POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Building high-speed network functions in the Linux kernel

Supervisor
prof. Fulvio Risso

Candidate
Kevin Corizi

Academic year 2017-2018

Contents

Chapter 1

Introduction

The ever-growing need for faster, more flexible cloud applications demands for increasingly complex networking architectures.

When a virtual infrastructure is set up, a virtual networking overlay must be imposed over the physical network. This is especially true for those infrastructures where several components are continuously deployed: the ability to connect them without changing the physical configuration allows unprecedented scalability and flexibility.

This is the realm of network function virtualization, which is used to build arbitrary networking configurations on top of a physical network. Of course such configurations must be complete and working, and therefore need to provide virtualized versions of the most common networking devices, such as routers, firewalls and NATs.

An emerging network function virtualization framework is Polycube, developed as a research project at Politecnico di Torino. Polycube enables the creation and deployment of arbitrary lightweight and fast network functions, which run in the Linux kernel and can be used to build complex service chains. This technology is based on the eBPF virtual machine.

eBPF is the evolution of the Berkeley Packet Filter (BPF), which is the engine that powers tools such as Wireshark. With eBPF it is possible to inject code in the Linux kernel at runtime, which is verified for safety and compiled before execution. eBPF programs, which are called *cubes* in the context of Polycube, can intercept, access, modify and redirect packets. eBPF allows *cubes* to communicate with each other and share memory areas, called *maps*.

To prove how powerful and complex a cube can be, a proof-of-concept version of *iptables* was implemented using the eBPF technology in Polycube, which is called *pcn-iptables*: this piece of software consists of several sub modules, linked together

to provide firewalling, packet filtering and connection tracking functionalities. An adapter was also created that allows to configure pcn-iptables with the same syntax as the original one.

The first part of this thesis project focuses on integrating the network address translation function in pcn-iptables, to get closer to the actual iptables functionality.

Polycube also provides a set of services out of the box, such as routers, switches and network address translators: the current version of the NAT cube, called *pcn-nat*, has several limitations that do not make it ideal to use in practical applications.

The second part of this thesis consists of the definition and implementation of a new version of *pcn-nat*, that could make it suitable for real-life deployment and usage.

Chapter 2

Background

2.1 Network Address Translation

Network Address Translation (NAT) - described in RFC 2663 [1] - is a method of remapping an IP address space into another one by modifying the network address information in the IP header of packets while they are in transit across a network routing device.

The technique was originally used to connect private address spaces to the public Internet. A useful side effect was that there was no need to reconfigure the IP addresses on each host when a network was moved to a new location. Nowadays it has become a popular and essential technique in conserving the global address space in the face of the IPv4 address space exhaustion.

In the following, a description of the different ways we can implement NAT is provided.

We will refer to *inside addresses* and *outside addresses* as the addresses to be translated and those with which to translate, respectively.

2.1.1 Basic address translation - NAT

Basic NAT [2] provides a one-to-one mapping of an inside IP address to an outside IP address. For each inside IP address, there is an outside address mapped to it.

This means that to perform Basic NAT all the outside addresses must be owned and routable.

In Basic NAT only Layer 3 properties are changed, thus leaving Layer 4 port numbers unchanged. This happens because there is no address mapping overload since the association between inside and outside address is N:N, with N being at least equal to one.

A simple scenario where Basic NAT is used is shown in Figure 2.1.

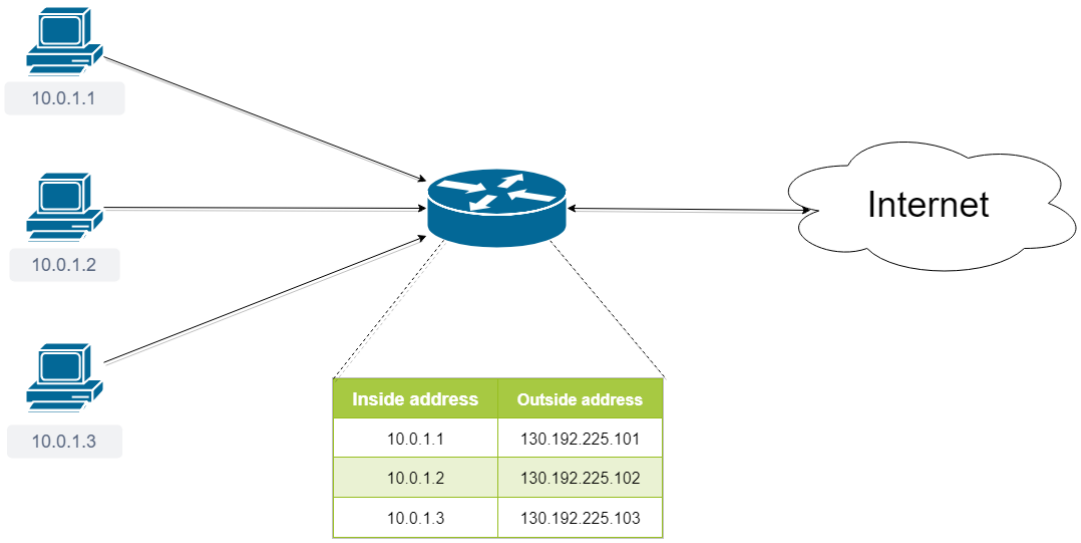


Figure 2.1: Example of Basic NAT configuration

2.1.2 Address and port translation - NAPT

Network Address Port Translation [3] provides a many-to-one - theoretically many-to-many - mapping of inside addresses on outside addresses.

Since the same outside address will be used to map more than one address or possibly an entire network, other fields are needed to keep track of the association; in this case, Layer 4 port numbers are used.

NAPT can be combined with Basic NAT to provide N:M address mapping, using more than one outside address to multiplex the possible values, perhaps in round-robin fashion to balance the usage of addresses and port numbers.

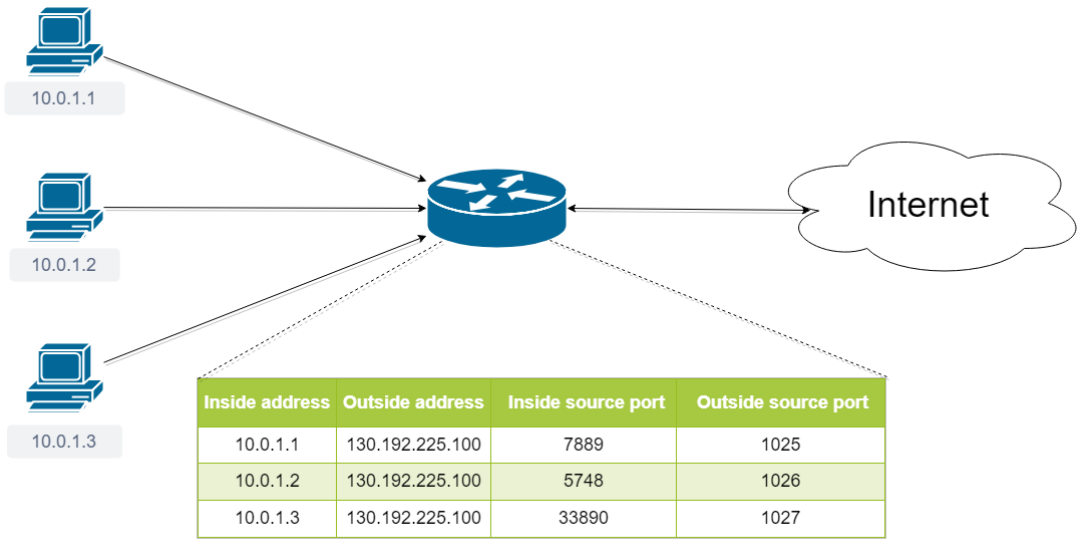
An example of NAPT is shown in Figure 2.2.

2.1.3 Bidirectional NAT

Neither Basic NAT nor NAPT allow inside hosts to be reachable from the outside, although from an implementation point of view this is possible if a connection - resulting in the creation of a mapping record - was started by the inside host itself.

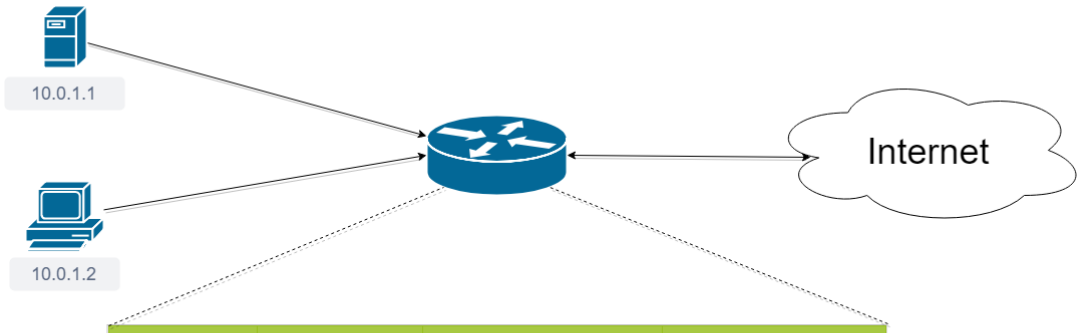
This may be problematic for inside servers (web, email...) that must be reachable from the Internet at all times.

Bi-directional NAT [4] provides a solution to such problem, allowing to map an outside address - and port, e.g. port 80 - to a specific host in the inside network. Figure 2.3 show a possible use case for Bidirectional NAT.



Inside address	Outside address	Inside source port	Outside source port
10.0.1.1	130.192.225.100	7889	1025
10.0.1.2	130.192.225.100	5748	1026
10.0.1.3	130.192.225.100	33890	1027

Figure 2.2: Example of NATP configuration



Inside address	Outside address	Outside destination port	Inside destination port
10.0.1.1	130.192.225.100	80	8080
10.0.1.2	130.192.225.100	22	22

Figure 2.3: Example of Bidirectional NAT configuration

2.2 Netfilter

Netfilter [5] [6] is a framework provided by the Linux kernel that allows the implementation of networking operations such as packet filtering, network address translation, routing and firewalling.

This framework is based on the concept of *hook*, which is a point of interception for a certain event. Developers can write *callback functions* and register them so that they are called every time a hook is triggered to perform operations on the packet that is traversing the hook.

Specifically, Netfilter provides five hooks: every packet that passes through the networking system will trigger one or more of these hooks, allowing the callback functions to interact with the traffic at specified points.

- NF_IP_PRE_ROUTING: triggered by an incoming packet as soon as it enters the networking stack, before any routing decision has been taken; for instance, the Linux networking stack does not know yet whether the packet will be consumed locally (i.e., it is directed to one of the IP addresses of the host) or will be forwarded to another host.

- NF_IP_LOCAL_IN: triggered by an incoming packet if the destination is the local system.

- NF_IP_FORWARD: triggered by an incoming packet if the destination is another host.

- NF_IP_LOCAL_OUT: triggered by a locally generated outbound packet as soon as it enters the networking stack.

- NF_IP_POST_ROUTING: triggered by an outgoing or forwarded packet just before transmission (before being delivered to the NIC driver).

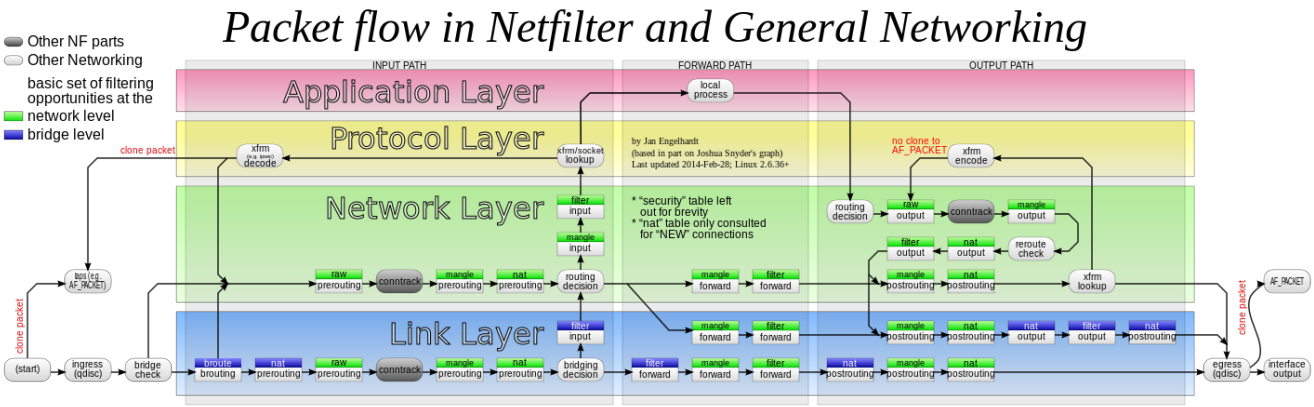


Figure 2.4: Netfilter packet interception hooks

2.3 Iptables

iptables [7] is a packet manipulation software included in Linux distributions: it works by interacting with the packet filtering hooks in the Linux kernel provided by Netfilter.

2.3.1 iptables architecture

Iptables provides a table-based system to define rules and actions to interact with the traffic flow.

Tables

Rules are stored in tables. Each table contains the necessary rules to perform a given operation: for instance, if a rule deals with network address translation, it will be put in the *nat* table.

Chains

Within each table, rules are organized in *chains*. The built-in chains represent the netfilter hooks they are associated with:

- PREROUTING: triggered by the NF_IP_PRE_ROUTING hook
- INPUT: triggered by the NF_IP_LOCAL_IN hook
- FORWARD: triggered by the NF_IP_FORWARD hook
- OUTPUT: triggered by the NF_IP_LOCAL_OUT hook
- POSTROUTING: triggered by the NF_IP_POST_ROUTING hook

Chains allow to control when a rule will be evaluated during the packet's path in the networking stack, and since each table can have multiple chains, the table's policy can be enforced at multiple steps of processing.

2.3.2 NAT in iptables

Network Address Translation in iptables [8] is managed by the NAT table. As packets enter the network stack, rules in the NAT table will determine if and how a packet has to be modified.

Supported chains

The iptables implementation of NAT allows to specify NAT rules that can be enforced in three different chains:

- PREROUTING
- POSTROUTING
- OUTPUT

which corresponds to three precise positions in the Netfilter architecture.

Note that in order to provide a working bi-directional translation specular operations have to be performed in the POSTROUTING/OUTPUT chains and the PREROUTING chain, as the former will deal with outgoing packets, the latter with incoming packets.

For example, a command to inject a translation rule to map an inside address to an outside address must also implicitly provide a way to map the outside address back to the inside address.

Iptables allows the configuration of four types of NAT, which are formally associated to different NAT *actions* that transform the packets that match a given NAT rule.

An action is an operation specified by a rule. It can be performed on a packet when it triggers the hook related to the chain the rule is associated to.

When the chain is hit, iptables performs a lookup in the NAT table to check if any rule of that chain applies for the packet.

Figure 2.5 shows how iptables handles incoming and outgoing packets: please note that only the filtering and natting operations are displayed, for the sake of brevity.

Source NAT - SNAT

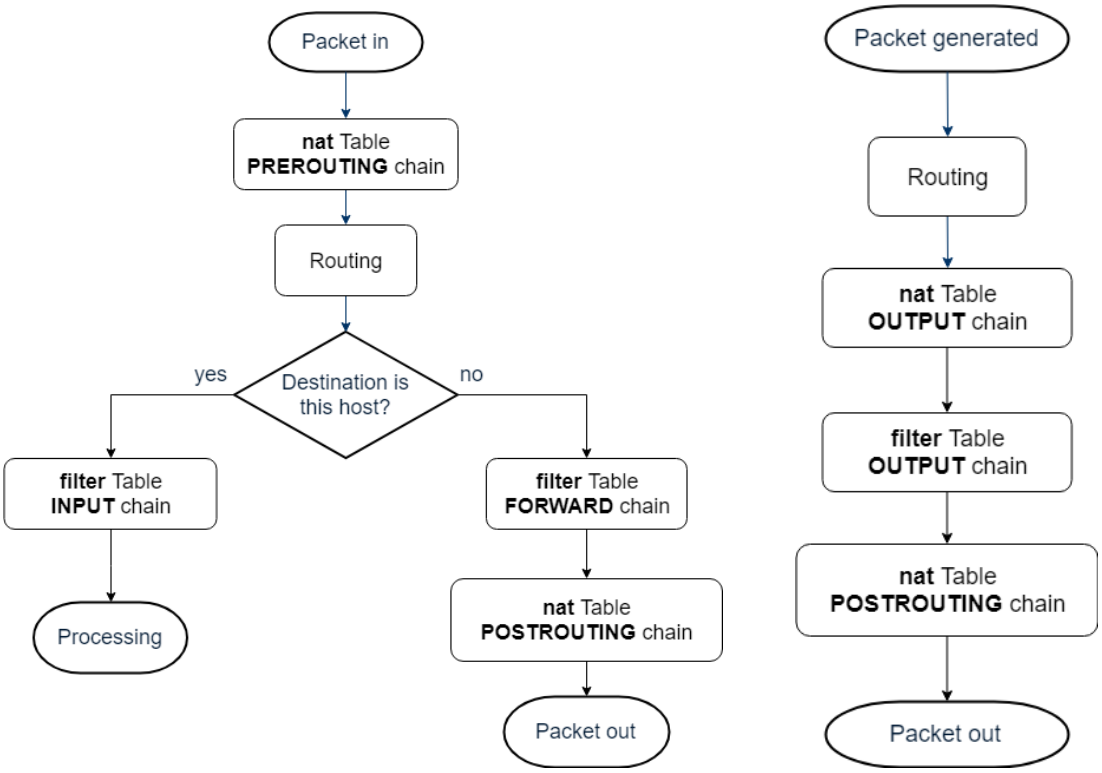
Source NAT (SNAT) allows to statically change the source address of a packet with a specified one. It corresponds to the Basic NAT described in Section 2.1.1, applied to the source IP address when packets exit toward a remote destination.

Some configuration examples for Source NAT in iptables are shown in Appendix A.1.

Masquerade

Masquerade allows to change the source address of a packet with the address of the outbound interface.

The advantage over SNAT is that if the interface address is dynamically assigned it is automatically inherited by all packets.



(a) Incoming packets (b) Outgoing packets

Figure 2.5: Iptables activities for incoming and outgoing packets

A possible disadvantage is that masquerade retrieves the current IP address of the interface each time a new packet has to be handled, resulting in a possibly noticeable impact in performance. If the interface address is fixed, SNAT is a better choice.

Some configuration examples for Masquerade in iptables are shown in Appendix A.2.

Destination NAT - DNAT

Destination NAT (DNAT) allows to statically change the destination IP address of a packet that matches a NAT rule. It corresponds to the bi-directional NAT presented in Section 2.1.3.

DNAT is mostly useful for servers running in the inside network that must be reachable from the outer network. The destination address must be changed before routing happens, therefore this is applied in the PREROUTING chain.

Some configuration examples for Destination NAT in iptables are shown in Appendix A.3.

Redirect and change port

Redirect is a special case of DNAT where packets are redirected to a local TCP/UDP port of the router behind the NAT. If the router is the recipient of the packet on a certain port, with this action we can change the destination port with another one.

