POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Final Project on eBPF

Network Computing Course

A.Y. 2023/2024

Sebastiano Miano, Gianni Antichi

Contents

1 | Environment Setup

Note: If you have already downloaded the VM and set it up following the instructions for the other eBPF labs, you can skip this chapter.

In this project, we utilize the *libbpf* library for building eBPF programs, as we did for the lab exercises.

To simplify the installation process, we provide a VirtualBox[1] Virtual Machine (VM) with all the necessary software for the project pre-installed.

1.1. Download Lab VM

Running the project in a VM helps maintain separation from the rest of the system. Moreover, since all the programs we will use are only available on Linux, users of other OSes can run them in a Linux VM.

Download the VirtualBox `ebpf-labs.ova` image at the following URL:

https://networkcomputingpolimi.page.link/labs

1.1.1. Install VirtualBox

Before executing the VM, you need to install VirtualBox, which is compatible with Windows, Linux, and macOS. Find the VirtualBox binaries at the following URL:

https://www.virtualbox.org/wiki/Downloads

After installation, open the `ebpf-labs.ova` file; the VM will boot and initiate the project environment.

1.1.2. Log into the Project VM

When the VM starts, log in using the following credentials:

[1] `https://www.virtualbox.org/`

- **Username**: ebpf
- **Password**: ebpf

1.1.3. Clone the Project Repository

After logging into the VM, clone the repository[2] containing all the lab exercises, and the project folder.

Open the terminal by clicking on the `Terminal` icon on the left bar, and enter the following command:

```
$ git clone https://github.com/Polimi-NetClasses/058172-network-
    computing-labs.git --recurse-submodules
```

1.1.4. Navigate to the Project Folder

After cloning the repository, navigate to the folder related to this project using these commands:

```
$ cd 058172-network-computing-labs
$ cd ebpf-labs && cd project
```

[2] https://github.com/Polimi-NetClasses/058172-network-computing-labs

2 | Heavy Hitter Detection

In this project, we ask you to implement a heavy hitter detector that utilizes a **counting bloom filter** for detecting heavy hitters.

Heavy hitters are traffic sources that generate abnormally large traffic and can be classified by either the source IP address (as in the lab classes) or the application session that transmits the traffic (as in this project). To implement a bloom filter, as discussed in the lectures, we require multiple hash functions.

Figure 2.1 shows the network topology for this exercise.

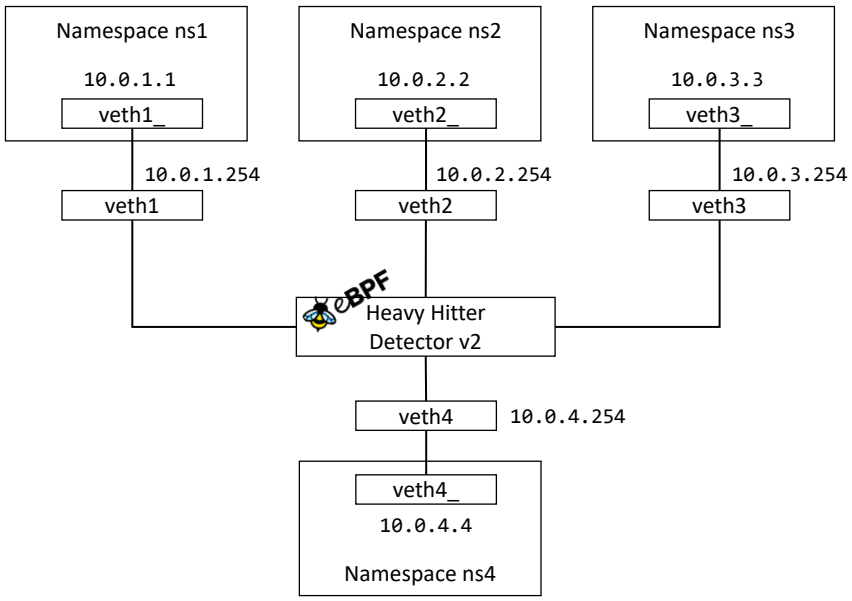

Figure 2.1: Network setup for the final project.

