POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Boosting the performance of NFV services with SmartNIC

Supervisor
prof. Fulvio Risso

Candidate
Raffaele Sommese

Academic Year 2017-2018

For the revolutionaries
that will shape
tomorrow

Contents

Abstract

The world of mobile networking is dramatically changing due to the new 5G technologies, which will contribute to the development of a smarter and interconnected planet by means of a ubiquitous and faster network.

These ambitious plans will establish new challenges in the networking field. In particular, one of the objectives to pursue is to find a way to increase the performance of network middleboxes, by means of the virtualization of network functions and the smart orchestration of traffic flows. The introduction of NFV technologies brought many benefits in terms of scalability and flexibility, but there are also some drawbacks, e.g., in terms of performance, compared to the solutions based on purpose-built hardware tightly coupled with network function software.

In this context, the problem previously introduced can be mitigated by performing an offload of software-defined network functions, with the aim of reaching the true speed of the underlying hardware.

In this work, we investigated the capabilities of the future generation of network interface cards, named SmartNICs, which promise to solve this problem by implementing offloading functionalities in hardware. In particular, we focused on the feasibility of offloading eBPF code into the hardware of the SmartNIC considered in our study.

At first, we explored the performance of the SmartNIC and its functionalities. We carried out some network performance tests with the aim of understanding the potentialities and the limitations of the hardware. Then, we examined the DDoS mitigator use case and we analyzed the advantages introduced by using the SmartNIC, in terms of dropped traffic, in order to exploit the possibility of reaching line rate performance without affecting the host system. Finally, we created a first proof of concept, which provides a flexible solution that allows the offloading of a generic eBPF program, i.e. a generic Virtual Network function (VNF). Moreover, the source code of a target VNF has been analyzed with the goal of extracting known patterns and stateless decision paths that can be offloaded. The feasibility of performing the offload of stateful paths has also been investigated, focusing on the recognition of the paths determined by userspace inputs.

Chapter 1

Introduction

In the last years, the traffic in communication networks is rapidly increasing, especially in mobile networks. Smartphones and other mobile devices have constantly evolved in terms of performance and dissemination. This gives the possibility to the people to connect to the Internet in any place and at any time. These new habits result in a substantial increase in data traffic. Moreover, the typology of network traffic is changed. Indeed, the biggest part of this network traffic is related to video and multimedia services [1]. The world of mobile communication is evolving with the new 5G technologies, which will contribute to the development of a smarter and interconnected planet by means of a ubiquitous and faster network.

These ambitious plans will place new challenges in the networking field. Furthermore, the ultra-low-latency requirement for innovative IoT applications, like autonomous driving systems and augmented reality, puts the networking through the wringer. The new technologies previously described and the challenging requirements that they pose lead us to rethink the architectures of the core and the edge of the network.

Nowadays, computer networks consist of a variety of active forwarding components like switches or routers, which are middleboxes dedicated to a specific purpose. Most of these devices are hardware-based and their reconfiguration and management require a big effort.

Software Defined Networking (SDN) and Network Function Virtualization (NFV) technologies have recently gained interest of network operators. The idea behind SDN is to separate the logic that makes forwarding decisions (control plane) from the forwarding itself (data plane). This introduces a great flexibility in the configuration of the networks, thanks also to the usage of an open and standardized protocol: OpenFlow.

SDN is the key solution to increase the network programmability, but nowadays, a significant part of network devices that are responsible for packet processing is running on dedicated hardware called appliances. These devices are generally hardware based because are performance-critical, but this design choice has several drawbacks in terms of flexibility and maintenance.

The introduction of Network Function Virtualization technologies brings a lot of advantages in terms of scalability and flexibility. Indeed, NFV uses the principle of virtualization and cloud computing for providing flexible network functions.

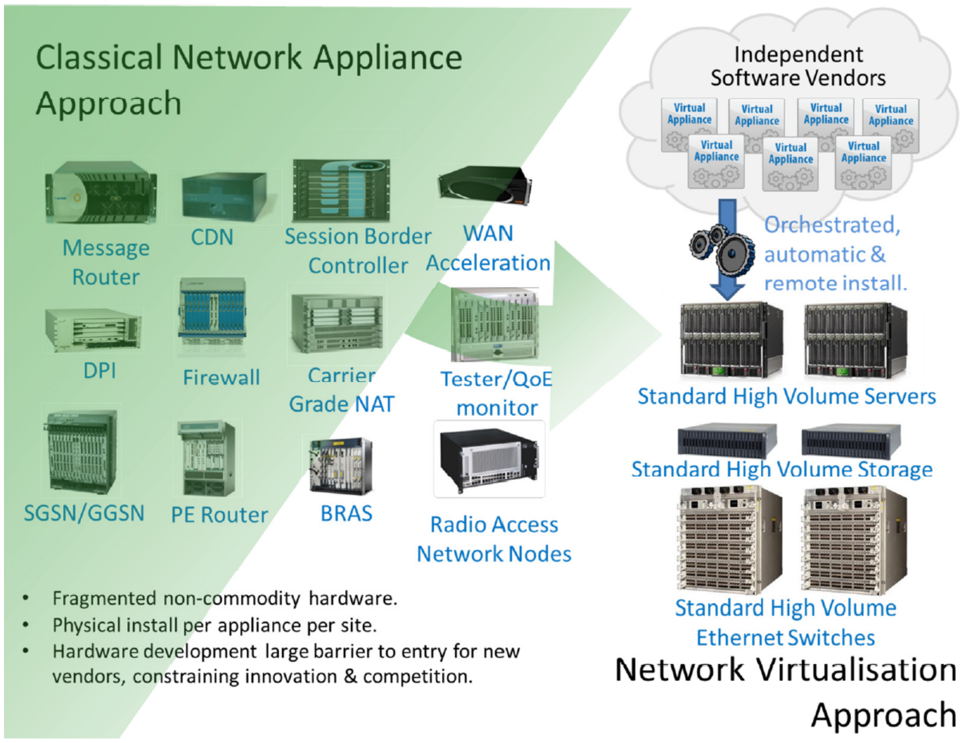

Classical Network Appliance Approach
Message Router
CDN
Session Border Controller
WAN Acceleration
DPI
Firewall
Carrier Grade NAT
Tester/QoE monitor
SGSN/GGSN
PE Router
BRAS
Radio Access Network Nodes
Fragmented non-commodity hardware.
Physical install per appliance per site.
Hardware development large barrier to entry for new vendors, constraining innovation & competition.
Independent Software Vendors
Virtual Appliance
Orchestrated, automatic & remote install.
Standard High Volume Servers
Standard High Volume Storage
Standard High Volume Ethernet Switches
Network Virtualisation Approach

Figure 1.1: Vision for Network Functions Virtualization [2]

The functions are defined in software and executed in virtual machines. This will result in a greater flexibility in terms of actions that can be performed and in an advantage in terms of cost, obtained thanks to the elasticity and the execution on commodity hardware. Nevertheless, network functions have critical requirements in terms of throughput and delay [3].

In this context, the problem introduced before can be mitigated by performing an offload of software-defined network functions, with the aim of reaching the true speed of the underlying hardware. This requires a total synergy between hardware and software, which is reached thanks to the utilization of Smart NICs. Indeed, Smart NICs support the possibility of performing the offload of part of the network traffic processing.

Chapter 2

State of the art

2.1 Background

2.1.1 BPF

The idea of performing in-kernel analysis and operations on network traffic comes from the past. In 1992 Steven McCanne and Van Jacobson presented the BSD Packet Filter [4], a filter virtual machine based on a register architecture, which promised to make the packet capture efficient and scalable. The proposed architecture was similar to the one used nowadays.

The BPF has two main components:

1. **The Network Tap** that collects the copy of the packet from the network device and sent it to the listening application.

2. **The Filter** that decides if accepts a packet and how much data of the packet must be sent to the listening application.

When BPF is attached to an interface, its driver first calls the BPF code that processes the packet and decides if it is an interesting packet (e.g. it matches some defined packet fields) and if so, it sends the packet to the userspace application. After doing that, the BPF code returns the control to the network driver that continues the execution. The original idea of BPF evolved and produced different implementations in different operating systems.

Linux Socket Filtering is the implementation introduced into the Linux kernel 2.1.75 in 1997. BSD BPF and LSF share the same idea and the same mechanism of filtering packets. However, there were some differences between the two architectures.

In Linux, the architecture of the packet filter is much simpler compared to the BSD BPF. First of all, we do not need to configure a special interface for sending

data to the userspace application.
We simply attach our filter code to the socket through the *SO_ATTACH_FILTER* option and the code is checked and executed.

The mechanism behind the BPF packet filtering is based on a virtual machine executed inside the kernel. BPF code is defined with a special bytecode that specifies the operation to do on the packet. This bytecode is injected into the kernel and executed from the virtual machine each time a packet arrives.

The code that we have written can perform data manipulation and comparison, and can trim the data of the packet that will be copied and passed to the userspace.

BPF is widely used by programs like tcpdump, libpcap, and Wireshark. These programs use the BPF virtual machine for performing an efficient in-kernel packet filtering. These programs automatically generate the BPF code. For example, this is the code related to IP packet filtering generated by *tcpdump*:

```
(000) ldh [12]
(001) jeq #0x800 jt 2 jf 3
(002) ret #96
(003) ret #0
```

As we can see the **ethertype** is loaded into a register and is compared with **#0x800** that is the IP ethertype.

2.1.2 eBPF

The original architecture of Berkley Packet Filter was born with the goal of capture and filter network traffic. As we have seen, a filter is implemented as a program that will be executed on an in-kernel register-based virtual machine.

The opportunity raised from the execution of userspace programs into the kernel space has proved to be a useful design decision but some other aspects of BPF original design have shown their weaknesses during the time. The original project of the virtual machine and its architecture was not kept up-to-date. Moreover, the modern architecture with 64-bit registers and the new type of instructions like the XADD (exchange and add) tempt the developers to change the BPF architecture [5].

For this reasons, Alexei Starovoitov started to extend the BPF architecture introducing the extended BPF or eBPF. Nowadays, the architecture of eBPF virtual machine is closer to the contemporary processors. This simplifies the mapping between eBPF bytecode and hardware ISA (Instruction Set Architecture) and guarantees better performance.

The most important changes introduced are:

- 64 bit Registers.
- 10 Registers (BPF use only 2 registers).
- New *BPF_CALL* for calling in-kernel defined function from eBPF code.

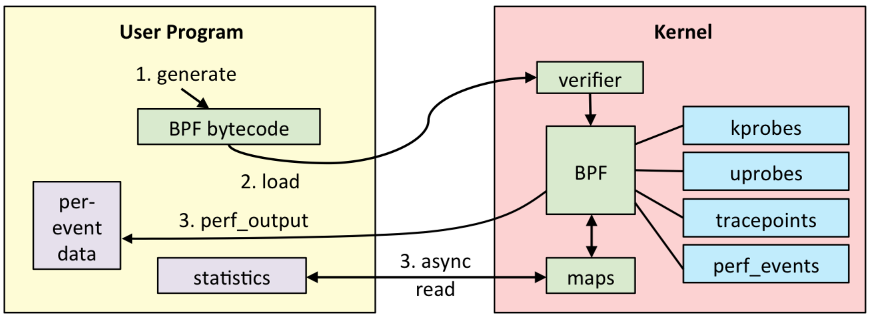


Figure 2.1: eBPF Internals [6]

In the last years, eBPF is regularly extended with the introduction of new functionalities like helpers and tail calls. Moreover, the original goal of the BPF project is changed. Indeed nowadays, eBPF is widely used for tracing kernel activities, detecting performance issues and monitoring or modifing network traffic.

The number of functions and of traceable points in the kernel rapidly increases.

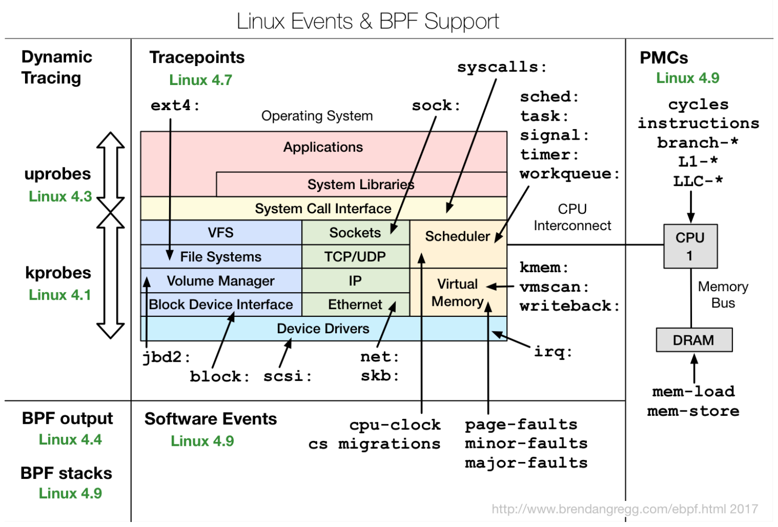


Figure 2.2: eBPF Support [6]

For writing eBPF programs, we can use a subset of C language with some limitations. This C code will be compiled and translated in a BPF assembler through a backend compiler (e.g. LLVM). This code is a sort of intermediate bytecode that will be remapped by the kernel in the native *opcode* of target ISA with a Just-In-Time compiler. This will speed up the execution time compared to the complete emulation of a virtual machine. However, the generated code is not directly executed for a security reason.

The kernel needs to check the safety of the executed code, which means that:

- The execution of the program is finished.
- The program does not contain loops (that can bring to a deadlock).
- The instructions are correct.
- The accesses to memory are safe (e.g. out of bound, dereference of a null pointer, access to forbidden memory area etc...) [5].

Maps

One key aspect introduced in eBPF is a new structure for store information, which is the eBPF map. These structures allow the programmer to store and exchange data between different eBPF executions and with userspace. This permits to create a stateful eBPF program. The map is a key/value data structure, which can be accessed through the lookup of a key.

There are different types of maps:

- **BPF_MAP_TYPE_HASH** Simple hash table,
- **BPF_MAP_TYPE_PERCPU_HASH** Per-Cpu hash table (Each core has its own version. From userspace, we will see all the versions of the map)
- **BPF_MAP_TYPE_ARRAY** Maps with uint32 key, optimized for fastest possible lookup.
- **BPF_MAP_TYPE_PERCPU_ARRAY** Same as the previous one, per-cpu.
- **BPF_MAP_TYPE_PROG_ARRAY** Stores the file descriptor of eBPF programs, used for tail calls.
- **BPF_MAP_TYPE_LRU_HASH** Hash table that stores only recently used object.
- **BPF_MAP_TYPE_PERCPU_LRU_HASH** Same as the previous one, per-cpu.

There are other types of maps, used in specific contexts, but here we have reported only the most used one. For a complete reference, the reader is referred to kernel documentation[1].

2.1.3 XDP

The eXpress Data Path (XDP) is a programmable high-performance packet processor implemented in the Linux networking data-path. XDP allows processing and filtering packets at the lowest point of the network software stack in the kernel.

With XDP we can completely bypass the higher levels of network software stack of the Linux kernel by obtaining in this way best performance for some applications that do not require an OS processing (e.g. packet dropping). XDP also avoids the creation of in-kernel meta-data structure such as *skb*. XDP processes directly RX packet-pages that come from the RX ring queue of the device driver.

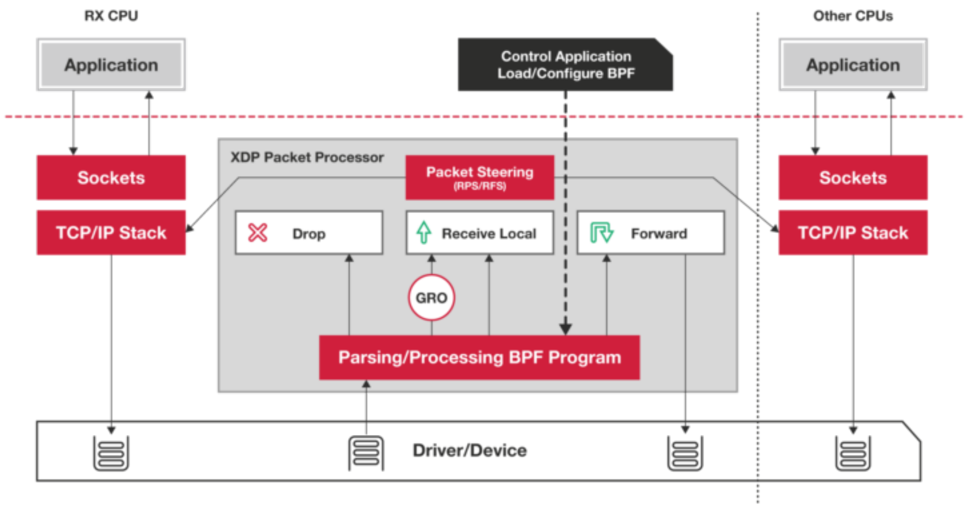


Figure 2.3: eXpress Data Path [7]

The typical use cases of XDP are:

- Pre-stack filtering (e.g. DDoS mitigation)
- Forwarding and load balancing
- Sampling [7]

[1]https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html

2.2 Related Work

One of the most interesting related projects is Metron [8], where the concept previously discussed in the Chapter 1 is implemented. In this article, authors prove that by using hardware assistance, Metron deeply inspects traffic at 40 Gbps and realizes stateful network functions at the speed of a 100 GbE network card on a single server.

Metron focuses on three main big aspects that can speed up the processing of network traffic:

1. Network Function Synthesis
2. CPU Affinity Selection
3. Partial Offloading of Network Function

The Network Function Synthesis reduces multiple network functions to a single one. This task is performed with the help of SNF, a framework that reduces a chain of network functions into a single optimized one. This type of optimization avoids that a packet flows through different network functions. This is important because often different network functions are allocated on different cores, so a transfer of a packet between two cores requires an inter-core communication or an inter-process communication that in the worst case require an access to the main memory.

For a 100 Gbit/s network the process time of a small packet is around 10 ns, so it is practically impossible to switch the packet through the various cores.

This brings another important consideration introduced by Metron: the classical VNF architecture has a CPU core dedicated to network packet switching. Metron, instead, uses the functionality provided by underlying hardware for classify and tag the packet and assign it to correct CPU core that is currently executing the VNF software. With this approach Metron realizes the CPU Affinity Selection. This type of approach avoids an incorrect choose of CPU core for a packet, which, as seen previously, introduces a huge delay in network processing.

The last advantage introduced by Metron is the offloading of part of network functions (the stateless part) into the hardware, in particular in the network interface card. This solution helps to provide better performance for the stateless application (ex. firewall) and reduces the load of the remaining part of the network function that can speed up its performance.

Chapter 3

SmartNIC: An overview

This chapter presents an overview of the architecture of the SmartNIC used in this thesis. The SmartNIC is a Broadcom SmartNIC. In particular, our model has the following main technical specifications:

CPU	8-core ARM v8 A72 at 3.0 GHz
RAM	8 GB DDR4 memory at 2,400 MT/s
FLASH	16 GB
PCI	x8 PCI Express v3.0 interface
PHY	Dual-port SFP28 25GbE/10GbE

Table 3.1: SmartNIC Technical Specifications

The SmartNIC is provided with a customised Linux-based OS on-board.

Figure 3.1 shows the traffic flows between Host, ARM processor and external port in the standard configuration.

- The traffic between the host and external port flows without passing inside the ARM processor.
- The traffic that has as source or destination the physical address of the internal ARM interfaces is the only traffic that flows through the ARM side.

Moreover, there are eight Ethernet interfaces both inside (in the ARM system) and outside (in the host) in the default configuration of NIC card.

Each interface corresponds to a different VF (Virtual Function) and four interfaces (the ones numbered with an even number) are linked to physical port 0 while the others are linked to physical port 1 of the dual port SmartNIC. The NIC supports also interface bonding, which enables 50GbE link speed (through two cables).

