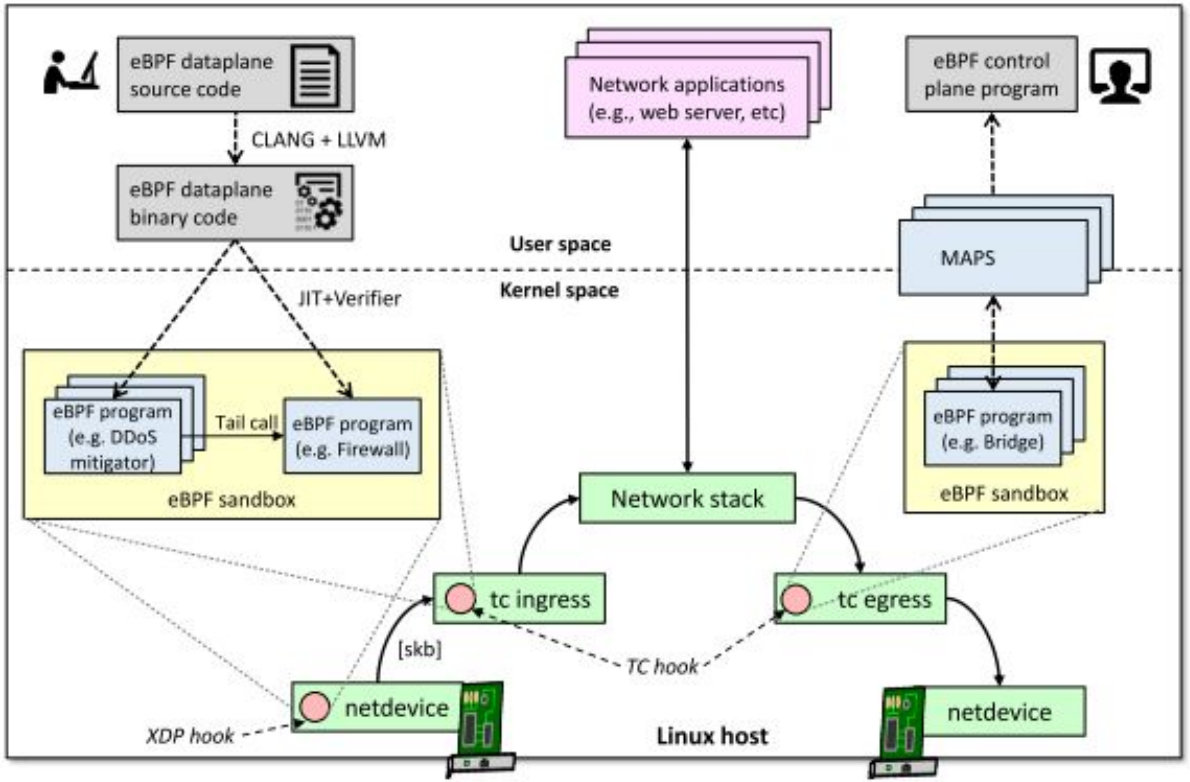
XDP: the eXpress Data Path

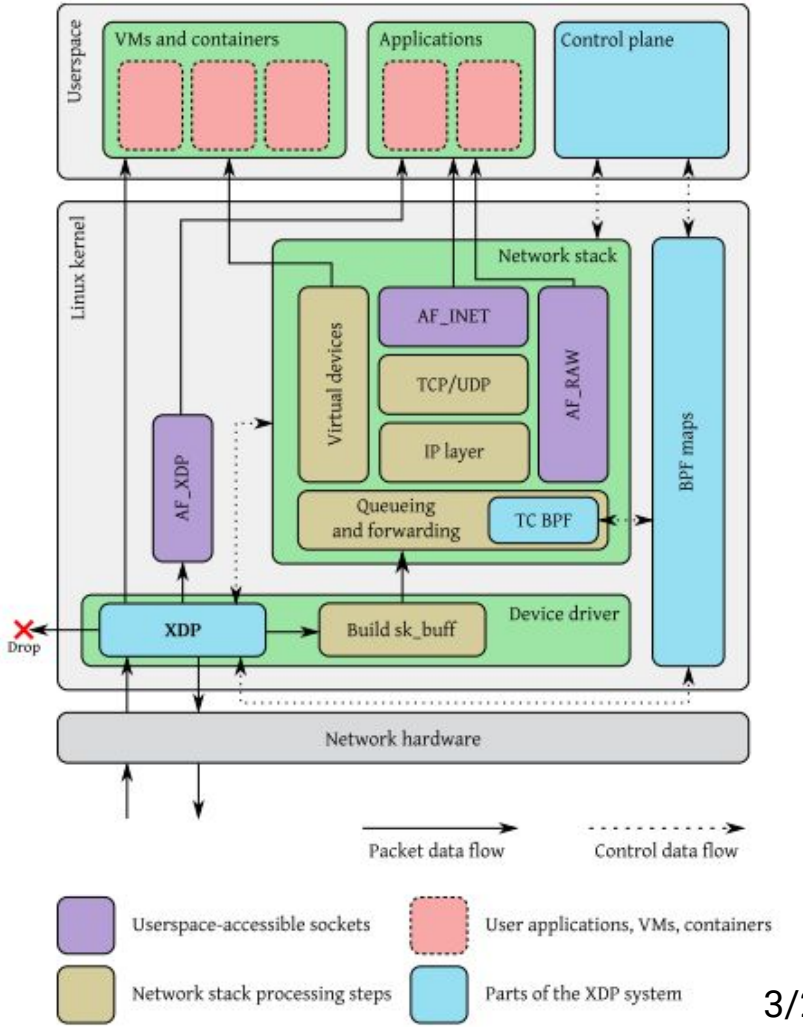
Alessandra Fais

PhD Student at University of Pisa

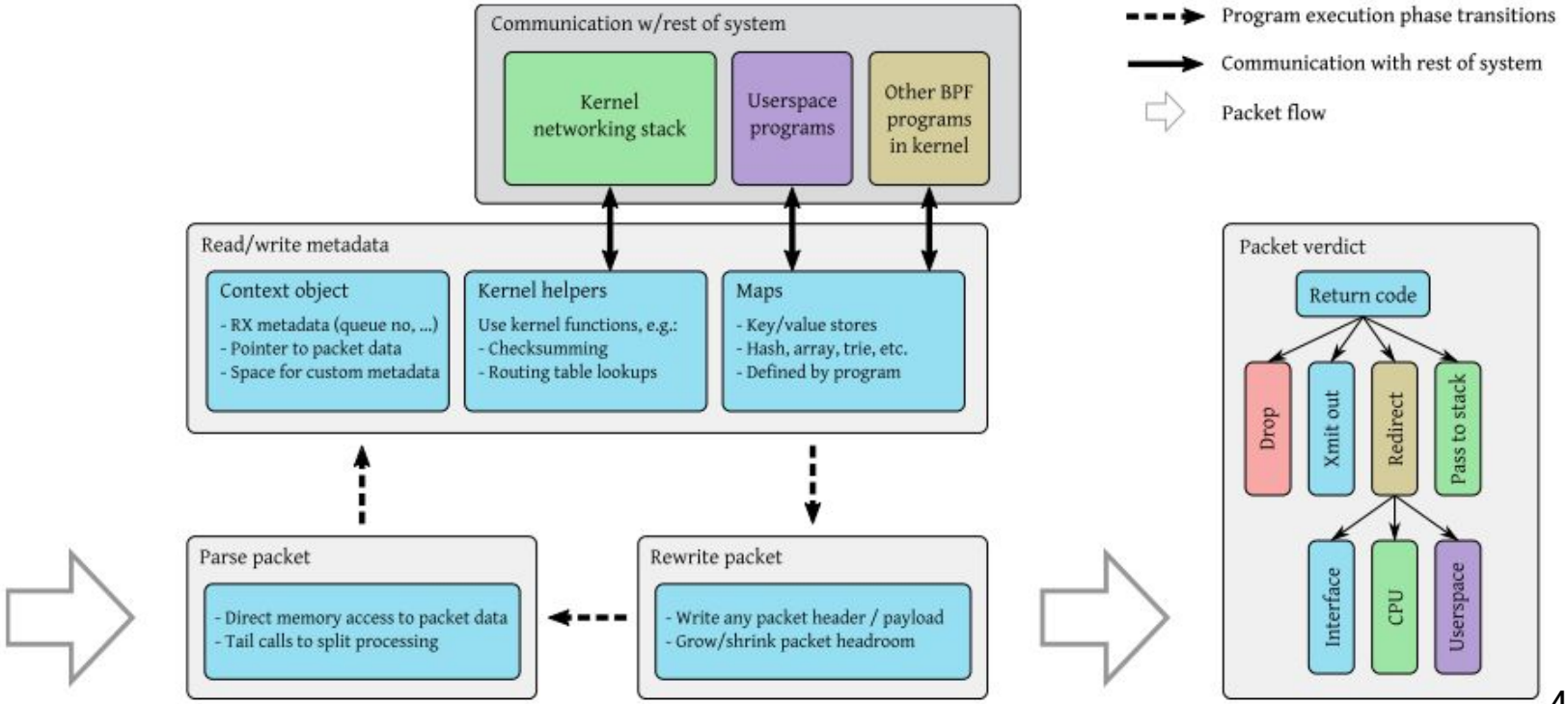
eBPF: overview