The XDP program should exhibit the following behavior:

- It receives packets from four different namespaces, specifically `ns1`, `ns2`, `ns3`, and `ns4`, and redirects all packets to one of the other namespaces based on the destination IP address. This XDP program will employ two *BPF_HASH* maps to *(i)* determine where to redirect the packet based on the destination IP address and

(ii) ascertain the source MAC address to use when forwarding the packet to the destination. It will also use a *BPF_DEVMAP* to redirect packets to the correct `ifindex` based on its destination port, and another *BPF_ARRAY* map used for the *Bloom Filter*.

- It parses the TCP and UDP headers of the packet and conducts heavy hitter detection on those flows, according to a threshold set by the user space program during startup (by default, the threshold equals 50).
- If the number of packets received for a specific flow (5-tuple) exceeds the threshold, it will drop all subsequent packets.
- All the packets that do not contain a valid header should be dropped (e.g., the IP protocol field is set to TCP, but the header is invalid).
- All the ARP packets should not go through the bloom filter, and the XDP program should return directly `XDP_PASS`.
- All the IPv4 packets that do not contain a TCP or UDP header (e.g., ICMP) should be forwarded based on their destination IP.

Note: Ensure you execute the following command before starting this project:

```
$ sudo apt update && sudo apt install libyaml-dev libnl-3-dev -y
```

2.1. Topology

You can setup the topology by running the following bash script:

```
$ sudo ./create-topo.sh
```

You can inspect the newly created interface inside the namespaces by running the following command:

```
$ sudo ip netns exec ns1 ifconfig
```

Note that the `create-topo.sh` bash script uses a program called `xdp_loader` to load a *dummy* program on the namespace side. This program is needed when performing a redirect from eBPF since the redirect needs an XDP program to be loaded on both sides of the virtual interfaces.

2.2. Project folder structure

Inside the `project` folder, we provide some files to assist you in completing the project.

- `ebpf/hhd_v2.bpf.c`: This source file is where the XDP program will be implemented. It is **incomplete** and should be filled with the code necessary to implement this exercise. Further information are available at 2.3.
- `create-topo.sh`: This BASH script sets up the topology described before. The script creates all the virtual interfaces, the namespaces, and sets up the network accordingly. The script is **complete**, and you don't need to modify anything.
- `hhd_v2.c`: This C code is executed in userspace and is responsible for loading and attaching the eBPF program. This program also retrieves some information about the network configuration and sets the appropriate variables and maps in the eBPF program. In particular, it fills the `ipv4_lookup_map` with the information about the destination port to be used for a given IP and its destination MAC address. It also fills the `src_mac_map` with the information about the source MAC address to use when redirecting a packet to a given port. This file is **complete**, and you don't need to modify anything.
- `config.yaml`: This YAML configuration file is read by the `hhd_v2.c` program at startup. The file contains the configuration parameters that should be used in the exercise. In particular, for every namespace it contains the *(i)* IP address to use for its internal interface (i.e., *vethN_*), *(ii)* the port number on the eBPF program used to indicate the interface for a given namespace, *(iii)* the destination MAC address to use when redirecting packets to the namespace, and finally *(iv)* the IP address of the default gateway for the namespace. The `hhd_v2.c` already contains the code to parse and validate the file, and you just need to use that information depending on your needs. This file is **complete**, and you don't need to modify anything.

2.3. Assignments

For this assignment, you will need to parse the IPv4 header of the packet, as well as the TCP or UDP header. The extracted 5-tuple (IP source, IP destination, source port, destination port, and protocol) will be used in the **counting bloom filter** to check if the flow is below the threshold.

The Heavy Hitter algorithm will be applied to packets coming from all the namespaces (i.e., `ns1`, `ns2`, `ns3` and `ns4`).

The threshold value can be found in the configuration structure already defined in the `hhd_v2.bpf.c` file. This struct is filled by the control plane before loading the program.

```
const volatile struct {
    __u64 threshold;
} hhd_v2_cfg = {};
```

2.3.1. Assignment #1: Return PASS on ARP packets (TODO 1)

After parsing the Ethernet header, you need to check if the frame contains an ARP packet. If this is the case, the XDP program should directly return `XDP_PASS`.

