# Chapter 1

# Introduction

## 1.1 Motivation

The network connections in Dresdner residential homes currently suffers from a simple but substantial problem: only IPv4 addresses are provided and there are only enough addresses to assign one to each student. The fact that everybody owns more than one IP capable device today leads to numerous problems: every student that uses more than one device needs to run his or her own router to perform NAT. This is bad in multiple ways. First of all, every student needs to buy a router. Second, every one of these routers creates an own wireless network. This leads to a number of networks as high as residents in a building. Each of those wireless LANs needs air time for beacons, since WLAN is a shared medium[28]. The number of channels is limited, too. One goal of the project is to allow centralized WLAN - currently this is not possible due to the limited number of IPs. Third, the performance of low-cost routers is mostly bad (e.g. slow WLAN, slow NAT). A centralized approach can address these issues, since the hardware will be sized properly.

## 1.2 Features

### 1.2.1 Private Networks

Every user of the network are going to have an private network (also referred to as “home network”). The term “network” refers to a VLAN according to IEEE 802.1Q[1]. This network will be located at the router next to the users flat. If the user tries to connect to the network via LAN or WLAN inside his building, he will be placed in this network. This implies that all private networks are distinct only inside the building, not in a global context. The IPs assigned inside an private network are going to be RFC6598 addresses[2], in practice an /24 network out of the Carrier-Grade NAT (CGN) range of 100.64.0.0/10. Every private network will be mapped via network address translation (NAT)[14] to

an external IP address on the NAT gateway. Every external IP is only used by one user.

### 1.2.2 Roaming Networks

All buildings get an building specific network (VLAN). A user without an home network in the building he connects to the network is placed inside the local roaming network. These networks will contain an subnet from the CGN allocation. Per default, connections from roaming networks are not translated to external IP addresses. As a result, no connectivity to addresses outside of the internal network is possible. During the authorization of a client in an roaming network, a rule is created on the NAT gateways. This rule translates connections to the assigned private IP to the public IP of the user. Another approach to solve these roaming issues are IP mobility as described in RFC3344[7]. This alternative approach and its ups and downs are described in the next chapter.

### 1.2.3 Rate Limiting

Due to an agreement with the university the network connection for individual users must be limited, if a certain amount of traffic is exceeded. Instead of blocking the access to the network for those users, rate limiting will be performed. Even if a user is affected by rate limiting, it should be possible to establish several connections that are not rate limited. The destinations to be excluded from the limiting shall be configurable.

### 1.2.4 Failover

The setup should contain multiple NAT gateways to allow failover. If one gateway fails, the network should stay operational. To achieve failover, the connection state has to be synchronized. The tool to synchronize connection tracking state between multiple hosts on linux are the conntrack-tools[5]. Besides the connection tracking state the availability of the gateways is ensured by using keepalived[10], a VRRP daemon for Linux. For failover, multiple setups are possible.

**Active - Passive** Only one gateway is active the other one is passive and becomes active if the other gateway fails. This setup is easy to realize but wastes resources, because one gateway is idling.

**Active - Active (Synchronous Routing)** The internal hosts get split into two sets and each gateway gets to perform NAT for one set. So both gateways are active but can be used as backup for the other gateway. The load balancing in this case is static and can be performed with policy based routing. Both directions of a connection get translated by the same gateway (Synchronous Routing).

**Active - Active (Asynchronous Routing)** This is the most complex setup, because it is not clear which gateway performs the translation. This is why the connection tracking state has to be synchronized continuously and with a very low latency. Example: A request by an internal host is translated by gateway X and the response is translated by gateway Y. During the time that the response needs to arrive the connection tracking state has to be synchronized otherwise the packets are rejected. This scenario allows dynamic load balancing without knowing to which connection a package belongs.

### 1.2.5 Port Forwarding

The NAT prevents users to use remotely initiated connections. To address this issue, port forwardings should be configurable. In detail, the NAT box should allow incoming connections to the external ("public") IPs and forward them to an configured internal host inside a private network. Related to port forwarding, a feature set comparable to publicly sold routers should be available. Popular examples are TP-Link routers (e.g. TL-WR841ND[26]) and AVM routers (e.g. Fritzbox 4020[6]), which allow the user to configure rules. Criteria for these rules are the inbound (external) port and the protocol. The redirection is possible to an configurable internal IP and port.

### 1.2.6 Internal API

To setup the NAT gateway, an interface to the configuration tools in place is required. This interface should be accessible by two configuration tools that are already present: the user management tool and the user self-service tool. In addition, the DHCP server for roaming networks needs to be able to perform updates on the NAT gateway. These other parts of the infrastructure are not part of this project, but the interface of the gateway should be designed on their behalf.

# Chapter 2

# Basics

## 2.1 IP Mobility

IP mobility[7][11] uses a complex setup to allow roaming users to retain their IP address in a home network. An endpoint in the home network of the user - called the home agent - forwards traffic for the user to his real point of presence in the network, the so-called foreign agent. The foreign agent establishes a tunnel to the home agent, using its own IP as a care-of address. All traffic for the client IP in the home network can now be forwarded in a tunnel to the foreign agents IP, which is called care-off address. The client itself is bound to an address of a foreign subnet.