Some configuration examples for Redirect in iptables are shown in Appendix A.4.

2.4 eBPF

2.4.1 BPF

BPF is a virtual CPU embedded in the Linux kernel that can filter packets. Introduced in the Linux kernel 2.1.75 in 1997, it was initially used as a packet filter by the packet capture tool *tcpdump*. Being in the Linux kernel, BPF does not suffer from syscall and context switching overheads: this means the filtering process can start as soon as the packet reaches the interface, which is very important for DDOS mitigation usages and for performance in general.

One of the reasons why BPF can work is the existence of a *network tap* that takes incoming packets and copies them: this means one copy goes to the TCP/IP stack, and a copy to the filter. The network TAP is a bifurcation with duplication. All operating systems have the network TAP built in: the work described in this thesis applies to Linux.

The general architecture of a packet filter, included BPF, is shown in Figure 2.6.

Some of the main features of BPF programs are:

- runtime bytecode injection: programs can be written, compiled and injected in the kernel at any time, possibly changing the behaviour of existing BPF programs according to new conditions
- safety: BPF programs injected in the Linux kernel are safe, which means that a program cannot take control of the CPU or access unauthorized memory areas. The BPF compiler uses a validator that checks, among other things
  - the existence of (possibly) infinite loops
  - invalid memory accesses
  - program size
  - number of instructions: since loops are unrolled, the number of instructions can dramatically increase if the number of iterations is high
- portability: BPF compiled bytecode is hardware independent and can be executed on any architecture

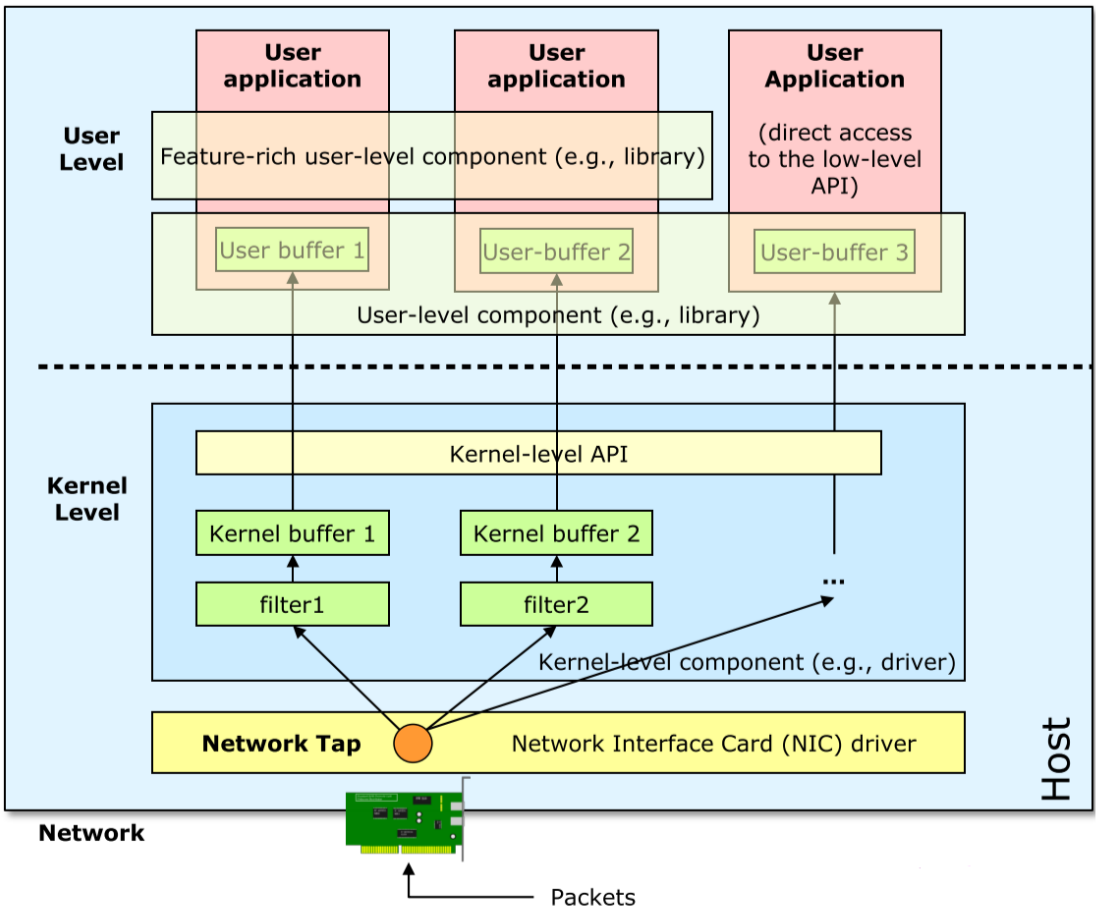


Figure 2.6: Architecture of a packet filter

- efficiency: the BPF runtime consumes very few resources and it is very close to kernel events. The JIT compiler optimizes the code immediately before compiling.

2.4.2 eBPF

Initially proposed by Alexei Staravoitov in 2013, eBPF [9][10][11] is the next version of BPF, which includes both modifications to the underlying virtual CPU (64-bit registers, additional instructions) and to the possible usages of BPF in software products. "Classic" BPF is not used anymore, and legacy applications are adapted from the BPF bytecode to the eBPF.

An overview of the runtime achitecture of eBPF is reported in Figure 2.7.

Let us explain some of the relevant parts of the architecture and point out some of the main improvements in eBPF.

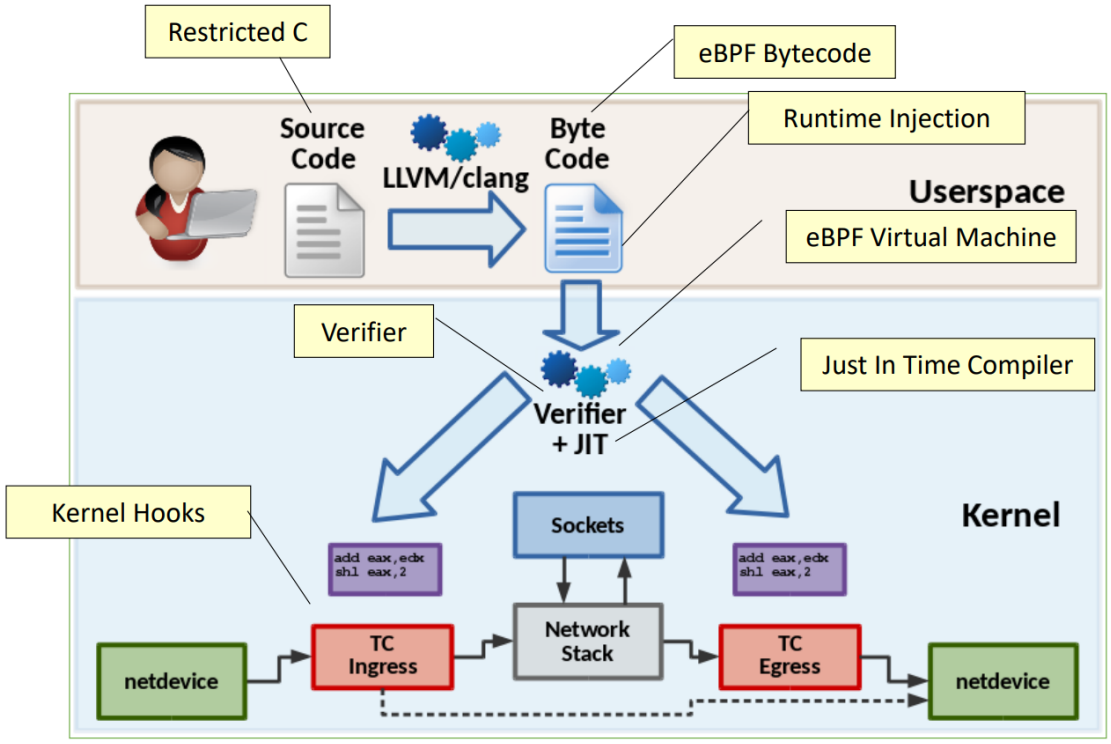


Figure 2.7: eBPF architecture

C-based programming

eBPF code can be written in (a restricted version of) C, which allows for easier program development and more powerful functionalities with respect to bare assembly.

Maps

An eBPF program is triggered by a packet received by the virtual CPU. But how do we store the packet in order to process it? BPF defines a volatile "packet memory", which can only store the current packet: this means there is no way to store information needed across subsequent packets.

eBPF defines the concept of state with a set of memory areas, which are called *maps* [12]. Maps are data structures where the user can store arbitrary data with a key-value approach: data can be inserted in a map by providing the value and a key that will be used to reference it.

An important feature of maps is that they can be shared between eBPF programs and between eBPF and user-space programs. This is so important for all those applications that need to perform operations that exceed the complexity allowed

by the eBPF bytecode, because it allows to split complex processing in fast eBPF datapaths and slow user space control paths keeping the state information shared and synced. Another important advantage of using maps is that their content is preserved across program executions.

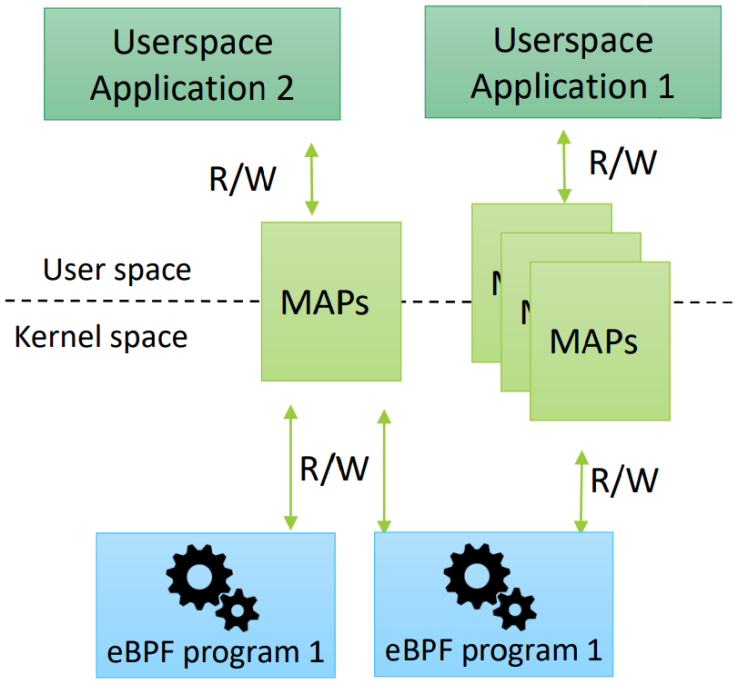


Figure 2.8: eBPF maps

Maps are not technically buffers. If they were, there would be a certain number of issues, such as concurrent access. This means maps are never accessed directly: we read and write maps with predefined system calls. An important side effect of using maps is that the state of the program is decoupled from the code. Instructions are in the program, the data used by such instructions are in the maps.

Here are reported some of the map types available in the Linux kernel and used in the implementation of this thesis project [13].

- array: data is stored sequentially and can be accessed with a numeric index from 0 to size - 1. This map type is ideal when the key corresponds to the position of the value in the array.
- hash: data is stored in a hash table. This map type is very efficient when it comes to direct lookups: a hash of the provided key is computed and used as an index to access the corresponding value.

- LRU hash: sharing its internals with the hash type, it provides the ability to have a hash-table which is smaller than the total elements that will be added to it, because when it runs out of space it purges elements which haven't recently been used. This map type is useful to keep track of caching entries, which will eventually expire forcing their own refresh.
- LPM trie: data is stored as an ordered search tree. As the name suggests, this map type allows to perform lookups based on the Longest Prefix Match algorithm. This is extremely handy when we want to manage the granularity of a rule match field, granting that the most fitting rule is applied to a packet when other matching rules are also present.

Hooks

eBPF programs can react to generic kernel events, not only packet reception: they can react to any syscall that exposes a hook.

Considering a network packet, recalling how the netfilter hooks work, with eBPF we can listen to any of the predefined hooks to trigger programs only at certain steps during packet processing. Netfilter is a set of linked modules but it has no filtering concept: attaching to a hook means receiving all the packets. eBPF can attach to hooks and filter packets.

Speaking of hooks, the following Figure 2.9 shows the difference in number and position of the networking hooks in Netfilter and eBPF.

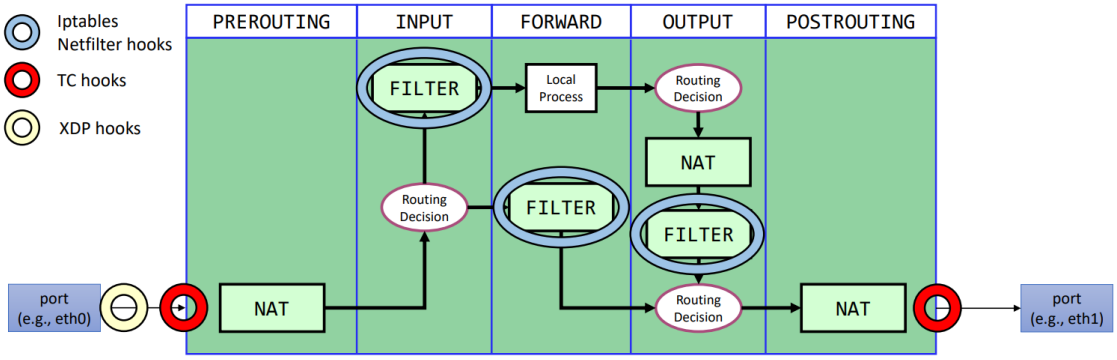


Figure 2.9: Networking hooks in eBPF and Netfilter

eBPF hooks are colored in red: the one on the left is called the *TC Ingress* hook and intercepts all the packets that reach the network adapter from the outside, while the one on the right - the *TC Egress* hook - deals with the outgoing packets immediately before sending them to the network adapter.

It is clear that Netfilter can provide a much more fine-grained control over a flow of packets, whereas the intermediate hooks must be emulated in eBPF. As we will see in the following Chapter, this will be a relevant issue we will have to face.

Service chains

BPF did not quite have the concept of multiple cooperating programs: each parallel program receives a copy of the packet and process it.

eBPF can link multiple programs to build service chains. Service chains can be created exploiting direct virtual links between two eBPF programs or *tail calls*.

Tail calls can be imagined as function calls: the eBPF programs are separated, but the first one triggers the execution of the second by calling it. This allows developers to overcome the program size limitation in the JIT compiler: starting from one big program, we can split it in multiple modules, perhaps functionally distinct. As we will see in the following chapters, tail calls are widely used in pcn-iptables.

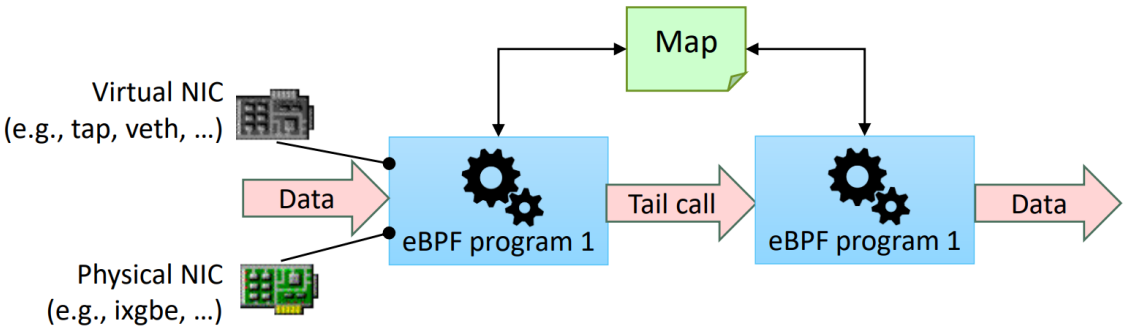


Figure 2.10: Sample eBPF service chain

Helpers

Coding in C is fun, libraries are better. Helpers are sets of functions precompiled and ready to be used inside the Linux kernel. eBPF programs can call such functions, which are outside the virtual CPU (e.g. function to get the current timestamp).

Helpers delegate complex tasks to the operating system, overcoming the complexity restrictions in the eBPF validator and allowing developers to exploit advanced OS functionalities.

One drawback of helpers is that they must be available in the Linux kernel, which means that the kernel must be recompiled every time a new helper is added, which is not very fast considering the Linux kernel release schedule.

2.5 Polycube

Polycube [14] is a framework that enables the creation and deployment of arbitrary lightweight and fast network functions, running in the vanilla Linux kernel. Polycube services - called *cubes* - can be rearranged in complex service chains. The second part of this thesis project consists in the implementation of a new version of the service that provides network address translation in a service chain.

Polycube exploits some existing components such as the BCC compiler collection to dynamically create and inject the code in the kernel, and the eBPF virtual machine to execute it.

Its main features are:

- support generic network services through the definition of a fast and slow path - in kernel and in user space respectively - hence overcoming the limitations of the eBPF in terms of supporting arbitrary processing
- offers a service-agnostic configuration and control interface that allows to interact with all running network services through the same interface and tools, namely both REST and CLI
- enables the creation of arbitrary service chains, hence simplifying the creation of complex services through the composition of many elementary components

2.5.1 Polycube architecture

The Polycube architecture is depicted in Figure 2.11 and it includes four main components:

- *polycubed*: a service-agnostic daemon to control the polycube service
- a set of Polycube *cubes*: services which actually implement the network functions
- *polycubectl*: the CLI that allows to interact with polycube and all the available services
- *libpolycube*: a library that keeps some common code to be reused across multiple network functions

polycubed

polycubed is a service-agnostic daemon that allows to control the entire Polycube service, such as starting, configuring and stopping all the available network functions. This module acts mainly as a proxy: it receives a request from its REST interface, forwards it to the proper service instance, and returns back the answer to the user.