2.3.2. Assignment #2: Parse IPv4 header (TODO 2-4)

After parsing the Ethernet header, you need to parse the corresponding IPv4 header (if the packet contains an IPv4 header, of course). To do this, you can implement the `parse_iphdr()` function that is already defined in the BPF source file.

The `struct iphdr` is the structure that defines the IPv4 header, and it is available in the ip.h header file.

The `parse_iphdr()` should return the next protocol type that is contained in the packet.

2.3.3. Assignment #3: Define structure for the 5-tuple (TODO 5-6)

After you implement the code necessary to parse the IPv4 header, you should create a C structure that will contain the information about the 5-tuple of the current packet. The structure should contain the IP source, IP destination, source port, destination port, and protocol, where every field contains the correct number of bits.

While parsing the packet, you should fill the structure with the data of the current packet.

2.3.4. Assignment #4: Check if the packet is TCP or UDP (TODO 7)

Depending on the next protocol type returned by the `parse_iphdr()` function, you should check if the packet contains a TCP or UDP header.

If the packet is neither TCP nor UDP, you can just redirect the packet by *jumping* to the `forward` statement.

2.3.5. Assignment #5: Parse TCP header (TODO 8-10)

If the next protocol type is TCP, you should parse the TCP header to extract the source and destination port. To do this, you can implement the `parse_tcphdr()` function that is already defined in the BPF source file.

The `struct tcphdr` is the structure that defines the TCP header format, and it is available in the tcp.h header file.

Note: The TCP header has a variable size, which depends on the *option* field. When you do the bounds check for the packet, you need to make sure you consider the entire size of the TCP header. This size can be calculated using the *Data offset (4 bits)* field of the TCP header[1], which specifies the size of the TCP header in 32-bit words (4 bytes). The minimum size header is 5 words and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes, allowing for up to 40 bytes of options in the header.

2.3.6. Assignment #6: Parse UDP header (TODO 11-12)

If the next protocol type is UDP, you should parse the UDP header to extract the source and destination port. To do this, you can implement the `parse_udphdr()` function that is already defined in the BPF source file.

The `struct udphdr` is the structure that defines the UDP header format, and it is available in the udp.h header file.

2.3.7. Assignment #7: Implement the heavy hitter detector using the bloom filter (TODO 13-14)

After you have completed the packet parsing, you should now apply the bloom filter algorithm based on the flow information that you extracted from the current packet. The Bloom Filter map is already defined in the BPF program, and it is an `ARRAY` map with a fixed size. The value of every entry in the map is a 64-bit counter that can be used to count the packets.

In order to implement the bloom filter, you need to use two different hash functions. In

[1] `https://en.wikipedia.org/wiki/Transmission_Control_Protocol`

this project, we should use two different hash functions, i.e., the `jhash` function, and the `fasthash`. The prototype of those functions is the following:

```
u32 jhash(const void *buf, u64 length, u32 seed);
u32 fasthash32(const void *buf, u64 len, u32 seed)
```

where the first parameter of both functions is the data to hash, the second parameter is the size of the data to hash, and the third parameter is the seed to use for the hash function. For the seed, you can use the `define` values `FASTHASH_SEED` and `JHASH_SEED`.

After you obtained the two hashes, you can use the normal C modulo operation with the `BLOOM_FILTER_ENTRIES` define to get the index on the BPF map of the element you want to read/modify.

Note: If the verifier complains, make sure you add a check that the two hashes are not greater then or equal to `BLOOM_FILTER_ENTRIES`.

Finally, after you get the to values from the bloom filter map, you can compare them with the threshold value set in the `hhd_v2_cfg` structure.

If both values are over the threshold, you should drop the packet; otherwise, you can let the packet pass and go to the `forward` statement.

2.3.8. Assignment #8: Forward the packet (TODO 15)

As part of this project, we already provide the code necessary to forward the packet to the correct destination. The only thing you need to do here is to copy the *destination* IP address (in network byte order) that you extracted from the packet into the `ipv4_lookup_map_key` variable, so that the rest of the code can find the right information about the destination of the packet.

2.4. Userspace program

In this project, the userspace program is already implemented, so you don't need to modify or implement anything on it.

2.5. Test the program

This section provides a set of tests that you can run to ensure your program works correctly.