XDP's integration with the Linux network stack

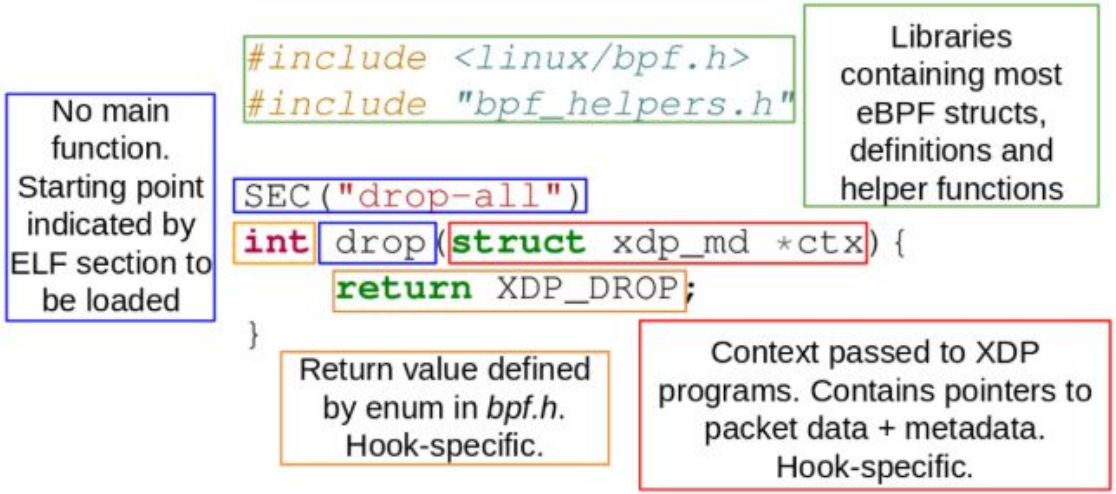


Execution Flow of a typical XDP program



XDP program: structure and characteristics

- Can be attached to a specific network interface
- Run inside the kernel
- The kernel calls the provided function for every packet received on the specified NIC (event processing)
- Function signature:
  `int xdp_function(struct xdp_md *ctx)`