### 2.1.1 Pro

- No dynamical NAT configuration
- L2 connectivity to the home network (e.g. printers)

**Con**

- Home- and foreign agents for every network necessary
- traffic is encapsulated (overhead)

### 2.1.2 Aruba IP Mobility

- only available where centralized WLAN is available
- costly
- needs verification
- may be in concurrent use with implementation

## 2.2 Netfilter Hooks

The netfilter hooks[12] allow to tap into the network traffic on a linux machine in specific places in the network stack. The different hooks allow to change the network traffic in specific stages, making it flexible how network traffic is filtered, changed and analysed. Figure {} shows an overview of the different hooks an how traffic flows through the stack. The important hooks for NAT are PREROUTING and POSTROUTING. For source NAT the source IP has to be changed after the routing, because based on the routing decision the IP has to be changed (For example you have 2 different networks that you want to use NAT for). Destination NAT is just the opposite and requires to be performed before the routing, otherwise its not clear what the actual destination is and how it has to be routed.

## 2.3 Nomenclature

**Network Address Translation (NAT)** Network Address Translation is the process of exchanging the address information of a network packet. It is often needed because there are not enough public ip addresses for all hosts of a network. A private network is used for those host and the internal ip address gets translated by a NAT gateway before the packet leaves the network. The gateway tracks the connection of each host in the network to the public network and thereby which internal host needs to receive the incoming traffic from the public network.

**Source NAT (SNAT)** The source ip address gets exchanged by the gateway, this is the normal NAT mode that is used by consumer routers.

**Destination NAT (DNAT)** Destination NAT is needed if an internal hosts should be available on the public network. When the gateway receives traffic from the public network on a specific port it forwards this traffic to the host (usually known as port forwarding in customer routers). So the destination address of a packet gets changed without the internal host creating a connection first.

**CIDR**

# Chapter 3

# Management Component

## 3.1 Structure

The management component is a python3 celery application. Celery is a framework to allow asynchronous task queue execution[3]. It is used in the project to enable management components in the network to trigger updates at the gateway. Updates are triggered by a rabbitmq message broker[25]. The configuration of the application can be found in the appendix.

## 3.2 Data Model

### 3.2.1 Translations

Translations are used to configure NAT mappings. They contain the public IP and the private subnet. The private subnets do not have a fixed size. This design allows single forwarding entries for roaming users with private subnets as small as a singe host (/32 in CIDR notation). Home networks with an expected size of 254 hosts (/24 in CIDR notation) can be represented in these objects, too.

### 3.2.2 Forwardings

These objects are used for DNAT to allow incoming connections to users. They containthe following data: public IP, private IP, protocol, source port and destination port.

### 3.2.3 Throttles

Throttle objects are used to describe limited connections. They contain the private network or host, the public IP and the limiting speed in kbytes/second.

Public and private information are both required, because inbound and outbound connections need to be limited. The speed limit is shared for inbound and outbound connections, the same queue is used.

## 3.3 Program State

The application supports two ways to update its configuration.

At first, a task named initialization is possible. It is called after the start of the application. During the initialization, the first configured database is queried to generate a complete ruleset. The conntrack state of the NAT gateway for the configured private network is dropped. The NFT configuration on all relevant tables is dropped and the new state is applied. The initialization process can later be utilized to reset the gateway. This task generates a file similar to a full nft listing and applies it afterwards.

Second, incremental state updates are possible. They are triggered by the message broker with parameters to exactly address the required change. During an update, the relevant database is queried for the data. The content of the change is not communicated via the message broker but by the database, so the broker only signals the application that an update needs to by applied.

## 3.4 State Updates

As mentioned above, the application queries the database for the concrete action to perform during a signalized state update. Two scenarios are possible: the data may or may not be present in the database. If there is no relevant data in the database, the application assumes that the represented data (translation, forwarding or throttle) has to be dropped from the applied nftables configuration. Then it performs the necessary tasks to delete the entries. This may include dropping conntrack state information. If the relevant data is present in the database, the application updates nftables. This may result in a newly created rule or in an updated rule, depending on the object in question and the current nftables state. For example, translations allow direct rule replacements. An already present rule may be updated with the “nft replace rule” command.

# Chapter 4

# Implementation

## 4.1 packet filter

Packet filter (pf)[20][21] is a firewall for the BSD-Family and was original written for OpenBSD. Besides filtering pf can also perform NAT and bandwidth control. Due to time issues we did not implement a solution with pf but evaluated some basic constrains. Instead of using OpenBSD we would recommend using FreeBSD, because the kernel of FreeBSD can use a multi core processor. Most of the heavy lifting for NAT is done in the kernel (like the connection tracking), so this feature is very important. Also the Solarflare network cards of our gateways are only supported on FreeBSD. Like the other solution pf keeps track of connections and only traverses the NAT rule set if the packet does not belong to an existing connection. For redundancy pf uses pfsync[23], which allows the synchronization of state tables entries between multiple pf instances.