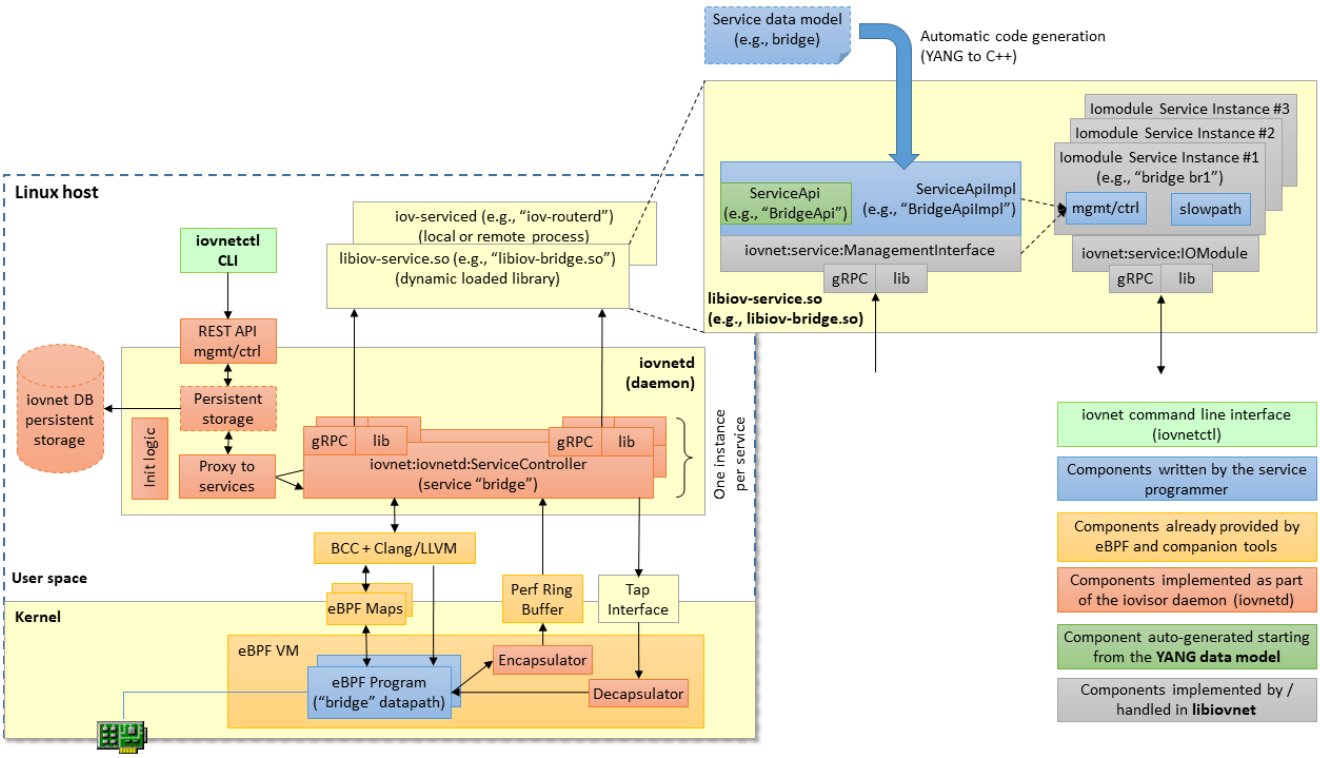


Figure 2.11: Polycube architecture

Polycube services

Polycube implements a flexible architecture that supports multiple services, not known a-priori. Polycube services are similar to plug-ins that can be installed and launched at run-time. Obviously, each service has to implement a specific interface to be recognized and controlled by polycubed.

Each service implementation includes the datapath, namely the eBPF code to be injected in the kernel, the control/management plane, which defines the primitives that allow to configure the service behaviour, and the slow path, which handles packets that cannot be fully processed in the kernel. While the former code runs in the Linux kernel, the latter components are executed in user-space.

polycubectl

This module implements the CLI that allows to control the entire system, such as deploying new instances of a service and configure the existing ones.

This module cannot know which services it will have to control: its internal architecture is able to interact with any service through well-defined interfaces that have to be implemented in each service. Since creating a compatible interface from scratch is definitely not trivial, a code generation tool is provided, that generates

all the necessary boilerplate code starting from a YANG data model describing the service itself.

libpolycube

This library contains some common code that can be reused across multiple network functions: it facilitates the creation of links between services, provides common primitives such as a logging system, allows to access eBPF maps.

2.5.2 pcn-iptables

The *pcn-iptables* service is intended to emulate iptables by using the same semantic but a very different backend, based on eBPF programs and more efficients algorithms and runtime optimizations instead of Netfilter.

pcn-iptables was developed to show not only that eBPF programs can be arbitrarily complex if properly chained, but also that an alternative to iptables is possible: in fact, iptables has been vastly adopted for more than 20 years, which makes system administrator unwilling to trust other technologies.

Among the main functionalities currently implemented in pcn-iptables are forwarding, filtering and connection tracking.

The architecture of pcn-iptables is illustrated in Figure 2.12 and Figure 2.13.

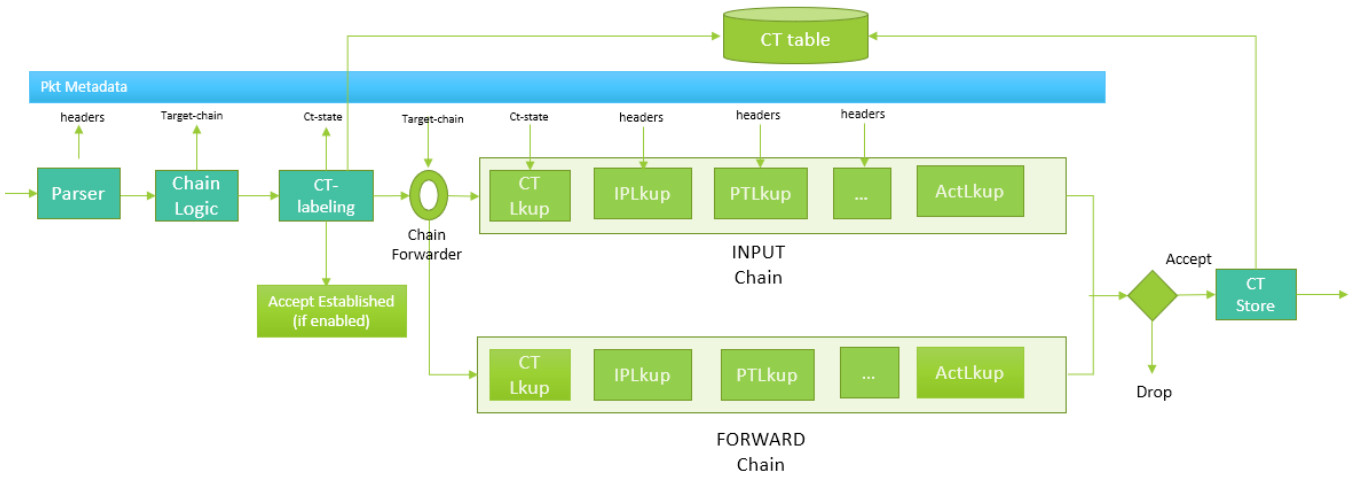


Figure 2.12: Ingress architecture in pcn-iptables

A first module, the `Parser`, reads the packet headers from the raw packet and stores them into eBPF maps for easier handling.

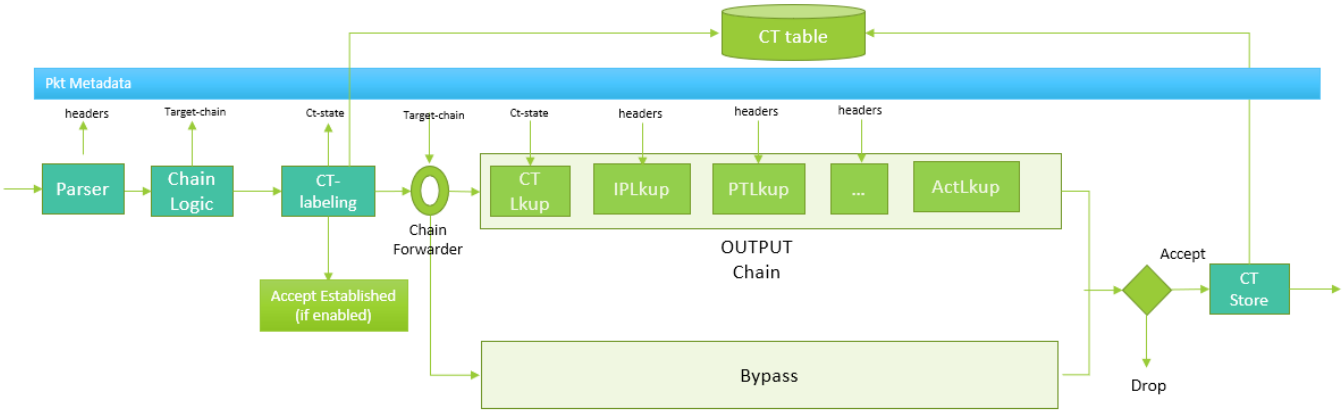


Figure 2.13: Egress architecture in pcn-iptables

Forwarding

As we thoroughly discussed, iptables rules are organized in *chains* triggered by specific Netfilter hooks. We also discussed about how the Netfilter hooks differ from the eBPF hooks both in position and number.

To preserve the forwarding functionality, pcn-iptables must emulate what iptables can do natively, which is to support the INPUT, OUTPUT and FORWARD chains. These chains need a routing decision to be taken in order to be consistently multiplexed, therefore pcn-iptables mimics a routing decision as follows:

- the FORWARD chain is used on the ingress hook if the destination IP does not belong to the receiving interface
- the INPUT chain is used on the ingress hook if the destination IP does not match FORWARD
- the OUTPUT chain is used on the egress hook if the source IP did not match the FORWARD chain

This heuristic routing decision is taken by the `ChainLogic` module and applied by the `ChainForwarder` module.

Filtering

Being iptables a firewalling tool, filtering is most certainly a key feature to be implemented.

The algorithm is conceptually simple, and we will omit the implementation details for sake of brevity: a first sub module, called `Parser`, extracts the raw packet headers and stores them in eBPF maps to make them shared across all sub modules without the need to read the raw packet every time.

The subsequent sub modules - `IPLookup`, `PortLookup`, ... - check individual packet properties, such as IP addresses, Layer 4 port numbers and protocols and TCP flags and update a bit vector which is then used to find a matching rule for the packet, if any.

Connection tracking

Connection tracking in pcn-iptables is based on a finite state machine that defines a set of possible statuses for a connection, such as `NEW`, `ESTABLISHED`, `INVALID`.

This state machine is maintained across program iterations because it is stored in a eBPF map, which is called `connections` and which will refer to as *connection tracking table* as well.

The connection tracking table is a direction-independent data structure that holds one entry for each active connection. The key of the map consists of the packet headers, while the value stores connection information, such as the `ttl` and of course the `state`.

A connection is based on an exchange of packets, which means the state machine in an entry can be correctly updated only if the entry is used for packets in both directions, hence the property of direction-independence. When a packet comes, the connection tracking table is first searched with the current packet headers - *forward* direction - otherwise with the headers reversed - *reverse* direction. The algorithm that manages the state machine performs different operations and checks, depending on the actual direction of the packet, to verify that the connection is evolving correctly - i.e. that the three-way TCP handshake steps are being performed in order.

Connection tracking is handled by two modules: `ConntrackTableUpdate` and `ConntrackLabel`.

Integrating network address translation in pcn-iptables will require several modifications to these last two modules.

2.5.3 pcn-nat

The *pcn-nat* service is intended to provide network address translation as part of a Polycube service chain. It implements a *transparent NAT*, which means that the service itself is never the target of a packet.

This implies that for a working configuration, the NAT service must be preceded by a router in the service chain, which handles ARP requests and replies, ICMP packet generation and packet routing of course. What this service does is to simply forward a packet from the receiving port to the other, changing the packet headers if needed.

The most basic service chain that must be set up to make NAT work is reported in the following picture.

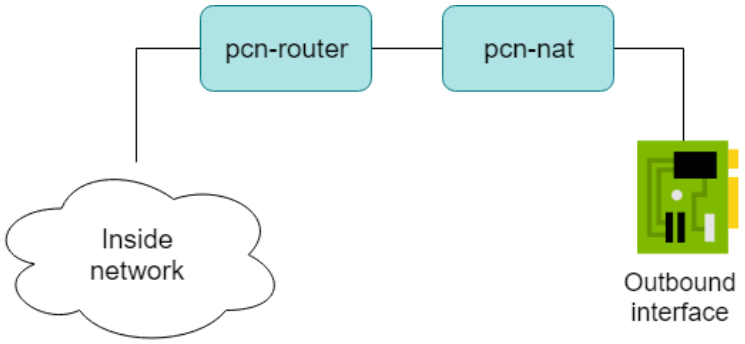


Figure 2.14: Simple Polycube service chain with NAT

Chains

Recall that iptables is based on the netfilter framework, which provides five hooks for packet interception.

Polycube, on the other hand, is based on the eBPF framework, which only provides two hooks: as soon as they get to the network interface (similar to the netfilter NF_IP_PRE_ROUTING hook), and immediately before they leave it (similar to the netfilter NF_IP_POST_ROUTING hook). These two hooks are called *ingress hook* and *egress hook* respectively.

Natting in Polycube can therefore rely on two chains only instead of the three chains available in iptables: PREROUTING and POSTROUTING. These chains are enough to implement NAT as a building block for complex Polycube service chains: rules specify how to translate source addresses in the egress hook and destination addresses in the ingress hook. This is based on the assumption that the NAT instance exists in a chain of services, therefore exposing only two interfaces and one bidirectional path for packets.

Supported NAT types

The current implementation of `pcn-nat` provides a limited subset of the functions available in iptables and listed in Section 2.3.1.

- Source NAT: although command-wise available, pure SNAT is not implemented yet, because of two main reasons.

  First, the external address is bound to be the one of the public network interface of the NAT instance: this is a direct consequence of having only two interfaces, one of which exposed to the public network.

  Second, it is mandatory to specify an external port number. Iptables, on the other hand, sticks to the semantics of static source natting, which is used to set a one-to-one mapping, thus making only the outside address a mandatory parameter.

Another important difference is the lack of support for address pools and ranges: all NAT commands allow for only a single address to be specified each time, making bulk configuration somehow cumbersome.

- Masquerade: available, it is the default behaviour of NAT instances if no rule is explicitly set. The NAT instance configuration requires the user to specify an IP address for the public interface, and all packets will have their source IP address changed with that one.
- Destination NAT: not available. This means pcn-nat is currently unsuitable for any environment where a host has to be reachable in the inside network.
- Redirect: not available

Rule match fields

pcn-nat rules are called *map entries*. Map entries can only match packets based on the following fields:

- Source IP address
- Source port number
- Layer-4 protocol (ICMP, TCP, UDP)

All the matching fields are mandatory. The configuration of a map entry requires the user to provide a new source IP address and a new source port number, which will be used to modify the packet headers.

Chapter 3

NAT support in pcn-iptables

The first part of this thesis project is the extension of pcn-iptables to include support for network address translation.

3.1 Proposal

In this Section a possible solution is proposed. The goal of this architecture is to support NAT without using additional eBPF maps, extending the *connection tracking table* accordingly.

3.1.1 Design

In this part we present the design of the proposed solution, trying to avoid mentioning the implementation.

Extending the Connection Tracking table

In order to use the connection tracking table to store natting information the following entry fields were added:

- Translation type, to understand whether to change the source or destination header fields
- New IP address, to replace in the packet IP header
- New port number, to replace in the packet L4 header

The details about the data structures and the algorithms used with the new connection tracking table can be found in the next section.

Available hooks

Let us recall that eBPF provides two hooks for packet interception, whereas Netfilter provides five of them. Figure 2.9 provides a quick overview of the relative position of these hooks, and a good starting point for the discussion of the design of the proposed solution.

Prerouting

The PREROUTING chain must be hit before any routing decision has been made, and before any further packet processing. Therefore, in this proposal the Prerouting NAT works immediately after the `Parser` module, but before the `ChainLogic`. With respect to the other pcn-iptables modules, an incoming packet will be already destination-natted, therefore all the filtering operations occur on natted packets. This means that pcn-iptables filtering rules must take into account the existence of NAT, and the user must configure his rules accordingly.

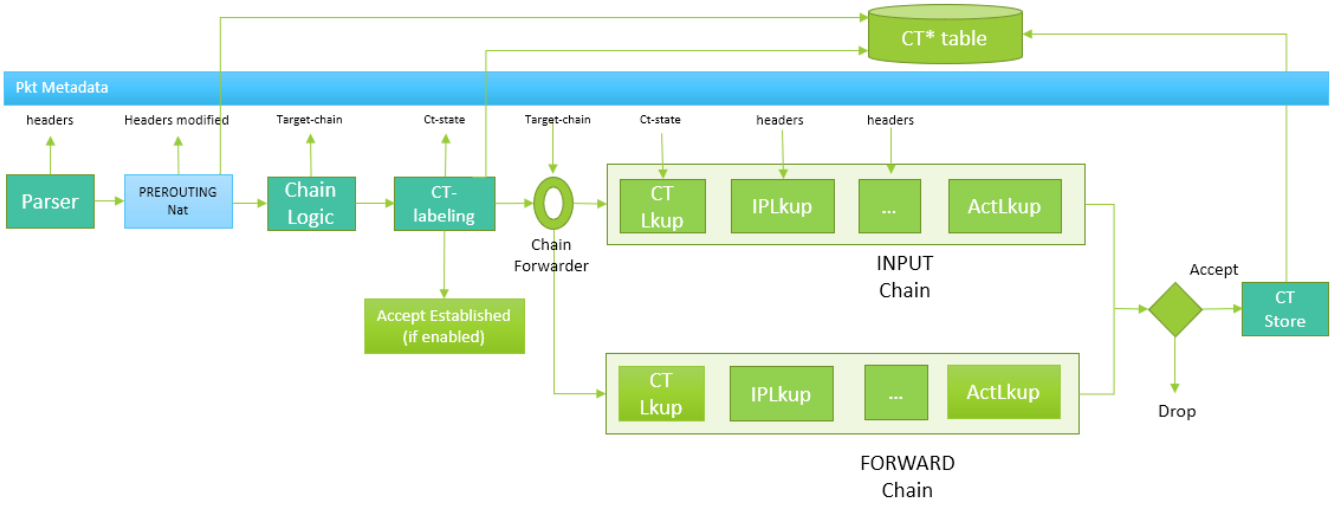


Figure 3.1: Ingress NAT in pcn-iptables

Postrouting

On the other hand, the POSTROUTING chain must be the last step in the packet processing, immediately before being sent to the network card. Therefore Postrouting NAT works immediately after the `ConntrackTableUpdate` module. With respect to the other pcn-iptables modules, an outgoing packet will still have its original headers.

This asymmetry between Prerouting and Postrouting NAT, and the fact that the pcn-iptables modules deal with natted incoming packets and non-natted outgoing

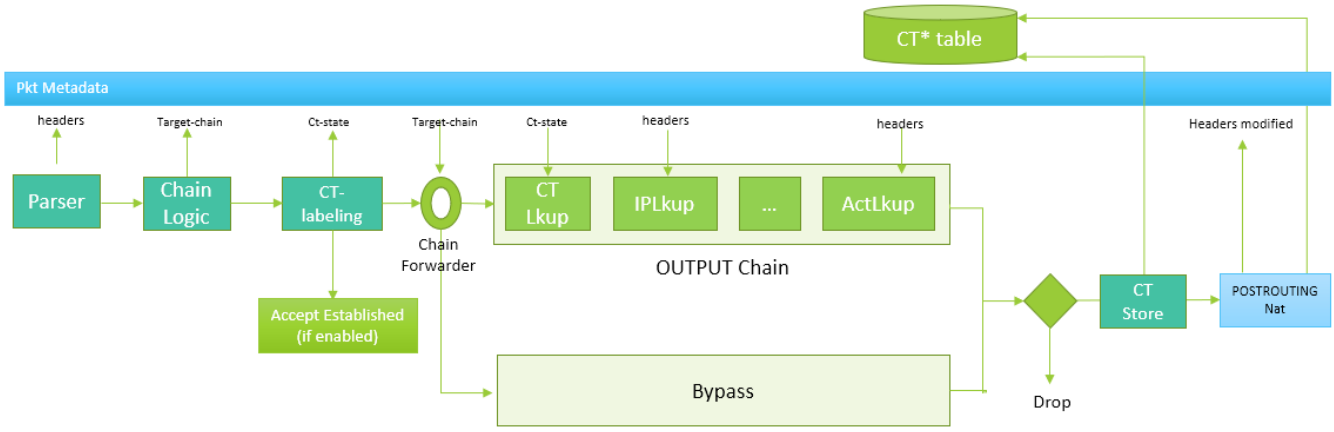


Figure 3.2: Egress NAT in pcn-iptables

packets has a huge impact on the final implementation, as we will see in the following section.

