BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PORT BLOCK ALLOCATION FOR NETWORK ADDRESS TRANSLATION

ALOKACE BLOKU PORTŮ PRO PŘEKLAD ADRES

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR Bc. TOMÁŠ ODEHNAL
AUTOR PRÁCE

SUPERVISOR Ing. MATĚJ GRÉGR, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2022

Department of Information Systems (DIFS) Academic year 2021/2022

Master's Thesis Specification

24650

Student:	**Odehnal Tomáš, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Networks
Title:	**Port Block Allocation for Network Address Translation**
Category:	Networking

Assignment:

1. Study the address translation mechanism that uses port block allocation.
2. Study the implementation of address translation in the Linux OS kernel and in the iptables tool.
3. Based on the supervisor's recommendation, create an implementation that allows per-user port block allocation.
4. Evaluate the implementation and discuss future extensions.

Recommended literature:

- Rosen, R. Linux kernel networking: implementation and theory. New York, NY: Apress, 2014. ISBN 9781430261964.
- Perreault, S., Yamagata, I., Miyakawa, S., Nakagawa, A., and H. Ashida, "Common Requirements for Carrier-Grade NATs (CGNs)", BCP 127, RFC 6888, DOI 10.17487/RFC6888, April 2013, <https://www.rfc-editor.org/info/rfc6888>.

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Grégr Matěj, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 18, 2022
Approval date:	October 13, 2021

Abstract

This term project aims to study the issue of the Carrier-Grade NAT (CGN) technique, which has to create log messages with address translation for every new connection. Because the CGN is stationed between large networks, it may daily record hundreds of thousands of connections. This amount of records have high memory requirements and even more difficult is to search for a specific log record. These problems solve the port block allocation for address translation. The output of this work is the creation of a rule in the iptables that performs this port block allocation for address translation. It consists of a user part that processes the rules and a kernel module that implements the functionality of the rule.

Abstrakt

Cílem této semestrální práce je nastudovat problematiku Carrier-Grade NAT (CGN) přístupu, který musí provádět záznam o překladu adres každého nového spojení. Protože CGN leží na rozhraních rozsáhlých sítí, může denně zaznamenat statisíce spojení. Toto množšství záznamů má vysoké paměťové nároky a ještě složitější je hledání konkrétního záznamu. Tyto problémy je možné řešit pomocí alokace bloku portů pro překlad adres. Výstupem této práce je vytvoření pravidla do iptables, které provádí tuto alokaci bloků pro překlad adres. To se skládá z uživatelské části, která zpracovává pravidla a kernelovský modul provádějící funkcionalitu pravidla.

Keywords

Linux, Linux kernel, Linux network stack, CGN, Carrier-Grade NAT, NAT, SNAT, Netfilter, IPTables, IPTables extensions, Port Block Allocation, PBA

Klíčová slova

Linux, Linux kernel, Linux síťový stack, CGN, Carrier-Grade NAT, NAT, SNAT, Netfilter, IPTables, rozšíření iptables, Alokace bloku portů, PBA

Reference

ODEHNAL, Tomáš. *Port Block Allocation for Network Address Translation.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

Rozšířený abstrakt

Přechod na IPv6 neproběhl tak rychle jak se předpokládalo. Jeden z důvodů je, že poskytovatelé internetu nechtěli udělat první krok i přes to, že vyčerpání adresového prostoru v IPv4 nastalo již v roce 2011. Jeden z mechanismů, který měl kompenzovat tento nedostatek místa v adresovém prostoru IPv4 je překlad síťových adres (NAT). Tento mechanismus se například používá pro přístup více počítačů z lokální sítě do Internetu prostřednictvím jedné veřejné IP adresy.

Poskytovatelé internetu samozřejmě tento překlad adres také používají. Mají i netriviální překlad adres ve své vnitřní topologii, který se označuje jako Carrier-grade NAT (CGN). Ten musí denně zpracovat obrovské množství (i v řádech milionů) spojení, protože se CGN používá na rozhraních velkých sítí. Navíc musí o každém spojení vytvořit log záznam. Poskytovatelé internetu jsou ze zákona povinni schraňovat tyto záznamy po několik měsíců.

Toto ovšem vede na dva problémy. Prvním je, že takto velké množství záznamů má velké paměťové nároky a to i v případě, kdy se použije komprese obsahu záznamů. Další problém je s vyhledáním konkrétní stanice, která v daný čas měla aktivní spojení, což znamená vyhledání konkrétního záznamu.

Možným řešením těchto dvou problémů je technika, na kterou se zaměřuje tato diplomová práce. Označuje se jako alokace bloku portů (angl. Port block allocation -– PBA) pro překlad adres. Tato metoda umožnuje plně (nebo i částečně) deterministicky v daný časový okamžik určit, na kterou veřejnou adresu a port se přeložila zdrojová IP adresa z privátní sítě. Díky tomu úplně (nebo alespoň částečně) odpadá nutnost logovat záznamy o každém spojení.

PBA funguje tak, že lze mapovat privátní zdrojovou IP adresu na konkrétní veřejnou IP adresu a blok portů. Velikost bloku portů znamená, kolik portů se nachází v tomto bloku. Také to určuje, jaký je povolený maximální počet současně aktivních spojení z jedné privátní zdrojové IP adresy. Toto nastává v případě, kdy je privátní adrese přidělen pouze jeden blok. Následně, když operátorovi přijde zpráva o možném bezpečnostním incidentu, kdy má k dispozici čas spojení, veřejnou zdrojovou IP adresu s portem, a potřebuje zjistit privátní zdrojovou IP adresu. V případě plně deterministického PBA mu stačí pouze konfigurace PBA v čas, kdy spojení probíhalo, aby zjistil, jak bylo provedeno mapování.

PBA se běžně používá u Carrier-grade NAT, což často bývá specializovaný hardware. Hlavní motivací této práce je snaha naimplementovat PBA jako rozšiřující kernel modul do frameworku iptables. Cílem této práce je základní prvotní přiblížení, jestli lze tento algoritmus vytvořit jako kernelovský modul, a jestli lze využít některé stávající prostředky z kernelu.

První částí této práce bylo nastudování fungování CGN a především fungování PBA, které je popsáno v RFC 7422. Tato část také zahrnuje studium, jakým způsobem je zpracováván paket v síťovém zásobníku v kernelu a fungování Netfilter frameworku, který umožňuje například, filtrovat síťový provoz, provádění překladu adres a další.

Další částí této práce bylo studium, jakým způsobem se vytváří kernelovský modul a jak se vytváří rozšiřující pravidlo do iptables. Také se zde vytváří návrh implementace a řešení vzniklých problémů, které se objevily. Například se musela řešit registrace notifikátoru do modulu, který si drží informace o probíhajících spojení.

Výsledná implementace se skládá ze dvou částí. Uživatelská část, která zpracovává a kontroluje validitu vytvořeného pravidla v iptables. Druhá část je kernelovský modul, který implementuje funkcionalitu vytvořeného iptables pravidla. Tento module je pojmenován

PBA. Vytvořené pravidlo má pouze dva parametry: rozsah IP adres (nebo jedna IP adresa) ve veřejné síti, na kterou se budou překládat IP adresy z privátní sítě a velikost jedno bloku.

PBA modul funguje tak, že při přidání pravidla do iptables si alokuje všechny bloky, které se budou přiřazovat skupinám spojení, které jsou vytvářeny ze stejných privátních IP adres. Blok je přiřazen privátní IP adrese při přijetí prvního paketu prvního spojení z této skupiny spojení. Privátní zdrojová IP adresa tohoto spojení se přeloží na IP adresu bloku a jeden z portů v tomto bloku. Následujícím spojením jsou z tohoto bloku přiřazeny zbylé volné porty.

Implementace PBA metody v této práci není plně deterministická, ale bloky se přiřazují postupně, jak jsou volné. Může se tedy stát, že stejné privátní IP adrese jsou přiděleny v různé časy různé bloky, tedy se tato IP adresa přeloží na jinou veřejnou IP adresu a porty. Z tohoto důvodu je při přiřazení bloku privátní IP adrese vytvořena log zpráva, která nese informaci o mapování dané privátní IP adresy na veřejnou IP adresy a porty. Při odstranění přiřazení daného bloku se také vytváří log zpráva informující o zrušení tohoto mapování.

Nakonec bylo provedeno základní testování vytvořené implementace. Toto testování probíhalo na virtuálních počítačích.

Port Block Allocation for Network Address Translation

Declaration

I hereby declare that I have authored this Master's thesis as an original work under the supervision of Ing. Matěj Grégr, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Tomáš Odehnal
May 17, 2022

Acknowledgements

I would like to express my gratitude towards my supervisor Ing. Matěj Grégr, Ph.D., for his patient guidance and invaluable advice.

Contents

Chapter 1

Introduction

Global deployment of IPv6 did not happen as fast as was forecast. Network providers were hesitant to make the first move while IPv4 was and still is working well. Exhaustion of the IPv4 address pool occurred in February 2011. Moreover, it becomes increasingly difficult to obtain new address assignments from Regional or Local Internet Registries. As a result, the Service Providers started to use Carrier-Grade Network Address Translation (CGN).

The CGN has to handle hundreds of thousands (even millions) of connections every day because it is stationed between large networks. The CGN must create a log record of address translation for every connection. This amount of logging records has substantial memory requirements. The bigger problem is when an operator needs to find a specific log record. Searching for this particular record is very tedious work.

Both problems are solved by the port block allocation (PBA) technique for the Network Address Translation, which allows internal network addresses to be mapped to an external address and port pair. Then there is no need to create a log record for every connection. Operators only need to know the configuration of the CGN and external IP address and port to find a specific internal IP address.

The implementation of the PBA in this work is done directly in the Linux kernel network stack. The Linux stack contains the module Netfilter, which allows performing user-defined operations on packets. The most known submodule of Netfilter is the iptables framework. It allows users to create rules, and iptables will add them to the Linux kernel.

First, the document discusses the Carrier-Grade NAT technique and port block allocation for Network Address Translation (Chapter 2). Chapter 3 examines the function and implementation of the Linux Network Stack with Netfilter and IPTables modules.

Chapter 4 describes the design and discusses issues that showed up during the design stage and their solutions.

The implementation details are in chapter 5. Here are described the solutions to the issues from the Design chapter. It also shows how the implementation was tested.

Chapter 2

Carrier Grade NAT

Network Address Translation (NAT) is used between private and public IP networks and translates private IP addresses to a public IP address. It is designed for IP address conservation. NAT dynamically maps one or more IP addresses from a private network to one or more IP addresses from a public network (globally routable), most commonly using Network Adress and Port Translation (NAPT) techniques. Typical use of NAT is at home network gateways to translate multiple private IP addresses of home devices to a single public IP address that the service provider provisions. Service providers deploy NAT in such a way that various subscribers can share a single global IP address. The Carrier-Grade NAT (CGN) is a NAT that scales to several millions of NAT translations and is used by service providers. This address sharing comes from the increasing difficulty in obtaining a new IPv4 address from Internet Service Providers due to depleting unallocated IPv4 address space supplies.

However, sharing addresses with other subscribers creates additional problems/challenges for Service Providers. Operators must manage service entitlement, public safety requests, and attack/abuse/fraud reports. Because of that, there is a need to identify a specific user associated with an IP address. In response to such a request or service entitlement, for every user-initiated connection, operators will need to map their internal private source IP address and source port with the global public IP address and source port provided by CGN.

All CGN connections must be logged to satisfy the need to identify attackers and respond to abuse/public safety requests. However, that imposes significant operational problems on operators. In lab testing, CGN log messages are approximately 150 bytes long for NAT444. In 2014, according to [RFC7422, 1.], reports from several US operators set the average household number of connections at approximately 33 000 connections per day. Furthermore, because every connection must be logged, this makes log data size of approximately 5 MB per subscriber per day or 150 MB per month (but individual message size might vary). Based on this information, if Service Provider has 1 million subscribers, it generates approximately 150 terabytes of logging data per month. Data volume is still too high even though Service Providers use compression techniques.

This data volume poses a problem for both the law enforcement agency (LEA) and operators. It makes it harder and longer to locate data required to investigate an abuse report for the public safety community. Moreover, on the operator side, it requires putting more resources into infrastructure when implementing CGN. The following section describes a technique that reduces this impact and, at the same time, improves abuse response.

To reduce the volume of CGN logging, we can assign port ranges instead of individual ports. Then there is only a need to log the assignment of a new port range. This technique might significantly reduce logging volume. How significant this reduction can be, depend on the length of the assigned port range or if the port range is static or dynamic. However, even after this reduction, there is still an impact on operators for CGN logging and searching [7] [3] [2].

2.1 Deterministic Address Mapping in CGN

We can design and configure the CGN to deterministically map internal addresses to the pair external address with port range to algorithmically calculate this mapping. Then we can only log inputs and configuration of the used algorithm. This approach reduces both logging and time to identify a user. In some cases, when we use total deterministic allocation, it can even eliminate the need to log CGN translation. So this technique significantly reduces the burden on operators and, at the same time, offers the ability to map the inside IP address of the user to an external IP address and port (observed on the Internet). The term address space multiplicative factor must be described before showing the configuration parameters of deterministic CGN.

2.1.1 Address Space Multiplicative Factor

The purpose of sharing public IPv4 addresses is to increase the addressing space. Address space multiplicative factor is a parameter by which service providers want or need to multiply IPv4 public address space, and the consequence is the number of users who share the same public IPv4 address. Service providers use this parameter when they need bigger public IPv4 address space. The inverse is called the compression ratio. The compression ratio can be expressed as (*number of inside IP addresses / number of outside IP addresses*).