## 4.2 pfSense

An related project is pfSense[22], a firewall distribution based on OpenBSD and PF. The software includes an webinterface, is actively maintained and has an online community. Sadly, it's not possible for us to use this existing project, since it does not provide an API to do remote configuration. This feature is panned for the 3er release of pfSense, but it is unknown when it is going to be released.

## 4.3 Iptables

iptables[9] is the defacto standard for network packet manipulation on Linux. It is a user-space application that allows the configuration of the packet filter ruleset in the Linux kernel. Besides filtering it can be used to perform NAT. There are different implementation of iptables for different protocols like IPv6

and ARP. Iptables is has 5 standard tables: filter, nat, mangle, raw and security. Filter is the default table and is used for filtering unwanted packets, it has the INPUT, FORWARD and OUTPUT hooks registered. The nat table is for performing NAT, packets pass this table only when a new connection is created. It has built-in the PREROUTING, OUTPUT and POSTROUTING hooks. To alter packets the mangle table is used, raw is for configuring exemptions to the connection tracking and security is used for Mandatory Access Control. Besides the standard tables new tables can be added to perform specific tasks. Tables contain a set of chains which hold the rules. Rules match traffic and than perform a specific task, like NAT or jumping to another chain.

## 4.4 Nftables

nftables[15][17][16] is a new packet classification framework for Linux that aims to replace iptables. It was first introduced on the netfilter workshop 2008 and got is first official release in 2009. Besides iptables it also wants to replace ip6tables, ebtables and arptables in one simple tool.

The motivation behind this new tool is that iptables is old and has many problems like the representation of the ruleset as huge blob, which makes replacement of single rules not possible. The ruleset is not very memory efficient and can not be translated back to its rules. Another mayor drawback is that iptables, ip6tables, ebtables and arptables share a large codebase but are not the same tool, which makes the code management inefficient. nftables also comes with an easier syntax and allows multiple targets per rule, making rulesets less redundant. It is also possible to exchange single rules in a ruleset without reloading the entire table.

nftables consists of three main components: the kernel modules, nft the nftables user-space application and libl the netfilter netlink library (communication with the kernel).

The basic container in nftables is a table, it contains rules and chains of the same protocol family. Chains are the containers for rules and can be used as jump targets. A chain that registers with a netfilter hook is called a base chain and serves as entry point into a table. Rules are container of expressions, which define the runtime behavior and are the smallest unit that can be replaced. Besides these basic building blocks nftables allows the usage a sets and maps. They make it easy to specify rules without much redundancy working for the entire used set. Especially maps are a useful addition for NAT, because they allow to directly specify NAT mappings. There are two different implementation for sets, as rbtree or as hashmap. nftables internally decides which one to use.

## 4.5 Conntrack / Conntrackd

Conntrack[5] is a user-space application that provides access to the connection tracking module[13] of the Linux kernel. The sole use of the connection tracking

module is to store information about connections that pass the machine. Each connection is identified by the source and destination IP addresses, the port numbers, and protocol type. The connection tracking module stores information about the state of the connection and is vital for NAT, because it allows to match incoming traffic from the internet to internal hosts.

After the initial NAT through nftables, or iptables a conntrack gets created in the connection table, containing the new source ip address. All following packets of the same connection get directly translated with the conntrack and do not pass through the nat table.

Example conntrack:

```
icmp 1 4 src=100.64.0.10 dst=192.168.30.0 type=8  code=0 id=7224
src=192.168.30.0 dst=192.168.0.19 type=0 code=0 id=7224 mark=0 use=1
```

The first src, dst matches the original traffic from the internal host, the second pair matches the response from the host on the other side of the gateway. The conntrack contains all the information needed to perform the NAT.

Conntracks can have different states:

- NEW: The connection is starting, to reach this the state the packets only have to be valid, no reply packets have been received yet.
- ESTABLISHED: The connection is established, this means the gateway has seen replies to the connection.
- RELATED: An expected connection, that is used for some special protocols like FTP. They are created by so called helpers that can extract information out of a connection that lets you expect another connection in the future. For example the control flow for FTP is done over port 20 but the data connection is done with a different port that is send to the client by the server. The ftp helper extracts that port number and sets up an expected connection for that port, allowing this connection to pass the NAT gateway.
- INVALID: The expected behavior of a connections is not followed.

The Kernel modules uses a hash table for efficient lookups, that holds 2 hash values for every connection: One for the original direction and one for the reply direction. The hash contains all necessary information to match the connection like the source and destination ip address. The two hash values point to the conntrack which stores the state information of the connection. Another effect of the connection tracking is that when the NAT rules are changed they only get applied to new connections not to the old ones in the connection tracking table. To change all connections to the new ip address the old conntracks need to be dropped.

Besides the kernel module for conntrack and user-space tool there is the daemon conntrackd. The daemon allows the state table synchronization between different NAT gateways. Through the state synchronization it is possible to

setup fault tolerant NAT gateways that do not lose connections in the case of an failure. In our setup an extra link between the gateways is used to replicate the state with a low-latency. This should allow us to use asynchronous routing with our gateways. Meaning that a connection can be translated by one gateway in one direction and translated back by the other gateway in the other direction. We made tests that suggest that this setup works, but we have to wait for going live to see if it actually works. Should it not work we have to make sure that a connection passes the same gateway in both directions (for example by splitting up the ip addresses for the gateways).