Flowchart

When a packet reaches the NAT module, the natting process follows 3 steps:

- Connection tracking table lookup using the current packet headers:
  - if an entry exists and it includes natting information we proceed with the third step
  - if an entry exists but it does not provide natting information, the packet is left unchanged and forwarded
  - if no entry is found, we proceed with the next step
- Rule tables lookup: search the rule tables to find a matching rule for the packet. If no rule matches, the packet is left untouched and forwarded, otherwise we proceed with the third step.
- Packet manipulation: modify the packet headers and update the connection tracking table - either add a new entry or insert natting data in an existing one.

The previous can be summarized with the following picture.

3.1.2 Rule matching

The *iptables* approach to NAT is similar to its filtering behavior, based on a *match-action* pattern. Each time a hook is hit, iptables looks for the first matching rule,

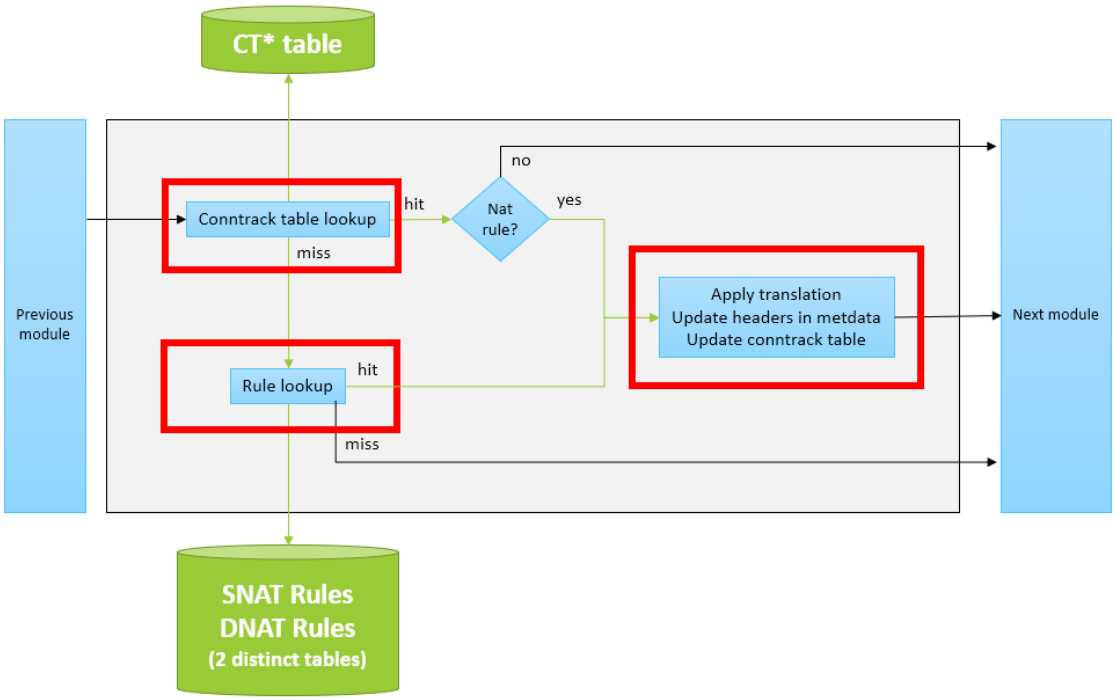


Figure 3.3: NAT modularization

then applies the correspondent NAT action. This allows to specify rather complex rules for the matching part (e.g. ip src, ip dst, port src, port dst, interface, protocol, tcp flags, ...).

The main challenge in finding the first matching NAT rule is to support the moltitude of fields iptables provides - as you can see in the examples reported in Appendix A - while keeping things simple and efficient.

Two approaches are possible when it comes to rule searching:

- hybrid search
- linear search

Hybrid search works well with a fixed number of fields. With this approach a direct access key value table can be used, where the key contains the fields and the value the corresponding action. One disadvantage is that all the key fields must be specified in the rule, which is not flexible.

Linear search allows rules to specify an arbitrary number of fields, but it is slower when it comes to rule matching unless the number of rules is fairly small.

In this proposal the linear search approach is used to get as close as possible to the iptables flexibility.

Since linear search does not limit the number of fields to be checked but only the number of rules, in this proposal the following packet parameters are supported as rule matching fields:

- Source IP address
- Destination IP address
- Source port
- Destination port
- Layer 4 protocol
- Input interface
- Output interface

With respect to iptables, TCP flags are not supported.

3.1.3 Actions

Recall from Section 2.3.2 that iptables supports four actions: source NAT, destination NAT, masquerade and redirect.

In this proposal redirect is not supported. Other actions are supported with the following limitations:

- no support for the OUTPUT chain
- no support for IP address ranges

A set of samples of configuration for pcn-iptables are reported in Appendix B.

3.2 Implementation

The following Section reports in more detail how the proposed design was implemented.

3.2.1 Data structures and eBPF maps

In order to provide support for the necessary natting operations, new data structures and eBPF maps were added to the existing pcn-iptables implementation.

Nat decision

The data structure `struct nat_decision` is used to store all the natting information that is needed to properly modify the packet and operate on the connection tracking table to update it correctly.

```
struct nat_decision {
    uint8_t entryPresent;
    uint8_t updateEntry;
    uint8_t match;
    uint32_t newIp;
    uint16_t newPort;
};
```

Let us briefly describe what each field means:

- `entryPresent`: used as boolean to indicate whether the connection tracking table already contains an entry for the current packet
- `updateEntry`: used as a boolean to indicate whether an existing entry in the connection tracking table has to be updated
- `match`: indicates the type of NAT that must be performed on the packet (`SRC` for source natting, `DST` for destination natting, `NO_NAT` to leave the packet as it is)
- `newIp`: the natted IP address (source IP or destination IP depending on `match`)
- `newPort`: the natted port number (source port or destination port depending on `match`)

The NAT decision is stored in a eBPF map of type array, which is named `natDecision`. The map contains only one entry.

NAT rules

NAT rules are represented in the datapath with the following data structures:

```
enum nat_type {
    SNAT,
    DNAT,
    MASQUERADE
};
```

```
struct nat_v {
    uint32_t bitmask;
    uint32_t srcIp;
    uint32_t dstIp;
    uint16_t srcPort;
    uint16_t dstPort;
    uint8_t l4proto;
    uint16_t inIface;
    uint16_t outIface;
    enum nat_type type;
    uint32_t newIp;
    uint16_t newPort;
};
```

The `enum nat_type` simply describes the type of NAT rule. `struct nat_v` has one field for each packet field supported in the Proposal, fields to store the data to be used to perform NAT, and a `bitmask`, the meaning of which will be explored in Section 3.2.4 during the analysis of the rule matching algorithm.

The NAT rules are stored in two separate but equal eBPF maps - one to perform lookup on incoming packets and one for the outgoing - of type hash, which are named `natrulesINGRESS` and `natrulesEGRESS` respectively.

3.2.2 Extending the Connection Tracking table

As mentioned in the Proposal in Section 3.1.1, three new natting fields were supposed to be added to the connection tracking table: to solve some unexpected issues - described in the examples in Section 3.2.5 - two additional fields had to be introduced, resulting in the following updated `struct ct_v` data structure:

```
struct ct_v {
  uint64_t ttl;
  uint8_t state;
  uint32_t sequence;
  uint8_t trans_type;
  uint32_t newIp;
  uint16_t newPort;
  uint8_t reverse_is_forward_because_added_by_nat;
  uint8_t trans_scope_only_dnat
};
```

`translation_type` can have four values:

- `NO_NAT`: the entry corresponds to a session that does not require natting
- `SRC`: the entry corresponds to a session that needs to undergo source natting

- `DST`: the entry corresponds to a session that needs to undergo destination natting
- `UNDEF_NAT`: the entry corresponds to a session for which we do not know yet whether NAT is needed or not

`reverse_is_forward_because_added_by_nat` and `trans_scope_only_dnat` are used to deal with connection tracking, and their purpose will be explained in detail in the discussion about the integration with the connection tracking algorithm in Section 3.2.5.

In addition to the new table fields, the `state` field was given an additional value, which is called `UNDEF_CONNTRACK`: if an entry has this state, it means that it is being used by the NAT module, but not yet used by the connection tracking algorithm.

3.2.3 Using multiple eBPF programs

In Section 2.4.2 we discussed some of the limitations of eBPF programs. Among them the maximum number of assembly instructions of the compiled bytecode.

Describing the natting process flowchart in Figure 3.3, we also noticed that there are three main self-contained parts in the algorithm: connection tracking table lookup, rule lookup and packet manipulation.

This is why the implementation of the natting module involves three different sub-modules, which are described in the following.

Nat

The first module in the natting process is called `Nat`, and its corresponding datapath is called `Nat_dp`. The responsibilities of this module are the following:

- Reset the `natDecision` map: since this is the first step for any packet and the content of eBPF maps is preserved across executions, it is important to reset it to avoid inconsistencies. All fields in the data structure are set to zero.
- Perform a lookup in the connection tracking table with the current packet headers
  - if no entry is found, the `natDecision` map is left untouched and the second sub-module is called using the eBPF tail call mechanism
  - if an entry exists but its `trans_type` is `UNDEF_NAT`, the decision `updateEntry` and `entryPresent` fields are set to one to indicate that the entry does not contain any valid natting information yet, and the second sub-module is invoked

– if an entry with valid natting data is found, the `natDecision` map is updated accordingly and the third sub-module is invoked, which will act based on the data contained in the updated decision

Please note that no rule lookup is performed in sub-module `Nat`, nor any modification to the packet.

NatLookup

The second module in the natting process is called `NatLookup`, and its corresponding datapath is called `NatLookup_dp`.

This sub-module implements the rule matching algorithm described in Section 3.1.2. Please note that no operation is performed on the packet nor on the connection tracking table.

However, since this module is invoked if and only if a rule lookup is necessary, it is its responsibility to update the `natDecision` map with the natting information collected with the lookup.

Whatever the result of the lookup may be - source, destination or no NAT - the third sub-module is invoked to either update the packet, update the connection tracking table, or both.

NatAction

The third and last module in the natting process is called `NatAction`, and its corresponding datapath is called `NatAction_dp`.

This sub-module uses the data available in the `natDecision` map to modify the packet headers. If necessary, the connection tracking table is updated according to the integrated CT algorithm described in Section 3.2.5.

The packet is then sent to the output interface if the natting process was triggered in the `POSTROUTING` chain, or to the `CHAINLOGIC` module if it was triggered in the `PREROUTING` chain, as described in the Proposal in Section 3.1.1.

Duplication of each sub-module

Each module in pcn-iptables exists twice in the processing flow: once for the egress direction and once for the ingress. This happens for the natting sub-modules as well: each sub-module is injected twice, therefore six eBPF programs are deployed in total. The ingress and egress instances are made different using conditional compiling and string replacing. For the sake of brevity, let us call `NatIngress` and `NatEgress` the whole natting module in the two directions.

When a packet arrives to a host with pcn-iptables running, the packet enters an interface, therefore triggering the `NatIngress` set of modules. If the recipient of the packet is not the host itself, the packet is bound to exit from an interface, therefore triggering the `NatEgress` set of modules.

This means that the packet is processed twice, and this will be a key point of discussion in the integration with the connection tracking algorithm.

3.2.4 Rule matching algorithm

As discussed in the Proposal in Section 3.1.2, a linear search approach has been used to scan the rule tables to find a matching rule for a packet.

Recall from Section 3.2.1 that `struct nat_v` includes a `bitmask` field. This field, although being an integer number, is considered as a bit array for the purpose of the matching algorithm: each of the seven available matching fields is assigned one bit, which is set to one if the rule specifies the field and to zero otherwise.

The rule matching algorithm is implemented in the `NatLookup` sub-module. The algorithm is simple in theory, but it is complicated in practice by the limitations of eBPF programs, first of all the necessity to unroll loops. Although this operation is automatically performed by the JIT compiler in the Linux kernel, there is a caveat: the compiler and the verifier do not handle loop unrolling if there are less than two iterations. Furthermore, they need to know exactly how many iterations have to be performed, which means the loop variable must be a hard-coded constant.

To overcome this issue, conditional compiling instructions and string replacing have been used. Polycube handles datapaths as if they were strings of text: this is very handy because it allows programmers to replace portions of code with actual values immediately before injecting the code in the kernel for compilation. In the datapath string a `_RULES` placeholder was used, which is replaced with the number of rules every time a rule is added or removed; the `NatLookup` sub-module's datapath is then re-injected. Once `_RULES` is replaced, three cases could happen during compilation:

- `_RULES = 0`: there is no need for any lookup at all. The only possible outcome is for the decision's match field to be `NO_NAT`
- `_RULES = 1`: there is no need to loop. Wrapping the `for` loop shell in conditional compiling instructions, it is not considered during compilation. However, the actual matching algorithm in the loop body is preserved, and it is applied on the only existing rule
- `_RULES > 1`: the loop must be unrolled. The shell of the `for` loop is compiled with the algorithm as the loop body

For sake of clarity it is hereby reported the conditional compiling structure.

```
#if _RULES == 0
    goto NOT_MATCHED;
#elif _RULES == 1
    int key = 0;
    nat_value = natrules_DIRECTION.lookup(&key);
    if (nat_value == NULL){
        // No rule with this ID
        goto NOT_MATCHED;
    }
#elif _RULES > 1
    #pragma unroll
    for (int i = 0; i < _RULES; i++) {
        int key = i;
        nat_value = natrules_DIRECTION.lookup(&key);
        if (nat_value == NULL){
            // No rule with this ID
            goto NOT_MATCHED;
        }
#endif

    // Matching algorithm...

NOT_MATCHED:;
    #if _RULES > 1
    }  // Close the for loop
    #endif
    decision->match = NO_NAT;
```

The loop being unrolled poses another issue if we consider another limitation of eBPF programs, which is the maximum number of instructions. The matching algorithm, although simple, requires a certain number of instructions, which results in the maximum number of possible rules being limited. Our tests showed that the eBPF validator accepts the program until 20 rules are inserted, which is suitable for small home applications, but is just not enough for any enterprise or backbone infrastructure.

Please do not mistake the number of rules with the number of sessions in the session table: the fact that only few rules are deployable does not limit the number of connections that can use them.

Let us finally present the matching algorithm with a small snippet:

```
#if _MATCH_SRC_IP
    /* Src IP Match */
    if (GET_BIT(nat_value->bitmask, 0)) {
        if (pkt->srcIp != nat_value->srcIp)
            goto NOT_MATCHED;
    }
#endif
```

Simple as anticipated: for each rule and for each available matching field, the corresponding bit in the bitmask is evaluated: if the bit has value one, the rule has to be applied on the field, and if there is a mismatch the rule is considered not applicable as a whole. If the bit is zero, the corresponding field is not evaluated at all and the algorithm proceeds to the next field.

To mitigate the disadvantage of unrolling the loop a small optimization was made, which is visible in the conditional compiling instructions in the previous snippet. When a rule is added or updated not only is the `_RULES` substring replaced, but also substrings that allow or prevent certain fields to be checked. To do this, before injection the rules are evaluated in the control plane: if at least one rule specifies a field, then that field's compilation is enabled, otherwise it is not, allowing for a smaller loop body and possibly more rules.

Although advantageous from a code-size point of view, this behaviour introduces a dependency between the complexity of the rules and their maximum number. Let us consider the case where lots of rules where added that specified one field only: the verifier accepts the injected code because the loop body is very small. Now one more rule is inserted which specifies all the matching fields: the newly injected code has all the matching fields enabled for all loop iterations, resulting in a much larger number of instructions, which could possibly fail to be verified.

Finally, if no rule matched the packet, the decision's match field is set to `NO_NAT`. On the other hand, as soon as one rule matches the packet, the decision is updated with the corresponding transformation data. In both cases, the `NatAction` submodule is invoked.

Please notice that the first rule that matches is applied, even if there are other rules matching on more fields than the selected one. This mimics the iptables behaviour, where rule priority is enforced by explicitly setting a position for the rule in the rule table and not with a most-matching approach. Setting a rule's position is possible with the pcn-iptables api, therefore this approach was kept.

As we will see, this policy is opposite with respect to the one used in pcn-nat.

3.2.5 Integration with the CT algorithm

The connection tracking mechanism implemented in pcn-iptables is quite complex: one of the biggest challenges of this thesis project was to find a way to extend it

in an efficient way, without breaking its internals or altering the way it treated packets. A fundamental requirement was that if no NAT rule was ever specified, the program should behave as if the NAT implementation never happened.

We will start our discussion of the integration with the connection tracking algorithm with two examples.

Connection tracking algorithm with masquerade

Let us suppose we have a masquerade rule deployed, that changes the source IP addresses of all packets exiting from interface `eth0` to `2.2.2.2`.

