XDP (eXpress Data Path)

Stefano Duo

XDP programs

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

```
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. */
}
```

- The function must return one of the `XDP_*` macros to tell the kernel how to handle the packet (e.g., `XDP_DROP` to drop the packet)

XDP programs

- Since they run in kernel space, they must not crash (otherwise the whole system crashes)
- We have to explicitly check that every location we access resides in `[data, data_end[`
- During attachment to an interface, a verifier checks that those checks are indeed performed
- Assume want to read the Ethernet frame of the packet:

```c
/* 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. */
```

Running an XDP program to an interface

- To accomplish that we use an additional program which runs in userspace
- There are a number of steps and they are a little convoluted

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; /* For us always BPF_PROG_TYPE_XDP (there are many more). */
};
```

Running an XDP program to an interface

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. */
);
```

Running an XDP program to an interface

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` (see Github repository's code linked later)

BPF maps

- Right now we can only take decisions using the packets’ content as information
- BPF maps are data structures shared between the kernel code and userspace program
- They can be used in a lot of ways:
  - Share packets/flows statistics to userspace (without having the userspace program access packets)
  - Maintain state between successive calls to out kernel XDP function
  - Provide state/behavior to the kernel code from the userspace program
  - Pass/copy packets to userspace before they traverse the kernel networking stack (see AF_XDP address family and do not reimplement the wheel)

Creating BPF maps

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

```
struct bpf_map_def SEC("maps") rx_flows = {
        .type = BPF_MAP_TYPE_HASH, /* Type of the map data structure, e.g. hash table, array, etc. */
        .key_size = sizeof(__u32), /* Size in bytes of the keys being used to access the map. */
        .value_size = sizeof(struct entry), /* Size in bytes of the values contained 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 userspace program

Accessing BPF maps from kernel code

- Retrieve elements using

```
*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. */
}
```

Accessing BPF maps from kernel code

- 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. */`

Accessing a BPF map from userspace

- As for loading and running an XDP program, there are a few steps needed before we can interact with a BPF map

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. */
   ```

Development environment used

- The code has been tested on Ubuntu 20.04 LTS
- Required packages:
  - `sudo apt-get install -y make gcc libssl-dev bc libcap-dev gcc-multilib libncurses5-dev pkg-config libmnl-dev graphviz bison clang flex libelf-dev llvm`
- The repository containing the code can be found at https://github.com/StefanoDuo/bpf_stuff
  - It contains an additional folder `bcc` which uses a different framework to write XDP programs
  - The kernel code remains the same except for the interaction with the BPF maps
  - The userspace can be written in python bindings and offers much easier program loading and interaction
  - The only difficulty comes from the types and how the C ⇔ Python conversion is handled (which I am not familiar with)
  - More info about that can be found at https://github.com/iovisor/bcc

Compiling an XDP program

- Download the linux-5.6 kernel source code: https://github.com/torvalds/linux/releases/tag/v5.6
- Unzip it and from inside the directory run
  - `make headers_install`
  - `make defconfig`
- Go inside `samples/bpf`
- Try to run `make`, everything should compile
- Copy your kernel and userspace program file there
- Add the following lines to the `Makefile` file (`prog_name`: your program name):
  - `progs-y += prog_name                  # Name of the executable that will be created.`
  - `prog_name-objs := prog_name_user.o # Userspace object file.`
  - `always-y += prog_name_kern.o        # Kernel object file.`
- Now running `make` should compile your program as well