The maximal size of the connection tracking table can be adjusted with the variable "net.ipv4.netfilter.ip_conntrack_max" and should be increased from the default value. Conntrackd also has an option where the maximum number of conntracks can be specified.

## 4.6 Keepalived

Keepalived[10] is a Virtual Router Redundancy Protocol (VRRP)[27] deamon for Linux. Its main goal is to provide simple loadbalancing and high-availability for routers and servers running Linux. VRRP allows to configure one virtual router with an IP address using a set of hardware routers. The virtual IP gets assigned to a specific hardware router by the VRRP, making this router the VRRP master. The others routers function as backup routers for the master. If the master fails the VRRP assigns the IP to a new hardware router. This functionality allows easy fail over for routers and servers.

In our setup both gateways are active at the same time, making it necessary to configure two virtual gateways with each gateway functioning as backup for the other one. We have added the config that we used for our evaluation scenario in the appendix.

# Chapter 5

# Setup

## 5.1 Live-Setup

The central router in our network is called Ianus and is connected to our uplink. Ianus consists of two hardware routers located at different locations. To ensure availability one NAT gateway will be located at each of the locations. Each gateway will be connected to both hardware routers of ianus. This ensures availability in case that one gateway, or one hardware router fails.

All outgoing user traffic will be send to the gateways and from the gateways back to ianus. Through this loop setup its very easy to not route traffic over the gateways that should not be translated (our servers).

## 5.2 Gateways

The general specifications for our two NAT gateways

Server (Dell PowerEdge R630 Server): CPU: 2x Intel Xeon E5-2690 v4 (2,6 GHz, 14 Cores/ 28 Threads) RAM: 2x 16GB RDIMM 2.400MT/s first NIC: Intel Ethernet i350 4 Ports 1GbE second NIC: 2x Solarflare Flareon Ultra SFN8522 10 Gigabit SFP+ harddrive: 200GB SSD Operation System: Debian stretch Kernel >= 4.8 nft: > 0.7 (build from the git repo)

# Chapter 6

# Evaluation

## 6.1 Introduction

To check if the NAT gateway is capable of servicing all our users we relied on performance/stress testing to see what the gateway is capable of. The first challenge in this was to generate adequate network traffic for testing. Network traffic can vary different characteristics: used protocols, bandwidth, packet amount, packet length and connections.

A simple test with iPerf[8] revealed that pure bandwidth had little to none effect on the NAT gateway. Another test with a few million packets per second showed the same. Only when we started using more connections (different source ip destination ip pairs) the cpu usage on the NAT gateway went up. This is because only the first packet of a network connection traverses the NAT rules in nftables, or iptables. All the following packets are directly processed by the connection tracking system of the kernel. This is why our stress testing relies on creating as many connections as possible.

The problem with connections is that they are not as easy generated as lots of packets or bandwidth. Mostly because there needs to be an response to a request that traverses the NAT gateway in the opposite direction. This makes techniques like IP-spoofing impractical. All used IP addresses need to be configured on the corresponding machines (now is a good time to increase the size of your ARP cache).

## 6.2 Setup

In {} you can see our setup. We used one virtual machine to simulate hosts in "The Internet" (192.168.0.0/16) and one for our home networks in the Carrier-grade NAT (CGN)(100.64.0.0/12). For "The Internet" the NAT gateways have all public IP addresses configured that are used for NAT (of course not both gateways at the same time). In the Carrier-grade NAT each gateway only needs

one IP. Between the two gateways there is a link for the state synchronization of the conntrack tables.

An internal host, simulated by the "Home Networks" vm, sends a packet over one of the gateways to a internet host. After the first packet has traversed the NAT rules a conntrack is created and synchronized with the second gateway. The internet host receives the packet and sends a response via the gateway that has the corresponding ip address configured. The gateway than forwards the message to the internal host.

## 6.3 Generating Traffic

After some initial testing with packet generation tools and python we had problems generating network traffic that puts much load on the machines. The key to generating a good workload is to generate lots of different connections, meaning lots of different source ip, destination ip pairs. To generate them easily we found the unix tool nping[19] very helpful. nping is part of nmap[18] and supports multiple target ips (up to 8900) and gives you a lot of control about how the packets are generated. It supports TCP, UDP and ICMP packet generation and can use a specified source IP address, port and interface. The payload of the nping packets can be specified as well. This feature set makes nping an excellent tool for our use case.

To generate more connections we started multiple instances of nping with different source ip addresses. We will include the used python script in the appendix.

## 6.4 Metrics

The metrics we are interested in are latency, resource consumption on the gateway and the synchronization of conntracks. All of them together should help us to find bottle necks and identify problems of the gateways.

