POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

Towards a faster Iptables in eBPF

Supervisor
Prof. Fulvio Risso

Candidate
Massimo Tumolo

PoliMi Supervisor
Prof. Antonio Capone

Academic year 2017-2018

To my family.
You did more then you could for me to get here, I hope I will be as great to my children as you were to me.

Alla mia famiglia.
Avete fatto oltre il possibile per farmi arrivare qui, e spero di poter essere per i miei figli quanto voi siete per me.

Acknowledgements

I would like to acknowledge my supervisor, or my mentor, Prof. Fulvio Risso. He trusted me to work on this project, he believed in me and helped me through all the steps. This thesis has been a wonderful opportunity to grow personally and professionally, and everything I learned was thanks to the opportunities he opened for me.

A huge acknowledgment goes to Matteo Bertrone. He developed this thesis with me, we had (*very*) long chats on problems, possible solutions, and consequent problems. We had some coffee just to talk. He was a really nice friend and collaborator. Thanks.

Thank you to Sebastiano Miano, for being a good friend and a source of precious tips. I owe him a bunch of coffees that he offered during our chats. And thank you for suggesting me my favorite pizza place in Turin.

Thank you to Mauricio Vásquez Bernal, for teaching me a lot on eBPF, and for being always available to provide me with help and informations.

Last but not least, thank you to Simona. Every time I chose the work over you, every time I spent our time to do this or that activity, you were always there to support me, you were always there to bring me back from my mind deadlocks. I owe you so much.

Abstract

`Iptables` is the *de-facto standard* Linux firewall. Features are its strength, but its low scalability and poor performance can represent the bottleneck of network systems. This thesis describes the challenges encountered and the choices made while developing a prototype to replace Iptables back-end, the `Netfilter`, improving performance but keeping the same syntax and semantics. The implementation is based on the separation of the *fast and thin* data plane in charge of the traffic processing and the *slow and fat* control plane, in charge of managing the overall logic. While the latter leverages on the possibilities offered by `C++`, the former is implemented in eBPF, a technology recently added to the Linux kernel, that allows fast in-kernel packet processing. The results show a great gain over Iptables, and the architecture modularity opens opportunities for further algorithmic improvement and features development.

Contents

Acknowledgments II

List of Figures VII

1 Introduction 1

1.1 Security through firewalls 3

1.2 The Linux firewall: Iptables & Netfilter 4

1.3 Motivation 6

2 Tools 9

2.1 Technologies 9

2.1.1 BPF 9

2.1.2 eBPF 9

2.2 Frameworks 13

2.2.1 BCC 13

2.2.2 Polycube 15

3 Related work 17

3.1 Ipset 17

3.2 Nftables 18

3.3 Bpfilter 19
3.4 The packet classification problem 20
3.4.1 Algorithms for packet classification 21

4 Design 25

4.1 Semantic 25
4.2 Matching algorithm 27
4.3 Architecture 29
4.3.1 Pipeline architecture 30
4.3.2 Chain 31
4.3.3 Connection tracking 33
4.3.4 Counters 38
4.4 Dataplane optimizations 40
4.4.1 Tailored dataplane 40
4.4.2 Leveraging on connection tracking 41
4.4.3 Atomic update 42

5 Implementation 45

5.1 Syntax 45
5.2 eBPF Maps 46
5.3 Code structure 48

6 Benchmarking 55

6.1 Methodology 55
6.1.1 Rule set 56
6.2 Results 57

6.2.1 Nic-To-Nic setup 58
6.2.2 Host firewall setup 61
6.2.3 Rule insertion 63

7 Conclusions 67

7.1 Future work 67
7.2 Final remarks 68

A Firewall data model 71

References 75

List of Figures

1.1 Firewall 4
1.2 Netfilter architecture 6
2.1 eBPF Tracing 11
2.2 eBPF TC vs XDP benchmarks 12
2.3 eBPF overview 14
3.1 Bpfilter performance 20
4.1 Linear Bit Vector Search example 29
4.2 Overview of the main components. 30
4.3 Data plane architecture TODO CITARE 31
4.4 Matching chain 33
4.5 TCP Connection tracking 36
4.6 UDP Connection tracking 36
4.7 Tracking counters 38
4.8 Tailored data plane 41
4.9 Atomic update 43
5.1 User interface architecture 46

5.2 Flags management 48
5.3 Code structure 49

6.1 NIC-to-NIC Benchmark setup 56
6.2 Throughput UDP (64B frames) 59
6.3 Throughput UDP (512B frames) 59
6.4 Throughput UDP (512B frames, no Linux routing) 60
6.5 Latency ICMP 60
6.6 Host firewall setup 61
6.7 Input UDP Throughput 62
6.8 Input TCP Throughput 64
6.9 Input TCP Throughput 64
6.10 Latency ICMP 65
6.11 Rule appending time 66
6.12 Rule insertion time 66

7.1 Tree and LBVS 67

Chapter 1

Introduction

Every system connected to a network, from a smartphone to enterprise data centers, have to employ some protection mechanism when connected to a network to avoid potential attacks.

A widely used system is the firewall, usually representing the first layer of defense. A firewall is a *traffic filter* that allows only a specific and customizable class of traffic to be exchanged with or by the protected device. Firewalls are widely used due to their effectiveness and low overhead, as they allow with a relatively reasonable cost to reduce the attack surface of a network by discarding most of the malicious traffic.

Linux, the most widely employed operating system on data centers, accomplishes security through the Netfilter subsystem and its widely used frontend, `Iptables`. This software suite is a firewall that allows the inspection of the traffic to modify, redirect, drop or allow each packet.

The system consists of a kernel part, the Netfilter, organized as a collection of tables which exploits the hooks provided by the framework to apply the security policies. The second part represents a user-level utility that allows to configure and create these chains/rules appropriately. For brevity, this thesis will refer to the entire system as Iptables.

Iptables has been used for over twenty years and many Linux-based systems still use it to enforce security. Nevertheless, with the advance of the technologies

and the increase of the networks speed, its limits are becoming evident. Stricter requirements expect the firewall to keep up with the performance needs of the data centers, without the possibility to sacrifice the configuration complexity.

Such a scenario is not favorable to Iptables, that is not scalable enough to satisfy the demands. To answer to this problem, in 2014 a new firewall was included in the kernel, `nftables`, with the aim of replacing the existing Iptables framework. Although this yields advantages over its predecessor, `Nftables` has not yet had the desired success: system administrators are reluctant to migrate to new solutions due to the difficulties and the risks related to reconfigure their systems with different tools, without any notable gain.

Recently, the extended BPF technology has been introduced into the Linux kernel, offering the possibility to process outgoing or incoming traffic by executing code directly in the kernel. This premise, together with the potential showed by the bpfilter, a proof of concept to replace Iptables proposed in the kernel [1], makes eBPF a perfect candidate to fully reimplement Iptables, keeping both its semantics and syntax.

Such an approach would allow customizing the in-kernel internals of Iptables introducing improvements both by the algorithmic and the filtering point of view, leveraging on the possibility offered by eBPF to inject code at runtime without recompiling the kernel. Moreover, programs could be offloaded to hardware, obtaining an even greater performance benefit.

This thesis presents an eBPF-based prototype cloning Iptables. The goal of the prototype is to show that it is possible to achieve high performance without disrupting the users' security configurations and without requiring the user to recompile custom versions of the kernel with external modules.

The description will scroll through the main background topics, from firewalls in general to the internals of Iptables. It will then present the other attempts to replace Iptables, the lessons learned from them and why they did not succeed.

A detailed presentation is then provided on the developed prototype with emphasis on the design choices that have been made to succeed in the goal, first from

the architectural perspective and then from the implementation one, highlighting the differences between eBPF and Iptables, and the adopted solutions in order to correctly emulate the filtering semantics while improving the classification algorithm.

By the end, the eBPF prototype will be evaluated by comparing it with the current implementation of Iptables, showing how the replacement would lead to higher performance particularly when a high number of rules is involved.

1.1 Security through firewalls

In the past, terraced houses were built using wood. This meant that if one house got on fire, all other houses would have been affected as a chain reaction. The answer was to build stone walls between them to prevent fire propagation: the firewalls.

The firewall network function takes the name from those brick walls, as its aim is to enforce security and prevent attacks to go from one network to another, acting exactly as a wall (figure 1.1). To do so, a firewall is put between two networks and configured to select which traffic should pass through them and which should not. Ideally, this should prevent malicious traffic to pass, practically is not always easy to foresee from where the malicious traffic will come from, and firewalls are only a first protection against attacks.

There are two ways to classify firewalls, one based on the traffic they observe, the other based on the granularity on which they can inspect the traffic [2].

The first classification distinguishes *Ingress* firewalls from *Egress* ones. The formers filter incoming connections and usually allows them based on the services offered to the external networks. The latter filter outgoing traffic, and are usually used to prevent certain applications to be used in a network (e.g. to prevent employees to go on social networks).

The second classification is based on how far in the packet a firewall can see:

1. Packet filter: inspects packets at the network level, with basic support for transport headers. Simple and fast, but it may be easily tricked using packet spoofing.

2. Stateful packet filter: same support of a packet filter, but state-aware. Can track connections and take decisions based on them.

3. Application-level gateway: explores the traffic up to the application layer. Usually, it's a packet filter with modules that can be attached, each module specific to an application protocol. It's the most secure choice but introduces high overhead on the traffic processing.

4. Local firewall: A firewall that is directly installed on the node to be protected, usually filters the traffic that reaches or comes from applications running on it.

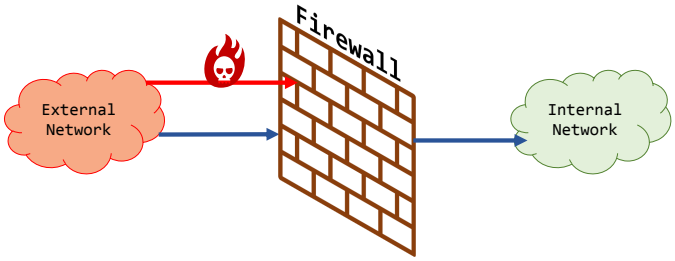

Figure 1.1: Visual representation of the firewall job.
Brick wall by Creative Stall from the Noun Project. Flaming skull by Andrew Cameron from the Noun Project.

1.2 The Linux firewall: Iptables & Netfilter

The Iptables/Netfilter firewall is the most used Linux firewall. It is a local ingress and egress firewall, as it is installed locally on the host, and it can work as a stateful packet filter or an application-level gateway, based on the configuration.

The system is split into two components: Iptables, and Netfilter.

Iptables is a frontend application, it offers an API to the clients (other applications or human users) to configure the underlying Netfilter. This configuration is expressed as a sequence of rules that associate some properties of the traffic with the action to be taken. For example,

```
iptables -A INPUT -p tcp --dport 22 -s 192.168.0.0/24 -j DROP
```

prevents hosts outside the network with addresses ranging from `192.168.0.0` to `192.168.0.255` to open *SSH* connections with the host that is behind the firewall.

Moreover, Iptables can configure the Netfilter to perform Network Address Translation and packet mangling. The syntax is very similar to the filter one. For example,

```
iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to 10.0.0.1
```

changes all the outgoing traffic's source address in `10.0.0.1`.

The Netfilter is the backend component. It runs only in the kernel, and it is in charge of processing the traffic. It is not only a firewall, but instead a complex cooperation between components like a NAT (as hinted in the previous paragraph), a Router, a Bridge, and a Firewall. A packet that reaches a Linux host is processed by each one of this components before it can proceed to the destination.

The Netfilter exposes *hooks*, interfaces on which functions can be attached to process the traffic in the kernel. Iptables attaches the firewall functionalities in the Netfilter in three hooks: *input*, *output*, and *forward*. The first one is in charge of processing the traffic directed to applications running on the host; the second one is in charge of the traffic coming from these applications; the third one is reserved for packets traversing the host as if it was a router or a bridge, i.e. just traversing from one interface to another. The forwarding chain is used for traffic between namespaces (and so, containers), too.

The Netfilter behavior and functionalities are strongly based on the cooperation of these components, chained together to process the traffic. The traffic is first received by the NAT, that possibly changes the addresses, then it is processed by the routing and switching system and, at this point, there is enough knowledge to filter the packet in the proper hook.

The Netfilter offers the possibility to configure a different set of policies for packets captured on the input, output, or forward hooks. These policies are chained as a linked list and each time a packet is received they are scrolled linearly until the first one matching the packet is found.

Figure 1.2 depicts a simplified version of the Netfilter structure, with emphasis on the main components and particularly on the filtering functionalities.

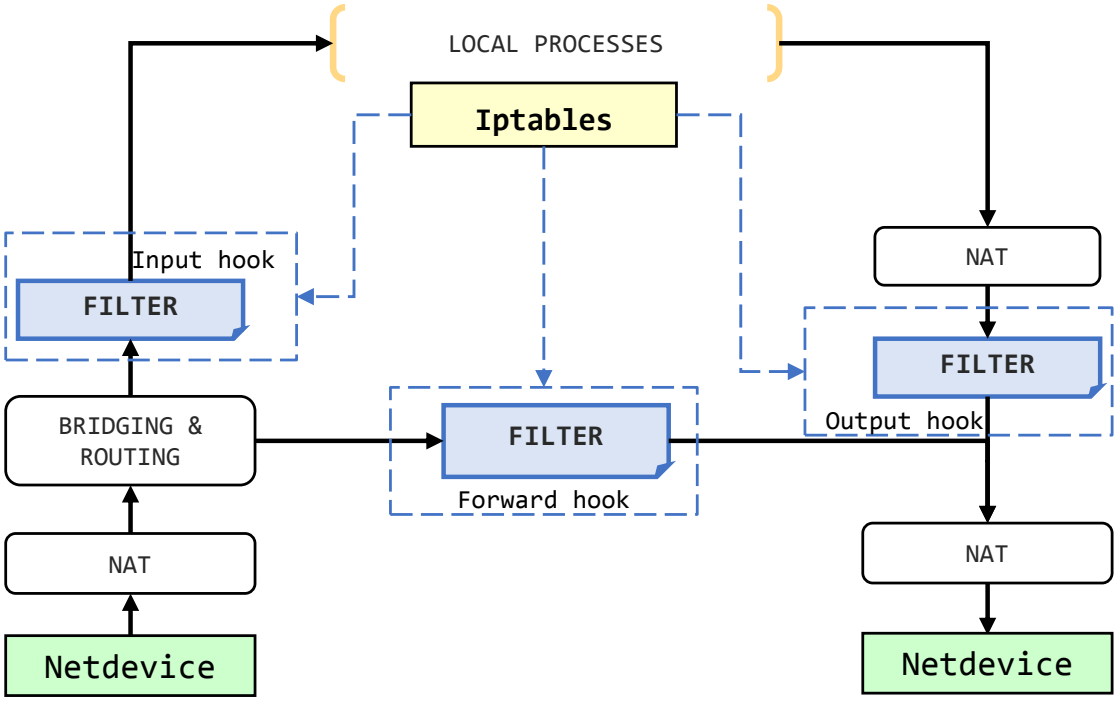

Figure 1.2: Overview of Netfilter's main components.

1.3 Motivation

The NetGroup[1] at Politecnico di Torino is exploring the eBPF technologies to develop high-performance network functions on Linux systems, in collaboration with Huawei and in the framework of the European ASTRID project.

Every network function, however, needs a layer of protection. During the research, Iptables was found as a bottleneck in service chains, where most of the overhead was coming from letting Linux enforce security. As an example, the Kubernetes orchestrator is strongly limited as it requires to inject rules in Iptables.

[1] `http://netgroup.polito.it/`

A first attempt to develop a lightweight security tool was a `Policy-based forwarder`, developed by the author of this thesis. Such a tool was just a way to stress the limits of the technologies to explore if it was suited for a more complex use case, a firewall.

The positive outcome lead to the development of this thesis, with the more ambitious aim of replacing Iptables and possibly the entire Netfilter architecture. The project gained the interests of the kernel developers thanks to the Netdev [3] and SIGCCOM [4] conferences, and it may represent a connection with the Linux community to start a collaboration.