Such a rule can be specified with the following syntax in the Polycube CLI:

```
pcn-iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Now let us suppose that a packet with the following headers wants to leave the inside network and is processed by pcn-iptables, and that such packet belongs to a new connection - that is, no former entry exists in the connection tracking table.

PKT 1			
srcIp	dstIp	srcPort	dstPort
10.0.0.1	8.8.8.8	1234	80

The protocol is TCP, so there is a packet exchange to perform the handshake, which makes sure that packet are sent in both directions.

Recall from Section 3.2.2 the fields of an entry in the connection tracking table, and keep in mind that masquerade happens in the `POSTROUTING` chain, as the last step of the pcn-iptables processing.

Also recall from Section 3.2.3 that when a packet hits pcn-iptables it is processed by each module twice - the ingress and egress version.

When the packet arrives at the host it triggers the ingress instances of the natting sub-modules. Since the ingress NAT processing is done before connection tracking, the connection tracking table has no matching entry for the current packet, therefore the ingress instance of `NatLookup` is called. Since there is no prerouting rule - only the masquerade one, which is processed in postrouting - the sub-module invokes the ingress version of `NatAction` setting the decision match field to `NO_NAT`, which proceeds by adding an entry in the connection tracking table using as key the headers of the current packet and and setting the value's fields as following:

- `trans_type`: `NO_NAT`, to indicate that no natting operation must be performed on the packet. This is not true, but we will see how such situation is handled in a moment
- `trans_scope_only_dnat` is set to one, to indicate that this rule is meaningful only for the ingress direction and should be ignored by the egress

- `state` is set to `UNDEF_CONNTRACK` because connection tracking processing has not occurred yet

The ingress natting module forwards the packet to the next program to perform connection tracking, but a keen observer may have noticed that we violated the initial assumption of the connection tracking working fine both with NAT and without: in this case without natting no entry in the connection tracking table would exist when the tracking process begins. Unfortunately a seamless integration was not possible, therefore the connection tracking algorithm had to be modified.

The approach was the following: consider any entry whose `state` field is set to `UNDEF_CONNTRACK` as a non-existing entry. The connection tracking module, tricked into thinking that no entry exists for the current packet, will try to insert one to store the connection tracking information.

However, being the key the same of the existing entry, the operation simply replaces the value's fields, included the state, which will be something else than `UNDEF_CONNTRACK` - in case of a TCP connection, `state = SYN_SENT`.

To sum up, at this point the connection tracking table includes one entry for this connection, whose structure is reported in the table below.

Entry 1								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	10.0.0.1		8.8.8.8		1234	80	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	SYN_SENT	?	NO_NAT	0	0	false	true

The packet proceeds its path entering the egress processing, which begins with the connection tracking and ends with natting.

The current packet's source IP address does not match any of the host's interfaces, therefore the `OUTPUT` chain is not hit, and the packet is directly passed to the `POSTROUTING` chain without further connection tracking processing.

Now the egress instances of the natting sub-modules are hit.

The `Nat` sub-module finds an entry in the connection tracking table for the current packet, with `trans_type = NO_NAT`. This would make us believe that no natting operation is performed, which would be wrong, but we are forgetting that `trans_scope_only_dnat = 1`. When these two conditions are true, the entry is treated as if the state was `UNDEF_NAT`, and the decision is set to force the `NatAction` to update the connection tracking entry with the proper NAT data.

Since `UNDEF_NAT` was used, the `NatLookup` is invoked, which finds the masquerade entry and updates the decision before passing the packet to the `NatAction`. This sub-module takes care of modifying the packet headers, and to deal with the connection tracking table.

Recall from Section 2.5.2 that the connection tracking table was designed to be direction-independent, which means one entry should be enough for packets belonging to a session independently of whether they are exiting or entering the inside network.

However, because of the asymmetry pointed out in Section 3.1.1, the packet headers will be different in the two directions when the NAT processes them: they are the original packet in the outgoing direction, and the natted packet in the incoming direction. Being the key of the session table the packet header itself, it is clear that now two entries are needed for each session, one with the original packet headers and one with the natted ones: we will refer to such two entries as `forward entry` and `reverse entry` respectively.

Now we have all the elements to proceed with our example: `NatAction` updates the existing entry to include natting information for the outgoing packets, and inserts a new entry for incoming packets.

The forward entry is the entry added by the ingress natting process, as shown before. The key of the forward entry does not change, and neither do the `ttl`, `state` and `sequence` value fields. The other fields are modified as follows:

- `trans_type`: SRC, to indicate that the source headers must be modified
- `newIp`: the IP address of the interface `eth0`, following the masquerade semantics
- `newPort`: a port number chosen according to a certain policy
- `trans_scope_only_dnat` is set to zero, to indicate that the entry is now fully qualified to provide natting information for outgoing packets
- `reverse_is_forward_because_added_by_nat` is set to zero because `NatAction` only updated the entry

The resulting forward entry is the following:

Forward Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	10.0.0.1		8.8.8.8		1234	80	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	SYN_SENT	?	SRC	2.2.2.2	1025	false	false

The reverse entry has to match incoming packets. Since the packet headers are natted before being sent out, the other party will respond to the natted IP address and port. When such a response hits pcn-iptables, it must match the reverse entry.

Therefore the key of the entry is the following:

- `srcIp`: the destination IP of the current packet, which will be the source of the response packet
- `dstIp`: the natted source IP address, which will be the one the other party responds to
- `srcPort`: the destination port number of the current packet
- `dstPort`: the natted source port number, which will be the one the other party responds to
- `l4proto`: the same as the current packet

The value's field will be the following:

- `ttl`: a numerical value
- `state`: `UNDEF_CONNTRACK`, to indicate that when the packet will come there will be no tracking information yet, because prerouting happens before connection tracking
- `sequence`: the current sequence number plus one
- `trans_type`: DST, to indicate that the destination headers must be modified
- `newIp`: the current packet's original source IP address, which is the actual recipient of the incoming packet
- `newPort`: the current packet's original source port number
- `trans_scope_only_dnat` is set to zero
- `reverse_is_forward_because_added_by_nat` is set to zero

The resulting reverse entry is the following:

Reverse Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	8.8.8.8		2.2.2.2		80	1025	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	UNDEF_CT	?	DST	10.0.0.1	1234	false	false

At this point the TCP `SYN` packet has been sent, and the `SYN ACK` comes back. When it does, it first triggers the ingress natting and tracking modules, then the egress, just like the previous packet, with the difference that now the necessary entries are already present in the connection tracking table.

The ingress `Nat` sub-module finds the reverse entry added before, which already provides valid natting information: `NatLookup` is skipped and `NatAction` is invoked. The packet is modified and sent to the tracking logic, and no further operation is performed on the connection tracking table.

Since the packet headers have changed, the connection tracking will ignore the reverse entry we added before, and will instead use the forward entry, which will match when searched with the packet headers reversed: this happens because although there are two entries for each connection, as far as connection tracking is concerned the table is still direction independent.

The `state` field in the forward entry is set to `SYN_RECV` in case of TCP, waiting for the final `ACK` to mark the connection as `ESTABLISHED`.

The two entries for the connection are now the following:

Forward Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	10.0.0.1		8.8.8.8		1234	80	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	SYN_RECV	?	SRC	2.2.2.2	1025	false	false

Reverse Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	8.8.8.8		2.2.2.2		80	1025	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	UNDEF_CT	?	DST	10.0.0.1	1234	false	false

The `ACK` packet, sent by `10.0.0.1`, will match the forward entry: therefore the connection tracking will work as expected and will mark the connection as `ESTABLISHED`.

Connection tracking algorithm with destination nat

Let us now suppose we have a destination NAT rule deployed, that changes the destination IP addresses and port numbers of all packets directed to `2.2.2.2`, port `80` to those of an internal host, say `10.0.0.1` port `8080`

Such a rule can be specified with the following syntax in the Polycube CLI:

```
pcn-iptables -t nat -A PREROUTING
             -d 2.2.2.2 --dport 80
             -j DNAT --to-dest 10.0.0.1:8080
```

Now let us suppose that a TCP `SYN` packet with the following headers arrives, trying to establish a new connection:

PKT 1			
srcIp	dstIp	srcPort	dstPort
3.3.3.3	2.2.2.2	1234	80

Similarly to the previous case, the packet first triggers the ingress natting modules, and the connection tracking table has no information whatsoever about the current packet.

The ingress `NatLookup` is invoked, which finds the matching prerouting rule for the packet, updates the decision and invokes `NatAction`.

As usual `NatAction` has to modify the packet and deal with the connection tracking table. With respect to the previous case, the forward and reverse entries are added together.

The key of the forward entry corresponds to the current packet headers, before natting is applied: in this case the forward key will match subsequent packets of the connection - with masquerade this happens to the reverse entry. The value of the forward entry is set as following:

- `sequence` and `ttl` are properly set
- `trans_type`, `newIp` and `newPort` are set to `DST`, `10.0.0.1` and `8080`
- `state` is set to `UNDEF_CONNTRACK`

Forward Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	3.3.3.3		2.2.2.2		1234	80	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	UNDEF_CT	?	DST	10.0.0.1	8080	false	false

The reverse entry is the one that matches outgoing packets in the connection, which is initiated from the outside. Therefore the key must match the packet before natting is performed. The value of the reverse entry is set as following:

- `sequence` and `ttl` are properly set
- `trans_type`, `newIp` and `newPort` are set to `SRC`, `2.2.2.2` and `80`
- `state` is set to `UNDEF_CONNTRACK`

- `reverse_is_forward_because_added_by_nat` is set to one: this is the first and only case where this field is set to one, and we will see in a moment why this happens here

Reverse Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	10.0.0.1		3.3.3.3		8080	1234	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	UNDEF_CT	?	SRC	2.2.2.2	80	true	false

After `NatAction` finishes, the ingress connection tracking begins: the first iteration finds the matching entry - the reverse entry - with `state = UNDEF_CONNTRACK`, which is considered as missing entry and is updated in the `ConntrackTableUpdate` module as discussed in the previous example. Also, after NAT the packet looks like this:

PKT 1			
srcIp	dstIp	srcPort	dstPort
3.3.3.3	10.0.0.1	1234	8080

Once again this process is direction independent, and since the packet headers have been natted it will refer to the reverse entry after finding it with the headers switched. The algorithm would thus consider the matching entry as a reverse - from a tracking point of view - entry, which would break the state machine on which the connection tracking itself is based.

This is why the `reverse_is_forward_because_added_by_nat` field was set to one: we are explicitly informing the algorithm that the current entry is actually a forward one, even if it was found after a lookup with the packet headers switched. The connection tracking ends by updating the `state` field of the reverse entry value to `SYN_SENT`.

At this point two entries exist in the connection tracking table for the current packet, which are shown in the following tables.

Forward Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	3.3.3.3		2.2.2.2		1234	80	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	UNDEF_CT	?	DST	10.0.0.1	8080	false	false

Reverse Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	10.0.0.1		3.3.3.3		8080	1234	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	SYN_SENT	?	SRC	2.2.2.2	80	true	false

Now the very same packet hits the egress connection tracking, which does not perform any operation - just like before, the OUTPUT chain is not hit - and passes the packet to the egress natting modules. The natted packet headers do not match any entry in the table because natting is not direction-independent and it requires a direct match. `NatLookup` is invoked, but no rule is present, therefore `NatAction` is invoked.

The decision is `NO_NAT` and there is no entry, so the most intuitive operation would be to add a new entry: but look at the current packet headers and compare them to the key and value of the reverse entry. If we added a `NO_NAT` entry, that would be exactly the reverse entry with switched headers, which is forbidden because it would break the direction-independent behaviour of the algorithm: a packet would match the entry as a forward one, when the correct behaviour would be to match the reverse entry as a reverse one.

Therefore, the connection tracking table is not updated in this step.

Now the response packet comes back from the inside host, with the following headers:

PKT 2			
srcIp	dstIp	srcPort	dstPort
10.0.0.1	3.3.3.3	8080	1234

The ingress NAT is hit, and the connection tracking table provides a matching entry with natting data, but the suggested transformation is `SRC`, which cannot happen in prerouting, and therefore the rule is ignored as if it was `NO_NAT`.

The connection tracking algorithm proceeds and finds a direct entry for the current packet, but the `reverse_is_forward_because_added_by_nat` field is 1, so the entry is treated as a reverse from the algorithm point of view. This makes sure that the state machine keeps working as expected. The entries are updated as following:

Now the egress connection tracking is hit, but as usual the `Nat` sub-module is immediately invoked. The lookup in the table is successful as it retrieves the reverse entry, which has source natting information and is therefore accepted as a valid natting entry for the postrouting. The packet is modified and sent.

When the next packet comes in, it will match the forward entry, which provides

Forward Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	3.3.3.3		2.2.2.2		1234	80	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	UNDEF_CT	?	DST	10.0.0.1	8080	false	false

Reverse Entry								
Key	srcIp		dstIp		srcPort	dstPort	proto	
	10.0.0.1		3.3.3.3		8080	1234	TCP	
Value	ttl	state	seq	trans type	newIp	newPort	reverse is forward	trans only dnat
	?	SYN_RECV	?	SRC	2.2.2.2	80	true	false

destination natting information which is considered valid for the ingress. Just like before the ingress connection tracking will consider the reverse entry as a forward one for the current packet, and will update the state from `SYN_RECV` to `ESTABLISHED`, because the TCP handshake is complete.

3.3 Open issues

3.3.1 Complexity of the integrated algorithm

Integrating NAT in the connection tracking algorithm has been way more difficult than what we initially planned.

A clear signal of this is the fact that instead of the three additional table fields we thought would suffice, two additional had to be added just to deal with problematic situations we just did not foresee.

During the analysis of the two examples in the previous Section we already pointed out some critical points that had to be addressed during development to make sure that NAT worked, and solutions to them were found which greatly complicated the final code, sometimes making it unreadable unless very thorough comments were added: we used the `trans_scope_only_dnat` field to make sure that the egress would ignore incomplete rules added by the ingress and we were forced to add the `reverse_is_forward_because_added_by_nat` field to keep the connection tracking working and we had to pay attention not to add a reverse entry in some cases, not to break the state machine.

The root cause of this is actually quite simple: the connection tracking table is direction independent, whereas NAT is not.

When dealing with eBPF programs the single most expensive operation is a map lookup: the worst case scenario in pcn-iptables with NAT implies five lookups in

the connection tracking table only, ignoring those for the packet headers and for the NAT decision. This causes a massive decrease in performance, as we will report in Chapter 5.

Consider a different architecture where NAT is completely independent from connection tracking: we would still need two sets of rules for the outgoing and incoming packets, but we could exploit the separation of ingress and ingress to setup two different natting maps, one for the incoming packets and one for the outgoing.

This solution, which leans toward the one proposed in the following chapter for pcn-nat, would indeed bring some advantages, both to complexity and performance:

- `trans_scope_only_dnat` would be useless, because the egress NAT would not consider the ingress natting entries
- `reverse_is_forward_because_added_by_nat` would also be useless, because the connection tracking table would be left untouched from the whole integration

A very interesting thing about this architecture is the fact that the connection tracking algorithm would not change with respect to the pcn-iptables without NAT, because either ingress NAT happens before, or egress NAT happens afterwards. All the natting operations would be managed by the natting modules with their own data structures.

Speaking of performance, if we ignore the connection tracking table and the linear rule matching algorithm, only two map lookups would be needed, one for the ingress and one for the egress, and most importantly there would be no worst case, because the same process would apply to all packets.

3.3.2 Port selection policy

Whenever a masquerade rule or a source NAT rule - with no port explicitly set - has to be enforced, a new source port number has to be selected in order to properly modify the packet and to provide the bidirectional mapping in the connection tracking table.

The algorithm currently implemented to address this necessity is very simple: all the numbers between 1025 and 65535 are used in round robin. The first connection will be assigned port 1025, the second one 1026 and so on.

This process is handled by the following piece of code.

```
static inline __be16 get_free_port() {
    u32 i = 0;
    u16 *new_port_p = first_free_port.lookup(&i);
    if (!new_port_p)
        return 0;
    rcu_read_lock();
    if (*new_port_p < 1024 || *new_port_p == 65535)
        *new_port_p = 1024;
    *new_port_p = *new_port_p + 1;
    rcu_read_unlock();
    return bpf_htons(*new_port_p);
}