XDP program: the input context

```
struct xdp_md {  /*Used inside the function to access the packet's content.*/
    __u32 data;  /*Contains a pointer to the beginning of the packet being processed.*/
    __u32 data_end;  /*Contains a pointer to the end+1 of the packet being processed.*/
    __u32 data_meta;  /*Holds the address of a memory area to exchange packet metadata*/
    __u32 ingress_ifindex;  /*The interface that received the pkt (rxq->dev->ifindex)*/
    __u32 rx_queue_index;  /*The RX queue (rxq->queue_index)*/
}
```

XDP program: the action set

- Actions are specified as a program return code
- Stored at register R0 right before the eBPF program exits
- This defines how the packet must be handled by the kernel

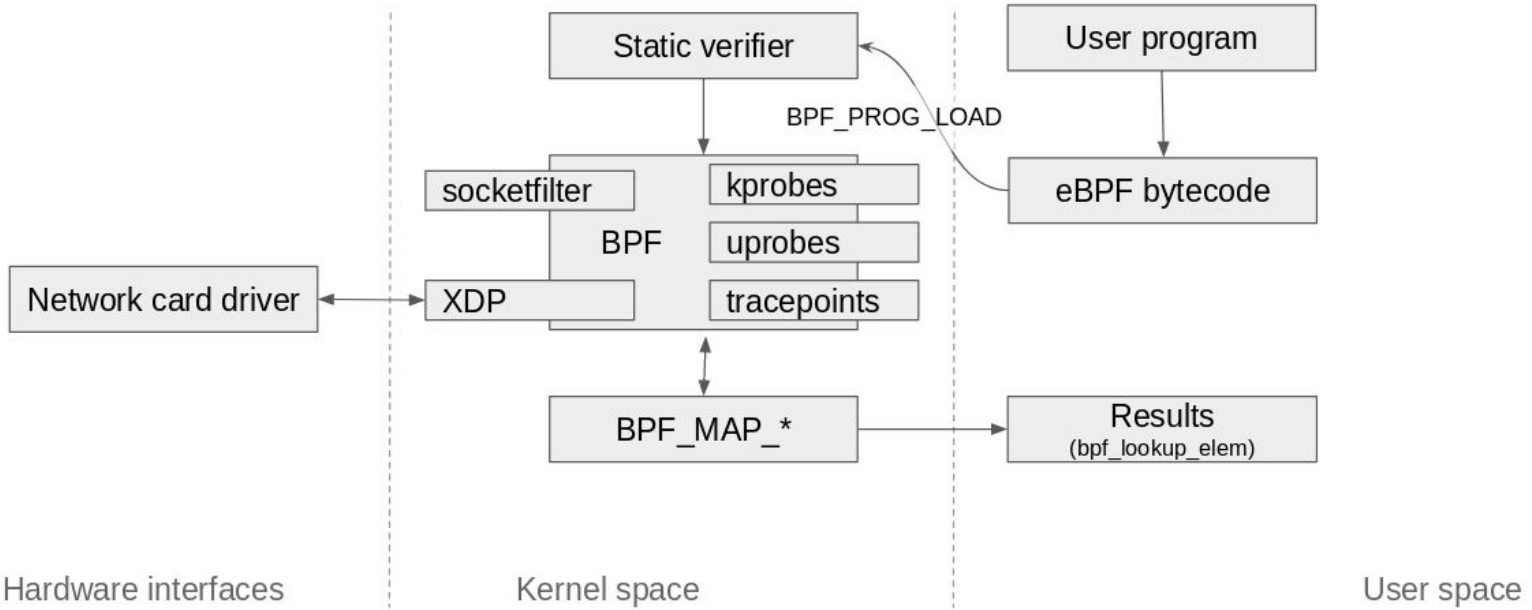
Value	Action	Description
0	XDP_ABORTED	Error. Drop packet.
1	XDP_DROP	Drop packet.
2	XDP_PASS	Allow further processing by the kernel stack.
3	XDP_TX	Transmit from the interface it came from.
4	XDP_REDIRECT	Transmit packet from another interface.