The latency is interesting because it should be as low as possible to not harm the hosts that are behind the NAT. We measured the latency by using ping[24] from the internal hosts VM to the Internet Hosts VM. To measure the resource consumption we collectl[4] a unix tool that can measure various resources on a pc. We used it to measure the cpu utilization, network traffic and interrupts on the gateways. The polling interval was one second. For the synchronization of the state tables we used "conntrackd -s" and polled this statistic every one second with a script. An other metric we looked at was the memory consumption, but we could not find anything interesting here, so we excluded it.

## 6.5 Results

Coming soon to an email server of your choosing...

# Chapter 7

# Appendix

## 7.1 keepalived.conf

```
global_defs {
    # Name of the local router
        router_id       chomsky

        # The email server
        smtp_server     192.168.0.30

    # Where to send notifications if a machine is down
        notification_email {
                test@test.de
        }
}


# Groups of vrrp_instances that failover together, allows to call
# scripts and send notifications
vrrp_sync_group agdsn_nat_1 {
        group {
                cgn_1
                ext_1
        }

        smtp_alert
        global_tracking
}

# Describes movebale IPs, sets the priority for the machine
vrrp_instance cgn_1 {
```

```
        state           BACKUP

        interface       eth7

        virtual_router_id 10

        priority        150

        virtual_ipaddress {
                100.64.0.1/12
        }
}

vrrp_instance cgn_2 {
        state           BACKUP

        interface       eth7

        virtual_router_id 11

        priority        100

        virtual_ipaddress {
                100.64.0.2/12
        }
}

vrrp_instance ext_1 {
        state           BACKUP

        interface       eth6

        virtual_router_id 12

        priority        150

        virtual_ipaddress {
                192.168.0.1/16
        }
}

vrrp_instance ext_2 {
        state           BACKUP

        interface       eth6
```

```
            virtual_router_id 13

            priority        100

            virtual_ipaddress {
                    192.168.0.2/16
            }
    }
```

## 7.2 nping

### 7.2.1 Example usage

```
Example usage:
nping -c 0 --tcp -p 80-1080 --interface eth1 -S 100.64.0.10 -g 5000 --rate 6000 <list of

-c              -> Number of repeats, 0 is for infinity
--tcp           -> Probe mode
port            -> destination port, or port range
--interface eth1-> the interface to use
-src            -> the source ip of the interface to use (if there are multiple)
-g              -> the src port to use
--rate 6000     -> the packets per minute
```

### 7.2.2 Python script

```
import ipaddress
from subprocess import call


IP = ipaddress.ip_address('192.168.30.0')

dest_ips = ""
for i in range(8900):
        dest_ips += str(IP) + " "
        IP += 1

SRC_IP = ipaddress.ip_address("100.64.0.10")

call("date", shell=True)

for i in range(30):
        call("nping -c 0 --tcp --interface eth1 -S " + str(SRC_IP) + " -g " + str(5000 +
        SRC_IP += 256
```

## 7.3 nftables

### 7.3.1 nftables-map

This is a simple Map config for nftables. We wanted to use Maps for our solution but found that they are still to buggy to used in a live setup. We hope of using them in the future, since they are easier than building a tree. Maps use a RBTree internally so the complexity is still log(n) like in our other solution. The issues we had so far with maps:

- Adding new elements to a map had a memory leak (this was fixed by the maintainer already)
- Very high memory consumption when updating a map entry
- Only the first two rules in a map worked (the maintainer send us a patch for the kernel module but we still have to try it)
- Deleting elements of the map is currently not possible.

```
#!/usr/sbin/nft
add chain nat postrouting { type nat hook postrouting priority 100 ;}
add chain nat prerouting { type nat hook prerouting priority 0 ;}
add map nat subnettoip { type ipv4_addr: ipv4_addr ; flags interval ; }
add rule ip nat postrouting snat ip saddr map @subnettoip;
add element nat subnettoip { 100.64.0.0/24 : 192.168.0.19 }
add element nat subnettoip { 100.64.1.0/24 : 192.168.0.20 }
add element nat subnettoip { 100.64.2.0/24 : 192.168.0.21 }
add element nat subnettoip { 100.64.3.0/24 : 192.168.0.22 }
```