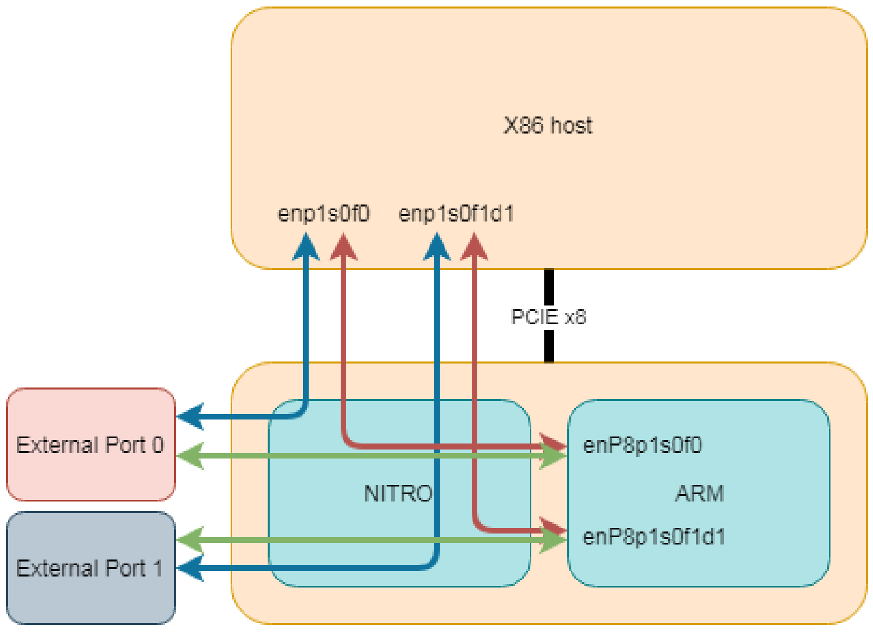


Figure 3.1: NIC Ethernet Traffic Flow

3.1 OS

The customised Linux-based OS provided is Poky Linux with kernel 4.14. Poky is a reference distribution of the Yocto Project.

The NIC has also EFI as BIOS, so it is possible to try to boot also another operating system on it.

Unfortunately, there is a small amount of flash on-board, so it is difficult to cross-compile and brings software into the NIC.

3.1.1 Flash overview

The flash is an eMMC flash with an actual capacity of 13.8 GiB.

The flash layout is divided into different partitions reported below:

mmcblk0p1	UEFI boot	512 MB
mmcblk0p2	Linux kernel	768 MB
mmcblk0p3	Linux root primary	4 GB
mmcblk0p4	Linux root secondary	4 GB
mmcblk0p5	Linux filesystem	2 GB

Table 3.2: Flash Layout

3.1.2 Chroot

Due to the limited space available (only 100 Mb free), we have decided to adopt another technique for porting the software to the NIC. The current OS in the NIC supports both NFS and loopfs, so we have created an NFS share on the host system (see the following section for the complete architecture), and we have exported it.

- We have mounted the created share folder inside the NIC
- We have created a loop file inside the NFS mount and mounted it as loop device inside the NIC.
- We have installed a new OS inside this device. The chosen OS is Ubuntu 16.04.3 LTS, and we have used *debootstrap* for installing it.

Unfortunately, we can not use an NFS share directly due to lack of file lock support (this is a known bug that makes impossible to create a *debootstrap* setup over NFS), this is why we have decided to create a loop file-system.
We have installed Ubuntu with *debootstrap* in two stages:

- In the first stage, we have executed *debootstrap* on the host machine, which download all the base installation system inside the loop device.
- In the second stage, we have performed the setup process inside the NIC after entering in the chroot environment.

As a result, we have a complete Ubuntu 16.04 OS installed on the NIC and the performance seems to be not affected by the complexity of storage that is behind it.

This seems to be the best setup for running our test and for porting software, because it gives us the opportunity to use a package manager and repositories (absents in Poky) with a large amount of software pre-compiled.

We choose **Ubuntu 16.04** because it is the distribution used in our lab for developing and testing the **PolyCube** framework.

Nevertheless, the adopted solution has weak points, one of this is the complex setup, which is unfeasible for a large scale adoption. Another problem is the kernel: the chroot environment shares the same kernel of the parent OS, and we cannot execute a different version of the kernel.

3.2 Performance Test

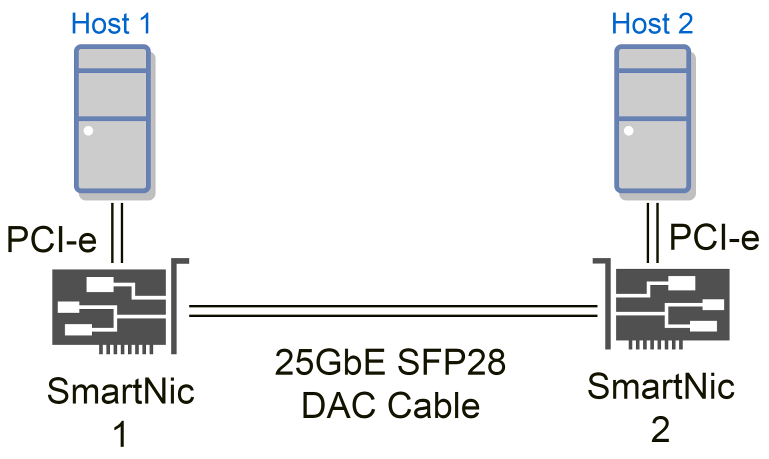


Figure 3.2: Testbed Setup

We have implemented a testbed for evaluating the performance of the SmartNIC. The previous figure shows the architecture of our setup. In particular, we have two SmartNIC interconnected through a 25GbE SPF28 DAC cable. We propose a comparison of network performance between:

- Host1 - Host2
- Host1 - SmartNIC1
- SmartNIC1 - Host1

Below we report the characteristics of the two hosts:

CPU	Intel Core i7-3770 CPU @ 3.40GHz
Memory	32 GB DDR3 @ 1600MHz
Disk	Sata III SSD 120GB
OS	Ubuntu 16.04
Kernel	Linux 4.18

Table 3.3: Hosts Specification

In SmartNIC1 we use the chroot-ed environment with Ubuntu 16.04, because in the base Poky system we do not have the tools for collecting CPU usage statistics. Anyway, we have confirmed that this solution does not have any significant impact on the measurements.

We have developed a little suite of scripts with the purpose of automating the tests. See Appendix A for further details.

3.2.1 Host1-NIC1 Performance

In this section, we describe the performance test performed between Host 1 and NIC 1.

Performance Test with default parameters

We have executed a performance test with default parameters with this command:

```
$ iperf -f k -s
```

```
$ iperf3 -f k -c PCIP -o 15 -i 1 -t 60
```

Test	H1-Srv	H1-Srv-R	N1-Srv	N1-Srv-R
TCP	28.9Gbit/s	14.2Gbit/s	14.7Gbit/s	28.6Gbit/s

Table 3.4: Test suite Host1-NIC1 TCP results

In the previous table, we have reported the result of a standard TCP test. The tests marked with the R name are the tests executed with *Iperf* in reverse mode. During this test, we have discovered a strange behaviour in terms of throughput of traffic generated from the PC to the NIC.

For this reason, we have decided to go more in deep, for finding the root cause of this anomaly. Figure 3.3 shows the throughput graph for a 60 seconds test. Excluding the first part of the assessment of TCP flow, we can note that traffic speed is around 30Gbit/s with a peak of 36Gbit/s.

This result is related to the PCI-E interface, indeed the NIC is connected to a PCI-E 2.0 x8 Slot that supports a maximum speed of 4GB/s (32GBit/s). The peak result is certainly related to some internal buffering.

We will propose in future a test with PCI-E 3.0 x8 that reaches the theoretical speed of 7.88 GB/s (63Gbit/s).

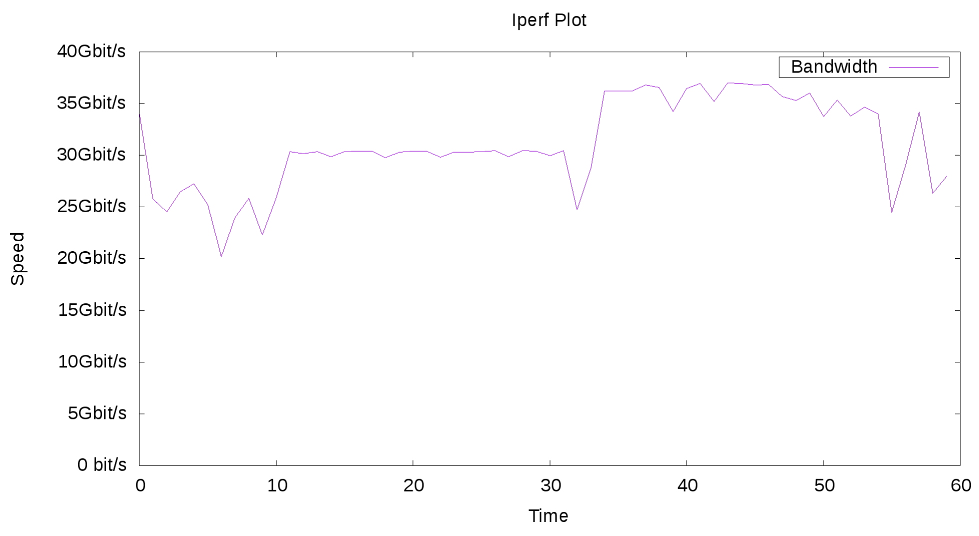


Figure 3.3: Throughput Graph TCP Standard Flow NIC-PC

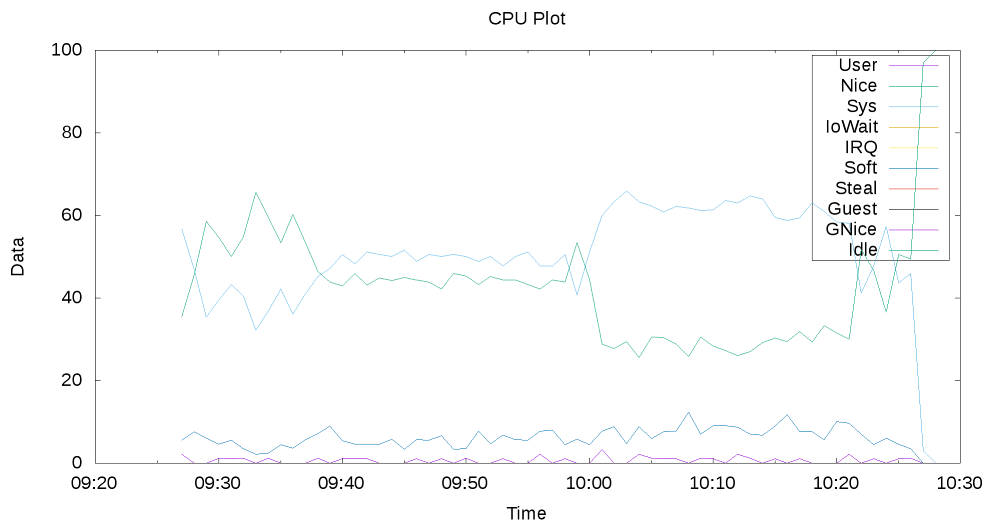


Figure 3.4: Cpu Usage NIC

Figure 3.4 shows the CPU utilisation of the core used by the *iperf3* process, the graph clearly demonstrates that the CPU was not limiting the process performance. We avoid reporting the ram usage because it was found to be negligible. The reverse test, with traffic generated from the PC to the NIC, shows the strange behaviour presented before, in particular in the Figure 3.5 we reported the throughput graph for a 60 second test and we can note that traffic speed is around 15 Gbit/s.

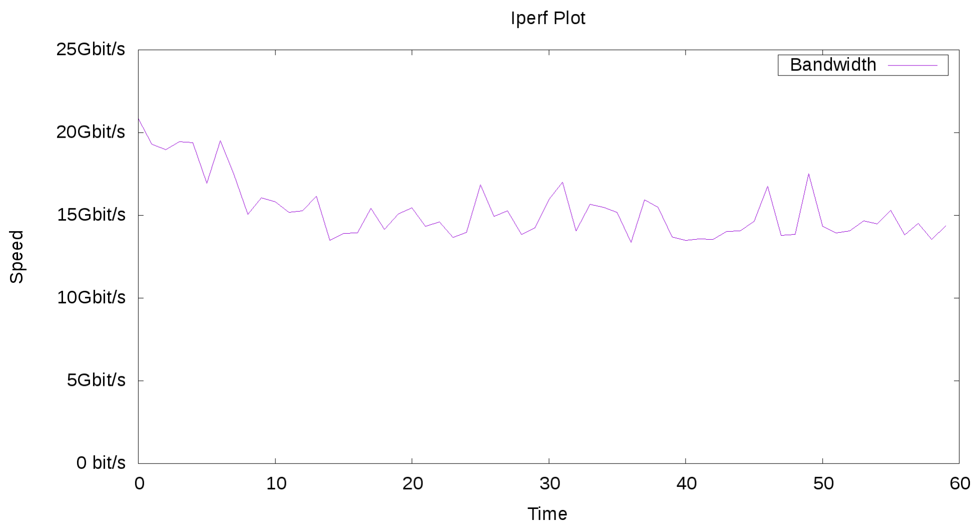


Figure 3.5: Throughput Graph TCP Reverse Flow PC-NIC

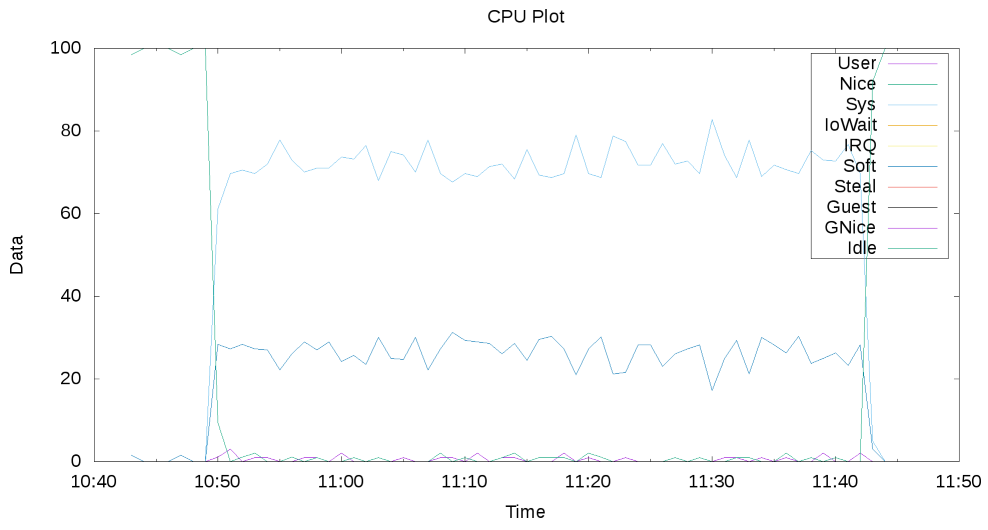


Figure 3.6: Cpu Usage NIC

Figure 3.6 shows the CPU utilisation of the core used by the *iperf3* process, the graph clearly demonstrates that the CPU limits the process performance, indeed the core usage raises to 100% during the test. The difference between the two tests is probably related to the TSO (TCP Sender offloading). TSO is a technique for increasing the egress throughput of high-bandwidth network connections by reducing CPU overhead. It works by passing a multipacket buffer to the network interface card (NIC). The NIC then splits this buffer into separate packets [9].

TSO is enabled only on the NIC side, and this probably will result in this behaviour. At the time of writing, we cannot change TSO on PC side so we decide to perform a test disabling it on the NIC side.

Test	**H1-Srv**	**H1-Srv-R**	**N1-Srv**	**N1-Srv-R**
TCP	14.5Gbit/s	15.1Gbit/s	15.5Gbit/s	14.9Gbit/s

Table 3.5: Test suite Host1-NIC1 TCP results TSO off

In Table 3.5 we report the result of the same test executed with TSO turned off on NIC side.

It is clear in this case, the performance in both ways are the same. For this reason, we have decided to execute the next tests both with TSO enabled and disabled.

Performance Test with custom parameters

ETH Frame	**TCP**	**UDP**	**Parameter TCP**	**Parameter UDP**
64 Byte	6 Byte	18 Byte	-l 6 -b0	-u -l 18 -b0
128 Byte	N/A	82 Byte	N/A	-u -l 82 -b0
256 Byte	198 Byte	210 Byte	-l 198 -b0	-u -l 210 -b0
512 Byte	N/A	466 Byte	N/A	-u -l 466 -b0
1024 Byte	966 Byte	978 Byte	-l 966 -b0	-u -l 978 -b0
1280 Byte	N/A	1234 Byte	N/A	-u -l 1234 -b0
1518 Byte	N/A	1472 Byte	N/A	-u -l 1472 -b0
4096 Byte	4038 Byte	N/A	-l 4038 -b0	N/A
9018 Byte	N/A	8972 Byte	N/A	-u -l 8972 -b0
64 KByte	65478 Byte	N/A	-l 65478 -b0	N/A

Table 3.6: Test suite parameters

The following tables show the results of the test, for TCP the behaviour is similar to the one analysed before. In the UDP case, instead, we have reported between brackets the percentage of loss packets. As we can see, in some cases it will be very high. We have found that this behaviour is again strictly linked to the CPU usage on the NIC. Indeed when we have a high percentage of lost packets, the CPU usage goes to 100%.

Test	**H1-Srv**	**H1-Srv-R**	**N1-Srv**	**N1-Srv-R**
TCP 64B	35.5Mbit/s	35.2Mbit/s	29.5Mbit/s	36.1Mbit/s
TCP 256B	1Gbit/s	947.7Mbit/s	883.2Mbit/s	1Gbit/s
TCP 1024B	4Gbit/s	2.8Gbit/s	3.1Gbit/s	4.6Gbit/s
TCP 4KB	12.9Gbit/s	7.1Gbit/s	7.4Gbit/s	13.1Gbit/s
TCP 64KB	28.7Gbit/s	13.1Gbit/s	11.7Gbit/s	29.1Gbit/s

Table 3.7: Test suite Host1-NIC1 TCP results TSO offload On

Test	**H1-Srv**	**H1-Srv-R**	**N1-Srv**	**N1-Srv-R**
TCP 64B	34.6Mbit/s	33.5Mbit/s	30.3Mbit/s	36.7Mbit/s
TCP 256B	1Gbit/s	1Gbit/s	912Mbit/s	1Gbit/s
TCP 1024B	4.2Gbit/s	2.7Gbit/s	2.9Gbit/s	4.3Gbit/s
TCP 4KB	5.4Gbit/s	6.7Gbit/s	8.1Gbit/s	6.8Gbit/s
TCP 64KB	12.9Gbit/s	13.4Gbit/s	11.4Gbit/s	13.2Gbit/s

Table 3.8: Test suite Host1-NIC1 TCP results TSO offload Off

Test	**H1-Srv**	**H1-Srv-R**	**N1-Srv**	**N1-Srv-R**
UDP 64B	76.3Mbit/s (16%)	60.4Mbit/s (2.4%)	67.5Mbit/s (24%)	66.4Mbit/s (0.33%)
UDP 128B	349.8Mbit/s (23%)	258.8Mbit/s (0.062%)	312.8Mbit/s (28%)	297.1Mbit/s (18%)
UDP 256B	879.6Mbit/s (10%)	657.5Mbit/s (0.052%)	708.5Mbit/s (16%)	776.3Mbit/s (17%)
UDP 1024B	3.6Gbit/s (6.8%)	3.1Gbit/s (38%)	3.2Gbit/s (40%)	3.2Gbit/s (13%)
UDP 1280B	4.6Gbit/s (2.1%)	3.5Gbit/s (32%)	4.2Gbit/s (52%)	4Gbit/s (18%)
UDP 1518B	5.4Gbit/s (7.7%)	4.6Gbit/s (18%)	5.1Gbit/s (22%)	4.8Gbit/s (10%)
UDP 9018B	5.1Gbit/s (1.4%)	10.3Gbit/s (63%)	10.5Gbit/s (60%)	5.2Gbit/s (0.28%)

Table 3.9: Test suite Host1-NIC1 UDP results TSO offload Off

Test	**H1-Srv**	**H1-Srv-R**	**N1-Srv**	**N1-Srv-R**
UDP 64B	77.7Mbit/s (3.1%)	63.1Mbit/s (12%)	72.8Mbit/s (16%)	66.4Mbit/s (1.8%)
UDP 128B	344.6Mbit/s (17%)	291.4Mbit/s (17%)	336.4Mbit/s (8.3%)	298.9Mbit/s (0.091%)
UDP 256B	905Mbit/s (18%)	663.6Mbit/s (0.057%)	812.9Mbit/s (29%)	773.5Mbit/s (0.38%)
UDP 1024B	3.8Gbit/s (0.16%)	3.1Gbit/s (42%)	3.6Gbit/s (57%)	3.2Gbit/s (36%)
UDP 1280B	4.7Gbit/s (0.21%)	4Gbit/s (47%)	4.5Gbit/s (61%)	4.1Gbit/s (43%)
UDP 1518B	5.4Gbit/s (19%)	4.7Gbit/s (47%)	5.4Gbit/s (60%)	5.4Gbit/s (0.2%)
UDP 9018B	5.2Gbit/s (0.48%)	10.5Gbit/s (70%)	11Gbit/s (69%)	6.2Gbit/s (1.2%)

Table 3.10: Test suite Host1-NIC1 UDP results TSO offload On

3.2.2 Host-Host Performance

In this section, we describe the performance test performed between Host 1 and Host 2.

