket hook (SK_SKB) – Attachment types (2)

- Stream parser program:
  - The job of the program is to parse the L7 data/packet and to tell the kernel how long the L7 message is.
- The return value is interpreted as follows:
  - >0 - indicates length of successfully parsed message
  - 0 - indicates more data must be received to parse the message
  - -ESTRPIPE - current message should not be processed by the kernel, return control of the socket to userspace
  - other < 0 - Error in parsing, give control back to userspace



Socket hook (SK_SKB) – Attachment types (3)

- Verdict program:
  - When this attach type is used the program acts as a filter, comparable to TC or XDP programs.
- The program gets called for every message indicated by the parser and returns a verdict
  - SK_PASS - The message may pass to the socket or it has been redirected with a helper.
  - SK_DROP - The message should be dropped.



[1] https://ebpf-docs.dylanreimerink.nl/linux/program-type/BPF_PROG_TYPE_SK_SKB/