Chapter 2

Tools

2.1 Technologies

2.1.1 BPF

The Berkeley Packet Filter (BPF) is an in-kernel virtual machine for packet filtering, initially introduced in FreeBSD and in Linux 2.1.75. It is shipped with its own instruction set to be interpreted by the virtual machine.

BPF was initially used in packet capture tools like `tcpdump` or `wireshark`. Its purpose is to perform fast in-kernel packet filtering without the overhead of the context switch needed to process packets in the user space. Moreover, being run on a VM, BPF is platform independent and can be executed on any architecture, from the home router to an enterprise Linux server.

2.1.2 eBPF

BPF was deeply revisited by Alexei Starovoitov and proposed in 2013[1] under the name of eBPF. It was officially introduced in Linux since kernel 3.15. From this moment, BPF was referred to as cBPF (*classic* BPF) and eBPF as BPF.

[1] `https://lkml.org/lkml/2013/12/2/1066`

eBPF introduced deep architectural improvements:

- Reacting to generic events in the kernel. eBPF is not only a packet filtering mechanism but instead, it can be used to react to a range of system calls (shown in figure 2.1). This mechanism is really powerful and pushed a lot the development of the technology, as shown by one of the main developers of eBPF tracing tools, Brendan Gregg [5].
- Networking hook points. Further explained later, eBPF introduced the possibility to trace and mangle packets at the TC and XDP hooks.
- Stateful processing. eBPF programs are event-driven, each execution is independent of the other. However, eBPF introduced *maps*, data structures to preserve the state and share it between executions and the userspace.
- Tail calls. eBPF introduced a mechanism to connect programs. Tail calls are *long jumps*, i.e. function calls that do not return to the caller.
- Helpers. From eBPF code it is not possible to invoke kernel routines through system calls. This is a strong limit in the interaction with the kernel, however, some functionalities are exposed through helper functions. These functions are considered safe can be used by eBPF programs to perform operations like accessing the maps, retrieving the time, modifying the packets, execute tail calls and so on [6].
- 64 bit architecture and Just-In-Time compilation for increased performance and flexibility.

eBPF allows an user-space application to inject code in the kernel at runtime without the need to recompile it or add some module. However, programs have to be already in the kernel tree [7]. This limitation has been mitigated by BCC, presented in section 2.2.

Natively, eBPF offers its own assembler. Programs can be written using it and converted to bytecode using the Linux command `bpf_asm`. However, a much more

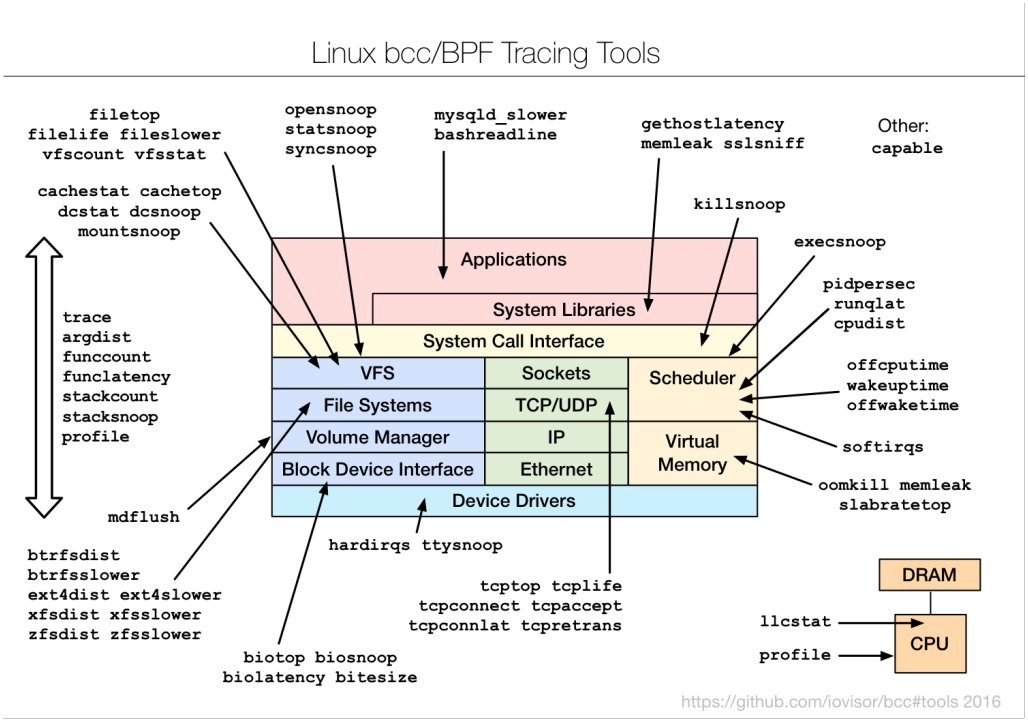

Figure 2.1: System calls that eBPF can intercept. From [5].

user-friendly approach to the technology is writing programs in restricted C, and let the Clang/LLVM compiler suite handle the translation to the bytecode.

Independently from how the code has been written, the assembly code has to be injected into the kernel. The `bpf()` system call allows to do so, but the code has to go through a number of strict checks before it can start running. The eBPF verifier checks every memory access and pointers to guarantee that no memory leak can happen, to do so it simulates the execution and ensures that after it the registers and the memory are in the right state. Moreover, it walks through the code flow graph to ensure the termination.

After a successful load, the execution of an eBPF program is event-driven. A program is hooked to a particular type of event, and every manifestation will cause the program to be executed. Based on the event category, it may be possible to alter the event itself.

When an eBPF program is attached to a networking event, i.e. the arrival of a packet, it can intercept packets and modify, forward or drop them. The traffic can

be captured in three points of its lifecycle in the kernel, namely in three hooks:

1. **Socket filter**. It intercepts packets after the IP routing. However, no modifications can be made to the traffic, that can only be observed.

2. **Traffic Control (TC)**. The Traffic Control hook intercepts incoming and outgoing traffic (namely *ingress* and *egress*) right before it is passed to the Netfilter.

3. **eXpress Data Path (XDP)**. The XDP hook is located right after the networking card driver. It can intercept traffic in the *Driver (or Native)* mode and in the **Generic (or skb)** mode. The first mode has to be supported by the particular driver, it's the closest point to the card and it has high-performance benefits as the kernel does not create any data structure, so there is practically no overhead. The second mode is a way to use XDP on drivers that do not support it. In both modes, XDP hooks can only operate on RX traffic (i.e. the traffic received by the card from outside the host). A comparison of the performance of these hooks against a program run in the control plane is depicted in 2.2, from [8].

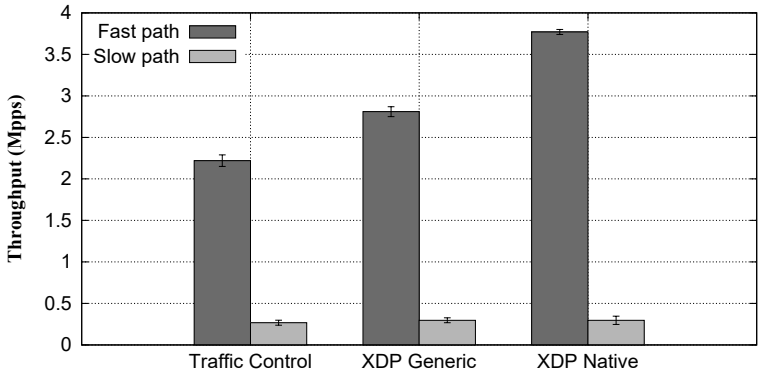

Figure 2.2: Performance of a bridge service running in TC or XDP (in the figure, fast path). From [8].

The overall eBPF architecture is depicted in figure 2.3, from the code injection to the run-time processing.

While eBPF presents the advantage of allowing fast and early in-kernel packet processing, using it to create complex network services presents a number of challenges to the programmer due to the constraints forced on the coding paradigms. Following, these limits and possible solutions will be explained based on the article written by the authors behind the eBPF firewall [8].

- **Limited program size.** An eBPF program can't be longer than 4096 assembly instructions. The reason behind this choice is that programs are executed in the kernel, and are meant to perform fast processing and terminate in a bounded amount of time. When complex functions are needed, this program size can be a severe limit. However, this limitation can be circumvented by designing the programs in a modular way, splitting them into multiple programs connected through *tail calls*.

- **Limited number of tail calls.** Strictly connected to the limited program size, this limitation prevents the programmer to chain more than 32 programs.

- **Unbounded loops.** Since eBPF programs are run in the kernel, it is necessary that they are checked to avoid infinite loops, that may result in the kernel being blocked and possibly crashing. The solution adopted is to refuse any backward jump, in this way it is ensured that the program will terminate. As a consequence, unbounded loops can't be implemented. In the next section, a partial solution will be presented by using the LLVM directive `pragma unroll` that unrolls the (bounded) loop in a sequence of instructions. This allows the programmer to write loops, but at the expense of a greater program size.

2.2 Frameworks

2.2.1 BCC

The developed eBPF firewall leverages on the eBPF Compiler Collection (BCC).

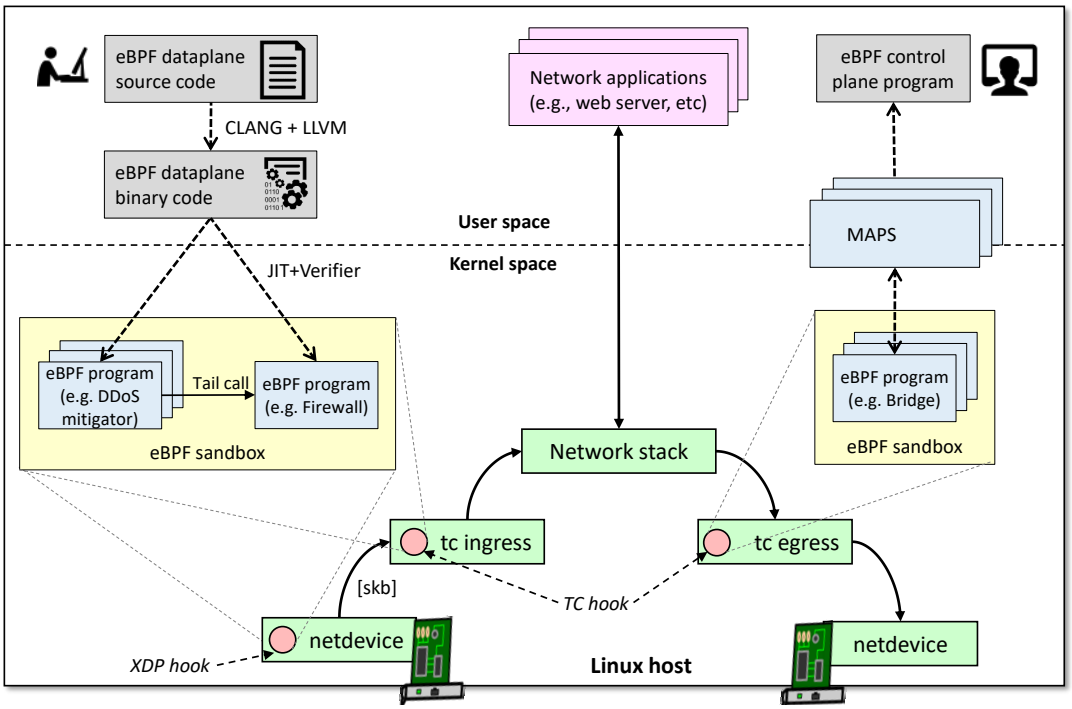

Figure 2.3: eBPF overview, from [8].

BCC is a toolkit for creating efficient kernel tracing and manipulation programs [9]. In practice, it is a framework acting as a frontend for eBPF that offers abstractions to the developer to ease the development of eBPF programs.

The toolkit provides a mechanism to write eBPF programs using a restricted version of C. Through Python scripts it is possible to let BCC handle the cycle of the restricted C code from the user space to the kernel. Internally, BCC leverages on Clang to translate to an intermediate representation that is then provided to LLMV for compilation, and the output is eBPF assembler that can be directly injected in the kernel. All this process is transparent to the user.

Moreover, BCC encapsulates eBPF functionalities and LLVM features offering easy-to-use macros and directive to the user. For example, to create a hash map from an eBPF program it is enough to write the following line:

```
BPF_HASH("MapName",struct key*, struct value*);
```

An example of directive is `pragma unroll`, that can be used before bounded

loops (i.e. loops in which the number of iterations is known at compile time) to tell LLVM to unroll that loop in a sequence of instructions, hence overcoming the limit of not being able to use backward jumps.

2.2.2 Polycube

The Netgroup group from Politecnico di Torino is building a framework for the development of network services based on BCC, `Polycube`, not yet public. The aim of the projects is to allow the development of fast and dynamically loadable network functions running in the Linux kernel through the eBPF technology.

The framework allows the user to split the logic of the application between a fast simple data plane and a complex slow control plane. The first one handles the traffic, the second one handles the management and the configuration. The division is reflected in the technology used by the two components: the data plane requires to be written in restricted C, to be later translated in eBPF, the control plane is entirely in C++.

All services running on this framework expose an API to the framework itself in order to handle the client commands. The framework receives commands through a REST API or a command line.

To build a service using the framework, the first step is building a data model that defines the structure, syntax, and semantics of the service data. From it, the framework will automatically handle the code scaffolding through Swagger[2]. The model of the developed firewall, written in YANG, is attached in the Appendix A.

Moreover, a number of abstractions are offered to the developer to load, unload, reload (that is, unloading and loading the program without losing the state) the eBPF programs. These functionalities are very handy for handling code optimization, as the code can be written by the control plane and dynamically injected in the kernel.

[2]https://github.com/swagger-api/swagger-codegen

Chapter 3

Related work

The Iptables/Netfilter firewall has been around for many years, and in the Linux community some attempts to improve the system or replace it have been made. Some projects were just front-end to ease the configuration, like `ufw` [10], without any internal change. Others were trying to deeply deal with the architectural limits, either by adding functionalities, like `Ipset`, or by proposing a complete alternative to the Netfilter, like `Nftables` or `Bpfilter`.

This section will shortly provide a description of these projects to highlight their limits, as they were thoroughly considered during the development of the eBPF firewall.

3.1 Ipset

After the development of Iptables, a common use case was to insert long lists of rules just requiring an exact match on IP addresses to reject the traffic from malicious hosts. This lead to performance degradation linearly proportional to the number of rules required. Ipset was included as a Netfilter module to address this problem [11].

As the name suggests, Ipset provides the user the possibility to gather a set of IP addresses and then use Iptables to filter them all. Internally, instead of storing the addresses as a sequence of rules, it stores them in a hash table [12], moving the

complexity from linear to constant. For example, supposing the following rules are needed in Iptables:

```
iptables -A INPUT -s 192.168.0.1 -j DROP
iptables -A INPUT -s 192.168.0.2 -j DROP
iptables -A INPUT -s 192.168.0.3 -j DROP
iptables -A INPUT -s 192.168.0.4 -j DROP
```

They can be translated in a single rule using Ipset:

```
ipset -N toreject iphash
ipset -A toreject 192.168.0.1
ipset -A toreject 192.168.0.2
ipset -A toreject 192.168.0.3
ipset -A toreject 192.168.0.4
iptables -A INPUT -m set --set toreject src -j DROP
```

However, this project helped but did not solve the problem at his root, as it improved only a particular situation.

3.2 Nftables

Nftables started with the aim of replacing Iptables by introducing a new subsystem in the kernel. This system uses an in-kernel virtual machine, like eBPF, from which the userspace utility can inject raw instructions that will be compiled and executed by the VM. The goal was to have a flexible implementation that did not require to change the kernel to support new matching, but instead allowed to update only the user space program.

An important innovation introduced by Nftables is the structures of the rules. There are no default chains in which rules are linearly scanned. The user is free to organize the rules, and a smart configuration results in Nftables using optimized data structures for increasing performance. For example, the same thing done by Ipset can be achieved by Nftables for any field if the rules express a group of values to be matched.