When the multiplicative factor is large, the average number of ports per subscriber is small. Different service providers might have different requirements. A service provider with a stable number of subscribers may have a small existing address pool (small multiplicative factor, less than 10). Service providers for a new line of business will require a much bigger multiplicative factor (e.g., 1000).

2.1.2 Deterministic Port Ranges

A subscriber can use thousands of connections per day, but most of the subscribers use far fewer resources at any given time. When service providers have a low compression ratio (For example, the ratio of the number of public IPv4 addresses allocated to CGN to the number of subscribers is closer to 1:10 rather than 1:1000), then each subscriber could have access to thousands of TCP/UDP ports at any given time. Service providers do not need to log every connection. They can use deterministic CGN. It deterministically maps the customer's internal private address, received on the inner side of the CGN (customer-facing interface of the CGN), to the public IPv4 address and public ports, on the outer side of the CGN (Internet-facing interface of the CGN). So when operators receive a report of abuse with a public IPv4 address and port, they can identify the subscriber's internal private address without examining the CGN translations logs. Thanks to this algorithm, operators do not have to transport and store massive amounts of logs from the CGN and then process them to identify a subscriber [7].

The algorithmic mapping can be expressed as:

$$\text{(external IP address, port range)} = \text{function 1 (internal IP address)} \tag{2.1}$$

$$\text{internal IP address} = \text{function 2 (external IP address, port number)} \tag{2.2}$$

Where function 1 is a mapping function that maps an internal IP address to an external IP address and port. This is what we require from NAT. Function 2 is its inverse function. It maps an external IP address with a port to an internal IP address. This function is used when, for example, a report comes in and the operator needs to look up a specific internal IP address. It is recommended that the CGN provide a method to test both mapping functions.

Deterministic Port Range allocation requires configuration of the following parameters:

- **I** – Inside IPv4 / IPv6 range.
- **O** – Outside IPv4 address range.
- **C** – Compression ratio (inside IP addresses I / outside IP addresses O).
- **D** – Dynamic address pool factor, to be added to the compression in order to create an overflow address pool.
- **M** – Maximum ports per user.
- **A** – Address assignment algorithm (see 2.1.3)
- **R** – Reserved TCP / UDP port list

A subscriber is identified by an internal IPv4 address. However, those hosts, who share an internal IP address (there is another address translation on the inner side of the CGN), cannot be retrieved because the algorithm is not designed this way [7].

2.1.3 Address-assignment algorithms

It is possible to use several address-assignment algorithms. Service providers can use predefined algorithms (described next) depending on their requirements. These simplify the process of reversing the algorithm when needed. Nevertheless, the CGN used by service providers can support additional algorithms, and it is not required to support all the algorithms described next. For example, a subscriber might be restricted to ports from a single IPv4 address or could have allocated ports across all addresses in a pool [7].

The following algorithms and corresponding values of **A** (in parameters in 2.1.2) are as follows:

0: Sequential – for example, the first block goes to address 1. the second block to address 2, and so forth.

1: Staggered – for example, for every n (n is number of public addresses) between 0 and $((65536 - \mathbf{R})/(\mathbf{C} + \mathbf{D})) - 1$, address 1 receives ports $n * \mathbf{C} + \mathbf{R}$, address 2 receives ports $(n + 1) * \mathbf{C} + \mathbf{R}$, etc.

2: Round robin – for example, the subscriber receives the same port number across a pool of external IP addresses. If the subscriber is to be assigned more ports than there are in the external IP pool, the subscriber receives the next highest port across the IP pool, and so on. So if there are 10 IP addresses in a pool and a subscriber is assigned 1000 ports, the subscriber would receive a range such as ports 2000 - 2099 across all ten external IP addresses.

3: Interlaced horizontally: for example, each address receives every **C**th port spread across a pool of external IP addresses.

4: Cryptographically random port assignment – If this algorithm is used, the service providers must keep the keying material and specific cryptographic function to support reversibility.

5: Vendor-specific – other vendor-specific algorithms also might be supported [7].

2.1.4 The ports Reservation Process

When the CGN has all required parameters and defined algorithms, it then reserves ports as follows:

1. Firstly the CGN removes reserved ports (**R**) from the port candidate list (for example, 0-1023 for TCP and UDP). Nevertheless, at minimum, it is recommended that the CGN should remove system reserved ports from the port candidate list for deterministic assignment.

2. Then the CGN must calculate the total compression ratio ($\mathbf{C}+\mathbf{D}$). Then it allocates $1/(\mathbf{C}+\mathbf{D})$ of the available ports to each internal IP address. Moreover, the CGN has configured an assignment algorithm (**A**), so it can allocate specific ports deterministically. Any remaining ports are allocated to the dynamic pool. If parameter **D** is set to 0, it disables the dynamic pool. This option eliminates the need to log per-subscriber connections, limiting the number of concurrent connections that „demanding users“ can initiate.

3. When a subscriber makes a connection, the CGN must create a translation mapping between the subscriber's internal private IP address and port and the CGN's external public IP address and port. The CGN must use one of the allocated ports from step 2 for the translation as long as these ports are available. Furthermore, the CGN has to allocate ports randomly within the ports range assigned by the deterministic algorithm (**A**) to increase subscriber privacy. The CGN maintains its mapping table, but it does not need to generate log entries for translation mappings created in this step. However, the CGN still needs to generate log entries for dynamic translation mappings (created in the next step).

4. If the dynamic address pool factor (**D**) is greater than 0, the CGN will have a pool of ports left for dynamic assignment. If the subscriber requires more than the ports range allocated in step 2 but fewer than the configured maximum ports per subscriber (**M**), the CGN must assign a block of ports from the dynamic assignment range for such a connection. However, the CGN must log these dynamically assigned port blocks to facilitate subscriber-to-address mapping.

5. Lastly, the configuration of reserved ports (for example, system reserved ports) is left to the operator [7].

The CGN will maintain translation mapping for each connection within its internal translation tables thanks to this process. It only needs to log translations for dynamically assigned ports.

2.1.5 Logging Consideration

This technique, however, is not entirely without any form of logging. Service providers need to identify a subscriber based on observed external IPv4 address, port, and timestamp. The timestamp is required when the CGN configuration can change, and the subscriber's IP address is translated to a different external IP address between two connections. An operator needs to know how the CGN was configured concerning internal and external IP addresses, dynamic address pool factor, maximum ports per user, and reserved port range at any given time. Thus, the CGN has to generate a record with configuration every time any of these parameters are changed. The CGN should generate a log message when any of these parameters are changed. Alternatively, if the CGN does not generate any log message, it is up to the operator to maintain version control of the CGN configuration.

If any log message is generated, it must, at minimum, include the timestamp, inside prefix (**I**), the inside mask, outside prefix **O**, the outside mask, **D**, **M**, **A**, and reserved port list **R** [7].

2.2 Deterministic CGN Example

This section is a simple example to demonstrate the function of deterministic CGN. The example is taken from RFC7422. The service provider operator configures an inside address pool (**I**) of 198.51.100.0/28 and one outside address (**O**) of 192.0.2.1. Then he set the dynamic address pool factor (**D**) to value 2. The total compression ratio (**C**) is then 1:(14+2) = 1:16. Reserved ports (**R**) are only system ports (< 1024). The CGN will, according to the configuration, pre-allocate (65536-1024)/16 = 4032 TCP/UDP ports per inside IPv4 address. To keep this example simple, let us assume that the ports are allocated sequentially (198.51.100.1 maps to 192.0.2.1 ports 1024-5055, 198.51.100.2 maps to 192.0.2.1 ports 5056-9087, to 198.51.100.14 maps to 192.0.2.1 ports 53440-57471). The dynamic port range then contains ports 57472-65535. The port allocation can be seen in the table 2.1. And finally, the operator sets the maximum ports per subscriber (**M**) to 5040.

When subscriber 1 with inside address 198.51.100.1 uses a low volume of connections (< 4032 concurrent connections), the CGN will map the outgoing source address/port to the pre-allocated range. An important aspect is that these translation mappings are not logged.

If subscriber two (with inside address 198.51.100.2) needs to use more connections than allocated 4032 ports simultaneously, the CGN allocates up to additional 1008 ports using bulk port reservations. The CGN allocates 1008 ports because 1008 + 4032 = 5040, the maximum number of ports per subscriber. In this example, subscriber two uses outside ports 5056-9087 and then 100-port blocks between 58000-58999. Moreover, connections that use outside ports 5056-9087 are not logged, while 10 log entries are created for ports 58000-58099, 58100-58199, ..., 58900-58999.

It is advised that content providers should log the source address, source port, and timestamp for all log entries. Because if there is a need to identify a subscriber behind

Inside Address (Pool)	**Outside Address + Port**
Reserved	192.0.2.1:0-1023
198.51.100.1	192.0.2.1:1024-5055
198.51.100.2	192.0.2.1:5056-9087
198.51.100.3	192.0.2.1:9088-13119
198.51.100.4	192.0.2.1:13120-17151
198.51.100.5	192.0.2.1:17152-21183
198.51.100.6	192.0.2.1:21184-25215
198.51.100.7	192.0.2.1:25216-29247
198.51.100.8	192.0.2.1:29248-33279
198.51.100.9	192.0.2.1:33280-37311
198.51.100.10	192.0.2.1:37312-41343
198.51.100.11	192.0.2.1:41344-45375
198.51.100.12	192.0.2.1:45376-49407
198.51.100.13	192.0.2.1:49408-53439
198.51.100.14	192.0.2.1:53440-57471
Dynamic	192.0.2.1:57472-65535

Table 2.1: Port allocation to inside IPv4 addresses.

a CGN, public safety agencies need to collect these three values from content providers' log files. Then if there is any abuse report from 192.0.2.1 and port 2001, the operator can reverse the mapping algorithm to determine that the internal IP address (198.51.100.1) belongs to subscriber one. The critical fact is that operator can determine this address without consulting the CGN logs. If there is another abuse report from 192.0.2.1 and port 58204, the operator knows that this port is within the dynamic address pool range. Thus he needs to consult the log file and correlate report data with the connection records, and the operator finds that subscriber two generated this traffic.

There are no log entries for most subscribers in this example because they only use pre-allocated ports. Logging only needs subscribers who exceed their pre-allocated ports and obtain extra bulk port assignments from the dynamic pool range [7].

Chapter 3

The Linux Network Stack

According to Open Systems Interconnection (OSI) model, there are seven logical networking layers. The lowest layer is the physical layer, which contains hardware. The highest layer is the application layer, where userspace software is running. Figure 3.1 (left) shows the seven layers according to the OSI model. Simple description of seven layers:

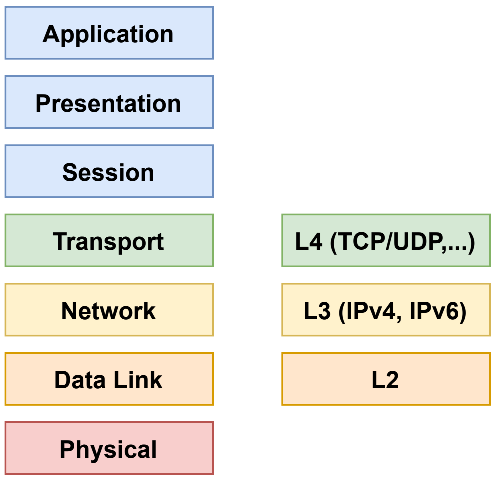


Figure 3.1: The OSI seven-layer model (left) and The Linux Kernel Networking layers (right).

1. The physical layer – Contains all hardware aspects of the computer network (electrical signals, low-level details).
2. The data link layer – Handles data transfer between network interfaces, and the most known data link layer is Ethernet.
3. The network layer – Contains host addressing and packet forwarding. The most common network protocols are IPv4 and IPv6.
4. The transport layer – Handles sending data between nodes and running applications on ports.
5. The session layer – Handles sessions between endpoints.

6. The presentation layer – Handles delivery and formatting.
7. The application layer – Provides network services to end-user applications.

Figure 3.1 (right) also shows the three layers that the Linux Kernel Networking stack handles. The L2, L3, and L4 layers correspond to the OSI seven-layer model's data link, network, and transport layers. The Linux kernel handles only these three layers. It does not handle the physical layer (L1) and any layer above L4. The Linux kernel stack passes incoming packets from L2 (the network device drivers) to L3 (the network layer, mostly IPv4 or IPv6). Then passes the packet to L4. If an incoming packet is for local delivery, it is at the L4 layer passed to, for example, TCP or UDP listening sockets. Alternatively, it is passed back to the L2 layer for transmission if the packet should be forwarded. If the packet is locally generated and is outgoing, it is passed from L4 to L3 and then to the L2 layer for transmission by the network device driver [9].

3.1 The Network Device

The lower layer, L2, is the data link layer. It contains network device drivers. This network device is in the Linux kernel, represented as `net_device` structure. This structure is huge and consisting network device parameters. It contains, for example, the MTU of the device, the IRQ number of the device, the MAC address, the name of the device (like `eth0`), flags of the device, and other vital device parameters. This structure is located in `<include/linux/netdevice.h>`. The main tasks of the network device driver are these:

- If the packets are destined for the local host, they are passed to L3 (the network layer). And from there to the L4 (the transport layer).
- Transmit outgoing packets that were generated locally and sent outside.
- Alternatively, forward packets that were received locally.

Incoming or outgoing packets, including their headers, are in the Linux kernel stack represented as `sk_buff` structure. This structure is called socket buffer (and stands for SKB). The socket buffer (`sk_buff`) structure is a large structure. An SKB API must be used when someone wants to work with SKBs. For example, when we want to use `skb->data` pointer, we do not do it directly, but with the function `skb_pull_inline()`. L4 header can be accessed from SKB by calling the `skb_transport_header()` method. Similarly, the L3 header is accessed by calling the `skb_network_header()` method or the L2 header by calling the `skb_mac_header()` method. The following section describes what Netfilter is and how it works [9].