Performance Test with default parameters

Test	H1-Srv	H1-Srv-R	H2-Srv	H2-Srv-R
TCP	20.8 Gbit/s	22.4 Gbit/s	22.3 Gbit/s	20.6 Gbit/s

Table 3.11: Test suite Host1-Host2 TCP results

With standard parameters we are near to the line speed of 25 GBbit.

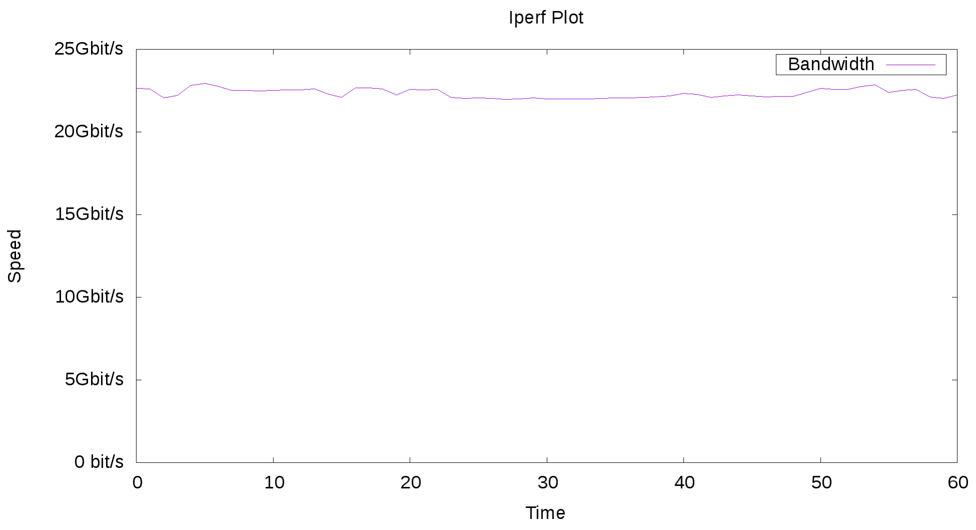


Figure 3.7: Throughput Graph TCP Standard Flow PC1-PC2

As can be seen in the previous figure, the throughput is constant. The CPU core usage on the receiver is near to 100% and this probably has an impact on performance.

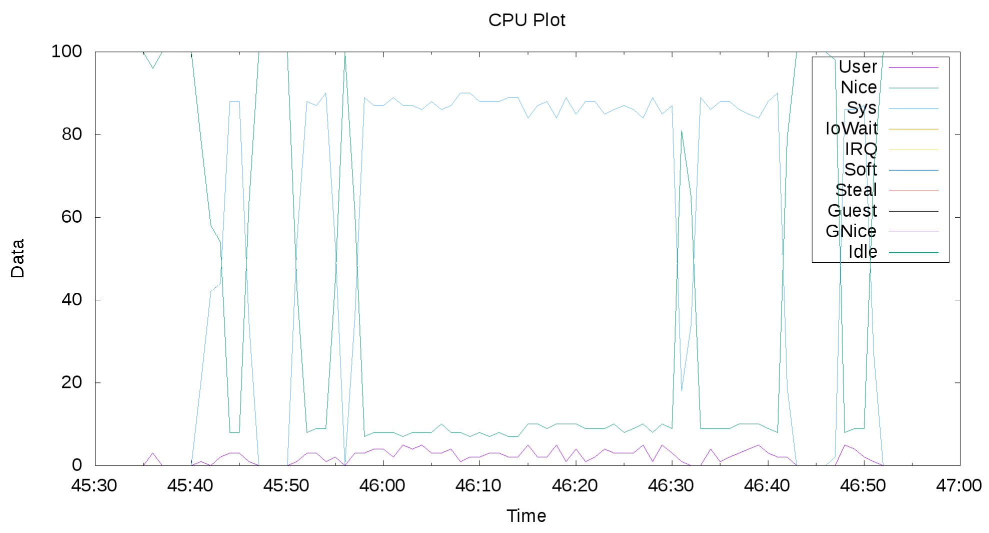


Figure 3.8: CPU usage PC2

Performance Test with custom parameters

Below, there are reported all the tests with the parameters defined in Table 3.6

Test	H1-Srv	H1-Srv-R	H2-Srv	H2-Srv-R
TCP 64B	36.2Mbit/s	27.6Mbit/s	26.2Mbit/s	37.1Mbit/s
TCP 256B	1Gbit/s	892.6 Mbit/s	828.1Mbit/s	1Gbit/s
TCP 1024B	2.7Gbit/s	2.7Gbit/s	2.8Gbit/s	2.4Gbit/s
TCP 4KB	10.3Gbit/s	7.9Gbit/s	7.9Gbit/s	8.7Gbit/s
TCP 64KB	20.4Gbit/s	20.9Gbit/s	21.6Gbit/s	22.1Gbit/s

Table 3.12: Test suite Host1-Host2 TCP results

Test	H1-Srv	H1-Srv-R	H2-Srv	H2-Srv-R
UDP 64B	53Mbit/s (0.36%)	58.2Mbit/s (30%)	60.6Mbit/s (38%)	49.8Mbit/s (1.3%)
UDP 128B	258.4Mbit/s (5.6%)	251Mbit/s (24%)	274.7Mbit/s (34%)	227Mbit/s (0.094%)
UDP 256B	641.9Mbit/s (1.8%)	641.7Mbit/s (26%)	740.2Mbit/s (36%)	565.9Mbit/s (0.31%)
UDP 1024B	2.8Gbit/s (14%)	2.9Gbit/s (49%)	3.3Gbit/s (0.68%)	2.4Gbit/s (4.5%)
UDP 1280B	3.5Gbit/s (6.2%)	3.4Gbit/s (43%)	4Gbit/s (0.35%)	3.1Gbit/s (6.9%)
UDP 1518B	3.9Gbit/s (23%)	4.2Gbit/s (49%)	4.9Gbit/s (32%)	3.7Gbit/s (7.2%)
UDP 9018B	8.8Gbit/s (22%)	9.9Gbit/s (40%)	8.8Gbit/s (32%)	8.3Gbit/s (13%)

Table 3.13: Test suite Host1-Host2 UDP results

3.3 PolyCube DDoS Mitigator

The initial phase of this work was devoted to explore the opportunity of running eBPF code directly into the hardware of the SmartNIC. PolyCube is a framework for virtual network functions written in eBPF developed in our research group. We used the chroot environment for installing and executing the PolyCube framework inside the SmartNIC.

In this section, we present the performance results of this solution compared to the execution of the same software inside the host CPU of overlaying hardware.

3.3.1 Performance Test Single Flow

The Testbed is implemented with two identical machines and SmartNICs as presented before.

The software used for the generation of the traffic is *Pktgen*. We use the kernel version of *pktgen* instead of *pktgen-dpdk* due to the simpler configuration.

The generated traffic has the following characteristics:

Flows	1 flow (same IP for each CPU)
Source and destination Port	9/udp
Packet Size	64 Byte
Burst	32
Million of Packet per Second	36.7

Table 3.14: Pktgen Parameter Single Flow

The testing flow is generated in burst mode and a 64 byte packet size, we obtain a generated Ethernet packet of 64Byte. The generated traffic is UDP traffic on port 9, and we generate $\sim$ 37 Million of packets per second.

We use these scripts[1] for configure pktgen and we launch the script with this arguments:

```
$ sudo ./pktgen_sample03_burst_single_flow.sh  -i enp1s0f0 \
-s 64 -d 192.168.1.20 -m 00:0a:f7:ec:08:a8 -t 8 -f 0 -n 0 -b 32
```

[1]https://github.com/netoptimizer/network-testing/tree/master/pktgen

We report the result in the following graph:

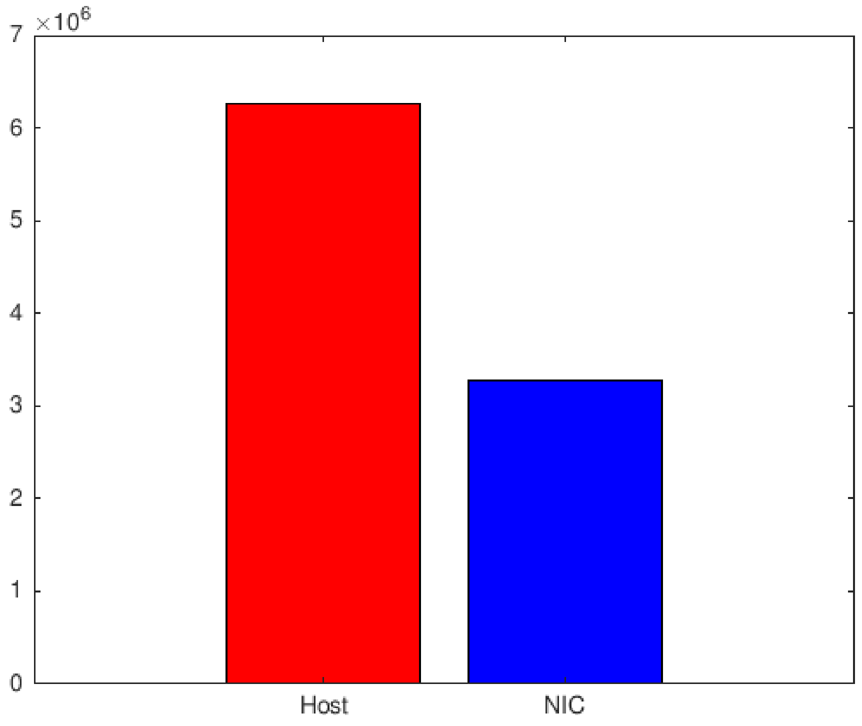


Figure 3.9: PolyCube DDoS Host-NIC Single Flow

In the same condition we have compared the performance with this XDP program[2] that drops all the packets on an interface with this result (the source is again a single flow):

Test	**PC**	**NIC**
Dropped pps	10055230	6120200

Table 3.15: Drop Packets Performance

[2]https://github.com/iovisor/bcc/blob/master/examples/networking/xdp/xdp_drop_count.py

3.3.2 Performance Test Multi Flows

We perform also a multi flows (a flow for thread) test.
The generated traffic has the following characteristics:

Flows	8 flow (one for each core with different IP)
Source and destination Port	9/udp
Packet Size	64 Byte
Burst	32
Million of Packets per Second	36.7

Table 3.16: Pktgen Parameter Multi Flows

```
$ sudo ./pktgen_sample03_burst_single_flow.sh  -i enp1s0f0 \
-s 64 -d 192.168.1.20 -m 00:0a:f7:ec:08:a8 -t 8 -f 0 -n 0 -b 32
```

The result of our test is shown below:

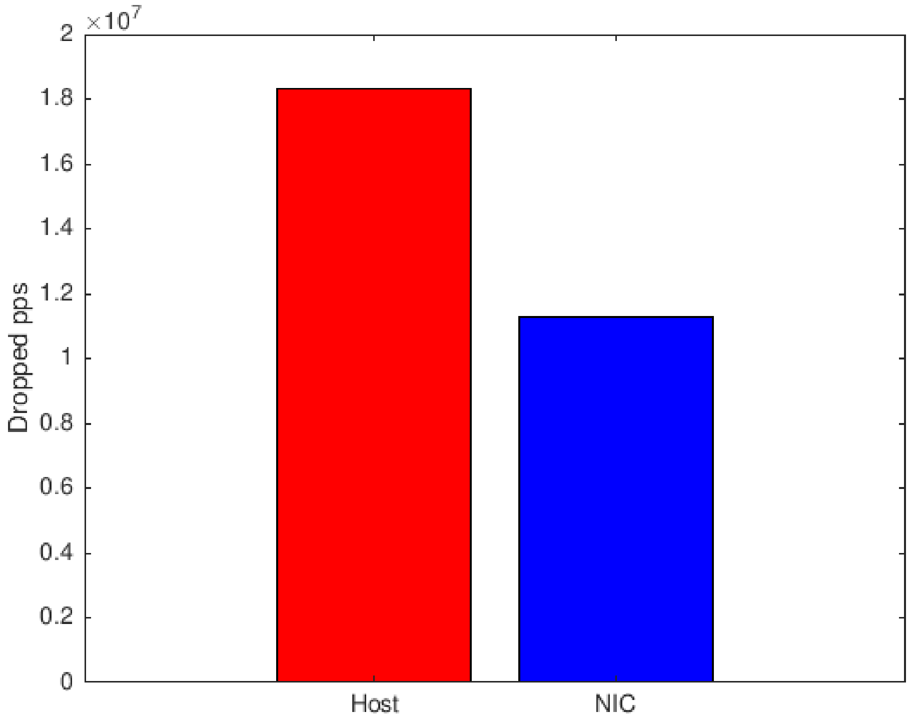


Figure 3.10: PolyCube DDoS Host-NIC Multi Flows

3.3.3 Consideration

We have found that the performance of the eBPF code executed inside the SmartNIC is worse compared to the same code executed inside the host CPU.

This result is surprising because we expect that there was some offloading support inside the SmartNIC that can execute this software faster. We have discovered that this version of SmartNIC does not support this offloading feature, but on the market, there are some other manufacturers that implement the offloading of XDP code.

However, it was interesting to compare the performance between the SmartNIC ARM processor and the Host processor. Our main focus is the boosting of network performance with the helping of the offloading function of underlying hardware, so we decided to stop to experiment with the ARM system inside the NIC, but it is important to remark that the software executed into the OS of the NIC take responsibility of part of CPU Host load, and reduces its load.

For this reason, these NIC can be useful in situations in which we want to avoid to execute network software in the host system. As an example, in a scenario with multiple OSes executed in the host system through virtualization, the task of coordinate network traffic between virtual machines, perform routing, load balancing or other tasks can be assigned to the NIC.

Chapter 4

SmartNIC: Offloading Features

Due to the poor performance detected with the execution of XDP code inside the ARM processor of Smart-NIC, we have decided to move on and look if there is native support for any other type of Hardware Offloading. Our SmartNIC supports an offloading characteristic called TruFlow.

In particular, the documentation states that: “The TruFlow flow processing engine is integrated into the NetXtreme C-Series Ethernet controllers, and TruFlow implements the fundamental SDN protocol constructs of classification/match/action processing in hardware, including the latest OpenFlow standard” [10].

Initially, we started to explore this type of offloading through OpenVSwitch, but we have noticed that there is the possibility of using the Linux Traffic Control tool to reach the same results, so we decided to use it. Indeed, the use of OpenVSwitch makes the configuration a bit more tricky and it offers no advantage.

For this reason, we choose to use Traffic Control. Moreover, we use particular features of Traffic Control implemented recently, so we need to install a recent version of TC (in our case we install the package *iproute2* version 4.15 from the Ubuntu Bionic repository).

4.1 Traffic Control

Traffic Control is the name used to identify a series of mechanism and scheduling systems that govern the network traffic. This includes the choice about which packets we want to receive, at what rate, which packets retransmit, and so on.

In a standard situation, Traffic Control is not used intensively, indeed the common queuing discipline used is the FIFO, where the queue collects the incoming packets and tries to dequeues them as soon as possible.

The Linux kernel calls the queuing discipline **qdisc** and the default qdisc applied to all the network interfaces (unless stated otherwise) is the *pfifo_fast*, which is slightly different from the standard FIFO. The pfifo_fast queuing discipline uses different bands, which are basically three different FIFO queues, for separating the traffic. The highest priority traffic is always placed in the first band and served for first [11]. In pfifo_fast, packets are enqueued into the three bands relying on their Type of Service(TOS) bits or assigned priority.

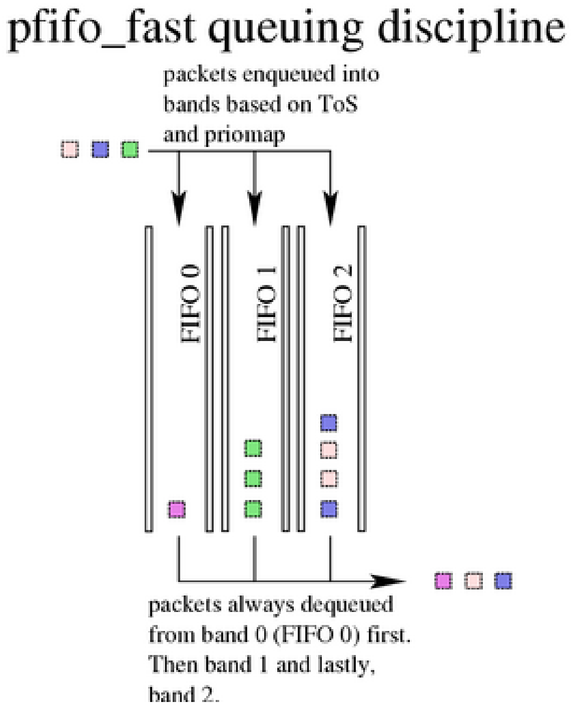

pfifo_fast queuing discipline
packets enqueued into bands based on ToS and priomap
FIFO 0
FIFO 1
FIFO 2
packets always dequeued from band 0 (FIFO 0) first. Then band 1 and lastly, band 2.

Figure 4.1: Pfifo_Fast queueing discipline

Besides pfifo_fast, another way to classify the traffic is by dividing it into different flows. A Flow is a distinct connection between two hosts, and generally, it is defined by the tuple of IPs (source and destination), Layer 4 Protocol and Ports (source and destination). Traffic control frequently adopts a flow based decision for assigning the packet to a queue or for choosing the action to follow.

The traffic control system is part of kernel, and it is possible to enable or disable different features during the compilation of the kernel. There are some command line utilities, part of **iproute2** package, which allow the user to interact with kernel traffic control system. In particular we use the **tc** utilities.

4.2 TC Flower

The TC Flower Classifier is a tool that gives users the opportunity to control the flows of network traffic and to define rules that match some packet fields or metadata.

TC Flower idea came from the flow classification introduced by OpenFlow and implemented in software by OpenvSwitch. Moreover, the matching actions offered by TC Flower are similar to the ones defined for OpenVSwitch. For this reason, it is simple to migrate rules between the two solutions.

Furthermore, TC Flower gives the possibility to offload rules to hardware, if supported by underlying NIC.

TC Flower is part of TC tool, which is available in most common Linux distributions.

With TC several actions can be implemented: we can drop the packet, we can modify it or send it to another interface etc. TC can match a packet by using different fields: IP, Port, Protocol, particular fields and so on [12].

4.2.1 Tc Flower Syntax

We have started to explore the TC Flower capability with a simple experiment. We tried to insert a rule for dropping packets from a given ip.