From the benchmarks [13], two conclusions can be drawn. First, Nftables can widely outperform Iptables if the entire rule set is designed for Nftables, as the architecture allows the firewall to scale for large rule sets, so if there are no special requirements, Nftables is the way to go. Second, if Nftables has to be just a back-end for Iptables, using conversion tools to adapt the syntax, there is no point in doing that, because there is no performance gain. This situation is common both for who already has a working firewall and wants to migrate it and for who uses software that works only on Iptables (for example, Kubernetes). Benchmarks show that configuring Nftables with the same semantic of Iptables does not provide any benefits, instead, it only leads to performance degradation.

For this reason, Nftables did not succeed in replacing Iptables but still, it is a valid alternative.

3.3 Bpfilter

Recently [1], a new attempt to replace Iptables/Netfilter was proposed to the kernel developers: bpfilter. This project is entirely based on filtering traffic through eBPF.

The first Proof Of Concept was really promising [11] from a performance point of view. Through an userspace utility, bpfilter injects in the kernel eBPF assembler code to create a list of addresses to filter the traffic from. Results are impressive (figure 3.1), especially by leveraging on the XDP hook.

However, the prototype currently introduced in the kernel is far from being an Iptables replacement. The focus was showing the technology potential, not reproducing the semantics of Iptables, and simple address filtering is a use case way too simple to be actually used in an enterprise-firewall as a standalone solution.

The latest developments in the kernel are discussions and patches proposal to achieve compatibility with the existing systems from a syntax point of view, i.e. the developers are trying to find the best solution to steer the rules from the user-space to the kernel space without incourring in the same problems of Iptables and keeping enough flexibility to support Nftables syntax, too. In this direction, two patches

were proposed to show a possible architecture of the framework that will possibly be in charge of this task [14] [15].

The eBPF firewall prototype developed during this thesis is working on the semantics problem by trying to address the difficulties related to reproducing Iptables behavior. Hence, the work is compatible with the one done in the kernel, and the two projects may eventually converge in a full-fledged replacement of Iptables.

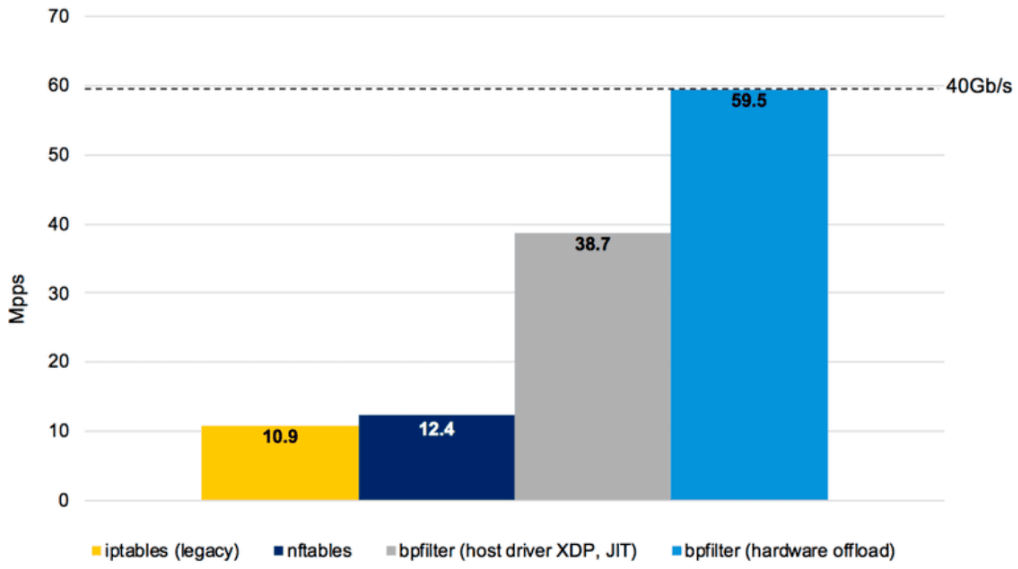

Figure 3.1: Bpfilter benchmarks, from [11].

3.4 The packet classification problem

Packet classification is the process in which given a packet and a set of rules, the packet is associated with exactly one rule. To do so, some of the packet fields are used to look for the matching rules, and if more than one exists, only the best (i.e. the *least-cost*) is chosen.

Supposing that for the classification process the headers $h0, h_2, ...h_N$ are required (for example, source and destination IP addresses and transport protocol), and that the rule set is composed by a set of rules $R_1, R_2, ...R_K$, each rule specifies a combination of N values, one for each field, and has an associated cost. A packet will match

the rule R_x if each field matches the field of R_x. The result of the classification will be the rule with the lowest cost. A more general definition allows rules to do not specify a field, that is therefore treated as a *don not care*, or *wildcard*.

The classification problem is found in many applications (e.g. for routing, or for QoS policies), and the firewall uses it to take actions on the traffic based on the configured policies.

3.4.1 Algorithms for packet classification

There are many algorithms available in the literature to perform packet classification, but there is not one considered the best, yet. Instead, each one is suited to solve a particular declination of a problem. For example, some algorithms support only the longest prefix matching (useful for IP addresses), others do not support wildcards, and so on.

The choice of the most suited algorithm for the firewall was based on three parameters: performance, flexibility, and feasibility. To evaluate performance, the algorithm complexity is taken into account, for the flexibility it is useful to understand which kind of fields are supported and if the algorithm allows for wildcard matching, feasibility means the possibility to actually implement the algorithm given the limits of the eBPF technology.

Following, a selection of algorithms that were evaluated as candidates for the eBPF firewall will be presented and commented, mostly based on the extensive study performed in Varghese's book [16]. All these algorithms are capable of matching a general rule set, i.e. not just based on one or two fields but on an arbitrary number. This flexibility is mandatory to clone the Iptables semantics.

- **Linear search:** The straightforward solution, this algorithm consists of scrolling through the rule set until the first one that matches the packet is found. It is a flexible algorithm, allows for matching on an arbitrary number of fields of any type and supports wildcard. The main drawback is performance, due to the linear complexity, that makes it not feasible for large datasets. This is the algorithm used by Netfilter [17].

- **Geometric View of Classification:** This is more a technique that an algorithm. Each value specified by a rule is a number. If a rule specifies K fields, it can be seen as a point in a K-dimensional space. However, rules may specify ranges or wildcards and, in both cases, this would translate in a set of values for the same coordinate of a given rule. The resulting geometry would be a polygon (for example, a rectangle in two dimensions) that spans over the area that the rule matches. To classify a packet any geometric algorithm that can detect if a point belongs to a shape can be used. These algorithms have two drawbacks: they complex implementations that require unbounded loops (for example, a binary search in two dimensions) that can not be used in eBPF, and require either exponential memory or exponential time.

- **Cross-Producting:** This algorithm is strongly based on precomputation. Given a rule set, it computes each possible combination of the values specified by the rules. For each point, it evaluates the rule that will match that combination. During the runtime, classifying the packet is straightforward as it is enough to look at the point representing the packet. This algorithm can be implemented in eBPF and is fast, but has a major memory problem: in the worst case, given N rules with K fields, the algorithm can require memory up to N^K. This makes the algorithm unfeasible for large or complex rule sets.

- **Recursive Flow Classification:** This algorithm is an improvement of the Cross-Producting. It builds cross products for pairs of fields and then combines the results by building a cross-product of the cross products. The advantage is that during the combination, many of the pairs are equivalent. This can be used to optimize the memory occupation, by creating classes of cross-products that are therefore not duplicated. This algorithm was discarded as it still requires N^K memory in the worst case, and the runtime is linear, so it does not give any special benefit for large rulesets.

- **Decision tree approaches:** These algorithms are based on the idea of building trees to be walked during the classification, each node represents a value of a field, and the leaves represent the rule matching the branch. However, these

algorithms were discarded as tree exploration requires unbounded loops, that are not allowed in eBPF.

- **Linear Bit Vector Search:** The Linear Bit Vector Search ($LBVS$) is a divide and conquer algorithm, that splits the computation into steps and combines the results of each step to obtain the matching rule. This algorithm can be implemented in eBPF, has a linear complexity scaled by a factor based on the memory parallelism, and can support an arbitrary number of fields and wildcards. This is the algorithm that was used in the eBPF firewall and that will be extensively presented in the section 4.2.

Chapter 4

Design

4.1 Semantic

The aim of the developed firewall is to let the user switch from Iptables to it and notice only the performance gain, without any difference in the behavior. This means preserving Iptables semantic, by ensuring that the action taken on each packet would be exactly the same by Iptables and the eBPF firewall. Under the constraints expressed in 4.3.2 and 7, the goal has been reached.

The major problem to solve was the semantic and architectural difference between eBPF and the Netfilter. As presented in the section 1.2, the Netfilter follows the packet to take routing and mangling (i.e. NAT) decisions. However, eBPF programs can intercept the traffic on the TC and XDP hooks, and these hooks are hit before any Netfilter component. The consequence is that during the execution of the eBPF programs, there is no available information about the routing decision that the Netfilter will take.

This lack of information about the path that each packet will follow represents a problem in respecting the semantics of the Netfilter. For example, a user configuring policies on the input chain expects only the packets directed to the applications running on the host to be filtered by these policies. A possible solution is to implement a new eBPF hook point where this information is available. However, delaying

packet processing would most probably not be the best solution: it would not require any additional computation to simulate Netfilter's behavior, but it would delay the packet processing (with respect of the TC and XDP hooks) and consequently lead to performance degradation.

The solution adopted in the eBPF firewall is to guess the choice that the Netfilter will make before the Netfilter itself is hit. This guess provides the information on where the packet will be forwarded, hence allowing to use the proper set of policies. The decision of which chain should process which packet is taken by a dedicated eBPF program that listens and processes all the incoming and outgoing traffic of each interface.

The program is based on a map containing the IP addresses of all the network devices (virtual or physical) registered on the host. With this information, the following decision process is applied based on where the packet is captured.

- On the **ingress** hook, where only the traffic coming from outside the interfaces is caught, each packet may be directed either to an application running on the host or to another network interface. To classify these packets, the firewall checks the destination IP address against the addresses map. If a match is found, the packet will be processed by the forward chain, otherwise by the input one.

- On the **egress** hook, each packet may come from the host or from another network device. The firewall uses the same method as for the input hook, but by checking the source address of the packet. If it is found, the packet is processed by the output chain. Otherwise, the packet has already been processed by the filter chain, and so it is just forwarded without any further check.

To always keep the address map up-to-date, the firewall control plane listens for Netlink notifications and modifies the tables accordingly (e.g. if a new network card is detected, its address is inserted in the map).

This solution has a strong performance benefit, as it allows to leverage on the early packet processing, but is based only on IP addresses. Such a solution does

not address the bridged traffic, that is managed by Iptables independently from the addresses. This is a current limitation of the implementation.

4.2 Matching algorithm

This section presents a description of the chosen classification algorithm, the Linear Bit Vector Search. Further details can be found in [18].

The LBVS is a *divide-and-conquer* algorithm. It requires the rule database to be split into multiple classification data structures, based on the number of protocol fields it is operating on. Each structure is used by a separate algorithm step, and each step provides intermediate results that in the end are combined to obtain the final solution.

Classification.

The classification process is based on a number of data structures that depends on the number of supported fields. There is a table for each field, and each table contains all the values that appear in the rule set for that field and a wildcard for rules not requiring matching at all. Each value is mapped to a list of rules matching it.

To link a packet with the proper rule, each of the packet fields is used to perform a lookup on the corresponding map. Each lookup returns a list of rules that are satisfied by that particular field. For example, in figure 4.1, the source address of the incoming packet matches all rules (hence, the sequence of ones). By crossing these partial results, it is possible to retrieve the set of rules matching the whole packet, and the first one is taken. In the example, the result is that the first two rules match the packet, and the first one is selected.

In any step of the algorithm, if a lookup fails, the algorithm can infer that no match has been found and it can apply the default action. This may happen if nothing is returned by a lookup in a map.

Preprocessing.

The algorithm preprocessing starts from a sequence of rules and returns a sequence of separate maps, each one mapping all the values taken by the field in the rule set with the list of rules matching it. The rules are tracked using bitmaps, where a bit set means that the rule corresponding to that bit requires matching on that field.

The algorithm follows these steps for each field:

1. If a rule requires matching on the field, the value is searched in the map and, if present, the corresponding bit vector is updated, otherwise a new entry is created with a bit set on the position of the rule. In this step, it is guaranteed that the value will appear exactly once in the map, and its bit vector will track all rules. For example, starting with the first rule of the example, the destination port map does not contain the value 80 and a bit vector is created with the first bit set. When the second rule is elaborated, there is already an entry for the port, and so its bit vector is updated by setting the second bit.

2. If at least one rule requires wildcard matching (e.g., a rule that does not specify any value for the given field), a wildcard entry is inserted (based on the field) and mapped with a bit vector where the only bit set is the one corresponding to the current rule. In the example, the last rule has a wildcard destination port, so a wildcard entry is inserted with the last bit set.

3. For each wildcard rule, all bit vectors in the map are updated by setting the bit corresponding to the rule to 1, as a wildcard rule matches independently from the value. In the example, only the last rule requires matching on the address, so all other rules are a wildcard. Hence, both bit vectors have the first four bits set.

In the end, for each field, the map associated to it contains a number of entries N equal to the number of distinct values of the field itself, plus an additional entry representing the wildcard entry (for rules that do not require a matching on this field).

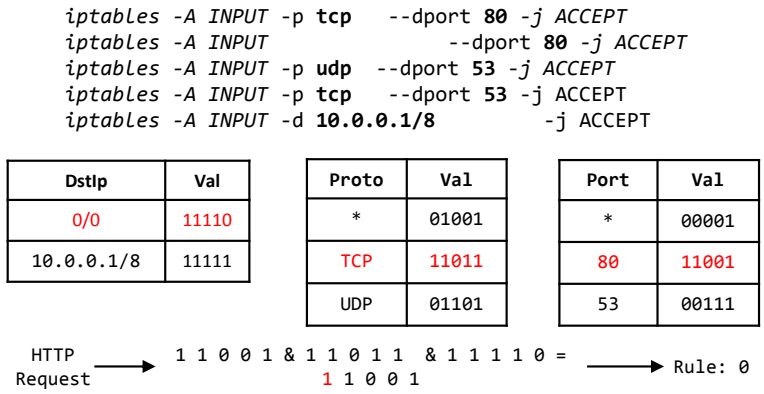

Figure 4.1: An example of preprocessing and classification using the LBVS.

Complexity.

Tracking rules using bitmaps enables the evaluation in large groups: the larger the memory parallelism the larger the groups. While theoretically the LBVS is a linear algorithm, this factor enables on a 64-bit architecture a speedup of 64 times, compared to a traditional linear search.

The main drawback of the LBVS is the preprocessing cost, as presented in section 6: it is not trivial to cache results of the tables, and it is usually easier to populate the data structures from the ground every time there is an update [18].

Memory occupation is not optimal but is acceptable: given N rules, the bit vectors will be N bit long, and in the worst case the memory occupation will be $N^2 * K$ bits, where N is the number of rules (and, consequently, the length of the bit vectors), K the number of fields required by the rule set (and, consequently, the number of different tables), and W the width of the memory access.

4.3 Architecture

The eBPF firewall has been developed in three main layers, depicted in figure 4.2.

The first one is in charge of interacting with the user. It strongly leverages on `libiptc` and will be presented in section 5.1. This layer converts Iptables commands

(e.g. rule insertion or dump) into an intermediate format and pushes them to the control plane.

The second layer is the control plane. It is in charge of processing the commands, prepare the data structures and manage the data plane. Considering that the control plane has no role in packet classification, it is not written using eBPF but instead using C++, allowing for complex processing. It will be presented in the section 5.

The third layer is the data plane. It will be extensively described as it is the core of the Firewall. It processes packets and it has to perform the minimum necessary operations for filtering the traffic at a high rate.