### 7.3.2 nftables-tree

We have up to 6000 /24 Networks that we want to specific IP addresses. Simply adding them all in one chain would be to slow that is why we created a tree of chains. Starting with 2 networks at the top and than 8 networks per level. In the leaf chain the actual NAT rule is defined. Below is the start of our tree config for nftables down to the first 2 blocks of NAT rules. The rest of the tree is structured the same.

```
#!/usr/sbin/nft
add table nat
add chain nat prerouting { type nat hook prerouting priority 0 ;}
add chain nat postrouting { type nat hook postrouting priority 0 ;}
add chain nat postrouting-level-0
add rule nat postrouting ip saddr 100.64.0.0/12 goto postrouting-level-0
add chain nat postrouting-level-0-0
add rule nat postrouting-level-0 ip saddr 100.64.0.0/15 goto postrouting-level-0-0
add chain nat postrouting-level-0-0-0
add rule nat postrouting-level-0-0 ip saddr 100.64.0.0/18 goto postrouting-level-0-0-0
add chain nat postrouting-level-0-0-0-0
```

```
add rule nat postrouting-level-0-0-0 ip saddr 100.64.0.0/21 goto postrouting-level-0-0-0
add rule nat postrouting-level-0-0-0-0 ip saddr 100.64.0.0/24 snat 192.168.0.19
add rule nat postrouting-level-0-0-0-0 ip saddr 100.64.1.0/24 snat 192.168.0.20
add rule nat postrouting-level-0-0-0-0 ip saddr 100.64.2.0/24 snat 192.168.0.21
add rule nat postrouting-level-0-0-0-0 ip saddr 100.64.3.0/24 snat 192.168.0.22
add rule nat postrouting-level-0-0-0-0 ip saddr 100.64.4.0/24 snat 192.168.0.23
add rule nat postrouting-level-0-0-0-0 ip saddr 100.64.5.0/24 snat 192.168.0.24
add rule nat postrouting-level-0-0-0-0 ip saddr 100.64.6.0/24 snat 192.168.0.25
add rule nat postrouting-level-0-0-0-0 ip saddr 100.64.7.0/24 snat 192.168.0.26
add chain nat postrouting-level-0-0-0-1
add rule nat postrouting-level-0-0-0 ip saddr 100.64.8.0/21 goto postrouting-level-0-0-0
add rule nat postrouting-level-0-0-0-1 ip saddr 100.64.8.0/24 snat 192.168.0.27
add rule nat postrouting-level-0-0-0-1 ip saddr 100.64.9.0/24 snat 192.168.0.28
add rule nat postrouting-level-0-0-0-1 ip saddr 100.64.10.0/24 snat 192.168.0.29
add rule nat postrouting-level-0-0-0-1 ip saddr 100.64.11.0/24 snat 192.168.0.30
add rule nat postrouting-level-0-0-0-1 ip saddr 100.64.12.0/24 snat 192.168.0.31
add rule nat postrouting-level-0-0-0-1 ip saddr 100.64.13.0/24 snat 192.168.0.32
add rule nat postrouting-level-0-0-0-1 ip saddr 100.64.14.0/24 snat 192.168.0.33
add rule nat postrouting-level-0-0-0-1 ip saddr 100.64.15.0/24 snat 192.168.0.34
```

### 7.3.3 HowTo nftables

This is a short collection of useful nftables commands we used during our work.
Useful to know:

- If you use the type "nat" for a hook than only the first packet of each connection is processed by the chain
- For NAT there always have to be a chains with the prerouting and postrouting hook. For source NAT the prerouting chain is empty but still needed!
- Jump returns to the calling chain after completion, goto never returns.
- If a rate limit rule gets exhausted the next rule in the chain is called, or the default policy applies. If a host should be throttled than all traffic that exhausts the rate limit needs to be droped.

```
# Adds the table nat to nftables
nft add table nat

# List the content of the table nat
nft list table nat

# List all tables in nftables
nft list tables
```

```
# Can also be used to make a backup of the ruleset
nft list table nat > backup.ruleset

# Load the ruleset backup.ruleset into nftables, see the tree config for an example confi
nft -f backup.ruleset

# Deletes the table nat
nft delete table nat

# Deletes all rules in the table nat, chains are preserved
nft flush table nat

# Adds the chain prerouting to the table nat. The chain has the type nat
# and uses the prerouting hook with priority 0. Priority 0 means the its first.
nft add chain nat prerouting { type nat hook prerouting priority 0 \; }

# Add the chain postrouting to the table nat
nft add chain nat postrouting { type nat hook postrouting priority 100 \; }

# Deletes the chain postrouting in the table nat
nft delete chain nat postrouting

# Add a rule to the postrouting chain in the table nat. If the source ip
# is in the network 100.64.0.0/15 goto chain targetChain
add rule nat postrouting ip saddr 100.64.0.0/15 goto targetChain

# Adds a snat rule to the chain postrouting in the table nat. All source ips from
# 100.95.255.0/24 of the interface eth0 get translated to 141.30.233.255
nft add rule nat postrouting ip saddr 100.95.255.0/24 oif eth0 snat 141.30.233.255

# Addes a rule to the input chain in the table filter. Rate limits the traffic
# to 10 mbytes/second and accepts the traffic. When the rate limit is exhausted
# the next rule is applied. If you want to throttel a host then a drop rule is
# needed !.
nft add rule filter input limit rate 10 mbytes/second accept

# Adds a new named map subnettoip to the table nat. The map projects ipv4 addresses
# to ipv4 addresses. The interval flag allows you to use not just single addresses
# but entire networks: { 100.64.0.0/10 : 192.168.0.1 }
nft add map nat subnettoip { type ipv4_addr: ipv4_addr\; flags interval \; }

# Adds an element to the map subnettoip in the table nat.
nft add element nat subnettoip { 100.64.1.0/24 : 141.30.202.171 }

# Deletes the element 1.2.3.0/24 from the map subnettoip
nft delete element nat subnettoip { 1.2.3.0/24 }
```

```
# Lists the rule handles for the table nat
nft list table nat -a

# Deletes the rule with the handle 4
nft delete rule nat prerouting handle 4

# Adds the a snat rule to the postrouting chain in the table nat. The snat is
# performed using the subnettoip map.
nft add rule ip nat postrouting snat ip saddr map @subnettoip;
```