XDP program: required checks

- Explicitly check packets' boundaries
    - Access in the interval [data, data_end[
- Assume we want to read the Ethernet frame of the packet

```
/* Casts are needed to retrieve pointer type safely */
void *data_end = (void *)(long)ctx->data_end;
void *data     = (void *)(long)ctx->data;
struct ethhdr *eth = data;
    if ((void *)(eth + 1) <= data_end) {
        /* We can now access the Ethernet frame through eth. */
```

Loading an XDP program



Loading an XDP program: user program

1. Load XDP program in kernel memory:

```
int bpf_prog_load_xattr(
    const struct bpf_prog_load_attr *attr,
    struct bpf_object **pobj, /* Output parameter, used by successive calls. */
    int *prog_fd /* Output parameter, used by successive calls. */
);

struct bpf_prog_load_attr {
    const char *file; /* Name of the XDP program object file. */
    enum bpf_prog_type prog_type; /* In this case BPF_PROG_TYPE_XDP */
};
```

Loading an XDP program: user program

2. Retrieve interface index from its name:

```
int if_nametoindex(const char *if_name); /*    Returns 0 on failure. */
```

3. Attach XDP function to the NIC:

```
int bpf_set_link_xdp_fd(
    int if_index, /* Obtained from if_nametoindex. */
    int prog_fd,  /* Obtained from bpf_prog_load_xattr. */
    u32 flags     /* Additional advanced settings. */
);
```

Loading an XDP program: user program

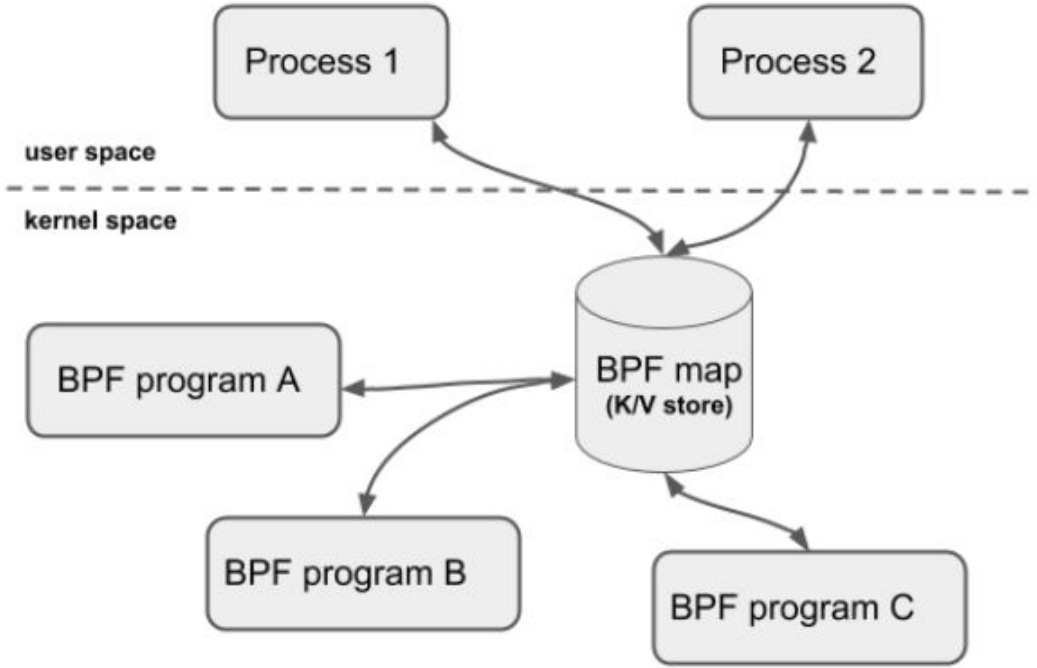
4. Detach an XDP function from an interface: call attach function `bpf_set_link_xdp_fd()` with `prog_fd = -1`

```
bpf_set_link_xdp_fd(if_index, -1, 0);
```