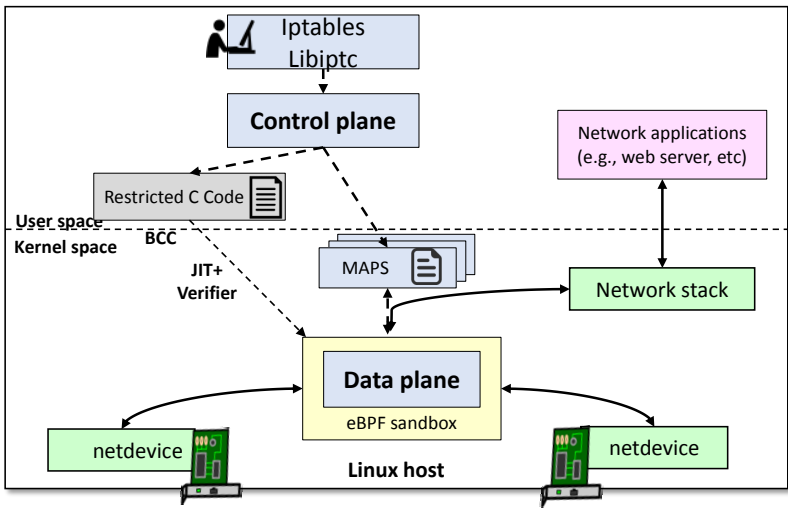

Figure 4.2: Overview of the main components.

4.3.1 Pipeline architecture

Given the limited size of the eBPF programs, the firewall data plane has been divided in a sequence of programs, each one implementing a piece of the overall filtering logic. Figure 4.3 depicts a high-level view of the data plane structure.

Starting from the leftmost part, the first program encountered by the traffic is a parser in charge of interpreting the raw bytes and moving the content in the structures further detailed in 5.2. The choice of using such a module was taken as eBPF provides to the program a raw array of bites, that must be scrolled, converted

and copied to be properly used. Considering that almost every program of the chain requires some of the packet fields, an optimization is to perform this operation once and share the results between programs.

After the parsing, packets are intercepted by the first connection tracking module. This program, presented in section 4.3.3, is in charge of associating each packet with the connection it belongs to.

The next module is the one in charge of enforcing the filtering semantic (section 4.1). Its role is to guess the traffic path and forward it to the chain in charge of filtering it. In the eBPF firewall, a chain (presented in section 4.3.2) is the sequence of eBPF programs that implement the LBVS and perform the packet classification.

If the rule matching the packet allows it to continue in its path (i.e., the firewall does not drop the packet), it is intercepted by another connection tracking module, also presented in section 4.3.3, that tracks if the packet has triggered changes in the connection.

By leveraging on the property of the LBVS, any step of the chain can assert that the packet does not match any of the policy configured. In this case, the default action is taken by forwarding the packet to a module (not represented in the figure) that will enforce the action. This choice is motivated in section 4.3.4.

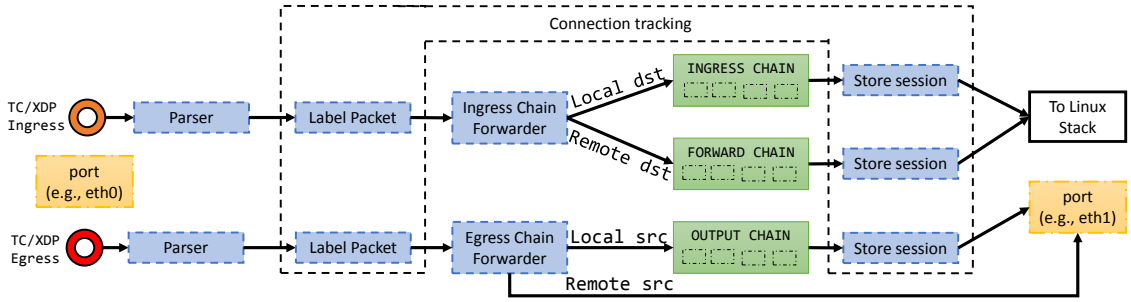

Figure 4.3: Overall data plane structure. From[4]

4.3.2 Chain

A typical policy set for an enterprise firewall requires many (in the order of thousands) complex (on many fields) rules. Hence, supporting such sets is a key feature

for a firewall. According to [19], enterprise filter sets are mostly based on a combination of IP addresses (and netmasks), transport protocol and ports, and sometimes TCP flags. Typically these fields are not all specified in each rule, therefore wildcard matching is needed.

Even if eBPF puts constraints on the program size and on the loop usage, the LVBS can be implemented in eBPF for complex rule sets. It is, in fact, possible to overcome the technology limits by leveraging the algorithm modularity and splitting it into multiple programs interconnected using tail calls.

The main limit in the LBVS implementation is that it requires performing bitwise AND between the bit vectors to combine the partial results. This requires a bounded loop (the size of the bit vectors depends on the size of the rule set, known a priori), consequently, it can be implemented at the expense of increasing the number of instructions. This translates to a limit to the size of the vectors (and as a consequence, to the number of supported rules). By splitting the data plane in multiple programs, the space necessary to unroll each loop increase, as each program is dedicated to a single field.

Thanks to this architecture, the current implementation supports for each policy a combination of one or more of the mentioned fields and a total of 8k rules per chain.

Each time a packet is received by the firewall, it is forwarded to the proper chain (see section 4.1), in charge of performing the classification. Figure 4.4 represents a high-level view of a chain that performs a lookup on a few fields.

Each eBPF program is in charge of matching one field and update the overall bit vector shared between all programs. To do so,

1. It retrieves the bit vector associated to the field value;
2. If nothing is found, it calls another program in charge of executing the default action;
3. It updates the overall bit vector by applying the bitwise AND with the retrieved one;
4. It calls the next program.

After all the fields have been checked, the result is a bit vector from which the first-bit set represents the matched rule. Finding the first bit can be an expensive operation, so the bitscan algorithm based on the De Bruijn sequence [20] was used to reduce the overhead.

As a side note, the number of fields on which the filter set may require matching at the same time is constrained only by the limit of subsequent tail calls dictated by eBPF (i.e. 32 [21] minus the ones uses for internal purposes for processing the packet before and after the classification). Increasing the number of fields supported is really straightforward as it is enough to write and register one more program in the chain.

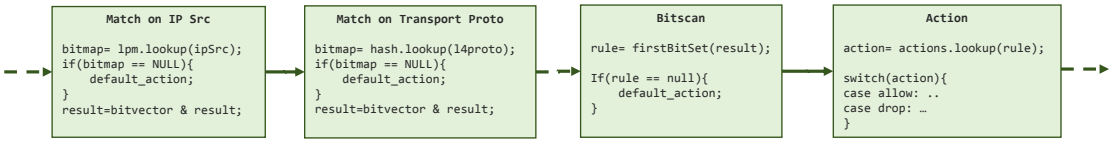

Figure 4.4: Example of a chain of eBPF programs in charge of classifying the packets.

4.3.3 Connection tracking

A network device implementing connection tracking is a device that can store information on the network connections that traverse it and track the state that each connection takes based on the traffic traversing it or on other conditions like timeouts.

A firewall that supports connection tracking can use this additional information to allow the user to configure advanced policies, for example allowing packets that belong to established connections. It is therefore important for the user to know the semantic behind the states assigned to each connection.

States

In the eBPF firewall, the semantic follows the one enforced in the Netfilter connection tracking module. Each packet is associated with the following labels, based on the connection it belongs to [22]:

1. **NEW:** The packet does not belong to any existing connection, hence it is starting a new one. A common case is the TCP *SYN* flag.

2. **ESTABLISHED:** The packet belongs to a connection considered opened. Such a connection has seen both a packet flowing from one host to the other and a reply in the opposite direction.

3. **RELATED:** There is an established connection to which the packet does not belong to, but is related to. The related connection is just a consequence of the established one. As an example, ICMP errors are related to the connection that generates the error, or FTP data connections are related to the control ones.

4. **INVALID:** The packet can't be associated with a connection. It may happen because a protocol is not supported, or the packet is not foreseen (for example an ICMP error not related to any connection).

Architecture

The semantic of Iptables/Netfilter has to be preserved in every feature of the firewall, connection tracking too. However, there are two architectural limits to achieve this goal: the Netfilter hooks and the technology limits.

The first point was already presented in section 4.1, and it is a problem for connection tracking as the Netfilter can follow the packet through all its lifecycle, whereas eBPF is not able to. Consider that each packet may trigger a state change in a connection (e.g. a *RST* that closes a connection), it is important that only the modifications associated with legit traffic are kept. By not being able to follow the packet through the Netfilter, it is not possible to completely enforce this constraint.

The second point is important if a feature-complete implementation of the connection tracking is needed. To completely replace the Netfilter connection tracking, a module capable of performing deep validation (e.g. checksums, IP reassembling, TCP window checks) and supporting many protocols is needed. A possible approach

in eBPF would be to move the current implementation in an eBPF helper embedded in the kernel, as it is being discussed in the mailing list [1].

Nonetheless, the eBPF firewall supports a basic connection tracking for stateful filtering of UDP, TCP, ICMP traffic.

The implementation leverages on a shared map that tracks the connections. The key used to index the map is represented by the tuple *source IP address, destination IP address, transport protocol, source transport port, destination transport port*. The value associated with the key is the connection state, the connection lifetime and the last seen sequence number (meaningful only for TCP). When the fields of the packet are used to perform a lookup, they may either match it in the forward direction (i.e. source address, destination address, source port, destination port) or in the reverse direction (i.e. destination address, source address, destination port, source port). From this structure, the program can infer information on the state of the connection, as will be presented in the next paragraph.

The workflow is split into two steps. The first step is to associate each packet with the connection it belongs to, while the second step (executed only on the allowed traffic) tracks possible modifications on the connections. As shown in Figure 4.3, these two functionalities were assigned to two separate eBPF programs, one in charge of intercepting all the traffic at the beginning of the chain, the other reached only by allowed packets at the end of the filtering process.

Both the connection tracking programs implement a state machine for each supported protocol. The first program checks the packet protocol and if it is not among the supported ones, the process stops and the packet is labeled as invalid. The difference between the two programs is that when a decision is taken on the packet, the first program labels it, the second program updates the table with the next state. The invalid state is used to label any packet outside the expected flow (e.g. an Echo Response without an Echo Request). The actions performed are different for each protocol.

[1] Implementing connection tracking in eBPF as a kernel helper has been discussed in the Linux mailing list: https://www.mail-archive.com/netfilter-devel@vger.kernel.org/msg11139.html

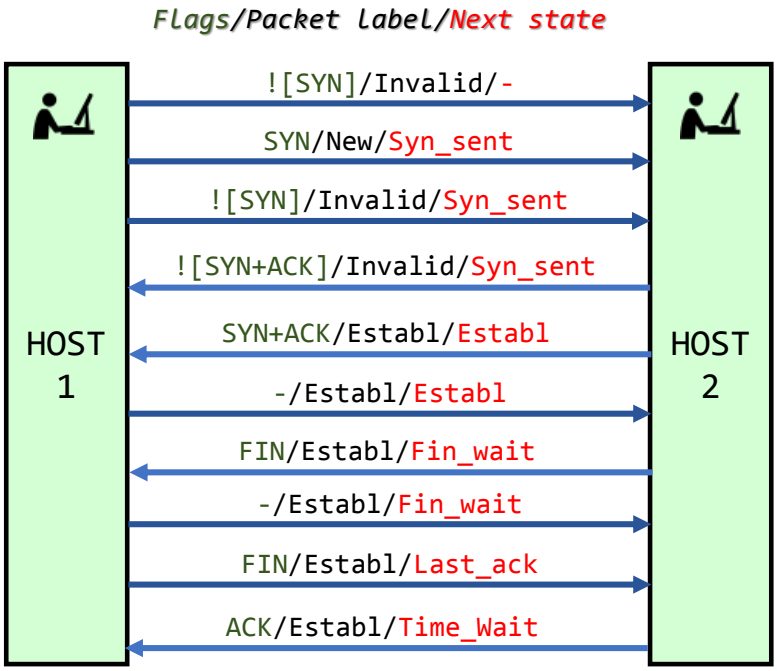

Figure 4.5: Simplified version of the TCP connection tracking state machine.

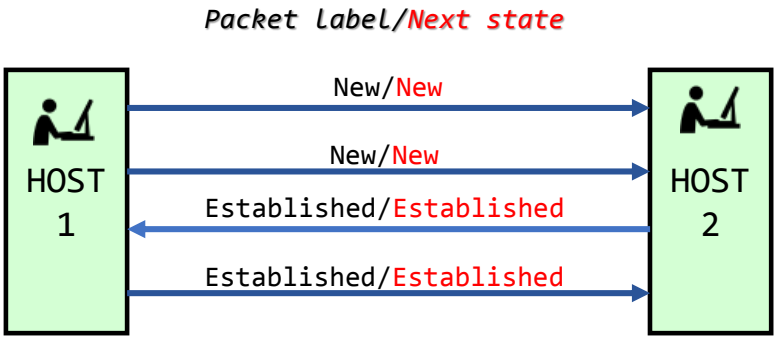

Figure 4.6: Simplified version of the UDP connection tracking state machine.

For **UDP**, a new connection is started by any packet that does not match any entry in the map. When a packet that does not belong to any connection is allowed, the just-opened connection is stored in the session table and all subsequent packets will be treated as new until a packet in the opposite direction (i.e. an answer) is allowed by the firewall. An answer can be recognized as it will match the entry in the reverse direction. In this case, the connection will be established. UDP has no closing mechanism, so the connection will eventually just expire.

ICMP packets may either represent an error (e.g. Network unreachable) or carry some special message (e.g. ICMP Echo). In the former case, the standard guarantees that the header of the original packet can be retrieved from the error payload. This provides enough information to perform a lookup in the session table. Any match to an existing connection implies that the packet belongs to a RELATED connection, otherwise it's treated as invalid. In the latter case, only ICMP Echo Request/Response is handled. An echo request is treated as new, the response is treated as established.

TCP has many ways of starting or closing connections. In the firewall, the canonical opening three-way handshake (*syn, syn+ack, ack*) and the closing four-way handshake are supported, plus the reset. A connection is considered new during the three-way handshake. When this is concluded, the connection becomes established. However, the internals of the state machine need more states to track the entire connection lifecycle:

1. First SYN received, the packet sequence number is stored in the session table. The connection is new.
2. $SYN + ACK$ seen in the reverse direction with a valid acknowledgment number. The connection is in the *syn_rcv* state.
3. ACK received in the forward direction with a valid acknowledgment number, the connection is established.
4. $FIN + ACK$ received, the connection is in *fin_wait* until the other side closes, too. Packets are still treated as belonging to an established connection.
5. $FIN + ACK$ received the other side. The connection will be shortly considered closed.
6. RST closes the connection independently from its state.

When a packet is labeled as being part of a connection, the matching algorithm can treat the state as any other field. A specific program is added to the chain to specifically match the state.

An important limitation of the implementation is a missing clean-up mechanism. Each connection has an associated lifetime, that allows to the connection tracking modules to understand if a connection is still valid or it is expired, every time a packet belonging to it is found. However, dangling connections (e.g. the one opened by UDP, that has no explicit closing mechanism) are not removed from the session table when they expire. The problem behind this choice is that it's not possible to clean up the tables from the control plane without incurring in race conditions, as no locking or synchronization mechanisms are available. The adopted solution is to use a Least Recently Used map, available in the kernel for eBPF programs so that the session table can't be saturated. However, a beneficial approach would be having the support for a specific helper in the kernel that supports automatic clean-up.

4.3.4 Counters

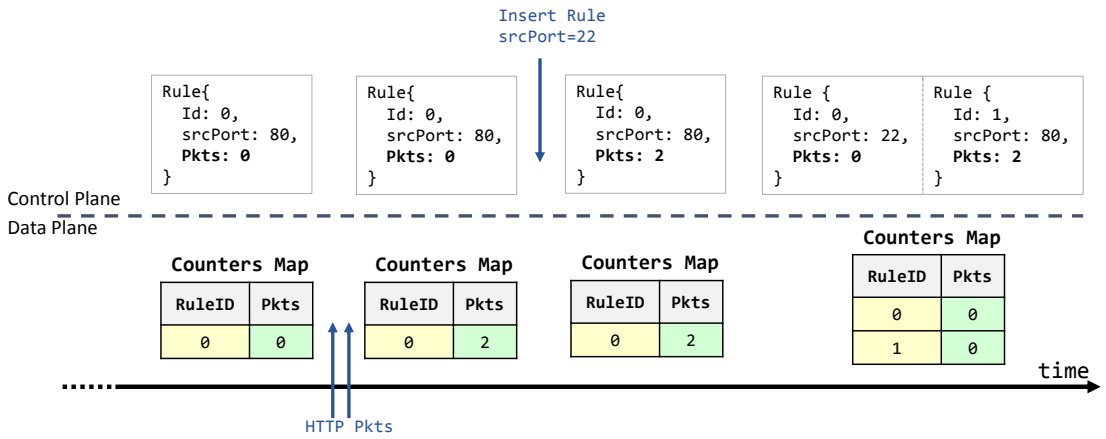

Figure 4.7: Example of counters management.