First of all, we have created a new queuing discipline for the ingress packets. The ingress qdisc allows to apply the tc filters to incoming packets on a network interface, regardless of whether they have a local destination or a remote one and so they need to be forwarded. The standard qdisk applies only to egress traffic therefore we need another qdisk for ingress.

```
$ tc qdisc add dev enp1s0f0 ingress
```

The following command allows to block an ip:

```
$ tc filter add dev enp1s0f0 parent ffff: handle 0x055245 prio 65535
  protocol ip flower skip_sw src_ip 192.168.1.30 action drop
```

If we analyze this command we can spit it into different parts:

- **tc filter**: Base filter command.
- **add dev enp1s0f0**: Device to which the rule will be added.
- **parent ffff:**: Parent qdisk which the rule belong to (in our case ingress).
- **handle 0x055245**: A unique identifier for the rule.
- **prio 65535**: The priority of the rule.

- **protocol ip**: The rule will match only ip packets.
- **flower**: The rule will execute a flow-based matching.
- **skip_sw**: Do not process filter by software. Offload rule to hardware.
- **src_ip 192.168.1.30**: The rule matches flows based on source ip address.
- **action drop**: Action applied when a packet matches the rules.

The supported options for the flower filtering are:

dst_mac	Match on Destination Mac.
src_mac	Match on Source Mac.
vlan_id	Match on vlan tag id.
vlan_prio	Match on vlan tag priority.
vlan_ethtype	Match on layer three protocol.
mpls_label	Match the label id in the outermost MPLS label stack entry.
mpls_tc	Match on the MPLS TC field.
mpls_bos	Match on the MPLS Bottom Of Stack field.
mpls_ttl	Match on the MPLS Time To Live field.
ip_proto	Match on layer four protocol.
ip_tos	Match on ipv4 TOS or ipv6 traffic-class.
ip_ttl	Match on ipv4 TTL or ipv6 hop-limit.
dst_ip	Match on source IP address.
src_ip	Match on source IP address.
dst_port	Match on layer 4 protocol destination port number.
src_port	Match on layer 4 protocol source port number.
tcp_flags	Match on TCP flags represented as 12bit bitfield in in hex format.
ip_flags	This field could be used to match on fragmented packets

Table 4.1: TC Flower option [13]

Tc Flower allows to specify three operating modes: skip_sw, skip_hw and not specified.

In the skip_sw mode TC Flower tries to offload the rule to the NIC driver. If this operation fails the rule will not be added.

In the skip_hw mode TC Flower ignores completely the underling hardware and sets the rules in software.

If we do not specify the operating mode, TC Flower first tries to put the rule in hardware and if the operation fails tries to allocate it in software.

4.3 Tc Flower Offloading Test

We will evaluate the difference in terms of performance between a tc rule executed in hardware and in software mode.

We use for this test a simple tc rule that blocks a source ip address and we generate packets with *pktgen kernel*.

The parameters used for the traffic generation are reported in the following table:

Flows	1 flow (same IP for each CPU)
Source and destination Port	9/udp
Packet Size	64 Byte
Burst	32
Million of Packets per Second	36.7

Table 4.2: Pktgen Parameter Single Flow

We start *pktgen* script with these arguments:

```
sudo ./pktgen_sample03_burst_single_flow.sh  -i enp1s0f0 \
-s 64 -d 192.168.1.20 -m 00:0a:f7:ec:08:a8 -t 8 -f 0 -n 0 -b 32
```

The result of our test is shown below:

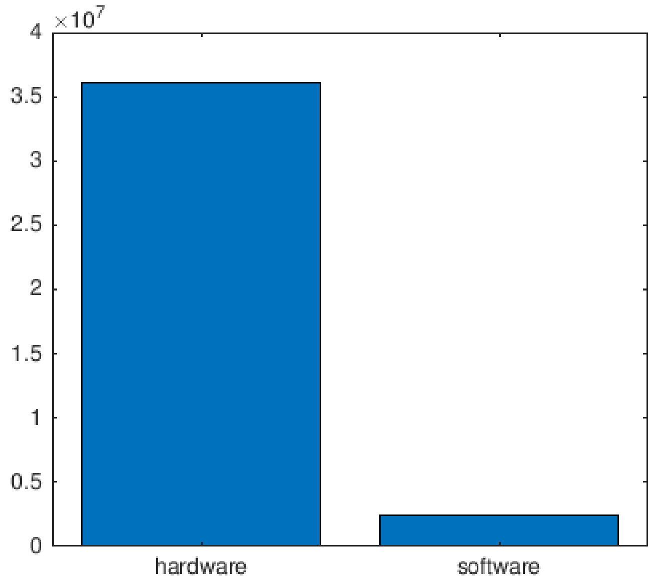


Figure 4.2: Tc Hardware-Software Comparison

As we can see in hardware mode we reach around 35 million of packets dropped per second.

At 25 Gb/s we can generate at maximum 37202380 packets per second with packets of 64 bytes.

Indeed:

$$\frac{\frac{25Gb/s}{8bit}}{64byte + 20byte(preamble + interframe)} = 37.2Mpps$$

It is possible to see that the dropping rate of software mode is nearly one order of magnitude lower.

This is why we decide to explore the possibility of using the tc hardware offloading capabilities.

4.4 TC Flower-PolyCube Integration

With the aim of using the advantages described before, we start to find a way for integrating TC Flower rules in an eBPF program of PolyCube Framework.

The first support added to eBPF code is a declarative support: in particular, we define different annotations, which allow the programmers to specify the scope of a specific map inside the program.

For example, if the programmer defines into the code a map and uses it to store the list of source IPs to block, he can annotate this map with a special annotation and our preprocessor will interpret this annotation, by understanding the behaviour and the scope of the map.

In our demo we support five types of maps:

- **IP_SRC_MAP** : Used for specify that it is a map of source ip blocked.
- **IP_DST_MAP** : Used for specify that it is a map of destination ip blocked.
- **PORT_SRC_MAP** : Used for specify that it is a map of source port blocked.
- **PORT_DST_MAP** : Used for specify that it is a map of destination port blocked.
- **COMPLETE_MAP** : Used for specify that it is a map of complete quintuple to be blocked.
- **TOTAL_STAT_MAP** : Used for specify an array map of statistics of dropped packets.

4.4.1 Proposed Solution

We have modified the mechanism of the injection of eBPF code defined inside PolyCube.

We analyse the code by looking for annotated map, and after the loading of eBPF code, we start a new eBPF program that performs the following operations:

- Look for an event of update or deletion on maps.
- When an event is triggered, check if the event comes from a map previously annotated.
- Extract the data from the event (e.g. IP added or deleted from the map).
- Send the data to userspace.

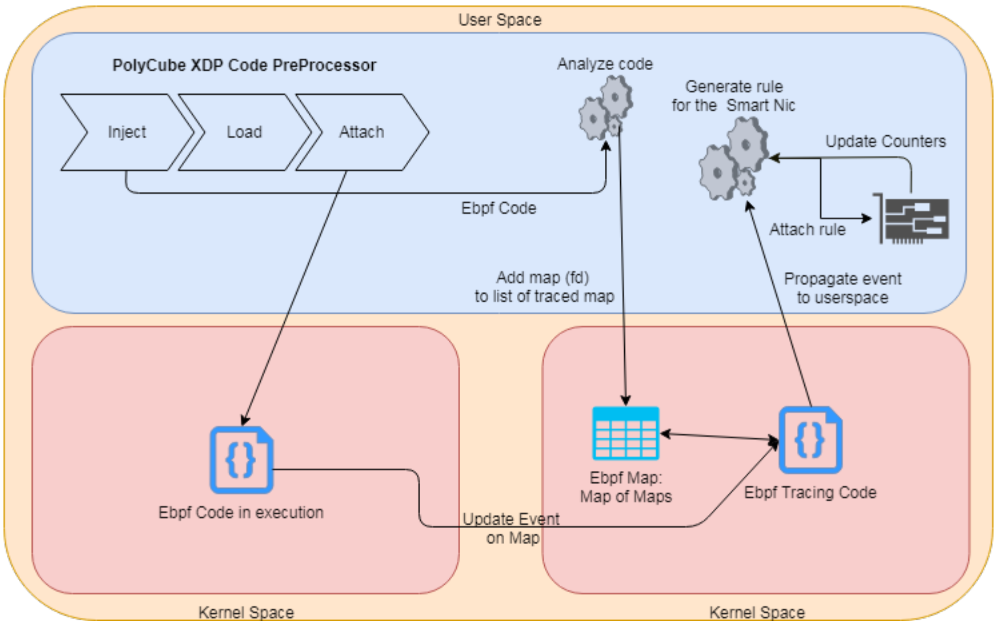


Figure 4.3: Proposed Solution

- Create or remove a TC flower rule.
- Update each 10 seconds the map counter and the statistics recovered from tc command output.

The analysis of code is performed at the beginning of *IOModuleXDP::inject* function, instead the injection of the eBPF monitor code is performed at the end of the function.

We need also to introduce another modification inside PolyCube framework. We need to keep trace of the physical interfaces to which an XDP program is attached to, because we need to check if the interfaces belong to a SmartNIC and specify it in TC commands.

4.4.2 Tracing Update/Delete Event on Map

With the aim to trace the event of an update or a delete of an element on map, we have developed an eBPF tracing program that relies on **tracepoint** mechanism provided by the kernel.

A tracepoint is a static probe point located in prefixed points inside the Linux kernel. It is possible to trace an event triggered on a tracepoint by registering a callback function to the tracepoint.

The callback function parameters of each traced tracepoint are defined inside the Linux kernel [14]. So far, tracepoints have provided a convenient way for intercepting kernel event for tracing and debugging purpose, nevertheless they cannot be used in future because this functionality will be removed in the next release of kernel.

The alternative introduced in the Linux kernel for tracing is called Kprobe, in the next chapters we will analyze it.

Listing 4.1: Tracing Attach Code

```
std::vector<std::string> cflags = {};
this->bpf = new ebpf::BPF();
auto init_res = bpf->init(this->BPF_PROGRAM, cflags, {});
if (init_res.code() != 0) {
        logger->error("Error␣init␣XDP␣Tracing␣Map␣Program:{0}",
        init_res.msg());
        return;
}
auto attach_res = bpf->attach_tracepoint("bpf:bpf_map_update_elem",
        "on_bpf_map_update_elem");
if (attach_res.code() != 0) {
        logger->error("Error␣attach␣XDP␣Tracing␣Map␣Program:{0}",
                init_res.msg());
    return;
}
attach_res = bpf->attach_tracepoint("bpf:bpf_map_delete_elem",
        "on_bpf_map_delete_elem");
if (attach_res.code() != 0) {
        logger->error("Error␣attach␣XDP␣Tracing␣Map␣Program:{0}",
                init_res.msg());
                return;
}
```

In the previous listing we have reported the code responsible of loading the eBPF tracing code and of attaching the tracepoint. The two functions to which we have attached the tracepoint are:

1. *on_bpf_map_delete_elem(struct bpf_map_delete_elem *p_elem)*
2. *on_bpf_map_update_elem(struct bpf_map_update_elem *p_elem)*

The parameters passed to these functions are defined as a struct and are described by the Linux kernel. This description is accessible from:

1. */sys/kernel/debug/tracing/events/bpf/bpf_map_update_elem/format*
2. */sys/kernel/debug/tracing/events/bpf/bpf_map_delete_elem/format*

Below we report the struct of map_update_elem:

Listing 4.2: struct bpf_map_update_elem

```
struct bpf_map_update_elem {
  u64 pad;            // First 8 bytes are not accessible
  u32 type;              // offset:8;     size:4; signed:0;
  u32 key_len;       // offset:12;        size:4; signed:0;
  u32 key;               // offset:16;    size:4; signed:0;
  bool key_trunc;    // offset:20;        size:1; signed:0;
  u32 val_len;       // offset:24;        size:4; signed:0;
  u32 val;               // offset:28;    size:4; signed:0;
  bool val_trunc;    // offset:32;        size:1; signed:0;
  int ufd;               // offset:36;    size:4; signed:1;
};
```

In this structure, there is not a unique identifier for each map inside the parameters passed to the tracepoint when an event occurs on the map. Indeed, we have in the tracepont the file descriptor but this is unique only at process level, hence if there are two processes with different eBPF code in execution, there could be events on tracepoint where different maps of different processes trigger the events having the same file descriptor.

Afterwards, we have bypassed this limitation with the kprobe approach, but at the time of developing of this solution, we decide to use tracepoints, as is, making the assumption that there is at most one eBPF program in execution inside the operating system.

Listing 4.3: Ebpf Tracing Code

```
struct bpf_map_update_elem elem;
bpf_probe_read(&elem, sizeof(elem), p_elem);
map_block_type *type=mapofmaps.lookup(&(elem.ufd));
if(!type){
        return 0;
}
data.type=*type;
data.fd=elem.ufd;
data.event = U;
...
bpf_probe_read(&data.key, sizeof(data.key), (char *)p_elem + offset);
data.key_len = len;
map_change.perf_submit(p_elem, &data, sizeof(data));
return 0;
```

In the previous listing we presented the eBPF code that traces the map update and as we can see it simply filters the event (looking for the file descriptor of current updated map into the map that contain all file descriptor of maps in who we are interested) and sends the data to user-space through a perf-ring buffer.

In the userspace program we keep track of events on maps and generate **TC rule** accordly. We also apply the rule to network interface and collect the statistics from the NIC and write them back on the map.

4.4.3 Drawbacks

Our implementation is strictly bounded to DDoS Mitigator use case, for this reason we define the structures of our integration keeping in mind the architecture of DDoS Mitigator.

However, we think that this approach introduces too many limits, because it forces the programmer to adopt a predefined set of structures (for example use a map for blocking source address etc...). Anyhow, for research purposes we decided to investigate this solution.

4.4.4 Performance

DDoS Mitigator with offloading support

In this section, we report the performance obtained with this approach. The

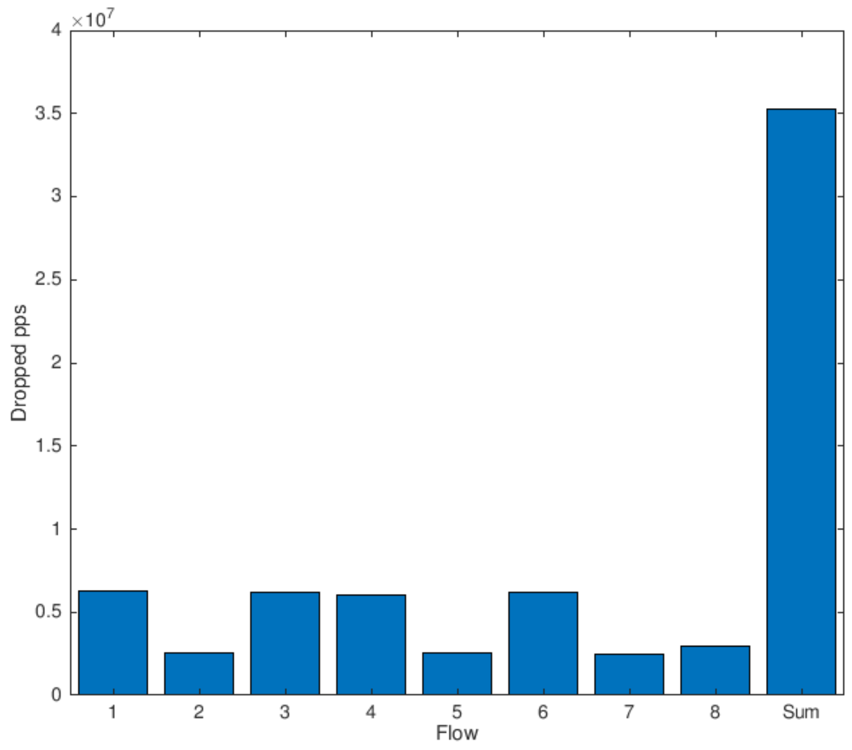


Figure 4.4: Tc PolyCube DDoS Dropped Packets Per Second with Offloading

parameters of generated traffic and the testbed are the same as those described before in section 4.3 and in section 3.3.

We reach the same result shown before of 35 million of packets dropped per second, as it is possible to see in the last column.

The non-uniform behaviour showed between different flows is due to the non-uniform generation of the traffic from the other host.

Indeed, the traffic distribution generated in the other host is not uniform as we can see from the output of pktgen.

```
Device: enp1s0f0@0 Result: OK:
  6259317pps 3204Mb/sec (3204770304bps) errors: 0
Device: enp1s0f0@1 Result: OK:
  2592791pps 1327Mb/sec (1327508992bps) errors: 0
```

```
Device: enp1s0f0@2 Result: OK:
  6228129pps 3188Mb/sec (3188802048bps) errors: 0
Device: enp1s0f0@3 Result: OK:
  5871284pps 3006Mb/sec (3006097408bps) errors: 0
Device: enp1s0f0@4 Result: OK:
  2560623pps 1311Mb/sec (1311038976bps) errors: 0
Device: enp1s0f0@5 Result: OK:
  6224274pps 3186Mb/sec (3186828288bps) errors: 0
Device: enp1s0f0@6 Result: OK:
  2562417pps 1311Mb/sec (1311957504bps) errors: 0
Device: enp1s0f0@7 Result: OK:
  3030245pps 1551Mb/sec (1551485440bps) errors: 0
```

The speed is not constant in all the flows but if we compare the generated rate of each flow we can see a similarity with the rate of each flow blocked at receiver shown in the Figure 4.4.

Here, instead, we report the overall CPU usage (all cores). We have recorded the CPU usage from the starting of the DDoS attack to the insertion of the last block rule.

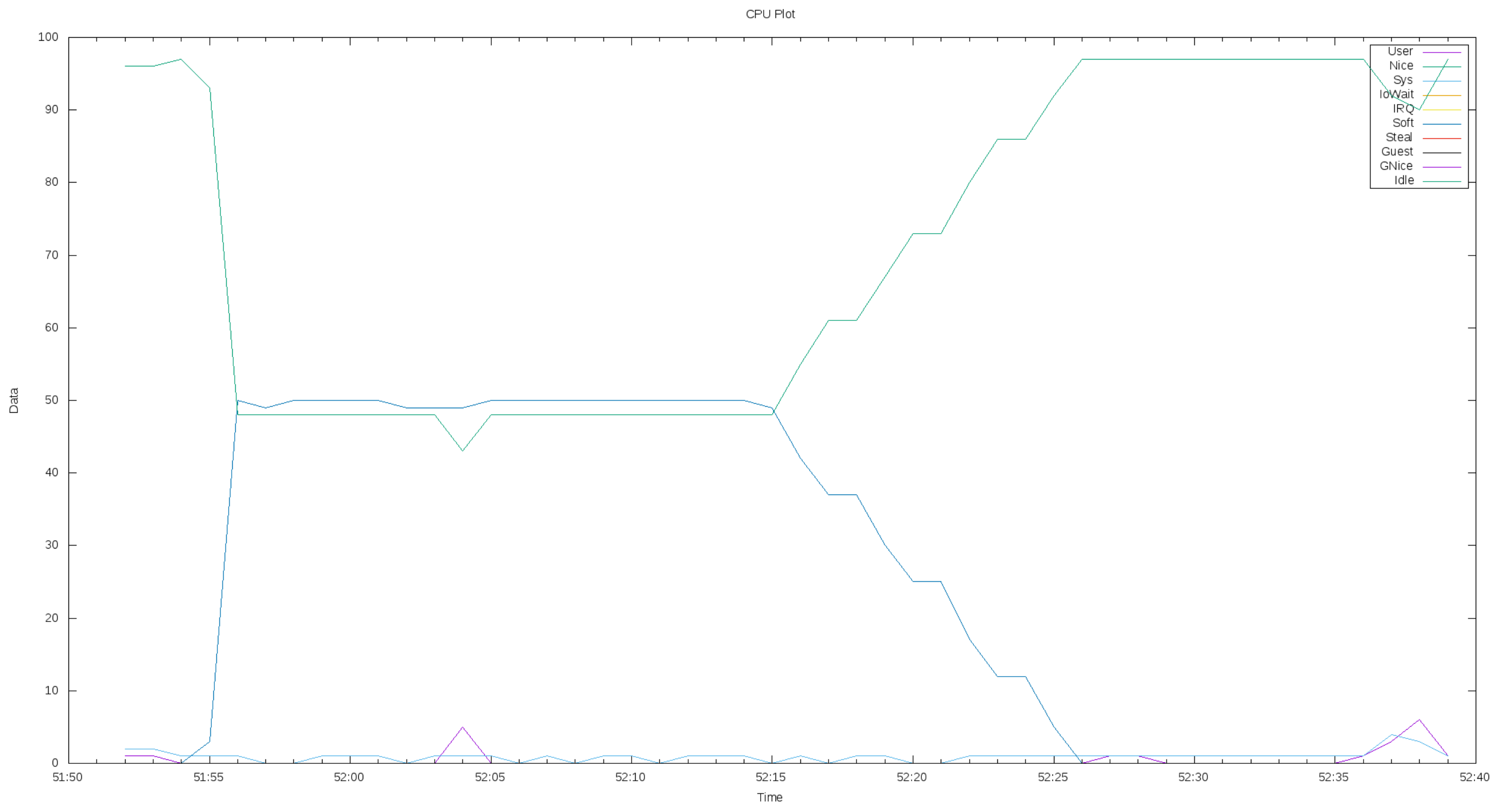


Figure 4.5: Tc PolyCube DDoS Cpu Usage w Offloading

When the DDoS attack starts, the CPU usage rises up very quickly and reaches a 50 % of CPU utilization.