3.2 Netfilter

This section discusses the Netfilter subsystem. The Netfilter is a framework that was created in 1988 by Rusty Russell. It is an improvement of the older implementation of ipchains. The Netfilter subsystem framework enables registering callbacks in various vital points (Netfilter hooks) of packet traversal in the Linux kernel stack. These callbacks perform various operations on packets, such as changing addresses or ports, dropping packets, logging, and more.

The incoming packet traversal (described here) is illustrated in figure 3.2. When receiving packets, the `ip_rcv()` is the primary IPv4 method, which is the handler for all incoming IPv4 packets. This method mainly checks if the incoming packet is valid. This method's end is the NF_INET_PRE_ROUTING Netfilter hook, which is invoked by calling the `NF_HOOK` macro. Every Netfilter hook (callback) is invoked by calling this macro. Registered callbacks (described later in <section>) are passed to macro as pointer parameters. The real work is done in the `ip_rcv_finish()`, and a lookup in the routing subsystem should be performed immediately after calling the function. The received packet might be forwarded and handled by the `ip_forward()` method, or it might be destined for the local machine and is handled by `ip_local_deliver()` method.

If the packet is destined for the local machine, it will reach `ip_local_deliver()` method. Here, the packet is checked whether it is fragmented, and if it is, the `ip_defrag()` method is called to handle the reassembling of all the fragments of the packet. At the end of the `ip_local_deliver()` method is the `NF_INET_LOCAL_IN` Netfilter hook, which is invoked by the `NF_HOOK` macro. After that packet will reach the `ip_local_deliver_finish()` method.

If the packet is to be forwarded, it will be handled by the `ip_forward()` method. Here, the outgoing packet is checked if it has not a bigger size than the outgoing MTU. The `ttl` (Time to Live) field is the IPv4 header counter which is decreased by 1. If the `ttl` reaches 0, the packet is dropped, and the corresponding time exceeded ICMPv4 is sent. At the end of this function is another call of `NF_HOOK` macro for `NF_INET_FORWARD` Netfilter hook. Then the packet is handled by the `ip_forward_finish()` method, which updates statistics and checks if the IPv4 packet includes IP options. If it does, then it will handle them. In the end, the `dst_output()` method receives the packet, and the only thing this method does is invoke `skb_dst(skb)->output(skb)` [9].

3.2.1 Netfilter Hooks

There are five critical points in the Linux kernel stack where Netfilter hooks (red ones in figure 3.2) are placed. The names of hooks are the same for IPv4 and IPv6 (the network stack methods were described above):

- `NF_INET_PRE_ROUTING` – This hook ends with the `ip_rcv()` method in IPv4 and `ip6_rvc()` in IPv6. All the valid incoming packets (with no exceptions) hit this hook. This hook is reached before the routing decision. The Destination Network Translation (DNAT) is implemented in this hook.

- `NF_INET_LOCAL_IN` – This hook is in the `local_deliver()` method in IPv4 and the `ip6_input()` method in IPv6. All the incoming packets going to the local machine reach this hook. After performing routing, the packets reach this hook after passing the `NF_INET_PRE_ROUTING` hook point.

- `NF_INET_FORWARD` – This hook is in the `ip_forward()` method in IPv4 and the `ip6_forward()` method in IPv6. Packets that are not addressed to the local machine and going through the machine reach this hook. After performing a routing, the forwarded packets reach this hook after passing the `NF_INET_PRE_ROUTING` hook point.

- `NF_INET_POST_ROUTING` – This hook is in the `ip_output()` method in IPv4 and the `ip6_finish_output2()` method in IPv6. Source Network Address Translation

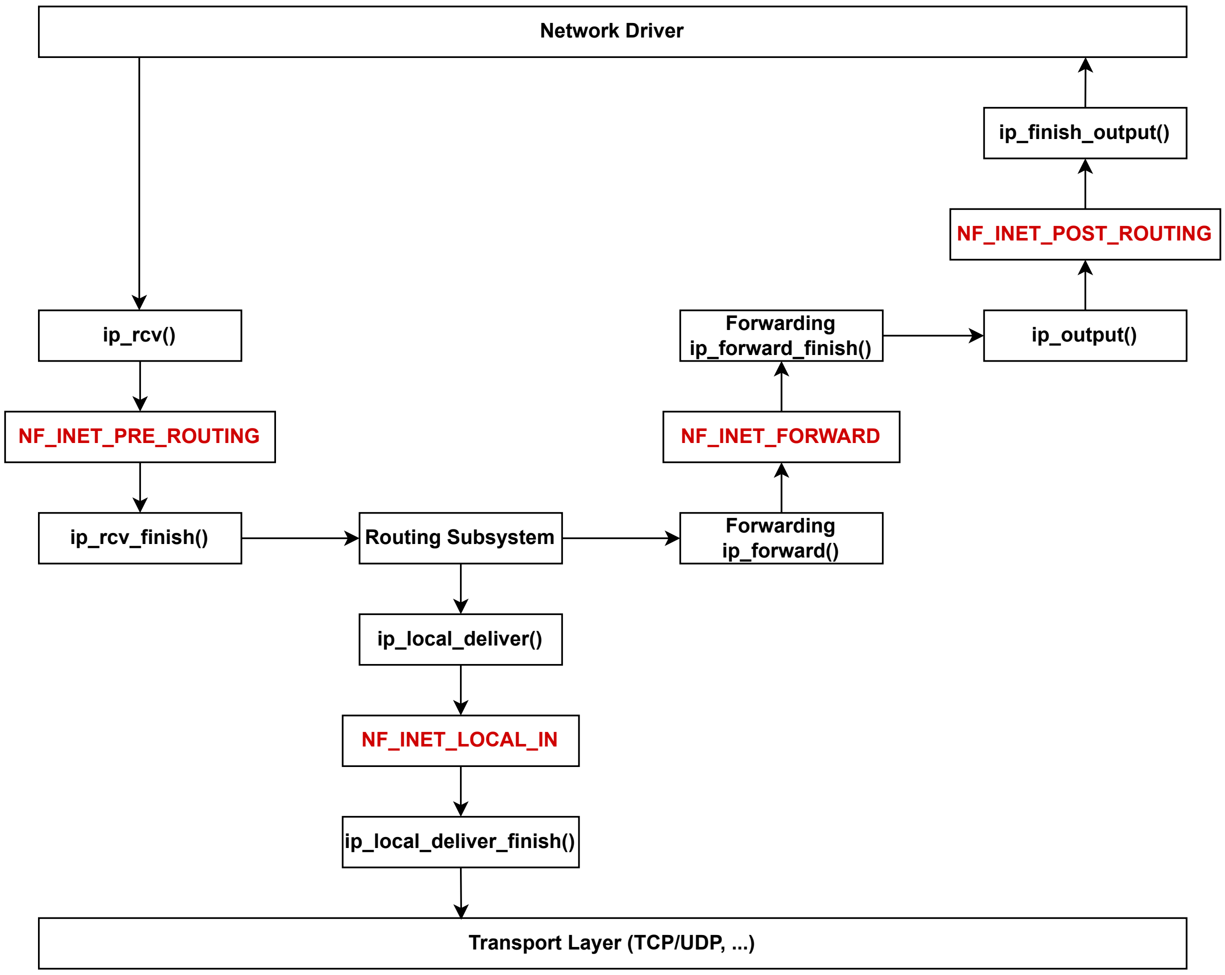


Figure 3.2: Simplified diagram of receiving IPv4 packets in the Linux kernel. Taken from [9].

(SNAT) is registered to this hook. Packets that are forwarded and passed the `NF_INET_FORWARD` hook point reach this hook. Furthermore, packets created in the local machine and outgoing will reach this hook after passing the `NF_INET_LOCAL_OUT` hook point.

- `NF_INET_LOCAL_OUT` – This is the only hook point that is not in the figure 3.2. This hook is in the `__ip_local_out()` method in IPv4 and in `__ip6_local_out()` method in IPv6. All outgoing packets created in this local machine reach this hook before reaching the `NF_INET_POST_ROUTING` [9, 5].

The return value of the Netfilter hook must be one of the following values (also termed Netfilter verdicts):

- NF_DROP (0) – Discard the packet silently. If multiple methods are attached to a hook, the packet will be dropped if a single method returns NF_DROP.
- NF_ACCEPT (1) – Packet continues its traversal in the Linux kernel network stack. It means that the next hook attached to that point will be called.

- NF_STOLEN (2) – Indicates that the packet has been consumed hook callback. The packet does not continue traversal in the Linux kernel network stack, but it is processed by the hook method.
- NF_QUEUE (3) – Queue the packet for userspace so that the userspace program will process the packet instead of the kernel stack.
- NF_REPEAT (4) – Indicates that the hook method should be called again [9, 5, 6].

3.2.2 Registration of Netfilter Hooks

A `nf_hook_ops` structure must be defined to register a hook callback at one of the five hook points. Hooks are per-protocol. Thus, one structure can only be registered on the protocol. So more hooks must be registered as an array of `nf_hook_ops`. There are two methods to register Netfilter hooks. The first (`nf_register_hook()`) method registers a single `nf_hook_ops` object and the second (`nf_register_hooks()`) registers an array of `nf_hooks_ops` objects. The structure with main attributes is defined like this:

```
struct nf_hook_ops {
    nf_hookfn     *hook;
    struct module *owner;
    u_int8_t      pf;
    unisgned int  hooknum;
    int           priority;
}
```

These are some vital members of the `nf_hook_ops` structure:

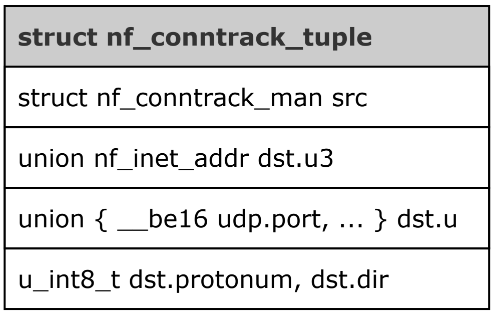
- `hook` - The hook callback to registering.
- `pf` - The protocol family (NFPROTO_IPV4 for IPv4 and NFPROTO_IPV6 for IPv6).
- `hooknum` - Is one of the five Netfilter hooks mentioned earlier.
- `priority` - More hook callbacks can be registered on the same hook. Callbacks with lower priorities are called first [9, 8].

3.3 Connection Tracking

The days when packet filtering policies were based uniquely on the packet header information, such as the IP source and destination addresses and ports, are over. There is also a need to consider cases when the traffic is based on sessions, for example, FTP sessions, where a session is a sequence of expected events. So, the Connection Tracking layer stores information about the state of a connection in a memory structure. This layer's primary goal is to serve as the basis of NAT. The IPv4 (and IPv6) NAT module cannot be built if CONFIG_NF_CONNTRACK_IPV4 (CONFIG_NF_CONNTRACK_IPV6) is not set [9, 5].

Firstly *Connection Tracking hooks* must be defined (initialized) as an array of `nf_hook_ops` objects. Then these hooks must be registered as any other hooks.

The essential data structure is `nf_conntrack_tuple` (see figure 3.3 of simplified representation). This structure represents a flow in one direction by its network-layer and transport-layer addresses. The structure use unions to contain protocol-specific fields and generic fields in `dst.u`. These unions make the source code easier to understand and allows new protocol-specific fields to be added without breaking the existing code [9, 6].



struct nf_conntrack_tuple
struct nf_conntrack_man src
union nf_inet_addr dst.u3
union { __be16 udp.port, ... } dst.u
u_int8_t dst.protonum, dst.dir

Figure 3.3: Simplified structure `nf_conntrack_tuple`. Taken from [6].

A Connection Tracking module for each transport layer (L4) protocol implements the protocol-specific part. The modules conform to the interface defined by `struct nf_conntrack_l3proto` and `struct nf_conntrack_l4proto`. These structures contain function pointers that are initialized to the appropriate functions in the protocol-specific modules [6].

3.3.1 Hashing

The Connection Tracking stores the states of active connections in a hash table for optimized performance. The method `nf_conntrack_raw()` returns 32 bit hash of a tuple. The hash value is based on the source and destination IP addresses and protocol-specific identifiers.

Figure 3.4 shows Connection Tracking entry `nf_conn` with an array of `nf_conntrack_tuple_hash` structure. This structure stores this Connection Tracking entry in the hash table and contains a tuple and pointer to a linked list of Connection Tracking entries associated with the tuple. The linked list is used to handle hash collisions.

The following is a description of some of the crucial members of the `nf_conn` structure:

- `ct_general` – A reference count.
- `tuplehash` – The array contains two `tuplehash` objects: `tuplehash[0]` is the original direction, and `tuplehash[1]` is the reply. They are usually referred to as `tuplehash[IP_CT_DIR_ORIGINAL]` and `tuplehash[IP_CT_DIR_REPLY]`.
- `status` – The status of the entry.
- `master` – An expected connection.
- `timeout` – When the connection entry expires. It contains a list of timers related to the connection state. These are typically timers that handle protocol timeouts and connection expiration [6].