Firewalls usually offer to the user the possibility to export statistics on how the rule set is filtering the traffic, i.e. how many packets or bytes have been matched by each rule. This is particularly useful for debugging (e.g. it allows to understand if the traffic is matching the expected rules or the configuration is wrong) and for detecting attacks (e.g. a rule dropping a large number of packets from a given address may show a DoS).

To track these counters, the eBPF firewall leverages on the module that associates the rule with the packet once the classification process is concluded. However, this module has no knowledge of the default action, because as pointed in section 4.3.2, any module can at any time infer that the packet does not match any rule and apply the default action.

Starting from the non-default rules, the last program of the chain has a map entirely dedicated to tracking the number of packets and the number of bytes matched by every rule. This structure maps the rule number (e.g. first or second rule) with the respective counters. When an action is taken, the program performs a lookup using the rule number and increments the associated statistics.

For the default action, a different strategy is needed as the programs in the chain do not execute directly the operation, but instead executes a tail call to a dedicated program that performs it. Therefore, it is left to this program the responsibility to update the counters for the default action using a separate map.

However, use this approach two problems need to be faced. The first one is related to the chain update, that requires a new set of programs with new tables to be injected. This choice is motivated in section 4.4.3, but its main drawback is that the content of the tables is not preserved in the operations, hence the counters are lost. The second one is related to the coherence of the data: if there are three rules and the second rule is deleted, the third rule will become the second, and counters reference will become inconsistent.

The adopted solution solves both problems. It uses an incremental approach in which the control plane keeps the whole story of the counters of each rule, and the data plane tracks only packets between one update and the other. When the control plane receives a request to change the rule set, it loads the statistics from the data plane and sums them with the one he has. In this way, they are saved, associated with the proper rule, and will not be lost. Then, it triggers the atomic update of the rules that will clear the map in the data plane.

To avoid any overhead on the traffic processing due to the operations on the counters, the data plane is never stopped. As a consequence, the traffic processed right after the counters have been fetched from the data plane until the new chain is

injected, won't be tracked. This is a trade-off that may result in a reduced precision of the statistics but won't affect performance.

Figure 4.7 shows an example of the operations flow. It represents the evolution over time of the control plane (on the top) and on the data plane (on the bottom) with respect to the counters. If the starting point is a rule set with only one rule, when packets match that rule the data plane increments the associated counters. When the control plane receives an update, it first fetches the counters from the data plane, then updates the rules and in the end, it flushes the counters in the data plane.

4.4 Dataplane optimizations

The data plane presents a modular architecture that, combined with eBPF, allows for optimizations to be performed at runtime when it is injected by the control plane.

This section presents the three optimizations currently implemented in the firewall. The first one is the code tailoring, meaning that the control plane injects only the pieces of the data plane that are strictly needed to respect the configuration. The second one exploits the possibility of using tail calls to jump from one program to another and the information provided by the connection tracking to avoid performing the classification where possible. The third one leverages on the dynamic code injection at runtime to guarantee atomic update of the rule set without any locking mechanism.

4.4.1 Tailored dataplane

The Linear Bit Vector Search is a modular algorithm, meaning that it is a sequence of steps that can be split into multiple programs, as in the firewall data plane. Moreover, the combination of eBPF and BCC enables to build the data plane code from the control plane and inject it in the kernel at runtime.

This combination is an opportunity to build the data plane on an as-needed basis: the code handles only the fields whose matching is required by the configured rules,

without unnecessary parsing and processing of the fields that are not required. For example, if there is no rule requiring a match on the transport destination port, the module in charge of matching it is omitted and not injected at all. This translates into a higher efficiency due to less computation and memory accesses.

Furthermore, when a rule requiring that match is configured, the pipeline can be updated by inserting the respective module to support the new classification step (in the example, the one that operates on the destination port), and all the bit-vectors in the maps are updated in order to handle the new rule as well.

A practical example of how the chain may be changed based on the rule set is depicted in figure 4.8.

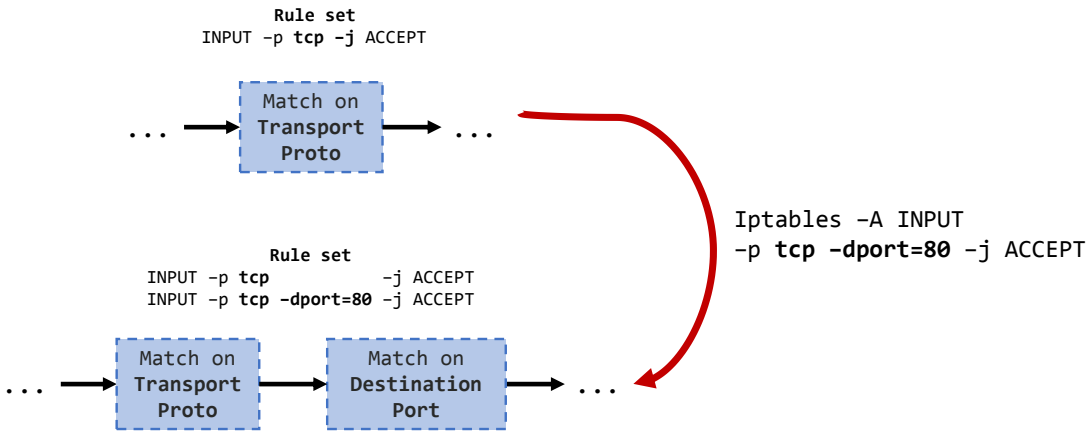

Figure 4.8: Example of data plane tailored on the rule set.

4.4.2 Leveraging on connection tracking

Stateful firewalls, i.e. firewalls that support connection tracking, are usually configured with the first rule allowing all the traffic belonging to established connections. The reason behind this is that usually most of the traffic belongs to these connections and, in the algorithms with linear complexity, the sooner a match is found, the higher are the performance.

In the developed eBPF firewall, this situation was considered and an optimization was built on it. If the configuration respects the described criteria, the firewall detects

packets belonging to established connections in the first connection tracking module (hence, before the classification process) and just accepts them skipping all the matching pipeline.

Such an optimization avoids unnecessary computation and translates in the common case in a considerable performance gain, as shown in section 6.

4.4.3 Atomic update

As presented in section 4.3.2, the processing chain is a sequence of programs, each one using its own map. However, if the rule set is updated, it is likely that the update is reflected on more than one map because each field of the new rule is tracked in its own table.

The content of *all* the maps must be updated atomically: it must not happen that a packet is processed by some tables updated and some not. In a case like this, the resulting decision after the combination of the partial results cannot be foreseen, and this is not acceptable in a firewall. The solution is to update all the maps *in a single shot*.

The eBPF suite guarantees that the update of a single map is atomic, so if a program is changing a map the other programs will not see a partial result. However, there is no mechanism to update more than one map at once. The reason behind this may be that a synchronized update may require to prevent programs to accept traffic during the update, with a consequent unacceptable service disruption because of the inability of the data plane to process traffic during that interval.

To guarantee both atomicity and negligible disruption during the update, our software control plane implements a strategy heavily relying on the possibility to inject new eBPF programs at runtime and reloading the existing ones. Every time the policy set is changed, the current chain continues to process the traffic while the control plane handles the update by following these steps, also depicted in figure 4.9:

1. Based on the new rule set, a new chain is assembled and injected, and the new maps are filled with the updated values.

2. The module used to enforce the semantic is reloaded in order to point to the updated chain and steer the traffic towards it.

3. The old chain, obsolete, is unloaded.

As soon as the module is reloaded (in step 2), the configuration is updated and the traffic is filtered by the new chain. There is no service disruption as there is always a chain filtering the traffic, and reloading the first program has no impact. This is possible thanks to the reload technique described in [8]: while the new program is compiled and injected, the old one keeps handling the traffic, and when the new one is ready, maps of the old instance are attached to it and the programs are atomically swapped by substituting the pointer to the old program with the new one, and the old program can be safely unloaded.

It is worth noticing that eBPF guarantees that a program, during its execution, cannot be unloaded, interrupted or modified until it completes the task. In this way, if a packet is inside the chain, it is guaranteed that it will complete its path (in the old configuration) and only then the chain will be unloaded.

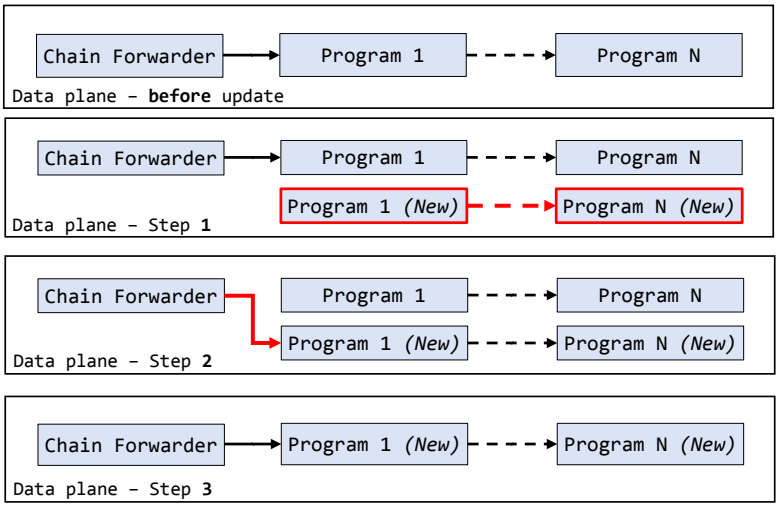

Figure 4.9: Atomic update of the rule set.

Chapter 5

Implementation

5.1 Syntax

Even if the eBPF firewall preserves the Netfilter syntax, the project aim is to offer a clone of Iptables. To do so, exporting the same interface to the clients is a must.

Designing an interface that allows user-space applications to inject configuration in the kernel space is a non-trivial task. One of the main problems of Iptables itself is exactly its user-space interface, due to its low efficiency and its undefined behavior when multiple clients inject rules. This topic is subject of many discussions due to bpfilter being included in the kernel[1].

For the eBPF firewall, it was chosen to re-use the existing code already implemented and largely tested of Iptables itself, plus its underlying library Libiptc. Moreover, instead of already replacing Iptables, the eBPF firewall is shipped as a separate executable. In this way, if the user needs functionalities not yet supported, he can just use the Linux Iptables.

As depicted in figure 5.1, each command issued to the firewall is intercepted by a slightly modified version of Iptables/Libiptc. First, the commands are sanity-checked and parsed by the default code of these two modules. After this, before the

[1]https://www.mail-archive.com/netdev@vger.kernel.org/msg217183.html

modules actually inject the commands in the kernel, a customized code blocks them and uses the already built data structures to create an intermediate representation forwarded to a shell script. This script in charge of forwarding them to the REST daemon exposed by the control plane, that finally updates the configuration.

If the commands require functionalities not yet implemented, an error is returned to the client. For example, the current implementation does not support ranges (for example, to express a matching between port 10 and 12 it is not possible to write 10:12) or negations (except for the TCP flags).

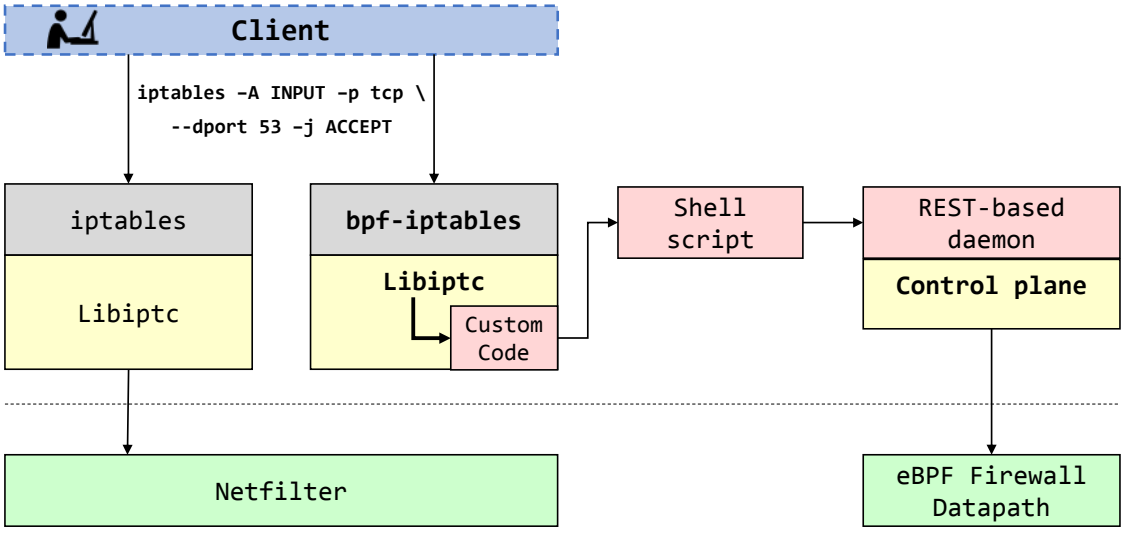

Figure 5.1: Userspace utils to preserve iptables syntax. This picture is based on [3]

5.2 eBPF Maps

In eBPF, maps are data structures needed to store data between different executions of the programs and to exchange data between user and kernel space (i.e. between the control plane and the data plane). In the firewall implementation, maps are used to share the state of the matching through the chain and to perform the matching in each program (i.e. to store the bit vectors).

The state shared between programs is composed of two components: the packet fields and the bit vector. For both, *percpu maps* are used because they allow sharing

data between programs and with minimum overhead. The packet fields are stored in a structure (5.1) that contains only the minimum necessary fields to perform connection tracking and classification. Bit vectors are treated as arrays of 64-bit elements, and every step of the chain stores the result up to that point in the shared map.

Each program in the chain requires a structure to map the fields and the rules matching them, as illustrated in section 4.2. These structures are independent, so the most suitable one can be chosen for each supported field. In the implementation, the following choices were made:

- **IP addresses:** A Longest Prefix Match map was chosen. It is the most natural way to treat addresses as it allows the rule set to support both IP and netmasks by using the LP algorithm.
- **Transport protocol** and **ports:** Hash tables were used. In this case, an array seemed the most obvious choice because it was possible to index it using the fields (unsigned integers) with a constant complexity. The problem was that an array as to be entirely allocated, so for example to support port 16000, 16000 entries needed to be allocated, without a real use case. This memory allocation was considered not acceptable, and hashmaps were chosen. However, for a moderate number of fields, these structures still provide constant complexity.
- **TCP flags:** to manage the flags, it was necessary to consider that every single flag can be either wildcard, set or not set. The map storing flags is an array map of 256 elements, where each index represent combinations of the flags. When a rule is inserted, the control plane chooses which combinations the one required by rule satisfies. Figure 5.2 shows an example of preprocessing and matching in a simple situation with only three flags.