Then we insert the rule for blocking the flows. We insert one rule every three seconds, and we can see that the CPU usage decreases accordingly. When all the rules are inserted, the CPU usage is near to zero.

This result is obtained thanks to the fact that the dropping operation is performed by the hardware and does not involve at all the host system.

DDoS Mitigator without offloading support

For comparison, we also report the same test executed with the original PolyCube DDoS Mitigator without offloading support.

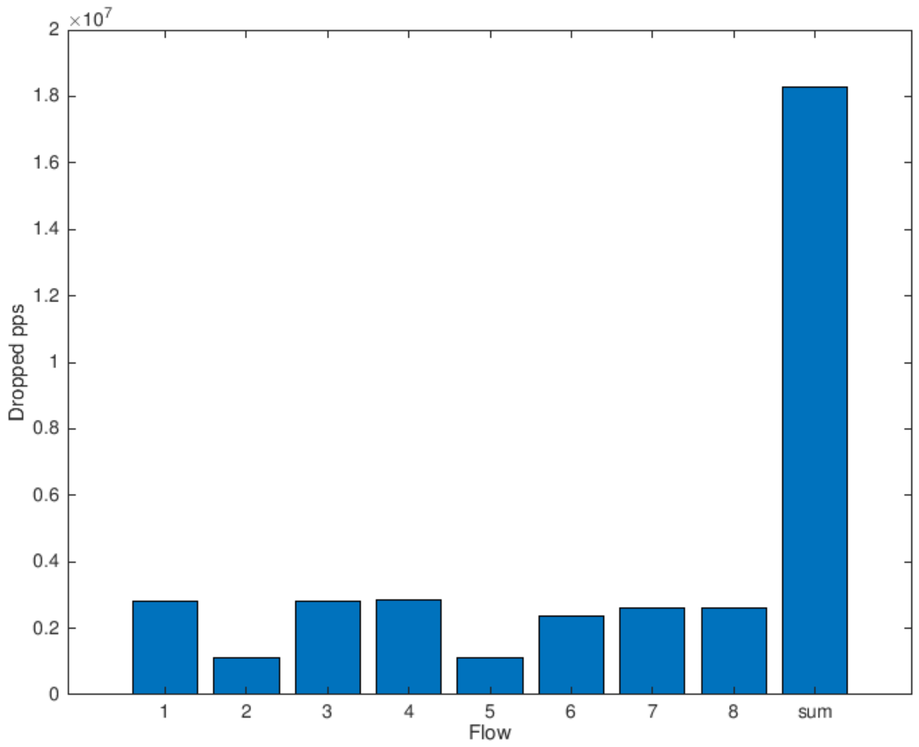


Figure 4.6: Tc PolyCube DDoS Dropped Packet per Second w/o Offloading

As we can see, we reach a lower result in terms of dropped packets per second. We also report the Cpu Usage. The result obtained is certainly lower compared to the offloaded case.

In this case, the CPU of the host is stressed into the activity of dropping packets.

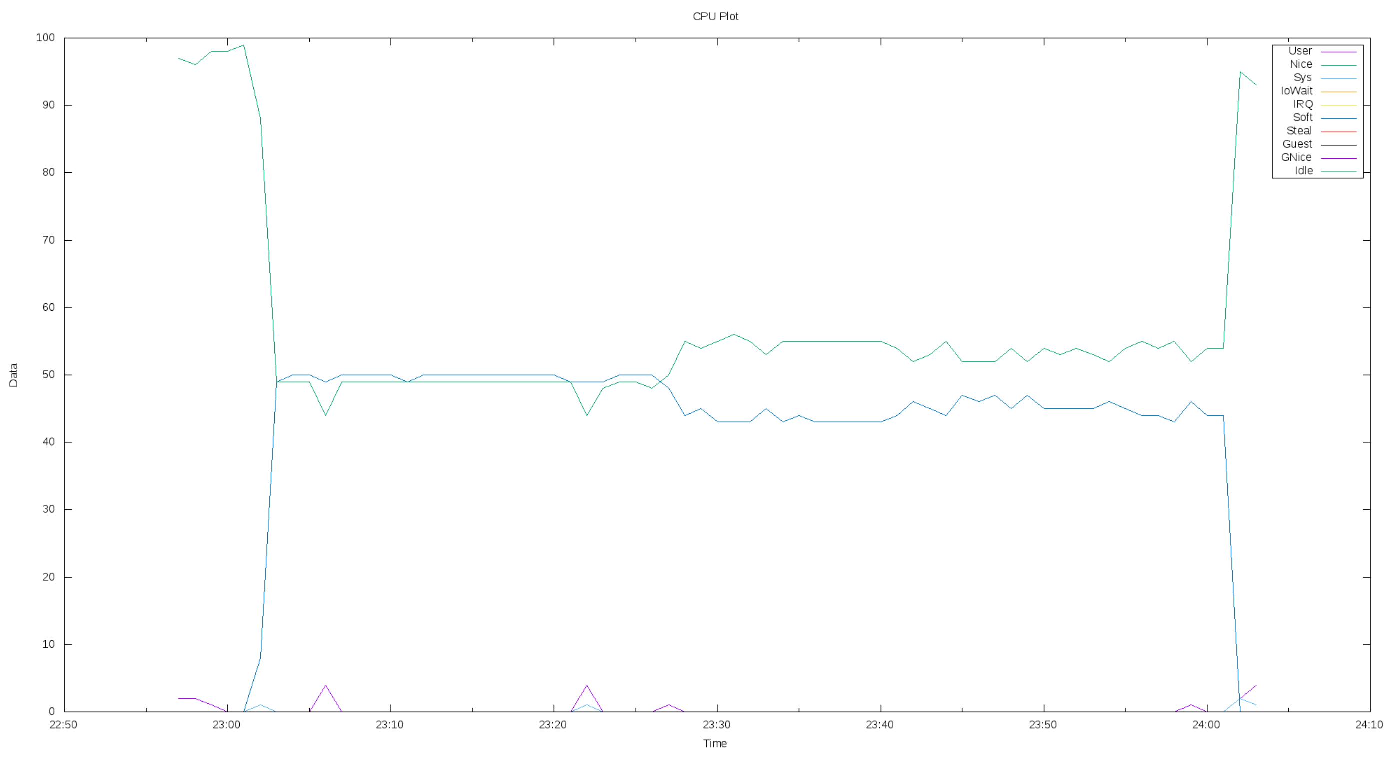


Figure 4.7: Tc PolyCube DDoS Cpu Usage w/o Offload

Chapter 5

Diving inside eBPF Code

The introduction of a programming paradigm based on explicit annotations of the source code forces the programmer to use a strict coding structure. This is not a limitation in some simple cases (like DDoS mitigator), but in complex programs a lot of problem arise.

With the aim to solve this limitation we start to explore the possibility of making a static analysis of eBPF code for extracting static or dynamic paths. **Static path** is the path of the code where the decisions are based on conditions wired into the code. **Dynamic path**, instead, is the path of the code where the decisions are based on external element values (e.g. contents of a map in eBPF case).

The theory behind the analysis of code is wide and detailed. In this work we try to use part of this theory for reaching our scope. In particular, the main concept we are focusing on is the concept of Control Flow Graph analysis, Abstract Syntactical Tree and Data Dependency Graph exploring.

5.1 Compiler and Interpreter

A **compiler** is a tool that takes a program written in a language that we call source language and translates into another language that we will call target language.

Generally, the source language is a high-level language and the target language is a low-level programming language. Basically, we will translate our eBPF program into a series of tc rules, so we can see our source language as the **eBPF language** and our target language as the **tc rule language**.

However, this is not perfectly accurate because it is true only in the case of a static eBPF program that has all the rule, which are translated from a logic wired inside the code, but in our case the decisions are based on a value on the map, therefore a rule must be generated for each update of an element on the map. If we see the problem from this point of view, our approach is closer to the work

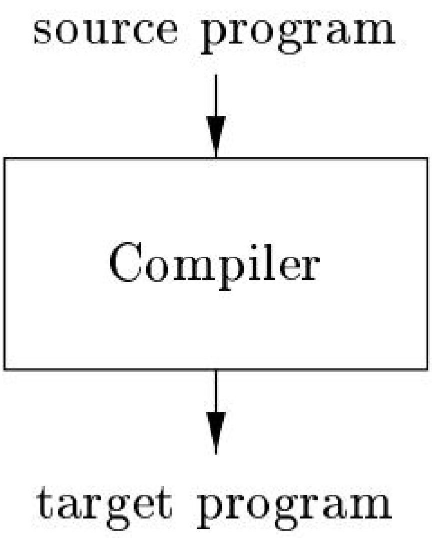


Figure 5.1: A Compiler [15]

that an **interpreter** does, taking the source code and the input of the program and producing an output at run-time. But if we return back to the compilers, it

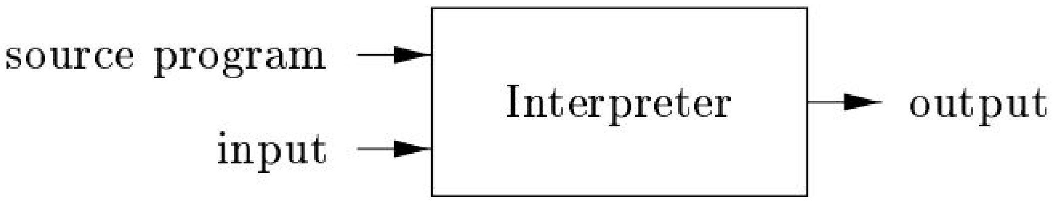


Figure 5.2: An Interpreter [15]

is interesting to analyse how a compiler works. The theory divides the compiler into a two big building blocks, one is responsible for the **analysis** part and one is responsible for the **synthesis** part.

Firstly, in our work, we focused on the **analysis** part that can answer to the question: *what the program does?* In the analysis part, the compiler produces an abstract representation of the program by breaking the program in atomic structure and by checking the coherence and the correctness of the source program.

During the phase of analysis, the compiler keeps also track of all symbols detected inside the code, for example variables. The compiler keeps these symbols into a special table called symbol table and during the synthesis phase it will use that table for making the translation.

5.2 Internal Structure of a Compiler

A compiler is a complex software composed by different modules, where each module is responsible for a part of the process of translation and optimisation.

The figure below shows all phases of the compilation process. In this section, we mainly focus on the Lexical Analyser phase, on the Syntax Analyser phase, and on Semantic Analyser phase.

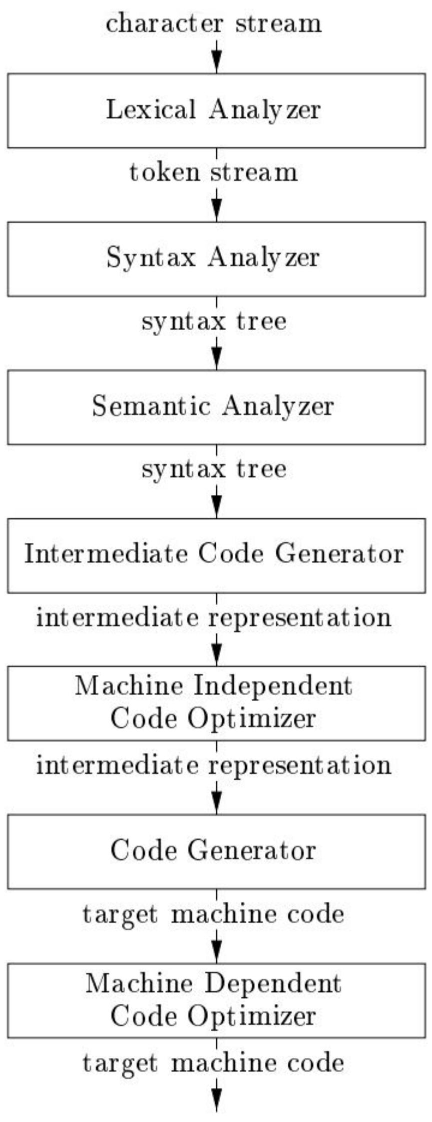


Figure 5.3: Compiler Structure [15]

5.2.1 Lexical Analysis

The first block that we found in all compilers is a lexical analyser that converts the raw series of source code character into a sequence of tokens.

```
res=var1+var2
```

If we consider the program defined up here, a lexical analyser will break down it into five lexemes: res, =, var1, +, var2. It will mark res,var, and var2 as an **identifier**, = as the **equal sign** and + as **plus sign**.

When the lexical analyser finds an unknown lexeme, such as a generic identifier, interacts with symbol table and inserts the lexeme into a table or recovers the related id if the lexeme has been previously discovered.

Another task performed by the lexical analyser is to remove white-space, new lines, and comments.

Generally, it is very difficult for a lexical analyser to raise an error due to the fact that any unknown sequence can be an identifier, but it is possible for example that this phase can fail for an invalid charset.

5.2.2 Syntax Analysis

In the syntax analysis phase, the compiler creates a tree-like structure that reflects the grammatical structure of the program.

The syntactical analyser is generally called **parser**, while the tree produced by the parsed is called syntactical tree. In the syntactical tree, each node with leaf represents an operation and each final leaf represents the argument of the operation.

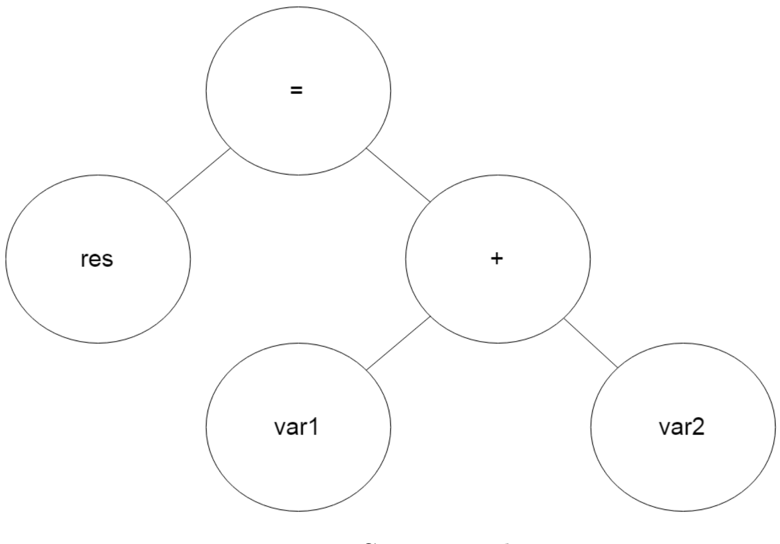


Figure 5.4: Syntactical Tree

In the previous figure, we presented the syntactical tree of the program shown in the lexical analysis section.

In the syntactical analysis, the compiler recognises error likes: undeclared variable, missing semi-column, etc.

5.2.3 Semantic Analysis

The main purpose of semantic analysis is to check the semantic consistency of code with the language definition. Type checking is performed during this phase (e.g. check if an operator has all required operands).

During this phase, another operation performed is *coercion*. This operation is responsible of implicit casting or of selection of correct operands. For example: a floating point number divided by an integer requires the conversion of the integer number into a floating and the use of divide operation of floating point numbers.

In this phase the compiler can recognise error likes: missing arguments, out of bound exception, etc.

5.3 Code Intermediate Representation

In this section we will present two of the most important structures for representing information regarding source codes.

1. **AST**: Abstract Syntax Tree
2. **CFG**: Control Flow Graph

In our project we will use all the above structures for analysing the code.

5.3.1 Abstract Syntax Tree

An abstract syntax tree is a syntax tree that reports in an abstract way the content of a source code written in a programming language.

The reason why it is called abstract is that the tree does not take in account semi-columns or bracket, because these tokens are implicitly defined by the the tree structure. This structure is built in the syntactical analysis phase and is enriched in the other phases.

Compared to direct analysis of the code, the use of an abstract syntactical tree brings a series of advantages:

1. Avoid to include unnecessary information (e.g. brackets, semi-column etc).
2. Possibility to annotate and store information regarding each node.
3. Store information regarding the position of a node and relation with original source code.

AST is an important structure widely used in compilers due to the fact that programming language generally can be ambiguous [16].
Below we report an Abstract Syntax Tree of a very simple program:

```
if(a==b){
    int c=10;
    myfunction(10,c);
}
```

In this simple program we find:

- An If statement.
- A variable declaration and initialisation.
- A function call.

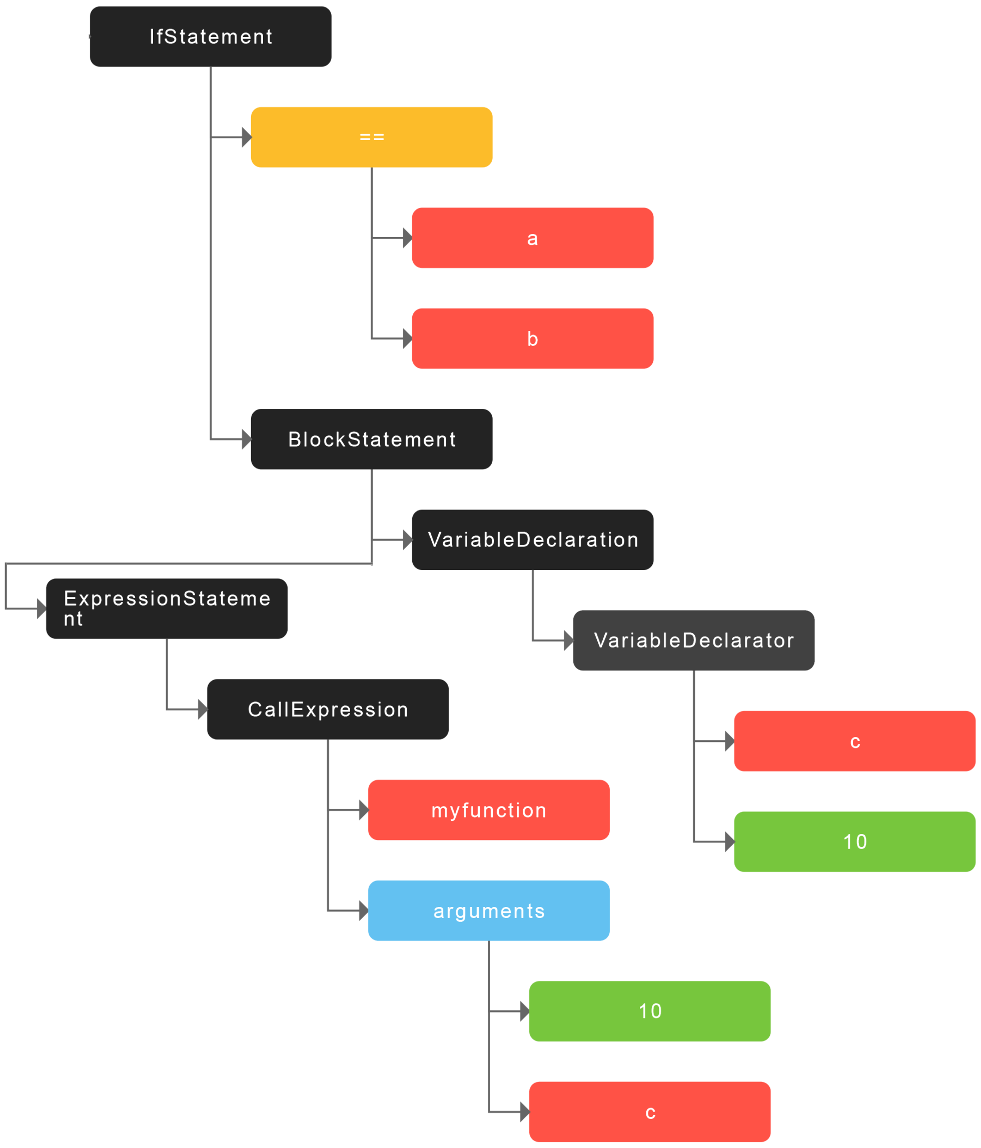


Figure 5.5: Resulting Abstract Syntax Tree

The root of AST is the *IfStatement*, one leaf of the statement is the condition (in this case the equality condition), and the other one is the *BlockStatement* that is the body of if statement. The equality condition has itself two leaves, one is the variable **a** and the other one is the variable **b**. The block statement inside has a variable declaration statement with a variable declarator and the name and value of the variable. Following we find the ExpressionStatement with the CallExpression as leaf and, in turn, with function name and arguments as leaves.