3.3.2 Tracking

The Connection Tracking modules use three Netfilter hooks to track incoming and outgoing packets. Registered callbacks at hook points `NF_INET_PRE_ROUTING` and `NF_INET_LOCAL_OUT`

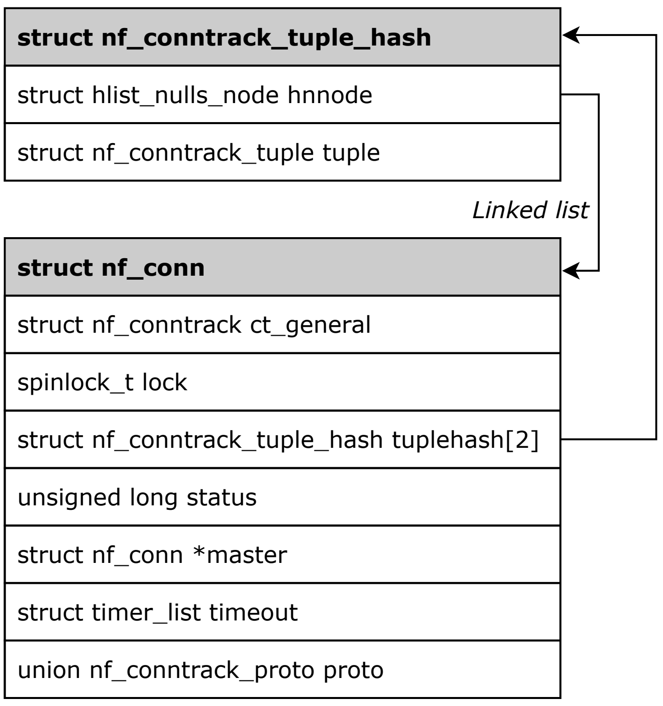

struct nf_conntrack_tuple_hash
struct hlist_nulls_node hnnode
struct nf_conntrack_tuple tuple
Linked list
struct nf_conn
struct nf_conntrack ct_general
spinlock_t lock
struct nf_conntrack_tuple_hash tuplehash[2]
unsigned long status
struct nf_conn *master
struct timer_list timeout
union nf_conntrack_proto proto

Figure 3.4: Structure `nf_conn` with structure `nf_conntrack_tuple_hash`. Taken from [6].

call `nf_conntrack_in()` method. Hook `NF_INET_POST_ROUTING` calls `nf_conntrack_confirm()`. The method `nf_conntrack_in()` is the main function of the Connection Tracking module.

When processing the packet, the initial steps of the `nf_conntrack_in()` methods are to get network-layer and transport-layer protocols. If the protocols can be tracked (was registered), then structures `nf_conntrack_l3proto` and `nf_conntrack_l4proto` are initialized. Then the function checks protocol-specific error conditions (for example, in the case of UDP, it checks malformed packets with invalid payload size or invalid checksums). The `resolve_normal_ct()` method is called if these checks are successful [6].
The `resolve_normal_ct()` method performs the following:

- Calculates the hash of the tuple by calling the `hash_conntrack_raw()` method (mentioned earlier).
- With the calculated hash, it performs a lookup for a tuple match.
- If there is no match, it creates a new `nf_conntrack_tuple_hash` object by calling the `init_conntrack()` method. This `nf_conntrack_tuple_hash` object is added to the list of unconfirmed `tuplehash` objects. If any other Netfilter modules do not drop the packet, the packet should be observed by the `nf_conntrack_confirm()` method at the `NF_INET_POST_ROUTING` hook. This function checks that other modules did not drop a packet belonging to a connection.
- Next is called the `nf_ct_timout_lookup()` method to decide the timeout policy suitable for the flow. For example, UDP flows set the timeout to 30 seconds for one-direction connections and 180 seconds for bidirectional connections.

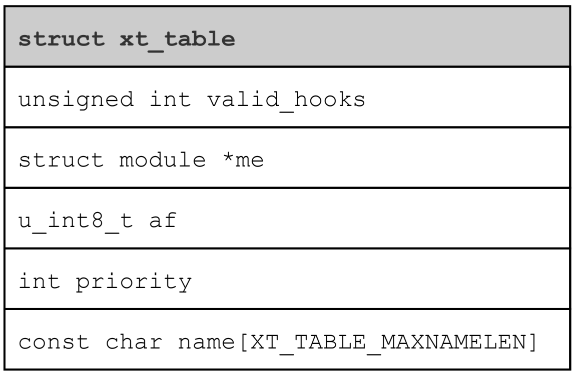
- Then the protocol-specific `packet()` method is called (for example, the `upd_packet()` for UDP or the `tcp_packet()` method for TCP). The `udp_packet()` method extends the timeout according to the status of the connection. It will be set to 30 seconds for connections without reply, and for replied connections, it will be set to 180 seconds [6, 9].

In the next section is described the iptables module.

3.4 IPTables

The Iptables framework is probably the most known part of the Netfilter framework. There are two parts. The kernel part - the core is in `net/ipv4/netfilter/ip_tables.c` for IPv4, and in `net/ipv6/netfilter/ip6_tables.c` for IPv6. Then there is the userspace part, most familiar from using the iptables(8) command. It provides a frontend for accessing the kernel iptables layer (example: adding and deleting rules). Each table is represented by the `xt_table` structure (defined in `include/linux/netfilter/x_tables.h`). Moreover, the iptables (Xtables) framework can add features (extensions). These features are implemented as kernel modules that register against this framework.

The network namespace object contains IPv4- and IPv6-specific objects (`netns_ipv4` and `netns_ipv6`). Both of these objects contain pointers to `xt_table` objects. For IPv4, in structure, `netns_ipv4` is, for example, `iptable_filter`, `iptable_mangle`, `nat_table`, and more.



struct xt_table
unsigned int valid_hooks
struct module *me
u_int8_t af
int priority
const char name[XT_TABLE_MAXNAMELEN]

Figure 3.5: Structure `xt_table` with important attributes.

The following is a description of some of the main attributes of the `xt_table` structure:

- `valid_hooks` – the bitmask that may contain zero or more of the five flags matches the hook points in 3.2.1.

- `me` – This is used for the Linux kernel infrastructure. It serves for reference counting so that the module is not unloaded while a rule exists. It is set to `THIS_MODULE` or `NULL`.

- `af` – Protocol family (`NFPROTO_IPV4` for IPv4 or `NFPROTO_IPV6` for IPv6).

- `priority` – Same as a priority in section 3.2.2

- name – Unique name of the table XT_TABLE_MAXNAMELEN is 32 characters, including '\0', which leaves 31 characters for the name of a table.

Kernel module initializing function needs to call xt_register_table() function. This function is called on module loading. Later, when unloading the module, the table must be unregistered again. The table is unregistered by calling xt_unregister_table() function [9].

3.4.1 Target extensions

Before showing the implementation of SNAT using iptables, there is a need to describe how to define target extensions.

The target is in iptables represented as xt_target structure. This structure is defined in include/linux/netfilter/x_tables.h. Here is the description of the xt_target structure without internal fields (with parameters structures):

```
struct xt_action_param {
        const struct xt_target *target;
        const void *targinfo;
        const struct net_device *in, *out;
        unsigned int hooknum;
        uint8_t family;
};
struct xt_tgchk_param {
        const char *table;
        const void *entryinfo;
        const struct xt_target *target;
        void *targinfo;
        unsigned int hook_mask;
        uint8_t family;
};
struct xt_tgdtor_param {
        const struct xt_target *target;
        void *targinfo;
        uint8_t family;
};

struct xt_target {
        const char      name[XT_EXTENSION_MAXNAMELEN];
        uint8_t         revision;
        unsigned short  family;
        const char      *table;
        unsigned int    hooks;
        unsigned short  proto;

        unsigned int    targetsize;
        unsigned int (*target)(struct sk_buff *skb,
                              const struct xt_action_param *par);
        int (*checkentry)(const struct xt_tgchk_param *par);
        void (*destroy)(const struct xt_tgdtor_param *par);

        struct module *me;
};
```

The number of arguments to the functions has grown over time, and it became a long process to update all of them whenever an API change was required. Moreover, many extensions do not use all parameters. Thus, the parameter structures were created and named `struct xt_*_param` to collect all the arguments. The `xt_action_param` are parameters for targets' callbacks. The `xt_tgchk_param` are parameters for target extensions' checkentry functions. The `xt_tgdtor_param` are parameters for target destructor [8].

The following is a description of essential attributes of the `xt_target` structure:

- `name` – Unique name of the table `XT_EXTENSION_MAXNAMELEN` is 29 characters, including '\0', which leaves 28 characters for the name of a target.
- `revision` – An integer that can be used to denote a „version" or feature set of a given match.
- `family` – Type of family the `xt_target` structure handles. Protocol family (`NFPROTO_IPV4` for IPv4 or `NFPROTO_IPV6` for IPv6).
- `table`, `hooks`, and `proto` – These fields can limit where the match may be used. No table, hook, or protocol restriction will be applied if the field is not provided.
- `targetsize` – This field specifies the size of the private structure
- `target` – Method that is called when a packet is passed to the module.
- `checkentry` – Called when inserting rule.
- `destroy` – Called when removing rule.

Kernel module initializing function needs to call `xt_register_target()` function. This function is called on module insertion. Later, when removing the module, the target must unregistered again. The target is unregistered by calling `xt_unregister_target()` function [8, 9].

Each rule can be assigned a target, which can be seen as an „action" that is to be done. Target extensions need to return a verdict. Depending on the nature of the target, either `NF_ACCEPT` or `NF_DROP` is chosen for terminating targets.

Possible verdict return values for the function are:

- `XT_CONTINUE` – used for targets that do not cause traversal to stop.
- `NF_DROP` – stop traversal in the current table hook and indicate packet drop.
- `NF_ACCEPT` – stop traversal in the current table hook and indicate packet acceptance.
- `XT_RETURN` – return to the previous chain or default chain policy [8].

In the following section is an description of ´the userspace plugin.

3.5 Userspace plugin

The idea for the output of this work is to implement the iptables rule that will do the PBA algorithm. However, this rule needs parameters for its initialization. A userspace program (plugin) must be implemented to parse this rule and react to iptables commands. The primary responsibility of the userspace program is to fill the private structure with the values provided by a user of rule parameters. Moreover, an iptables extension is a tool to interact with the user.

The vital part of the userspace program is the structure `struct xtables_target` that defines the vtable[1] for one address family of a target extension. It is available from `xtables.h`. The following structure description includes only the crucial parameters or the most frequently used.

```
struct xtables_target {
    const char *version;
    const char *name;
    uint8_t revision;
    uint16_t family;

    size_t size;
    size_t userspacesize;

    void (*help)(void);
    void (*init)(struct xt_entry_target *target);
    int (*parse)(int c, char **argv, int invert, unsigned int *flags,
            const void *entry, struct xt_entry_target **target);
    void (*print)(const void *entry,
            const struct xt_entry_target *target,
            int numeric);
    void (*save)(const void *entry,
            const struct xt_entry_target *target);
    const struct option *extra_opts;
};
```

The following is a description of essential attributes of the `xtables_target` structure:

- `version` – is always initialized to `XTABLES_VERSION`. This value avoids loading old modules with newer, potentially incompatible iptables versions.
- `name` – specifies the name of the module. It has to match the name set in the kernel module.
- `revision` – specifies that this `xtables_target` can only be used with the same-revision Xtables target kernel module.
- `family` – specifies what IP family this target operates on. The available variants are IPv4 (`NFPROTO_IPV4`), IPv6 (`NFPROTO_IPV6`), or `NFPROTO_UNSPEC`, which acts as a wildcard.

[1]A virtual function table is a mechanism used in a programming language to support run-time function binding.

- `size` – specifies the size of the private structure in total. The kernel module will get this structure as a parameter in its functions.

- `userspacesize` – specifies the part of the structure relevant to rule matching when replacing or deleting rules.

- `help`- - whenever a user enters command `'iptables -m module -h'`, this function is called. It should show the available options and a brief description.

- `init` – it can initialize the private structure with default values before the `parse` function is called.

- `parse` – is called a new rule is entered (added into the iptables). It has to validate the arguments. This function is vital because it verifies if the arguments are used correctly.

- `print` – it aims to print information about the rule. The command `'iptables -L'`, calls this function. The function is similar to the `save` function, but its output might be whatever we want.

- `save` – it has to interpret the private structure, where rule parameters are stored. The produced output must be options as can be passed to iptables.

- `extra_opts` - an array of elements that specify the extra options for the target rule. Every option is mapped to a single user-defined parameter using the structure `struct option` from `<getopt.h>` library.

The attributes `parse` and `extra_opt` have newer variants (with `x6_` prefix) and allow easier parameter parsing. It is possible to omit the `init`, `print`, and `save` members (functions). However, the `help` and `parse` functions must be defined. Every userspace plugin must register to the iptables program (or ip6tables) by calling `xtables_register_target`. This function is wrapped in the `_init` function that is called when iptables load the module [8].

In the following section is an implementation description of the SNAT module.

3.6 NAT

The Network Address Translation (NAT) module deals mostly with IP address translation or port manipulation. As said in chapter 2, one of the most common uses of NAT is to enable a group of hosts with a private IP address on a Local Area Network (LAN) to access the Internet via some residential gateway. It can be achieved by setting a NAT rule. The Netfilter subsystem has NAT implementation for IPv4 (in `net/ipv4/netfilter/iptable_nat.c`) and IPv6 (in `net/ipv6/netfilter/ip6table_nat.c`).There are many types of NAT setups. The two common configurations are SNAT (source NAT), where the source IP address is changed, and DNAT (destination NAT), where the destination IP address is changed. SNAT or DNAT can be selected by the `-j` flag in the rule. The implementation of both DNAT and SNAT is in `net/netfilter/xt_nat.c` [8].

3.6.1 SNAT Example Implementation

In this example, The SNAT implementation with `revision = 1` is described. This module requires a range of addresses and possible port range that the NAT engine may use. The SNAT target is only valid in the `nat` table. The `nf_nat_range` structure represents the range:

```
struct nf_nat_range {
        unsigned int                    flags;
        union nf_inet_addr              min_addr;
        union nf_inet_addr              max_addr;
        union nf_conntrack_man_proto    min_proto;
        union nf_conntrack_man_proto    max_proto;
};
```

The structure contains the IP address range as two IP addresses with the lowest and highest values. The port range is represented the same way. Attribute `flags` indicate how the address mapping uses the ranges.