Listing 5.1: Packet fields shared across the data plane

```
struct packetHeaders {
  uint32_t srcIp;
  uint32_t dstIp;
```

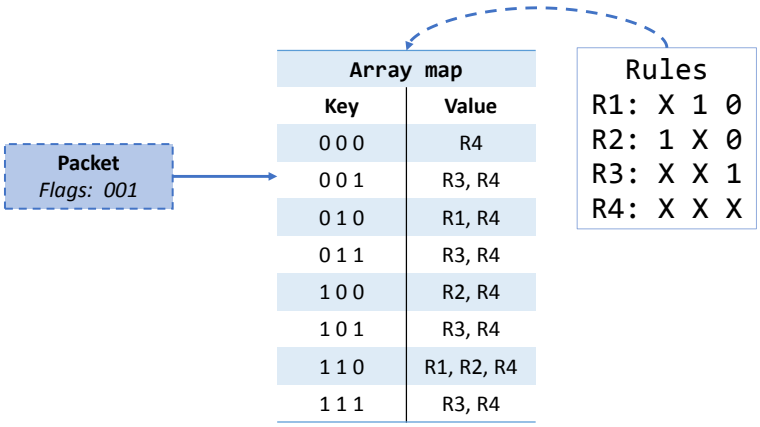

Figure 5.2: Flags processing in the map.

```
  uint8_t l4proto;
  uint16_t srcPort;
  uint16_t dstPort;
  uint8_t flags;
  uint32_t seqN;
  uint32_t ackN;
  uint8_t connStatus;
};
```

5.3 Code structure

The firewall implementation has been designed based on the following considerations:

1. The software has to live on two different levels using two different technologies, namely the control and data plane. The control plane is written in C++, hence has the maximum flexibility and usability. The data plane is written in restricted C following the limits of eBPF and has to be modified at runtime by the control plane.

2. The software is encapsulated by a framework under development that, while

offering a number of primitives to help the developer abstract the technicalities of eBPF, forces a certain structure. This framework is itself based on another framework, BCC.

Based on these considerations, all the code was designed to achieve the maximum flexibility under the constraints forced by the framework and encapsulate the complexity of the underlying data plane in C++ classes. The overall structure is depicted in figure 5.3 using a *slimmer* UML notation to highlight the dependencies of the main components.

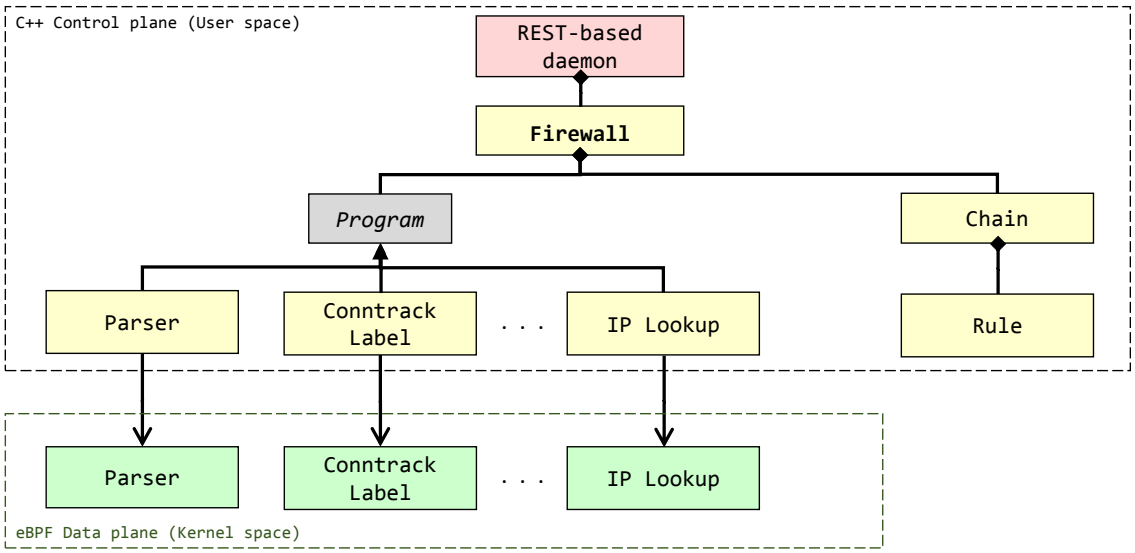

Figure 5.3: UML-like diagram of the code structure.

Starting from the top, the Firewall class is the only entry point of the service. This class exposes an API to insert and fetch data from the service. In the current implementation, the API is used by the REST daemon offered by the overlying framework, but there are no restrictions on the client. A Firewall object can be seen as an orchestrator of the whole system: it has knowledge of every chain, of every program belonging to each chain, and of the overall configuration (e.g. to which network devices it is attached to).

The main abstraction is the Program class. It is a pure virtual function, and it forces the classes implementing it to offer an almost uniform interface to manage

the data plane. Using eBPF programs from the control plane is not straightforward, even less if the programs are built during runtime based on a number of parameters. This complexity is encapsulated inside each class implementing Program, offering to the clients some clean and readable method to perform the tasks. A simplified version of this class is in the listing 5.2. A program is characterized by a unique identifier, the index, the chain it belongs to (the *direction*), and its code. When a program is created, the skeleton of the data path code is provided in the constructor. The `getCode()` method is the most important one, as it is the one starting from the skeleton and building the real data path code based on it. Further details on how the datapath is built will be provided in the next paragraphs.

Listing 5.2: Program interface.

```
1  class Program {
2  protected:
3    int index;
4    ChainNameEnum direction;
5  public:
6    virtual std::string getCode() = 0;
7    void updateHop(int hopNumber, Program *hop,
8                   ChainNameEnum hopDirection);
9    Program *getHop(std::string hopName);
10   int getIndex();
11   bool reload();
12
13   Program(const std::string &code, const int &index,
14           const ChainNameEnum &direction, Firewall &outer);
15 };
```

As hinted, each of the classes representing a program in the data plane implements the Program interface. In the figure, these programs are *Parser* and so on. Moreover, each program offers a method to fill the map of the encapsulated data plane program. For example, 5.3, the method accepts a C++ fashioned map of IP addresses and bit vectors, and internally it manages the update of the data plane

map by creating a proper string and building the map name based on the proprieties of the program.

Listing 5.3: Accessing to the data plane maps.

```
1  void IpLookup::updateMap(
2      const std::map<struct IpAddr, std::vector<uint64_t>>
           &bitvector) {
3    for (auto ele : bitvector) {
4      updateTableValue((ele.first),
           fromContainerToMapString(ele.second.begin(),
           ele.second.end()));
5    }
6  }
7
8  void IpLookup::updateTableValue(int netmask, std::string ip,
       std::string value) {
9    std::string key = utils::to_map_format(
10       netmask, utils::ip_string_to_hexbe_string(ip));
11
12   std::string tableName = "ip";
13   if (type == SOURCE_TYPE) {
14     tableName += "src";
15   } else if (type == DESTINATION_TYPE) {
16     tableName += "dst";
17   }
18   tableName += "Trie";
19   if (direction == ChainNameEnum::INGRESS)
20     tableName += "Ingress";
21   else if (direction == ChainNameEnum::EGRESS)
22     tableName += "Egress";
23
24   firewall.get_table(tableName, index).update_value(key, value);
```

```
25 }
```

Datapath programs are written using a combination of *static* code and *dynamic code*. The programs are filled with macros that enable or disable pieces of code, and they are defined from the control plane getCode() function based on the environment. For example, in the listing 5.4, the `_NR_ELEMENTS` macro defines if the code has to be activated based on the size of the bit vector, and determines how many time the loop will be unrolled. The macro `_TYPE` defines if the module is used to match the source or the destination address, and so on. There are two reasons for this choice. First, a single program can be used for multiple functionalities, for example, thanks to the macros there is no need to write a different program to match the source and destination address in each chain, one program can take any function. Second, it's proven [8] that embedding parameters in the data path code instead of letting the program retrieve the values in maps benefits performance.

There are two more *main* classes in the implementation: the Chain and the rule. The chain keeps all the Rules configured on it. When the firewall receives an update, it forwards it to the proper chain that effectively manages to update its configuration. To do so, a function is triggered that starting from the C++ rule set returns a set of maps ready to be used by the LBVS. These maps are consumed by the function in charge of performing the update of the data plane programs that first loads only the programs needed, second populates their maps

Listing 5.4: Snippet of datapath code with macros.

```
1 static __always_inline struct elements *getShared() {
2   int key = 0;
3   return sharedEle_DIRECTION.lookup(&key);
4 }
5
6 static __always_inline struct elements *getBitVect(struct lpm_k
     *key) {
7   return ip_TYPETrie_DIRECTION.lookup(key);
8 }
9
```

```
10 static int handle_rx(struct CTXTYPE *ctx, struct pkt_metadata
       *md) {
11 #if _NR_ELEMENTS > 0
12 #if _NR_ELEMENTS == 1
13         // Code to match a single element
14 #else
15 #pragma unroll
16       for (int i = 0; i < _NR_ELEMENTS; ++i) {
17                 // Code to match multiple elements elements
18       }
19 #endif
20   call_bpf_program(ctx, _NEXT_HOP);
21 #else
22   return RX_DROP;
23 #endif
24 }
```

Chapter 6

Benchmarking

6.1 Methodology

For performing correct and meaningful benchmarks of the developed firewall, there are a number of factors to consider:

- **Rule set:** each firewall configuration leads to different performance measurements. For example, if the testing traffic is allowed by rules at the beginning or at the end of the set, it results will be different. Even the complexity of the rules themselves has a consequence on the performances. This aspect will be further presented in section 6.1.1.
- **Test traffic:** the RFC-3511[1] states that each test should be performed using a set of different packet sizes. However, it gives no indications on protocols or other traffic characteristics. An important consequence of choosing the traffic structure is that it strongly influences how the DUT is tested. For example, if the traffic flow is composed by a single repeated packet, the DUT will process it using only a single CPU. A variegated traffic will be distributed on more processors.

[1]Benchmarking Methodology for Firewall Performance: `https://tools.ietf.org/html/rfc3511`

- **Testing conditions:** the RFC-3511 states that the test should be performed as in figure 6.1, where two devices are connected through two network interfaces, the traffic is generated from a device, forwarded from one NIC to the other by the device under test, and received back by the generator to track the performance.

- **Parameters:** the RFC-3511 suggests a number of parameters to benchmark. Particularly interesting are the IP throughput (defined by the RFC-1242 as at the maximum rate at which none of the offered frames are dropped) and the latency (defined by the RFC-1242 as the *interval starting when the end of the first bit of the input frame reaches the input port and ending when the start of the first bit of the output frame is seen on the output port*). Moreover, parameters like the time to insert rules in batches or to insert a single rule in the head or in the tail of the rule set can be of interest [23].

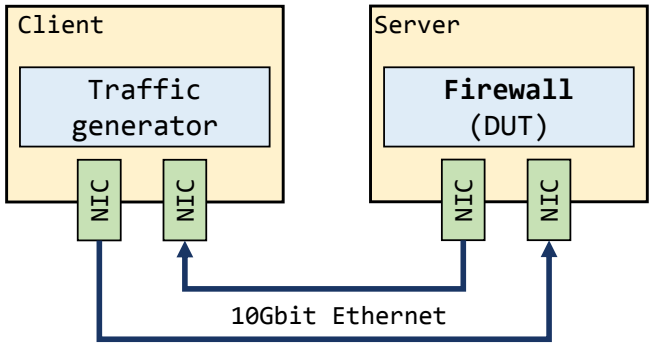

Figure 6.1: NIC-to-NIC Benchmark setup.

6.1.1 Rule set

The rule set has a direct consequence on the quality of the benchmark. To design a proper rule set, three main strategies can be found in literature:

1. Synthetic rule sets. This approach, used in [24], is based on generating rule sets based on some internal criteria, usually random rules.

2. Synthetic rule sets based on real ones. This approach is the most common [25] [26] [27] [23] [28]. Real rule sets are analyzed to get some statistics (e.g. fields. the variance of these fields, the type of matching required) and based on these, new synthetic rule sets are created. This technique allows to not disclose real rule sets and to create configurations of many dimensions to perform more tests.

3. Real rule sets [25] [28]. The rule sets can be taken from ISPs firewalls and routers. These sets are usually hard to get and can't be disclosed.

The approach used in the benchmarking the firewall is in between the second and the third. An enterprise firewall configuration based on a Jupiter firewall was provided to the group. A Juniper firewall sees the network as a number a number of zones, each one with its own rule set, and applies application-level filtering. To translate the rule to fit the Iptables syntax, the original rules had to be interpreted and translated accordingly, and this operation could not be performed manually.

The resulting rule set is strongly based on the original one (i.e. same fields, same values) but it has a different semantic. However, for the sake of the benchmarking, the semantic was not important.

6.2 Results

The following tests were performed by comparing Iptables and the developed eBPF firewall in the exact same conditions, using variable rule sets as described in the previous section.

The tests were performed according to the RFC-3511 (here called *NIC-To-NIC setup*) and by using Iptables as an host firewall. Both setups employ to machines, that run an Ubuntu Server 16.04.4 LTS, kernel 4.14.1[2]. Both machines are based on an Intel i7-4770 CPU running at 3.40GHz (four cores plus hyper-threading, 8MB of

[2]In [8] it is highlighted how in newer kernels eBPF is subject to strong performance degradation, probably due to the Spectre/Meltdown fixes

L3 cache and 32GB RAM) connected to each other through two direct 10Gbps links terminated in an Intel X540-AT2 Ethernet NIC.

6.2.1 Nic-To-Nic setup

The configuration in figure 6.1 was used. Linux was configured to enable IP forwarding between the interfaces using the following command

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

to use eBPF JIT with the following one

```
# echo 1 > /proc/sys/net/core/bpf_jit_enable
```

And to disable Conntrack when the eBPF firewall was tested using

```
# echo 'blacklist nf_conntrack' >> /etc/modprobe.d/conntrack.conf
```

UDP Throughput

To measure the throughput, Pktgen-dpdk[3] was used as a traffic generator and receiver. This software has the advantage of directly attaching itself to the network interfaces, skipping the Linux stack and achieving higher precision as there is no delay due to the stack processing. Pktgen was configured to a stream of identical UDP packets, in order to saturate a single core of the DUT. Consequently, throughput in case of real deployment will be much higher due to the traffic balancing implemented by the Linux kernel, which automatically exploits multiple cores.

The throughput was measured when the packet loss rate was one percent. The test was performed for various packet size, However, the most significant measure is when packets are 64Bytes long, as it can easily saturate the DUT.

Figures 6.2 and 6.3 show the results of the tests for Ethernet frames of size 64B and 512B.

[3] http://pktgen-dpdk.readthedocs.io/en/latest/getting_started.html

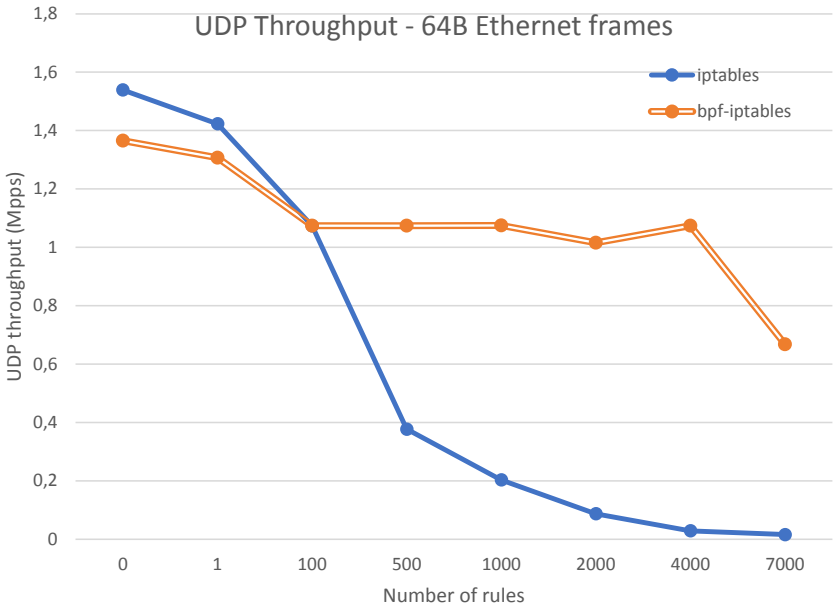

Figure 6.2: Throughput using UDP traffic with 64B frames.