BPF_TABLE("array", u32, u16, first_free_port, 1);
```

The next port number is stored in a eBPF map of type array, with only one unsigned short integer number which it is read and updated whenever a new port number is needed.

This approach has the only advantage of being conceptually simple, but it does suffer from some serious issues:

- all port numbers are used, ignoring that other services running on the host may already being using them on their own
- when a port number is selected, Linux is not informed that that port is no more available for other running services
- when the round robin algorithm starts back from 1025, there may be overlapping natting information for more than one connection

While the last one is strictly related to the implementation and may actually be solved by adding some complexity to the algorithm, the first and the second are more related to how eBPF and Linux communicate, and to which tools are available to the developers to make this communication effective.

A theoretical solution to the first problem is simple and naive: either obtain a list of the used ports and skip them, or a list of free ports and choose from them.

Although very basic in purpose, this solution is not viable.

If we assumed this operation to be performed in the control plane, immediately before injecting the eBPF code in the kernel, we would need to use some sort of Linux function that returns either the free or the busy ports: unfortunately, there is no such function available yet.

If we assumed to delegate this to the data path, we would need dedicated eBPF helpers, which simply do not exist.

The same problems are also valid for the second problem: a port number could be reserved with a socket bind operation from the control plane, but this would require the data path to rely on the control plane for every new translation it has to perform, which would considerably reduce performance.

A possible workaround that may actually solve both problems is to reserve a range of port numbers from the very beginning, possibly passing this range as a parameter during the service instantiation: the control plane would have to reserve all the ports at once, and a proper eBPF map would either store the first and last port available, or all of them.

Speaking of eBPF maps, a particularly efficient data structure to store available port numbers would be a FIFO queue that contains all the available port numbers: as soon as a new number is needed a pop operation is performed, whereas a push operation would be needed to make unused port numbers available again. This FIFO map is not available in the Linux kernel yet, but it is currently in the works in the Polycube workgroup and may be used in the next updates.

Chapter 4

A new API for pcn-nat

The second part of this thesis project is the definition and implementation of a new API for the pcn-nat cube, which is the service that provides natting features in a Polycube service chain.

With respect to pcn-iptables, pcn-nat does not aim to emulate or provide compatibility with iptables, since it is designed to be part of a more complex service chain. This gave us a vast freedom of choice when it came to decide how to design and implement the service.

4.1 Proposal

In this Section a proposal is presented: a new API was necessary to improve its functionality and flexibility, and to overcome the limitations of the previous implementation, which we discussed in Section 2.5.3.

4.1.1 iptables-like versus custom

When the discussion about reworking pcn-nat first started, the first decision we had to deal with was what type of API we wanted for it. Two alternatives were possible: an API very similar to the one implemented in pcn-iptables, or a totally new one.

iptables-like API

As discussed in the previous chapter, the iptables API for NAT is quite complex: it provides the maximum possible flexibility by allowing any field to be optional, it allows to enter very specific rules and to specify the chain where such rules had to be enforced.

The drawback, however, was that such a freedom took a toll on performance, especially when it comes to the rule matching algorithm, which as discussed is

implemented with a linear search and has all the eBPF-related problems we already discussed.

Custom API

Because of all this, designing and implementing a brand new API seemed like a good idea, and it did even more when we considered how this pcn-nat cube is meant to be used in a service chain: it is not designed to be the only piece of software running as pcn-iptables was, but as the boundary element before the outer network, the element in charge of providing that set of features that are needed in order to allow a functional networking between an inside and an outside network.

As you can imagine, this is the API we decided to implement, and that we will discuss in the rest of this section.

4.1.2 Design

What we wanted to achieve was a clear, simple and efficient design, that could provide a user-friendly command line interface to setup the service.

The first step we made in this direction was a key change with respect to the iptables behaviour, which is that we defined different type of NAT rules for the different actions we wanted to perform: the action is not a parameter anymore, it is defined by the rule itself.

A second dramatically important decision we decided to make was to define individual sets of configurable fields for each type of rule, and to make the fields in these sets mandatory. This may seem a very strict constraint on usability, but as we will see everything was worked out in order to keep a flexible interface and a very efficient implementation.

Four types of rules were introduced:

- Source natting rules
- Masquerade rules
- Destination natting rules
- Port forwarding rules

Source natting rules

With these rules the user can configure what we called the Basic NAT in Section 2.1.1.

The only two fields available for these rules are:

- `internal-net`: the internal network address that we want to translate. It can be either a single IP address - 10.0.0.1/32 - or a network range - 10.0.0.0/24. CIDR notation must be used.
- `external-ip`: the outside IP we want to use to translate the internal ones. It has to be a single IP address

Now think about what the user can and cannot do with this API with respect to the iptables API: of course the only matching field is the packet's source IP, and we totally cut off any other parameter. But also think about what source natting is about: it was introduced to perform a many-to-one mapping of inside address on outside address, and this is still possible.

Masquerade

Masquerade is a per-se type of rule: while in theory it would be possible to apply masquerade only to a subset of packets defined by a set of matching parameters, this rarely happens in practical applications, and also the NAT cube does not have more than one output interface on which two different masquerade rules can be enforced.

This is why we decided to consider masquerade not like a rule, but like an option that can be enabled and disabled: when masquerade is enabled, all the packets that do not match a source NAT rule are automatically intercepted and their source headers are replaced with the IP address of the outside interface and with a new port number. When masquerade is disabled, any packet that would not match any source NAT rule would exit the network unchanged.

Think of this masquerade as a default policy that can be either added as a whole or removed, and not configured.

Destination natting rules

With these rules it is possible to map an outside IP to an inside IP for incoming packets, for all destination port numbers. We decided to provide this possibility to mimic what happens on many home gateways when a DMZ is enabled: all traffic directed to the public IP is redirected to a specific host in the inside network.

The only two fields available for this type of rule are:

- `external-ip`: the external address we want to map
- `internal-ip`: the internal address to which we want to redirect the traffic

Port forwarding rules

Port forwarding rules implement the Bidirectional NAT discussed in Section 2.1.3. With these rules it is possible to associate an internal host to a certain outside IP

and port numbers, for example to have a running web server in the inside network accessible from the outside.

The available fields for these rules are:

- `external-ip`: the external IP to associate to the host
- `external-port`: the external Layer 4 port to associate to the host
- `internal-ip`: the actual IP address of the host
- `internal-port`: the actual port number of the running service in the host
- `proto`: the Layer 4 protocol. This field is optional, and this is the only case when optional parameters are provided by this API. In general a service running on a certain port works either with TCP or UDP, therefore it is up to the user to decide whether to explicitly set a protocol. The protocol can be `TCP`, `UDP`, `ALL` - not setting the protocol defaults to `ALL`.

4.1.3 Debug view

It is easy to forget about debugging when everything works fine, but when it does not it is important to have an intuitive way to find out what is going wrong.

Therefore, during the design of this new API, we had to introduce a feature to allow users to keep track of how and why packets are being translated in a certain way. We called this feature the **natting table**.

The natting table is similar in meaning to the connection tracking table in pcn-iptables: it provides an overview of the ongoing natting transformations and can be read - not written - by the end user. Each row of the natting table contains the following fields:

- `internal-src`: the inside source address of a packet
- `internal-dst`: the inside destination address of a packet
- `internal-sport`: the inside source Layer 4 port of a packet
- `internal-dport`: the inside destination Layer 4 port of a packet
- `proto`: the protocol of the packet (`ICMP`, `TCP`, `UDP`)
- `originating-rule`: the type of rule that was applied to the packet (`SNAT`, `Masquerade`, `DNAT`, `Port forwarding`)
- `external-ip`: the external IP address of the packet
- `external-port`: the external Layer 4 port number of the packet

As we will see in the implementation section, the natting table the user can view is one of the two natting tables used by the datapath to perform natting transformations. Two birds with one stone.

4.1.4 Rule matching

The new rule matching algorithm is the greatest improvement we could achieve with respect to the linear search algorithm implemented in pcn-iptables.

It is also the main reason why the decision of introducing four type of rules with individual sets of fields was made.

This leap forward was made possible by the existence of LPM maps in eBPF. As mentioned in Section 2.4.2, with LPM maps it is possible to obtain - with one simple lookup with a well-formed key - the most matching entry in a table.

It is not easy to abstract the explanation of this algorithm from the implementation details which will be shown later. In general, the four types of rules have been grouped to obtain all the rules applied to outgoing packets (source NAT and masquerade) and incoming packets (destination NAT and port forwarding): let us call them the *outgoing rules* and *incoming rules*.

Outgoing rules

Source NAT rules specify an internal net and an external IP address. Masquerade is implemented as a default policy to match all the packets that do not match any source NAT rule.

Abstracting from the LPM map per-se, and thinking about the longest prefix match algorithm in general, we can immediately come up with a solution to merge such rules to apply LPM to both of them at the same time: consider masquerade as a rule that can be added or removed from a rule table.

In order to be the last match for a packet, this rule should have an internal net of `0.0.0.0/0`, which is very similar to what happens with the default route in router devices.

Incoming packets

The process is less straightforward when it comes to destination NAT and port forwarding rules, for two main reasons:

- they have a different amount of configurable fields
- the protocol is optional for port forwarding rules

Also, it is important that more precise rules are matched before more general ones: for example, if a destination NAT and a port forwarding rule are configured with the same IP addresses, the port forwarding must have a higher priority, because it also matches a port number. LPM maps are not capable of considering priorities by themselves, but it is possible to enforce this policy if we build the map key accordingly: all the incoming rules must have an IP address configured, some have a port number and some of them a protocol.

If the key is built so that the IP address is before the port number, and the protocol is the last element, we can let the map perform a LPM lookup by setting all the missing values to zero.

Let us clear this analysis with an example: we have two incoming rules and a packet:

PKT			
srcIp	dstIp	srcPort	dstPort
3.3.3.3	2.2.2.2	1234	80

Incoming Rules					
Rule	external-ip	external-port	proto	internal-ip	internal-port
DNAT	2.2.2.2	-	-	10.0.0.1	-
Port Forwarding	2.2.2.2	80	-	10.0.0.1	8080

When the incoming rules are searched with the LPM algorithm the second one will match, even if the first one would have been a valid match too, because the second is more specific.

4.2 Implementation

In this Section the key aspects of the implementation of the previously discussed design are analyzed. Some command line examples of the new API can be found in Appendix C.

4.2.1 Data structures and eBPF maps

To implement the discussed design in an efficient way, new data structures and eBPF maps have been introduced to deal with packets in the datapath: they can be grouped into two main sections, which are those for the *natting table* and those for the *rules*.

The datapath relies on four maps in total:

- outgoing rules map
- incoming rules map
- egress natting table
- ingress natting table

The control plane does not define any additional data structure, since it reads and updates the maps in the data path.

4.2.2 Rules

As we anticipated discussing the design, rules are stored in LPM maps, which guarantee a longest prefix match with one lookup, therefore avoiding a linear search or more complicated approaches.

LPM maps need a well-formed key, where the first field is an integer that indicates the length in bits of the data that must be considered when matching, and the other fields are the actual matching data. There are no constraints on the value, which can contain as many fields as we want.

Without further ado, let us introduce the data structure used for the outgoing rules:

```
struct sm_k {
    u32 internal_netmask_len;
    __be32 internal_ip;
};
struct sm_v {
    __be32 external_ip;
    uint8_t entry_type;
};

BPF_F_TABLE("lpm_trie", struct sm_k, struct sm_v,
            sm_rules, 1024, BPF_F_NO_PREALLOC);
```

The `sm_` prefix stands for source NAT and masquerade.

`struct sm_k` defines the key of the LPM map: the `internal_ip` is the actual match field, whereas the `internal_netmask_len` records how many bits of the `internal_ip` must be checked when trying to match a packet.

When rules are injected in the `sm_rules` map, the `internal-net` parameter is parsed to obtain a network address and a network mask, which are the fields in the key of the map. The `entry_type` field is set to either `SNAT` or `MASQUERADE`, depending on the type of rule we are injecting.

`struct sm_v` defines the value returned by a successful lookup: of course there is the `external_ip` to use for packet translation, but there is a `entry_type` field as well, which will be used to correctly set the `originating-rule` in the natting table.

Now let us analyze the data structures used to represent incoming rules:

```
struct dp_k {
    u32 mask;
    __be32 external_ip;
    __be16 external_port;
    uint8_t proto;
};
```

```
struct dp_v {
    __be32 internal_ip;
    __be16 internal_port;
    uint8_t entry_type;
};
BPF_F_TABLE("lpm_trie", struct dp_k, struct dp_v,
            dp_rules, 1024, BPF_F_NO_PREALLOC);
```

The `dp_` prefix stands for destination NAT and port forwarding.

`struct dp_k` defines the key of the LPM map. With respect to `struct sm_k`, there are three matching fields and `mask`, which plays the same role as `internal_netmask_len`.

When an incoming rule is injected in the `dp_rules` map, one of this scenarios happen, depending on the type of the rule:

- destination nat
  - `mask` is set to `32` to indicate that the rule is supposed to match only the first 32 bits of the key - which is, the `external_ip`
  - `external_ip` is set to the `external-ip` provided by the rule
  - `external_port` is set to zero
  - `proto` is set to zero
  - `internal_ip` is set to the `internal-ip` provided by the rule
  - `internal_port` is set to zero
  - `entry_type` is set to `DNAT`
- port forwarding
  - `mask` is set to `48` to indicate that the rule is supposed to match only the first 48 bits of the key - which is, the `external_ip` and the `external_port`
  - `external_ip` is set to the `external-ip` provided by the rule
  - `external_port` is set to the `external-port` provided by the rule
  - `proto` is set to zero
  - `internal_ip` is set to the `internal-ip` provided by the rule
  - `internal_port` is set to the `internal-port` provided by the rule
  - `entry_type` is set to `PORTFORWARDING`
- port forwarding with protocol
  - `mask` is set to `56` to indicate that the rule is supposed to match 56 bits of the key - which is the whole key, protocol included
  - `external_ip` is set to the `external-ip` provided by the rule

- `external_port` is set to the `external-port` provided by the rule
- `proto` is set to the `proto` provided by the rule
- `internal_ip` is set to the `internal-ip` provided by the rule
- `internal_port` is set to the `internal-port` provided by the rule
- `entry_type` is set to `PORTFORWARDING`

`struct dp_v` defines the value returned by a successful lookup: as in `struct sm_v` there are fields to translate packets - `internal_ip` and `internal_port` - and the field used in the natting table - `entry_type`.

4.2.3 Natting table

The natting table is used as a cache for already natted packets.

```
struct st_k {
    uint32_t src_ip;
    uint32_t dst_ip;
    uint16_t src_port;
    uint16_t dst_port;
    uint8_t proto;
};
struct st_v {
    uint32_t new_ip;
    uint16_t new_port;
    uint8_t originating_rule_type;
};
BPF_TABLE("lru_hash", struct st_k, struct st_v,
        egress_session_table, NAT_MAP_DIM);
BPF_TABLE("lru_hash", struct st_k, struct st_v,
        ingress_session_table, NAT_MAP_DIM);
```

Being defined as hash maps, lookups are faster that those in the rule maps, because there is no LPM algorithm to perform.

`struct st_k` defines the direct access key, and it includes all the relevant packet headers. `struct st_v` contains the natting information - `new_ip` and `new_port` - and `originating_rule_type`, which records the type of rule that matched the packet the entry is associated with.

Two natting tables are defined: one for the incoming packets and one for the outgoing - `ingress_session_table` and `egress_session_table`. This decision was driven by two main reasons:

- from a performance point of view, it is better to have two maps with respect to one with a double size

- only `egress_session_table` is displayed to the user in the debug view

About the second point, let us explain how the natting table entries are inserted: when a packet matches a rule, two entries are created, one for each natting table. This approach is similar to the forward and reverse entries in the connection tracking algorithm integration.

- Matching rule from `sm_rules`
  - Entry for `egress_session_table`, must match outgoing packets
    - key: the inside packet headers are used, before applying nat
    - value: natted source IP and port, `rule_type` either `SNAT` or `MASQUERADE`
  - Entry for `ingress_session_table`, must match packets coming back
    - key: `src_ip` and `src_port` are set to the original destination IP and port, `dst_ip` and `dst_port` are set to the natted source IP and port, `proto` is the same
    - value: `new_ip` and `new_port` are set to the original source IP and port, `rule_type` either `SNAT` or `MASQUERADE`
- Matching rule from `dp_rules`
  - Entry for `egress_session_table`, must match packets going back
    - key: `dst_ip` and `dst_port` are set to the original source IP and port, `src_ip` and `src_port` are set to the natted destination IP and port, `proto` is the same
    - value: `new_ip` and `new_port` are set to the original destination IP and port, `rule_type` either `DNAT` or `PORTFORWARDING`
  - Entry for `ingress_session_table`, must match incoming packets
    - key: the outside packet headers are used, before applying nat
    - value: natted destination IP and port, `rule_type` either `DNAT` or `PORTFORWARDING`

4.2.4 Packet processing

Packet processing is performed in four main steps:

- parsing
- natting table lookup
- rule table lookup
- translation

Figure 4.1 reports the logical sequence of operations performed on any packet that enters pcn-nat.

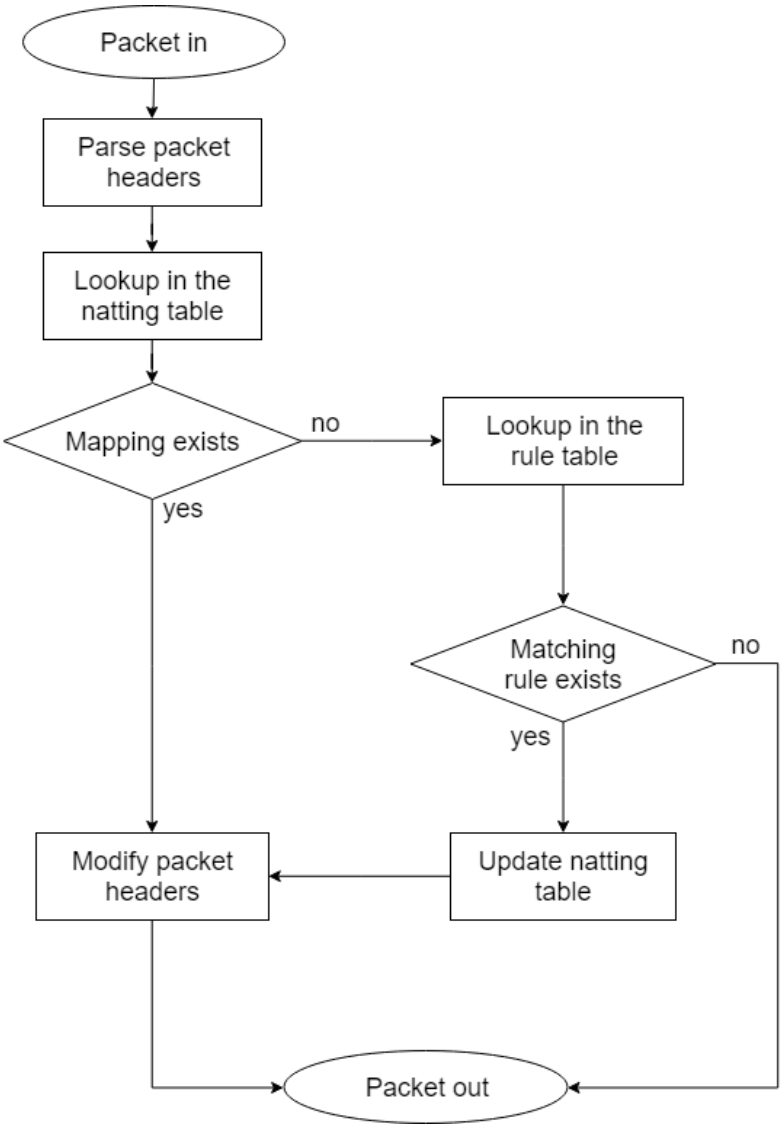


Figure 4.1: Packet processing in pcn-nat

Parsing

During this step the raw packet is analyzed and the headers are extracted for easier handling in the subsequent steps.

The source and destination IP addresses and the Layer 4 protocol are read from the IP header, and the port numbers are read from the Layer 4 header, be it TCP or UDP. In case of ICMP, the source and destination ports are set to the ICMP ID.

Natting table lookup

The packet headers gathered in the previous step are used to build the key with which a lookup in the natting table is performed.

If the packet is coming from the outside the `ingress_session_table` is used, otherwise the `egress_session_table`.

This is a direct access lookup, which can either succeed or fail: if it succeeds, natting information is extracted by the returned value and the next step is skipped.

Rule table lookup

Since there is no entry in the natting tables for this packet yet, a rule lookup must be performed.

If the packet is coming from the inside, the `sm_rules` table is used. The actual lookup is performed with a key with `internal_ip` set to the source IP address of the packet, and `internal_netmask_len` set to `32`, to indicate that we want to match the whole IP address. If there is no entry that completely matches the IP address, the LPM algorithm will look for entries with a shorter `internal_netmask_len`. If no entry matches, the lookup fails, otherwise the value corresponding to the selected entry is returned.

If the packet is coming from the outside, the `dp_rules` table is used. The lookup is performed with a key with `mask` set to `56`, because we want to obtain the rule that matches all the key parameters, if any. `external_ip` and `external_port` are set to the destination IP address and port number of the packet, and `proto` is set to the protocol of the packet. Since we perform the lookup with `mask = 56` and keys in the map can have `mask` set to 32, 48 or 56, we can be sure that priority is maintained based on the specificity of the rule, and that even if no `56-bit mask` is matched, we match other fallback entries, if any.

If no rule was found that matches the current packet, the next step is skipped and packet is forwarded to the opposite interface as-is, otherwise natting information is extracted from the lookup value.

Natting table update and translation

This step is executed only if either a natting table entry or a rule table entry matched the current packet.

If there is no natting table entry yet, the two natting entries are injected in the natting tables as previously described.

In any case, the natting information we gathered in the previous steps is used to modify the packet headers - both IP and Layer 4 or ICMP. The packet is then forwarded to the opposite interface.

4.2.5 eBPF-related notes

On a eBPF level, there are several differences in pcn-nat with respect to the NAT implementation of pcn-iptables.

Number of injectable rules

As we already discussed, the linear-search rule matching algorithm implemented in pcn-iptables has been replaced by a faster and easier direct-access lookup in an LPM map. This choice not only brings a performance improvement, but also avoids one limitation of eBPF, which is loop unrolling.

Without loop unrolling, the number of possible rules is virtually unlimited, bounded solely by the maximum size of maps in the Linux kernel.

One eBPF datapath

Moving to a direct lookup also made possible to write the entire NAT functionality in one datapath, instead of using three different submodules, which means that no tail call is needed, and this is another improvement in performance, because as much as efficient tail calls may be, they still add some overhead to the packet processing.

No datapath reloading

This new implementation of pcn-nat never requires the datapath to be reloaded and injected in the kernel. In pcn-iptables this happened every time a new rule was inserted, because the `_RULES` constant had to be updated in order to provide a functional loop unrolling.

In pcn-nat, rules are simply injected to the datapath maps from the control plane, without altering the code of the program in any way.

4.3 Open issues

With respect to pcn-iptables, where most of the issues derived from using the connection tracking table instead of having different data structure, this implementation of pcn-nat, considered the design that we wanted to achieve, has only one problem left, which is the same as in pcn-iptables: the port selection policy.

The analysis of the problem is not reported here since it can be found in Section 3.3.2.

Chapter 5

Results

Both *pcn-iptables* and *pcn-nat* are designed to be very efficient, at least on paper. Performance tests have been executed to measure how much better these solutions are with respect to *iptables* and *netfilter* in general.

For each service three types of tests have been run: *throughput tests*, *latency tests* and *stress tests*. Each test has been executed comparing two different packet processing techniques: TC and XDP.

Introduction to XDP

XDP - *eXpress Data Path* - is a programmable, high performance packet processor in the Linux networking data path [16] [17].

It provides an additional hook to be used with eBPF programs to intercept packets in the driver space of the network adapter, before they are manipulated by the Linux kernel.

The main advantage of this early processing is that it avoids the overhead and the memory consumption added by the kernel to create the *socket buffer* - namely the `skb` data structure - which wraps the packet for standard Linux processing in TC mode.

XDP runs in the lowest layer in the packet processing stack, as soon as the NIC driver realizes a packet has arrived: one of the main use cases is pre-stack processing for filtering or DDOS mitigation.

It is important to mark the difference between XDP and *kernel bypass*: XDP does not exclude the TCP/IP stack in the Linux kernel as it is designed to perform a preliminary packet processing to achieve better performance. Kernel bypass ignores the TCP/IP stack and performs all the packet processing on its own.

XDP typically provides a noticeable boost in performance in any use case, included the tests we are about to present. An important thing to notice is that *iptables* cannot run on XDP, whereas eBPF programs can.

Although this may seem like a biased comparison, it was performed anyway to

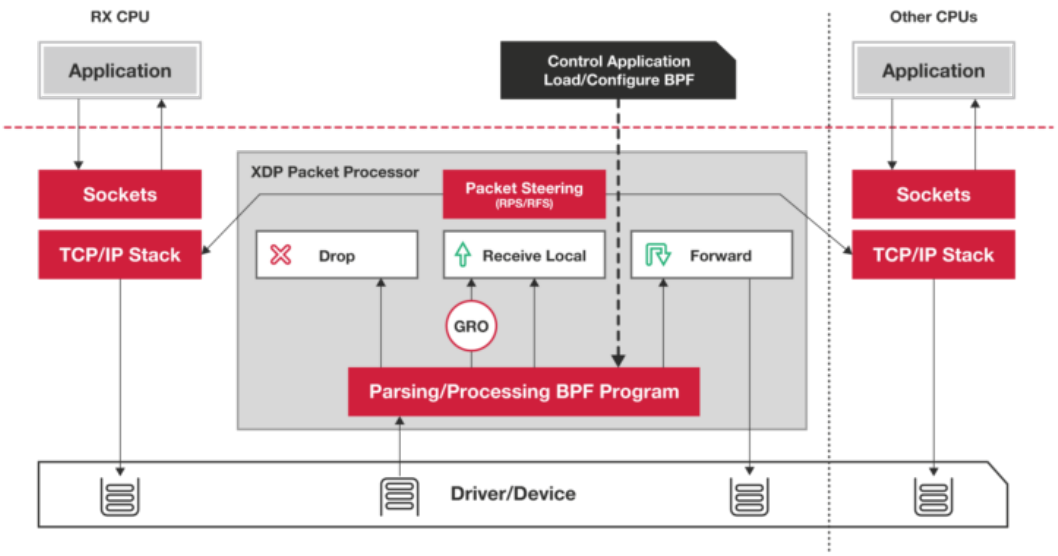


Figure 5.1: Overview of the XDP processing architecture

stress the fact that Polycube services offer this additional possibility while iptables does not.

5.1 Test configuration

To perform the presented tests two different machines were used, which we will refer with the following names:

- Server: the machine that runs Polycube. We will also refer to this machine as the Device Under Testing (DUT)
- Client: the machine from which tests are executed, also called the Generator (GEN)

Server machine

The server machine is a HP EliteDesk 800 G1 SFF.

- OS: Linux 16.04.5 LTS
- Kernel: 4.18
- CPU: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
- RAM: 16GB DIMM DDR3 Synchronous 1600 MHz

- NICs: two Intel 10-Gigabit X540-AT2
- Storage: Samsung SSD 850, 256GB

Client machine

The client machine is also a HP EliteDesk 800 G1 SFF.

- OS: Linux 16.04.5 LTS
- Kernel: 4.15
- CPU: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
- RAM: 32GB DIMM DDR3 Synchronous 1600 MHz
- NICs: two Intel 10-Gigabit X540-AT2
- Storage: Seagate ST1000DM003-1SB1, 1TB

5.2 Throughput tests

Raw throughput tests were performed using the *pktgen* tool [18]. *pktgen* is a software based traffic generator powered by the DPDK fast packet processing framework.

This tool is capable of generating **10Gbit wire rate traffic** with a minimum frame size of 64 bytes and it can act simultaneously as transmitter and receiver, therefore measuring how much traffic is correctly processed by the **Device Under Testing** - DUT - and sent back to receiving interface.

It can generate sequences of packets iterating over ranges of source and destination MAC addresses, IP addresses and Layer 4 port numbers, simulating an arbitrary number of traffic flows.

The configuration for a test execution can be provided by LUA scripts, which configure the packet rate, the size of the frame, the used addresses and interfaces, and the duration of the test.

The final results of the tests were obtained following the guidelines in RFC 2544 - Benchmarking Methodology for Network Interconnect Devices [19]: the throughput, measured in processed mega packets per second (Mpps), is considered valid when the percentage of lost packets is below 0.1%.

These tests are designed to measure performance in the **worst case scenario**: the selected frame size is 64 bytes, which maximizes the number of individual packets that must be processed by the DUT.

To test single-core and multi-core performance, *pcn-iptables* and *pcn-nat* were tested using 1, 128, 256, 1024, 2048, 4096 and 8192 traffic flows - which has been

possible by properly modifying the LUA configuration scripts: the 1-flow test measures the single-core performance, exploiting the kernel packet allocation algorithm which assigns each packet to one CPU core according to a hash function computed over the packet headers. The other packet flows test the multi-core performance and the capability of the eBPF modules to handle different numbers of simultaneous connections in their internal data structures.

To adhere to the guidelines of RFC 2544 the LUA script acts as following: it starts sending 64-bytes packets to 100% of the line rate and measures the packet loss. If this is greater than 0.1%, it halves the burst rate to 50% and repeats the test. If the packet loss is greater than 0.1% it halves the rate to 25%, otherwise it increases the rate to 75% and so on. The next rate is decided using a binary search algorithm which guarantees convergence in a finite number of steps.

Each iteration sends packet bursts of 10 seconds from the GEN TX interface and receives a percentage of them on the GEN RX interface.

The physical configuration is displayed in Figure 5.2.

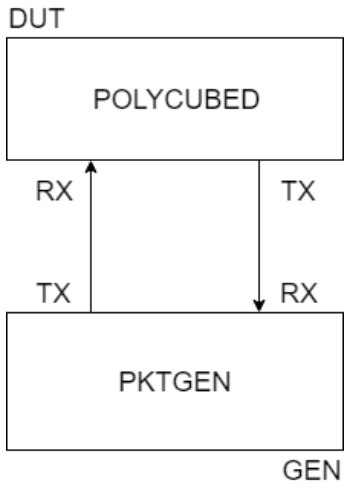


Figure 5.2: Configuration for the throughput tests

All the physical interfaces are 10Gbit NICs as stated in the test configurarion description in Section 5.1.

5.2.1 Testing pcn-iptables

pcn-iptables was tested to obtain an absolute value for the raw throughput and a comparison with the netfilter *iptables*. The pcn-iptables service on the DUT was configured to receive traffic from the DUT RX interface, process it and send it back on the DUT TX interface. Only two rules were set in iptables and pcn-iptables: one to enable packet forwarding and one to enable masquerade NAT on the outgoing interface. These rules were added using the following commands:

```
iptables -P FORWARD ACCEPT
iptables -t nat -A POSTROUTING -o DUT_TX -j MASQUERADE
```

Please note that these tests are not designed to measure the rule matching algorithm performance but only how the services behave with different numbers of connections. A logical view of the setup is reported in Figure 5.3.

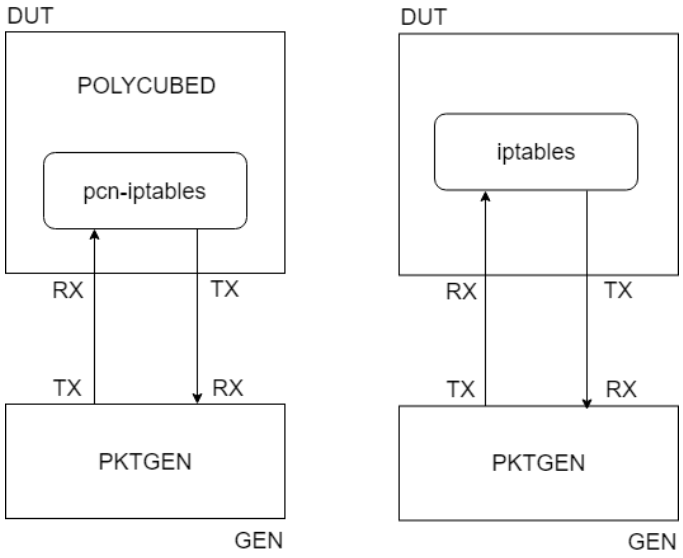


Figure 5.3: Comparison between iptables and pcn-iptables: setup

The test results are shown in the following Figure 5.4.

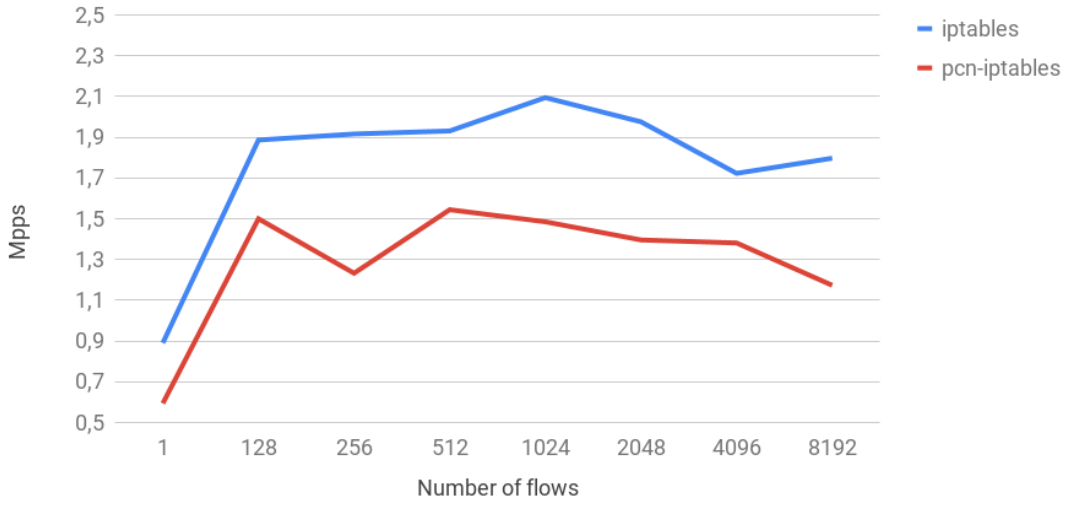


Figure 5.4: Comparison between iptables and pcn-iptables: results

pcn-iptables has worse performance with respect to *iptables* for any packet flow number. To understand why it is important to recall the issues presented in Section 3.3: the whole integrated connection tracking algorithm is complex and requires several map lookups to perform a complete NAT translation. The NAT module consists of three submodules linked by tail calls, which add an overhead, and the possible advantage of an optimized linear search for the rule matching algorithm is minimized by the fact that only one rule was set to perform these tests.

It is much likely that the design discussed in Section 3.1.1 is not the best solution to achieve NAT in pcn-iptables, and this causes the results to be sub-optimal.

5.2.2 Testing pcn-nat

pcn-nat is involved in two sets of tests: one considers the NAT module alone, while the other tries to mimic the minimum configuration necessary to achieve what iptables can do using a chain of Polycube services composed by a router and a NAT. iptables is always considered as a yardstick to compare our performance with real data. In all the reported tests the only injected rule is a masquerade rule that applies to all the traffic going out of the DUT TX interface. The command used to inject such rule is the following:

```
polycubectl nat1 rule masquerade enable
```

Only pcn-nat

The configuration deployed to test pcn-nat is shown in Figure 5.5, and the results are shown in Figure 5.6.

We start to notice some major improvement in performance: in this case the whole DUT is dedicated to execute *polycubed* and one instance of *pcn-nat*, which acts as a bridge between the two interfaces and translates whatever packet comes from the DUT RX interface and sends it out of the DUT TX interface.

pcn-nat in TC mode can process twice as much traffic as iptables, and the difference is just so much bigger if we look at what the NAT module can do when runned in XDP mode, where we reach a peak performance of ten millions packets per second.

This result was largely expected: pcn-nat is a module that only performs natting, it has no other task except for it, whereas iptables is bound to execute all its internal steps even if the actual task is to perform a NAT translation. This proves once again the advantage in using a modular, extensible architecture with respect to one monolithic software.

pcn-nat and pcn-router

This configuration implements the minimum set of features that are required to properly forward and apply NAT to a stream of packets, and it is intended to be

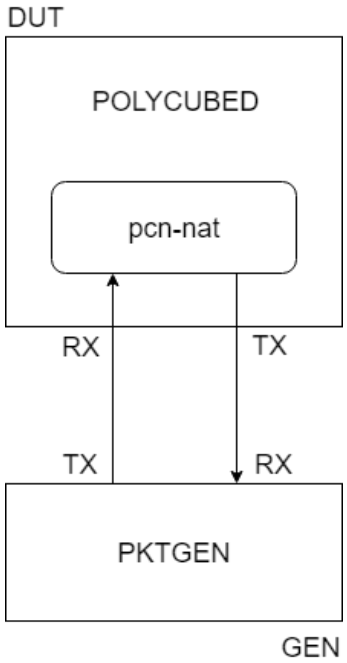


Figure 5.5: pcn-nat throughput test: configuration

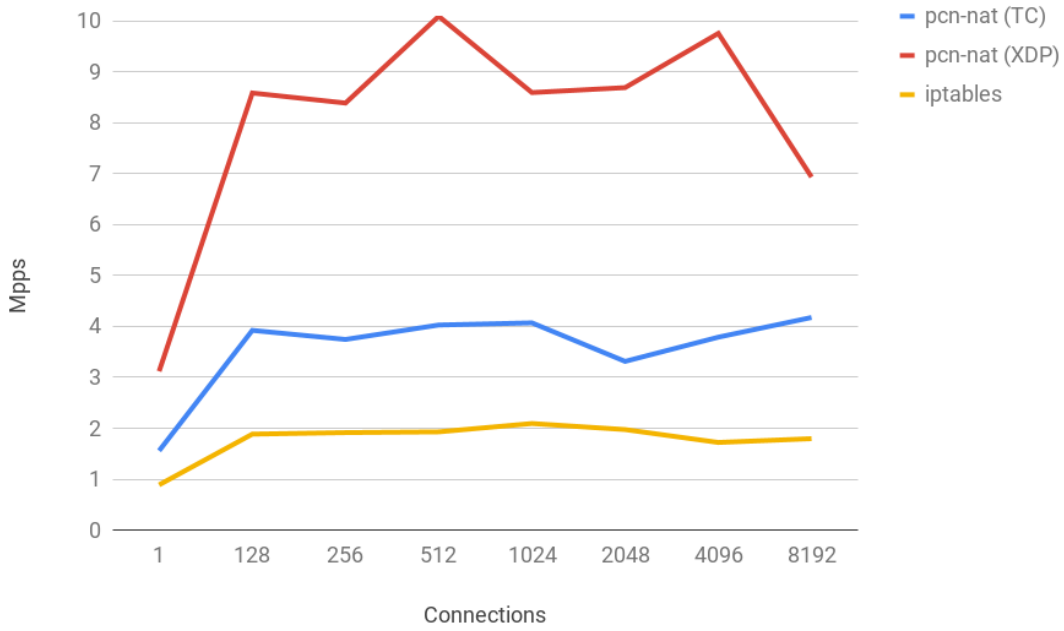


Figure 5.6: pcn-nat throughput test: results

compared with iptables, which performs forwarding in addition to natting. Let us say this is a more fair comparison with respect to the bare NAT module in the previous section.

Packets sent by the GEN TX interface will reach the DUT RX interface, pass through the router, then the NAT, which will finally change the packet headers to perform masquerade and redirect the packet on the DUT TX interface.

The setup needed to perform these tests is reported in the following Figure 5.7.

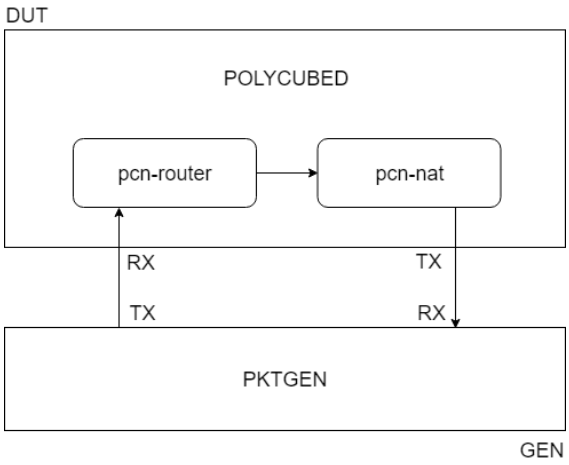


Figure 5.7: nat and router throughput test: configuration

As we can see in the following Figure 5.8, performance is still on our side.

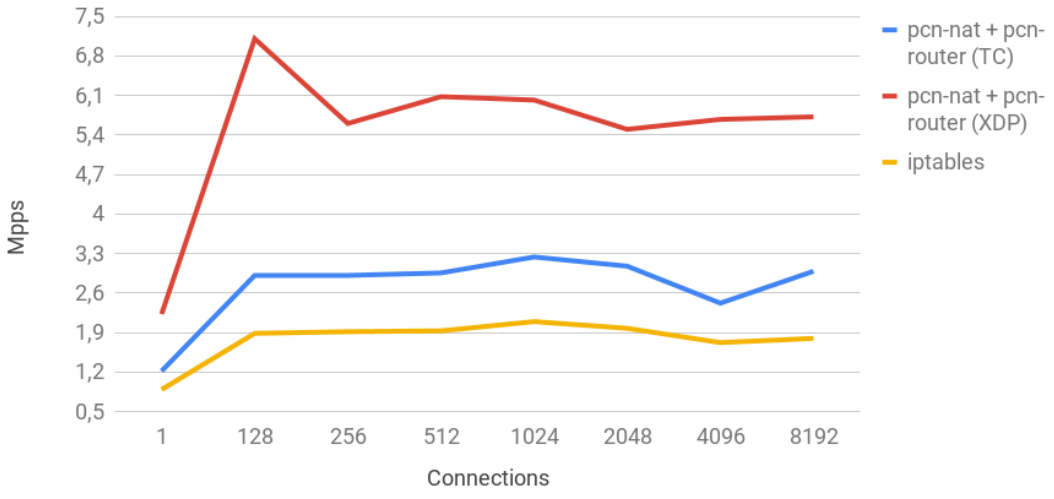


Figure 5.8: nat and router throughput test: results

The results are slightly worse with respect to the only NAT case, but this is

a direct consequence of the fact that each packet has to pass through two eBPF programs before going out of the DUT TX interface.

We can observe once again how much using XDP instead of TC impacts the overall performance of the system: the fact that iptables and netfilter-based programs in general cannot access such technology is an important advantage of using eBPF to write packet processing programs.

5.3 Stress tests

The *stress test* acts exactly like the throughput tests presented in Section 5.2 but with **one million different flows**. Such a high number of flows can never be tracked by the eBPF modules because their internal maps have a much lower number of entries, therefore this test measures the drop in performance which happens when the caching mechanisms - namely the *connection tracking table* for pcn-iptables and the *natting table* for pcn-nat - are never used and the programs must resort to a new rule lookup for any new flow.

The configurations used to perform these tests are the same reported in Figure 5.3, Figure 5.5, and Figure 5.7 for pcn-iptables, pcn-nat and pcn-nat plus pcn-router respectively.

The compared results for the configurations are reported in Figure 5.9.

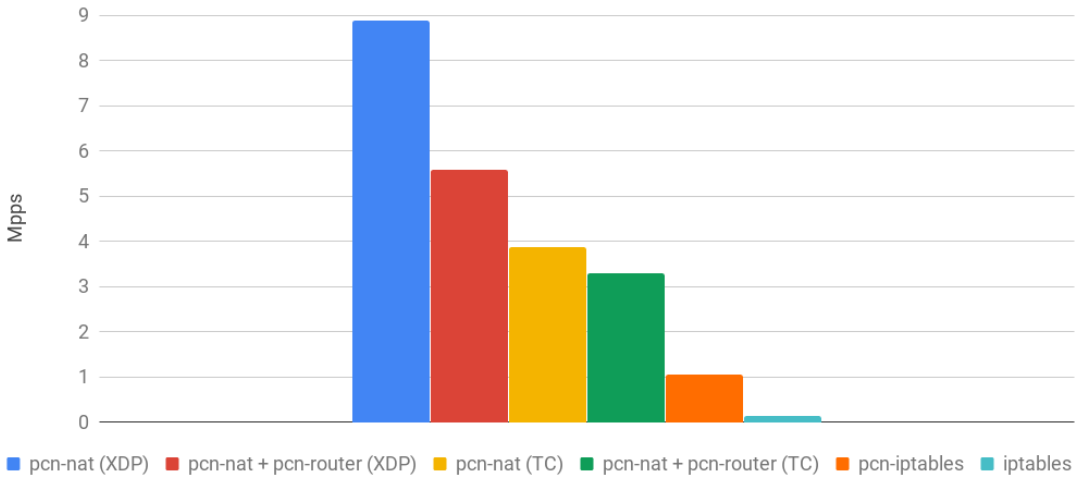


Figure 5.9: Stress test: result

In this test we can see a comeback for pcn-iptables, which outperforms iptables by roughly a factor of 10: in fact, the overall throughput of iptables is 0.14Mpps, whereas pcn-iptables manages to process 1.05Mpps. This value is just slightly less to the one obtained in the throughput tests for 8192 flows - 1.17Mpps - which

proves how the eBPF solution maintains a more steady behaviour when the number of flows increases.

As far as the other Polycube modules are concerned, the results are aligned to what we were expecting: pcn-nat performs better than the chain of pcn-nat and pcn-router, and the XDP versions outperform the TC versions in both cases.

Comparing these results to those in Figure 5.6 we can also notice that pcn-nat in XDP mode is barely affected by such a high number of flows, as it keeps processing up to 9Mpps.

5.4 Latency tests

Latency tests were performed using the *MoonGen* tool [20]. *MoonGen* is very similar to *pktgen*: they both rely on DPDK to generate up to 10Gbit wire rate traffic and can be configured with LUA scripts, but *MoonGen* also provides hardware timestamping capabilities on supported NICs, which allows to measure latency of a packet transmission on a sub-microsecond scale.

With respect to the throughput tests in Section 5.2, latency tests have been performed to measure the worst case scenario, that is single-core performance.

Being equivalent in nature although different in usage, the base configuration for the latency tests is the same as the one for the throughput tests shown in Figure 5.2, as well as the standard test configurations.

One new configuration has been deployed to provide a yardstick with respect to the latency of processing using a dummy eBPF program, *Forward*. This program implements the minimum functionalities to perform packet forwarding and it is only three lines long as reported in the following snippet.