The SNAT target is represented as the `xt_target` object. It is registered on two hooks `NF_INET_POST_ROUTING` and `NF_INET_LOCAL_IN`:

```
static struct xt_target xt_nat_target_reg[] __read_mostly = {
        ...
        {
                .name           = "SNAT",
                .revision       = 1,
                .checkentry     = xt_nat_checkentry,
                .destroy        = xt_nat_destroy,
                .target         = xt_snat_target_v1,
                .targetsize     = sizeof(struct nf_nat_range),
                .table          = "nat",
                .hooks          = (1 << NF_INET_POST_ROUTING) |
                                  (1 << NF_INET_LOCAL_IN),
                .me             = THIS_MODULE,
        },
        ...
};
```

The array `xt_nat_target_reg` does contain more revisions of SNAT and DNAT implementations. Registration and unregistration of targets is done by calling `xt_nat_init()` and `xt_nat_exit()` methods.

```
static unsigned int
xt_snat_target_v1(struct sk_buff *skb, const struct xt_action_param *par);
```

The most important function is `xt_snat_target_v1()` which is called for every packet that matches the chain of match modules. The first step is to get `nf_nat_range` object from `xt_action_param *par`. Then the function will get connection tracking (`nf_conn` structure) and tuple hash (see chapter 3.3) for the `skb` parameter. The function will pass the connection tracking structure, IP address, port range, and type of IP address alteration (`NF_NAT_MANIP_SRC` for SNAT) to the `nf_nat_setup_info()` function.

The `nf_nat_setup_info()` method tries to set up an info structure to map into a given range using Connection tracking. Firstly, the method checks if the info structure has already been created for the `ct` structure. If it has, the method will return `NF_ACCEPT`. Then the method calls the `get_unique_tuple()` function to obtain an external IP address and port number (in the case of UDP). The tuples in the Connection tracking are then updated with the new external IP address and port number. Because the tuples were changed, the hash value of the tuples will no longer point to the correct entry in the Connection tracking module's hash table. So, the hash value is recalculated, and the Connection tracking status is moved accordingly [6].

3.7 Netlink module

The Netlink is a socket-based protocol for Inter-Process Communication (IPC), and it is based on RFC 3549 („Linux Netlink as an IP Service Protocol“). This mechanism allows a bidirectional communication channel between userspace and the kernel or among some parts of the kernel itself. It is an extension of the standard socket implementation. The connection tracking code implements Netlink API (next used as CT Netlink API), which can notify about changes in the records in the connection tracking table. (reference: Linux kernel networking - implementation and theory)

The userspace has Netlink libraries that can receive notifications from the connection tracking module in the kernel. This library can even edit the connection tracking table or create new records (connections) [9].

Many programs can use this library, for example:

- `ulogd2` - userspace logging daemon for netfilter/iptables related logging
- `conntrack-tools` - is a set of tools targeted at system administrators. They are `conntrack`, the userspace command-line interface, and `conntrackd`, the userspace daemon. Using the `conntrack`, one can view and manage the in-kernel connection tracking state table from userspace. Moreover, the `conntrackd` covers the specific aspects of stateful firewalls.

This chapter describes vital resources and tools to design and implement the solution to this work. The next chapter shows the research process of resources and their issues that were directly used in the design and later in the implementation.

Chapter 4

Design

This chapter addresses the research of current resources, which were essential for the functionality and implementation of this thesis. The work must implement the Port Block Allocation algorithm (used next as PBA) in the Linux system (CentOs 7). Programs started to be created for kernel version 5.4.53, which was changed to version 5.4.186 during implementation. All research and implementation were for this system and kernel versions. Fortunately, there are no differences in functionality between versions that would affect the implementation of the module.

The idea was to use resources that are available in the Linux system. For example, the kernel of the system already implements SNAT. So, the implementation in this thesis tries to use some of its parts. The design of the solution was created simultaneously with this research, so it would be possible to implement it in the kernel.

4.1 Developing and testing setup

The work was developed and tested on virtual computers. The figure 4.1 shows how these computers are connected.

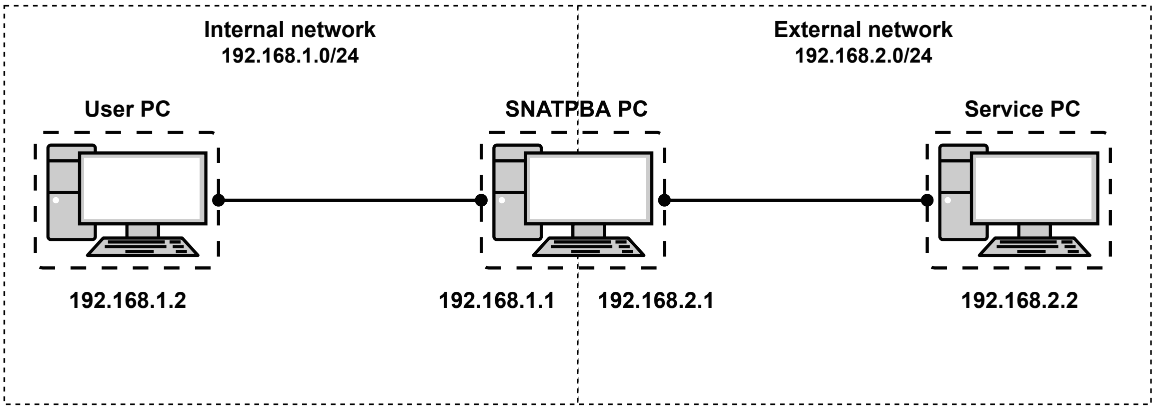


Figure 4.1: The virtual PC setup is used to develop and test the PBA module.

The user computer with IP address 192.168.1.2 is located in the internal network (192.168.1.0/24) and connects to the network interface of the SNATPBA PC with IP address 192.168.1.1. The service computer with IP address 192.168.2.2 is located in the external network and is connected to the network interface with IP address 192.168.2.1 of SNATPBA PC. The user PC initiates connections to the service PC. These connections go through SNATPBA PC, which performs address translation. The behavior of the SNAT rule was tested before the implementation of the PBA module.

4.2 Issues and their possible solutions

The first part of the implementation (and design) is a basic structure. When looking at iptables rules, the SNAT rule is the closest to the PBA algorithm. The SNAT is implemented as a target rule, and logically the PBA should be implemented the same way. Like the SNAT rule implementation, the PBA structure will consist of a user program and a kernel module. The userspace program process iptables commands. The most important part of this program is to parse the PBA rule when added into iptables and sends its parameters to the kernel module. The kernel module receives parameters for specific rules. When a packet triggers one of the rules, the target function (see target functions) will perform the PBA algorithm.

An important part of the design is to determine which parts of SNAT can be used in the implementation of this work. Because it is the SNAT rule, only the station in the internal network can initiate a connection. It implies that the first packet originating the connection will trigger the target function. When activated, the target function will create connection tracking for the connection. After this, the `'nf_nat_setup_info()'` method creates the mapping and accepts parameters: the initialized connection tracking, pool of IP addresses and ports, and the direction of NAT. The pool, a parameter of the SNAT rule, can be an address range or a single IP address and a port range or a single port. IP address with a port from this pool is assigned to the connection, and the SNAT will map the IP address from the internal network to the assigned IP address and port.

After consultation with the work supervisor, we decided that the PBA algorithm, at minimum, requires: a pool of IP addresses from an external network and the size of one block. The block (IP address and port range) represents maximum simultaneous connections between one IP address from the internal network and various stations in the external network (next used as a group of connections). Another block represents connections from a different internal IP address.
Here is what the PBA rule looks like:

```
iptables -t nat -A POSTROUTING -p <protocol> -s <source> -d <destination>\
  -j SNATPBA --to-source <address and port pool>\
  --block-size <block_size>
```

The PBA module as the target rule is named SNATPBA. Description of parameters of SNATPBA target:

- `to-source` – is IP addresses and port to which the internal source IP addresses will translate. It expects a single IP address (without a network mask) or a range of addresses. It does not require port range like the SNAT rule does.
- `block-size` – is the number of available ports for one internal source IP address. It represents the maximum number of simultaneous connections from a single internal source IP address.

Like SNAT, the PBA rule (and module) requires protocol specification (`-p` in match part of the rule). All parameters above are mandatory, and a user must provide them.

Example what the PBA rule might look like:

```
iptables -t nat -A POSTROUTING -p TCP -s 192.168.1.0/24 -d 192.168.2.0/24
  -j SNATPBA --to-source 192.168.2.1-192.168.2.2 --block-size 20
```

This rule matches TCP packets that initiate a new connection from the internal network 192.168.1.0/24 to the external network 192.168.2.0/24. Then send them into the SNATPBA target that maps IP address with the port from the pool (192.168.2.1-192.168.2.2) as the source IP address and sends these packets into the external network 192.168.2.0/24.

The goal here was to use the `nf_nat_setup_info()` function without the necessity to change the internal working of NAT in the kernel. The target function will get an IP address with a port range from the pool, which is the size of one block, and pass it as the argument to the `nf_nat_setup_info()` function. Next is how the target function will get the required IP address and port range.

After adding the rule in iptables, the userspace program will call the `checkentry()` function with these rule parameters after parsing the parameters. The function can initialize all required data structures, including all blocks (IP and port range), and put these created blocks into the linked list. Then these created blocks can be taken from the front of this list and are assigned to the internal-external connections in the target function. When all the connections for the particular internal IP address are closed, the block is put back at the end of the linked list.

The target function assigns one of the created blocks to the new internal-external connection and inserts it into the hash table for a fast lookup. All the following connections for this particular internal IP address will be linked to the hash table record. The internal source IP address is used as a hash table key. The station (user) from the particular internal network has limited simultaneous connections to an external network with this hash table key.

The problem with designing and implementing the PBA module is that it assigns a specific block (external source IP address with port range) to the particular internal source IP address. That means the PBA module must keep track of these assignments. The SNAT module only assigns the external source IP address with a port (from the provided pool) to every new connection via the `nf_nat_setup_info` function. However, the Connection tracking module (implemented in the kernel) tracks these assignments. The main problem is removing the connection (and address mapping) from the Connection tracking. The Connection tracking stores the connection, and the SNAT module does not need to be notified about its removal (when the connection is closed). Nevertheless, the PBA module requires notification about this action.

The connection tracking module generates several events: `NEW`, `RELATED`, `DESTROY`, and others. The PBA module only requires two:

- `NEW` - When the Connection tracking saw a new connection and created a new record.
- `DESTROY` – When the entry in connection tracking expires, the Connection tracking removes this entry.

The target function assigns a block to the group of connections from the same internal source IP address and inserts it into the hash table. However, at this time, the new connection is still not established. When the Connection tracking starts tracking this new connection, it generates and sends the `NEW` event to the notifier. Now, the block is in the hash table, and the connection has an assigned port from the block. The PBA module receives a notification with this `NEW` event and now knows that one port from the block is not available. If the connection has not been established and is removed from connection tracking, it waits for the `DESTROY` event. The Connection tracking does not immediately send the `DESTROY` event because there are timers set that must expire first. The PBA module needs the `NEW` event to decrease the 'still free ports block' counter assigned to the internal source IP address for other connections from the same IP address. The `DESTROY` event is required to increment the 'still free ports block' counter. When the counter indicates that all block's ports are free, the block is removed from the hash table and returned to the list of available blocks.

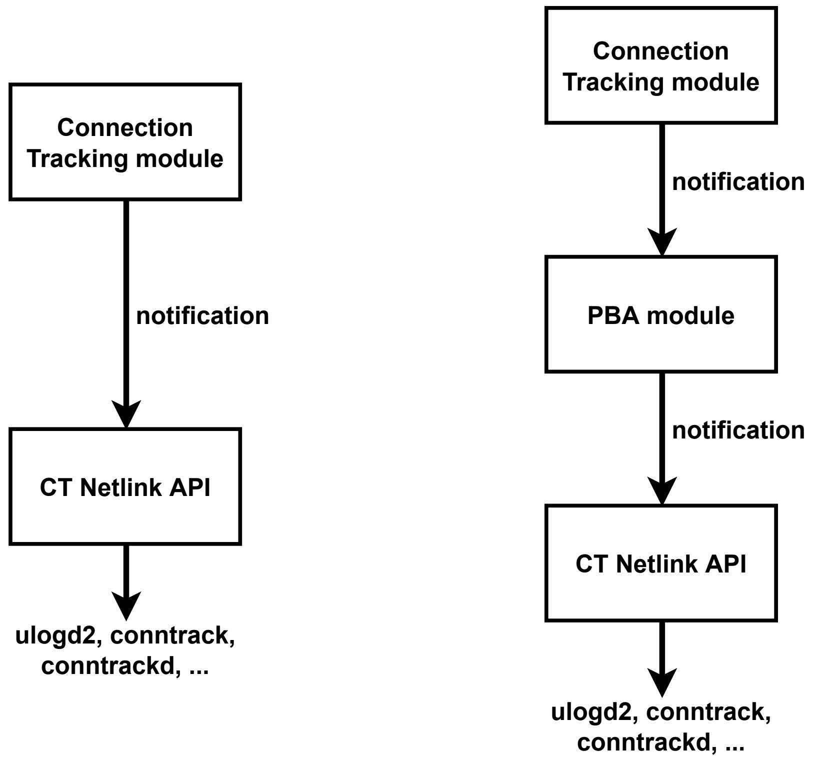

Connection Tracking module
notification
CT Netlink API
ulogd2, conntrack, conntrackd, ...
Connection Tracking module
notification
PBA module
notification
CT Netlink API
ulogd2, conntrack, conntrackd, ...