5.3.2 Control Flow Graph

A control flow graph (CFG) is a graphical representation of all the possible execution paths that can be traversed in a program during its execution.

In a control flow graph each node is a basic block composed by a piece of code without jumps. The edge instead is used for representing the jumps, and they interconnect 2 nodes.

Every control flow graph has 2 special blocks:

ENTRY : used to represent the starting point of a CFG

EXIT : used to represent the final point of a CFG

Control flow graphs are very useful for performing optimisation and static analysis and are widely used.

The detection of dead code (unreachable code) or of some types of infinite loops is performed with the help of control flow graph [17].

Following we report the control flow graph for a simple program.

```
int main(){

        for(int i=0;i<10;i++){
                if(i%2==0){
                        a();
                }else{
                        b();
                }
        }
}
```

This is a really simple program, however it contains a lot of elements that we will find in the Control Flow Graph.

In particular we will find:

1. The For statement (with initialisation of the variable, the increment and the comparison)
2. The If-Else statement
3. The two Function call

As previously described, in the following control flow graph we can find also the ENTRY and the EXIT nodes that correspond to the starting and the ending of the graph. The advantage of keeping an EXIT node is that we can traverse the graph backward even though in the program we have multiple exit points (e.g. multiple return statements).

Generally we put the trueness of the condition on the edge for mark and distinguish the True branch from the False one.

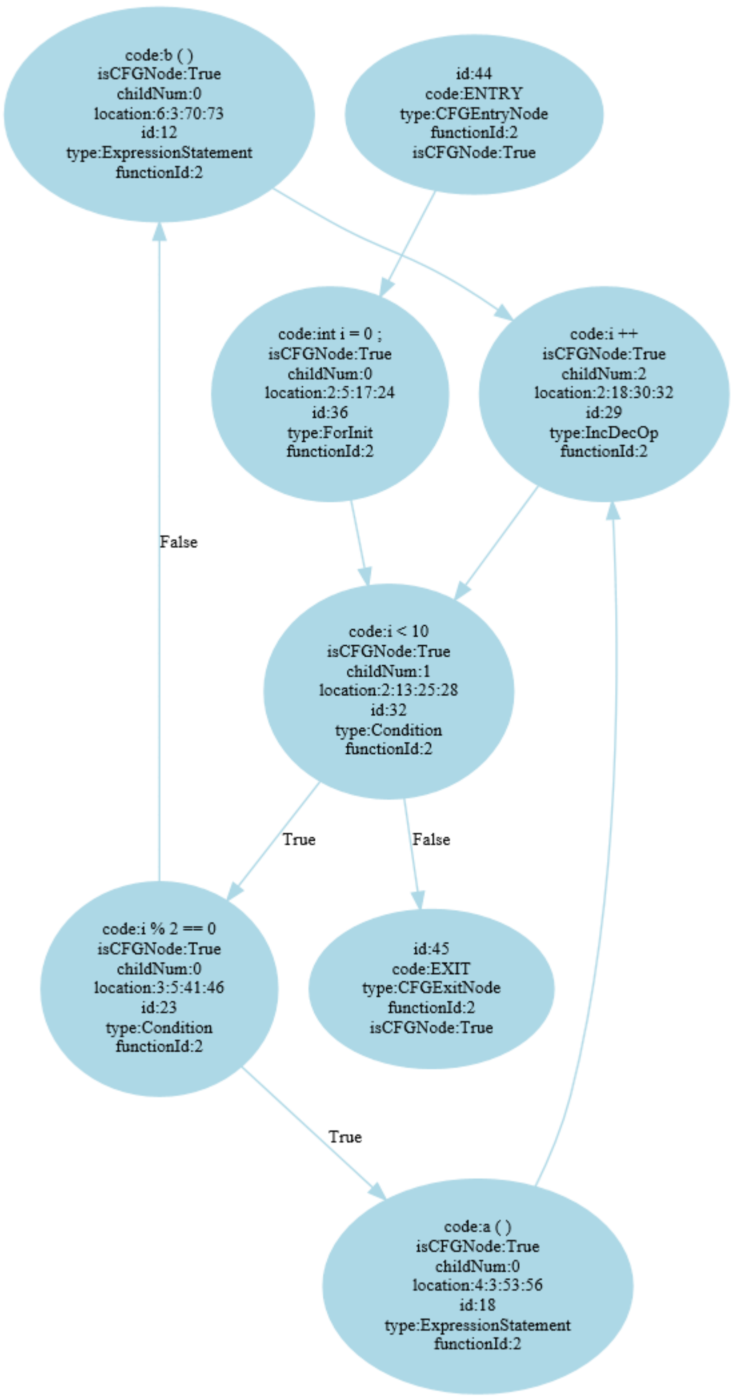


Figure 5.6: Resulting Control Flow Graph

Chapter 6

Kowalski Parser

So far, we have analysed the possibility of recognising and translating a source code of a programming language.

In this chapter, we will illustrate our developed solution that transforms XDP code into a set of Traffic Control rules.

6.1 Library Used

6.1.1 ANTLR

ANTLR[1] is ANother Tool for Language Recognition developed by Terence Parr. As the name said ANTLR is a parser generator that helps the programmer to automatise the construction of a language recognizer. It basically generates a program that writes programming language from a specification.

ANTLR takes as input a formal language description and generates a program that can recognize a program written in the language that we have specified. The strengths of ANTLR are:

- Human-Readable code generated and easily extensible.
- Extremely flexible.
- Easily to define a new grammar.
- Multiple target languages ex: Java, C#, Python etc.
- Open-source, written in Java and a very active project.

[1]http://www.antlr.org/

ANTLR is widely used for building translator and interpreter for domain-specific languages, which are languages related to a specific task. ANTLR automatically generates the parser and the lexical analyser by analysing the grammar provided in input.

The remaining phases of translation are left to the programmer, but it is possible to define inside ANTLR grammar some actions that might be executed when the parser recognizes the particular expression.

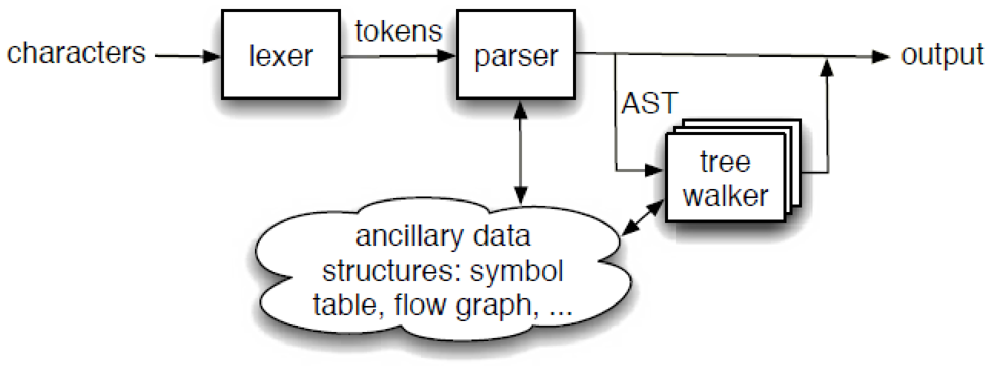


Figure 6.1: Overall ANTLR processing flow [18]

ANTLR uses the abstract syntax tree as an intermediate representation.
The previous figure illustrates how basically ANTLR works. The lexer breaks the input character stream into a series of tokens, then passes this token to the parser that tries to recognize the structure.

After this we have many choices, we can directly produce an output, we can store the token information into some structure or we can build an AST and perform more complex operations [18].

6.1.2 Joern-ANTLR

Joern is a tool developed by Fabian Yamaguchi. It is an intelligent code analysis platform for C/C++ built on the top of ANTLR. The main scope of Joern is extracting control flow, data flow, and other graph type representation from C source code.

Joern uses a graph database for storage and for searching purpose, and uses the Gremlin language for graph traversal [19].

We would like to use Joern in our project because it avoids the generation of all graph structures by hand. Nevertheless, we have considered that the use of a graph database implies a complication of the architecture of our solution. In particular, Joern generates a database for an old version of Neo4J that is incompatible with the newest version.

This is the reason why we start to look if there is a solution similar to Joern totally written in Java that does not require a graph database as a backend. We found joernANTLR [2]: a completely rewritten version of Joern for Java. We decided to use this project and to extend it to implement our solution.

This project allows to recover:

1. Abstract Syntax Tree (using ANTRL)
2. Control Flow Graph
3. Data Dependency Graph

So it fits perfectly for our scope.

6.1.3 ProcBridge

ProcBridge[3] is a lightweight socket-based IPC (Inter-Process Communication) protocol between Java and Python programs. The purpose of the project is to provide a simple and convenient way to implement inter-process applications.

We use ProcBridge for communication between the Python tracing program and the Java developed compiler.

[2]https://github.com/electricalwind/joernANTLR

[3]https://github.com/gongzhang/proc-bridge

6.2 Main architecture

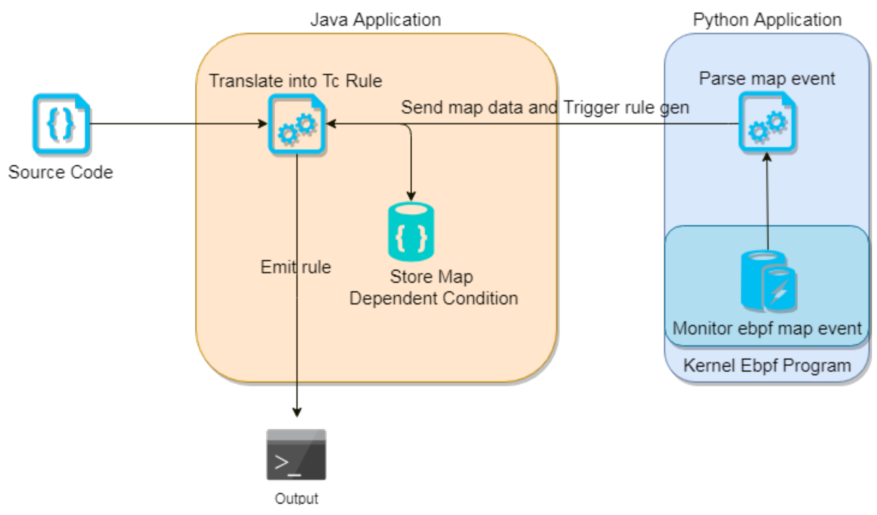


Figure 6.2: Conceptual Architecture

The previous figure illustrates the overall conceptual architecture of our software. As we can see, the software is divided into two main blocks.

The scope of the first block, the one written in Python, is to keep track of all kernel events related to the creation, the deletion and the modification of ebpf maps.

The second block, instead, is responsible of translating our source code into a series of Tc Flower rules.

6.3 Java Module

The main component of our application is the Java module that executes the following task:

1. Initialize ANTRL lexter and parser with grammar
2. Read the input eBPF source code and pass it to the lexer
3. Pass the output of lexter to the parser
4. Create the Abstract Syntax Tree
5. Create the Control Flow Graph and the Data Dependency Graph from the AST

6. Expand the Control Flow Graph into a series of separated branches for each possible path

7. Explore all possible branches and translate the code and the taken decision into a TC flower rule

8. Store the incomplete information related to maps for further decision

9. Generate the rules when the Python program sends the events.

The structure of the program is clearly visible from the main function reported below:

Listing 6.1: Main Function

```java
public static void main(String [] args) throws IOException {
      MapTracer mapTracer=new MapTracer();
      mapTracer.start();
      ANTLRParserDriver parser = new ANTLRCModuleParserDriver();
      TestASTWalker walker=new TestASTWalker();
      parser.addObserver(walker);
      ANTLRInputStream inputStream=new ANTLRInputStream(
        Files.readAllLines(
        Paths.get("test.c"))
              .stream()
              .collect(Collectors.joining("\n")));
      ModuleLexer lexer=new ModuleLexer(inputStream);
      TokenSubStream token= new TokenSubStream(lexer);
      List<ASTNode> listnode=walker.codeItems;
      HashMap<String, CFG> cfgForFunction = CfgFactory.create(listnode);
      CFG main=cfgForFunction.get("xdp_prog1");
      CodeFlows expandedpath= PathExpander.expand(main);
      DDG ddg=DDGFactory.create(main);
      expandedpath.explore();
}
```

Map tracer is the class that provides the Python IPC and the map event triggering. **Module Lexer** and **TokenSubStream** are the lexter and the parser. The aim of **CfgFactory** is to create the Control Graph. We can see that it returns a HashMap of strings and of CFGs. This is because the CFG implementation of Joern is limited to the function scope, so each function has its own CFG.

For this reason we extend the CFG generator for keeping track of the CFG and the related function name. During the next phases we look for a fixed name (the main function of eBPF program).

Lastly we expand and explore each single code path.

6.3.1 Path Expander

The path expander function adopts a recursive approach, the function returns when it finds a **CFGExitNode** or a **ReturnStatement**.

The function parses the current statement and then looks for all the outgoing edge of the control flow graph and parses the next statements storing the trueness of the encountered condition if it is a boolean condition

Listing 6.2: Path Expander

```
private static CodeFlows parseNextNode(CFG cfg, CFGNode node){
        if(!(node instanceof CFGEntryNode) &&
                (node instanceof CFGExitNode ||
                ((ASTNodeContainer)node).getASTNode()
                instanceof ReturnStatement)){
            CodeFlow flow=new CodeFlow();
            flow.add(node);
            CodeFlows flows=new CodeFlows();
            flows.add(flow);
            return flows;
        }
        CodeFlows allflows=new CodeFlows();
        cfg.outgoingEdges(node).forEach(cfgEdge -> {
            CodeFlows flows = parseNextNode(cfg, cfgEdge.getDestination());
            if(node instanceof ASTNodeContainer
              && ((ASTNodeContainer)node).getASTNode()
              instanceof Condition){
                ASTNodeContainer astNode= (ASTNodeContainer) node;
                Boolean condtionalPath=cfgEdge.getLabel().contains("True");
                flows.forEach(flow -> flow.add(new ASTNodeContainer(
                  astNode.getASTNode(),
                  astNode.getConditionalStatement(),
                  condtionalPath)));
            }else{
                flows.forEach(flow -> flow.add(node));
            }
            allflows.addAll(flows);
        });
        return allflows;
```

With this approach we take the CFG, which is a graph structure and we explode it into a series of paths. Indeed, CodeFlows is a list of path where each node of the path is an element of the AST.

It is easy to notice that the loops are not supported by using this approach, but this is not a big problem because also eBPF does not support the loops (it requires the unrolling of loops). However, if in our eBPF program we define the loop with the **#pragma unroll** notation, we can unroll it with into pre-compiler stage and then pass the result to our program.

6.3.2 Helper Function

Before introducing the expander phase we need to clarify some aspects of our translation process.

The eBPF code is composed by a C complete syntax (with some limitation in term of operations) and this result into a very complex syntax to parse. In particular, we have found very difficult to reconstruct the flows that bring a condition to be true or false.

This difficulty comes principally from the management of variables and from the use of pointers of C language, this complex structure makes hardly also detecting to which field of the packet we are referring too.

The other problem is that we need to guarantee that if the offloaded rule has a **drop** action as result, the semantics of the program does not change. So, basically, we need to guarantee that the path of the control flow graph interested from the rule is stateless. But we cannot assure that, if we detect the changes on a map and the value of the map is assigned to a pointer and we change the value of the pointer dereferencing it, it will be quite impossible to track this modification.

For this reason and also for keeping the architecture simple we define a list of function for compare packet fields and for access to maps. We have implemented these functions in eBPF language and we assume that the programmer uses our functions or at least others semantically equivalent.

Function	**Parameter**	**Return Value**
match_src_mac	Packet, expected source mac	1 if equals, 0 if not
match_dst_mac	Packet, expected destination mac	1 if equals, 0 if not
match_ip_proto	Packet	1 if the packet is IP, 0 if not
match_src_ip	Packet, expected source ip	1 if equals, 0 if not
match_dst_ip	Packet, expected destintation ip	1 if equals, 0 if not
match_tcp_proto	Packet	1 if the packet is TCP, 0 if not
match_udp_proto	Packet	1 if the packet is UDP, 0 if not
match_src_port	Packet, expected source port	1 if equals, 0 if not
match_dst_port	Packet, expected destintation port	1 if equals, 0 if not
match_src_mac	Packet, Matching map	1 if the map contains the mac as key, 0 if not
match_dst_mac	Packet, Matching map	1 if the map contains the mac as key, 0 if not
match_src_ip	Packet, Matching map	1 if the map contains the ip as key, 0 if not
match_dst_ip	Packet, Matching map	1 if the map contains the ip as key, 0 if not
match_src_port	Packet, Matching map	1 if the map contains the port as, 0 if not
match_dst_port	Packet, Matching map	1 if the map contains the port as key, 0 if not

Table 6.1: Helper Function

In the previous table, we have reported all the implemented functions, they will cover the most used operation that can be translated into a TC rule.

6.3.3 Path Explorer

The expander path is the most complex part of the program, so we will analyze it breaking into different parts.

Listing 6.3: Check Unexpected Access

```
this.forEach(currentpath->{
   Collections.reverse(currentpath);
   CodeFlow path=currentpath;
   if(path.hasUnsafeAccess()) {
      path=path.trimToUnsafeAccess();
   }
   if(path.hasUnpredictableDecision()){
      path=path.trimUnpredictableDecision();
   }
}
```

The explorer stage is executed for each path previously discovered.

Firstly, we reverse the path because during the construction we have built it from the ending point to the starting point with the recursive method. Then we check if the path contains some statement that is **unsafe**, with unsafe notation we mean that the code inside the path access to maps through a direct function, so we cannot determine if the function is stateless or not.

If we detect a map access, we trim the path to the statement before the action and consider it as it returns XDP_PASS, this because with this mechanism the SmartNIC lets pass the packet and the host XDP software will be executed and can decide the action to take on the packet with the consciousness of the state and with the possibility to modify it.

To detect a map access we search into the statement for these functions:

- bpf_map_lookup_elem
- bpf_map_update_elem
- bpf_map_delete_elem

In the same way, we detect the unpredictable decision, which is a decision taken on the basis of something that we cannot understand. In particular, in this function, we check if there were these types of unsupported statements:

- Loop Statement (For, While and Do)
- Switch Statement (this in future can be supported)

- Non-structured programming constructs (Goto, Continue and Break)

We assure also that *IfStatement* takes the decision on a boolean condition that is a combination of previously defined helper functions.