```
static int handle_rx(struct CTXTYPE *ctx,
                     struct pkt_metadata *md) {
    uint8_t out_port = 0;
    if (md->in_port == 0) out_port = 1;
    return pcn_pkt_redirect(ctx, md, out_port);
}
```

This program suffers from no memory lookup overhead or complex operations and it is useful to compare the latency of a complex service with that of the most basic one.

Since we wanted to measure the latency excluding other delays such as buffering as much as possible, *MoonGen* was configured to send bursts of UDP packets at 100Mbps with respect to the theoretical 10Gbps allowed by the NICs.

A compared chart of the services configurations is reported in Figure 5.10.

Consider that with this test we are measuring how much a program execution lasts, but we are dealing with nanoseconds, therefore some fluctuation is inevitable.

In any case, this test is useful to demonstrate that the actual overhead added by executing an eBPF solution instead of a Netfilter one is basically negligible.

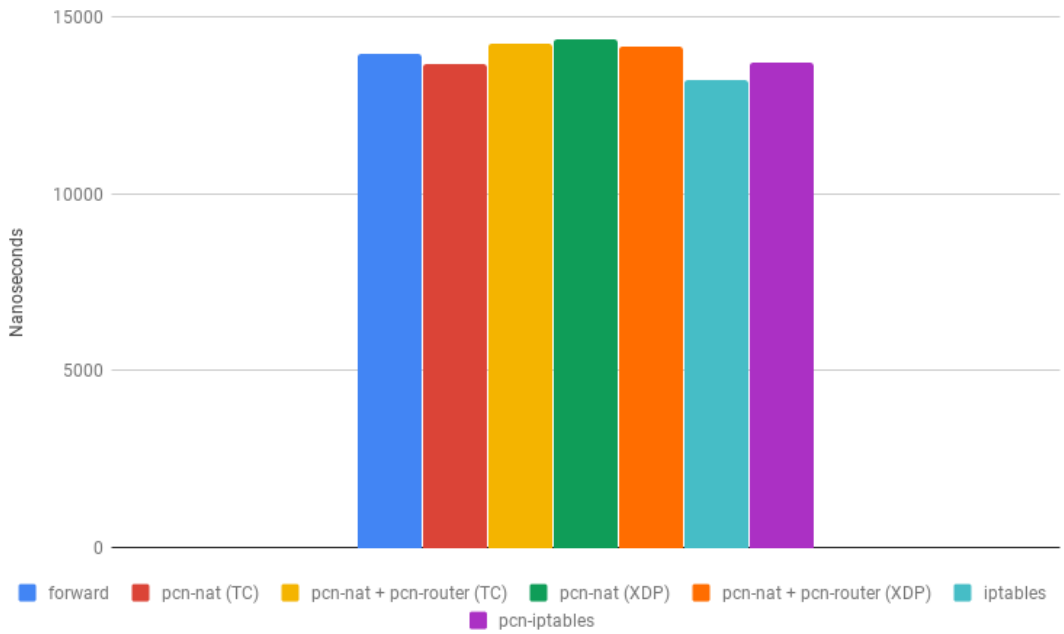


Figure 5.10: Latency test: result

Chapter 6

Conclusion

The subject of this thesis has been challenging for a number of reasons.

An operation so trivial as network address translation may seem presents many details and corner cases which were difficult to anticipate during the design of the proposed solutions.

The work on *pcn-iptables* required our solution to be seamlessly integrated into an already existing complex algorithm. This is the first conclusion we can draw: to design an implementation based on the *connection tracking table* was probably not the best choice. A viable solution would be to prefer an addition with respect to an integration, which means implementing NAT as an individual step, relying on its own data structures and algorithms, that can be enabled or disabled at will to guarantee the best performance in any situation.

As we saw in the results, the efficiency of this design choice proves that another way must be found: the average performance is lower than *iptables*, which is why this implementation will probably never see the light of the day. However, from a research point of view, the difficulties we had to face have been unvaluable.

In fact, the know-how and the experience gathered about network address translation during the development and testing stages of *pcn-iptables* proved fundamental when the discussion about *pcn-nat* first began: we knew what was working and what was not, and we built a tailored, easy and fast API, designed with efficiency in mind.

This is what made the outstanding results we previously showed possible: although we trimmed some of the flexibility of *iptables* here and there, the resulting service is powerful enough to serve most natting necessities, blazingly fast.

Besides the need to improve *pcn-iptables*, it is also key that a more effective solution is found to manage Layer 4 port numbers. This is part of a more general lack of integration between the Linux kernel and eBPF, which is bound to use predefined helper functions. These helpers have to be built in the kernel itself, which makes their addition and update slow and difficult.

Following this approach at least two new helpers should be created: the first to

obtain a valid port number, and the second one to reserve it when it is being used by the eBPF program.

In conclusion, the main goal this thesis achieved is to prove that network address translation in eBPF is not only possible, but also efficient and effective. It also proved that the Polycube framework, although in its early stages, is powerful enough to replace *netfilter* in many common use cases, guaranteeing scalability, performance and unparalleled ease of configuration.

The journey towards a new Linux networking has begun.

Appendices

Appendix A

NAT configuration with iptables

A.1 Source-NAT

A SNAT rule in iptables requires the outside address to be explicitly set in the command.

It is possible to specify address pools or ranges both for the inside and outside addresses. For pools the CIDR notation is used. For ranges, the first and last address of the range are specified.

```
iptables -t nat -A POSTROUTING
         -o eth0 -s 192.168.0.1-192.168.1.5
         -j SNAT --to-source 2.2.2.1-2.2.2.5
```

In case the number of inside addresses is greater than the number of outside addresses, the source port number has to be changed as well, to allow for multiplexing.

For example, consider the case where we have 3 routable addresses and an inside network with 3 IP subnets. We can specify a different outside address for each of them.

```
iptables -t nat -A POSTROUTING
         -s 10.0.2.0/24
         -j SNAT --to-source 2.2.2.2
iptables -t nat -A POSTROUTING
         -s 10.0.3.0/24
         -j SNAT --to-source 2.2.2.3
iptables -t nat -A POSTROUTING
         -s 10.0.4.0/24
         -j SNAT --to-source 2.2.2.4
```

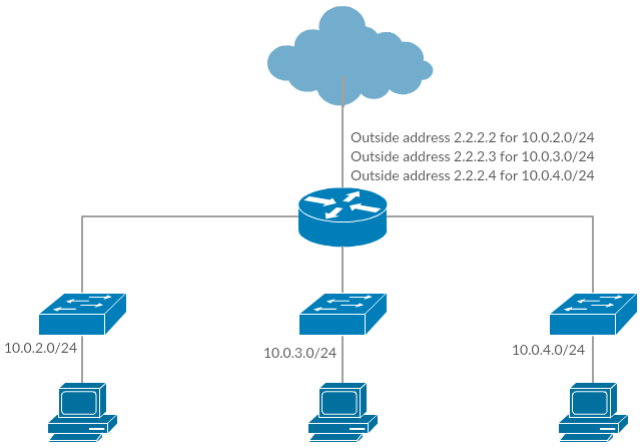


Figure A.1: Three networks with 3 publicly routable addresses

A.2 Masquerade

The syntax and the matching rule format is the same as SNAT, so we can decide to masquerade only a subset of addresses and specify static mappings for another subset.

```
iptables -t nat -A POSTROUTING
         -o eth0
         -j MASQUERADE
```

A.3 Destination NAT

In the following example, all packets that reach interface eth0 match the rule, but it is possible to specify IP addresses and other parameters as well [15].

```
iptables -t nat -A PREROUTING
         -i eth0
         -j DNAT --to-destination 10.0.0.1
```

Another example, to perform port forwarding.

```
iptables -t nat -A PREROUTING
         -p tcp -d 10.10.20.99 --dport 80
         -j DNAT --to-destination 10.10.14.2
```

A.4 Redirect

```
iptables -t nat -A PREROUTING
```

```
-i eth0 -p tcp --dport 80
-j REDIRECT --to-ports 8080 --to-destination 10.0.0.1
```

Appendix B

Sample pcn-iptables rules

B.1 Source NAT

Source NAT configuration in pcn-iptables allows to specify all the parameters that appear in the following command.

```
iptables -t nat -A POSTROUTING
     -o <interface>
     -s <ip address> -d <ip address>
     --sport <port number> --dport <port number>
     -p <protocol>
     -j SNAT --to-source <ip>
```

B.1.1 Output interface and source IP address

```
iptables -t nat -A POSTROUTING
         -o <interface> -s <ip address>
         -j SNAT --to-source <ip>
```

B.1.2 Output interface

This rule is equivalent to a masquerade rule with a static IP address.

```
iptables -t nat -A POSTROUTING
         -o eth0
         -j SNAT --to-source 130.192.1.1
```

B.1.3 One to one mapping

```
iptables -t nat -A POSTROUTING
         -s 192.168.1.1
```

```
        -j SNAT --to-source 130.192.1.1
```

B.1.4 Source IP address with CIDR notation

```
iptables -t nat -A POSTROUTING
        -o eth0 -s 192.168.1.0/24
        -j SNAT --to-source 130.192.1.1
```

B.2 Destination NAT

Destination NAT configuration in pcn-iptables allows to specify all the parameters that appear in the following command.

```
iptables -t nat -A PREROUTING
     -i <interface>
     -s <ip address> -d <ip address>
     --sport <port number> --dport <port number>
     -p <protocol>
     -j DNAT --to-destination <ip>
```

B.2.1 Input interface and destination IP address

```
iptables -t nat -A POSTROUTING
        -i <interface> -d <ip address>
        -j DNAT --to-destination <ip>
```

B.2.2 One to one natting

```
iptables -t nat -A PREROUTING
        -i eth0 -d 130.192.1.1
        -j DNAT --to-destination 192.168.1.1
```

B.2.3 Protocol, destination IP address and destination port

```
iptables -t nat -A PREROUTING
        -p tcp -d 130.192.1.1 --dport 80
        -j DNAT --to-destination 192.168.1.1
```

B.2.4 Protocol, destination IP address and destination port with port redirection

```
iptables -t nat -A PREROUTING
```

```
        -p tcp -d 130.192.1.1 --dport 80
        -j DNAT --to-destination 192.168.1.1 --to-dport 8080
```

B.3 Masquerade

Masquerade configuration in pcn-iptables allows to specify all the parameters that appear in the following command.

```
iptables -t nat -A POSTROUTING
     -o <interface>
     -s <ip address> -d <ip address>
     --sport <port number> --dport <port number>
     -p <protocol>
     -j MASQUERADE
```

B.3.1 Output interface

```
iptables -t nat -A POSTROUTING
         -o <interface>
         -j MASQUERADE
```

B.3.2 Output interface and source IP address with CIDR notation

```
iptables -t nat -A POSTROUTING
         -o eth0 -s 192.168.1.0/24
         -j MASQUERADE
```

Appendix C

Sample pcn-nat rules

C.1 Source NAT

Source NAT configuration in pcn-nat allows to specify all the parameters that appear in the following command.

```
polycubectl nat1 rule snat append
            internal-net=10.0.0.0/24 external-ip=130.192.1.1
```

C.2 Destination NAT

Destination NAT configuration in pcn-nat allows to specify all the parameters that appear in the following command.

```
polycubectl nat1 rule dnat append
            external-ip=130.192.1.1 internal-ip=10.0.0.1
```

C.3 Masquerade

Masquerade in pcn-nat can be enabled and disabled with the following commands.

```
polycubect nat1 rule masquerade enable
```

C.4 Port Forwarding

Port forwarding configuration in pcn-nat allows to specify all the parameters that appear in the following command.

```
polycubectl nat1 rule port-forwarding append
            external-ip=130.192.1.1 external-port=80
```

```
internal-ip=10.0.0.1 internal-port=8080
proto=tcp
```

C.5 Deleting rules

It is possible to delete all rules together, all the rules of the same type together, or single rules.
To delete all rules:

```
polycubectl nat1 rule del
```

To delete all rules of a type (SNAT in the example):

```
polycubectl nat1 rule snat del
```

To delete a single rule (an SNAT rule in the example):

```
polycubectl nat1 rule snat del RULE_ID
```

Bibliography

[1] IP Network Address Translator (NAT) Terminology and Considerations
https://tools.ietf.org/html/rfc2663
[2] Basic NAT
https://tools.ietf.org/html/rfc2663#section-4.1.1
[3] Network Address Port Translation (NAPT)
https://tools.ietf.org/html/rfc2663#section-4.1.2
[4] Bi-directional NAT (or) Two-Way NAT
https://tools.ietf.org/html/rfc2663#section-4.2
[5] Netfilter
https://en.wikipedia.org/wiki/Netfilter
[6] A Deep Dive into Iptables and Netfilter Architecture
https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-architecture
[7] iptables
https://en.wikipedia.org/wiki/Iptables
[8] NAT in iptables
https://www.karlrupp.net/en/computer/nat_tutorial
[9] eBPF - extended Berkeley Packet Filter
https://prototype-kernel.readthedocs.io/en/latest/bpf/
[10] Exploring eBPF, IO Visor and Beyond
https://www.iovisor.org/blog/2016/04/12/exploring-ebpf-io-visor-and-beyond
[11] eBPF, part 1: Past, Present, and Future
https://ferrisellis.com/posts/ebpf_past_present_future/
[12] eBPF maps
https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html
[13] Types of eBPF maps
https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps_types.html
[14] Polycube
https://github.com/netgroup-polito/polycube
[15] Destination NAT with netfilter
http://linux-ip.net/html/nat-dnat.html

[16] eXpress Data Path - Tom herbert and Alexei Starovoitov
https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf
[17] Introduction to XDP - The IOVisor Project
https://www.iovisor.org/technology/xdp
[18] The Pktgen Application
https://pktgen-dpdk.readthedocs.io/en/latest/index.html
[19] Benchmarking Methodology for Network Interconnect Devices
https://tools.ietf.org/html/rfc2544
[20] MoonGen Packet Generator
https://github.com/emmericp/MoonGen

Acknowledgements

Here I am, writing the part of the thesis which is supposed to be the easiest but is actually the most difficult one.

There are just so many people which made this achievement possible, I will try to mention them all.

I will start with my supervisor professor Fulvio Risso and the doctorate candidate Matteo Bertrone with which I had the pleasure to work for the past six months. With their passion and relentless assistance they inspired me not to be content of any solution, but to challenge myself to find the best one.

If I got to write this notes after countless years of study it is also because I had amazing people around me: my friends. My friends in Turin, those in Pisa and those in Farnese before that are the fuel that keeps me going and the reason why I never gave up even when it seemed the easiest way out of problems. Thanks to you I know there are so many places I can call *home* and that I will always have someone to freak out with when things get messy.

The fact that I left my family for last is not about priority, it's just that they are the people I have to thank most. They know me, they know how difficult I can be but here they are, supporting me in everything I do. They were there when I decided to move to Pisa for my bachelor's degree, they were there when I moved to Turin, and they will be there pretty much wherever I will go next. I am who I am and I could do what I have done thanks to them.

Thank you all.