User Guide

1. **Performance tool**
    1. Install BCC tool since it is used for compiling the BPF program.
       Reference link: https://github.com/iovisor/bcc/blob/master/INSTALL.md
       Please follow the aforementioned guide to install BCC, and alongside any other dependencies related to the kernel version.
    2. Execute (tracepkt.py) using: ***sudo python tracepkt.py***
       The program execution will internally compile the BPF program (tracepkt.c), which attaches tracepoints on various networking functions.

2. **Fast data path**

    Here, we are attaching our BPF code on Traffic Control (TC) hooks.

    1. Now we need to first add a queueing discipline on the interfaces where we intend to attach our BPF object file using a 'classifier' and 'action' section (clsact)
    sudo tc qdisc add dev <target interface> clsact
    This needs to be done on each interface where we want to attach our BPF code.
    2. . Now, we will attach our BPF code on the interface. This can be done on the ingress / egress hook of the interface.
    sudo tc filter add dev <target interface> <ingress/egress> bpf direct-action obj bpf_filter_map.o

Developer Guide

1. **Performance tool**
    To know the architecture and low-level code information, refer to the milestone document.
    In order to run the code, refer to the user guide.
    Files of interest:
    tracepkt.c - Includes the logic that needs to be executed when a networking function to which the tracepoint is attached gets hit.
    tracepkt.py - Gives a call to the BPF program using the BPF functions, which then internally compiles the BPF program in 'tracepkt.c' since 'tracepkt.c' is passed to this BPF function. This is the user space code that later receives the data from the kernel space in a perf buffer, and then analysis it.

2. **Fast data path**

    To know the architecture and low-level code information, refer to the milestone document.

1. Compile the user space code, and make an application binary for the same. In order to do that, we had to first compile a github repo named **prototype kernel**

(https://github.com/netoptimizer/prototype-kernel**).** These were the changes that we had to do in order to compile prototype kernel successfully. We hopethat this proves out to be pretty helpful to you.

---

1. Commented shift_mac_4bytes_16bit() in xdp_vlan01_kern.c
2. Installed libpcap-dev, libpcap0.8, libpcap0.8-dev in order to support #include <pcap/pcap.h>
3. Took dlt.h from internet (libpcap/pcap/dlt.h) and created a new file in /usr/include/pcap
4. Took if_link.h file from internet and put it in prototype-kernel/kernel/samples/bpf/kernel/include/uapi/linux AND prototype-kernel/kernel/samples/bpf/kernel-usr-include/linux
5. a)

Added

```
enum nlmsgerr_attrs {
   NLMSGERR_ATTR_UNUSED,
   NLMSGERR_ATTR_MSG,
        NLMSGERR_ATTR_OFFS,
   NLMSGERR_ATTR_COOKIE,
   __NLMSGERR_ATTR_MAX,
   NLMSGERR_ATTR_MAX
};
```

b)

```
/* Flags for ACK message */
#define NLM_F_CAPPED            0x100  /* request was capped */
#define NLM_F_ACK_TLVS          0x200  /* extended ACK TVLs were included */

#define NETLINK_EXT_ACK                 11
```

in nlattr.h (prototype-kernel/kernel/samples/bpf/tools.lib.bpf)

You might have to do something less or additional depending on the kernel version etc.

Post this in order to make the application binary, please execute

gcc -O2 -g bpf_load.o ./tools/lib/bpf/libbpf.a -lelf -I./tools/lib/ -I./kernel-usr-include/ bpf_map_create_user.c -o bpf_map_create

Kindly keep the user level code in the **prototype kernel** repo (/prototype-kernel/kernel/samples/bpf/)

bpf_map_create - This is the application binary that gets generated after the aforementioned compilation. We call this application from a python script (**runtime_config.py**) that uses **configuration.json** and **runtimeConfiguration.json**. Using the data from the json, we give a call to this application passing relevant parameters as arguments, which is then used by the application to populate the BPF maps entries in the user space. It is these entries that are later used by the kernel in order to make forwarding decisions.

2. Compile the ‘C’ code using clang, thereby producing an object file. In our case, the file is **bpf_filter_map.c**

bpf_filter_map.c - This is the kernel space code that uses the BPF maps populated by the user space code and takes appropriate decision for redirection using the data received from the map. It is basically policy based routing here wherein we consider different fields from the map and take a routing decision. In order to know the details of the implementation, please refer to the milestone document and the source code.

sudo clang -O2 -target bpf -o bpf_filter_map.o -c bpf_filter_map.c

3. To attach the code on TC hooks, refer the user guide.