Figure 4.2: Left: The CT Netlink is registered as notifier to the Connection Tracking module. Right: Between the Connection Tracking module and the CT Netlink is inserted the PBA module.

As described in section 3.7, the PBA module must register to the CT Netlink API. However, there is a problem. Before kernel version 2.6.31, this API provided a notifier chain that other kernel codes could use to get notified about changes in the connection tracking state table. In the following versions, the implementation was changed. This new implementation did not use a notifier chain but only one notifier. The module that requires the notification defines a callback function. The module then registers the callback as a notifier, which the Connection tracking then calls when there is any change in the connection tracking state table. The notifier chain was removed because there was only one module registered in this chain, and it is CT Netlink, and the notifier chain added too much overhead for only a single client (module). Unfortunately, the result of this change is that there is no simple way to register a notifier for an out-of-tree kernel module. The left part of figure 4.2 shows this case [1].

Luckily, the implementation of the `ipt_NETFLOW` [4] kernel module used a little workaround to get notifications from the connection tracking module and, at the same time, send these notifications to the CT Netlink. This workaround is described in detail later in the implementation chapter 5.1. The constriction of this solution is that the CT Netlink must register as a notifier before the PBA module. However, the implementation takes this into account.

Because the PBA module needs to keep track of the connections from the same internal source IP address, it stores necessary information in this module's global variables. However, this brings a new problem. The PBA module is initialized before a user adds any SNATPBA rules into the iptables. So there is only one active PBA module for all SNATPBA rules. Unfortunately, it means there cannot be only one hash table and list of available blocks. The module has a global list that contains these data structures.

4.3 Summary of how the module works

This section describes how the PBA works after considering all these problems and their solutions.

4.3.1 Adding a new connection

The figure 4.3 shows the functionality of adding a new connection to the PBA module. When a packet from an internal network activates the PBA module's target function, it is the initial packet of a new connection, whether it is already in a created group of connections from the same internal source IP address or belongs to a new group.
Here are described individual steps:

- **Step 1:** The first thing the target function does is to find which rule this packet belongs to. It is a simple task because the target function has the packet information and rule parameters from the userspace program.
- **Step 2:** Next, the function will get the key used in the hash table to find the group of connections from the same internal source IP address where a packet belongs to.
- **Step 3:** The function will try to find the record in the hash table for all active connections from the internal source IP address.
- **Step 4.1:** If a record exists and it is in the hash table, the function will use the address with port range to map the original source IP address in the received packet to the external IP address with port. *The significant part of the functionality is that the module does not know (at execution of the target function) if there are any available ports in the found record. There is a maximum number of simultaneous connections for the group of connections from this particular internal source IP address. If there are no available ports (address translation was not created), the PBA will not receive any notification.*
- **Step 4.2:** If this record does not exist in the hash table, the function will try to create a new hash table record for a new group of connections from the internal source IP address, where the received packet belongs. If there are no available blocks, the target function will return `NF_DROP`. If there is at least one available block, it is put into a new hash table record and removed from the list of available blocks and inserts the

record into the hash table. The block in this record is used to map the source IP address in the received packet to the external IP address with port (from the block).

- **Step 5:** The target function will now try to create a new record in the connection tracking, but this new record needs to change the mapping from the internal source IP address to the external source IP address. At this point, the target function has a block (address with the range or ports) and can call`nf_nat_setup_info()` function. This function change created a connection tracking record, so the new external source IP address is from the provided block.

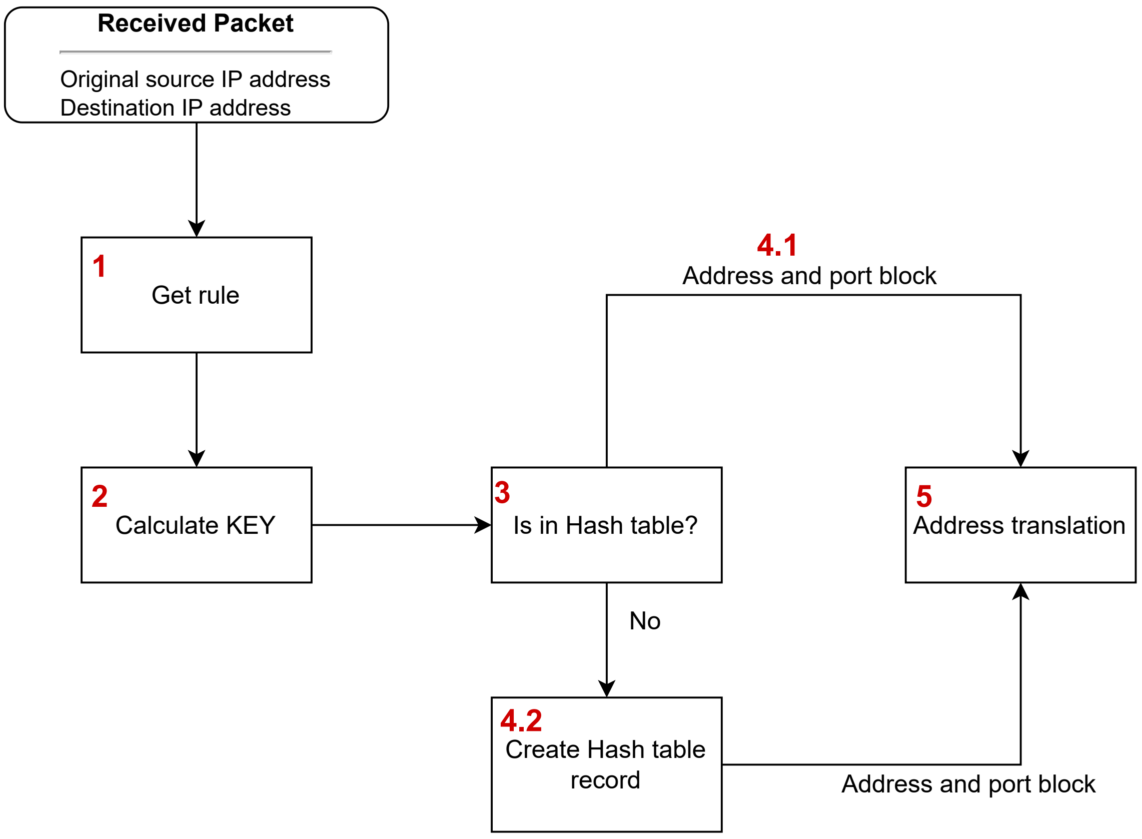


Figure 4.3: Description of adding new group of internal-external connection.

When the PBA module creates a new record and assigns a block, the module logs this action into the system logging daemon, so it is possible to track any connections that used any port from the port range of this block. The following is the format of this log message:

```
[timestamp]: ipt_SNATPBA: add:<orig_src_ip>:<new_src_ip>:<min_p>-<max_p>
```

Description:

- `timestamp` – the PBA module does not fill this field. The system logging daemon fills it.

- `orig_src_ip` – the original internal source IP address. From this address is initiated a new connection.

- `new_src_ip` – the external IP address from the address pool. To this IP address is mapped the `orig_src_ip`.
- `min_p` – the lowest port from the port range of the block.
- `max_p` – the highest port from the port range of the block.

The following section describes how the module receives and processes the notification from the Connection tracking module.

4.3.2 Processing of notifications

The PBA module has a registered callback function as a notifier in the Connection tracking module. The figure 4.4 shows how the function processes the received notifications:

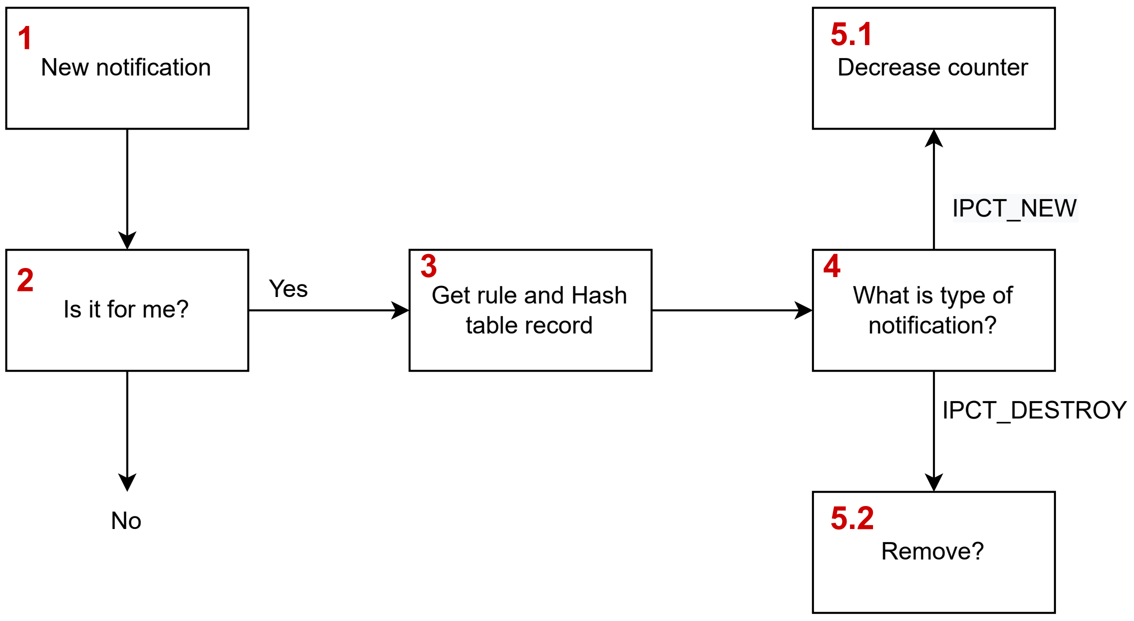


Figure 4.4: Diagram of how the PBA module processes received notifications from the Connection tracking.

- **Step 1:** When this callback function receives a notification, first, it sends it to the CT Netlink API, so all userspace programs that use this API work. For example, `conntrack-tools`.
- **Step 2:** This function receives all notifications that the Connection tracking module sends. It must determine if the received notification is for the PBA module. Firstly it checks the type of received notification. PBA module processes only `IPCT_DESTROY` and `IPCT_NEW` notifications. If the notification is one of these two types, the method will continue with Step 3. Otherwise, it will ignore the notification.
- **Step 3:** Now, the function needs to find the notification's rule and then the appropriate record in the hash table. Determining the appropriate rule is quite simple. Every notification is about one record in the Connection tracking. The function can

compare the external source IP address with the rule address pool, and if this address falls into this pool, then the notification belongs to this rule. Then the function needs to find the record in the hash table.

- **Step 4:** This step only again checks the type of notification. *It is more effective to check the type of notification and then find the rule. If it were the other way around, then every notification would have to check against every rule in the list where rules are stored.*
- **Step 5.1:** The `IPCT_NEW` notification means that the Connection tracking module created a new record for a new connection belonging to a group of connections that are from the same internal source IP address. However, the connection did not need to be established. If the counter value of available ports is greater than zero, the function will decrement this counter. The situation when the counter is equal to zero cannot occur.
- **Step 5.2:** The `IPCT_DESTROY` notification means an existing connection was closed, and the Connection tracking module removed its record. It also means that the port that was occupied by the dropped connection is now available. So, the PBA increments the counter of the available ports. After this, if the counter has a maximum value (all ports are available), there are no active connections in this group of connections from the same internal source IP address, and the PBA removes the record from the hash table.

When the PBA removes a cell from the hash table, the block is no longer assigned to the group of connections from the same internal source IP address, and the module logs this action into the system log daemon. The format of this log message and the fields were described in the section about the add log message:

```
[timestamp]: ipt_SNATPBA: del:<orig_src_ip>:<new_src_ip>:<min_p>-<max_p>
```

The following section discusses implementing the userspace program that communicates with the iptables and the kernel module.

Chapter 5

Implementation

The previous chapter described the design of this thesis output, including problems and their possible solutions. This chapter discusses the implementation in more detail, describes the implementation of solutions for these issues, and starts with the notifier registration in the Connection tracking module. The work implementation was done in the C programming language.

5.1 Registering notifier

The first part of the implementation was registering the PBA module to the Connection tracking as a notifier. As was written in the section 4.2, the implemented solution in project `ipt_NETFLOW` solves the problem with the option to register only one notifier (on callback function).

The `register_ct_events` method that registers the module as a notifier was put into the `__init` function. By analogy, the `unregister_ct_events` method that unregisters the module was put into the `__exit` function. Every kernel module has at minimum these two functions. The module is initialized/uninitialized only once.

The `register_ct_events` function firstly ensures that the CT Netlink API module (in the system, the module's name is `nf_conntrack_netlink`) is loaded (and initialized) before the PBA module. The CT Netlink module registers itself as a notifier in the Connection Tracking module at initialization. If the CT Netlink module did not register itself before the PBA module, all userspace programs that use this library would not work. Moreover, if the CT Netlink tried to initialize as a notifier later, it would fail because the PBA module would already be registered as a notifier, and the callback function would be occupied by it.

After loading the CT Netlink module, the `register_ct_events` function registers the `set_notifier_cb` as initialization function and `unset_notifier_cb` as exit function. These functions are put into structure `struct pernet_operations`. Calling the `register_pernet_subsys` method will register these functions. It is not crucial for registering the notifier, but it allows initializing the module in different network namespaces. The `unregister_ct_events` method contains the `unregister_pernet_subsys` (a clean-up) method and releases the CT Netlink module.

The function `set_notifier_cb` implements the actual registration of the callback function (`snatpba_conntrack_event`) as a notifier. Firstly, it checks if the CT Netlink notifier is already registered. If it is not, the `snatpba_conntrack_event` function is registered.