5. Common practice is to do the detach inside a signal handler for `SIGINT` and `SIGTERM`

BPF maps

- Data structures (generic key-value stores) available to eBPF programs
- Allow communication between kernel-space and user-space programs and between different eBPF programs



BPF maps

They can be used in a lot of ways:

- Share packets/flows statistics to user-space (without having the user-space program to access packets)
- Maintain state between successive executions of the same eBPF/XDP program
- Provide state/behavior to the kernel code from the user-space program

BPF maps: creation

- BPF maps can be created inside the XDP kernel code in the following way:

```
struct bpf_map_def SEC("maps") map_name = {
    .type = BPF_MAP_TYPE_HASH, /*Type of the data structure (e.g. hash table, array).*/
    .key_size = sizeof(__u32), /*Size in bytes of the keys used to access the map.*/
    .value_size = sizeof(struct entry), /*Size in bytes of the values in the map.*/
    .max_entries = 100,
};
```

- The `struct` being used as value is usually defined in a header file shared between the kernel code and user-space program

BPF maps: access from kernel-space

- Retrieve elements:

```
*void bpf_map_lookup_elem(struct bpf_map_def *bpf_map, key_t *key)
```

- Sample code:

```
value_t *value;
value = bpf_map_lookup_elem(&bpf_map, &key);
if (!value) {
     /*value == NULL, key not present in the BPF map.*/
} else {
     /*value != NULL, key present in the BPF map. It can be accessed through value.*/
     *value = /* Update the value or do something else. */
}
```

BPF maps: access from kernel-space

- Create/update elements using:

```
int bpf_map_update_elem(
        struct bpf_map_def *bpf_map,
        key_t *key,
        value_t *new_value,
        __u64 flags
)
```

- The flags that can be specified are:
  - BPF_ANY   /* Create new element or update existing. */
  - BPF_NOEXIST /* Create new element if it didn't exist. */
  - BPF_EXIST   /* Update existing element. */

BPF maps: access from user-space

1. Retrieve an internal data structure needed for the successive call:

```
struct bpf_map *map = bpf_map__next(NULL, obj);
```

2. Retrieve the BPF map file descriptor needed to interact with it:

```
int map_fd = bpf_map__fd(map);
```

3. Obtain a value from the BPF map using:

```
int bpf_map_lookup_elem(int map_fd, key_t *key, value_t *value, &value);
```

```
value_t value;
int error = bpf_map_lookup_elem(map_fd, &my_key, &value);
/* If no error, value is updated. */
```

AF_XDP

- Address family optimized for high performance packet processing
- AF_XDP sockets enable the possibility for XDP programs to redirect frames to a memory buffer in a user-space application
    - Use the XDP_REDIRECT action from an XDP program, the program can redirect ingress frames to other XDP enabled netdevs, using the bpf_redirect_map() function

Bibliography

- **The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel**
  Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, David Miller, et al.
  CoNEXT '18: Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, December 2018, Pages 54–66, https://doi.org/10.1145/3281411.3281443
- **Creating Complex Network Services with eBPF: Experience and Lessons Learned**
  Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, Mauricio Vásquez Bernal
  IEEE International Conference on High Performance Switching and Routing, June 2018, https://doi.org/10.1109/HPSR.2018.8850758
- **Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications**
  Marcos Augusto M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Luiz Vieira, et al.
  ACM Computing Surveys 53(1):1-36, February 2020, https://doi.org/10.1145/3371038
- **AF_XDP: the Linux Networking Documentation**
  https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html
- **Cilium: BPF and XDP Reference Guide**
  https://docs.cilium.io/en/latest/bpf/
- **Load XDP programs using the ip (iproute2) command**
  https://medium.com/@fntlnz/load-xdp-programs-using-the-ip-iproute2-command-502043898263