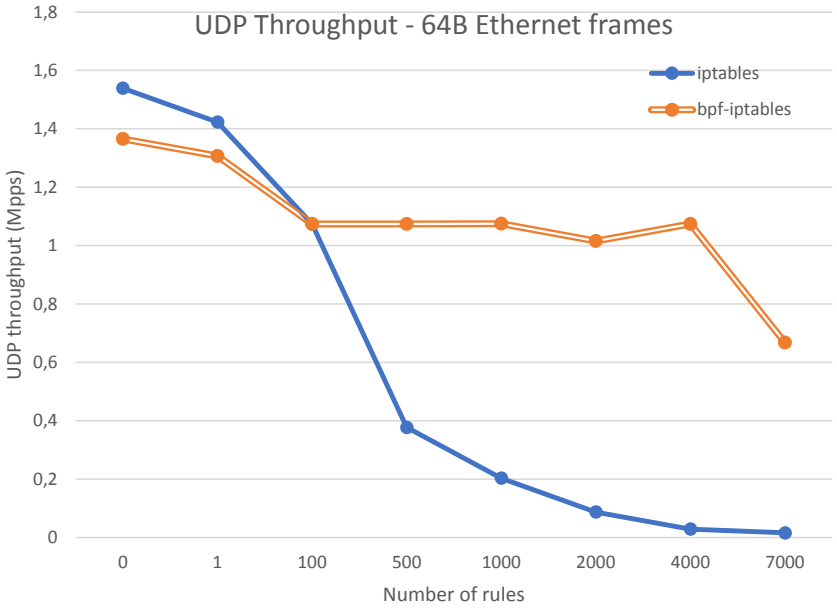

Figure 6.3: Throughput using UDP traffic with 512B frames.

Results are promising, in particular for a high number of rules, where the eBPF firewall definitely outperforms Iptables. An accurate analysis of the data suggests that the main overhead lowering the performance of the eBPF firewall derives from letting Linux perform the routing. This overhead is not negligible when few rules are

involved. For example, for the sake of the tests, in figure 6.4 the throughput without using the Linux routing (and hence, breaking the semantics) was evaluated.

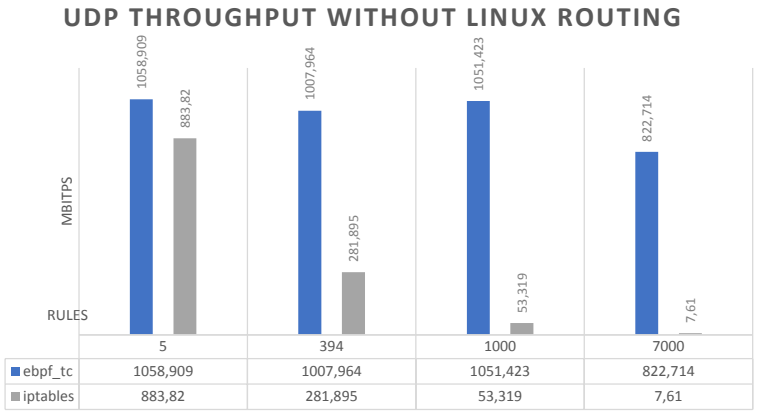

Figure 6.4: Throughput using UDP traffic with 512B frames and bypassing Linux routing.

ICMP Latency

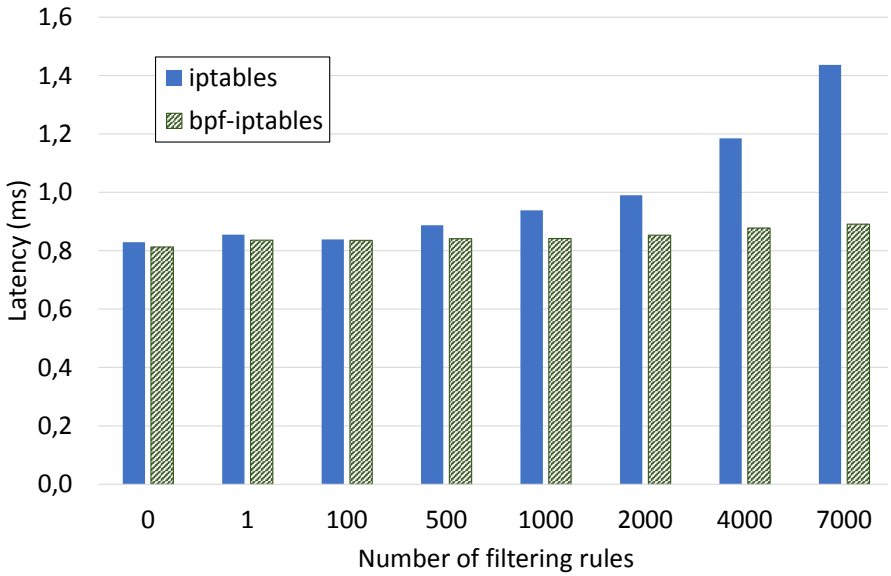

Figure 6.5: Latency measured using the Ping tool.

Latency was measured using the ping utility by sending an echo request per

second. The results, depicted in figure 6.5 are coherent with the ones performed on the throughput, and hence the same conclusions can be made.

6.2.2 Host firewall setup

These tests were performed to analyze the performance of the eBPF firewall in the more common situation in which Iptables is used to protect local applications, by configuring policies on the input chain.

In this configuration for UDP and TCP throughput, the `iperf3` benchmarking suite was used. Using this tool, one machine acts as a client and sends traffic to the server machine through one direct Ethernet link, as in figure 6.6.

To further stress the DUT and have meaningful results, the processor interrupt management was tweaked. To do so, the interrupts generated by the network cards were all directed to a single core, while iperf3 was assigned to the other seven cores.

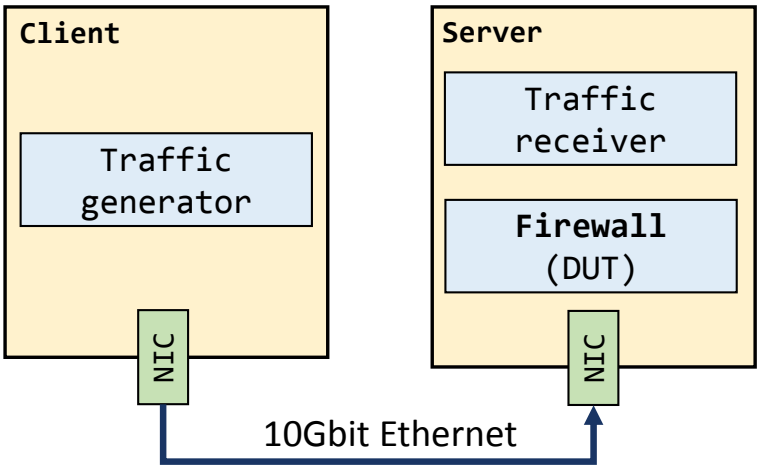

Figure 6.6: Setup used for benchmarking the input chain.

UDP Throughput

To perform these benchmarks, on the server machine iperf3 was run through `taskset` to set the processors affinity, using the command

```
taskset 0x000000FE iperf3 -s 10.0.0.3
```

and on the client machine, iperf was run using the command

```
ip netns exec nsclient iperf3 -c 10.0.0.3 --bandwidth 0 --omit 10
    --time 300 --udp --interval 60 -P 8
```

where `bandwidth 0` does not limit the bandwidth, `omit 10` does not log the results for the first 10 seconds, `time 300 interval 60` runs the test for five minutes and logs the average of the results every minute, `P 8` launches 8 separate connections to fully stress the DUT. The segment size using UDP is not customizable, hence it was the default Ethernet one.

The results of the UDP throughput tests are shown in figure 6.7. It can be noticed how, in this condition, the linearity of the algorithm is almost negligible, with a strong gain over iptables on an high number of rules. An explanation can be that by focusing the computation on a single core, the overhead of the other Linux networking components is much higher than the one of the eBPF firewall, making it almost negligible.

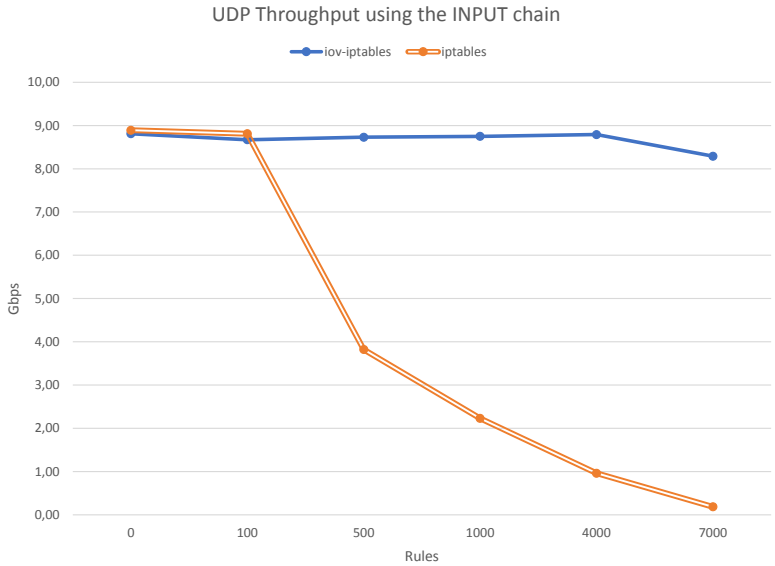

Figure 6.7: Input chain UDP Throughput.

TCP Throughput

To perform these tests, on the server machine iperf3 was ran through `taskset` to set the processors affinity, using the command

```
taskset 0x000000FE iperf3 -s 10.0.0.3
```

and on the client machine, iperf was run using the command

```
ip netns exec nsclient iperf3 -c 10.0.0.3 --bandwidth 0 --omit 10
   --time 300 --set-mss 88 --interval 60 -P 8
```

where `bandwidth 0` does not limit the bandwidth, `omit 10` does not log the results for the first 10 seconds, `time 300 interval 60` runs the test for five minutes and logs the average of the results every minute, `P 8` launches 8 separate connections and the segment size was reduced to `88` as it was useful to stress the DUT.

These benchmarks were performed both with the segmentation offload enabled (figure 6.8) and disabled (figure 6.9), the first scenario is more realistic and the second is more stressing for the DUT. In both cases, the eBPF implementation outperforms Iptables. In the first scenario, as for UDP, the overhead introduced by the eBPF firewall is negligible and hence the linearity of the algorithm has almost no impact.

ICMP Latency

Latency was measured using the ping utility by sending an echo request per second twenty times. From figure 6.10 it can be noticed that the tool was not able to stress the DUT enough to provide clear results, hence both Iptables and the eBPF firewall are close, with the latter slightly outperforming the former.

6.2.3 Rule insertion

Rule insertion time reflects the reactivity of the firewall to the changes made to its configuration.

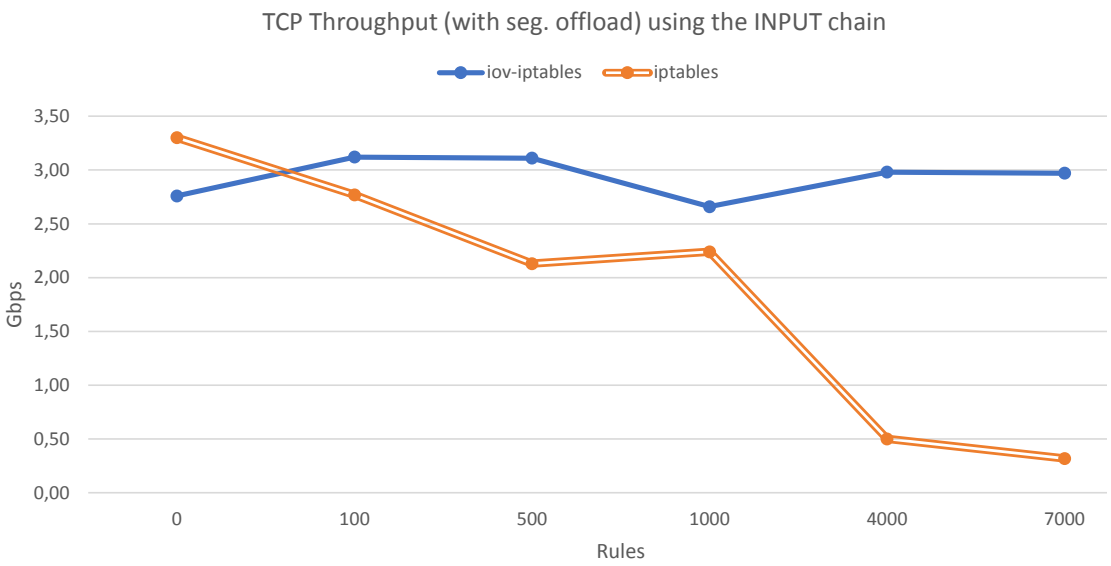

Figure 6.8: Input chain TCP Throughput with segmentation offload enabled.

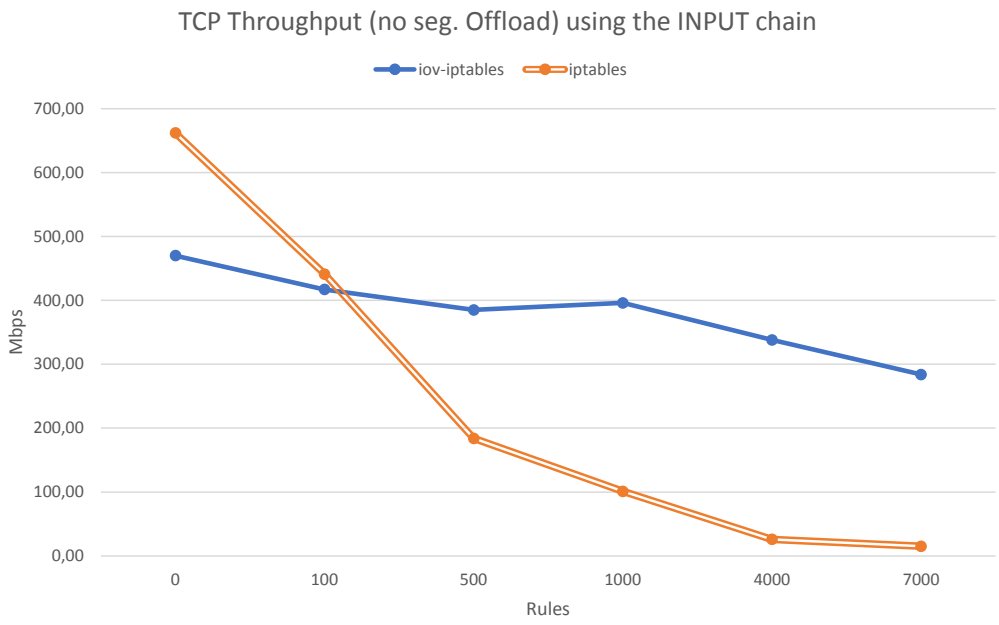

Figure 6.9: Input chain TCP Throughput with segmentation offload disabled.

Figures 6.11 and 6.12 present the insertion time with a variable number of rules.

In this aspect, Iptables outperforms the eBPF firewall. The dominant problems are two: first, the LBVS requires a complex pre-processing, as presented in section 4.2; second, the chain of programs is rebuilt and injected every rule update,

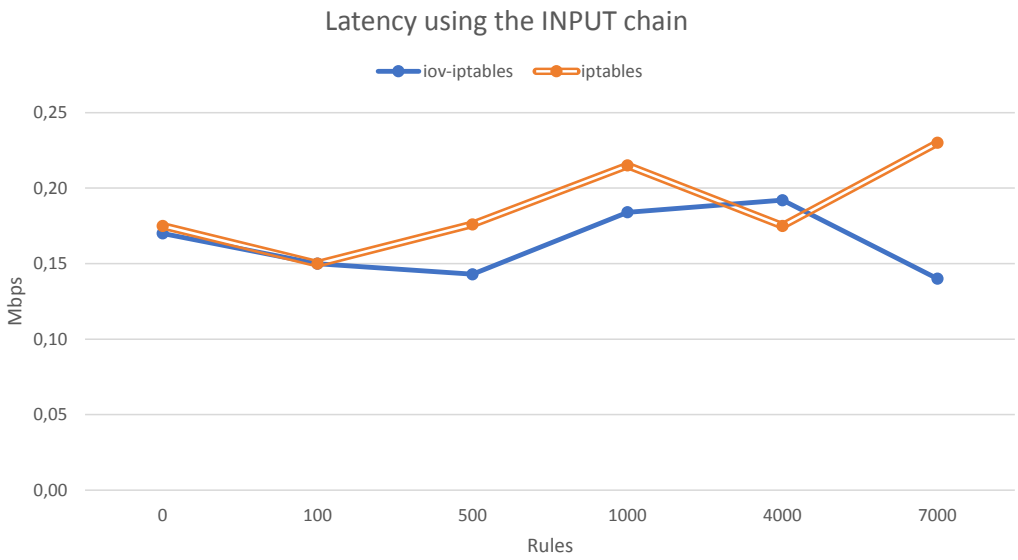

Figure 6.10: Latency measured using the Ping tool.

resulting in a high overhead due to the compilation and injection time of the programs.