Otherwise, the PBA module stores the pointer to the CT Netlink's callback function into a different variable, and the `snatpba_conntrack_event` function is registered instead. After this, the PBA module starts receiving the Conntrack tracking notifications.

The `unset_notifier_cb` function does reverse operations as the `set_notifier_cb` function. It unregisters the callback of the PBA module. If the callback of the CT Netlink module were registered as a notifier, it would be registered back into the Connection tracking module.

The following section describes details of the userspace program implementation.

5.2 Userspace program

Because the SNATPBA rule is added (or deleted) into (from) the iptables, it needs a userspace program that will implement iptables commands. When I was implementing the userspace program, I took inspiration from the userspace program implementation of the SNAT rule.

The userspace program and the PBA module use the private structure `struct xt_snatpba_info`. The userspace program fills this structure with data from rule parameters, and then it is given as an argument to the kernel module functions.

```
struct xt_snatpba_info {
    __u8                                  options;
    struct in_addr                        from_src_in;
    uint8_t                               from_src_mask;
    struct nf_nat_ipv4_range              from_src;
    struct nf_nat_ipv4_multi_range_compat to_src;
    __u32                                 block_size;
};
```

The following is a description of the `xt_snatpba_info` structure:

- `options` – the mask of options (parameters) that the userspace program gives to the PBA module.
- `to_src` – the range of IPv4 addresses (address pool) to which the PBA module will map the internal source addresses.
- `block_size` – is the size of one block in the kernel-side module (maximum number of available ports in the group of connections from the same internal source IP address).

As described in section 3.5, the structure `struct xtables_target` is one of the most crucial parts of the userspace program (for target extension). The structure is in this work initialized as follows:

```
static struct xtables_target snat_pba_tg_reg = {
        .version        = XTABLES_VERSION,
        .name           = "SNATPBA",
        .family         = NFPROTO_IPV4,
```

As written before, the `value` of the version field is always `XTABLES_VERSION`. The `name` must be the same as the kernel-side extension module. Moreover, this name specifies the target in the Iptables commands (for example: `'iptables ...  -j SNATPBA ...'`). The `family` is set to `NFPROTO_IPV4` because this extension only works with IPv4 addresses in the internal network.

```
        .size           = XT_ALIGN(sizeof(struct xt_snat_pba_info)),
        .userspacesize  = XT_ALIGN(sizeof(struct xt_snat_pba_info)),
```

The size and userspacesize attributes are both set to the exact size of the structure struct `xt_snatpba_info` because no kernel-private fields should be omitted from the comparison. The module usually needs to omit the fields that hold some global data. However, it is not needed for the SNATPBA rules because the PBA module stores these global data.

```
        .help           = SNAT_PBA_help,
        .x6_parse       = SNAT_PBA_parse,
        .print          = SNAT_PBA_print,
        .save           = SNAT_PBA_save,
```

The purpose of these functions was described in chapter 3.5. Their values are pointers to functions which iptables call at appropriate time:

- `help`: `'iptables -j SNATPBA -h'`
- `print`: `'iptables -t nat -L -n -v'`
- `save`: `'iptables -t nat -S'`

The `x6_parse` (`SNAT_PBA_parse`) function is newer variant of the `parse` function. It takes argument `struct xt_option_call`, which at every call of the `SNAT_PBA_parse` contains all important information about the currently parsed rule parameter. The xtable automatically parses the rule parameters. The `'--block-size'` is just an unsigned integer.

Because the xtables parser cannot parse range of IP addresses (only range or ports), it needs to be parsed manually as a string. Parsing of the `'--to-source'` parameter is done in the `parse_to_src` function. The parse function in the xtables SNAT extension inspired the implementation of this method. First, it checks if the string contains a dash (`'-'`), separating IP addresses in the range. If it does not have a dash character, the function uses the xtables function (`xtables_numeric_to_ipaddr`) to convert the string to IP address. If it has a dash character, the method splits the string into two strings and converts them from strings to IP addresses.

```
        .x6_options     = SNAT_PBA_opts,
};
```

The `x6_options` is a pointer to the `struct xt_option_entry` structure array that specifies parameters and expected values. The array has two elements because the SNATPBA rule has two parameters. Every element has a name, which has the same format as the rule parameter; id (a unique value), which the program uses to recognize a parameter in `parse` and `print` functions. Every element has a type that defines how the xtables parser will parse a parameter (for example, the option `XTTYPE_HOSTMASK` specifies that the xtables will parse a parameter as an IPv4 address with a mask). The flags field might define if the parameter is, for example, invertible, mandatory, and others. Because the PBA module requires all described parameters, the flags field has a value `XTOPT_MAND` (mandatory) value. An example of how is defined one element (`'-to-source'` parameter) in this work:

```
    {.name = "to-source", .id = O_TO_SRC, .type = XTTYPE_STRING,
            .flags = XTOPT_MAND}
```

This section describes the implementation of the PBA userspace program. The following section discusses the implementation of the PBA's kernel-side module.

5.3 Kernel module

After finishing the userspace program, the behavior of the SNATPBA rule must be implemented, which means implementing the kernel-side module. The first part was to implement the registration of the module into the Connection tracking as a notifier. The finished code for this registration is in section 5.1.

Then I implemented the target extension, which the iptables call when a user adds the SNATPBA rule. Chapter 3.4.1 shows what is needed to make the Xtables target extension. First, the `struct xt_target` must be set:

```
static struct xt_target xt_snatpba_target_reg[] __read_mostly = {
    {
        .name       = "SNATPBA",
        .checkentry = xt_snatpba_checkentry,
        .destroy    = xt_snatpba_destroy,
        .target     = xt_snatpba_target,
        .targetsize = sizeof(struct xt_snatpba_info),
        .family     = NFPROTO_IPV4,
        .table      = "nat",
        .hooks      = (1 << NF_INET_POST_ROUTING) |
                      (1 << NF_INET_LOCAL_IN),
        .me         = THIS_MODULE,
    },
};
```

The `name` is set to „`SNATPBA"`, which is the same value as the `name` in the userspace program. The `checkentry`, `destroy`, and `target` fields are pointers to `xt_snatpba_checkentry`, `xt_snatpba_destroy`, and `xt_snatpba_target` functions, which are described below. The `targetsize` field is the size of the private structure `struct xt_snatpba_info`, which is used in this module, and the userspace program (where it is filled). This structure is a parameter in this module's target extension specified functions. The PBA module is restricted to the IPv4 address family and the „`nat"` table in the iptables. The `hooks` field is set to the `POSTROUTING` and `LOCAL_IN` hook points.

For basic testing, if the PBA module is successfully starting and ending, the `checkentry` and `destroy` functions might be omitted. In the `__init` function must be called function `xt_register_target`, to register this target, and the `__exit` function must be called function `xt_unregister_target` to unregister this target extension.

After this, when the SNATPBA rule is added to the iptables, the userspace program will parse set parameters. The PBA module is initialized (if it is not already), which registers the PBA module callback function in the Connection tracking module.

5.3.1 Saving of rules

Chapter 4.2 mentions that the PBA module is initialized only once. However, there might be multiple rules that use this module, and all of them need to track assigned blocks to groups of connections from the same internal source IP addresses separately. That implies that the PBA module needs a data structure as a global variable, where individual rules are stored. I chose the kernel implementation of the linked list as this data structure.

The kernel uses the generic API to manipulate data structures such as linked lists or hash tables. The following description of the kernel's linked list behavior will be presented on the structure that represents one SNATPBA rule.

```
struct rule_entry {
        DECLARE_HASHTABLE(rule_hashtable, 10);
        struct list_head        avl_blc_list;
        struct list_head        all_blc_list;
        struct xt_snatpba_info  info;
        struct list_head        list;
};
```

The following is a description of the structure `struct rule_entry`, but for an explanation of the kernel's linked list is the essential `list` field:

- The first field is the declaration of kernel hash table, used for currently assigned blocks to groups of connections from the same internal source IP addresses. The kernel's hash tables' behavior will be explained later.
- `avl_blc_list` - the linked list of available blocks that can be assigned.
- `all_blc_list` - the linked list of all allocated blocks used only for easy deallocation.
- `info` - the information about the SNATPBA rule.
- `list` - the same structure as at fields `avl_blc_list` and `all_blc_list`, but it is used to connect the `rule_entry` elements in a linked list.

At first, this implementation is a little confusing because the more common implementation is to add a pointer into a structure, which points to the following similar structure in the linked list. In this approach, there is a need to write code to handle adding/removing/etc. elements for this structure. In kernel implementation, only the structure `struct list_head` field must be added to a structure in the linked list. The structure `list_head` only contains two pointers, prev and next, and their purpose is self-explanatory from their names.

In the `rule_entry` structure, the `avl_blc_list` and `all_blc_list` represent the whole linked lists, whereas the `list` field connects instances of the `rule_entry` structure into the `rule_list` list.

Now that the `rule_list` is initialized, the module can add the new rules. The `add_list` function does this.

So, when a new rule is inserted into the iptables, the `checkentry` (`xt_snatpba_checkentry`) function is called. Here, the new rule is initialized from data in the `xt_snatpba_info` structure and added at the end of the `rule_entry` linked list by the `list_add_tail` method.

5.3.2 Destroy function

The `xt_snatpba_checkentry` allocates blocks and adds rules into the PBA module. But, there is also a need to free the allocated data. That is why there is also the `destroy` function (`xt_snatpba_destroy`) that should deallocate data. However, the iptables does not add or remove rules. It replaces whole tables. This might cause problems when there is allocation in the `checkentry` function and deallocation in the `destroy` function. The following example describes how iptables adds and removes rules.

Let's already have 2 SNATPBA rules in iptables: `rule1` and `rule2`. When adding a new rule (`rule3`), the iptables creates a new inner table with all three rules, and then the old table is removed. This means that the `checkentry` function is called for all three rules, and then the `destroy` function is called for the first two rules:

```
After adding the rule3:
    checkentry - rule1
    checkentry - rule2
    checkentry - rule3

    destroy - rule1
    destroy - rule2
```

There are now three rules in the iptables. When removing `rule3`, the process is the same. The iptables creates a new table with `rule1` and `rule2` and then removes the old table.

```
After removing the rule3:
    checkentry - rule1
    checkentry - rule2

    destroy - rule1
    destroy - rule2
    destroy - rule3
```

Most iptables extensions and their attached kernel modules do not need to take appropriate actions because they do not perform data allocation and deallocation like the PBA module does.

Luckily, when I was searching for the solution to this problem, I found an implementation of the geoip[1] module. The solution for this problem in the module is quite simple. A new variable ref is added into the structure `struct rule_entry`, which represents one rule. This variable contains a number of references to this rule. When creating and adding a new rule, the ref is set to 1. When a rule is in iptables (and in the PBA kernel module), and the checkentry function is called for it, the ref variable is incremented by 1.

The `xt_snat_pba_destroy` function must free the rule's resources: all blocks, cells in the hash table, all elements of `all_blc_list`, and `avl_blc_list`, and the element representing the rule.

As was written before, the `all_blc_list` is used for the simple deallocation of created blocks. So, when the elements of lists and cells of the hash table are freed, the blocks have been deallocated.

[1] `https://fossies.org/linux/xtables-addons/extensions/xt_geoip.c`

5.3.3 Allocating blocks

Like the new rule, the blocks are created in the `xt_snatpba_checkentry` function. First, the `rule_entry` structure is allocated, and its lists and hash table are initialized. After that, the block must be calculated and initialized. The following is the structure `struct snatpba_block` that represents one block:

```
struct snatpba_block {
    struct nf_nat_ipv4_multi_range_compat   new_src;
    int                                     free_ports;
};
```

As said in chapter 4.2, the block consists of `new_src`, the IP address from the address pool (from the rule parameter `'--to-source'`), and the calculated port range. The number of ports in this range is the same as one block's size. The `free_ports` field shows how many ports are still available. The minimum value is zero, and the maximum value is the number of ports in one block. The crucial part of why blocks are created this way is that they have the same format as if the SNAT rule set them. And thanks to that, the `nf_nat_setup_info` function can use these blocks, and the nat core implementation creates mappings to ports in these blocks.

Not every port from all 65536 ports of one address is used for the blocks. The first 1024 (0 – 1023) system reserved ports are omitted. That implies that the first port used for the first allocated block is port 1024. All blocks are allocated using two nested for-cycles. The outer cycle iterates through a poll of IP addresses, and the inner cycle iterates through the number of blocks in one IP address. The number of blocks in one IP address:

$$blocks\ per\ IP = \frac{MAX\ PORTS\ PER\ IP}{block\ size},$$

where $MAX\ PORTS\ PER\ IP$ is always 64512 (65536 – 1024). Because the equation uses the integer division, there is a possibility that the PBA module won't use a few of the last ports (of a current IP address from the address pool).

The possible solution for maximum utilization of available ports to create blocks is to split the range in the block between two IP addresses. In the previous example a block might has ranges: `10.0.0.1:65524-65535`; `10.0.0.2:1024-1031`. The biggest issue with this approach is that the `nf_nat_setup_info` function does not accept these ranges (range of IP addresses with different port ranges). It implies using the `nf_nat_setup_info` function two times. The first time is called for the first range, and if that fails, it calls the function a second time with the second range.

Table 5.1 show example how are blocks allocated for rule with parameters: `--to-source= 10.0.0.1-10.0.0.2`, `--block-size=20`:

IP address	**Port range**
10.0.0.1	1024 - 1043
⋮	⋮
10.0.0.1	65504 - 65523
10.0.0.2	1024 - 1043
⋮	⋮
10.0.0.2	65504 - 65523

Table 5.1: Example of blocks allocation.