The trimming procedure is identical to the previous illustrated.

6.3.4 Condition Expander

This phase starts with this line of code: *path.findIfStatementCondition()*. We look for If statements and analyzing it keeping track of all condition that we will find and of their trueness.

In *exploreCondition()* we start the process of translation. First of all, we need to specify that for each condition we can have multiple TC flower rules in output.

This behavior is given by the fact that the *Condition* can contain some *OrExpression* and the TcFlower does not have a construct that translates an OrStatement. For this reason, we split the *OrExpression* into a set of separate rules.

The **exploreCondition** function relies on the list of condition finded previously. Firstly, it finds the return actions looking for a **ReturnStatement** then translate this statement into a ReturnAction. We will use this action at a later stage for translating it into a TC action.

Right now the supported return statements are:

- XDP_DROP
- XDP_PASS

Listing 6.4: Return Action Finder and Translation

```
List<ReturnAction> returnActions = this.getASTNodeStream()
  .filter(astNode -> astNode instanceof Statement)
  .map(astNode -> (Statement) astNode)
  .filter(statement -> statement instanceof ReturnStatement)
  .map(ReturnAction::new).collect(Collectors.toList());
if(returnActions.size()!=1){
        throw new RuntimeException(
    "Multiple_return_actions_in_same_flow");
}
ReturnAction returnAction=returnActions.get(0);
```

We do not support multiple return statements in the same path because it is unreachable code, so it indicates a problem in the source code.

After detecting the return action, we start to explore the If statements and we look to their condition. Then we start the process of translation. We expand recursively the IfStatement conditions looking for AndExpression, OrExpression or CallExpression.

The final condition of the recursion is a CallExpression, indeed if we find this type of expression we can look to which type of function is begin called and translates it into a piece of TcRule.

Listing 6.5: Call Expression Translator

```
if(expression instanceof CallExpression){
 CallExpression callExpression= (CallExpression) expression;
 HelperCallExpression hc=new HelperCallExpression(callExpression,condition);
 return Collections.singletonList(hc.toFlowerRule());
}
```

HelperCallExpression is the class that parses our *CallExpression*, extract the argument and detect the type of function that we have called from the ones defined in Table 6.1.

Instead, if we detect an *AndExpression*, we call the same function on the left and on the right argument and we combine the result obtained into a single rule. The *OrExpression* introduces slightly more complexity because we need to split the flow into three separate flows: in the first we consider the first expression true and the second false, in the second we consider the first false and the second true and lastly we consider both true.

So we produce three pieces of TC flower rule that need to be aggregated with other rules separately. When we aggregate a rule, we merge the conditions that bring to the execution of the rule.

It is clearer if we look to this example:

```
if(match_src_mac(data,AABBCCDDEEFF)&&match_dst_mac(data,DDEEFFAABBCC)){
        return XDP_DROP;
}
```

In this case, our program builds two pieces of rule separately:

1. src_mac AABBCCDDEEFF
2. dst_mac DDEEFFAABBCC

And then aggregate the two rules into a single one:

src_mac AABBCCDDEEFF dst_mac DDEEFFAABBCC

But if we look to this other example:

```
if(match_src_mac(data,AABBCCDDEEFF)
        &&(match_dst_mac(data,DDEEFFAABBCC)
        ||match_dst_mac(data,DDAABBCCEEFF))){
        return XDP_DROP;
}
```

The OR condition will produce three separate rules:

1. dst_mac DDAABBCCEEFF
2. dst_mac DDEEFFAABBCC
3. dst_mac DDAABBCCEEFF dst_mac DDEEFFAABBCC *(never occurs!)*

These rules aggregated with other rule of the AND condition produce a set of 3 rules again:

1. src_mac AABBCCDDEEFF dst_mac DDAABBCCEEFF
2. src_mac AABBCCDDEEFF dst_mac DDEEFFAABBCC
3. src_mac AABBCCDDEEFF dst_mac DDAABBCCEEFF dst_mac DDEEF-FAABBCC *(never occurs!)*

However during this phase we only store the atomic pieces that compose the rule, the creation of a TC complete rule is performed lately with the aim of add also the priority.

At the end of this process, for each path, we will find one or more Flower Rules, which contains:

- A list of pieces of TC Flower rules.
- A priority (established during the creation in a global way) with the principle that the first shorter path is the one with the highest priority.
- A return action.

At this point we are ready to generate the TC rules. We generate the rule invoking the FlowerRule method *getRule()*. The first operation performed by this method is the conversion of data from the internal C representation into a string format.

Let's consider the case of IP address: the TC flower syntax requires a string type dotted decimal representation of the ip address, but in the code, we write the address as an unsigned long integer (uint32_t) in network byte order. For this reason, we need this procedure of conversion.

After doing that we need to order the rules, indeed, the order of a rule inside a TC flower command is important.

When we build this command we need to follow this order:

1. protocol ip
2. flower skip_sw (required! added before the conversion)
3. src_mac

4. dst_mac

5. src_ip

6. dst_ip

7. ip_proto tcp OR ip_proto udp

8. src_port

9. dst_port

We must follow this order otherwise the rule will not be accepted from tc program.
This is the reason why we firstly store all the elementary pieces of the rules and then produce the complete rule.

With this mechanism, we can order them and detect duplicates or impossible conditions (ex. 2 dst_mac together in the same rule as seen before).

6.3.5 Map Depended Condition

If we encounter a helper that bases its decision on a map, we store the map name and we do not emit a rule.

The rule is emitted only when an event on the map is triggered. In particular we emit a new rule for each updated or inserted element on the map and we emit a rule deletion command for each deleted element on the map.

When we detect a rule dependent from a map value, we subscribe two callbacks to a structure that reflect the kernel map structure.

We keep this structure updated listening into the Python program for kernel events and we trigger the callbacks when an event occurs. We keep also an identifier inside the rule for rule naming, with handle field of TC flower, so we can delete safely the rule when it was required.

The Java structure that reflects the kernel maps have:

1. A name that identify the map

2. A map type

3. A key size

4. A list of keys

The Map Tracer class is responsible for the communication with the Python module. We use ProcBridge for this communication. ProcBridge is a library that relies on a

Request-Response model. The server module defines a set of APIs and the client invokes them as they invoke local functions.

All the communication use JSON object as intermediate representation of data. In our case the Java program act as server and the python program as client. We have defined two APIs one is for update events and one is for delete events. We have not defined an API for insertion because we consider insertion as a particular case of the update API.

When an API was triggered we decapsulate the data and execute the same operation on our Java representation of the map. With this approach when a field on the map is updated, it will call the interested callback in an automatic and transparent way.

As example if we have the following code:

```
if(match_map_src_ip(data,mysourceipmap){
        return XDP_PASS;
}
```

We subscribe a callback for an update and a delete event on the Java map *mysourceipmap* and when a new update event occurs, for example with the insertion of a new ip 192.168.33.12, we generate the rule with:

src_ip 192.168.33.12

6.4 Python Module

We have developed a tracing application that uses BCC tools for tracing the map event inside the kernel. The Python application simply takes the event from the kernel space, convert the C type and send the event to the Java program. The mechanism of tracing instead is implemented inside the eBPF code.

First of all, we remember that in the Chapter 4 we have talked about tracing of map event used tracepoint.

In that occasion, we illustrate the drawbacks and the limitation of using the tracepoint approach, this type of approach indeed is limited to one eBPF program at the time, otherwise, it can be possible that we mix up event from different maps of different programs without the ability of separate it.

This because with tracepoint we can only rely on the file descriptor as the identifier of a map, but file descriptors are not unique for different programs. We discovered also that tracepoint will be removed in the next version of the kernel so we decided to move on and use kprobes.

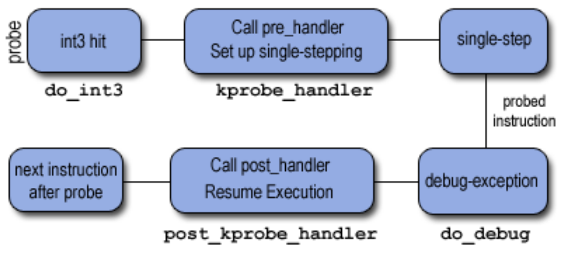
6.4.1 KProbes

Kprobes is the new debugging way introduced in the Linux kernel that can be used for monitoring events, collecting statistics, detecting faults, etc. A kernel probe is a set of handlers placed in a particular instruction address. There were two main types of probes KProbe and JProbe.

KProbe is composed by a pre-handler and a post-handler the first one is executed before the execution of the instruction, the second one later. JProbe instead has only one handler that must match the argument of the function that we want to trace, because it grants to us the access to that parameter [20].

Unfortunately, eBPF does not support JProbe, but generally the Kprobe pre-handler and post-handler is wrapped around the call to the interesting functions. This allows the programmer to access the parameters of the function, which are in the register at the moment where the Kprobe is triggered.

Indeed, when a Kprobe is triggered the context of the processor is copied inside a structure that can be accessed by the Kprobe. It is also possible with the post-handler to recover the return argument of the function.



Execution of a KProbe

6.4.2 KProbing Map Function

The most interesting point where we can trace an event on an eBPF map is surely the eBPF helper function related to maps. In particular, there were two helpers that we are interested in, the first is *bpf_map_update_elem* and the second is *bpf_map_delete_elem*.

These helpers take as arguments:

1. A *struct bpf_map map* that is a structure that contains the map relative information like map name, id, key size etc.

2. A key

3. A value for that key

However, it is impossible to trace this type of functions due to the fact that there are not a real functions but they are pointers to a specific functions of the specific map type. This requires a solution developed for each type of maps and in our tests we decided to focus on hash-map related functions.

The advantage of tracing these functions is that we can intercept event on a map either if there were triggered by userspace either from kernel-space. Moreover, the access to the eBPF map structure as a parameter of the function gives us the possibility to recover in a simple way the map name that we use to recognize the maps and to separate the event between different programs.

Nevertheless also tracing these functions is not a viable solution, because we can trace only the lookup function. Indeed, due to kernel development design choice, is impossible to attach kprobe on the update and delete functions.

This was clearly stated in the kernel with this comment:

```
/* must increment bpf_prog_active to avoid kprobe+bpf
* triggering from inside bpf map update or delete otherwise
* deadlocks are possible
*/
```

This is the reason why we decided to attach our Kprobes to the only viable functions that are the **syscall** for map update element and map delete element.

The problems of using this approach are the same as the tracepoint approach. We have in the argument only the key, the value, and the file descriptor. Moreover, we can trace only userspace events.

We have solved the first problem, with a solution that considers the tracing of the event of map creation. Our idea is that if we can trace the event of map creation, we can recover the relation that bounds map name with file descriptor and pid of a process. So we intercept the *bpf_map_new_fd* function, which is responsible for the assignment of a new file descriptor to a map, we collect all the information and store it into a map for the later usage.

The first parameter passed to the *bpf_map_new_fd* is *struct bpf_map map* and the return value is the file descriptor. Lastly, we can recover the current pid inside the eBPF code with a specific helper. With this approach, we have solved the problem of keeping a mapping between map structure, file descriptor and process id.

However, this approach works only if we trace the event of the creation of the maps, so only if our target program is executed after our program is attached.

6.4.3 eBPF Tracing Code

Listing 6.6: Structures and maps

```
struct data_t {
    char map[16];
    u32 map_type;
    u32 key_size;
    u64 key;
    char type;
};
struct fd_pid {
    u32 fd;
    u32 pid;
};
BPF_PERF_OUTPUT(events);
BPF_HASH(currmap, u32, struct bpf_map);
BPF_HASH(mapidfdpid,struct fd_pid,struct bpf_map);
```

In the previous listing, we report the structures and the maps used. The *data_t* structure contains the information that we send to userspace on an update or a delete event. We send:

- The map name
- The map type

- The key Size
- The key
- The event type (Update or Delete)

The *fd_pid* instead is the struct that we use for keeping the mapping previously described. Then, we use the *perf_output* for sending the event to userspace, the *currmap* to trace the execution of the *bpf_map_new_fd* function and the *mapidfdpid* to store the mapping.

To the *bpf_map_new_fd* function we attach two KProbe one is the pre-handler called in eBPF KProbe and the other is the post-handler called KRetProbe. This is the main difference between **Kprobe** and **Tracepoint**: we can access both to the return value and to the function parameters. From the pre-handler, we recover the function parameter and store it into the temporary map as following:

```
int kprobe__bpf_map_new_fd(struct pt_regs *ctx, struct bpf_map *map){
        u32 pid = bpf_get_current_pid_tgid();
        struct bpf_map map_dump={};
        bpf_probe_read(&map_dump,sizeof(struct bpf_map),map);
        currmap.update(&pid,&map_dump);
        return 0;
}
```

Then, we recover the return parameter of the function and collect the previous recovered map structure from the temporary map for creating the association that we need.

```
int kretprobe__bpf_map_new_fd(struct pt_regs *ctx){
        int ret = PT_REGS_RC(ctx);
        u32 pid = bpf_get_current_pid_tgid();
        struct bpf_map *map = currmap.lookup(&pid);
        if(map==0){
                return 0;
        }
        struct fd_pid fd_pid={};
        fd_pid.pid=pid;
        fd_pid.fd=ret;
        mapidfdpid.update(&fd_pid,map);
        return 0;
}
```

This mechanism of passing data between kprobe and kretprobe with a temporary map is widely used because it relies on the working principle of kprobe that guarantees that for the same pid the kprobe and the kretprobe are executed in sequence.

Lastly we report the update tracing kprobe:

```
int add(struct pt_regs *ctx, union bpf_attr *attr) {
    struct data_t data = {};
    u32 pid = bpf_get_current_pid_tgid();
    struct fd_pid fd_pid={};
    fd_pid.pid=pid;
    fd_pid.fd=attr->map_fd;
    struct bpf_map *map =mapidfdpid.lookup(&fd_pid);
    if(map==0){
            return 0;
    }
    __aligned_u64 key=attr->key;
    bpf_probe_read_str(&(data.map),16U,map->name);
    data.map_type=map->map_type;
    data.key_size=map->key_size;
    void __user *ukey = u64_to_user_ptr(key);
    if(data.key_size<=8){
        bpf_probe_read(&data.key, data.key_size, ukey);
    }else{
        return 0;
    }
    data.type='U';
    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
```

We recover the struct map related to the current map and the key and we send all the data to userspace.

For recovering the key data, we need to cast the pointer to an user_ptr this because the key value during the system call is passed through the user address space.

6.5 Examples

Following we report some coding examples with relative tc flower translation.

6.5.1 Simple code without map access

Here, we describe the case of a simple program with address encoded into the program and without access to maps. Note that IP addresses are encoded into network byte order.

```
int xdp_prog1(struct xdp_md *ctx){
    if(match_ip_proto(ctx)){
        if(match_src_ip(ctx,0x1501A8C0)){
              if(match_tcp_proto(ctx)){
                if(match_dst_port(ctx,22)){
                    return XDP_PASS;
                }
            }
            return XDP_DROP;
        }else{
            if(match_dst_ip(ctx,0x1506A8C0)){
                return XDP_DROP;
            }
        }
    }
    return XDP_PASS;}
```

This program is translated into the following series of TC rules:

1. prio 65530 protocol ip flower skip_sw src_ip 192.168.1.21 ip_proto tcp dst_port 22 action pass
2. prio 65524 protocol ip flower skip_sw src_ip 192.168.1.21 ip_proto tcp action drop
3. prio 65519 protocol ip flower skip_sw src_ip 192.168.1.21 action drop
4. prio 65514 protocol ip flower skip_sw dst_ip 192.168.6.21 action drop
5. prio 65509 protocol ip flower skip_sw action pass
6. prio 65506 flower skip_sw action pass

The first rule (with maximum priority) is the rule that allows the traffic from the TCP traffic on port 22 from 192.168.1.21. The second rule instead blocks all the TCP traffic from the ip 192.168.1.21. Note that this rule has a lower priority, so the traffic on port 22 continues to flow. The third rule blocks the traffic from the ip 192.168.1.21 (this rule inglobate the second one). The fourth rule blocks the traffic to 192.168.6.21. The last rule lets pass all the remaining traffic.

6.5.2 Simple code with unsupported operation

Here we report another program, in this case, we have to add at line 5 a goto operation that our translator does not recognize.

```
 1  int xdp_prog1(struct xdp_md *ctx)
 2  {
 3      if(match_ip_proto(ctx)){
 4          if(match_src_ip(ctx,0x1501A8C0)){
 5              goto test;
 6              if(match_tcp_proto(ctx)){
 7                  if(match_dst_port(ctx,22)){
 8                      return XDP_PASS;
 9                  }
10              }
11              return XDP_DROP;
12          }
13  test:   return XDP_DROP;
14      }
15      return XDP_PASS;
16  }
```

This will produce this set of rules:

1. prio 65524 protocol ip flower skip_sw src_ip 192.168.1.21 action pass
2. prio 65520 protocol ip flower skip_sw action drop
3. prio 65517 flower skip_sw action pass

As we can see the translator stops the process of translation in line 5 and it considers all the encountered condition as a flow with a return XDP_PASS as return action.

This approach is required because it assures us that the semantics of the program does not change. Indeed, the packet after the PASS action will be reprocessed by the XDP software in the host computer that will choose the right action to take.

6.5.3 Simple code with unknown decision statement

In this program, the decision does not depend on our function.

```
int xdp_prog1(struct xdp_md *ctx)
{
        int i=20;
    if(match_ip_proto(ctx)){
      if(i==20){
        if(match_src_ip(ctx,0x1501A8C0)){
            return XDP_DROP;
        }
      }
10      }
11      return XDP_PASS;
12  }
```

The IfStatement on line 20 is unpredictable without performing a more complex analysis, so we consider it as seen in precedence like an unsupported statement.

1. prio 65527 protocol ip flower skip_sw action pass
2. prio 65524 flower skip_sw action pass

The translator converts the condition that understands until the unsupported one and considers the action pass as seen in the precedent case.

6.5.4 Simple code with "unsafe map" access

This is a typical drop and count program, however, this program modifies the state of the system thought the map access so cannot be offloaded.

```
int xdp_prog1(struct xdp_md *ctx)
{
    int ip=0x1501A8C0;
    if(match_ip_proto(ctx)){
      if(match_src_ip(ctx,0x1501A8C0)){
         u64 *value=bpf_map_lookup_elem(&dropped_stat,&ip);
         if(value){
           return;
         }
         *value++;
         return XDP_DROP;
      }
    }
    return XDP_PASS;
}
```

As result we have the two precedent rules:

1. prio 65527 protocol ip flower skip_sw action pass
2. prio 65524 flower skip_sw action pass

6.6 DDoS Mitigator (Code with safe map access)

So far, we have analyzed only code that has decision enforced by a constant rule hardcoded.

In this section, we propose an use case that uses the maps to decide the action to take on a packet.

This is the case of DDoS mitigator, we have seen this use case before in the previous packet but here we took a third part DDoS mitigator and we will modify the dataplane for using our helper. The original application can be found here[4] named *xdp_ddos01_blacklist*.

Following we describe how we have rewritten *xdp_ddos01_blacklist_kern.c* dataplane.

```
int  xdp_prog1(struct xdp_md *ctx)
{
        if(match_map_src_ip(ctx,&blacklist)){
                return XDP_DROP;
        }else if(match_map_dst_port(ctx,&port_blacklist)){
                return XDP_DROP;
        }
        return XDP_PASS;
}
```

The original application filters the packet by analyzing the source ip and the destination port. This will result, in our case, in a very simple dataplane composed by two helper functions defined previously.

At startup, our application emits only the permit-all rule. When an IP is inserted into the DDoS mitigator application, our Python application is triggered and notifies the update to the Java module that emits a rule.

This command:

```
sudo ./xdp_ddos01_blacklist_cmdline --add --ip 192.168.1.40
```

will result in this rule emitted:

```
protocol ip  flower skip_sw src_ip 192.168.1.40 action drop
```

In the same way this command:

```
sudo ./xdp_ddos01_blacklist_cmdline --add --dport 50
```

will result in these rules emitted:

```
protocol ip  flower skip_sw ip_proto tcp dst_port 50 action drop
protocol ip  flower skip_sw ip_proto udp dst_port 50 action drop
```

[4]https://github.com/netoptimizer/prototype-kernel/tree/master/kernel/samples/bpf

6.7 Number of rules

The last case that we have analyzed shows that a single insert in a map can generate multiple rules.