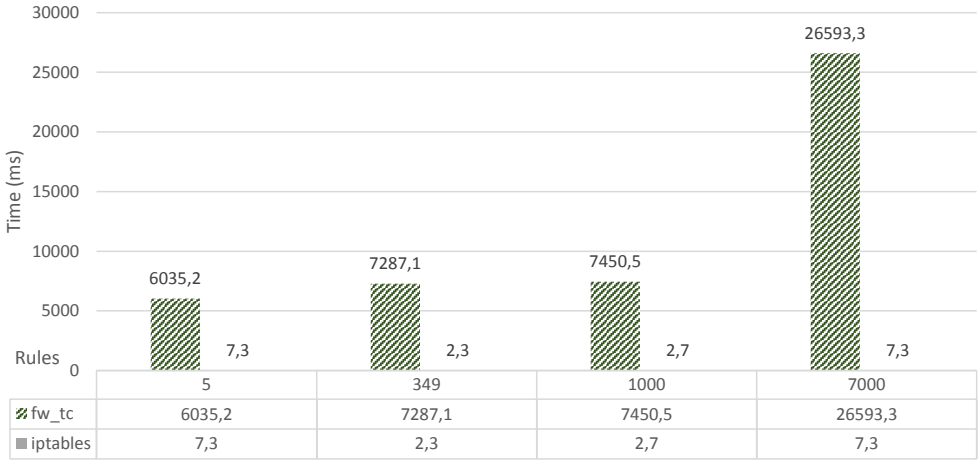

Figure 6.11: Time to update the rule set by inserting a rule in the tail.

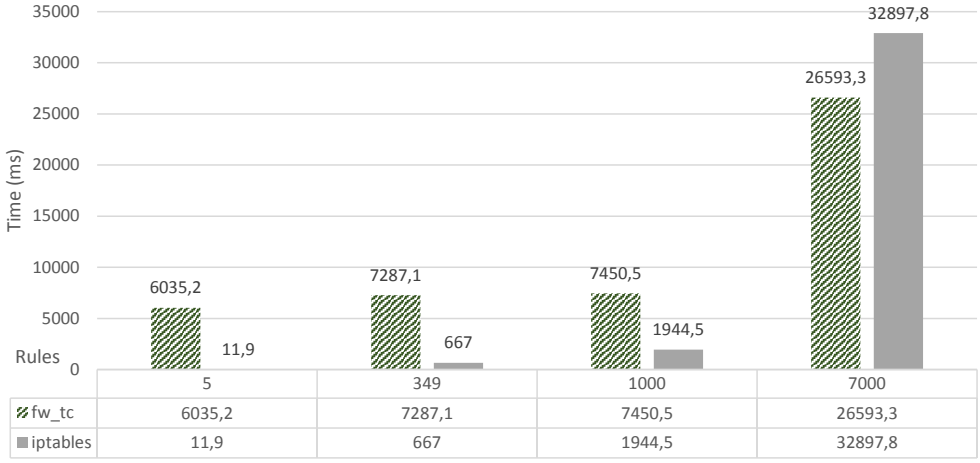

Figure 6.12: Time to update the rule set by inserting a batch of rules.

Chapter 7

Conclusions

7.1 Future work

The architecture modularity opens a wide range of possible future improvements. These may be from the performance or from the features point of view.

Starting from the performance, it's particularly interesting to explore possible optimizations to the algorithm. For example, the LBVS may be combined with a tree properly design to be implemented in eBPF. A schema of the idea is in figure 7.1 and would allow the algorithm to be split in a subset of rules, decreasing its linear complexity even more. To avoid the use of loops, this would require to teach the control plane how to *unroll* the tree exploration every time the rule set is updated.

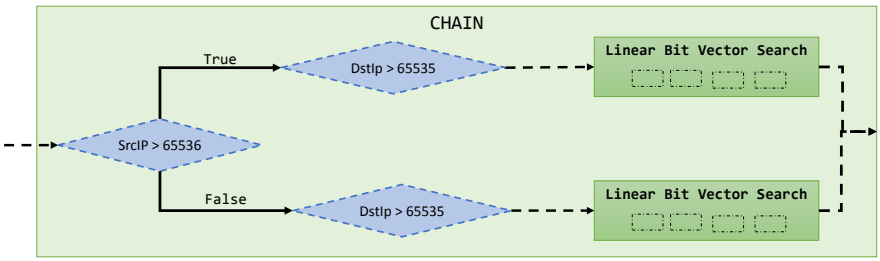

Figure 7.1: A concept on how to improve the LBVS using a tree.

Other two optimizations on the algorithm may be the introduction of a caching mechanism, to avoid executing the LBVS if the matched rule is stored in a cache

(and this opens questions on how to be sure that there is no rule at higher priority matching each packet) and the tailoring of the code based on the traffic pattern, for example by reordering the rules at runtime.

Moreover, a critical aspect is the rule injection time. At the moment, no caching of the preprocessing is performed, as the literature suggests that this is the most common solution. However, patterns may be found that allow re-using part of the already computed data and speed-up the change in configuration.

From the features point of view, it may be interesting to explore the added complexity to support multiple chains other than the *canonical* ones and allow the user to configure rules to jump between them, as in Iptables.

To fully reproduce Iptables behavior, it can be interesting to extend the eBPF implementation to other features Iptables provide, in particular, the NAT. As for the firewall, the NAT rules are based on the Netfilter hook points, raising similar challenges to the ones risen by the firewall. Iptables allows the user to configure *source-NAT* and *destination-NAT*, to change a single address or a pool of addresses into a different one. Moreover, it allows *masquerading*, to statically change the source address of a packet with one of the outbound interfaces.

Given the architecture modularity, a possible implementation of the NAT would be to add more programs in the chain before and after the filtering to enforce the new functionalities. Such an implementation would leverage on the already existing mechanisms to guess Linux routing decision and would allow a proper emulation. The group is already working on a prototype in this direction.

7.2 Final remarks

This work started by presenting the main limits of the currently most used Linux firewall, Iptables, and described the main challenges and solutions in developing a replacement that could overcome its limit by using eBPF.

To preserve Iptables semantic and syntax, in order for the users to do a migration in a transparent way to the new system, and to use eBPF, in order to offer greater

performance, the entire architecture was designed to be modular and leverage on chains of programs each one executing a piece of the overall task.

In the end, the developed solution was validated and compared to its *competitor*. Even if it still presents a number of constraints, like the limited features of the connection tracking or the limited number of field supported, it is a concrete proof that the technology limits can be overcome and the technology advantages can be used to obtain a fast and working replacement.

The great result is that showing to the Linux community this work can prompt a collaboration to improve it and eventually replace the Netfilter, moving to eBPF the Linux networking architecture.

Appendix A

Firewall data model

This chapters lists the data model of the eBPF firewall.

```
    caption
module firewall {
        yang-version 1.1;
        prefix "firewall";

        import base-iovnet-service-model { prefix "basemodel"; }

        organization "Politecnico di Torino";
        description "Data model for the IOVisor Firewall service";

        uses "basemodel:base-yang-module";

        extension cli-example {
                argument "value";
                description "A sample value used by the CLI generator";
        }

        typedef action {
                type enumeration {
                        enum DROP;
                        enum LOG;
                        enum FORWARD;
                }
                default DROP;
        }

        typedef conntrackstatus {
                type enumeration {
                        enum NEW;
                        enum ESTABLISHED;
                        enum RELATED;
                        enum INVALID;
                }
        }


        grouping rule-fields {
                leaf src {
                        type string;
```

```
                description "Source IP Address.";
                config false;
                firewall:cli-example "10.0.0.1/24";
        }

        leaf dst {
                type string;
                description "Destination IP Address.";
                config false;
                firewall:cli-example "10.0.0.2/24";
        }

        leaf l4proto {
                type string;
                config false;
                description "Level 4 Protocol.";
        }

        leaf sport {
                type uint16;
                config false;
                description "Source L4 Port";
        }

        leaf dport {
                type uint16;
                config false;
                description "Destination L4 Port";
        }

        leaf tcpflags {
                type string;
                config false;
                description "TCP flags. Allowed values: SYN, FIN, ACK, RST, PSH, URG, CWR, ECE. ! means set to
                        0.";
                firewall:cli-example "!FIN,SYN,!RST,!ACK";
        }

        leaf conntrack {
                type conntrackstatus;
                config false;
                description "Connection status (NEW, ESTABLISHED, RELATED, INVALID)";
        }


        leaf action {
                type action;
                config false;
                description "Action if the rule matches. Default is DROP.";
                firewall:cli-example "DROP, FORWARD, LOG";
        }

        leaf description {
                type string;
                config false;
                description "Description of the rule.";
                firewall:cli-example "This rule blocks incoming SSH connections.";
        }
}

leaf ingress-port {
        type string;
        description "Name for the ingress port, from which arrives traffic processed by INGRESS chain (by default it's
                the first port of the iomodule)";
}
```

```
leaf egress-port {
        type string;
        description "Name for the egress port, from which arrives traffic processed by EGRESS chain (by default it's
             the second port of the iomodule)";
}

leaf conntrack {
        type enumeration {
                enum ON;
                enum OFF;
        }
        description "Enables the Connection Tracking module. Mandatory if connection tracking rules are needed.
             Default is ON.";
}

leaf accept-established {
        type enumeration {
                enum ON;
                enum OFF;
        }
        description "If Connection Tracking is enabled, all packets belonging to ESTABLISHED connections will be
             forwarded automatically. Default is ON.";
}

leaf interactive {
        type boolean;
        description "Interactive mode applies new rules immediately; if 'false', the command 'apply-rules' has to be
             used to apply all the rules at once. Default is TRUE.";
default true;
}

list session-table {
        key "src dst l4proto sport dport";
        config false;
        leaf src {
                type string;
                config false;
                description "Source IP";
        }

        leaf dst {
                type string;
                config false;
                description "Destination IP";
        }

        leaf l4proto {
                type string;
                config false;
                description "Level 4 Protocol.";
        }


        leaf sport {
                type uint16;
                description "Source Port";
                config false;
        }

        leaf dport {
                type uint16;
                description "Destination";
                config false;
        }

        leaf state {
```

```
                type string;
                config false;
                description "Connection state.";
        }

        leaf eta {
                type uint32;
                config false;
                description "Last packet matching the connection";
        }
}

list chain {
        key name;

        leaf name {
                type enumeration {
                        enum INGRESS;
                        enum EGRESS;
                        enum INVALID;
                }
                description "Chain in which the rule will be inserted. Default: INGRESS.";
                firewall:cli-example "INGRESS, EGRESS.";
        }

        leaf default {
                type action;
                description "Default action if no rule matches in the ingress chain. Default is DROP.";
                firewall:cli-example "DROP, FORWARD, LOG";
        }

        list stats {
                key "id";
                config false;
                leaf id {
                        type uint32;
                        config false;
                        description "Rule Identifier";
                }

                leaf pkts {
                        type uint64;
                        description "Number of packets matching the rule";
                        config false;
                }

                leaf bytes {
                        type uint64;
                        description "Number of bytes matching the rule";
                        config false;
                }

                uses "firewall:rule-fields";
        }

        list rule {
                key "id";
                leaf id {
                        type uint32;
                        description "Rule Identifier";
                }

                uses "firewall:rule-fields";
        }

        action append {
```

```
                input {
                        uses "firewall:rule-fields";
                }
                output {
                        leaf id {
                                type uint32;
                        }
                }
        }

        action reset-counters{
                description "Reset the counters to 0 for the chain.";
                output{
                        leaf result{
                                type boolean;
                                description "True if the operation is successful";
                        }
                }
        }

        action apply-rules{
                description "Applies the rules when in batch mode (interactive==false)";
                output{
                        leaf result{
                                type boolean;
                                description "True if the operation is successful";
                        }
                }
        }

    }
}
```

Bibliography

[1] D. Borkmann. (Feb. 2018). Net: Add bpfilter, [Online]. Available: `https://lwn.net/Articles/747504/`.

[2] A. Lioy. (2017). Firewall and ids/ips, [Online]. Available: `http://security.polito.it/~lioy/02krq/firewall_en_6x.pdf`.

[3] M. Bertrone, S. Miano, J. Pi, F. Risso, and M. Tumolo, "Toward an ebpf-based clone of iptables", *Netdev'18*, 2018.

[4] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating linux security with ebpf iptables", *SIGCOMM'18*, 2018.

[5] B. Gregg, *Linux bpf superpowers*, Available at `http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html` (2018/21/07).

[6] *Ebpf: Next generation of programmable datapath*, Available at `http://www.openvswitch.org//support/ovscon2016/7/1030-graf.pdf` (2018/21/07).

[7] M. Fleming, *A thorough introduction to ebpf*, Available at `https://lwn.net/Articles/740157/` (2018/07/17).

[8] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned",

[9] *Bpf compiler collection (bcc) [on GitHub]*, Available at `https://github.com/iovisor/bcc/` (2018/21/07).

[10] J. Wallen. (2015). An introduction to uncomplicated firewall (ufw), [Online]. Available: `https://www.linux.com/learn/introduction-uncomplicated-firewall-ufw`.

[11] T. Graf, *Why is the kernel community replacing iptables with bpf?*, Available at `https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/` (2018/21/07).

[12] H. V. Styn, *Advanced firewall configurations with ipset*, Available at `https://www.linuxjournal.com/content/advanced-firewall-configurations-ipset` (2018/21/07).

[13] P. Sutter, *Benchmarking nftables*, Available at `https://developers.redhat.com/blog/2017/04/11/benchmarking-nftables/` (2018/21/07).

[14] F. Westphal. (Mar. 2018). Bpfilter: Add experimental imr bpf translator, [Online]. Available: `https://www.spinics.net/lists/netdev/msg486873.html`.

[15] P. N. Ayuso. (Feb. 2018). Nftables meets bpf, [Online]. Available: `https://www.mail-archive.com/netdev@vger.kernel.org/msg217425.html`.

[16] G. Varghese, *Network algorithmics: An interdisciplinary approach to designing fast networked devices*. San Francisco, CA, USA: Morgan Kaufmann, 2005.

[17] K. Karimi, A. Ahmadi, M. Ahmadi, and B. Bahrambeig, "Acceleration of iptables linux packet filtering using gpgpu",

[18] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching", *SIGCOMM CCR*, no. 2, pp. 203–214, 1998.

[19] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark", *IEEE/ACM Transactions on Networking*, no. 3, pp. 499–511, 2007.

[20] *Bitscan*, Available at `https://chessprogramming.wikispaces.com/BitScan` (2018/05/05).

[21] Cilium, *Bpf and xdp reference guide*, Available at `http://www.iptables.info/en/connection-state.html` (24/10/2017).

[22] *Chapter 7. the state machine*, Available at `http://cilium.readthedocs.io/en/latest/bpf/` (2018/30/04).

[23] H. Song and J. S. Turner, "Toward advocacy-free evaluation of packet classification algorithms", 2011.

[24] R. Thakker and C. Sheth, "Performance evaluation and comparative analysis of network firewalls", 2011.

[25] D. E. Taylor and J. S. . Turner, "Classbench: A packet classification benchmark", 2005.

[26] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to cams?", 2003.

[27] F. Baboescu and G. Varghese, "Scalable packet classification", 2003.

[28] P. Comerford, J. N. D., and V. Grout, "Reducing packet delay through filter merging", 2016.