2.5.1. Compile the project

First, compile the program using the following commands:

```
$ make clean
$ make
```

2.5.2. Start the topology

1. Start the topology by running this command:

   ```
   $ sudo ./create-topo.sh
   ```

2. Verify that all the namespaces are created correctly by running this command:

   ```
   $ sudo ip netns ls
   ```

   This should see that 4 namespaces `ns1`, `ns2`, `ns3`, and `ns4` have been created.

3. Now, run the userspace application (i.e., `hhd_v2`), which will load the BPF program in the kernel and attach it to the XDP hook of the *veth* interface linked to the `ns1`, `ns2`, `ns3`, and `ns4` namespaces.

   ```
   $ sudo ./hhd_v2
   ```

   By default, the threshold is set to 50, but you can also set a specific threshold (e.g., 100) using this command:

   ```
   $ sudo ./hhd_v2 -t 100
   ```

   If the program is executed correctly, it should print the message: `Successfully attached!`.

4. Keep the program open. Then, open a new terminal and run the tests described in the next section.

Note: You can check the debug messages printed with the `bpf_printk` call by running these commands. This is particularly useful when you are debugging your project.

```
$ sudo su
$ cat /sys/kernel/debug/tracing/trace_pipe
```

2.5.3. List of test

1. **Verify that you can *ping* every other namespace**.

   From all the namespaces, you should be able to ping the others.

   Ping `ns2` from `ns1`:

   ```
   $ sudo ip netns exec ns1 ping -c 1 10.0.2.2
   ```

   Ping `ns3` from `ns1`:

   ```
   $ sudo ip netns exec ns1 ping -c 1 10.0.3.3
   ```

   Ping `ns4` from `ns1`:

   ```
   $ sudo ip netns exec ns1 ping -c 1 10.0.4.4
   ```

   Repeat this set of operations from all the other namespaces.

2. **Check that Heavy Hitters are detected for TCP flows**

   If the heavy hitter algorithm that you implemented is correct, you should be able to send TCP flows towards a given destination until the threshold value is reached. If you set the threshold to 10, you can check the behavior of the program with the following command, which will send a set of TCP packets towards `ns4` with destination port `80`.

   ```
   $ sudo ip netns exec ns1 nping -c 15 --tcp -p 80 10.0.4.4
   ```

   Once the command finishes, you should see an output similar to this one:

   ```
   Raw packets sent: 15 (600B) | Rcvd: 10 (400B) | Lost: 5 (33.33%)
   ```

3. **Check that Heavy Hitters are detected for UDP flows** As before, you should be able to send UDP flows towards a given destination until the threshold value is reached. If you set the threshold to 10, you can check the behavior of the program with the following command:

   ```
   $ sudo ip netns exec ns1 nping -c 15 --udp -p 80 10.0.4.4
   ```

   Once the command finishes, you should see an output similar to this one:

   ```
   Raw packets sent: 15 (600B) | Rcvd: 10 (400B) | Lost: 5 (33.33%)
   ```

3 | Submission Instructions

To ensure a smooth and successful submission of your final project, please follow the instructions detailed below. Adhering to these guidelines will help facilitate the evaluation process and ensure that your work is properly assessed.

3.0.1. Deadlines

There are two deadlines for the project submission that you **must** follow, depending on whether you decide to attend the first or the second call. The deadlines are as follows:

- First call
  - Exam date: June 14th
  - **Project submission due date: June 5th 23:59 CET**
- Second call
  - Exam date: July 5th
  - **Project submission due date: June 26th 23:59 CET**

Make sure you submit the project at least two hours before the submission deadline to ensure no last-minute problems arise.

3.0.2. Prepare the submission

You must submit a single *.zip* file that includes all the files that are part of the *project* folder. You can create the *.zip* file to be submitted by issuing the following command (from the *project* directory):

```
$ make clean
$ cd ..
$ zip -r surname_studentID.zip project/
```

Replace *surname* and *studentID* with your own values.

3.0.3. Update the project on Webeep

The submission link on Webeep is the following:

https://webeep.polimi.it/course/view.php?id=9734&section=4.

Once you open the page, you will find two *Project submission* pages, depending on the exam that you want to attend. After clicking on the submission page, you can then click on the *Add submission* button and upload the *.zip* file.