Now consider this program:

```
int  xdp_prog1(struct xdp_md *ctx)
{
 if(match_map_src_ip(ctx,&blacklist_src)){
  if(match_map_dst_ip(ctx,&blacklist_dst)){
    return XDP_DROP;
  }
 }
 return XDP_PASS;
}
```

Consider that we have in the maps the following values:

blacklist_src	**blacklist_dst**
192.168.1.1	192.168.1.4
192.168.1.2	192.168.1.5
192.168.1.3	192.168.1.6

Table 6.2: Maps Content

This will produces nine rules.

```
protocol ip flower skip_sw src_ip 192.168.1.1 dst_ip 192.168.1.4 action drop
protocol ip flower skip_sw src_ip 192.168.1.1 dst_ip 192.168.1.5 action drop
protocol ip flower skip_sw src_ip 192.168.1.1 dst_ip 192.168.1.6 action drop
protocol ip flower skip_sw src_ip 192.168.1.2 dst_ip 192.168.1.4 action drop
protocol ip flower skip_sw src_ip 192.168.1.2 dst_ip 192.168.1.5 action drop
protocol ip flower skip_sw src_ip 192.168.1.2 dst_ip 192.168.1.6 action drop
protocol ip flower skip_sw src_ip 192.168.1.3 dst_ip 192.168.1.4 action drop
protocol ip flower skip_sw src_ip 192.168.1.3 dst_ip 192.168.1.5 action drop
protocol ip flower skip_sw src_ip 192.168.1.3 dst_ip 192.168.1.6 action drop
```

It is clear that this strategy can quickly degenerate and can produce a very large amount of rules.

The number of rules that we can offload is limited and depends on the vendor that manufactured the NIC. In our case we have discover that the maximum number of rules that we can offload is **512**.

One of the solutions that can be used for address this problem is the aggregation. Further analysis of this approach can be performed.

6.8 Software Performance

In this section, we report a series of performance test related to the generation of the rules from the source code.

The average time of execution for a simple program is ~ 1 second.

6.8.1 Line of Code

The last test wants to explore the execution time with different sized programs. We test our code with programs with: 10, 100, 1000, 10000, 100000 Line of Code. The added lines are simple operations that do nothing.

We obtained this result:

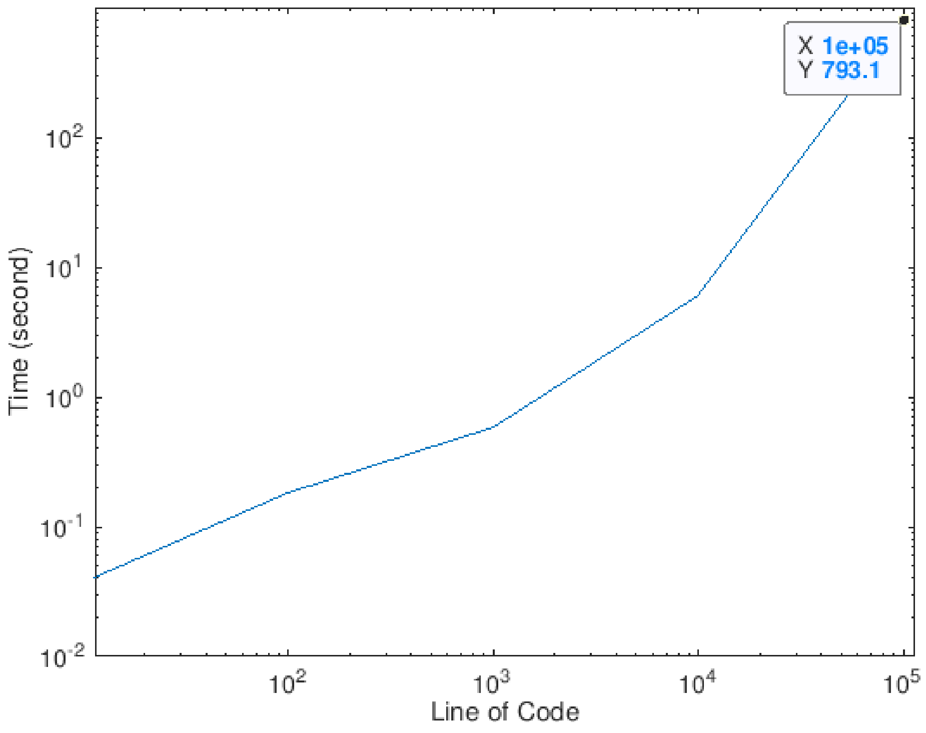


Figure 6.3: LoC: Time of execution

Considering that the XDP code is limited by the verifier in terms of lines of code, this seems to be a good result.

6.8.2 Nested IF testing

The first test performed is focused on analyzing the time of execution for the analysis of a code that contains nested if statements.

We start with this program from one to eight if statements and we compare the time of execution after adding each if statement.

```
int xdp_prog1(struct xdp_md *ctx){
    if(match_src_mac(data,0))
     if(match_dst_mac(data,0))
      if(match_ip_proto(ctx))
       if(match_src_ip(ctx,00000000))
        if(match_dst_ip(ctx,00000000))
         if(match_tcp_proto(ctx))
          if(match_dst_port(ctx,22))
           if(match_dst_port(ctx,50))
            return XDP_DROP;
    return XDP_PASS;
}
```

In the following figure, we report the results.

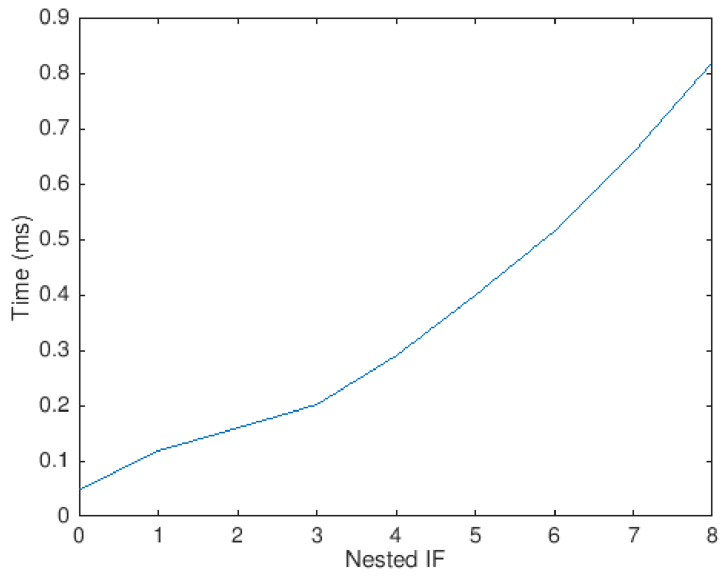


Figure 6.4: Nested If: Time of execution

As we can see it is quite linear and pretty low. In this test and in the next one we report only the exectuion time of our developed code and not of the ANTLR part.

6.8.3 OR Testing

In this test we replace each condition of the following code with an OR condition, we will evaluate the time of execution and the number of rules generated.

```
int xdp_prog1(struct xdp_md *ctx)
{
    if(cond0)
     if(cond1)
      if(cond2)
       if(cond3)
        return XDP_DROP;
    return XDP_PASS;
}
```

We replace each condition with a series of OrExpression. We have made the test from 0 to 48 Expression in or for each statement. Firstly, we report the results in terms of generated rules.

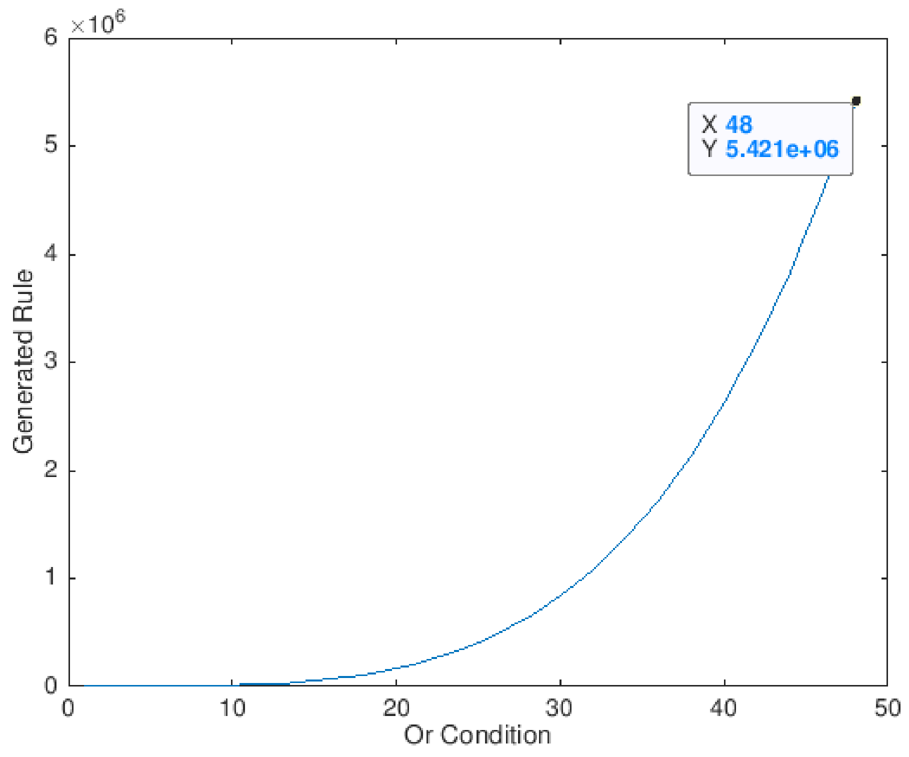


Figure 6.5: Or Statement: Number of Rules

The huge number of rules generated is expected due to the nested if each or condition is combined with each other ones.

This will produce a number of rules that can be described as a partial sum of a geometric series.

$$\sum_{n=0}^{y} x^n = 1$$

Where:

- **x**: the number of OR condition
- **y**: the number of Nested IF Statement (4 in our case)

With the 48 OR condition and 4 If Statement, the generated rules will be:

$$1 + 48 + 48^2 + 48^3 + 48^4 = 5421361$$

That is exactly the result found.

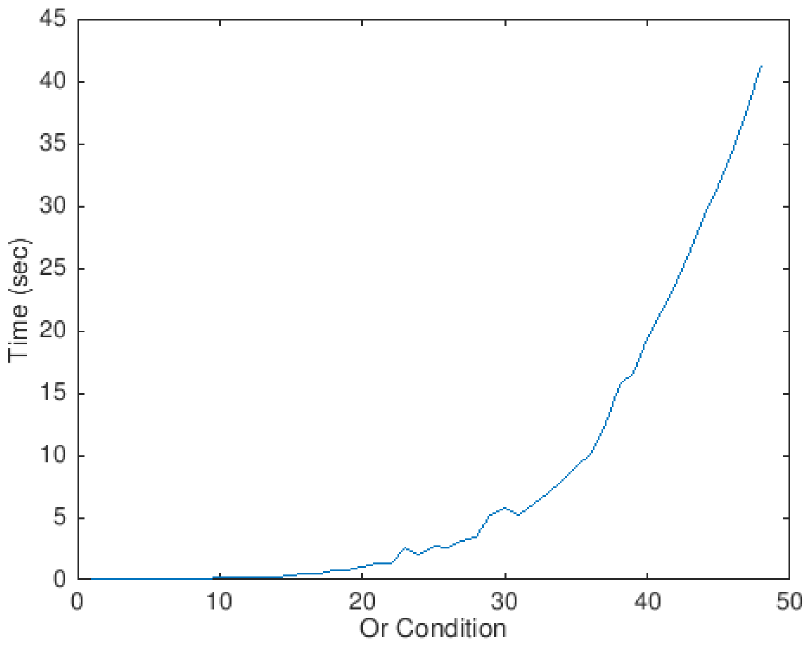


Figure 6.6: Or Statement: Time of execution

The time of execution rises in an exponential way after the 30 OR condition. After 48 OR condition the program crashed due to a memory error.

6.8.4 Map-Triggered rule generation time

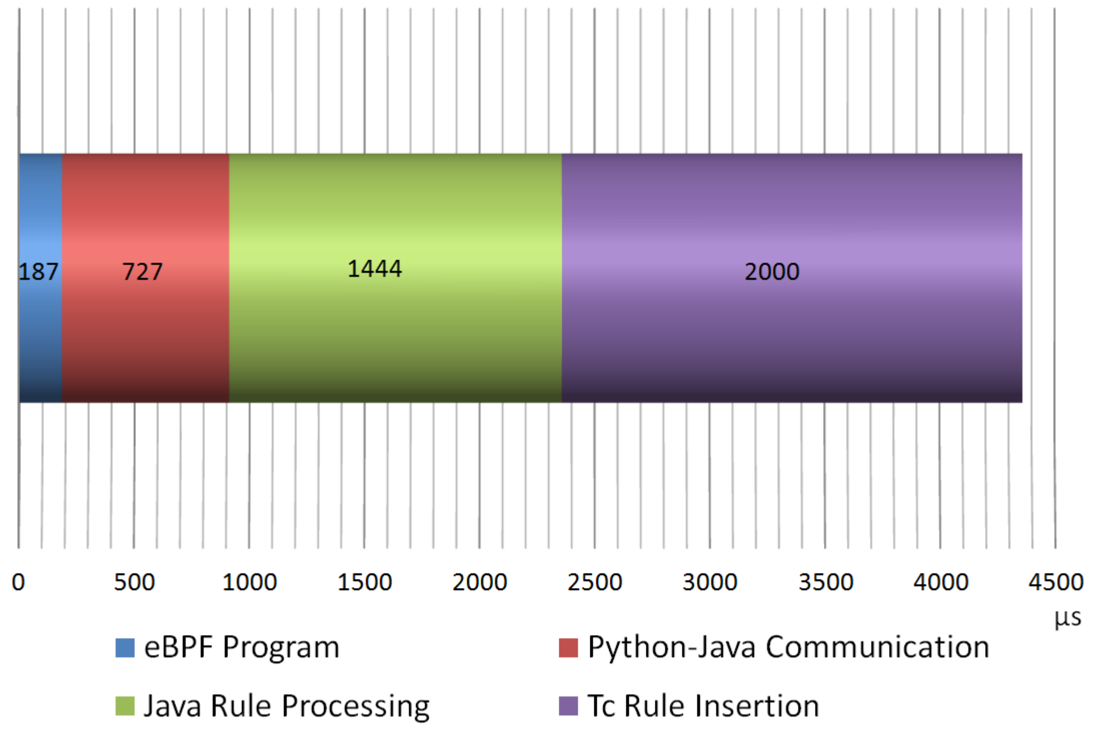


Figure 6.7: Execution time from Map insert to TC rule

The previous figure shows the execution time from the insert of a new element into the eBPF program to the insert of the rule in the Smart-NIC.

- The Blue time is the time spent from the insert of a new element to the detection of the event (with the notification to userspace) by the Python tracing module.
- The Red time is the time spent on the IPC between Python and Java modules.
- The Green time is the time spent from Java module for the generation of the rule.
- The Violet time is the time spent from the TC command for inserting the rule in the HW.

The whole time spent is $\sim$ 4500 µs.

Chapter 7

Conclusion and Future Work

In this work, we have examined some techniques to create high-performance network solutions with the support of hardware offloading.

In particular, we have discussed the technologies of Smart-NIC by considering the offered capabilities and the achievable performance results and their limitations. We have compared and mixed the Smart-NIC technologies with another great innovation introduced inside the Linux kernel that can help to realize high-performance network services: eBPF.

We have tried, firstly to offload eBPF code directly to the NIC hardware. Then, we have tried to extract from the eBPF code a set of stateless rules that can be offloaded into the layer 2 hardware switch of NIC.

With this last solution, we have reached the line rate speed in the packets processing. However, there were significant limitations introduced by this approach. In particular, the analysis of the code has been particularly complex

As future work, different approaches can be followed:

- The analysis of the code can be expanded by avoiding the introduced constraint of helper function.
- It can be possible to support other constructs of the programming language that we have ignored.
- It can be possible to perform a direct analysis of eBPF bytecode, in order to simulate the execution of the code and to extract in this way the rules without analyzing the complex structure of the C code.

Lastly, it can also be interesting to compare our developed solution with other ones that involve NICs produced by other vendors and in particular with the ones that support the direct offloading of eBPF code.

Bibliography

[1] M. Labraoui, M. M. Boc, and A. Fladenmuller. «Opportunistic SDN-controlled wireless mesh network for mobile traffic offloading». In: *2017 International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT)*. May 2017, pp. 1–7. DOI: 10.1109/MoWNet.2017.8045960.

[2] *Network Functions Virtualization-Introductory White Paper*. http://portal.etsi.org/NFV/NFV-White-Paper.pdf. 2012.

[3] L. Nobach and D. Hausheer. «Open, elastic provisioning of hardware acceleration in NFV environments». In: *2015 International Conference and Workshops on Networked Systems (NetSys)*. Mar. 2015, pp. 1–5. DOI: 10.1109/NetSys.2015.7089057.

[4] Steven McCanne and Van Jacobson. «The BSD Packet Filter: A New Architecture for User-level Packet Capture». In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, 1993, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=1267303.1267305.

[5] Matt Fleming. *A thorough introduction to eBPF*. https://lwn.net/Articles/740157/. 2018.

[6] Brendan Gregg. *Linux Extended BPF (eBPF) Tracing Tools*. http://www.brendangregg.com/ebpf.html. 2018.

[7] *XDP - eXpress Data Path*. https://www.iovisor.org/technology/xdp. 2018.

[8] Georgios P Katsikas et al. «Metron: NFV Service Chains at the True Speed of the Underlying Hardware». In:

[9] Wikipedia contributors. *Large send offload*. https://en.wikipedia.org/wiki/Large_send_offload. 2018.

[10] Broadcom. *Cloud Scale Networking*. https://www.broadcom.com/applications/datacenter-networking/cloud-scale-networking. 2018.

[11] Martin A. Brown. *Traffic Control How-TO*. http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html. 2006.

[12] Simon Horman. «TC Flower Offload». In: *NetDev Conference*. 2017.

[13] Michael Kerrisk. *flower - flow based traffic control filter*. http://man7.org/linux/man-pages/man8/tc-flower.8.html. 2015.

[14] William Cohen Jason Baron. *The Linux Kernel Tracepoint API*. https://www.kernel.org/doc/html/v4.18/core-api/tracepoint.html. 2018.

[15] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006.

[16] Wikipedia contributors. *Abstract syntax tree*. https://en.wikipedia.org/wiki/Abstract_syntax_tree. 2018.

[17] Wikipedia contributors. *Control flow graph*. https://en.wikipedia.org/wiki/Control_flow_graph. 2018.

[18] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013.

[19] Joern Project. *Joern*. http://mlsec.org/joern/. 2018.

[20] Sudhanshu Goswami. *An introduction to KProbes*. https://lwn.net/Articles/132196/. 2005.

Appendix A

PerfTracer

We have developed a little suite of scripts with the purpose of automating the tests[1].

This solution aims to collect statistics in terms of CPU and RAM usage, with the purpose of detect workload anomalies.

Our solution uses the following set of tools, available in the most common Linux distributions:

- bash (scripting support)
- mpstat (CPU statistics)
- free (ram statistics)
- perl (data parsing and processing)
- gnuplot (data plotting)
- iperf3 (network performance measurement)

The **perfmon.sh** script provides a simple way to execute any type of test, collecting in background the statistics, plotting data and compressing all the results into a tgz archive. You can launch it with:

```
$ ./myscript/perfmon.sh command
```

After the termination of command (it is possible to terminate it with ctrl-c), the script prompts to the user to insert a name for the archive. It is possible to also pass the archive name as an external variable

```
$ FILENAME=myarchive.tgz ./myscript/perfmon.sh command
```

[1]https://github.com/raffysommy/PerfTracer