After allocating a block, it is inserted at the end of the `avl_blc_list` and `all_blc_list` linked lists. The pointer that points to the allocated block can be put only into two structures:

- `struct list_record` - an element of `avl_blc_list` and `all_blc_list`. This structure contains only two fields. The `block` field is a pointer to `struct snatpba_block`, and the `list` field is used to link other `list_record` elements into the linked list.
- `struct hashtable_cell` - a cell of the `rule_hashtable`, which is the hash table of one SNATPBA rule. One cell represents an active group of connections from the same internal source IP address. It means if a block is in this structure, it is assigned and occupied. The structure has more fields than struct `list_record`. It contains an attribute key, an unsigned int variable used to address cells (find bucket) in the hash table. The key is made from `orig_src_ip` (an original source IP address in the internal network). The `block` field is the same as in the structure `struct list_record`. And the `node` field is used to connect a cell that falls into the same bucket.

5.3.4 Hash table

The PBA module uses a hash table to keep track of assigning blocks to groups of connections from the same internal source IP address. A hash table is used to fast search a particular group (block).

The kernel implementation of hash tables uses the linked lists described above. Because the hash tables use a statistically defined array, the hash table size (number of buckets) must be specified at compile time. Unfortunately, it implies that the size of a hash table cannot be set at run-time (for example, as a parameter in the iptables rule). The size of a hash table is specified as a number of bites (size is always a power of two).

The definition of a hash table is creating an array, where one element of an array is one bucket of a hash table. The array elements contain kernel implemented linked lists to resolve collisions that might occur.

5.4 Testing

I tested the implementation on the setup that is described in chapter 4.1. The test aims to check that I can add the SNATPBA rule to the iptables on the SNATPBA PC. Another part of the test is to create maximum connections from the User PC (192.168.1.2) to the Service PC (192.168.2.2). The number of connections is limited by the number of ports in one block. So, it should not be possible to establish more connections than the number of ports in the block. The last part of the test shows how the assigned block is removed from the hash table.

```
$ make
$ make install
```

The PBA module is now installed, but it needs to run before I can add the SNATPBA rule to the itpables. I will start the module by command insmod. To check that the module is running is, I use the lsmod command:

```
$ insmod ipt_SNATPBA.ko
$ lsmod | grep ipt_SNATPBA
```

Later, when I want to stop the module, the command rmmod is used:

```
$ rmmod ipt_SNATPBA
```

Now, I can add a new SNATPBA rule into the iptables. I will use the same rule that I described in chapter 4.2. The difference between the rule used for testing and the described one is that I will use a smaller block size from value 20 to 2. That way, I would not have to create many connections (20), but only two. The other difference is that I will use only one external IP address, not a range of addresses.

```
$ iptables -t nat -A POSTROUTING -p TCP -s 192.168.1.0/24 -d
  192.168.2.0/24 -j SNATPBA --to-source 192.168.2.1 --block-size 2
```

The User PC must create connections to the Service PC to test the module. So on the Service PC, I make a server in brokering mode. That means the server allows multiple connections to the same listening instance, takes the input from one connection, and sends it as output to all other connections. This way, I can create and test the number of possible connections. To create a server, I am using the command `ncat`. The server is on the port 20000 that the User PC connections will connect to. The following command is how I create a server:

```
$ ncat --broker -l -p 20000
```

The User PC initiates connections to the server on the Service PC by the command `nc`. Every command execution with the same parameters creates a new connection that uses one of the ports from the block. So for this test, to use every port in one block, I must execute the `nc` command two times.

```
$ nc 192.168.2.2 20000
```

The following is a log message from the PBA module when the block is assigned to the group (192.168.1.2, 192.168.2.2) of connection. That implies that the first packet of the new connection has activated the target function.

```
May  4 22:12:17 kernel: ipt_SNATPBA: add:192.168.1.2:192.168.2.1:1024-1025
```

5.4.1 One connection

Next is the output of the `conntrack -E` command that shows how are changed records' states in the Connection tracking. I can show the output of the `conntrack` tool, thanks to the implemented solution (chapter 5.1) of the notifier. The output shows three changes in the state of the new connection. The connection protocol is TCP, which establishes the connection by a three-way handshake. For better clarity, I numbered changes in the record state.

```
1:[NEW] tcp   6 120 SYN_SENT src=192.168.1.2 dst=192.168.2.2 sport=44424\
dport=20000 [UNREPLIED] src=192.168.2.2 dst=192.168.2.1 sport=20000\
dport=1024

2:[UPDATE] tcp   6 60 SYN_RECV src=192.168.1.2 dst=192.168.2.2 sport=44424\
dport=20000 src=192.168.2.2 dst=192.168.2.1 sport=20000 dport=1024

3:[UPDATE] tcp   6 432000 ESTABLISHED src=192.168.1.2 dst=192.168.2.2\
sport=44424 dport=20000 src=192.168.2.2 dst=192.168.2.1 sport=20000\
dport=1024 [ASSURED]
```

The first state change is when an SYN packet activates the target function. It means that this packet generated the above log message about assigning the block. The target function calls the `nf_nat_setup_info`, which maps the original source IP address (`192.168.1.2`) to the IP address from the pool (`192.168.2.1`) and port (1024) from the assigned block. So the first record shows that the Connection tracking saw the SYN packet but did not see a reply from the server on the Service PC (`192.168.2.2:20000`). But the Connection tracking module created the new record and sent a notification with a `NEW` event, which the PBA module receives. When the PBA module gets this notification, it will find the assigned block to the group (`192.168.1.2, 192.168.2.2`) of connections and decrement the counter of available ports.

The second state change is when the Connection tracking sees the reply (SYN+ACK packet) from the server (`192.168.2.2:20000`) to the User PC. However, the User PC is located behind NAT, so the response is to the SNATPBA PC (`192.168.2.1:1024`), which maps this IP address to the actual recipient, the User PC (`192.168.1.2:44424`). The Connection tracking module saw traffic of this connection in both directions, but the connection is still not established. However, the PBA module has already received notification about this connection.

The third state change occurs when the User PC reply to the received SYN+ACK packet from the server with an ACK packet to confirm and establish the connection. The Connection tracking now sees the connection has been established and sends the `ASSURED` notification.

Sometime later, the connection is finished. The following is output from `conntrack -E` command. The first two rows show finishing the connection closed from the server. It shows the finishing packet from the server and the acknowledging packet from the User PC. The records also show that even if the connection is closed, a timer is still set to `60` seconds that must expire first.

```
1: [UPDATE] tcp   6 120 FIN_WAIT src=192.168.1.2 dst=192.168.2.2\
sport=44424 dport=20000 src=192.168.2.2 dst=192.168.2.1\
sport=20000 dport=1024 [ASSURED]

2: [UPDATE] tcp   6 60 CLOSE_WAIT src=192.168.1.2 dst=192.168.2.2\
sport=44424 dport=20000 src=192.168.2.2 dst=192.168.2.1\
sport=20000 dport=1024 [ASSURED]
```

```
3: [DESTROY] tcp   6 src=192.168.1.2 dst=192.168.2.2\
sport=44424 dport=20000 src=192.168.2.2 dst=192.168.2.1\
sport=20000 dport=1024 [ASSURED]
```

After the timer expires, the Connection tracking module deletes the record for this closed connection and releases the occupied port. The PBA module receives the notification with the `DELETE` event. It finds the appropriate group (`192.168.1.2`) of connections and increments the counter of available ports in this block. If this released port was the last occupied, it means that all ports are now available, and the block is taken away from this group of connections and put back into the list of available blocks. The cell in the hash tables is removed and deallocated, which shows the following log message:

```
May  4 22:15:08 kernel: ipt_SNATPBA: del:192.168.1.2:192.168.2.1:1024-1025
```

All packets that belong to the connections that use the assigned block do not need to be logged. If there is any report to the connection that needs to find a particular source IP address and this connection falls into the interval between `add` and `del` log messages, the connection was from the User PC.

5.4.2 Maximum number of connections

The previous section shows logs for one connection from the User PC to the Service PC. This test shows the behavior of the PBA module when a maximum number of connections are established from a single internal source IP address (`192.168.1.2`). The maximum number of connections is two because the SNATPBA rule added into the iptables has the `block-size` parameter set to `2`.

The following is the output of the `conntrack -E` command. It shows that two connections are established. Rows `1-3` were described in the previous section 5.4.1 and are for the first connection. Lines `4-6` show the same thing, except that they are for the second connection. Both connections were initiated by executing the `nc` command two times (`nc 192.168.2.2 20000`) on the User PC.

```
1:[NEW] tcp   6 120 SYN_SENT src=192.168.1.2 dst=192.168.2.2 sport=44424\
dport=20000 [UNREPLIED] src=192.168.2.2 dst=192.168.2.1 sport=20000\
dport=1024

2:[UPDATE] tcp   6 60 SYN_RECV src=192.168.1.2 dst=192.168.2.2 sport=44424\
dport=20000 src=192.168.2.2 dst=192.168.2.1 sport=20000 dport=1024

3:[UPDATE] tcp   6 432000 ESTABLISHED src=192.168.1.2 dst=192.168.2.2\
sport=44424 dport=20000 src=192.168.2.2 dst=192.168.2.1 sport=20000\
dport=1024 [ASSURED]


4:[NEW] tcp   6 120 SYN_SENT src=192.168.1.2 dst=192.168.2.2 sport=44426\
dport=20000 [UNREPLIED] src=192.168.2.2 dst=192.168.2.1 sport=20000\
dport=1025

5:[UPDATE] tcp   6 60 SYN_RECV src=192.168.1.2 dst=192.168.2.2 sport=44426\
dport=20000 src=192.168.2.2 dst=192.168.2.1 sport=20000 dport=1025
```

```
6:[UPDATE] tcp   6 432000 ESTABLISHED src=192.168.1.2 dst=192.168.2.2\
sport=44426 dport=20000 src=192.168.2.2 dst=192.168.2.1 sport=20000\
dport=1025 [ASSURED]
```

There are two active connections, which is a maximum number of simultaneous connections from the User PC. Now when I try to make a new connection, it fails because the PBA module has no available port for the next connection from the IP address 192.168.1.2. If no new connection is created, the Connection tracking module will not create a new record, and therefore the PBA module will not receive any notification from the module.

This chapter describes implementation details of issues that appeared in the design stage of this work and one that occurred during the implementation stage. The following chapter discusses the conclusion of the work.

Chapter 6

Conclusion

This thesis' goal was to research the algorithm for Address Translation using the Port Block Allocation and, after discussion with the work supervisor, implement this algorithm as the iptables' extension module in the Linux kernel.

The first step of this work was to research recommended materials. The problem with these materials is that they are relevant for older kernel versions (2.x and 3.x), but the implementation must be made in kernel version 5.4.53 and later 5.4.186. However, even though the materials were a little older, they still give a vast amount of information about how some things work in the kernel, for example, packet traversal, Netfilter hooks, NAT, and creating iptables extensions.

After research, I started discussing the design of the PBA algorithm with my supervisor. Here I had to look through the implementation of the SNAT module for how to use part of it in my implementation. And the `ipt_NETFLOW` project [4] on how to register the PBA module as a notifier in the Connection tracking module. The implementation of the kernel module (PBA) uses the core NAT functionality of the kernel, and the userspace program uses part of the parsing from the userspace program of the SNAT rule.

Next, I tested the implementation of the userspace program and the kernel-side (PBA) module on a virtual PC. The setup and network topology are described in chapter 4.1.

The biggest obstacle during the working on this thesis was finding the necessary information. Sometimes finding required information about the issue or the possible solution was tedious (and frustrating) work. For example, the issue with the multiple calling of the checkentry and destroy functions.

A possible future extensions might be to implement the hash table as dynamic. So there could be a size of the hash table as a new parameter in the SNATPBA rule. Thanks to this, a user adding the rule could adjust the size of the hash table to the size of the internal network (number of users).

Another extension may be to allow more than one block to be assigned to the internal source IP address. However, this leads to an unlimited maximum number of simultaneous connections from one IP address. This limit can be specified as an additional SNATPBA rule parameter.

Bibliography

[1] *How to register conntrack notifier* [online]. 2009 [cit. 2022-04-20]. Available at: https://netfilter-devel.vger.kernel.narkive.com/V9wwFLzh/how-to-register-conntrack-notifier.

[2] *IP Addressing: NAT Configuration Guide, Cisco IOS XE Release 3S - Carrier Grade Network Address Translation [Support]* [online]. Cisco, 2019 [cit. 2022-01-03]. Available at: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipaddr_nat/configuration/xe-3s/nat-xe-3s-book/iadnat-cgn.html.

[3] *Network Address Translation (NAT) FAQ* [online]. Cisco, 2021 [cit. 2022-01-03]. Available at: https://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/26704-nat-faq-00.html.

[4] aabc. *Ipt_NETFLOW* [online]. 2008-2021 [cit. 2022-04-15]. Available at: https://github.com/aabc/ipt-netflow.

[5] Ayuso, P. N. Netfilter's Connection Tracking System. *Login Usenix Mag.* 2006, vol. 31.

[6] Boye, M. *Netfilter Connection Tracking and NAT Implementation* [online]. 2012 [cit. 2022-01-05]. Available at: https://wiki.aalto.fi/download/attachments/69901948/netfilter-paper.pdf.

[7] Donley, C., Grundemann, C., Sarawat, V., Sundaresan, K. and Vautrin, O. *Deterministic Address Mapping to Reduce Logging in Carrier-Grade NAT Deployments* [Internet Requests for Comments]. RFC 7422. RFC Editor, December 2014.

[8] Jan Engelhardt, N. B. *Writing Netfilter modules.* 2012.

[9] Rosen, R. *Linux kernel networking : implementation and theory.* New York, NY: Apress, 2014. ISBN 9781430261964.