## 7.4 iptables

### 7.4.1 iptables-tree

This is the equivalent to the nftables tree for iptables.

```
*nat
:PREROUTING ACCEPT
:INPUT ACCEPT
:POSTROUTING ACCEPT
:postrouting-level-0 -
-A POSTROUTING -s 100.64.0.0/12 -g postrouting-level-0
:postrouting-level-0-0 -
-A postrouting-level-0 -s 100.64.0.0/15 -g postrouting-level-0-0
:postrouting-level-0-0-0 -
-A postrouting-level-0-0 -s 100.64.0.0/18 -g postrouting-level-0-0-0
:postrouting-level-0-0-0-0 -
-A postrouting-level-0-0-0 -s 100.64.0.0/21 -g postrouting-level-0-0-0-0
-A postrouting-level-0-0-0-0 -s 100.64.0.0/24 -j SNAT --to 192.168.0.19
-A postrouting-level-0-0-0-0 -s 100.64.1.0/24 -j SNAT --to 192.168.0.20
-A postrouting-level-0-0-0-0 -s 100.64.2.0/24 -j SNAT --to 192.168.0.21
-A postrouting-level-0-0-0-0 -s 100.64.3.0/24 -j SNAT --to 192.168.0.22
-A postrouting-level-0-0-0-0 -s 100.64.4.0/24 -j SNAT --to 192.168.0.23
-A postrouting-level-0-0-0-0 -s 100.64.5.0/24 -j SNAT --to 192.168.0.24
-A postrouting-level-0-0-0-0 -s 100.64.6.0/24 -j SNAT --to 192.168.0.25
-A postrouting-level-0-0-0-0 -s 100.64.7.0/24 -j SNAT --to 192.168.0.26
:postrouting-level-0-0-0-1 -
-A postrouting-level-0-0-0 -s 100.64.8.0/21 -g postrouting-level-0-0-0-1
-A postrouting-level-0-0-0-1 -s 100.64.8.0/24 -j SNAT --to 192.168.0.27
-A postrouting-level-0-0-0-1 -s 100.64.9.0/24 -j SNAT --to 192.168.0.28
-A postrouting-level-0-0-0-1 -s 100.64.10.0/24 -j SNAT --to 192.168.0.29
-A postrouting-level-0-0-0-1 -s 100.64.11.0/24 -j SNAT --to 192.168.0.30
-A postrouting-level-0-0-0-1 -s 100.64.12.0/24 -j SNAT --to 192.168.0.31
-A postrouting-level-0-0-0-1 -s 100.64.13.0/24 -j SNAT --to 192.168.0.32
```

```
-A postrouting-level-0-0-0-1 -s 100.64.14.0/24 -j SNAT --to 192.168.0.33
-A postrouting-level-0-0-0-1 -s 100.64.15.0/24 -j SNAT --to 192.168.0.34
```

### 7.4.2 HowTo

```
# Show all chains of the table nat
iptables -t nat -L

# Add the chain custom-chain to the table nat
iptables -t nat -N custom-chain

# Removes the chain custom-chain
iptables -t nat -X custom-chain

# Add the rule to the chain POSTROUTING in the table nat.
# The rule jumps to the chain custom-chain if the source ip matches.
iptables -t nat -A POSTROUTING -s 100.64.0.0/12 -j custom-chain

# Removes the jump rule from the chain POSTROUTING
iptables -t nat -D POSTROUTING -j custom-cahin

# Change the source ip to 123.123.123.123
iptables [...] -j SNAT --to-source 123.123.123.123

# Uses masquerade on the source ip -> changes it to the ip of the outgoing interface
iptables [...] -j MASQUERADE

# Changes the destination ip and port to 123.123.123.123:22
iptables [...] -j DNAT --to-destination 123.123.123.123:22

# Saves the ruleset
iptables-save > dump.ruleset

# Loads the ruleset
iptables-restore < dump.ruleset
```

### 7.4.3 conntrackd.conf

```
# Sync section to specifies how the synchronisation is done,
# which protocol is used, etc.
Sync {
            # The syncronisation mode and its options
            Mode FTFW {
                 ResendQueueSize 131072
                 PurgeTimeout 60
                 ACKWindowSize 300
```

```
                # Disables the external cache, the state entries
                # are directly injected in the state table
                DisableExternalCache On
        }
        # The protocol that is used for syncronisation and
        # the used ip addresses, buffer size etc.
        UDP {
                IPv4_address 10.77.0.1
                IPv4_Destination_Address 10.77.0.2
                Port 3781
                Interface net1
                SndSocketBuffer 1249280
                RcvSocketBuffer 1249280
                Checksum on
        }
        # General options
        Options {
                TCPWindowTracking On
                # Also sync the expectation table
                ExpectationSync On
        }
    }
# General options
General {
        #Systemd support
        Systemd on
        Nice -20
        Scheduler {
                Type FIFO
                Priority 99
        }
        # Number of buckets in the cache hashtable
        HashSize 32768
        # Number of conntrack entries, should be twice the size
        # of /proc/sys/net/netfilter/nf_conntrack_max
        HashLimit 131072
        LogFile on
        Syslog off
        LockFile /var/lock/conntrack.lock
        UNIX {
                Path /var/run/conntrackd.ctl
                Backlog 20
        }
        NetlinkBufferSize 2097152
        NetlinkBufferSizeMaxGrowth 8388608
        NetlinkOverrunResync On
```

```
        NetlinkEventsReliable Off
        EventIterationLimit 100
    }
```

## 7.5 celery application

```
netfilter:
tree: # The cidr size of the last tree element lowlevel: 21
\# Maximum rules on a tree level
jumpcount: 8
nft: call: '/usr/local/sbin/nft'
tmpfile: '/tmp/rules.nft'
forwarding: table: nat
translation: table: nat throttle: table: filter map: throttle\_map

conntrack: call: '/usr/sbin/conntrack'

# Multiple databases can be configured
databases: - name: static host: 10.10.233.1 user: nat password: test123
db: nat-static - name: dynamic host: 10.10.233.1 user: nat passwort:
test123 db: nat-dynamic

broker: host: 10.10.233.1 user: natgateway password: test123 queue: nat

cgn: ip: 100.64.0.1 net: 100.64.0.0/11 interface: eth4

inet: interface: eth7

# connections from/to these networks are not throttled
whitelist: - 141.30.0.0/16 - 141.76.0.0/16

# these are hosts in the whitelist networks, that should not be spared
blacklist: - 141.30.3.50 - 141.30.4.125 - 141.30.61.140 - 141.30.61.141 - 141.30.117.36
```

# Bibliography


[1] 802.1q. http://www.ieee802.org/1/pages/802.1Q-2014.html. Accessed: 2017-28-02.

[2] Carrier-grate nat. https://tools.ietf.org/html/rfc6598. Accessed: 2017-28-02.

[3] Celery. http://www.celeryproject.org/{[}1{]}http://www.rabbitmq.com/. Accessed: 2017-28-02.

[4] collectl. http://collectl.sourceforge.net/. Accessed: 2017-28-02.

[5] Conntrack-tools. http://conntrack-tools.netfilter.org/. Accessed: 2017-28-02.

[6] Fritzbox-4020. https://avm.de/produkte/fritzbox/fritzbox-4020/. Accessed: 2017-28-02.

[7] Ip mobility. https://tools.ietf.org/html/rfc3344. Accessed: 2017-28-02.

[8] iperf. https://iperf.fr/. Accessed: 2017-28-02.

[9] iptables. https://www.netfilter.org/projects/iptables/index.html. Accessed: 2017-28-02.

[10] keepalived. http://www.keepalived.org/. Accessed: 2017-28-02.

[11] Mobile ip cisco. http://www.cisco.com/c/en/us/td/docs/ios/solutions_docs/mobile_ip/mobil_ip.html. Accessed: 2017-28-02.

[12] Netfilter hooks. https://wiki.nftables.org/wiki-nftables/index.php/Netfilter_hooks. Accessed: 2017-28-02.

[13] Netfilters connection tracking system. http://people.netfilter.org/pablo/docs/login.pdf. Accessed: 2017-28-02.

[14] Network address translation. https://tools.ietf.org/html/rfc2663. Accessed: 2017-28-02.

[15] nftables. https://www.netfilter.org/projects/nftables/index.html. Accessed: 2017-28-02.

[16] nftables first release. http://article.gmane.org/gmane.comp.security.firewalls.netfilter.devel/28922. Accessed: 2017-28-02.

[17] nftables presentation. http://people.netfilter.org/kaber/nfws2008/nftables.odp. Accessed: 2017-28-02.

[18] nmap. https://nmap.org. Accessed: 2017-28-02.

[19] nping. https://nmap.org/nping/. Accessed: 2017-28-02.

[20] packet filter freebsd. https://www.freebsd.org/doc/handbook/firewalls-pf.html. Accessed: 2017-28-02.

[21] packet filter openbsd. https://www.openbsd.org/faq/pf/filter.html. Accessed: 2017-28-02.

[22] pfsense. https://www.pfsense.org/{[}3{]}https://blog.pfsense.org/?p=1588. Accessed: 2017-28-02.

[23] pfsync. https://www.freebsd.org/cgi/man.cgi?query=pfsync. Accessed: 2017-28-02.

[24] ping. https://linux.die.net/man/8/ping. Accessed: 2017-28-02.

[25] Rabbitmq. https://www.rabbitmq.com/. Accessed: 2017-28-02.

[26] Tl-wr841nd. http://www.tp-link.de/products/details/cat-9_TL-WR841ND.html. Accessed: 2017-28-02.

[27] Virtual router redundancy protocol. https://tools.ietf.org/html/rfc5798. Accessed: 2017-28-02.

[28] Wifi optimization. https://www.lancom-systems.de/blog/wlan-optimierung-schnelles-wlan/. Accessed: 2017-28-02.